Theme Feature

Trends in Shared Memory Multiprocessing

Current application and technology trends are causing researchers and developers to revisit shared memory multiprocessing. The authors look at what is needed to maintain growth, particularly for commercial applications.

Per Stenström
Chalmers University of Technology

Erik Hagersten
Sun Microsystems Inc.

David J. Lilja
University of Minnesota, Minneapolis

Margaret Martonosi
Princeton University

Madan Venugopal
Silicon Graphics

Progress in shared memory multiprocessing research has led to its industrial recognition as a key technology for application domains such as decision support systems and multimedia processing. Although uniprocessor performance continues to increase roughly 50 percent each year, developers have recognized that these applications have computational needs that current uniprocessors cannot meet. Because such applications often have inherent parallelism, parallel processing is an effective way to meet their computational needs.

Like uniprocessors, current shared memory multiprocessors are often built from high-performance microprocessors, so there is a clear transition path from uniprocessor to multiprocessor program implementations. The challenge lies in making this transition as smooth as possible, both in performance and the programming required to achieve it.

The first step in meeting this challenge is to carefully examine the current use of shared memory multiprocessing and arrive at intelligent projections of future use based on application and technology trends. The second step is to begin filling gaps in programming models and architectures for shared memory multiprocessing. Finally—and possibly concurrently with the second step—researchers must look at ways to make the development of parallel software more feasible, including the development of compilers and tools. In this article, we look at the issues in these three steps. We also show how architectural issues are tightly interwoven with application trends.

APPLICATION AREAS

Research has focused heavily on scientific applications—sometimes to the detriment of other important domains. Commercial applications have been sadly neglected. Most users do not develop their own applications, primarily because they lack the appropriate tools and the shared memory programming paradigm is far from easy to manage.

Dominant domains

Figure 1 shows the dominant applications in the 1995 market and the projected market for the same applications in 2000, according to Dataquest. An "application" not shown, but nonetheless an important piece of software for a shared memory machine, is the operating system.

- *Databases.* Commercial databases are primarily *online transaction processing systems*, such as airline reservation systems or *decision support systems*, such as the selection of addresses for direct-mail campaigns. Database vendors typically support both shared memory and message-passing versions of their products. Significant concerns include reliability, availability, and serviceability.
- *File servers.* Commercial file servers service a large number of workstations or PCs. Concerns include the performance of the I/O subsystem (both on the network and disk sides) and reliability, availability, and serviceability.
- *Scientific and engineering.* These applications include those used both commercially and in research. They typically solve some scientific or engineering problem, such as the simulation of physical phenomena and the simulation of engineering systems. Even when run on a shared memory architecture, these applications can use a message-passing style of programming, typically with a message-passing library, such as MPI (Message-Passing Interface).
- *Media and e-mail servers.* These servers service a large number of accesses, or hits, to a widely available information pool. Once again, reliability, availability, and serviceability are important concerns.
- *Other commercial applications.* These applications do not fall into any of the above categories, such as customized applications developed by end users.

Figure 2 shows scientific and engineering and other commercial applications in the context of cost per machine for the 1995 market, according to Dataquest. As the figure shows, 79 percent of the 1995 server market (combined bars for <$10,000, <$50,000, and <$250,000) was covered by machines below $250,000. These machines, which typically have fewer than 16

