Mark D. Hill
University of Wisconsin, Madison

Multiprocessors Should Support Simple Memory-Consistency Models

The author argues that multiprocessors should support sequential consistency because—with speculative execution—relaxed models do not provide sufficient additional performance to justify exposing their complexity to the authors of low-level software.

In the future, many computers will contain multiple processors, in part because the marginal cost of adding a few additional processors is so low that only minimal performance gain is needed to make the additional processors cost-effective.[1] Intel, for example, now makes cards containing four Pentium Pro processors that can easily be incorporated into a system. Multiple-processor cards like Intel's will help multiprocessing spread from servers to the desktop.

But how will these multiprocessors be programmed? The evolution of the programming model already under way is likely to continue.

- Multiprocessors will continue to be used for *multitask programming*, which lets developers optimize conventional, single-threaded programs for multiprocessing.
- Critical parts of processor-intensive applications will continue to be parallelized for multiprocessing—to use multiple threads that share data through shared memory.
- Someday, we may be able to build compilers that can effectively parallelize programs or we may be able to provide tools or abstractions that allow developers to program in parallel.

But what hardware do we need to support shared memory threads? The hardware should provide a well-defined *interface* for shared memory and it should provide a high-performance implementation of the interface.

UNIPROCESSOR MEMORY

Defining a shared memory multiprocessor's interface to memory is easier if we first consider how memory is defined in a uniprocessor. A uniprocessor executes instructions and memory operations in a dynamic execution order called *program order*. Each read must return the value of the last write to the same address. If the uniprocessor is capable of multitasking, two options exist. If a program executes as a single thread without sharing memory, then the programmer's memory interface is the same as for a uniprocessor without multitasking.

The situation is more complex, on the other hand, if a program has multiple threads sharing memory (or the

0018-9162/98/$10.00 © 1998 IEEE

program shares memory with other running programs or is the OS itself). In this case, the last write to an address could be made by itself (the same thread) or by another thread (that was context-switched onto the processor since this thread's last write to the address).

Programmers can model a multitasking uniprocessor as a merging of the program orders of each executing thread into a single, totally ordered processor execution. Most programmers, for example, would expect the code fragment in Figure 1 to set `data_copy` to the value of `new`. Here, accesses to `flag` are an example of *synchronization*, because their purpose is to coordinate accesses to `data`.

```
Thread or processor P1          Thread or processor P2
data = new;
flag = SET;
                                while (flag != SET) {}
                                data_copy = data;
```

Figure 1. Is `data_copy` always set to `new`? Most programmers would expect the code fragment to set `data_copy` to the value of `new`. Here accesses to `flag` are an example of synchronization, because their purpose is to coordinate accesses to `data`.

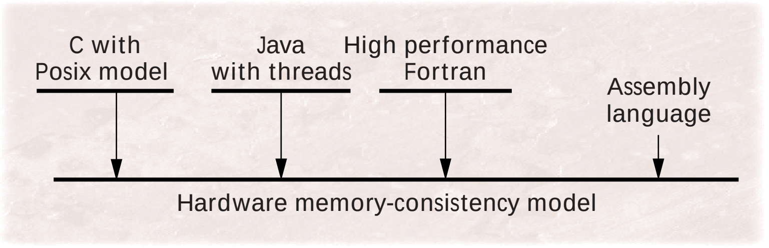

Figure 2. A shared memory multiprocessor has one or more high-level language memory consistency models (depicted by the upper lines) and one hardware memory consistency model (depicted by the lower line). Middleware must preserve high-level memory semantics when translating programs to the hardware model. In contrast, assembly language programs are written directly to the hardware model.

MIDDLEWARE

Most computers today, however, are programmed in high-level languages such as C, C++, and Java. This practice creates two memory interface levels. At the higher level, each language defines memory for its programmers. At the lower level, hardware defines memory for low-level software, called middleware. Middleware includes compilers, libraries, device drivers, OSs, and some key components of important applications (such as databases, messaging systems, and distributed computing frameworks).

Software written at a high level requires that compilers translate high-level memory operations into low-level ones in a manner that preserves memory semantics. In Figure 1, for example, a compiler should not reorder P1's stores to `data` and `flag`. Posix threads, for example, recommend that synchronization be encapsulated in library calls, such as `pthread_mutex_lock()`.

MEMORY CONSISTENCY MODELS

The interface for memory in a shared memory multiprocessor is called a *memory consistency model*. Similar to a uniprocessor, high-level programming induces the two levels of memory consistency models, depicted in Figure 2: high-level models for each high-level language and one low-level model for hardware. This article is primarily concerned with hardware memory consistency models.

