DOI:10.1145/1787234.1787255

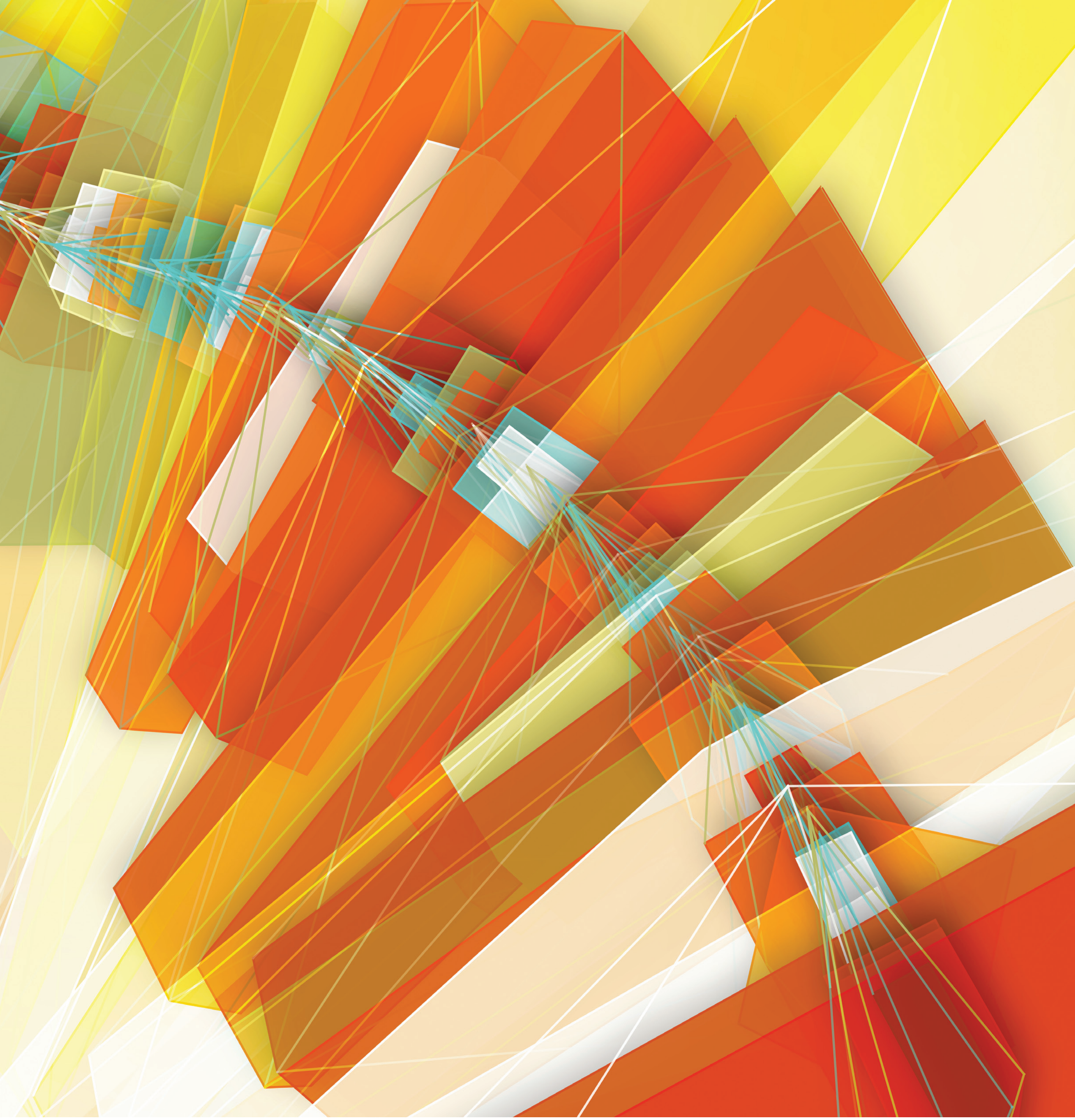
Solving the memory model problem will require an ambitious and cross-disciplinary research direction.

BY SARITA V. ADVE AND HANS-J. BOEHM

Memory Models: A Case for Rethinking Parallel Languages and Hardware

MOST PARALLEL PROGRAMS today are written using threads and shared variables. Although there is no consensus on parallel programming models, there are a number of reasons why threads remain popular. Threads were already widely supported by mainstream operating systems well before the dominance of multicore, largely because they are also useful for other purposes. Direct hardware support for shared-memory potentially provides a performance advantage; for example, by implicitly sharing read-mostly data without the space overhead of complete replication. The ability to pass memory references among threads makes it easier to share complex data structures. Finally, shared-memory makes it far easier to selectively parallelize application hot spots without complete redesign of data structures.

ILLUSRATION BY GWEN VANHEE



The memory model, or memory consistency model, is at the heart of the concurrency semantics of a shared-memory program or system. It defines the set of values that a read in a program is allowed to return, thereby defining the basic semantics of shared variables. It answers questions such as: Is there enough synchronization to ensure a thread's write will occur before another's read? Can two threads write to adjacent fields in a memory location at the same time? Must the final value of a location always be one of those written to it?

The memory model defines an interface between a program and any hardware or software that may transform that program (for example, the compiler, the virtual machine, or any dynamic optimizer). It is not possible to meaningfully reason about either a program (written in a high-level, byte code, assembly, or machine language) or any part of the language implementation (including hardware) without an unambiguous memory model.

A complex memory model makes parallel programs difficult to write, and parallel programming difficult to teach. An overly constraining one may limit hardware and compiler optimization, severely reducing performance. Since it is an interface property, the memory model decision has a long-lasting impact, affecting portability and maintainability of programs. Thus, a hardware architecture committed to a strong memory model cannot later forsake it for a weaker model without breaking binary compatibility, and a new compiler release with a weaker memory model may require rewriting source code. Finally, memory-model-related decisions for a single component must consider implications for the rest of the system. A processor vendor cannot guarantee a strong hardware model if the memory system designer provides a weaker model; a strong hardware model is not very useful to programmers using languages and compilers that provide only a weak guarantee.

Nonetheless, the central role of the memory model has often been downplayed. This is partly because formally specifying a model that balances all desirable properties of programmability, performance, and portability has proven surprisingly complex. At the same time, informal, machine-specific descriptions proved mostly adequate in an era where parallel programming was the domain of experts and achieving the highest possible performance trumped programmability or portability arguments.