0018-9162/97/$10.00 © 1997 IEEE

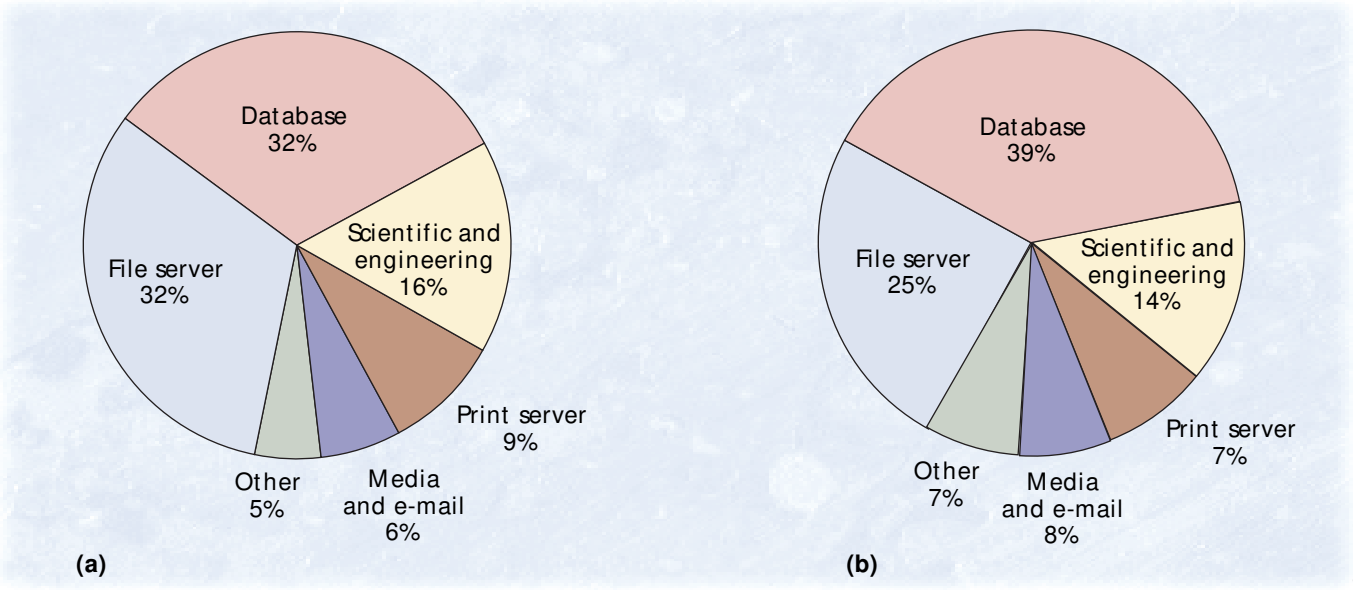

Figure 1. (a) 1995 and (b) 2000 market volume for servers. The market has grown and is projected to grow 15 percent annually.

processors, probably dominated the market for one of three reasons or some combination: they were affordable, many small machines are more available to users than one large machine, programs could not scale beyond 16 processors.

Although predicting the future computer market is difficult at best, two trends seem to be emerging: the continued domination of small and medium machines (Figure 2) and a slower annual growth for scientific and engineering applications relative to commercial ones (Figure 1).

Applications used in research

Shared memory multiprocessing research has recently begun focusing on engineering and scientific applications that run on hundreds of processors. While pushing scalability in this area is an interesting open problem, this area is not very representative of how servers are used. As Figure 2 shows, machines above $1 million represent only three percent of the server market revenue for engineering and scientific applications. Moreover, according to Dataquest, the annual growth from 1995 to 2000 is expected to be only 14 percent for this size server—a smaller increase than all the other server price categories.

This suggests increasing research activities that target other applications and smaller servers, which would address the remaining 97 percent of the market.

Compared with the scientific and engineering applications used in research, commercial applications typically have a larger code size, a much larger data set, fewer floating point operations, a different branch behavior, more users, higher use of the operating system, more context switches, and a higher I/O rate.[1] Because reliability is important in commercial servers, the research community must begin to address not only performance issues, but also reliability, availability, and serviceability.

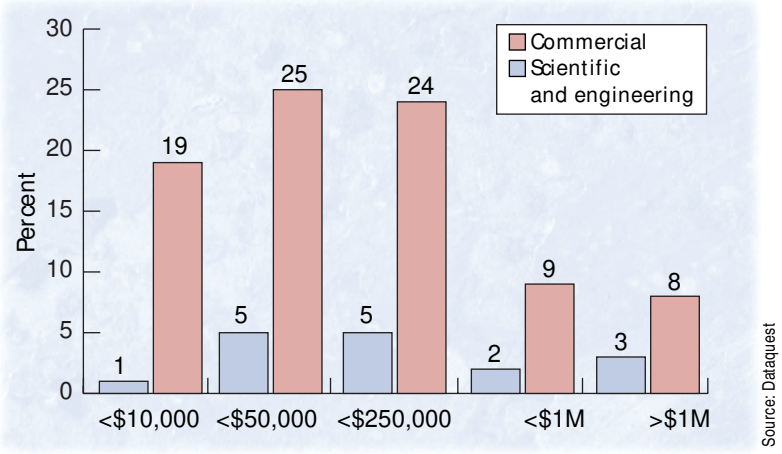

Figure 2. 1995 market volume by machine price. Only a small percentage is occupied by scientific and engineering applications, yet that is where most research efforts are concentrating.

