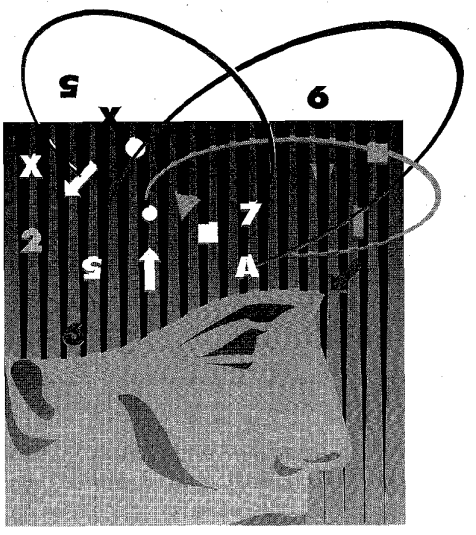

Shared Memory Consistency Models: A Tutorial

Sarita V. Adve
Rice University

Kourosh Gharachorloo
Digital Equipment Corporation

The memory consistency model of a system affects performance, programmability, and portability. This article describes several models in an easy to understand way.

The shared memory programming model has several advantages over the message passing model. In particular, it simplifies data partitioning and dynamic load distribution. Shared memory systems are therefore gaining wide acceptance for both technical and commercial computing.

To write correct and efficient shared memory programs, programmers need a precise notion of shared memory semantics. For example, in the program in Figure 1 (a fragment from a program in the Splash application suite), processor P1 repeatedly updates a data field in a new task record and then inserts the record into a task queue. When no tasks are left, P1 updates a pointer, `Head`, to point to the first record in the task queue. Meanwhile, the other processors wait for `Head` to have a non-null value, dequeue the task pointed to by `Head` in a critical section, and read the data in the dequeued task. To ensure correct execution, a programmer expects that the data value read should be the same as that written by P1. However, in many commercial shared memory systems, the processors may observe an older value, causing unexpected behavior.

The *memory consistency model* of a shared memory multiprocessor formally specifies how the memory system will appear to the programmer. Essentially, a memory consistency model restricts the values that a read can return. Intuitively, a read should return the value of the "last" write to the same memory location. In uniprocessors, "last" is precisely defined by the sequential order specified by the program, called the *program order*. This is not the case in multiprocessors. For example, in Figure 1 the write and read of `Data` are not related by program order because they reside on two different processors.

The uniprocessor model, however, can be extended to apply to multiprocessors in a natural way. The resulting model is called *sequential consistency*. Informally, sequential consistency requires that all memory operations appear to execute one at a time and that all operations of a single processor appear to execute in the order described by that processor's program. For Figure 1, this model ensures that the reads of the data field will return the new values written by processor P1. Sequential consistency provides a simple, intuitive programming model. However, it disallows many uniprocessor hardware and compiler optimizations. For this reason, many *relaxed consistency* models have been proposed, several of which are supported by commercial architectures.

The memory consistency model is an interface between the programmer and the system, so it influences not only how parallel programs are written but virtually every aspect of parallel hardware and software design. A memory consistency model specification is required at every interface between the programmer and the system, including the interfaces at the machine-code and high-level language levels. In particular, the high-level language specification affects high-level language programmers, compiler and other software writers who convert high-level

0018-9162/96/$5.00 © 1996 IEEE

```
        Initially all pointers = null, all integers = 0.
P1                                  P2, P3, ..., Pn
while (there are more tasks)        while (MyTask == null) {
  Task = GetFromFreeList();           Begin Critical Section
  Task → Data = ...;                  if (Head != null) {
  insert Task in task queue             MyTask = Head;
}                                       Head = Head → Next;
Head = head of task queue;            }
                                      End Critical Section
                                    }
                                    ... = MyTask → Data;
```

Figure 1. Illustration of the need for a memory consistency model.

code into machine code, and the designers of hardware that executes the code. At each level, the memory consistency model affects both *programmability* and *performance*. Furthermore, due to a lack of consensus on a single model, *portability* can be affected when moving software across systems supporting different models.

Unfortunately, the vast literature that describes consistency models uses nonuniform and complex terminology to describe the large variety of models. This makes it difficult to understand the often subtle but important differences among models and leads to several misconceptions, some of which are listed in the "Myths about memory consistency models" sidebar.

In this article, we aim to describe memory consistency models in a way that most computer professionals would understand. This is important if the performance-enhancing features being incorporated by system designers are to be correctly and widely used by programmers. Our focus is consistency models proposed for hardware-based shared memory systems. Most of these models emphasize the system optimizations they support, and we retain this system-centric emphasis in this article. We also describe an alternative, programmer-centric view of relaxed consistency models that describes them in terms of program behavior, not system optimizations. A more formal treatment is covered in our other work.[1-3]

UNIPROCESSOR MEMORY CONSISTENCY

Most high-level uniprocessor languages present simple sequential-memory semantics, which allow the programmer to assume that all memory operations will occur one at a time in program order. Fortunately, this illusion of sequentiality can be supported efficiently by simply ensuring that two operations are executed in program order if they are to the same location or if one controls the execution of the other. The compiler or the hardware can freely reorder other operations, enabling several optimizations. Overall, the sequential-memory semantics of a uniprocessor provide a simple and intuitive model and yet allow a wide range of efficient system designs.

UNDERSTANDING SEQUENTIAL CONSISTENCY

The most commonly assumed memory consistency model for shared memory multiprocessors is *sequential consistency*, which gives programmers a simple view of the system. A multiprocessor system is 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 appear in this sequence in the order specified by its program."[4]

There are two requirements for sequential consistency:

- maintaining program order among operations from a single processor, and
- maintaining a single sequential order among all operations.

Myths about memory consistency models

Myth: A memory consistency model applies only to systems that allow multiple copies of shared data (through caching, for example).

Reality: There are several counterexamples, some of which are shown in Figure 3.

Myth: Most modern systems are sequentially consistent.

Reality: Several commercial systems are not, including the ones listed in Table 2.

Myth: Memory consistency models affect only hardware design.

Reality: Memory consistency models affect many aspects of system design, including which optimizations the compiler can exploit.

Myth: A cache coherence protocol inherently supports sequential consistency.

Reality: The cache coherence protocol is only part of the memory consistency model. Other aspects include the order in which a processor issues memory operations to the memory system and whether a write executes atomically.