In the late 1980s and 1990s, the area received attention primarily in the hardware community, which explored many approaches, with little consensus.[2] Commercial hardware memory model descriptions varied greatly in precision, including cases of complete omission of the topic and some reflecting vendors' reluctance to make commitments with unclear future implications. Although the memory model affects the meaning of every load instruction in every multithreaded application, it is still sometimes relegated to the "systems programming" section of the architecture manual.

Part of the challenge for hardware architects was the lack of clear memory models at the programming language level. It was unclear what programmers expected hardware to do. Although hardware researchers proposed approaches to bridge this gap,[3] widespread adoption required consensus from the software community. Before 2000, there were a few programming environments that addressed the issue with relative clarity,[40] but the most widely used environments had unclear and questionable specifications.[9,32] Even when specifications were relatively clear, they were often violated to obtain sufficient performance,[9] tended to be misunderstood even by experts, and were difficult to teach.

Since 2000, we have been involved in efforts to cleanly specify programming-language-level memory models, first for Java and then C++, with efforts now under way to adopt similar models for C and other languages. In the process, we had to address issues created by hardware that had evolved without the benefit of a clear programming model. This often made it difficult to reconcile the need for a simple and usable programming model with that for adequate performance on existing hardware.

Today, these languages and most hardware vendors have published (or plan to publish) compatible memory model specifications. Although this convergence is a dramatic improvement over the past, it has exposed fundamental shortcomings in our parallel languages and their interplay with hardware. After decades of research, it is still unacceptably difficult to describe what value a load can return without compromising modern safety guarantees or implementation methodologies. To us, this experience has made it clear that solving the memory model problem will require a significantly new and cross-disciplinary research direction for parallel computing languages, hardware, and environments as a whole.

This article discusses the path that led to the current convergence in memory models, the fundamental shortcomings it exposed, and the implications for future research. The central role of the memory model in parallel computing makes this article relevant to many computer science subdisciplines, including algorithms, applications, languages, compilers, formal methods, software engineering, virtual machines, runtime systems, and hardware. For practitioners and educators, we provide a succinct summary of the state of the art of this often-ignored and poorly understood topic. For researchers, we outline an ambitious, cross-disciplinary agenda toward resolving a fundamental problem in parallel computing today—what value can a shared variable have and how to implement it?

» key insights

- **Memory models, which describe the semantics of shared variables, are crucial to both correct multithreaded applications and the entire underlying implementation stack. It is difficult to teach multithreaded programming without clarity on memory models.**
- **After much prior confusion, major programming languages are converging on a model that guarantees simple interleaving-based semantics for "data-race-free" programs and most hardware vendors have committed to support this model.**
- **This process has exposed fundamental shortcomings in our languages and a hardware-software mismatch. Semantics for programs that contain data races seem fundamentally difficult, but are necessary for concurrency safety and debuggability. We call upon software and hardware communities to develop languages and systems that *enforce* data-race-freedom, and co-designed hardware that exploits and supports such semantics.**

Sequential Consistency

A natural view of the execution of a multithreaded program operating on shared variables is as follows. Each step in the execution consists of choosing one of the threads to execute, and then performing the next step in that thread's execution (as dictated by the thread's program text, or *program order*). This process is repeated until the program as a whole terminates. Effectively, the execution can be viewed as taking all the steps executed by each thread, and interleaving them in some way. Whenever an object (that is, variable, field, or array element) is accessed, the last value stored to the object by this interleaved sequence is retrieved.

For example, consider Figure 1, which gives the core of Dekker's mutual exclusion algorithm. The program can be executed by interleaving the steps from the two threads in many ways. Formally, each of these interleavings is a total order over all the steps performed by all the threads, consistent with the program order of each thread. Each access to a shared variable "sees" the last prior value stored to that variable in the interleaving.

Figure 2 gives three possible executions that together illustrate all possible final values of the non-shared variables r1 and r2. Although many other interleavings are also possible, it is not possible that both r1 and r2 are 0 at the end of an execution; any execution must start with the first statement of one of the two threads, and the variable assigned there will later be read as one.

Following Lamport,[26] an execution that can be understood as such an interleaving is referred to as *sequentially consistent*. Sequential consistency gives us the simplest possible meaning for shared variables, but suffers from several related flaws.

First, sequential consistency can be expensive to implement. For Figure 1, a compiler might, for example, reorder the two independent assignments in the red thread, since scheduling loads early tends to hide the load latency. In addition, modern processors almost always use a store buffer to avoid waiting for stores to complete, also effectively reordering instructions in each thread. Both the compiler and hardware optimization make an outcome of r1 == 0 and r2 == 0 possible, and hence may result in a non-sequentially consistent execution. Overall, reordering any pair of accesses, reading values from write buffers, register promotion, common subexpression elimination, redundant read elimination, and many other hardware and compiler optimizations commonly used in uniprocessors can potentially violate sequential consistency.[2]

There is some work on compiler analysis to determine when such transformations are unsafe (for example, Shasha and Snir[37]). Compilers, however, often have little information about sharing between threads, making it expensive to forego the optimizations, since we would have to forego them everywhere. There is also much work on speculatively performing these optimizations in hardware, with rollback on detection of an actual sequential consistency violation (for example, Ceze et al.[14] and Gharachorloo et al.[21]). However, these ideas are tied to specific implementation techniques (for example, aggressive speculation support), and vendors have generally been unwilling to commit to those for the long term (especially, given non-sequentially consistent compilers). Thus, most hardware and compilers today do not provide sequential consistency.

Second, while sequential consistency may seem to be the simplest model, it is not sufficiently simple and a much less useful programming model than commonly imagined. For example, it only makes sense to reason about interleaving steps if we know what those steps are. In this case, they are typically individual memory accesses, a very low-level notion. Consider two threads concurrently assigning values of 100,000 and 60,000 to the shared variable X on a machine that accesses memory 16 bits at a time. The final value of X in a "sequentially consistent" execution may be 125,536 if the assignment of 60,000 occurred between the bottom and top half of the assignment of 100,000. At a somewhat higher level, this implies the meaning of even simple library operations depends on the granularity at which the library carries out those operations.

More generally, programmers do not reason about correctness of parallel code in terms of interleavings of individual memory accesses, and sequential consistency does not prevent common sources of concurrency bugs arising from simultaneous access to the same shared data (for example, data races). Even with sequential consistency, such simultaneous accesses can remain dangerous, and should be avoided, or at least explicitly highlighted. Relying on sequential consistency without such highlighting both obscures the code, and greatly complicates the implementation's job.