A multiprocessor can use the same relatively simple memory interface as a multitasking uniprocessor. Leslie Lamport formalized this memory consistency model as sequential consistency (SC).[2]

Perhaps surprisingly, the hardware memory consistency models of most commercial multiprocessors are not SC because alternative relaxed models are believed to facilitate high-performance implementations better. Some relaxed models just relax the ordering from writes to reads (processor consistency, IBM 370, Intel Pentium Pro, and Sun's Total Store Order), while others aggressively relax the order among all normal reads and writes (weak ordering, release consistency, DEC Alpha, IBM PowerPC, and Sun's Relaxed Memory Order).[3-5] (Also see William Collier's tools for distinguishing memory models at http://www.infomall.org/diagnostics/archtest.html.)

Many academics, including myself, have long advocated relaxed models over SC. But the advent of *speculative execution* has changed my mind about relaxed models and SC. Multiprocessor hardware should implement SC or, in some cases, models that just relax the ordering from writes to reads. I now see aggressively relaxed models as less attractive, because the future performance gap between the aggressively relaxed models and SC will be in the range of 20 percent or less.

I present my case by first reviewing SC and relaxed models and then examining the performance gap and complexity of reasoning with relaxed models.

SEQUENTIAL CONSISTENCY

Lamport defined a multiprocessor to be sequentially consistent if

- the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and
- the operations of each individual processor were to appear in this sequence in the order specified by its program.

The principal benefit to selecting SC as the interface to shared memory hardware is that SC is what people expect. If you ask moderately sophisticated software professionals what shared memory does, they will likely define SC (albeit less precisely and less concisely than Lamport). Since good interfaces should not astonish their users, SC should be preferred.

```
Processor P1        Processor P2
x = new;            y = new;
y_copy = y;         x_copy = x;
```

Figure 3. Processor consistency (PC), IBM 370, Pentium Pro, and Total Store Order allow both `x_copy` and `y_copy` to get old values, which violates sequential consistency.

A literal interpretation of SC's definition, however, might lead one to believe that implementing it requires one memory module and precludes per-processor caches, but this is not the case. Both SC and the relaxed models (described below) allow many optimizations important for high-performance implementations.[6]

SC and the relaxed models allow coherent caching, nonbinding prefetching, and multithreading. SC, however, permits the following optimizations and keeps the software interface simple. In particular, SC enables middleware authors to use the same target as a multitasking uniprocessor. Thus, it seems hardware architects should choose SC.

Coherent caching

All models permit coherent caching. Caches may be private to processors or shared among some processors. They may be in one level or in multilevel hierarchies. A standard implementation of coherence processes operations of each processor in program order and does not perform a write until after it invalidates all other cached copies of the block.

Nonbinding prefetching

All models can use nonbinding prefetching, which moves a block into a cache (in anticipation of use) but keeps the block under the jurisdiction of the coherence protocol. Nonbinding prefetches affect memory performance but not memory semantics, because the prefetched block can be invalidated if another processor wishes to write to the block. Nonbinding prefetches can be initiated to overlap cache miss latency (with computation or other misses) by either hardware or software, through special prefetch instructions.

Multithreading

Finally, all models can support multithreading, where a processor contains several "hot" threads (or processes) that it can run in an interleaved fashion. A multiprocessor with *n* *k*-way multithreaded processors behaves like a multiprocessor with *n* *k* conventional processors. The implementation of multithreaded hardware, however, can switch threads on long-latency events (to hide cache misses, for example), can switch every cycle, or can simultaneously execute multiple threads each cycle.

RELAXED MODELS

Despite SC's advantages, most commercial hardware architectures have selected alternatives to SC called *relaxed* (or *weak*) memory consistency models. Relaxed models were initially used to facilitate additional implementation optimizations, whose use is precluded by SC without the complexity of speculative execution.

SC, for example, makes it hard to use write buffers, because write buffers cause operations to be presented to the cache-coherence protocol out of program order. Straightforward processors are also precluded from overlapping multiple reads and writes in the memory system. This restriction is crippling in systems without caches, where all operations go to memory. In systems with cache coherence—which are the norm today—this restriction impacts activity whenever operations miss or bypass the cache. (Cache bypassing occurs on uncacheable operations to I/O space, some block transfer operations, and writes to some coalescing write buffers.)

