Hiding the Java Memory Model with Compilers

Jaejin Lee
Department of Computer Science & Engineering
Michigan State University
East Lansing, MI 48824
jlee@cse.msu.edu

Abstract

The Java memory model is very difficult for programmers to understand, and there are several ways of interpreting the memory model. In addition, like most programming languages that follow the shared memory parallel programming model, non-deterministic behaviors due to data races can also occur in Java concurrent programs. Data races and synchronization make it impossible to apply classical compiler optimization and analysis techniques directly to shared memory parallel programs because the classical methods do not account for updates to variables in threads other than the one being analyzed. The synergic effect of the non-intuitive Java memory model and the non-deterministic behavior makes it difficult for a programmer to write correct and efficient Java concurrent programs. We describe a compiler technique that hides an underlying relaxed memory consistency model by using Shasha and Snir's delay set analysis so that the compiler presents to the programmer an intuitive and natural memory model based on sequential consistency while performing the optimizations expected from a compiler.

1 Introduction

Java is the first widespread concurrent programming language that explicitly specifies a memory model in the language level. However, as discussed in the Java memory model mailing list [4], the Java memory model is very hard for programmers to understand, and there are several ways of interpreting the memory model. A Java virtual machine can support many threads of execution at the same time. Objects and values can be shared between all threads through a shared main memory. Like most programming languages that follow the shared memory parallel programming model, non-deterministic behaviors due to data races can also occur in Java concurrent programs. The synergic effect of the non-intuitive Java memory model and the non-deterministic behavior makes it difficult for a programmer to write *correct* Java concurrent programs.

For example, consider the code in Figure 1 (taken from Pugh [17]). Under the current Java memory model specification with prescient stores [8], one possible outcome of this code is `X=1` and `Y=1`. The result is not what the programmer would expect from an intuitive interpretation of the original program. By "the output the programmer would expect," we mean the output that comes from executing the program as written with an execution that follows *sequential consistency*. The intuitive interpretation leads one to assume that if the value of `X` is equal to `1` then `Y` should be 0. The reason for this assumption is that if `X` is equal to `1`, `x = 1` should have executed before `X = x`. This means that `y = 1` should have executed after `Y = y`. Thus, the value of `Y` should be 0. One possible execution ordering for the incorrect outcome appears to be *s12, s21, s22, s11*, i.e., *s11* and *s12* are reordered in this execution. The reordering violates sequential consistency. This reordering can be avoided by inserting a fence instruction in between *s11* and *s12* (also in between *s21* and *s22*). In fact, the same problem can occur under the shared memory multiprocessor with a relaxed memory consistency model [11, 14].

In the shared memory parallel programming model, programmers expect the behavior of the memory to be similar to that of a uniprocessor undertaking concurrent execution of several tasks. Lamport formalized an extension of the uniprocessor memory model for multiprocessors called *sequential consistency* [10]. Sequential consistency is arguably the most intuitive and natural memory consistency model for programmers [9]. Sequential consistency is what programmers assume when they program in the shared memory parallel programming model, even if they do not know exactly what sequential consistency is.

Initially, x = 0, y = 0

Thread 1	Thread 2
s11: `X = x`	*s21:* `Y = y`
s12: `y = 1`	*s22:* `x = 1`

Incorrect outcome: `X = 1, Y = 1`

Figure 1: The example of incorrect outcome under the Java memory model with prescient stores

Many shared memory multiprocessors follow a relaxed memory consistency model and provide a wide variety of hardware level optimizations, such as dynamic instruction reordering, speculative execution, and prefetching that take advantage of the memory model. There are many relaxed memory consistency models depending on the degree in which the order between read and write accesses is relaxed. Popular examples include processor consistency, weak ordering and release consistency [1]. The Java memory model is one of the relaxed memory consistency models. There also are relaxed consistency models specific to processor architectures, such as DEC Alpha [19], IBM PowerPC [5], SUN SPARC [21], and Intel IA-64 [7].