Data-Race-Free

We can avoid both of the problems mentioned here by observing that:

- The problematic transformations (for example, reordering accesses to unrelated variables in Figure 1) never change the meaning of single-threaded programs, but do affect multithreaded programs (for example, by allowing both r1 and r2 to be 0 in Figure 1).
- These transformations are detectable only by code that allows two threads to access the same data simultaneously in conflicting ways; for example, one thread writes the data and another reads it.

Programming languages generally already provide *synchronization* mechanisms, such as locks, or possibly transactional memory, for limiting simultaneous access to variables by different threads. If we require that these be used correctly, and guarantee sequential consistency only if no undesirable concurrent accesses are present, we avoid the above issues.

We can make this more precise as follows. We assume the language allows distinguishing between synchronization and ordinary (non-synchronization or data) operations (see below). We say that two memory operations *conflict* if they access the same memory location (for example, variable or array element), and at least one is a write.

We say that a program (on a particular input) allows a *data race* if it has a sequentially consistent execution (that is, a program-ordered interleaving of operations of the individual threads) in which two conflicting ordinary operations execute "simultaneously." For our purposes, two operations execute "simultaneously" if they occur next to each other in the interleaving and correspond to different threads. Since these operations occur adjacently in the interleaving, we know that they could equally well have occurred in the opposite order; there are no intervening operations to enforce the order.

To ensure that two conflicting ordinary operations do not happen simultaneously, they must be ordered by intervening synchronization operations. For example, one thread must release a lock after accessing a shared variable, and

Figure 1. Core of Dekker's algorithm. Can r1 = r2 = 0?

Initially X=Y=0

Red Thread	Blue Thread
X = 1;	Y = 1;
r1 = Y;	r2 = X;

Figure 2. Some executions for Figure 1.

Execution 1	Execution 2	Execution 3
X = 1;	Y = 1;	X = 1;
r1 = Y;	r2 = X;	Y = 1;
Y = 1;	X = 1;	r1 = Y;
r2 = X;	r1 = Y;	r2 = X;
// r1 == 0	// r1 == 1	// r1 == 1
// r2 == 1	// r2 == 0	// r2 == 1

the other thread must acquire the lock before its access. Thus, it is also possible to define data races as conflicting accesses not ordered by synchronization, as is done in Java. These definitions are essentially equivalent.[1,12]

A program that does not allow a data race is said to be *data-race-free*. The data-race-free model guarantees sequential consistency only for data-race-free programs.[1,3] For programs that allow data races, the model does not provide any guarantees.

The restriction on data races is not onerous. In addition to locks for avoiding data races, modern programming languages generally also provide a mechanism, such as Java's `volatile` variables, for declaring that certain variables or fields are to be used for synchronization between threads. Conflicting accesses to such variables may occur simultaneously—since they are explicitly identified as *synchronization* (vs. ordinary), they do not create a data race.

To write Figure 1 correctly under data-race-free, we need simply identify the shared variables X and Y as synchronization variables. This would require the implementation to do whatever is necessary to ensure sequential consistency, in spite of those simultaneous accesses. It would also obligate the implementation to ensure that these synchronization accesses are performed indivisibly; if a 32-bit integer is used for synchronization purposes, it should not be visibly accessed as two 16-bit halves.

This "sequential consistency for data-race-free programs" approach alleviates the problems discussed with pure sequential consistency. Most important hardware and compiler optimizations continue to be allowed for ordinary accesses—care must be taken primarily at the explicitly identified (infrequent) synchronization accesses since these are the only ones through which such optimizations and granularity considerations affect program outcome. Further, synchronization-free sections of the code appear to execute atomically and the requirement to explicitly identify concurrent accesses makes it easier for humans and compilers to understand the code. (For more detail, see our technical report.[11])

Data-race-free does not give the implementation a blanket license to perform single-threaded program optimizations. In particular, optimizations that amount to copying a shared variable to itself; such as, introducing the assignment `x = x`, where `x` might not otherwise have been written, generally remain illegal. These are commonly performed in certain contexts,[9] but should not be.

Although data-race-free was formally proposed in 1990,[3] it did not see widespread adoption as a formal model in industry until recently. We next describe the evolution of industry models to a convergent path centered around data-race-free, the emergent shortcomings of data-race-free, and their implications for the future.

Industry Practice and Evolution

Hardware memory models. Most hardware supports relaxed models that are weaker than sequential consistency. These models take an implementation- or performance-centric view, where the desirable hardware optimizations drive the model specification.[1,2,20] Typical driving optimizations relax the program order requirement of sequential consistency. For example, Sparc's TSO guarantees that a thread's memory accesses will become visible to other threads in program order, except for the case of a write followed by a read. Such models additionally provide fence instructions to enable programmers to explicitly impose orderings that are otherwise not guaranteed; for example, TSO programmers may insert a fence between a thread's write and read to ensure the execution preserves that order.

Such a program-orderings + fences style of specification is simple, but many subtleties make it inadequate.[1,2] First, this style implies that a write is an atomic or indivisible operation that becomes visible to all threads at once. As Figure 3 illustrates, however, hardware may make writes visible to different threads at different times through write buffers and shared caches. Incorporating such optimizations increases the complexity of the memory model specification. Thus, the full TSO specification, which incorporates one of the simplest atomicity optimizations, is much more involved than the simple description here. PowerPC implements more aggressive forms of the optimization, with a specification that is complex and difficult to interpret even for experts. The x86 documentation from both AMD and Intel was ambiguous on this issue; recent updates now clarify the intent, but remain informal.

Second, in well-written software, a thread usually relies on synchronization interactions to reason about the ordering or visibility of memory accesses on other threads. Thus, it is usually overkill to require that two program-ordered accesses always become visible to *all* threads in the same order or a write appears atomic to *all* threads regardless of the synchronization among the threads. Instead, it is sufficient to preserve ordering and atomicity only among mutually synchronizing threads. Some hardware implementations attempt to exploit this insight, albeit often through ad hoc techniques, thereby further complicating the memory model.

Third, modern processors perform various forms of speculation (for example, on branches and addresses) which can result in subtle and complex interactions with data and control dependences, as illustrated in Figure 4.[1,29]

Figure 3. Hardware may not execute atomic or indivisible writes.

Assume a fence imposes program order. Assume core 3's and core 4's caches have X and Y. The two writes generate invalidations for these caches. These could reach the caches in a different order, giving the result shown and a deduction that X's update occurs both before and after Y's.

Initially X = Y = 0

Core 1	Core 2	Core 3	Core 4
X = 1;	Y = 1;	r1 = X;	r3 = Y;
		fence;	*fence;*
		r2 = Y;	r4 = X;

Can r1 = 1, r2 = 0, r3 =1, r4 = 0, violating write atomicity?

