Memory Model = Instruction Reordering + Store Atomicity

Arvind
MIT CSAIL
32-G866, 32 Vassar St.
Cambridge, MA 02139
arvind@csail.mit.edu

Jan-Willem Maessen
Sun Microsystems Laboratories
UBUR02-311, 1 Network Dr.
Burlington, MA 01803
JanWillem.Maessen@sun.com

Abstract

We present a novel framework for defining memory models in terms of two properties: thread-local Instruction Reordering *axioms and* Store Atomicity*, which describes inter-thread communication via memory. Most memory models have the store atomicity property, and it is this property that is enforced by cache coherence protocols. A memory model with Store Atomicity is serializable; there is a unique global interleaving of all operations which respects the reordering rules. Our framework uses partially ordered execution graphs; one graph represents many instruction interleavings with identical behaviors. The major contribution of this framework is a procedure for enumerating program behaviors in any memory model with Store Atomicity. Using this framework, we show that address aliasing speculation introduces new program behaviors; we argue that these new behaviors should be permitted by the memory model specification. We also show how to extend our model to capture the behavior of non-atomic memory models such as SPARC® TSO.*

1. Introduction

One may think of a multithreaded shared-memory system as a single *atomic memory* on which multiple *apparently-sequential threads* are operating. In practice, of course, the memory system on a modern multiprocessor is a complex beast, comprising a tangle of caches, queues, and buffers. Memory consistency models exist to describe and constrain the behavior of these complex systems.

When we refer to an atomic memory we mean that there is a single monolithic memory shared by program threads. Actions on this memory are *serializable*: there is a single serial history of all Load and Store operations, consistent with the execution behavior of each thread, which accounts for the observed behavior of the program.

When we say threads are apparently sequential, we mean that a single thread in isolation will always behave as if it is running sequentially. This implies a few constraints which must not be taken for granted: for example, a Store cannot be reordered with respect to another Load or Store to the same memory location, or the illusion of sequential execution will be shattered. Similarly, we expect that dependencies between branches and subsequent stores are respected; if branch prediction occurs, it cannot have an observable effect.

In this setting, Sequential Consistency [19], or SC, remains the gold standard against which all other multiprocessor memory models are judged [15]. In SC, sequential behavior is enforced by requiring that serializations respect the exact order in which operations occurred in the program. There are two views of SC which are widely understood. The first is a *declarative* view, in which an existing execution is verified by showing that an appropriate serialization of program operations exists. The declarative view is most useful for determining the correctness of a particular implementation of SC. The second view is an *operational* view, in which we model the execution of a program under SC by choosing the next instruction from one of the running threads at each step. The operational view is useful for verifying the correctness of programs running under SC.

By contrast, the behavior of more relaxed memory models is not very well understood. This paper presents a unifying framework in which SC and relaxed memory models with an atomic memory can be understood. These models are distinguished by different rules for instruction reordering, described in Section 2; for our purposes, we consider the instructions within a thread to be *partially ordered* rather than totally ordered as in SC.

As a running example, we choose a relaxed model which permits aggressive instruction reordering. Our

model is similar in spirit to the memory model of the PowerPC™ architecture [24] or the RMO model for the SPARC® architecture [29], though it differs from both in certain minor respects. Such a model is a good choice for future computer systems: it is flexible, and permits fairly ambitious architectural features; it treats all threads uniformly, increasingly important when multiple threads share execution resources that were previously private; and it is simple.

All communication between threads occurs through memory, which we discuss in Section 3. All models with an atomic memory are serializable. However, serializability alone gives very little intuition about the ordering dependencies between instructions in different threads. The most important contribution of this paper is the *Store Atomicity* property (Section 3.3). Store Atomicity describes the ordering constraints which must exist in serializable models.

We represent a program execution as a partial order (or equivalently as a directed acyclic graph). This has the advantage of capturing many indistinguishable serializations in a single, compact form. Store Atomicity operates by connecting Loads directly to Stores; there is no explicit memory in the system. Store Atomicity also imposes additional constraints which capture orderings which must occur in every serialization. These constraints are necessary to maintain Store Atomicity as execution continues.