Myth: The memory consistency model depends on whether the system supports an invalidate or update-based coherence protocol.

Reality: A given memory consistency model can allow both invalidate and update coherence protocol.

Myth: A system's memory model may be defined solely by specifying processor (or memory system) behavior.

Reality: The memory consistency model is affected by the behavior of both the processors and the memory system.

Myth: Relaxed memory consistency models may not be used to hide read latency.

Reality: Many models can hide both read and write latencies.

Myth: Relaxed consistency models require the use of extra synchronization.

Reality: Most models do not. Our programmer-centric framework requires only that the operations be distinguished or labeled correctly. Other models provide safety nets that let the programmer enforce any constraints for achieving correctness.

Myth: Relaxed memory consistency models do not allow chaotic (or asynchronous) algorithms.

Reality: Most do. With system-centric models, the programmer can reason about correctness of such algorithms by considering the optimizations the model enables. The programmer-centric approach simply requires the programmer to explicitly identify operations involved in a race. For many chaotic algorithms, the former approach may provide higher performance, since such algorithms often do not rely on sequential consistency.

The second requirement makes a memory operation appear to execute atomically (instantaneously) with respect to other memory operations. A sequentially consistent system can be thought of as consisting of a single global memory connected to all the processors by a central switch. At any time step, the switch connects memory to an arbitrary processor, which may then issue a memory operation. Each processor issues memory operations in program order, and the switch provides the global serialization among all memory operations.

Figure 2a illustrates the first requirement for program order. The figure depicts Dekker's algorithm for critical sections. It involves two flag variables initialized to 0. When processor P1 attempts to enter the critical section, it updates Flag1 to 1, and checks the value of Flag2. The value 0 for Flag2 indicates that processor P2 has not yet tried to enter the critical section, so it is safe for P1 to enter. The algorithm assumes that if P1's read returns 0, then P1's write occurred before P2's write and read. P2 will read the flag and return 1, which will prohibit it from also entering the critical section. Sequential consistency ensures this by maintaining program order.

Figure 2b illustrates the atomicity requirement. In this case, three processors share variables A and B, which are initialized to 0. Suppose P2 returns 1 when it reads A and then writes to B, and suppose P3 returns 1 when it reads B. Atomicity allows us to assume that P1's write is seen by the entire system at the same time. Since P3 sees P2's write to B after P2 sees P1's write to A, it follows that P3 is guaranteed to see P1's write and return 1 when it reads A.

IMPLEMENTING SEQUENTIAL CONSISTENCY

In this section we explain how to practically realize sequential consistency in a multiprocessor system. We will see that unlike uniprocessors, preserving only per-processor data and control dependencies is insufficient. We first focus on how sequential consistency interacts with common hardware optimizations and then briefly describe compiler optimizations. To separate the issues of program order and atomicity, we begin with implementations for architectures without caches and then discuss the effects of caching shared data.

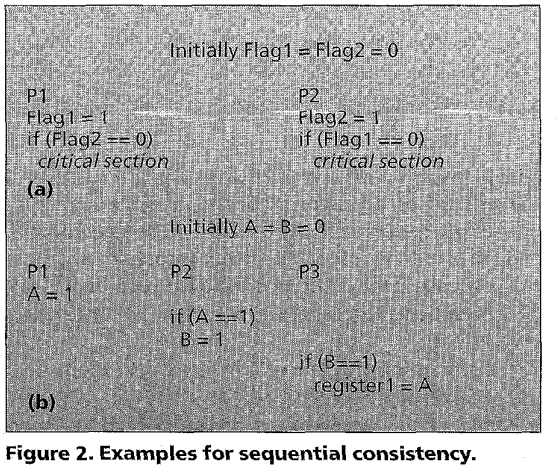

Figure 2. Examples for sequential consistency.

Architectures without caches

The key issue in supporting sequential consistency in systems without caches is program order. To illustrate the interactions that arise in such systems, we will use three typical hardware optimizations, shown in Figure 3. The notations $t1$, $t2$, and so on in the figure indicate the order in which the corresponding memory operations execute at memory.

WRITE BUFFERS WITH READ BYPASSING. The optimization depicted in Figure 3a shows the importance of maintaining program order between a write and a following read, even if there is no data or control dependence between them. In this bus-based system, assume that a simple processor issues operations one at a time, in program order. Now add the optimization of a write buffer. A processor can insert a write into the buffer and proceed without waiting for the write to complete. Subsequent reads of the processor can bypass the buffered writes (to different addresses) for faster completion.

Write buffers can violate sequential consistency. For the code in Figure 3a, a sequentially consistent system must not allow both processors' reads of flags to return 0. However, this can happen in the system in Figure 3a: Each processor can buffer its write and allow the subsequent read to bypass it. Therefore, both reads may be serviced by memory before either write, allowing both reads to return 0.

OVERLAPPING WRITES. The optimization depicted in Figure 3b shows the importance of maintaining program order between two writes. Again, we consider operations with no data or control dependencies. This system has a general (nonbus) network and multiple memory modules, which can exploit more parallelism than the system in Figure 3a. Now multiple writes of a processor may be simultaneously serviced by different memory modules.

This optimization can also violate sequential consistency. In the code fragment in Figure 3b, assume that `Data` and `Head` reside in different memory modules. Because the write to `Head` may be injected into the network before the write to `Data` has reached its memory module, the two writes could complete out of program order. Therefore, P2 might see the new value of `Head` and yet get the old value of `Data`, a violation of sequential consistency.

To maintain program order among writes, an acknowledgment can be returned to the processor that issued the write once the write has reached its target memory module. The processor could be constrained from injecting another write until it receives an acknowledgment of its previous write.

This write acknowledgment technique can also maintain program order from a write to a subsequent read in systems with general networks.

NONBLOCKING READS. The optimization in Figure 3c illustrates the importance of maintaining program order between a read and a following operation. While most early RISC processors blocked on a read until it returned a value, recent processors proceed past reads, using techniques such as lockup-free caches and dynamic scheduling. In Figure 3c, two overlapped reads violate

sequential consistency in a manner similar to overlapped writes. A read overlapped with a following write causes similar problems, but this kind of overlap is not common.