However, the relaxed memory consistency model complicates programming and porting because the programmer is exposed to the various instruction reordering optimizations and the atomicity constraints of memory operations. In addition, variations in memory semantics must be considered when porting programs to guarantee correctness of execution. For example, the SUN SPARC V9 architecture supports two relaxed memory models (Partial Store Order (PSO) and Relaxed Memory Order (RMO)) as well as Total Store Order (TSO) model [21]. Switching the default memory model (TSO) into the conceptually more efficient PSO or RMO models would require redesigning the whole operating system running on SPARC multiprocessors [20]. The magnitude of the difficulties introduced by relaxed memory models have led some to argue that future systems should implement sequential consistency as their hardware memory consistency model because the performance boost of relaxed memory consistency models does not compensate for the burden placed on system software programmers [9].

However, we do not believe it is necessary to go that far. If a compiler can control the underlying relaxed memory system by inserting fence instructions through analysis, the programmer could treat the compiler and architecture as a sequentially consistent system and still profit from the performance advantage of the relaxed memory model.

Data races and synchronization make it impossible to apply classical optimization and analysis techniques directly to shared memory parallel programs because the classical methods do not account for updates to variables in threads other than the one being analyzed [12, 13, 15, 11]. Consider the code in Figure 2(A) (taken from Pugh [17]). We assume that `p` and `q` are aliased in this code. Because `p.k` is not modified in between *s11* and *s13* if we consider only Thread 1, we can use `x` in *s11* for the value of `p.k` in *s13*, i.e., it is a legal compiler transformation to replace `p.k` in *s13* with `x` in *s11* if we consider only Thread 1. This optimization is one form of classical common subexpression elimination. However, one of the possible result of the optimized code in Figure 2(B) is `x = 0, y = 1, z = 0`. In this case, `p.k = 1` is interleaved in between *s11* and *s12* in Figure 2(B). This result is not one of the possible outcomes obtained by executing the original code in Figure 2(A). Since the value of `z` is equal to 0 and `p` and `q` are aliased, `p.k = 1` should have executed after `z = p.k`. Thus, `y` should be 0 in the original code. In fact, this compiler transformation has the same effect as moving *s13* immediately after *s11* in Figure 2(A) and executing *s11* and *s13* together without allowing *s21* interleaved in between them, i.e., *s12* and *s13* are reordered. This transformation violates sequential consistency.

Initially, `p` and `q` are aliased and `p.k = 0`

Thread 1	Thread 2
s11: `x = p.k;`	
s12: `y = q.k;`	*s21:* `p.k = 1`
s13: `z = p.k;`	

(A)

Initially, `p` and `q` are aliased and `p.k = 0`

Thread 1	Thread 2
s11: `x = p.k;`	
s12: `y = q.k;`	*s21:* `p.k = 1`
s13: `z = x ;`	

Incorrect outcome: `x = 0, y = 1, z = 0`

(B)

Figure 2: The effect of a compiler optimization. We assume that `p` and `q` are aliased.

In this article, we describe analysis and optimization techniques for an optimizing compiler for ex-

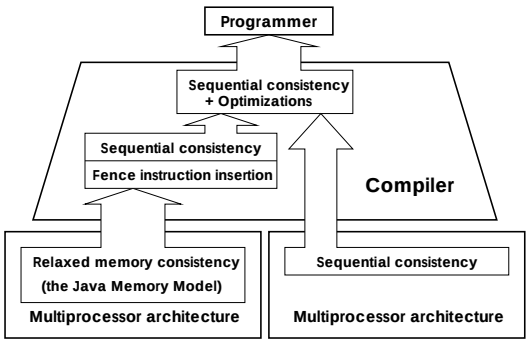

Figure 3: The compiler.

plicitly parallel programs, such as the programs written in Java. The compiler presents a sequentially consistent view of the explicitly programs to programmers by hiding the relaxed memory consistency model of underlying machine architectures while performing the optimizations expected from a compiler [Figure 3]. The characteristics of the compiler are:

- the compiler provides programmers with a sequentially consistent view of the underlying architecture irrespective of the fact that it follows a sequentially consistent model or a relaxed model.
- the compiler makes it possible to apply compiler optimization techniques correctly to parallel programs that are not handled by conventional compilers.