Like SC, we also provide an operational view of the memory model. Store Atomicity gives a safe and compact way to identify the possible candidate Store operations which might be observed by a given Load operation in a particular execution. These different choices are the sole source of non-determinism in our memory models. In Section 4, we outline an operational model for enumerating all the possible executions of a program in a relaxed memory model.

In Section 5 we present detailed case study of address aliasing speculation in a relaxed memory model. Speculative execution is distinguished by the fact that it can *go wrong*. In general, adding speculation to a memory model is unsafe: it will add new observable behaviors to the model. For example, Martin, Sorin, Cain, Hill, and Lipasti [23] show that naive value speculation violates sequential consistency. We are interested in clearly defining the boundary between safe and unsafe forms of speculation in relaxed memory models. We capture speculation by ignoring ordering constraints. We must roll back execution if enforcing the constraints results in an inconsistency.

Address aliasing speculation is particularly interesting because it allows behaviors which are not permitted by a non-speculative execution, but the resulting model is arguably *simpler* than the non-speculative model (there are fewer dependencies to enforce). Moreover, to our knowledge no relaxed memory model correctly accounts for the additional dependencies required by non-speculative alias detection. Memory models therefore ought to permit this form of speculation.

2^{nd} instr→ 1^{st} instr↓	+, *etc.*	Branch	L y	S y, w	Fence
+, *etc.*	*indep*	*indep*	*indep*	*indep*	
Branch		never		never	
L x	*indep*	*indep*	*indep*	$x \neq y$	never
S x, v			$x \neq y$	$x \neq y$	never
Fence			never	never	

Figure 1. Weak Reordering Axioms. Entries indicate when instructions can be reordered.

Not every memory model obeys Store Atomicity directly. A particularly interesting example of a non-atomic memory model is the Total Store Order (TSO) memory model of the SPARC Architecture. In Section 6 we show how to extend our model in order to capture the behaviors of TSO. However, as processors grow to share execution resources among threads we prefer models which treat memory uniformly. Our relaxed model captures all TSO executions, but permits additional non-TSO executions as well.

We conclude by discussing several possibilities for future extensions to this work.

2. Instruction reordering

In a uniprocessor instructions can be reordered as long as the data dependencies between the instructions are preserved. In a shared memory system one has to be more careful because a parallel program may rely on the relative order of Loads and Stores to different addresses.

It is necessary (for programmers' benefit, if nothing else) for every memory model weaker than SC to provide some mechanism to order any pair of memory operations. Modern processors provide memory fences [29, 24, 16] for this purpose. Fences allow memory operations to be reordered between two fences but force all the Loads and Stores before a fence to be ordered with respect to subsequent Loads and Stores.

Figure 1 presents in tabular form one possible set of rules for reordering instructions. Table entries indicate when instruction reordering is permitted. Instruction pairs with blank entries may always be reordered. Entries marked "never" may never be reordered. Data dependencies constrain execution order, indicated by the entries marked *indep*. Finally, the three entries labeled $x \neq y$ prevent the reordering of Stores with respect to Loads and Stores to the same address; this ensures that

single-threaded execution will be deterministic. In line with present practice, we ignore resource limitations encountered by an actual processor; we permit unbounded register renaming and arbitrarily deep Load and Store pipelines. To the authors' knowledge, no extant memory model imposes resource bounds.

All modern architectures speculatively execute past branch instructions. However, Stores after a speculative branch are not made visible until the speculation is resolved. This is reflected in the "never" entries for Branch.

The reordering rules specify local constraints on the execution of a single thread. For example, the reordering rules in Figure 1 give rise to directed acyclic graphs like the one shown in Figure 5 (ordinary black edges represent reordering constraints). Here in Thread A the reordering constraints permit L_3 and L_5 to be reordered, but neither instruction can be reordered with respect to S_1. Formally, we describe these constraints with a partial order $A \prec B$ ("A precedes B"). It is important to stress that the order of instructions in the original program is relevant *only* in that it governs $\prec$.