Relaxing write-to-read order

Defining relaxed models is complex. The class that Adve and Gharachorloo called *relaxing write-to-read program order*[3]—sometimes called the *processor-consistent models*—more or less exposes first-in first-out (FIFO) write buffers to low-level software. This means that a processor's writes may not immediately affect other processors, but that when they do the writes are seen in program order.

If a processor does `write x`, `write flag`, and `read y`, for example, it can be sure that `x` is updated before `flag`, but it cannot know if either is done when it reads `y`. All common processors also make sure that they see their own writes immediately (through write buffer bypassing, for example). This ensures that these models have no effect on uniprocessor behavior.

Furthermore, the difference from SC makes no difference to most shared memory programs, because most programs produce shared data by writing the data and then writing a flag or counter (as in Figure 1). Programs only observe differences from SC in convoluted examples. In the code fragment illustrated in Figure 3, for example, these models allow both `x_copy` and `y_copy` to obtain old values, while SC requires that at least one gets the value of `new`.

Specific models in this class include Wisconsin/Stanford processor consistency (PC), IBM 370, Intel Pentium Pro, and Sun's Total Store Order. These models differ in subtle ways, such as whether a processor reading its own write ensures that other processors also see it. All of the commercial models also guarantee *causality*. Causality requires that all other processors see the effect of a processor's operation when any other processor sees it. In Figure 4, causality ensures that `data_copy` gets the value `new`. Without causality, processor consistency can fail to look like SC in many cases involving three or more processors.

These hardware memory consistency models make it easier for hardware implementors to use hardware optimizations found in uniprocessors. In particular, processor writes can be buffered in a FIFO write buffer in front of the cache and coherence protocol. Values of these buffered writes can often be bypassed to subsequent reads—by that processor to the same address—even before coherence permission has been obtained. This optimization is especially important for architectures with few general-purpose registers, such as Intel's IA 32.

Furthermore, these models have negligible impact on middleware authors. If these authors assume SC, they will rarely be surprised by the model. These models look exactly like SC for the common idioms of data sharing (as in Figure 1 and Figure 4, for example, but not in Figure 3).

Relaxing all orders

The class *relaxing all orders* seeks to allow all the hardware implementation options of uniprocessors. Members of this class may completely reorder reads and writes and include USC/Rice weak ordering (WO), Stanford release consistency (RC), DEC Alpha, IBM PowerPC, and Sun's Relaxed Memory Order). Regardless of reordering, these models make sure that a processor sees its own reads and writes in program order.

The models differ in subtle ways and in how programmers restore order between memory operations to make examples like Figure 1 behave as expected. WO and RC ask programmers to distinguish certain reads and writes as synchronization, so the hardware can handle these more carefully.

The commercial models add special operations—variously called *fences*, *barriers*, *membars*, and *syncs*—to tell the system when order is required. Figure 5 illustrates how the example in Figure 1 could be augmented for Sun RMO. The membar `#StoreStore` ensures `data` is written before `flag`, while membar `#LoadLoad` ensures `flag` is read before `data`. Implementations of the models in this class can exploit many optimizations, because they need only implement order between operations when software asks for it and can be aggressively out of order the rest of the time. Processors can complete reads and writes to cache, for example, even while previous reads and writes (in program order) have not obtained coherence permission.

Furthermore, a hardware model from the *relaxing all orders* class does not appear to be too great a challenge for compiler writers. For sequential high-level languages with threads, programmers often use synchronization libraries or declare critical variables volatile. In these cases, the compiler or library writer can add appropriate fences. For sequential languages with parallelizing compilers, the compiler inserts the synchronization so it can know where the fences need to be.

```
Processor P1        Processor P2          Processor P3
data = new;
flag1 = SET;
                    while
                    (flag1 != SET){}
                    flag2 = SET;
                                          while
                                          (flag2 != SET){}
                                          data_copy = data;
```

Figure 4. Causality is needed to ensure `data_copy` is set to `new`.

```
Processor P1                       Processor P2
data = new;
membar #StoreStore
flag = SET;
                                   while (flag != SET) {}
                                   membar #LoadLoad
                                   data_copy = data;
```

Figure 5. Memory barriers (membars) to ensure `data_copy` are always set to `new` under Sun RMO.

THE PERFORMANCE GAP

Basically, relaxed models offer more hardware implementation options than SC and appear to use information that programmers know anyway. So it appears hardware should use relaxed models instead of SC.