PROGRAM ORDER REQUIREMENT. The above discussion shows that in straightforward hardware implementations, a processor must ensure that its previous memory operation is complete before proceeding with its next memory operation in program order. We call this requirement the *program order requirement*.

Architectures with caches

Caching, or replication, of shared data can lead to scenarios similar to those described for systems without caches. Systems that use caching must therefore take similar precautions to maintain the illusion of program order. Most notably, even if a read hits in its processor's cache, reading the cached value without waiting for the completion of previous operations can violate sequential consistency. The replication of shared data introduces three additional issues.

CACHE COHERENCE PROTOCOLS. The presence of multiple cache copies requires a mechanism, often called a *cache coherence protocol*, to propagate a new value to all copies of the modified location. A new value is propagated by either *invalidating* (or eliminating) or *updating* each copy.

The literature includes several definitions of cache coherence (which is sometimes called *cache consistency*). The strongest definitions are virtually synonymous with sequential consistency. Other definitions impose extremely relaxed orderings. One common definition requires two conditions for cache coherence:

- A write must eventually be made visible to all processors.
- Writes to the same location must appear to be seen in the same order by all processors.[5]

These conditions are clearly not sufficient to satisfy sequential consistency. This is because sequential consistency requires that writes to *all* locations (not just the same location) be seen in the same order by all processors, and also explicitly requires that a single processor's operations appear to execute in program order.

We do not use the term cache coherence to define a consistency model. We view a cache coherence protocol as simply a mechanism to propagate a newly written value. The memory consistency model is the policy that places the bounds on when the value can be propagated to a given processor.

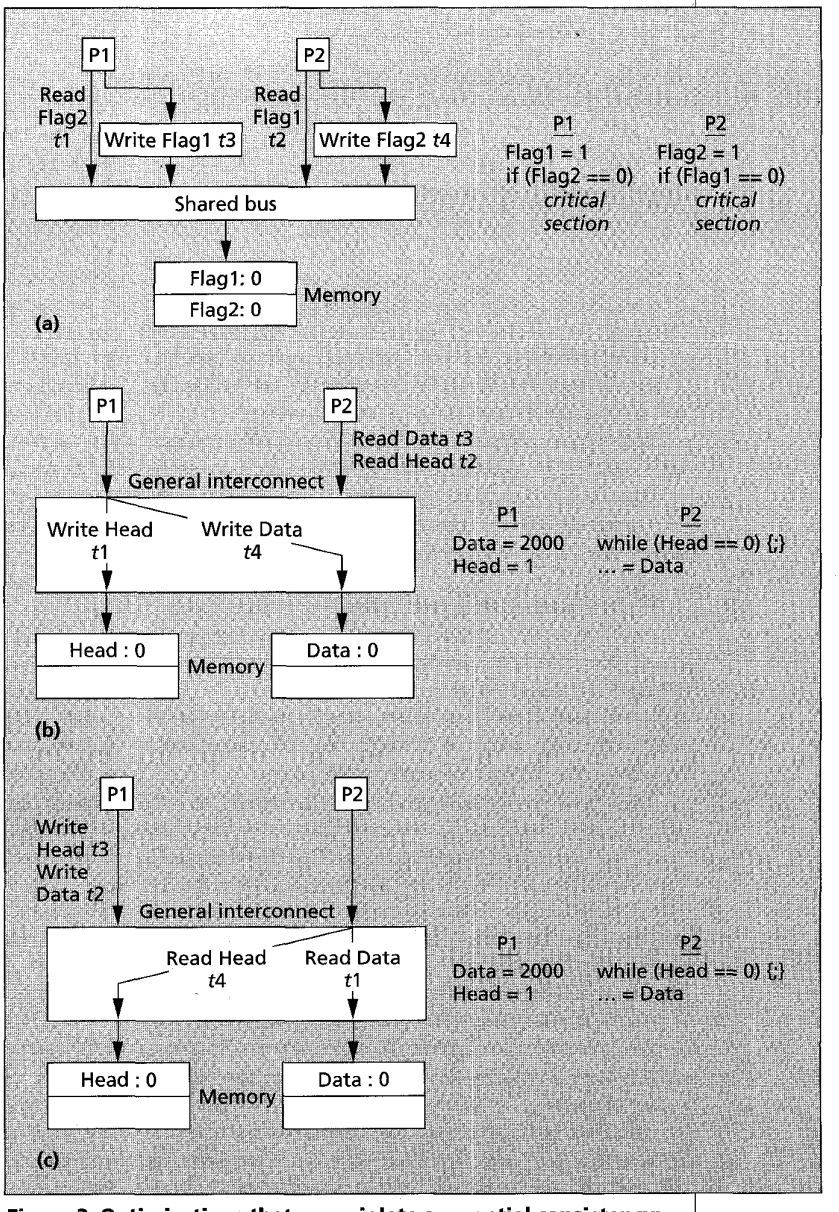

Figure 3. Optimizations that may violate sequential consistency; *t1, t2, ...* indicate the order in which the corresponding memory operations execute at memory

DETECTING WRITE COMPLETION. When there are no caches, a write acknowledgment may be generated when the write reaches its target memory. However, an acknowledgment at this time is too early for a system with caches.

Suppose write-through caches were added to each processor in Figure 3b. Assume P2 initially has `Data` in its cache. Now suppose P1 proceeds to write to `Head` after the write to `Data` reaches its target memory but before its value has been propagated to P2. It is possible that P2 could read the new value of `Head` and still return the old value of `Data` from its cache, a violation of sequential consistency.

P1 must wait for P2's copy of `Data` to be updated or invalidated before it writes to `Head`. Thus, a write to a line replicated in other caches typically requires an acknowl-

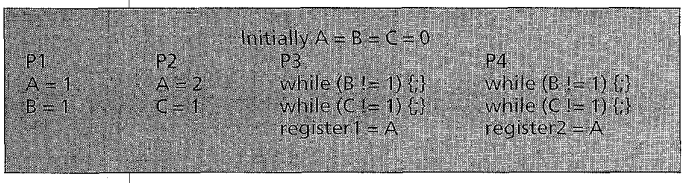

Figure 4. Example for serialization of writes.

edgment of invalidate or update messages as well. Furthermore, the acknowledgments must be collected either at the memory or at the processor that issues the write. In either case, the writing processor must be notified when all acknowledgments are received. Only then can the processor consider the write to be complete.