Another obstacle is that few studies include the effects of the operating system. The NAS and Splash parallel benchmarks are just two examples of the one-track emphasis on scientific and engineering research applications. There is a pressing need for a more realistic evaluation framework for shared memory multiprocessing research.

MODELS AND ARCHITECTURES

Parallel programming models and emerging hardware technology are becoming increasingly disconnected. Many computer manufacturers are producing systems that are based on the shared memory model at the same time that many parallel applications are being based on a message-passing model. Actually, the community in general seems confused about how to define a smooth line from the parallel algorithm, the programming model by which it is implemented, and the system architecture that realizes it. Adding to the problem is the lack of standards to port applications among shared memory systems.

There is a pressing need for a more realistic evaluation framework for shared memory multiprocessing research.

Defining characteristics

The *programming model* is the interface between the programmer and the machine. The *architecture* is the underlying structure of the machine itself. The choice of a specific programming model determines how a programmer views the machine, independent of the machine's actual structure. For example, in distributed shared memory architectures such as the Stanford DASH and the Silicon Graphics Origin, the memory is physically distributed among processors. The hardware passes messages to automatically ensure cache coherence while providing the illusion of a single shared global address space to the programmer.

Consequently, every processor uses the same name to refer to the same global location in memory, which makes sharing data among processors relatively simple. In a message-passing programming model, on the other hand, each processor has its own private address space. To have processors share data, the programmer must include explicit `send` and `receive` statements in the source code.

It is thus possible to implement a message-passing model on top of a shared memory machine and vice versa.

Model-architecture trade-offs

Performance is generally higher when the programming model matches the underlying architecture, but architects can write an application for a particular architecture, making it insensitive to variations in memory access delays. Although writing applications for specific architectures is time-consuming and largely defeats the ease of use that comes with the shared memory model, for production-level code (written once and executed many times), designers may want to extract as much performance as possible by tuning the code in this manner. Performance models that estimate the effects of variable access costs would make it easier to write such applications, as we describe later.

Scalability issues

The shared memory model enables incremental parallelization at varying levels of granularity. Because this model features a low access latency, programmers can efficiently parallelize loops that contain very little work. In fact, this feature is one reason designers have been able to port large, legacy serial applications to shared memory systems. Although much less than ideal scalability often results, the performance enhancements and resulting speedups are desirable to many users, as we describe later.

The message-passing model's high scalability is often attributed to the new message-passing version of the code. However, parallelizing the code usually forces extensive algorithmic changes to achieve that scalability—changes that could also be implemented with a shared memory model. In fact, a message-passing model on a shared memory machine can offer both good performance and portability.

Moreover, we have found that scalability is possible with the shared memory model, although most people seem to believe otherwise. For example, consider the results in Table 1 of implementing Aerosoft's GASP, a computational fluid dynamic application, on the Silicon Graphics Origin 2000 system. The table shows the computation time and speedup in 10 time steps. The problem involved a $321 \times 321 \times 101$ grid (10.4 million grid points) with 512 zones, requiring 2.8 Gbytes of memory and about 1 Gbyte of disk space. The Origin 2000 system it ran on has 128 processors, 4-Mbyte caches, and 16 Gbytes of memory. We removed the startup overhead from the first step so that the I/O cost usually amortized over hundreds of steps did not affect the speedup. All CPU times are from within GASP.

As the table shows, shared memory architectures can scale well for a scientific application. In fact, the parallel implementation used for the Origin 2000 was originally developed for a bus-based multiprocessor.[2]

Some believe that massively parallel message-passing systems will be more scalable than shared memory architectures for database applications. However, according to results of the decision-support benchmark TPC-D (http://www.tpc.org), shared memory implementations can outperform massively parallel message-passing systems, even at 128 processors. In fact, the current performance leader, the Sun E10000 (64 processors at 300 Gbytes) is a shared memory architecture. Compared with message-passing systems, the E10000's sharing of memory and I/O makes database partitioning less critical to performance,[3] and porting a database to other shared memory architectures is much easier.