To the contrary, the performance gained by using relaxed models does not justify their complexity. The principal argument for relaxed models is that using them can yield higher performance than with SC. So what is the performance gap between relaxed models and SC? The answer depends on many processor implementation parameters.

Two observations by Kourosh Gharachorloo, Anoop Gupta, and John Hennessy[7] have reduced the performance gap relative to initial expectations.

Overlapped coherence operations

SC hardware does not need to serialize the operations that obtain coherence permission (such as nonbinding prefetches and cache misses). Instead, SC can overlap these operations just like relaxed implementations.

SC implementations, however, should perform the actual reads and writes to and from the cache in program order. Thus, SC implementations can handle four cache misses on the sequence `read A, write B, read C, write D` in time modestly longer than handling one miss and three hits. Using a nonblocking cache, an SC implementation could pipeline `get shared block A, get exclusive block B, get shared block C, get exclusive block D` and then rapidly perform `read A, write B, read C, write D` as a series of cache hits.

Speculative execution

The advent of speculative execution allows both relaxed and SC implementations to be more aggres-

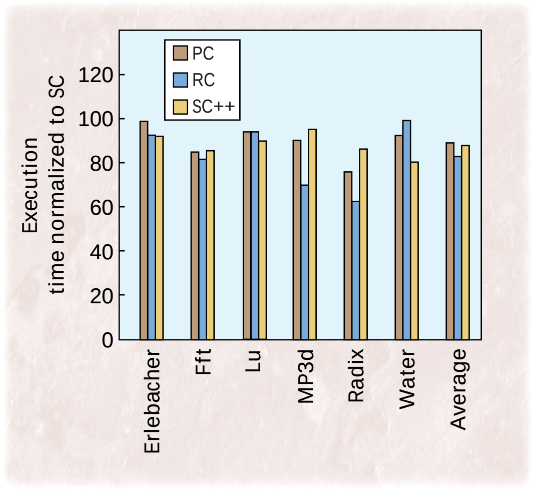

Figure 6. Execution time results using various memory consistency models for six well-known scientific application tests used to simulate workload on a MIPS R10000-like processor.

sive. With speculative execution, a processor performs instructions eagerly. Sometimes instructions must be undone when speculations on previous instructions prove incorrect (as they are in mispredicted branches, for example).

A processor *commits* (or *retires*) an instruction when it is sure that an instruction will not need to be undone. Doing so usually frees up implementation resources. Instructions commit when

- all previous instruction has already committed, and
- the instruction's operation itself commits.

A load or store operation commits when it is certain to read or write an appropriate memory value.

Speculative execution allows both relaxed and SC implementations to perform load and store operations speculatively and out of order. In some cases, however, relaxed implementations can commit memory operations sooner than SC implementations.

Consider, for example, a program that wishes to `read A` (which misses) and `read B` (which hits). Both relaxed and SC processors can perform the `read B` before even beginning the `read A`. Furthermore, relaxed processors can sometimes commit `read B` without waiting for `read A` to commit.

SC processors, however, cannot commit `read B` until `read A` commits or at least obtains coherence permission for the block containing A. `Read B` cannot be committed because it may need to be unrolled if the block containing B must be invalidated due to an exclusive request by another processor before this processor obtains coherence permission to block A. These sorts of techniques are already used in the HP PA-8000, Intel Pentium Pro, and MIPS R10000.

While the speculative techniques are complex, their implication is simple: Relaxed and SC implementations can do all the same speculations, but sometimes relaxed implementations can commit memory operations sooner. Thus, the performance gap between relaxed and SC implementations should narrow. The gap will be nonzero, however, if SC implementations

- undo instructions more often or
- more frequently exhaust implementation resources for uncommitted instructions.

Measuring the current performance gap depends on benchmarks, memory latencies, and myriad processor and cache implementation parameters.

Some current numbers

Parthasarathy Ranganathan and colleagues[8,9] provide an example of the academic state of the art in 1997. They simulate a workload of six scientific applications on a MIPS R10000-like processor using four-way instruction issue, dynamic scheduling with a 64-instruction window, speculative execution, caches that support outstanding misses to up to eight distinct blocks, and many other assumptions that can be found in the paper. The six scientific applications tested in Figure 6 are Radix, Fft, and Lu,[10] Water and MP3d,[11] and Erlebacher.[12]