A common optimization is to have each processor acknowledge an invalidate or update immediately on receipt and potentially before its cache copy is affected. This design can satisfy sequential consistency if it supports certain ordering constraints in processing all incoming messages.[3]

MAINTAINING WRITE ATOMICITY. Propagating changes to multiple cache copies is inherently a nonatomic operation. Therefore, special care must be taken to preserve the illusion of write atomicity.

In this section we describe two conditions that together ensure the appearance of atomicity. We will refer to these conditions as the *write atomicity requirement*.

The first condition requires writes to the *same* location to be *serialized*. That is, all processors should see writes to the same location in the same order. Figure 4 illustrates the need for this condition: Assume an update protocol and that all processors in Figure 4 execute memory operations one at a time and in program order. Sequential consistency is violated if the updates of the writes of A by P1 and P2 reach P3 and P4 in a different order. If this happens, P3 and P4 will return different values when they read A and the writes of A appear nonatomic. This can occur in systems with a general (nonbus) network that do not guarantee the delivery order of messages that traverse different paths. Requiring serialization of writes to the *same* location solves this problem. One way to achieve serialization is to ensure that all updates or invalidates for a location originate from a single point (such as the directory) and the network preserves the ordering of messages between a given source and destination. An alternative is to delay updates or invalidates until those issued for a previous write to the same line are acknowledged.

The second condition prohibits a read from returning a newly written value until all cached copies have acknowledged receipt of the invalidates or updates generated by the write (that is, until the write becomes visible to all processors). Assume, for example, that all variables in Figure 2b are initially cached by all processors. Furthermore, assume a system with all the precautions for sequential consistency except for the above condition. It is still possible to violate sequential consistency with a general network with an update protocol if

1. P2 reads the new value of A,
2. P2's update of B reaches P3 before the update of A, and
3. P3 returns the new value of B and the old value of A from its cache.

P2 and P3 will thus appear to see the write of A at different times, violating atomicity. A similar situation can arise in an invalidate scheme. This violation occurs because P2 returns the value of P1's write before the update for the write reaches P3. Prohibiting a read from returning a newly written value until all cached copies have acknowledged the updates for the write avoids this.

It is straightforward to ensure the second condition with invalidate protocols. Update protocols are more challenging because updates directly supply new values to other processors. One solution for update protocols is to employ a two-phase update scheme: The first phase sends updates and receives acknowledgments. In this phase, no processor is allowed to read the updated location. In the second phase, a confirmation message is sent to the updated processor caches to confirm the receipt of all acknowledgments. A processor can use the updated value from its cache once it receives this confirmation.

Compilers

Compilers that reorder shared memory operations can cause sequential consistency violations similar to hardware. For all the program fragments discussed so far, compiler-generated reordering of shared memory operations will lead to sequential consistency violations similar to hardware-generated reorderings. Therefore, in the absence of more sophisticated analysis, the compiler must preserve program order among shared memory operations. This prohibits any uniprocessor compiler optimization that might reorder memory operations, including simple optimizations—code motion, register allocation, and eliminating common subexpressions—and more sophisticated optimizations—loop blocking and software pipelining.

Besides reordering, compiler optimizations such as register allocation can also cause the elimination of shared memory operations. This can also lead to sequential consistency violations in subtle ways. In Figure 3b, for example, if the compiler register allocates `Head` on P2 (by doing a single read of `Head` into a register and then reading the value from the register), the loop on P2 may never terminate if the single read returns the old value of `Head`. Sequential consistency requires this loop to terminate in every execution.

Optimizations like register allocation are key to performance, so most compilers for sequentially consistent systems perform them. It is left to the programmer to explicitly disable them when necessary, using mechanisms such as the volatile declaration. But it is difficult to determine when disabling is necessary—it requires reasoning that is similar to the reasoning for relaxed consistency models.

The above discussion applies to compilers for explicitly parallel code; compilers that parallelize sequential code naturally have enough information about the generated parallel program to determine when an optimization is safe to apply.

Optimizations for sequential consistency

Several techniques have been proposed to enable the use of some hardware and compiler optimizations without vio-

lating sequential consistency. Here, we include the ones that have the potential to substantially boost performance.

HARDWARE TECHNIQUES. Two hardware techniques for cache-coherent systems are supported by several recent microprocessors (the Hewlett-Packard PA-8000, the Intel P6, and the MIPS R10000).[6] The first technique automatically prefetches ownership for any write operation that is delayed due to the program order requirement (for example, by issuing prefetch-exclusive requests for writes delayed in the write buffer), thus partially overlapping the service of the delayed writes with previous operations. The second speculatively services read operations that are delayed due to the program order requirement. Sequential consistency is guaranteed by simply rolling back and reissuing the read and subsequent operations, if the read line gets invalidated or updated before the read could have been issued in a more straightforward implementation. Because dynamically scheduled processors already include much of the necessary rollback machinery (to deal with branch mispredictions), they are particularly well-suited to this technique.

A recent study has shown that these two techniques dramatically improve the performance of sequential consistency.[7] However, in many cases a significant performance gap remains between sequential consistency and the relaxed consistency model of release consistency.

Other latency hiding techniques, such as nonbinding software prefetching or hardware support for multiple contexts, have also been shown to enhance the performance of sequentially consistent hardware. However, these techniques are also beneficial when used in conjunction with relaxed memory consistency.

COMPILER TECHNIQUES. A compiler algorithm to detect when memory operations can be reordered without violating sequential consistency has been proposed.[8] Such an analysis can be used to implement both hardware and compiler optimizations. This algorithm has exponential complexity. More recently, a new algorithm with polynomial complexity has been proposed.[9]

However, both algorithms require global dependence analysis to determine whether two operations from different processors can conflict. This analysis is difficult and often leads to conservative estimates that can decrease the algorithms' effectiveness. It remains to be seen if these algorithms can approach the performance of relaxed consistency models.