There is one very subtle point lurking in the reordering table. It is absolutely necessary to capture *every single* ordering dependency imposed by a particular execution model. For example, consider the instruction sequence S $r,7$; L y, where r is a register containing an address. According to Figure 1, these instructions can be reordered only if $r \neq y$. In a non-speculative execution, L y cannot be reordered until the instruction which computes r has executed. If we permit address alias speculation, these subtle dependencies can be dropped. This allows additional execution behaviors, at the cost of discarding executions which violate the reordering rules. We examine this topic further in Section 5.

3. Store Atomicity

Having established a local ordering $\prec$ among the instructions in a single thread, we must now describe the behavior of multiple threads which execute together. The only means of communication between threads is via Stores and Loads. We begin by giving a formal definition of serializability. This is straightforward, but gives very little insight into how programs behave in practice; serializability is best understood by examining non-serializable behaviors. This allows us to identify ordering relationships $A \sqsubset B$ which must exist in every serialization. We call the resulting definition of $\sqsubset$ *Store Atomicity*.

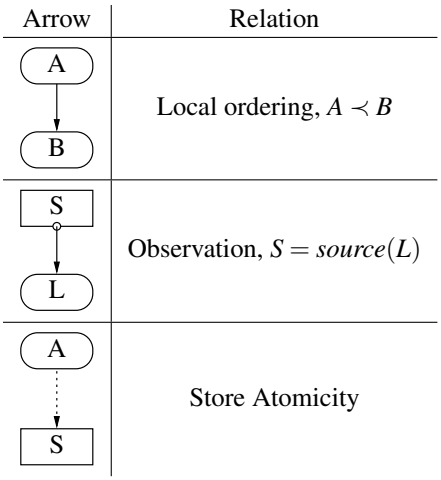


Figure 2. The three types of $\sqsubset$ edges.

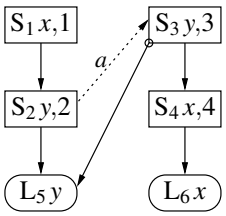
3.1. Serializability

The basic definition of Serializability is quite simple. An execution (or, equivalently, behavior) of a program is given by a partially ordered set of operations. We say $A \stackrel{a}{=} B$ if A and B operate on the same memory address. Every Load L observes the value of some Store S, which we refer to as $source(L)$; clearly $source(L) \stackrel{a}{=} L$. A *serialization* of an execution is a total order $<$ on all operations obeying the following conditions:

1. $A \prec B \Rightarrow A < B$: Local instruction ordering must be respected.
2. $source(L) < L$: A Load executes after the Store from which it obtains its value.
3. $\nexists S \stackrel{a}{=} L.\ source(L) < S < L$: Every load must obtain the value of the most recent store to the same address; there must be no intervening overwriting store.

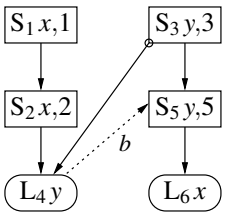
An execution is serializable if there is an order $<$ satisfying the above conditions. In the next section we give some examples of non-serializable executions. In general an execution will have a set X of many possible serializations. We say that $A \sqsubset B$ ("A is before B") if $A < B$ in every serialization in X.

An execution represents a distinct outcome of a multithreaded program: which instructions are executed in each thread, which reordering constraints apply, and most importantly which Store operation is observed by each Load. A program has a set of possible executions. By contrast, the fact that a single execution has many serializations is irrelevant detail: all of these serializations exhibit the same behavior in practice, and there is no real non-determinism involved. In fact, we say two executions are *equivalent* when they have the same set of serializations.



Thread A	**Thread B**
$S_1\ x,1$	$S_3\ y,3$
Fence	Fence
$S_2\ y,2$	$S_4\ x,4$
$L_5\ y$ **= 3**	$L_6\ x$ **= 1?**

Figure 3. When a Store to y is observed to have been overwritten, the stores must be ordered.



Thread A	**Thread B**
$S_1\ x,1$	$S_3\ y,3$
$S_2\ x,2$	$S_5\ y,5$
Fence	Fence
$L_4\ y$ **= 3**	$L_6\ x$ **= 1?**

Figure 4. Observing a Store to y orders the Load before an overwriting Store.