I report on four memory consistency model implementations. SCimpl is an aggressive implementation of SC that uses hardware prefetching, speculative loads, and store buffering. PCimpl relaxes write-to-read program order and uses prefetching, speculative loads, and store buffering. RCimpl relaxes all orders and can use prefetching, speculative loads, and store buffering. SCimpl++ is an SC implementation that uses much more hardware and some new ideas described in Ranganathan and colleagues.[9]

Figure 6 shows application execution times from Ranganathan and colleagues[9] normalized to the execution time for SCimpl. PCimpl improves on SCimpl's execution by 0.8 to 23.6 percent while RCimpl provides 0.0 to 37.2 percent improvement. SCimpl++ shows that the performance gap can be narrowed with much more hardware, but this comparison is unfair since PCimpl and RCimpl could also use more hardware. (I quote the RCimpl version that has the best performance overall. This version disables prefetching and speculative loads whose overheads slightly surpass their benefits because RCimpl already overlaps so many memory references.)

The performance gap for the whole workload depends on how often and long each program is run. If we simplistically assume that each program runs for a fixed amount of time under SCimpl, then workload execution time under a model is the arithmetic average of program execution times. Doing this means that PCimpl and RCimpl reduce execution time by 10 percent and 16 percent, respectively. These numbers correspond to performance improvements of 11 percent for PCimpl and 20 percent for RCimpl.

The performance gap on other workloads would be different and might be smaller. Relaxed models are designed for the instruction-level parallelism of scientific workloads, which tends to be larger than those for other workloads, such as OSs and databases.

The next ten years

It is easy to argue that over the next 10 years the performance gap will grow, because the latency to memory—measured in instruction issue opportunities—is likely to grow. On the other hand, I see two reasons that make the performance gap likely to shrink.

First, future microprocessor designers will be able to apply more transistors to enhance the effectiveness of known techniques for improving memory system performance. These techniques range from mundane measures like larger caches and more concurrent cache misses to sophisticated speculation and prefetching techniques. Increasing the instruction window size, for example, will improve the performance of both SC and relaxed implementations by making instruction-window-full stalls less likely. The increased window size will also reduce the performance gap if the absolute difference in stalls gets smaller.

Second, architects will invent new microarchitectural techniques that, with speculation, will be applied to both SC and relaxed models. Some of these techniques are already gestating, as can be found in a recent special issue of *Computer* on billion-transistor architectures (Sept. 1997) and in annual proceedings of conferences.

In 1996, Intel's Albert Yu[13] re-examined the company's 1989 predictions. He found that the predictions were accurate on technology (like the projected number of transistors per chip), but that they underestimated processor performance by a factor of four because they didn't anticipate the speed of microarchitecture innovation. I expect this innovation to continue.

If the performance gap is less than 20 percent, what will happen with relaxed models? Will middleware authors still find it worthwhile to program with relaxed models? The answer depends on how much burden relaxed models add to middleware authors.

REASONING WITH RELAXED MODELS

Before considering relaxed models, we need to consider the context. Authoring parallel middleware is difficult. Many programming projects already stretch the intellectual limits of programmers to manage complexity while adding features, making behavior more robust, and staying on schedule. Dealing with relaxed models must necessarily either add a real cost (like personnel or schedule delays) or an opportunity cost (meaning that something else is not done).

The costs of using relaxed models depend, in large part, on the complexity of reasoning with them. I find reasoning with relaxed models in the class *relaxing all orders* more difficult than reasoning with SC. Even though I have coauthored a half-dozen papers on the subject, I still have to think carefully before I can correctly make any precise statement about one of the existing models. Certainly middleware authors can understand the models, but do they want to spend their time dealing with definitions of various partial orders and nonatomic operations? (A nonatomic operation allows its effects to be seen by some processors before others, in a manner detectable by running programs.)

The performance gained by using relaxed models does not justify their complexity.

Middleware authors must understand the models to a fairly good level of detail to be able to include sufficient fences without adding too many unnecessary ones. Too many unnecessary fences will reduce the performance gap seen in practice. In addition, authors of portable middleware (like compilers) need to master different relaxed models for different hardware targets.

What about hardware memory consistency models in the class *relaxing write-to-read program order*? In my opinion, middleware authors targeting commercial models of this class have an easier task than those targeting the class *relaxing all orders*. Middleware authors can reason with SC and not have to consider placing fences as long as they avoid using convoluted code.

On the other hand, compiler writers must still understand these models if they want to ensure that convoluted code will behave as written.

THE BOTTOM LINE

I recommend that future systems implement SC as their hardware memory consistency model. I do not believe that the performance boost from implementing models in the class *relaxing all orders* is enough to justify the additional intellectual burden the relaxed models place on the middleware authors of complex commercial systems.