RELAXED MEMORY MODELS

Relaxed memory consistency models typically emphasize the system optimizations they enable and are based on widely varying specification methods and levels of formalism. We retain the system-centric emphasis, but describe the models using a simpler, more uniform terminology. A more formal and unified system-centric framework, along with formal descriptions of these models, has been published elsewhere.[2,3]

Model types

We use two key characteristics to categorize relaxed memory consistency models:

- *How they relax the program order requirement.* Models differ on the basis of how they relax the order from a write to a following read, between two writes, and from a read to a following read or write. These relaxations apply only to operation pairs with different addresses and are similar to the optimizations for sequential consistency described for architectures without caches.
- *How they relax the write atomicity requirement.* Some models allow a read to return the value of *another* processor's write before the write is made visible to all other processors. This relaxation applies only to cache-based systems.

We also consider a relaxation related to both program order and write atomicity, where a processor is allowed to read the value of its *own* previous write before the write is made visible to other processors and, in a cache-based system, before the write is serialized. A common optimization that exploits this relaxation is forwarding the value of a write in a write buffer to a following read from the same processor.

The relaxed models discussed here also typically provide mechanisms for overriding their default relaxations. For example, explicit *fence* instructions may be used to override program order relaxations. We call these mechanisms *safety nets*. We discuss only the more straightforward safety nets here.

Relaxed memory consistency models typically emphasize the system optimizations they enable.

Table 1 (on the next page) lists the relaxations and safety nets for the models we discuss here, and Table 2 lists example commercial systems that allow such relaxations. For simplicity, we do not attempt to describe the models' semantics with respect to issues such as instruction fetches, I/O operations, or multiple granularity operations (byte versus word operations, for example), even though some models define such semantics.

Throughout this section, we assume that the following constraints are satisfied:

- We assume that all models require both that a write eventually be made visible to all processors and that writes to the same location be serialized. If shared data is not cached, these requirements are trivial; otherwise they are met by a hardware cache coherence protocol.
- We assume that all models enforce uniprocessor data and control dependencies.
- We assume that models that relax the program order from reads to following write operations also maintain a subtle form of multiprocessor data and control dependence.[1,2] This constraint is inherently upheld by all processor designs we know of and can be easily maintained by the compiler.

Relaxing write to read program order

These models allow a read to be reordered with respect to previous writes from the same processor. Therefore, programs such as the one in Figure 3a may fail to provide sequentially consistent results.

Table 1. Simple categorization of relaxed models.

Relaxation	W → R order	W → W order	R → RW order	Read others' write early	Read own write early	Safety net
SC[1]					✓	
IBM 370[2]	✓					Serialization instructions
TSO[3]	✓				✓	Read-modify-write
PC[4]	✓			✓	✓	Read-modify-write
PSO[3]	✓	✓			✓	Read-modify-write, STBAR
WO[5]	✓	✓	✓		✓	Synchronization
RCsc[4]	✓	✓	✓		✓	Release, acquire, nsync, read-modify-write
RCpc[4]	✓	✓	✓	✓	✓	Release, acquire, nsync, read-modify-write
Alpha[6]	✓	✓	✓		✓	MB, WMB
RMO[3]	✓	✓	✓		✓	Various MEMBARs
PowerPC[7]	✓	✓	✓	✓	✓	Sync

A ✓ indicates that the corresponding relaxation is allowed by straightforward implementations of the corresponding model. It also indicates that the relaxation can be detected by the programmer (by affecting the results of the program) except for the following cases. The read own write early relaxation is not detectable with the SC, WO, Alpha, and PowerPC models. The read others' write early relaxation is possible and detectable with complex implementations of RCsc.

1. L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. Computers*, Sept. 1979, pp. 690-691.
2. *IBM System/370 Principles of Operation*, IBM Pub. GA22-7000-9, File S370-01, 1983.
3. *SPARC Architecture Manual*, D.L. Weaver and T. Germond, eds., Prentice-Hall, Englewood Cliffs, N.J.1994.
4. K. Gharachorloo et al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Int'l Symp. Computer Architecture*, 1990, pp. 15-26.
5. M. Dubois, C. Scheurich, and F. Briggs, "Memory Access Buffering in Multiprocessors," *Proc. 13th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1986, pp. 434-442.
6. *Alpha AXP Architecture Reference Manual 2-Ed.*, R.L. Sites and R.T. Witek, eds., Digital Press, Boston 1995.
7. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, C. May et al., eds., Morgan Kaufmann, San Francisco, 1994.

As Table 1 shows, the three models in this group—IBM 370, total store ordering (TSO), and processor consistency (PC)—differ in when they allow a read to return the value of a write. Figure 5 illustrates these differences.

As a safety net, the IBM 370 provides special *serialization instructions* that can be used to enforce program order between a write and a following read. Some serialization instructions, such as compare&swap, are memory operations used for synchronization. Others are nonmemory instructions, such as a branch. The IBM 370 does not need a safety net for write atomicity because it does not relax atomicity.

In contrast, the TSO and PC models do not provide explicit safety nets. Nevertheless, programmers can use read-modify-write operations to provide the illusion that program order is maintained from a write to a read or that writes are atomic.[2,3] Fortunately, most programs do not depend on write-to-read program order or write atomicity for correctness.

Relaxing program order as these models do can substantially improve performance at the hardware level by effectively hiding the latency of write operations.[10] At the compiler level, however, this relaxation alone is not beneficial. Most compiler optimizations require the extra flexibility of reordering any two operations (read or write) with respect to one another.

Relaxing write to read and write to write program order

These models allow writes to different locations from the same processor to be pipelined or overlapped, and so they may reach memory or other cached copies out of program order. Therefore, these models can violate sequential consistency for the programs in Figures 3a and 3b. The partial store ordering model (PSO) is the only model we describe here.

With respect to atomicity requirements, PSO is identical to TSO. However, PSO adds a safety net, the STBAR instruction, which imposes program order between two writes. As with the previous three models, the optimizations allowed by PSO are not sufficiently flexible to be useful to a compiler.

Relaxing all program orders

The final set of models relax program order between all operations to different locations, allowing a read or write to be reordered with respect to a following read or write. Thus, they may violate sequential consistency for all the examples shown in Figure 3. The key additional optimization relative to the previous models is that memory operations following a read operation may be overlapped or reordered with respect to the read. This flexibility allows hardware to hide the latency of reads with either statically (in-order) or

Table 2. Some commercial systems that relax sequential consistency.