Misconceptions about shared memory scalability will continue until the community begins separating algorithm, implementation, and architecture issues when reporting parallel results. Performance studies comparing the two models on similar algorithms[4] are required to clarify the distinction between the algorithm and the programming model. These studies, combined with a widely adopted standard for shared memory parallel programming, may naturally lead to greater use of the shared memory model in the software development community.

PARALLEL SOFTWARE DEVELOPMENT

To make it easier to develop parallel software, research must extend work in automatic (compiler) parallelization techniques and in developing tools to aid performance tuning.

Parallelization techniques

Although explicit manual parallelization remains the most common means for exploiting parallelism, automatic compiler parallelization is moving closer to reality. Several significant challenges remain, however.

Broadening application areas. Compilers can parallelize many interesting types of scientific code.[5] They are also identifying parallelism for an increasing number of applications that involve loop-oriented code.[5] However, broadening applications to include other commercial code is proving difficult. Such code is often several magnitudes larger than loop-oriented code and more complex. Further, large classes of applications such as databases, telecommunications, geographic information systems, graphics, and stock and option trade systems run sequentially on shared memory parallel machines. Some of these do not have the structure characteristic of scientific and engineering applications, so parallelizing them may be much more challenging. However, the payoff may also be greater.

One problem is how to identify parallelism that spans procedure boundaries. This typically involves including memory accesses through pointers where data dependencies are difficult to analyze. How to uncover data dependencies when pointers are involved remains one of the most significant obstacles to fully identifying program parallelism. Some strides have been made in analyzing C programs with pointers, but loops with very dynamic access patterns remain unparallelized. In such cases, interactive parallelization techniques may offer promise. In these techniques, compilers query users for higher level program information that may aid them in identifying parallelism or deducing when perceived dependencies need not impede parallelization.

Some experimental and commercial compilers have tackled this parallelism problem by applying compiler analysis passes interprocedurally. With techniques like making arrays private and conducting an interprocedural analysis, researchers can identify much coarser grained parallelism in the code, which may reduce both the code's synchronization and memory costs.

Another parallelization method used by applications experts is *algorithm recognition and substitution*. Many linear algebra algorithms, for example, have interesting parallel variants. Compilers often do some of the simpler cases of recognize and substitute, such as optimizing sum reductions. The compiler community should consider identifying the next few examples of recognize and substitute—and not just for scientific computing.

Managing parallelism. In many cases, compilers can successfully identify sufficient parallelism in major program loops. However, although some of these programs exhibit excellent speedups, others have properties, like memory behavior, that limit speedup. Parallelizing compilers often aggressively parallelize as many loops as possible, which can cause fine-grained loops to be parallelized in ways that lead to extreme true or false sharing. In some cases, these parallelized loops with sharing problems perform much worse than if the loop had executed sequentially. Better memory analysis and combinations of communication and computation analysis[6] could limit the number of poor parallelization choices. In some cases, the compiler can adequately analyze memory behavior, but it often has insufficient control of the memory hierarchy. Instruction-set architectures that let compilers have better control of caches could improve memory behavior.

Table 1. Speedups for the parallel version of the GASP computational fluid dynamic application running on Silicon Graphics Origin 2000.

Number of processors	CPU time (hr:min:sec)	Parallel speedup
1	03:28:28	1.00
2	01:44:49	1.99
16	00:14:14	14.65
32	00:07:24	28.17
64	00:03:52	53.91
128	00:02:20	89.34

Thus, although parallelizing compilers have become quite good at identifying parallelism, they are less adept at predetermining the degree of it. As a result, a compiler-parallelized application may execute its computation on more processors than it can effectively use, not only wasting processors that could be applied to other useful work but sometimes even slowing down the computation itself. This waste of computational resources becomes more acute with the number of processors, particularly for parallel computers used as multiprogrammed computational servers. Preliminary work shows the promise of adapting processing as the program runs. The idea is to limit the processes allocated according to system load and the application's available parallelism. Such approaches let the system use as many processors as possible without overloading.

Other issues. Even with state-of-the-art compiler technology, automatic parallelization of large applications is not fully practical. Thus, compilers should provide feedback to the user to assist in manual parallelization. For example, the compiler gathers much useful information about memory behavior and dependencies. Even if it does not succeed in automatic parallelization, the information gathered during analysis may help the programmer manually parallelize or optimize the application. Formal mechanisms that convey pertinent information from an analysis would also be useful.