There are, however, several other viable alternatives. First, you could provide a first-class SC implementation and add optional relaxed support through additional instructions—like Sun's `block copy` instructions—or multiple memory consistency model modes.[5]

You must exercise care when adding options, however, because doing so incurs both implementation and verification costs. Multiple modes, in particular, can add significant verification costs if they enable a large new cross-product of hardware interactions.

Second, you can implement a model in the class *relaxing write-to-read program order*. These models allow hardware to play a few tricks more easily than with SC without affecting most middleware authors. This option makes most sense for new implementations of existing systems that have already relaxed write-to-read program order. It can lead to subtle compatibility problems, however, if old systems provide SC. You can also use optional relaxed support with these models.

Let me close by comparing instruction sets and hardware memory consistency models, two interfaces on the hardware/software boundary. Almost all current instruction sets present programmers and compilers with a sequential model (for each processor). Current implementations, however, now use complex pipelines, out-of-order execution, and speculative execution to perform instructions out of program order, while at the same time using considerable logic to preserve the appearance of program order to software.

For the memory consistency model interface, we have a similar choice. With SC, we can hide the out-of-order complexity from software at some cost in implementation complexity. With relaxed models, complexity is visible to the software interface. As with instruction sets, we should use SC to keep complexity off the interface and in the implementation where it belongs. ❖

Acknowledgments

The ideas in this article crystallized through interactions with many people at Wisconsin and at Sun Microsystems during my 1995-1996 sabbatical, which Greg Papadopoulos graciously supported. I thank the following people, who may or may not agree with me, for their constructive comments: Sarita Adve, Doug Burger, William Collier, Babak Falsafi, Kourosh Gharachorloo, Andy Glew, John Hennessy, Rebecca Hoffman, Alain Kägi, Shubu Mukherjee, Thomas Ruf, Jim Smith, Guri Sohi, and Dan Sorin. This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, NSF Grants MIP-9225097 and MIPS-9625558, and donations from Sun Microsystems.

References

1. D.A. Wood and M.D. Hill, "Cost-Effective Parallel Computing," *Computer*, Feb. 1995, pp. 69-72.
2. L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. on Computers*, Sept. 1989, pp. 690-691.
3. S.V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Computer*, Dec. 1996, pp. 66-76.
4. S.V. Adve, *Designing Memory Consistency Models for Shared Memory Multiprocessors*, doctoral dissertation, CS Dept., Univ. Wisconsin-Madison, Nov. 1993.
5. K. Gharachorloo, *Memory Consistency Models for Shared Memory Multiprocessors*, doctoral dissertation, Computer Systems Laboratory, Stanford University, Stanford Calif. Dec. 1995.
6. A. Gupta et al., "Comparative Evaluation of Latency Reducing and Tolerating Techniques," *Proc. 18th Annual Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., June 1991, pp. 254-263.
7. K. Gharachorloo, A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," *Proc. 1991 Int. Conf. Parallel Processing*, IEEE CS Press, Los Alamitos, Calif., Aug. 1991, pp. 355-364.
8. P. Ranganathan et al., "The Interaction of Software Prefetching with ILP Processors in Shared Memory Systems," *Proc. 24th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., June 1997, pp. 144-156.
9. P. Ranganathan, V.S. Pai, and S.V. Adve, "Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models," *Proc. Ninth ACM Symp. Parallel Algorithms and Architectures (SPAA)*, ACM Press, New York, June 1997, pp. 199-210.
10. S.C. Woo et al., "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Annual Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., June 1995, pp. 24-36.
11. J.P. Singh, W.D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared Memory," *Proc. 19th Annual Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., May 1992, pp. 5-14.
12. V.S. Adve et al., "An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs," *Proc. Supercomputing '95*, ACM Press, New York, 1995.
13. A. Yu, "The Future of Microprocessors," *IEEE Micro*, Dec. 1996, pp. 46-53.

Mark D. Hill is professor and Romnes fellow in both the Computer Sciences Department and the Electrical and Computer Engineering Department at the University of Wisconsin, Madison. He also codirects the Wisconsin Wind Tunnel parallel computing project with James Larus and David Wood. His current research interests include memory systems of shared memory multiprocessors and high-performance uniprocessors. Hill received a BSE from the University of Michigan, Ann Arbor, and MS and PhD in computer engineering from the University of California, Berkeley. Contact Hill at markhill@cs.wisc.edu.