3.2. Violations of serializability

The conditions imposed by serializability are most easily understood by examining examples of executions which appear to violate memory atomicity, and attempting to understand which ordering dependencies exist in serialized executions which prevent those violations. Our goal is to identify ordering relationships which must always hold—that is, to determine when $A \sqsubset B$. We will do this by showing a program fragment and a corresponding execution graph (the $\sqsubset$ relation). The meaning of the edges in our illustrations is summarized in Figure 2. Solid edges are those required by local ordering $\prec$. Ringed edges are *source* edges indicating that the value written by Store S is observed by Load L. Our goal is to identify the dotted *Store Atomicity edges*: additional ordering constraints which must be respected in every execution.

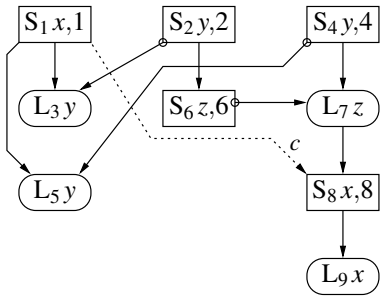
Figure 3 demonstrates that Loads can impose ordering relationships between Stores in different threads. Some notational rules are in order: Loads and Stores are numbered with small subscripts; this numbering is chosen to reflect one possible serialization of the observed execution. Letters refer to constant memory addresses. Non-subscripted numbers are simply arbitrary program data (in our examples we endeavor to have every Store S_k write its unique instruction number k to memory). Finally, the notation $L_5\ y$ **= 3** indicates that in the pictured execution, the Load of y observes the value 3 written by S_3. This is followed by question mark as in $L_6\ x$ **= 1?** if the observation violates serializability.

Here L_5 in Thread A observes S_3, so S_2 must have been overwritten. We capture this by adding the dotted dependency a, making $S_2 \sqsubset S_3$. Thus $S_1 \sqsubset S_4 \sqsubset L_6$; S_1 has been overwritten and cannot be observed by L_6.

Note that we have pictured only one of several possible executions of this fragment. It is possible for L_5 to instead observe S_2. In that case, no known ordering would exist between S_2 and S_3, and L_6 can observe either S_1 or S_4. □

Figure 4 shows that when a Load observes a value which is later overwritten, the Load must occur before the overwriting store. L_4 in Thread A observes S_3 in Thread B. It therefore must occur before S_5 overwrites S_3. We insert the dotted dependency b to reflect this fact, making $L_4 \sqsubset S_3$. Thus $S_1 \sqsubset S_2 \sqsubset L_6$, so L_6 cannot observe S_1, which was overwritten by S_2.

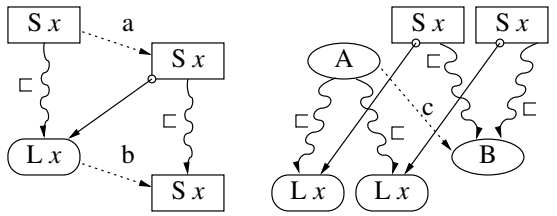
Notice that there is no overwriting Store between S_5 and L_6, so if L_4 instead observes S_5 can observe either S_1 or S_2. □



Thread A	**Thread B**	**Thread C**
$S_1\ x,1$	$S_2\ y,2$	$S_4\ y,4$
Fence	Fence	Fence
$L_3\ y$ **= 2**	$S_8\ z,6$	$L_7\ z$ **= 7**
$L_5\ y$ **= 4**		Fence
		$S_8\ x,8$
		$L_9\ x$ **= 1?**

Figure 5. Unordered operations on y may order other operations.

Figure 5 shows that operations on a single location (here y) may occur in an ambiguous order, but they may establish an unambiguous order of operations elsewhere



a. Predecessor S x must precede *source*(L x).
b. L x must precede successor S x.
c. Parallel pairs of observations of x order the ancestor of both L x before the successor of both S x.