Incorporating these considerations in a precise way adds another source of complexity to program-order + fence style specifications. As we discuss later, precise formalization of data and control dependences is a fundamental obstacle to providing clean high-level memory model specifications today.

In summary, hardware memory model specifications have often been incomplete, excessively complex, and/or ambiguous enough to be misinterpreted even by experts. Further, since hardware models have largely been driven by hardware optimizations, they have often not been well-matched to software requirements, resulting in incorrect code or unnecessary loss in performance, as discussed later.

High-level language memory models. Ada was perhaps the first widely used high-level programming language to provide first-class support for shared-memory parallel programming. Although Ada's approach to thread synchronization was initially quite different from both that of the earlier Mesa design and most later language designs, it was remarkably advanced in its treatment of memory semantics.[40] It used a style similar to data-race-free, requiring legal programs to be well-synchronized; however, it did not fully formalize the notion of well-synchronized and left uncertain the behavior of such programs.

Subsequently, until the introduction of Java, mainstream programming languages did not provide first-class support for threads, and shared-memory programming was mostly enabled through libraries and APIs such as Posix threads and OpenMP. Previous work describes why the approach of an add-on threads library is not entirely satisfactory.[9] Without a real definition of the programming language in the context of threads, it is unclear what compiler transformations are legal, and hence what the programmer is allowed to assume. Nevertheless, the Posix threads specification indicates a model similar to data-race-free, although there are several inconsistent aspects, with widely varying interpretations even among experts participating in standards committee discussions. The OpenMP model is also unclear and largely based on a flush instruction that is analogous to fence instructions in hardware models, with related shortcomings.

Hardware memory model specifications have often been incomplete, excessively complex, and/or ambiguous enough to be misinterpreted even by experts.

The Java memory model. Java provided first-class support for threads with a chapter specifically devoted to its memory model. Pugh showed that this model was difficult to interpret and badly broken—common compiler optimizations were prohibited and in many cases the model gave ambiguous or unexpected behavior.[32] In 2000, Sun appointed an expert group to revise the model through the Java community process.[33] The effort was coordinated through an open mailing list that attracted a variety of participants, representing hardware and software and researchers and practitioners.

It was quickly decided that the Java memory model must provide sequential consistency for data-race-free programs, where volatile accesses (and locks from synchronized methods and monitors) were deemed synchronization.

However, data-race-free is inadequate for Java. Since Java is meant to be a safe and secure language, it cannot allow arbitrary behavior for data races. Specifically, Java must support untrusted code running as part of a trusted application and hence must limit damage done by a data race in the untrusted code. Unfortunately, the notions of safety, security, and "limited damage" in a multithreaded context were not clearly defined. The challenge with defining the Java model was to formalize these notions in a way that minimally affected system flexibility.

Figure 4(b) illustrates these issues. The program has a data race and is buggy. However, Java cannot allow its reads to return values out-of-thin-air (for example, 42) since this could clearly compromise safety and security. It would, for example, make it impossible to guarantee that similar untrusted code cannot return a password that it should not have access to. Such a scenario appears to violate any reasonable causality expectation and no current processor produces it. Nevertheless, the memory model must formally prohibit such behavior so that future speculative processors also avoid it.

Prohibiting such causality violations in a way that does not also prohibit other desired optimizations turned out to be surprisingly difficult. Figure 5 illustrates an example that also appears to violate causality, but is allowed by the

Figure 4. Subtleties with (a) control and (b) data dependences.

It is feasible for core 1 to speculate that its read of X will see 1 and speculatively write Y. Core 2 similarly writes X. Both reads now return 1, creating a "self-fulfilling" speculation or a "causality loop." Within a single core, no control dependences are violated since the speculation appears correct; however, most programmers will not expect such an outcome (the code is in fact data-race-free since no sequentially consistent execution contains a data race). Part (b) shows an analogous causal loop with data dependences. Core 1 may speculate X is 42 (for example, using value prediction based on previous store values) and (speculatively) write 42 into Y. Core 2 reads this and writes 42 into X, thereby proving the speculation right and creating a causal loop that generates a value (42) out-of-thin-air. Fortunately, no processor today behaves this way, but the memory model specification needs to reflect this property.

Initially X=Y=0

Core 1	Core 2

r1 = X;
if (r1 == 1)
  Y = 1;
	r2 = Y;
	if (Y == 1)
	  X = 1;

Is r1 = r2 = 1 allowed?	

(a)

Initially X=Y=0

Core 1	Core 2
r1 = X; Y = r1;	r2 = Y; X = r2;
Is r1 = r2 = 42 allowed?	

(b)

Figure 5. Redundant read elimination must be allowed.

For thread 1, the compiler could eliminate the redundant read of X, replacing r2=X with r2=r1. This allows deducing that r1==r2 is always true, making the write of Y unconditional. Then the compiler may move the write to before the read of X since no dependence is violated. Sequential consistency would allow both the reads of X and Y to return 1 in the new but not the original code. This outcome for the original code appears to violate causality since it seems to require a self-justifying speculative write of Y. It must, however, be allowed if compilers are to perform the common optimization of redundant read elimination.

Initially X=Y=0

Original Code

Thread 1	Thread 2

r1 = X;
r2 = X;
if (r1 == r2)
  Y = 1;
	r3 = Y;
	X = r3;

After compiler transformation

Thread 1	Thread 2

Y = 1;
r1 = X;
r2 = r1;
if (true);
	r3 = Y;
	X = r3;

Is r1 = r2 = r3 = 1 allowed?	

common compiler optimization of redundant read elimination.[29] After many proposals and five years of spirited debate, the current model was approved as the best compromise. This model allows the outcome of Figure 5, but not that of Figure 4(b). Unfortunately, this model is very complex, was known to have some surprising behaviors, and has recently been shown to have a bug. We provide intuition for the model below and refer the reader to Manson et al.[26] for a full description.

Common to both Figure 4(b) and Figure 5 are writes that are executed earlier than they would be with sequential consistency. The examples differ in that for the speculative write in the latter (Y=1), there is *some* sequentially consistent execution where it is executed (the execution where both reads of X return 0). For Figure 4(b), there is no sequentially consistent execution where Y=42 could occur. This notion of whether a speculative write *could* occur in some well-behaved execution is the basis of causality in the Java model, and the definition of *well-behaved* is the key source of complexity.