2 Correctness Criteria

2.1 Correctness of Compiler Transformations

An *observable behavior* of a program is a functional behavior of the program (i.e., the program is treated as a black box, and the programmer is interested in only its input and output). We use *observable behavior*, *outcome*, *output*, and *result* interchangeably in this article. An *interleaving* is a permutation consisting of all the operations executed in a parallel program, in which operations in the same thread appear in the order specified by the thread. An interleaving represents a possible execution ordering of a parallel program. But, a possible execution ordering of a parallel program does not necessarily corresponds to an interleaving. First, we define the interleaving semantics of the execution of a parallel program [18].

Definition 2.1 ***Interleaving Semantics***
A parallel program has interleaving semantics if the result of an execution of the program is as if the operations in the program were executed sequentially and atomically in the order of an interleaving. Operations from the same thread are executed in program order; the order of execution of operations belonging to distinct threads is arbitrary.

We rephrase the definition of sequential consistency by Lamport [10] using interleaving semantics to fit into our correctness model.

Definition 2.2 ***Sequential Consistency***
If the result of any execution of a parallel program is the same as the result of an execution that conforms to interleaving semantics, then the execution is sequentially consistent.

Note that the order of operations does not have to be the same. Only the result is important. An execution ordering that corresponds to an interleaving is sequentially consistent.

Under the assumption of sequentially consistent execution of the original program, the defining characteristics of a correct compiler transformation are the following.

Definition 2.3 ***Subset Correctness***
A transformation is correct if the set of all possible observable behaviors of a transformed program is a subset of all possible observable behaviors of the original program.

The implication of sequential consistency with regard to subset correctness is that the transformed program, in which operations are reordered by optimizations, produces a subset of interleavings of the original program. Thus, the possible observable behaviors of the transformed program is a subset of those of the original program. A sequential consistency violation implies a subset correctness violation, but not *vice versa*. The examples shown in Figure 1 and Figure 2 also violate *subset correctness*.

2.2 Delay Set Analysis

A framework called *delay set analysis* is used to enforce sequential consistency. Delay set analysis was originally proposed by Shasha and Snir [18]. The analysis finds a minimal set of execution orderings that guarantees sequential consistency. We use these delays as constraints during compiler transformations.

Let $\mathbf{P}$ be the order enforced by the source program (i.e., program ordering) between operations. Throughout this discussion, operations are assumed to be atomic. $\mathbf{P}$ is the *transitive closure* of the graph which contains the control flow edges of all the control flow graphs of each thread in the parallel program. *A node in the control flow graph represents an operation.* Two nodes m and n will be $m\mathbf{P}n$ if there is a path between m and n. Let $\mathbf{C}$ be a *conflict* relation on variable accesses. The *conflict* relation consists of the set of all pairs (v_i, v_j), where v_i and v_j are operations containing conflicting accesses. Two memory references *conflict* if they access the same memory location in different threads that might execute concurrently, and at least one of them is a write.

A delay relation $\mathbf{D}$ between two operations u and v forces v to wait until u completes execution. A *critical cycle* is a cycle of $\mathbf{P} \cup \mathbf{C}$ that has no chords [1] in $\mathbf{P}$. The delay relation $\mathbf{D}$ enforces sequential consistency if all $\mathbf{P}$ edges in the critical cycles appear in $\mathbf{D}$. If $\mathbf{D}$ consists of all the $\mathbf{P}$ edges in the critical cycles, then $\mathbf{D}$ is a minimal delay relation that enforces sequential consistency in any execution of the program.

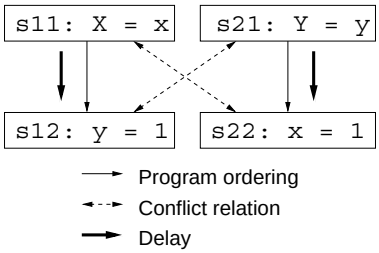

Figure 4: Critical cycles and delays in the example shown in Figure 1.