Relaxation	Example commercial systems providing relaxation
W→ R order	AlphaServer 8200/8400, Cray T3D/T3E, Sequent Balance and NUMA-Q, SparcCenter 1000/2000, Ultra Enterprise Servers, Convex SPP systems in weak ordering mode
W → W order	AlphaServer 8200/8400, Cray T3D/T3E, Convex SPP systems in weak ordering mode
R → RW order	AlphaServer 8200/8400, Cray T3D/T3E, Convex SPP systems in weak ordering mode
Read others' write early	Cray T3D
Read own write early	AlphaServer 8200/8400, Cray T3D/T3E, SparcCenter1000/2000, Ultra Enterprise Servers

dynamically (out-of-order) scheduled processors.[3,7]

We discuss six models in this class: the weak ordering (WO) model, two flavors of the release consistency model (RCsc and RCpc), and three models proposed for commercial architectures—the Digital Alpha, Sparc relaxed memory order (RMO), and IBM PowerPC. Except for Alpha, these models also allow the reordering of two reads to the same location.

Regarding atomicity, all models in this group allow a processor to read its own write early. RCpc and PowerPC are the only models whose straightforward implementations allow a read to return the value of another processor's write early. This can also happen in more complex implementations of WO, RCsc, Alpha, and RMO. From the programmer's perspective, however, all implementations of WO, Alpha, and RMO must preserve the illusion of write atomicity (while extremely aggressive implementations of RCsc may violate it). For WO, we assume that if a read and a following write are related by data or control dependence, then the write is delayed until both the read and the write read by the read are complete.

These six models fall into two main categories, on the basis of the type of safety net they provide. The WO, RCsc, and RCpc models distinguish memory operations based on their type and provide stricter ordering constraints for some operations. The Alpha, RMO, and PowerPC models provide explicit instructions to impose program orders between various memory operations.

WEAK ORDERING. The weak ordering model classifies memory operations into two categories: *data operations* and *synchronization operations*. To enforce program order between two operations, the programmer must identify at least one of them as a synchronization operation. Memory operations between two synchronization operations may still be reordered and overlapped. This model is based on the intuition that reordering memory operations to data regions between synchronization operations does not typically affect program correctness. Since WO ensures that writes appear to be atomic to the programmer, no safety net is required for write atomicity.

RELEASE CONSISTENCY. The release consistency models further distinguish memory operations. Operations are first distinguished as *ordinary* or *special,* categories that loosely correspond to the distinction between data and synchronization in WO. Special operations are further distinguished as *sync* or *nsync*. Sync operations are synchronization operations; nsyncs are either asynchronous data operations or special operations not used for synchronization. Finally, sync operations are further distinguished as *acquire* or *release* operations. An acquire is a read operation performed to gain access to shared locations (for example, a lock operation or spinning for a flag to be set). A release is a write operation performed to grant permission to access shared locations (for example, an unlock operation or setting of a flag).

There are two flavors of release consistency, RCsc and RCpc. RCsc maintains sequential consistency among special operations, while RCpc maintains processor consistency among such operations. RCsc maintains the program order from an acquire to any operation that follows it, from any operation to a release, and between special operations. RCpc is similar, except that the write-to-read program order among special operations is not maintained.

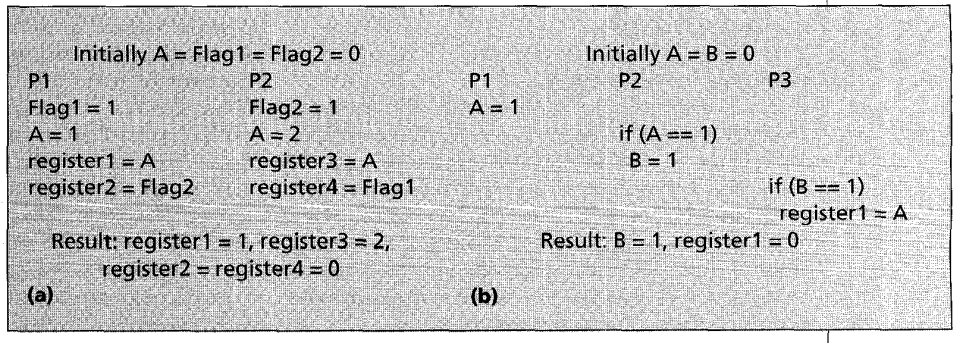

Figure 5. Differences between 370, TSO, and PC. The result for the program in part (a) is possible with TSO and PC because both models allow the reads of the flags to occur before the writes of the flags on each processor. The result is not possible with IBM 370 because the read of A on each processor is not issued until the write of A on that processor is complete. Consequently, the read of the flag on each processor is not issued until the write of the flag on that processor is done. The program in part (b) is the same as in Figure 2b. The result shown is possible with PC because it allows P2 to return the value of P1's write before the write is visible to P3. The result is not possible with IBM 370 or TSO.

Thus, for the RC models, program order between a pair of operations can be enforced by distinguishing or labeling appropriate operations based on the preceding information.

For RCpc, imposing program order from a write to a read or making a write appear atomic requires using read-modify-write operations as in the PC model.[2,5] Complex implementations of RCsc may also make writes appear nonatomic; one way to enforce atomicity is to label sufficient operations as special.[2,3] The RCsc model is accompanied by a higher level abstraction that relieves the programmer from having to use the lower level specification to reason about many programs.[5]

ALPHA, RMO, AND POWERPC. The Alpha, RMO, and PowerPC models all provide explicit *fence instructions*, as their safety nets.

The Alpha model provides two fence instructions: memory barrier (MB) and write memory barrier (WMB). Memory barrier instructions maintain program order between any memory operations that come before them and any memory operations that come after them. Write memory barrier instructions provide this guarantee only among write operations. The Alpha model does not require a safety net for write atomicity.

The RMO model provides more flavors of fence instructions. Effectively, a programmer can customize a memory barrier instruction (MEMBAR) to specify any combination of four possible pairs of orderings—between all writes followed by all reads, all writes followed by all writes, all reads followed by all reads, and all reads followed by all writes. This model also does not require a safety net for write atomicity.

The PowerPC model provides a single fence instruction, Sync. Sync behaves like the Alpha memory barrier, with one exception that can create subtle correctness problems: Even if a Sync is placed between two reads to the same location, the second read may return the value of an older write than the first read. In other words, the reads appear to occur out of program order. Unlike Alpha and RMO, PowerPC does not preserve write atomicity and may require the use of read-modify-write operations to make a write appear atomic.[3]

Compiler optimizations

The last set of models described are flexible enough to allow common compiler optimizations on shared memory operations. With WO, RCsc, and RCpc, the compiler can reorder memory operations between two consecutive synchronization or special operations. With the Alpha, RMO, and PowerPC models, the compiler can reorder operations between fence instructions. Most programs use these operations or instructions infrequently, so the compiler can safely optimize large regions of code.