Another issue, which affects both portability and ease of use, is the need to standardize compiler directives. Despite differences in shared memory multiprocessors, no standard for shared memory parallelization directives or compiler flags exists. Distributed memory machines have had success with standards such as the Message-Passing Interface (MPI). Such work must continue.

Despite the great importance of parallel performance analysis to good program speedups, parallel tools are still relatively immature.

Performance tools

Because manual parallelization is still the method most often used, programmers must have performance tools to tune their code. Unfortunately, despite the great importance of parallel performance analysis to good program speedups, parallel tools are still relatively immature.

Many tools provide basic high-level information on the program's computational performance and coarse-grained communication behavior. In shared memory multiprocessors, it is relatively easy to monitor and display statistics about interprocessor synchronization, for example. Fine-grained user-level timers have begun to appear on commercial multiprocessors, which also provide accurate timing statistics.

Memory behavior, however, is still quite difficult to monitor, primarily because software has traditionally been presented with the abstraction of a uniform high-speed memory. Unlike branch instructions—which alter control flow depending on the path taken—loads and stores offer no direct mechanism to determine if a particular reference was a hit or a miss. Shared memory parallel programs communicate via shared data, so statistics on memory accesses are crucial because they reveal much about the program's interprocessor communication behavior.

Another issue related to monitoring memory behavior is the need for models that can accurately predict performance levels. Such models would estimate the effects of variable access costs, which would help users see the performance they could achieve when parallelizing applications. For system designers, these models could suggest the required balance between processor performance and memory latencies and bandwidths. The lack of such models will become an increasing obstacle as processor technology continues to change and memory hierarchies become more complex.

Some work is already being done to observe memory behavior. Recently, many commercial CPUs have provided on-chip memory-event counters that the user can access. However, although these counters offer good aggregate views of memory behavior, it is not easy to determine if a particular reference hits or misses in the cache. Instead, the miss counter value must be read just before the reference is executed and just after, which sequentializes the pipeline. A more general approach might be to combine lightweight cache miss traps with performance counters in handler software.

At the software level, tools have also been hobbled by the lack of standardization in programming interfaces. With so many parallel languages and programming environments, tool interoperability is nearly impossible. As the community reaches consensus on a handful of common parallel programming models, it will become feasible to develop well-tested, widely used code for these models. Furthermore, we believe that integrating the tool infrastructure with a full programming and compilation environment offers the best hope that performance tools will become part of the standard program development cycle.

ARCHITECTURAL DESIGN DRIVERS

Given the current application and technology trends, we believe many proposed architectural design solutions may be inadequate because they fail to address relevant applications, the increasing speed gap between processor and memory technology, and the effects of processor technology. Moreover, many of the techniques we describe here apply primarily to scientific and engineering workloads. For other commercial workloads, the community still lacks experience because effective evaluation methodologies are not yet available. Current simulation-based approaches have severe shortcomings to realistically model the often complex interaction between software, processor, memory, and I/O subsystems in a commercial context.

Application push

Small-scale multiprocessors offer a *uniform memory access* model and are often called symmetric multiprocessors. Because of the efforts invested in designing them, symmetric multiprocessor and single processor computational nodes have become natural building blocks for larger configurations. The HP/Convex Exemplar, Sequent NUMA-Q, and Origin 2000 are examples. Although offering attractive cost, these systems have a *nonuniform memory access* model to the software. In this model, processors observe different access latencies to local and remote memories, which makes it difficult to tune the system for performance. Research has thus concentrated on techniques that aim to eliminate the NUMA model's negative effect on performance. These techniques fall (roughly) into two categories: latency reduction and latency tolerance.

Latency reduction. Efficient cache management is critical to reduce memory access latency, and significant research in cache coherence maintenance over the past several years[7] has responded to this need. Although fairly efficient solutions exist for scientific applications, it is unclear how commercial workloads interact with the proposed techniques.