One outcome of the code (taken from Pugh [17]) shown in Figure 1 is `X = 1`, and `Y = 1`. This is not sequentially consistent. There is a critical cycle (*s12, s21, s22, s11*) in Figure 4, and as a result, the edges (*s11*, *s12*) $\in \mathbf{P}$ and (*s21*, *s22*) $\in \mathbf{P}$ (i.e., $\mathbf{D} = \{$(`s11`, `s12`), (`s21`, `s22`)$\}$) should be enforced as delays to guarantee sequential consistency (to avoid the incorrect outcome).

3 Hiding Relaxed Memory Consistency with Compilers

Since the Java memory model is hard to understand, and there are several ways of interpreting the memory model[17], we show a solution for the weak ordering model [11, 14]. The same framework is used to hide the Java memory model with a compiler.

All the ordering constraints for shared read ($\mathcal{R}$) and write ($\mathcal{W}$) operations in the weak ordering model are given with respect to synchronization operations $\mathcal{S}$:

$$\mathcal{S} \rightarrow \mathcal{R},\ \mathcal{S} \rightarrow \mathcal{W},\ \mathcal{R} \rightarrow \mathcal{S},\ \mathcal{W} \rightarrow \mathcal{S},\ \mathcal{S} \rightarrow \mathcal{S}$$

Since these ordering constraints are already forced by the memory model, we do not need to enforce the delays that can be enforced by the ordering constraints. Let $\mathbf{D}$ be the delays found by the delay set analysis and $\mathbf{D_o}$ be the delays enforced by the ordering constraints (i.e., if $u\mathbf{D_o}v$, then $u \rightarrow v$ match one of the constraint patterns). We want to find a minimal delay relation that enforces correctness together with $\mathbf{D_o}$. Then,

$$\mathbf{D_m} = ((\mathbf{D} \cup \mathbf{D_o})^{+})^{\mathbf{tr}} - \mathbf{D_o}$$

[1] For two nonadjacent nodes u and v in a cycle, a chord is an edge (u, v).

is the minimal delay relation [18], where $^+$ and $^{\mathbf{tr}}$ denote transitive closure and transitive reduction operations respectively. Thus, only the delays in $\mathbf{D_m}$ need to be implemented with special instructions, such as fences [16] and special synchronization operations, depending on the consistency model.

Because the semantics of a fence differ from architecture to architecture, we assume that a *fence* (or a synchronization instruction) has the following semantics:

Definition 3.1 ***Fence***
A fence instruction imposes ordering between memory operations in such a way that when a fence instruction is executed by a processor, all previous memory operations of the processor are guaranteed to have completed. Furthermore, no memory operation of the processor that follow the fence instruction in the program is issued until the fence completes execution.

3.1 Exploiting the Property of Fence and Synchronization Instructions

Fences are inserted at a node of the control flow graph to enforce one or more delays. A naive algorithm may insert more fences than needed to enforce sequential consistency. For example, consider the delays $u\mathbf{D_m}v$ and $w\mathbf{D_m}x$ in Figure 5(A). These delays are in the same thread. Two fences, *f1* and *f2*, are inserted at nodes z and y to enforce the delays. If y executes whenever u and v executes, then we do not need fence *f1* at z [Figure 5(B)]. If we find such y, the fence used to enforce the delay $w\mathbf{D_m}x$ can be used to enforce the delay $u\mathbf{D_m}v$.

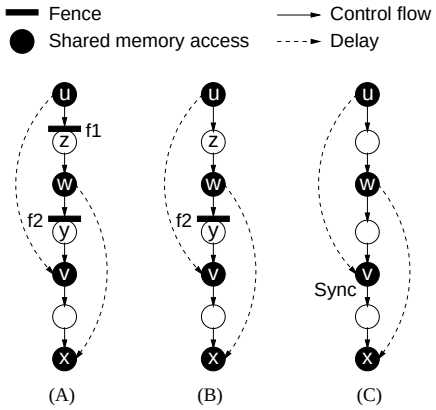

Figure 5: Inserting fence instructions.

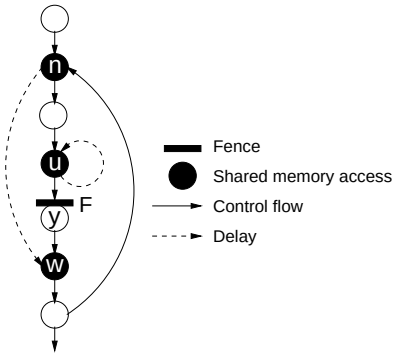

Figure 6: The fence F is enough to enforce both of the delays $n\mathbf{D}w$ and $u\mathbf{D}u$.