PROGRAMMER-CENTRIC MODELS

Relaxed memory models enable a wide range of optimizations that have been shown to substantially improve performance.[3,7,10,11] However, they are harder for programmers to use. Furthermore, the wide range of models supported by different systems requires programmers to deal with various semantics that differ in subtle ways and complicate porting.

We need a higher level abstraction that provides programmers a simpler view, yet allows system designers to exploit the various optimizations.

Relaxed models are complex to program because their *system-centric specifications* directly expose the programmer to the reordering and atomicity optimizations, requiring the programmer to consider such optimizations when reasoning about program correctness. Even though relaxed models do provide safety nets, the programmer must still identify the ordering constraints necessary for correctness.

Instead of exposing optimizations directly to the programmer, a *programmer-centric specification* requires the programmer to provide certain information about the program. This information is then used by the system to determine whether a certain optimization can be applied without affecting the correctness of program execution. To provide a formal programmer-centric specification, we must first define when a program is considered to be executed correctly by the system. An obvious choice for correctness is sequential consistency, because it is a natural extension of the uniprocessor notion of correctness and the most commonly assumed multiprocessor correctness model. Once we have defined a correctness notion, we must precisely define the information required from the programmer.

So our programmer-centric approach describes a memory model in terms of program-level information that a programmer must provide, and then exploits this information to perform optimizations without violating sequential consistency.

We have described various programmer-centric approaches elsewhere: The data-race-free-0 approach allows WO-like optimizations,[12] the properly-labeled approach is a simpler way to write programs for RCsc,[5] and other approaches exploit more aggressive optimizations.[1,3] We have also developed a unified framework to explore the design space of programmer-centric models and optimizations.[1]

Sample programmer-centric framework

To illustrate the programmer-centric approach, we describe program-level information that can enable WO-like optimizations. Recall that weak ordering is based on the intuition that memory accesses can be classified as either data or synchronization, and that data operations can be executed more aggressively than synchronization operations. However, the informal nature of this classification makes it ambiguous when applied over a wide range of programs. A key goal of the programmer-centric approach is to formally define the operations that should be distinguished as synchronization.

An operation is a synchronization operation if it forms a *race* with another operation in any sequentially consistent execution. All other operations are data operations. Given a sequentially consistent execution, two operations form a race with each other if they access the same location, if at least one is a write, and if there are no other operations between them. For example, in every sequentially consistent execution of the program in Figure 3b, the write and read of `Data` are separated by intervening operations on `Head`. In this case the former set are data operations. In contrast, operations on `Head` are not always separated by other

operations, so they are synchronization operations.

To provide this information, the programmer must reason only about sequentially consistent executions of the program and does not have to deal with any reordering optimizations. With this information, the optimizations enabled by the weak ordering model can be safely applied. In fact, this information enables more aggressive optimizations than those exploited by weak ordering,[5,12] and can also be used to efficiently port programs to all the other relaxed models.[1,3]

Figure 6 depicts the decision process for distinguishing memory operations. Correctness is not guaranteed if the programmer incorrectly distinguishes a race operation as data. However, an operation may be conservatively distinguished as a synchronization operation if the programmer is not sure whether the operation is involved in a race. This *don't-know* option is important because it allows a programmer to trivially ensure correctness by conservatively identifying all operations as synchronization. Of course, this forgoes any performance gains but potentially allows a faster path to an initial working program. The don't-know option also lets the programmer incrementally tune performance: The programmer can provide accurate information for memory operations in performance-critical areas of the program and conservative information for other areas.

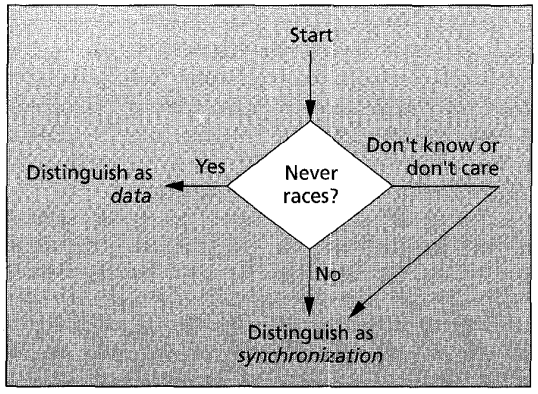

Figure 6. Deciding how to distinguish a memory operation.

Distinguishing memory operations

To provide the system with information on memory operations, we need a mechanism to distinguish operations at the language level. We also need a mechanism to pass this information to the hardware level.

LANGUAGE LEVEL. Here we consider languages that have explicit parallel constructs. The mechanism for conveying information about memory operations depends on how the language supports parallelism. Language support for parallelism may range from high-level parallelism constructs (such as `doall` loops) to low-level use of memory operations for achieving synchronization.

- A high-level `doall` loop implies that the parallel iterations of the loop do not access the same location if at least one of these accesses is a write. Thus, correct use of `doall` implicitly conveys that accesses across iterations are not involved in a race.
- A language may require that programmers use only low-level synchronization routines, such as those provided in a library, to eliminate races between other operations in the program. Again, correct use of such routines implies that only accesses within the synchronization library are involved in races. Of course, the compiler or library writers must ensure that appropriate information (synchronization or data) for operations used to implement the synchronization routines is appropriately conveyed to the lower levels of the system (for example, the hardware).
- At the lowest level, the programmer may be allowed to directly use any memory operation for synchronization purposes. For example, any location may be used as a flag variable. In this case, the programmer must explicitly convey information about operation types. One way to do this is to associate the information with static instructions at the program level. For example, special constructs may statically identify regions of code to be synchronization. Another option is to associate the synchronization attribute with a shared variable or address through, for example, type declarations. Or the language may provide a default mode that assumes, for example, that an operation is a data operation unless specified otherwise. Even though data operations are more common, making synchronization the default may make it simpler to bring up an initial working program and may decrease errors by requiring data operations (which are aggressively reordered) to be identified explicitly. We are not aware of any languages that provide appropriate mechanisms for conveying information at this lowest level. Mechanisms such as C's volatile type declaration lack the appropriate semantics.[3]

HARDWARE LEVEL. The information conveyed at the language level must ultimately be provided to the underlying hardware. Often the compiler is responsible for doing this.