The Java model tests for the legality of an execution by "committing" one or more of its memory accesses at a time—legality requires all accesses to commit (in any order). Committing a write early (before its turn in program order) requires it to occur in a well-behaved execution where (informally) the already committed accesses have similar synchronization and data race relationships in all previously used well-behaved executions, and the to-be committed write is not dependent on a read that returns its value from a data race. These conditions ensure that a future data race will never be used to justify a speculative write that could then later justify that future data race.

A key reason for the complexity in the Java model is that it is not operational—an access in the future can determine whether the current access is legal. Further, many possible future executions must be examined to determine this legality. The choice of future (well-behaved) executions also gives some surprising results. In particular, as discussed in Manson[29] if the code of one thread is "inlined" in (concatenated with) another thread, then the inlined code can produce more behaviors than the original. Thus, thread inlining is generally illegal under the Java model (even if there are no synchronization- and deadlock-related considerations). In practice, the prohibited optimizations are difficult to implement and this is not a significant performance limitation. The behavior, however, is nonintuitive, with other implications—it occurs because some data races in the original code may no longer be data races in the inlined code. This means that when determining whether to commit a write early, a read in a well-behaved execution has more choices to return values than before (since there are fewer data races), resulting in new behaviors.

More generally, increasing synchronization in the Java model can actually result in new behaviors, even though more synchronization conventionally constrains possible executions. Recently, it has been shown that, for similar reasons, adding seemingly irrelevant reads or removing redundant reads sometimes can also add new behaviors, and that all these properties have more serious implications than previously thought.[36] In particular, some optimizations that were intended to be allowed by the Java model are in fact prohibited by the current specification.

It is unclear if current hardware or JVMs implement the problematic optimizations noted here and therefore violate the current Java model. Certainly the current specification is much improved over the original. Regardless, the situation is still far from satisfactory. First, clearly, the current specification does not meet its desired intent of having certain common optimizing transformations preserve program meaning. Second, its inherent complexity and the new observations make it difficult to prove the correctness of any real system. Third, the specification methodology is inherently fragile—small changes usually result in hard-to-detect unintended consequences.

The Java model was largely guided by an emergent set of test cases,[33] based on informal code transformations that were or were not deemed desirable. While it may be possible to fix the Java model, it seems undesirable that our specification of multithreaded program behavior would rest on such a complex and fragile foundation. Instead, the section entitled "Implications for Languages" advocates a fundamental rethinking of our approach.

The C++ memory model. The situation in C++ was significantly different from Java. The language itself provided no support for threads. Nonetheless, they were already in widespread use, typically with the addition of a library-based threads implementation, such as pthreads[22] or the corresponding Microsoft Windows facilities. Unfortunately the relevant specifications, for example the combination of the C or C++ standard with the Posix standard, left significant uncertainties about the rules governing shared variables.[9] This made it unclear to compiler writers precisely what they needed to implement, resulted in very occasional failures for which it was difficult to assign blame to any specific piece of the implementation and, most importantly, made it difficult to teach parallel programming since even the experts were unable to agree on some of the basic rules, such as whether Figure 4(a) constitutes a data race. (Correct answer: No.)

Motivated by these observations, we began an effort in 2005 to develop a proper memory model for C++. The resulting effort eventually expanded to include the definition of `atomic` (synchronization, analogous to Java `volatile`) operations, and the threads API itself. It is part of the current Committee Draft[24] for the next C++ revision. The next C standard is expected to contain a very similar memory model, with very similar `atomic` operations.

This development took place in the face of increasing doubt that a Java-like memory model relying on sequential consistency for data-race-free programs was efficiently implementable on mainstream architectures, at least given the specifications available at the time. Largely as a result, much of the early discussion focused on the tension between the following two observations, both of which we still believe to be correct given existing hardware:

- A programming language model weaker than data-race-free is probably unusable by a large fraction of the programming community. Earlier work[10] points out, for example, that even thread library implementors often get confused when it comes to dealing explicitly with memory ordering issues. Substantial effort was invested in attempts to develop weaker, but comparably simple and usable models. We do not feel these were successful.
- On some architectures, notably on some PowerPC implementations, data-race-free involves substantial implementation cost. (In light of modern (2009) specifications, the cost on others, notably x86, is modest, and limited largely to `atomic` (C++) or `volatile` (Java) store operations.)

This resulted in a compromise memory model that supports data-race-free for nearly all of the language. However, `atomic` data types also provide low-level operations with explicit memory ordering constraints that blatantly violate sequential consistency, even in the absence of data races. The low-level operations are easily identified and can be easily avoided by non-expert programmers. (They require an explicit `memory_order_` argument.) But they do give expert programmers a way to write very carefully crafted, but portable, synchronization code that approaches the performance of assembly code.

Since C++ does not support sandboxed code execution, the C++ draft standard can and does leave the semantics of a program with data races completely undefined, effectively making it erroneous to write such programs. As we point out in Boehm and Adve[12] this has a number of (mostly) performance-related advantages, and better reflects existing compiler implementations.

In addition to the issues raised in the "Lessons Learned" section, it should be noted that this really only pushes Java's issues with causality into a much smaller and darker corner of the specification; exactly the same issues arise if we rewrite Figure 4(b) with C++ `atomic` variables and use low-level `memory_order_relaxed` operations. Our current solution to this problem is simpler, but as inelegant as the Java one. Unlike Java, it affects a small number of fairly esoteric library calls, not all memory accesses.

As with the Java model, we feel that although this solution involves compromises, it is an important step forward. It clearly establishes data-race-free as the core guarantee that every programmer should understand. It defines precisely what constitutes a data race. It finally resolves simple questions such as: If x.a and x.b are assigned simultaneously, is that a data race? (No, unless they are part of the same contiguous sequence of bit-fields.) By doing so, it clearly identifies shortcomings of existing compilers that we can now begin to remedy.

Reconciling language and hardware models. Throughout this process, it repeatedly became clear that current hardware models and supporting fence instructions are often at best a marginal match for programming language memory models, particularly in the presence of Java `volatile` fields or C++ `atomic` objects. It is always possible to implement such synchronization variables by mapping each one to a lock, and acquiring and releasing the corresponding lock around all accesses. However, this typically adds an overhead of hundreds of cycles to each access, particularly since the lock accesses are likely to result in coherence cache misses, even when only read accesses are involved.

`Volatile` and `atomic` variables are typically used to avoid locks for exactly these reasons. A typical use is a flag that indicates a read-only data structure has been lazily initialized. Since the initialization has to happen only once, nearly all accesses simply read the `atomic`/`volatile` flag and avoid lock acquisitions. Acquiring a lock to access the flag defeats the purpose.