We can further reduce the number of fences by exploiting the property of synchronization operations. Consider the delays $u\mathbf{D_o}v$ and $w\mathbf{D_m}x$ in Figure 5(C). We assume that operation v is an explicit synchronization variable access in the source program and is labeled *Sync*. If v executes whenever w and x executes, then the delay $w\mathbf{D_m}x$ (note that this delay does not belong to $\mathbf{D_o}$ because $w \rightarrow x$ does not match any pattern of the constraints) is enforced by the ordering constraints $u \rightarrow v$ and $v \rightarrow x$ which are enforced by the weak ordering model because v is a synchronization operation. Thus, we do not need to insert any fences to the nodes w and x to enforce the delay $w\mathbf{D_m}x$ in Figure 5(C).

3.2 Identifying Memory-Barrier Nodes

In weak ordering, we identify a node y as a synchronization instruction to enforce a delay $u\mathbf{D_m}v$ if y always executes after u and before v whenever u and v execute. From now on, if a node needs a fence instruction or to be identified as a synchronization instruction, we call the node a *memory-barrier node*. A conservative condition for finding a memory-barrier node y that enforces a delay $u\mathbf{D_m}v$ is: *If every path from u to v in the control flow graph of a thread goes through y, then y executes whenever u and v execute.* We want to find a memory-barrier node that enforces as many delays as possible. The example shown in Figure 6 describes this situation. Fence F is enough to enforce both delays $n\mathbf{D_m}w$ and $u\mathbf{D_m}u$. To detect this case, we need to examine all the paths from u to u and n to w, and check whether y is common in these paths.

To find memory-barrier nodes, we introduce the notion of *dominators with respect to a node*. A node n *dominates* a node m (n **dom** m) if every control flow path from the program entry node to m goes through n. The (classical) dominators of a node m are the *dominators with respect to the program entry node* of the control flow graph.

Definition 3.2 ***Dominators with respect to a Node***
A node s dominates a node v with respect to a node u if every control flow path from u to v goes through s. This relation is denoted by s $\mathbf{dom_u}$ v, and the set of dominators s are denoted by $\mathbf{dom_u}[v]$.

We use the iterative algorithm for classical dominators [3, 2] to find dominators with respect to a node u by treating u as the program entry node [11, 14].

3.3 Minimizing the Number of Memory-Barrier Nodes by Using Dominators with Respect to a Node is NP-hard

We can show that minimizing the number of memory-barrier nodes is an NP-hard problem by proving that its decision version is an NP-complete problem [11, 14].

A naive algorithm for minimizing the number of memory-barrier nodes will check each subset of the nodes in $\bigcup_{u\mathbf{D_m}v} \mathbf{dom_u}[v]$ to determine whether the nodes in the subset enforces all the delays $\mathbf{D_m}$. Obviously, this procedure has the exponential complexity. Instead, we use an approximation algorithm to minimize the number of memory-barrier nodes. This is a slight modification of the greedy approximation algorithm developed to solve the optimization version of the **MINIMUM COVER** problem [6].

3.4 Profitability

We also need to consider profitability when we identify memory-barrier nodes and when we insert a fence. Since the goal of the relaxed memory consistency model is to make memory operations to different locations overlapped (pipelined) or reordered in order to hide the memory latency, we want to insert a fence as close to node v as possible in order to maximize the reordering and overlapping by processors if (u,v) is a delay. Also, it is more desirable to insert a fence at the memory-barrier node $n \in \mathbf{dom_u}[v]$ that is located in a less frequently executed path.