Figure 6. Store Atomicity in brief. Wavy edges are arbitrary $\sqsubset$ relationships.

in the execution. Here S_1 is succeeded by two loads of y, L_3 and L_5. Meanwhile, S_8 is preceded by two stores to y, S_2 and S_4. There are two store/load pairings to y, $S_2 \sqsubset L_3$ and $S_4 \sqsubset L_5$. These pairings cannot be interleaved—for example, we cannot serialize $S_2 < S_4 < L_3$ even though S_4 is unordered with respect to the other two operations. Every serialization of the example in Figure 5 will either order $S_2 < L_3 < S_4 < L_5$ or $S_4 < L_5 < S_2 < L_3$. In either case, it is clear that $S_1 < L_7$. The mutual ancestors of L_3 and L_5 must always precede the mutual successors of S_2 and S_4; this requires the insertion of edge c between S_1 and L_7. Because of this, L_9 cannot observe S_1; it must have been overwritten by S_8. □

Note that we have motivated the examples in this section by looking for contradictory observations, and showing that there are ordering relationships which unambiguously rule them out. This is the chief purpose of the $\sqsubset$ relation: it lets us show not just that an execution is serializable, but also that execution can continue without future violations of serializability.

3.3. The Store Atomicity property

Given an execution $\langle \prec, source, \stackrel{a}{=} \rangle$, the definition of serialization directly tells us the following important facts about the $\sqsubset$ relation:

1. $A \prec B \Rightarrow A \sqsubset B$: local ordering is respected.
2. $source(L) \sqsubset L$: a Load happens after the Store it observes.
3. $\nexists S \stackrel{a}{=} L.\ source(L) \sqsubset S \sqsubset L$: A load cannot observe a Store which is certain to be overwritten.

Definition of Store Atomicity:
Store Atomicity imposes the following additional requirements on the $\sqsubset$ relation (shown graphically in Figure 6):

a. Predecessor Stores of a Load are ordered before its source: A predecessor store to the same location is either observed or it must have occurred before the Store which was observed (see example in Figure 3).

$$S \stackrel{a}{=} L \wedge S \sqsubset L \wedge S \neq source(L) \ \Rightarrow\ S \sqsubset source(L)$$

b. Successor Stores of an observed Store are ordered after its observers: If L sees $source(L)$ then L must have occurred before any subsequent S to the same location (see example in Figure 4).

$$S \stackrel{a}{=} L \wedge source(L) \sqsubset S \ \Rightarrow\ L \sqsubset S$$

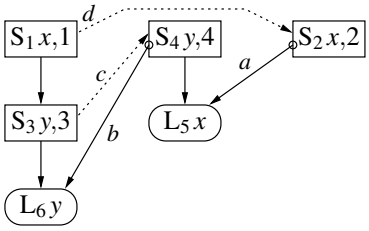
c. Mutual ancestors of unordered Loads are ordered before mutual successors of the Stores they observe.: When store/load pairs to the same address cannot be ordered, they can still impose an order on other nodes (see example in Figure 5).

$$L \stackrel{a}{=} L' \wedge A \sqsubset L \wedge A \sqsubset L' \wedge source(L) \sqsubset B \wedge source(L') \sqsubset B \ \Rightarrow\ A \sqsubset B$$

□

We have presented Store Atomicity as a *declarative* property—we can check an arbitrary execution graph and say whether or not it obeys Store Atomicity. There are two important points in this regard. First, it is legal to introduce additional edges in an execution graph so long as no cycles are introduced—however, doing so rules out possible program behaviors. For example, in Figure 5 we can insert an edge from L_5 to S_6. Doing so rules out any execution in which $S_6 < L_7 < S_4 < L_5$. Real systems make use of this fact to simplify implementation (see Section 4.2). The $\sqsubset$ ordering is the minimal ordering which obeys Store Atomicity; this ensures that legal program behaviors are not dropped.

Finally, note that adding a dependency to enforce Store Atomicity can expose the need for additional dependencies. In Figure 7 no dependency initially exists between S_1 and S_2, even after L_5 observes S_2 (edge a). However, when L_6 observes S_4 (edge b), Store Atomicity requires the insertion of edge c between S_3 and S_4.