Increasing cache capacity is a natural weapon to reduce memory latency. For some scientific and engineering codes, fairly small caches may suffice because the most performance-sensitive working sets in these applications typically grow slowly with problem size.[8] However, it is currently unclear how working sets for data-intensive applications, such as online transaction processing and decision support systems, grow with problem size. Designers can also reduce memory access latencies in NUMA systems by carefully laying out data in page-size chunks across the distributed memory mod-

ules. While such NUMA policies have proved effective for important scientific applications, it is not clear how useful they are for other commercial applications.

Latency tolerance. When latency cannot be reduced, techniques tolerate it by overlapping computation with communication. Recent research has distilled some promising latency toleration techniques but at the cost of higher memory system bandwidth. Also, each approach has weaknesses.

Relaxing memory consistency undoubtedly enables many of the standard optimization tricks for uniprocessor systems, but it does not hide all latency because ultimately the system must respect interthread data dependencies. As the latencies increase, these dependencies further erode performance. *Data prefetching* hides latency by letting a node signal that data is needed in advance. However, most existing prefetch approaches are useful only for the regular access patterns typically found in scientific applications. Finally, *multithreading* switches between independent threads when a long-latency operation is encountered. This approach requires hardware support for fast thread switches and more application parallelism to be effective. Two trends can make it more useful against memory access latencies. First, as the speed gap between processors and memories increases, thread-switch overhead will be less of a concern. Second, the difficulty of extracting more instruction-level parallelism at an affordable complexity level may lead to an increased interest in multithreaded architectures.

Commercial applications are often very difficult to parallelize using traditional approaches. They typically contain many loops with early exit conditions, such as `do-while` loops. Parallelizing these loops requires architectural and compiler support for thread-level speculative execution. Small granularity loops with low trip counts, also common in these applications, typically require fast communication and synchronization to be efficiently parallelized.

Parallelizing programs with extensive memory aliasing, which often occurs when using pointers, requires hardware support for data speculation or runtime dependence checking between threads. No system currently provides these features at a level that can support difficult-to-parallelize applications.

Technology push

Designers of high-performance microprocessors have had the luxury of borrowing architectural ideas from mainframe and supercomputer systems. As a result, almost all microprocessors available today use techniques such as out-of-order execution, register renaming, branch prediction, and multiple instruction issue per cycle to provide high uniprocessor performance.

Unfortunately, this grab bag of performance-enhancement tricks is now essentially empty. Within the next decade, it will be possible to integrate a billion transistors on a single component. Performance enhancement techniques will become critical as the community looks for the best way to exploit these huge numbers of transistors to deliver high performance. Should system designers build a single processor, should they follow the current trend and construct an entire shared memory multiprocessor system on a chip, or are more innovative solutions possible?

While there are many interesting suggestions for how to use these transistors,[9] processor technology and application changes will likely drive computer architects to develop radically new approaches for exploiting parallelism. Examples include the concurrent multithreaded architecture in the Multiscalar machine,[10] the superthreaded architecture,[11] and the single-program speculative multithreaded architecture.[12] These approaches provide multiple program counters to support thread-level parallelism, but each individual thread can simultaneously exploit instruction-level parallelism. Additionally, the use of control speculation allows concurrent multithreaded architectures to parallelize loops with early exit conditions. Their hardware support of either data speculation or runtime data dependency checking can help solve the problem of limited parallelism due to memory aliasing.

It would thus be feasible to use these or other new types of processor architectures as the individual computational nodes of a shared memory multiprocessor system. Furthermore, trends in distributed systems are to interconnect multiple heterogeneous systems using standardized data communication networks, such as Asynchronous Transfer Mode, Fibre Channel, and High-Performance Parallel Interface. These networks provide sufficiently high bandwidth and low latency. This allows the collection of systems to be viewed as a single, large metasystem[13] on which very large applications can be executed.

However, although these systems provide multiple levels of parallelism, their complex hierarchies severely complicate the tasks of the compiler and application programmer. Increasing the power of individual computational nodes will put additional stress on the memory system and the interconnection network, exacerbating problems from memory delays. Nevertheless, the tremendous opportunities these complex systems offer for improving performance will force system architects and software designers to develop new strategies to exploit them.

Processor technology and application changes will likely drive computer architects to develop radically new approaches for exploiting parallelism.

In the past several years, research in shared memory multiprocessing has clearly paid off commercially. Several vendors are now selling cost-effective commercial multiprocessors that improve the performance of many important applications. However, additional hurdles must be overcome for multiprocessing to continue to offer large performance improvements on a wider variety of applications.