4 Compiler Optimization Techniques

The optimization technique introduced in Figure 2(B) is one form of common subexpression elimination. As long as we guarantee the value of `p.k` in *s11* and the value of `p.k` in *s13* are equivalent in the multi-threaded environment, then the optimization is thread-safe and guarantees the correctness of execution. To do so, we conservatively summarize the interactions between threads at the use of each shared variable (e.g., `p.k`'s in *s11* and *s13* and `q.k` in *s12*) by using an intermediate representation that is similar to the CSSA (Concurrent Static Single Assignment) form [11, 15]. Also, a technique that is similar to to the concurrent global value numbering [11, 15] is applied in order to detect equivalent variables. Then, we can safely apply the common subexpression elimination for those equivalent variables.

Some examples of incorrect compiler optimizations for shared memory programs and their solutions are given in [13, 11, 15]. Essentially the same problems can occur in concurrent programs written in Java because Java assumes a shared memory programming model. These optimization techniques include constant propagation, copy propagation, dead code elimination, common subexpression elimination, redundant load/store elimination, and hoistable access detection. The hoistable access detection is a parallel counterpart of classical loop invariant detection.

By extending classical compiler optimization techniques to Java concurrent programs, we can both guarantee the correctness (sequential consistency) of the optimized program and maintain single processor performance in a multiprocessor environment.

5 Conclusions

The details of the Java memory model should be hidden from the programmer. Instead, an intuitive and natural memory model should be presented to programmers. We described a compiler technique that hides an underlying relaxed memory consistency model by using Shasha and Snir's delay set analysis

so that the compiler presents to the programmer an intuitive and natural memory model based on sequential consistency. It shifts the programmer's burden of considering the underlying memory model to the compiler. This facilitates programming and debugging.

To guarantee sequential consistency by hiding the underlying Java memory model, we identify a memory-barrier node for each delay found by the delay set analysis. We introduced dominance with respect to a node relation in order to locate memory-barrier nodes in the control flow graph. In addition, we showed that minimizing the number of memory-barrier nodes by using the dominators with respect to a node is NP-hard. An important implication of identifying memory-barrier nodes is that the compiler can freely apply instruction reordering techniques to the code section in between two consecutive memory-barrier nodes.

Studies of compilers and programming languages have not given much attention to the relationship between correctness and ease of programming issues with memory consistency models. We have shown in this article one method that guarantees correctness (i.e., sequential consistency) and provides ease of programming.

We believe that the techniques presented in this article are the first step towards an optimizing compiler for Java parallel programs.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, December 1996.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[3] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, New York, 1998.

[4] Maintained by Bill Pugh. Java Memory Model mailing list, archives available at `http://www.cs.umd.edu/~pugh/java/memoryModel/`.

[5] Apple Computer, IBM, and Motorola. *PowerPC Microprocessor Common Hardware Reference Platform.* Morgan Kaufmann Publishers, Inc., 1995.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[7] Intel Corporation. *IA-64 Application Developer's Architecture Guide*, May 1999. Rev. 1.0.

[8] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[9] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, pages 28–34, August 1998.

[10] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[11] Jaejin Lee. *Compilation Techniques for Explicitly Parallel Programs.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1999. Technical Report UIUCDCS-R-99-2112.

[12] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proceedings of The 10th International Workshop on Languages and Compilers for Parallel Computing*, number 1366 in Lecture Notes in Computer Science, pages 114–130. Springer, August 1997.

[13] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. A constant propagation algorithm for explicitly parallel programs. *International Journal of Parallel Programming*, 26(5):563–589, 1998.

[14] Jaejin Lee and David A. Padua. Hiding relaxed memory consistency with compilers. In *Proceedings of The 2000 International Conference on Parallel Architectures and Compilation Techniques*, October 2000.

[15] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of The 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1999.

[16] David A. Patterson and John L. Hennessy. *Computer Architecture: A Qunatitative Approach*. Morgan Kaufmann Publishers, Inc., second edition, 1996.

[17] William Pugh. Fixing the Java memory model. In *Proceedings of the ACM 1999 Java Grande Conference*, June 1999.

[18] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[19] Richard L. Sites and Richard T. Witek. *Alpha AXP Architecture Reference Manual*. Digital Press, second edition, 1995.

[20] SUN Microsystems Technical Support, December 1998. Personal Communication.

[21] David L. Weaver and Tom Germond. *The SPARC Architecture Manual*. Prentice-Hall, 1994.