On hardware that relaxes write atomicity (see Figure 3), however, it is often unclear that more efficient mappings (than the use of locks) are possible; even the fully fenced implementation may not be sequentially consistent. Even on other hardware, there are apparent mismatches, most probably caused by the lack of a well-understood programming language model when the hardware was designed. On x86, it is *almost* sufficient to map synchronization loads and stores directly to ordinary load and store instructions. The hardware provides sufficient guarantees to ensure that ordinary memory operations are not *visibly* reordered with synchronization operations. However it fails to prevent reordering of a synchronization store followed by a synchronization load; thus this implementation does not prevent the incorrect outcome for Figure 1.

This may be addressed by translating a synchronization store to an ordinary store instruction followed by an expensive fence. The sole purpose of this fence is to prevent reordering of the synchronization store with a subsequent *synchronization* load. In practice, such a synchronization load is unlikely to follow closely enough (Dekker's algorithm is not commonly used) to really constrain the hardware. But the only available fence instruction constrains *all* memory reordering around it, including that involving ordinary data accesses, and thus overly constrains the hardware. A better solution would involve distinguishing between two flavors of loads and stores (ordinary and synchronization), roughly along the lines of Itanium's `ld.acq` and `st.rel`.[23] This, however, requires a change to the instruction set architecture, usually a difficult proposition.

We suspect the current situation makes the fence instruction more expensive than necessary, in turn motivating additional language-level complexity such as C++ low-level atomics or `lazySet()` in Java.

Lessons Learned

Data-race-free provides a simple and consistent model for threads and shared variables. We are convinced it is the best model today to target during initial software development. Unfortunately, its lack of any guarantees in the presence of data races and mismatch with current hardware implies three significant weaknesses:

Debugging. Accidental introduction of a data race results in "undefined behavior," which often takes the form of surprising results later during program execution, possibly long after the data race has resulted in corrupted data. Although the usual immediate result of a data race is that an unexpected, and perhaps incomplete value is read, or that an inconsistent value is written, we point out in prior work[12] that other results, such as wild branches, are also possible as a result of compiler optimizations that mistakenly assume the absence of data races. Since such races are difficult to reproduce, the root cause of such misbehavior is often difficult to identify, and such bugs may easily take weeks to track down. Many tools to aid such debugging (for example, CHESS[30] and RaceFuzzer[35]) also assume sequential consistency, somewhat limiting their utility.

Synchronization variable performance on current hardware. As discussed, ensuring sequential consistency in the presence of Java `volatile` or C++ `atomic` on current hardware can be expensive. As a result, both C++, and to a lesser extent Java, have had to provide less expensive alternatives that greatly complicate the model for experts trying to use them.

Untrusted code. There is no way to ensure data-race-freedom in untrusted code. Thus, this model is insufficient for languages like Java.

An unequivocal lesson from our experiences is that for programs with data races, it is very difficult to define semantics that are easy to understand and yet retain desired system flexibility. While the Java memory model came a long way, its complexity, and subsequent discoveries of its surprising behaviors, are far from satisfying. Unfortunately, we know of no alternative specification that is sufficiently simple to be considered practical. Second, rules to weaken the data-race-free guarantee to better match current hardware, as through C++ low-level atomics, are also more complex than we would like.

The only clear path to improvement here seems to be to eliminate the need for going beyond the data-race-free guarantee by:

- Eliminating the performance motivations for going beyond it, and
- Ensuring that data races are never actually executed at runtime, thus both avoiding the need to specify their behavior and greatly simplifying or eliminating the debugging issues associated with data races.

Unfortunately, these both take us to active research areas, with no clear off-the-shelf solutions. We discuss some possible approaches here.

Implications for Languages

In spite of the dramatic convergence in the debate on memory models, the state of the art imposes a difficult choice: a language that supposedly has strong safety and security properties, but no clear definition of what value a shared-memory read may return (the Java case), versus a language with clear semantics, but that requires abandoning security properties promised by languages such as Java (the C++ case). Unfortunately, modern software needs to be both parallel and secure, and requiring a choice between the two should not be acceptable.

A pessimistic view would be to abandon shared-memory altogether. However, the intrinsic advantages of a global address space are, at least anecdotally, supported by the widespread use of threads despite the inherent challenges. We believe the fault lies not in the global address space paradigm, but in the use of undisciplined or "wild shared-memory," permitted by current systems.

Data-race-free was a first attempt to formalize a shared-memory discipline via a memory model. It proved inadequate because the responsibility for following this discipline was left to the programmer. Further, data-race-free by itself is, arguably, insufficient as a discipline for writing correct, easily debuggable, and maintainable shared-memory code; for example, it does not completely eliminate atomicity violations or non-deterministic behavior.

Moving forward, we believe a critical research agenda to enable "parallelism for the masses" is to develop and promote *disciplined shared-memory models* that:

- are *simple* enough to be easily teachable to undergraduates; that is, minimally provide sequential consistency to programs that obey the required discipline;
- enable the *enforcement* of the discipline; that is, violations of the discipline should not have undefined or horrendously complex semantics, but should

be caught and returned back to the programmer as illegal;

- are general-purpose enough to *express* important parallel algorithms and patterns; and
- enable high and scalable *performance*.

Many previous programmer-productivity-driven efforts have sought to raise the level of abstraction with threads; for example, Cilk,[19] TBB,[25] OpenMP,[39] the recent HPCS languages,[28] other high-level libraries, frameworks, and APIs such as java.util.concurrent and the C++ boost libraries, as well as more domain-specific ones. While these solutions go a long way toward easing the pain of orchestrating parallelism, our memory-models driven argument is deeper—we argue that, at least so far, it is not possible to provide reasonable *semantics* for a language that allows data races, an arguably more fundamental problem. In fact, all of these examples either provide unclear models or suffer from the same limitations as C++/Java. These approaches, therefore, do not meet our *enforcement* requirement. Similarly, transactional memory provides a high-level mechanism for atomicity, but the memory model in the presence of non-transactional code faces the same issues as described here.[38]

At the heart of our agenda of disciplined models are the questions: What is the appropriate discipline? How to enforce it?

A near-term transition path is to continue with data-race-free and focus research on its enforcement. The ideal solution is for the language to eliminate data races by design (for example, Boyapati[13]); however, our semantics difficulties are avoided even with dynamic techniques (for example, Elmas et al.,[17] Flanagan and Freund,[18] or Lucia et al.[27]) that replace all data races with exceptions. (There are other dynamic data race detection techniques, primarily for debugging, but they do not guarantee complete accuracy, as required here.)