Information about memory operations at this level may also be associated with either specific address ranges or static memory instructions. The former may be supported by distinguishing different virtual or physical pages. The latter may be supported through unused opcode bits (that is, multiple flavors of memory instructions) or unused address bits (that is, address shadowing) or by treating certain instructions (such as compare&swap) as synchronization by default.

Most commercial systems do not provide these mechanisms. Instead, this information must be transformed to explicit fence instructions supported at the hardware level. For example, to provide the semantics of synchronization operations in weak ordering on hardware that supports Alpha-like memory barriers, the compiler can precede and follow every synchronization operation with a memory barrier. Due to the widespread adoption of fence instructions, several languages also let programmers explicitly invoke them at the program level.

THERE IS STRONG EVIDENCE that relaxed memory consistency models provide better performance than sequential consistency models.[3,7,10,11] The increase in processor speeds

relative to memory and communication speeds will only increase the potential benefit from these models. In addition to gains in hardware performance, relaxed memory consistency models also play a key role in enabling compiler optimizations. For these reasons, many commercial architectures, such as the Digital Alpha, Sun Sparc, and IBM PowerPC, support relaxed consistency.

Unfortunately, relaxed memory consistency models increase programming complexity. Much of this complexity arises because many of the specifications presented in the literature expose the programmer to the low-level performance optimizations enabled by a model. Our previous work has addressed this issue by defining models using a higher level abstraction that provides the illusion of sequential consistency as long as the programmer provides correct program-level information about memory operations. Meanwhile, language standardization efforts such as High Performance Fortran have led to high-level memory models that are different from sequential consistency. In short, the question of which is the best memory consistency model is far from resolved. This question can be better resolved with a more active collaboration between language and hardware designers. ■

Acknowledgments

Most of this work was performed as part of our dissertations at the University of Wisconsin, Madison, and Stanford University. We thank our respective advisors, Mark Hill and Anoop Gupta and John Hennessy, for their direction. We especially thank Mark Hill for suggesting the need for this article and for encouraging us to write it.

We thank Sandhya Dwarkadas, Anoop Gupta, John Hennessy, Mark Hill, Yuan Yu, and Willy Zwaenepoel for their valuable comments. We also thank Tony Brewer, Andreas Nowatzyk, Steve Scott, and Wolf-Dietrich Weber for information on products developed by Hewlett-Packard, Sun Microsystems, Cray Research, and HaL Computer Systems. Finally, we thank the anonymous referees for their valuable comments and guest editor Per Stenström for his support. At Wisconsin, Sarita Adve was partly supported by an IBM graduate fellowship. At Rice, this work was partly supported by the US National Science Foundation under grants CCR-9502500 and CCR-9410457 and by the Texas Advanced Technology Program under grant 003604016. At Stanford, Kourosh Gharachorloo was supported by DARPA contract N00039-91-C-0138 and partly supported by a fellowship from Texas Instruments.

References

1. S.V. Adve, *Designing Memory Consistency Models for Shared Memory Multiprocessors*, PhD thesis, Tech. Report 1198, CS Department, Univ. of Wisconsin, Madison, 1993.
2. K. Gharachorloo et al., "Specifying System Requirements for Memory Consistency Models," Tech. Report CSL-TR-93-594, Stanford Univ., 1993.
3. K. Gharachorloo, *Memory Consistency Models for Shared Memory Multiprocessors*, PhD thesis, Tech. Report CSL-TR-95-685, Stanford Univ., 1995.
4. L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Computers*, Sept. 1979, pp. 690-691.
5. K. Gharachorloo et al., "Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 15-26.
6. K. Gharachorloo, A. Gupta, and J.L. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," *Proc. Int'l Conf. Parallel Processing*, The Pennsylvania State Univ. Press, University Park, Penn., 1991, pp. 355-364.
7. V.S. Pai et al., "An Evaluation of Memory Consistency Models for Shared Memory Systems with ILP Processors," *Proc. 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1996, pp. 12-23
8. D. Shasha and M. Snir, "Efficient Correct Execution of Parallel Programs That Share Memory," *ACM Trans. Programming Languages and Systems*, Apr. 1988, pp. 282-312.
9. A. Krishnamurthy and K. Yelick, "Optimizing Parallel SPMD Programs," in *Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, 1994.
10. K. Gharachorloo, A. Gupta, and J.L. Hennessy, "Performance Evaluation of Memory Consistency Models for Shared memory Multiprocessors," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1991, pp. 245-257.
11. R.N. Zucker and J.-L. Baer, "A Performance Study of Memory Consistency Models," *Proc. 19th Ann. Int'l Symp. on Computer Architecture*, ACM Press, New York, 1992, pp. 2-12.
12. S.V. Adve and M.D. Hill, "Weak Ordering—A New Definition," *Proc. 17th Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 2-14.

Sarita V. Adve *is an assistant professor in the Electrical and Computer Engineering Department at Rice University. Her research interests are parallel computer hardware and software, including the interaction between instruction-level and thread-level parallelism, architectural techniques to aid and exploit parallelizing compilers, and simulation methodology for parallel architectures. Adve received a BTech in electrical engineering from the Indian Institute of Technology, Bombay, and an MS and a PhD in computer science from the University of Wisconsin, Madison.*

Kourosh Gharachorloo *is a research scientist in the Western Research Laboratory at Digital Equipment Corporation. His research interests are parallel computer architecture and software, including hardware and software distributed shared memory systems and the study of commercial applications. Before joining Digital, he was a key contributor in the Stanford Dash and Flash projects. Gharachorloo received a BS in electrical engineering, a BA in economics, an MS in electrical engineering, and a PhD in electrical engineering and computer science, all from Stanford University.*

Contact Adve at Dept. of Electrical and Computer Engineering, MS 366 Rice University, Houston, TX 77005; sarita@rice.edu. Contact Gharachorloo at Western Research Laboratory, Digital Equipment Corp., 250 University Ave., Palo Alto, CA 94301-1616; kourosh@pa.dec.com.