A major challenge is to address the particular concerns of commercial code, but perhaps the greatest challenge is to develop new techniques in the face of a moving hardware target. The community must somehow improve the software and keep pace with constant increases in integration level, on-chip parallelism, and memory hierarchy complexity. ❖

Acknowledgments

We thank Yale Patt, who initiated the set of task forces that allowed us to develop our thoughts in a creative environment in Hawaii. We also thank the anonymous reviewers and those at Silicon Graphics who contributed their thoughts: Dan Lenoski, Brond Larson, Woody Lichtenstein, John McCalpin, and Jeff McDonald.

References

1. A. Grizzaffi Maynard et al., "Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workload," in *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 145-155.
2. M. Venugopal, D. Slack, and R. Walters, "A Commercial CFD Application on a Shared Memory Multiprocessor," in *High Performance Computing*, S. Sahni, V. Prasanna, and V. Bhatkar, eds., McGraw-Hill, New York, 1995, pp. 305-310.
3. B. Carlile, "Seeking the Balance: Large SMP Warehouses," *Database Programming Design*, Aug. 1996, pp. 44-48.
4. S. VanderWiel, D. Nathanson, and D. Lilja, "Complexity and Performance in Parallel Programming Languages," in *Proc. Int'l Workshop High-Level Parallel Programming Models and Supportive Environments*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 3-12.
5. R. Wilson et al., "An Infrastructure for Research on Parallelizing and Optimizing Compilers," *SIGPlan Notices*, Dec. 1994, pp. 31-37.
6. J. Anderson and M. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," in *Proc. SIGPlan Conf. Programming Language Design and Implementation*, ACM Press, New York, 1993, pp. 112-125.
7. P. Stenström et al., "Boosting the Performance of Shared Memory Multiprocessors," *Computer*, July 1997, pp. 63-70.
8. E. Rothberg, J. Pal Singh, and A. Gupta, "Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors," in *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 14-25.
9. *Computer,* special issue on billion-transistor architectures, D. Burger and J. Goodman, eds., Sept. 1997, pp. 46-93.
10. G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar Processors," in *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 414-425.
11. J.-Y. Tsai and P.-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Runtime Data Dependence Checking and Control Speculation," in *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 35-46.
12. P. Dubey et al., "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading, in *Proc. IFIP WG 10.3 Conf. Parallel Architectures and Compilation Techniques*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 109-121.
13. L. Smarr and C. Catlett, "Metacomputing," *Comm. ACM*, June 1992, pp. 45-52.

Per Stenström *is a professor of computer engineering at Chalmers University of Technology. He is on the editorial board of* Journal of Parallel and Distributed Computing. *Stenström received an MS in electrical engineering and a PhD in computer engineering from Lund University. He is a member of the IEEE Computer Society, IEEE, ACM, and SIGArch.*

Erik Hagersten *is the chief architect for Sun Microsystems' High-End Server Engineering group. Hagersten received an MS in electrical engineering and a PhD in computer science, both from the Royal Institute of Technology in Stockholm.*

David J. Lilja *is an associate professor and the director of graduate studies in computer engineering in the Dept. of Electrical and Computer Engineering at the University of Minnesota, Minneapolis. Lilja received a PhD and an MS, both in electrical engineering, from the University of Illinois at Urbana-Champaign, and a BS in computer engineering from Iowa State University. He is a senior member of the IEEE, a member of the ACM, and a distinguished visitor of the IEEE Computer Society.*

Margaret Martonosi *is an assistant professor of electrical engineering at Princeton University. Martonosi received a BS from Cornell University and an MS and a PhD from Stanford University, all in electrical engineering.*

Madan Venugopal *is a member of the technical staff at Silicon Graphics Computer Systems. Venugopal received a BTech in naval architecture and shipbuilding from the University of Cochin, an MASc in mechanical engineering from the University of British Columbia, and an SM and a PhD in ocean engineering from the Massachusetts Institute of Technology. He is a member of the IEEE Computer Society.*

Contact Stenström at the Dept. of Computer Engineering, Chalmers University of Technology, S-412 96 Göteborg, Sweden; pers@ce.chalmers.se; http://www.ce.chalmers.se/~pers.