A longer-term direction concerns both the appropriate discipline and its enforcement. A fundamental challenge in debugging, testing, and reasoning about threaded programs arises from their inherent non-determinism—an execution may exhibit one of many possible interleavings of its memory accesses. In contrast, many applications written for performance have deterministic outcomes and can be expressed with deterministic algorithms. Writing such programs using a deterministic environment allows reasoning with sequential semantics (a memory model much simpler than sequential consistency with threads).

Data-race-free provides a simple and consistent model for threads and shared variables. We are convinced that it is the best model today to target during initial software development.

A valuable discipline, therefore, is to provide a guarantee of determinism by default; when non-determinism is inherently required, it should be requested explicitly and should not interfere with the deterministic guarantees for the remaining program.[7] There is much prior work in deterministic data parallel, functional, and actor languages. Our focus is on general-purpose efforts that continue use of widespread programming practices; for example, global address space, imperative languages, object-oriented programming, and complex, pointer-based data structures.

Language-based approaches with such goals include Jade[34] and the recent Deterministic Parallel Java (DPJ).[8] In particular, DPJ proposes a region-based type and effect system for deterministic-by-default semantics—"regions" name disjoint partitions of the heap and per-method *effect* annotations summarize which regions are read and written by each method. Coupled with a disciplined parallel control structure, the compiler can easily use the effect summaries to ensure that there are no unordered conflicting accesses and the program is deterministic. Recent results show that DPJ is applicable to a range of applications and complex data structures and provides performance comparable to threads code.[8]

There has also been much recent progress in runtime methods for determinism.[4,5,16,31]

Both language and runtime approaches have pros and cons and still require research before mainstream adoption. A language-based approach must establish that it is expressive enough and does not incur undue programmer burden. For the former, the new techniques are promising, but the jury is still out. For the latter, DPJ is attempting to alleviate the burden by using a familiar base language (currently Java) and providing semiautomatic tools to infer the required programmer annotations.[41] Further, language annotations such as DPJ's read/write effect summaries are valuable documentation in their own right—they promote lifetime benefits for modularity and maintainability, arguably compensating for upfront programmer effort. Finally, a static approach benefits from no overhead or surprises at runtime.

In contrast, the purely runtime approaches impose less burden on the programmer, but a disadvantage is that the overheads in some cases may still be too high. Further, inherently, a runtime approach does not provide the guarantees of a static approach before shipping and is susceptible to surprises in the field.

We are optimistic that the recent approaches have opened up many promising new avenues for disciplined shared-memory that can overcome the problems described here. It is likely that a final solution will consist of a judicious combination of language and runtime features, and will derive from a rich line of future research.

We believe that hardware that takes advantage of the emerging disciplined software programming models is likely to be more efficient than a software oblivious approach.

Implications for Hardware

As discussed earlier, current hardware memory models are an imperfect match for even current software (data-race-free) memory models. ISA changes to identify individual loads and stores as synchronization can alleviate some short-term problems. An established ISA, however, is difficult to change, especially when existing code works mostly adequately and there is not enough experience to document the benefits of the change.

Academic researchers have taken an alternate path that uses complex mechanisms (for example, Blundell et al.[6]) to speculatively remove the constraints imposed by fences, rolling back the speculation when it is detected that the constraints were actually needed. While these techniques have been shown to work well, they come at an implementation cost and do not directly confront the root of the problem of mismatched hardware/software views of concurrency semantics.

Taking a longer-term perspective, we believe a more fundamental solution to the problem will emerge with a co-designed approach, where future multicore hardware research evolves in concert with the software models research discussed in "Implications for Languages." The current state of hardware technology makes this a particularly opportune time to embark on such an agenda. Power and complexity constraints have led industry to bet that future single-chip performance increases will largely come from increasing numbers of cores. Today's hardware cache-coherent multicore designs, however, are optimized for few cores—power-efficient, performance scaling to several hundreds or a thousand cores without consideration of software requirements will be difficult.

We view this challenge as an opportunity to not only resolve the problems discussed in this article, but in doing so, we expect to build more effective hardware and software. First, we believe that hardware that *takes advantage of* the emerging disciplined software programming models is likely to be more efficient than a software-oblivious approach. This observation already underlies the work on relaxed hardware consistency models—we hope the difference this time around will be that the software and hardware models will evolve together rather than as retrofits for each other, providing more effective solutions. Second, hardware research to *support* the emerging disciplined software models is also likely to be critical. Hardware support can be used for efficient enforcement of the required discipline when static approaches fall short; for example, through directly detecting violations of the discipline and/or through effective strategies to sandbox untrusted code.

Along these lines, we have recently begun the DeNovo hardware project at Illinois[15] in concert with DPJ. We are exploiting DPJ-like region and effect annotations to design more power- and complexity-efficient, software-driven communication and coherence protocols and task scheduling mechanisms. We also plan to provide hardware and runtime support to deal with cases where DPJ's static information and analysis might fall short. As such co-designed models emerge, ultimately, we expect them to drive the future hardware-software interface including the ISA.

Conclusion

This article gives a perspective based on work collectively spanning approximately 30 years. We have been repeatedly surprised at how difficult it is to formalize the seemingly simple and fundamental property of "what value a read should return in a multithreaded program." Sequential consistency for data-race-free programs appears to be the best we can do at present, but it is insufficient. The inability to define reasonable semantics for programs with data races is not just a theoretical shortcoming, but a fundamental hole in the foundation of our languages and systems. It is well accepted that most shipped software has bugs and it is likely that much commercial multithreaded software has data races. Debugging tools and safe languages that seek to sandbox untrusted code must deal with such races, and must be given semantics that reasonable computer science graduates and developers can understand.

We believe it is time to rethink how we design our languages and systems. Minimally, the system, and preferably the language, must enforce the absence of data races. A longer term, potentially more rewarding strategy is to rethink higher-level disciplines that make it much easier to write parallel programs and that can be *enforced* by our languages and systems. We also believe some of the messiness of memory models today could have been averted with closer cooperation between hardware and software. As we move toward more disciplined programming models, there is also a new opportunity for a hardware/software co-designed approach that rethinks the hardware/software interface and the hardware implementations of all concurrency mechanisms. These views embody a rich research agenda that will need the involvement of many computer science sub-disciplines, including languages, compilers, formal methods, software engineering, algorithms, runtime systems, and hardware.