Thread A	**Thread B**	**Thread C**
$S_1\ x, 1$	$S_4\ y, 4$	$S_2\ x, 2$
Fence	Fence	
$S_3\ y, 3$	$L_5\ x$ **= 2**	
$L_6\ y$ **= 4**		

Figure 7. Store atomicity may need to be enforced on multiple locations at one time.

This reveals that $S_1 \sqsubset L_5$. We must therefore also insert edge d, $S_1 \sqsubset S_2$. In general, we continue the process of adding dependencies until Store Atomicity is satisfied.

4. Enumerating program behaviors

In this section we give a procedure for generating all possible execution graphs for a program. This is conceptually very simple: Generate a node for each instruction executed, and connect those nodes by edges which correspond to the $\sqsubset$ relation. To generate the graph, a behavior must include the program counter (PC) and register state of each of its threads. Register state is represented by a map $RT[r]$ from a register name to the graph node which produces the value contained in the register at the current PC. When a node is generated, it is in an *unresolved* state. When its operands become available, a node's value can be computed and stored in the node itself; this places the node in a *resolved* state. Conceptually we imagine instructions such as Stores and Fences produce a dummy value; a Branch resets the thread's PC when it is resolved.

In general multiprocessor programs are nondeterministic, so we expect our procedure to yield a set of distinct executions. Every step in our graph execution is deterministic except for the resolution of a Load instruction. Resolving a Load requires selecting a *candidate* Store. Each distinct choice of a candidate store generates a distinct execution. Our procedure keeps track of all these choices; this is the heart of enumeration.

Definition of Candidate Stores For each Load operation L, $candidates(L)$ is the set of all stores $S \stackrel{a}{=} L$ such that:

1. All prior Loads $L' \sqsubset S$ and Stores $S' \sqsubset S$ have been resolved.
2. $\nexists S' \stackrel{a}{=} L.\ S \sqsubset S' \sqsubset L$: S has not been overwritten.

Memory is initialized with Store operations before any thread is started. This guarantees that there will always be at least one "most recent Store" S, so $candidates(L)$ is never empty.

Our definition of $candidates(L)$ is valid only if every predecessor Load of L has been resolved. This is because resolving a Load early can introduce additional inter-thread edges. These new dependencies may cause predecessor Loads to violate Store Atomicity when they choose a candidate Store. We might imagine restricting the definition of $candidates(L)$; however, any simple restriction rules out legal executions. By restricting Load resolution, we avoid this possibility.

4.1. Graph execution

In order to enumerate all the behaviors of a program, we maintain a set of *current behaviors* B; each behavior contains a PC and register map for each thread along with the program graph.

At each step, we remove a single behavior from B and refine it as follows:

1. Graph generation: Generate unresolved nodes for each thread in the system, starting from the current PC and stopping at the first unresolved branch. Insert all the solid $\prec$ edges required by the reordering rules. For example, for a Fence instruction we must add $\prec$ dependencies from all prior Loads and Stores. In effect we keep an unbounded instruction buffer as full as possible at all times.

2. Execution: Execution propagates values dataflow-style along the edges of the execution graph. An non-Load instruction is eligible for execution only when all the instructions from which it requires values have been executed (the Fence instruction requires no data and can execute immediately.) After executing an eligible instruction, update the node with its value. If the result of the instruction serves as an address argument for a Load or Store, insert any $\prec$ edges required by aliasing. Continue execution until the only remaining candidates for execution are Loads.

Repeat steps 1 and 2 until no new nodes are added to the graph.