Acknowledgments

This article is deeply influenced by a collective 30 years of collaborations and discussions with more colleagues than can be named here. We would particularly like to acknowledge the contributions of Mark Hill for co-developing the data-race-free approach and other foundational work; Kourosh Gharachorloo for hardware models; Jeremy Manson and Bill Pugh for the Java model; Lawrence Crowl, Paul McKenney, Clark Nelson, and Herb Sutter, for the C++ model; and Vikram Adve, Rob Bocchino, and Marc Snir for ongoing work on disciplined programming models and their enforcement. We thank Doug Lea for continuous encouragement to push the envelope. Finally, we thank Vikram Adve, Rob Bocchino, Nick Carter, Lawrence Crowl, Mark Hill, Doug Lea, Jeremy Manson, Paul McKenney, Bratin Saha, and Rob Schreiber for comments on earlier drafts.

Sarita Adve is currently funded by Intel and Microsoft through the Illinois Universal Parallel Computing Research Center.

References

1. Adve, S.V. *Designing Memory Consistency Models for Shared-Memory Multiprocessors.* PhD thesis. University of Wisconsin-Madison, 1993.
2. Adve, S.V. and Gharachorloo, K. Shared memory consistency models: A tutorial. *IEEE Computer 29*, 12 (1996), 66–76.
3. Adve, S.V. and Hill, M.D. Weak ordering—A new definition. In *Proceedings of the 17th Intl. Symp. Computer Architecture*, 1990, 2–14.
4. Allen, M.D., Sridharan, S. and Sohi, G.S. Serialization sets: A dynamic dependence-based parallel execution model. In *Proceedings of the Symp. on Principles and Practice of Parallel Programming*, 2009.
5. Berger, E.D., Yang, T., Liu, T. and Novark, G. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2009.
6. Blundell, C, Martin, M.M.K. and Wenisch, T. Invisifence: Performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of the Intl. Symp. on Computer Architecture*, 2009.
7. Bocchino, R. et al. Parallel programming must be deterministic by default. In *Proceedings of the 1st Workshop on Hot Topics in Parallelism*, 2009.
8. Bocchino. R. et al. A type and effect system for Deterministic Parallel Java. In *Proceedings of the Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2009.
9. Boehm, H.-J.. Threads cannot be implemented as a library. In *Proceedings of the Conf. on Programming Language Design and Implementation*, 2005.
10. Boehm, H.-J. Reordering constraints for pthread-style locks. In *Proceedings of the 12th Symp. Principles and Practice of Parallel Programming*, 2007, 173–182.
11. Boehm, H.-J. Threads basics. July 2009; http://www.hpl.hp.com/personal/Hans_Boehm/threadsintro.html.
12. Boehm, H.-J. and Adve, S.V. Foundations of the C++ concurrency memory model. In *Proceedings of the Conf. on Programming Language Design and Implementation*, 2008, 68–78.
13. Boyapati, C., Lee, R., and Rinard, M. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.
14. Ceze, L et al. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of the Intl. Symp. on Computer Architecture*, 2007.
15. Choi, B. et al. DeNovo: Rethinking Hardware for Disciplined Parallelism. In *Proceedings of the 2nd Workshop on Hot Topics in Parallelism*, June 2010.
16. Devietti, J. et al. DMP: Deterministic shared memory processing. In *Proceedings of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (Mar. 2009), 85–96.
17. Elmas, T., Qadeer, S. and Tasiran, S. Goldilocks: A race and transaction-aware Java runtime. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2007, 245–255.
18. Flanagan, C. and Freund, S. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the Conf. on Programming Language Design and Implementation*, 2009.
19. Frigo, M., Leiserson, C.E. and Randall, K.H. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1998, 212–223.
20. Gharachorloo, K. Memory consistency models for shared-memory multiprocessors. Ph.D. thesis, 1996, Stanford University, Stanford, CA.
21. Gharachorloo, K., Gupta, A. and Hennessy, J. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the Intl. Conf. on Parallel Processing*, 1991, I355–I364.
22. IEEE and The Open Group. *IEEE Standard 1003.1-2001.* 2001.
23. Intel. *Intel Itanium Architecture: Software Developer's Manual*, Jan 2006.
24. ISO/IEC JTC1/SC22/WG21. ISO/IEC 14882, Programming languages - C++ (final committee draft) 2010; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf.
25. Reinders, J. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Parallelism.* O'Reilly, 2007.
26. Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9 (1979), 690–691.
27. Lucia, B. et al. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the International Symposium on Computer Architecture,* 2010.
28. Lusk, E. and Yelick, E. Languages for high-productivity computing: The DARPA HPCS language project. *Parallel Processing Letters 17*, 1 (2007) 89–102.
29. Manson, J., Pugh, W. and Adve, S.V. The Java memory model. In *Proceedings of the Symp. on Principles of Programming Languages*, 2005.
30. Musuvathi, M. and Qadeer, S. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2007, 446–455.
31. Olszewski, M., Ansel, J., and Amarasinghe, S. Kendo: Efficient deterministic multithreading in software. In *Proceedings Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. Mar. 2009.
32. Pugh, W. The Java memory model is fatally flawed. *Concurrency-Practice and Experience 12*, 6 (2000), 445–455.
33. Pugh, W. and the JSR 133 Expert Group. The Java memory model (July 2009); http://www.cs.umd.edu/~pugh/java/memoryModel/.
34. Rinard, M.C. and Lam, M.S. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems 20*, 3 (1998), 483–545.
35. Sen, K. Race directed random testing of concurrent programs. In *Conf. on Programming Language Design and Implementation*, 2008.
36. Sevcik, J. and Aspinall, D. On validity of program transformations in the Java memory model. In *Proceedings of the European Conference on Object-Oriented Programming, 2008*, 27–51.
37. Shasha, D and Snir, M. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems 10*, 2 (Apr. 1998), 282–312.
38. Shpeisman, T. et al. Enforcing isolation and ordering in STM. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2007.
39. The OpenMP ARB. OpenMP application programming interface: Version 3.0. May 2008; http://www.openmp.org/mp-documents/spec30.pdf.
40. United States Department of Defense. *Reference Manual for the Ada Programming Language: ANSI/MIL-STD-1815A-1983 Standard 1003.1-2001.* Springer, 1983.
41. Vakilian, M. et al. Inferring method effect summaries for nested heap regions. In *Proceedings of the 24th Intl. Conf. on Automated Software Engineering*, 2009.

Sarita V. Adve (sadve@illinois.edu) is a professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign.

Hans-J. Boehm (Hans.Boehm@hp.com) is a member of Hewlett-Packard's Exascale Computing Lab, Palo Alto, CA.