3. Load Resolution: Insert any dotted $\sqsubset$ edges required by Store Atomicity into the graph. For each unresolved load L whose predecessor loads have been resolved, compute $candidates(L)$. For every choice of Store $S \in candidates(L)$, generate a new copy of the execution. In this execution, resolve $source(L) = S$, and update L with the value stored by S. Once again insert any dotted $\sqsubset$ edges required by Store Atomicity. Add each resulting execution to B. $\square$

Load Resolution is the only place where our enumeration procedure may duplicate effort. Imagine an execution contains two loads L_1 and L_2 which are candidates for resolution. We will generate a set of executions which resolve L_1 first, and then L_2, but we will also generate a set of executions which resolve L_2 first, and then L_1. In many (but not all) cases, the order of resolution won't matter. We discard duplicate behaviors from B at each Load Resolution step to avoid wasting effort. It is sufficient to compare the *Load-Store graph* of each execution. In a Load-Store graph we erase all operations except L and S, connecting predecessors and successors of each erased node. All the graphs pictured in this paper are actually Load-Store graphs; we have erased the Fence instructions.

We have written the above procedure to be as clear as possible. However, it is not a *normalizing strategy*: A program which contains an infinite loop can get stuck in the graph generation and execution phases and never resolve a Load. More complicated procedures exist which fix this problem (for example, by avoiding evaluation past an unresolved Load).

Note that while graph generation blocks at a branch instruction, we nonetheless achieve the effect of branch speculation: Once the graph has been generated, the rules for *candidates*(L) allow us to "look back in time" and choose the candidate store we would have chosen through branch speculation. Branch speculation in our model is captured *by the structure of the graph*, not by the details of graph generation.

4.2. Enforcing Store Atomicity in real systems

When defining $\sqsubset$ we are very careful to insert only those dependencies which are necessary to enforce local instruction ordering and Store Atomicity. But it is safe to impose an ordering between any pair of unordered nodes, so long as we add any Store Atomicity edges which result from doing so. This will eliminate some possible behaviors, but the behaviors which remain will be correct. Real systems have exactly this effect. We can view a cache coherence protocol as a conservative approximation to Store Atomicity. Ordering constraints are inserted eagerly, imposing a well-defined order for memory operations even when the exact order is not observed by any thread.

For example, consider an ownership-based cache coherence protocol. Such a protocol maintains a single canonical version of the data in each memory location, either in memory or in an owning cache. A Store must obtain ownership of the data—in effect ordering this Store after the Stores of any prior owners. Thus, the movement of cache line ownership around the machine defines the observed order of Store operations. Meanwhile, a Store operation must also revoke any cached copies of the line. This orders the Store after any Loads which used the cached data. Finally, a Load operation must obtain a copy of the data read from the current owner, ordering the Load after the owner's Store.

Within a processor, an ordering relationship between two instructions requires the earlier to complete before the later instruction performs any visible action. When operations are not ordered by the reordering rules, they can be in flight simultaneously—but limitations of dependency tracking, queuing, and so forth may force them to be serialized anyhow.

Showing that a particular architecture obeys a particular memory model is conceptually straightforward: simply identify all sources of ordering constraints, make sure they are reflected in the $\sqsubset$ ordering, and show that the resulting constraints are consistent with the local reordering rules and with Store Atomicity.

5. Speculation

We are interested in the general problem of determining whether a given speculative technique obeys a relaxed memory model. What distinguishes speculation from mere reordering is the possibility that it can *go wrong*. We can describe speculation in our graph-based formalism in two ways: First, we can speculatively guess some or all the values which will be manipulated by the program. Instruction execution checks the assumed values against the correct values. Value speculation is open-ended, and we leave it for future work. Second, we can resolve instructions early, in effect ignoring some dependencies. This can result in violations of Store Atomicity. In this section we explore a particular example: address aliasing speculation in our relaxed memory model.

5.1. Disambiguating addresses

Entries of the form $x \neq y$ for Load and Store instructions in Figure 1 are data-dependent, and require us to resolve the aliasing of memory addresses before they can be reordered. Consider the code fragment in Figure 8. Here the memory location x is a pointer containing the address of another memory location. In Thread B the pointer in x is loaded into register r_6. The value 7 is stored in the pointed-to address. L_8 can be reordered with respect to S_7 only if their addresses differ. There are thus two possible local reorderings for Thread B: one where $r_6 = y$ with the dependency $S_7 \prec L_8$, and a second in which $r_6 \neq y$ and no dependency is necessary.

In our non-speculative model we require these aliasing relationships to be resolved before instruction reordering can occur. Thus, every memory operation depends upon the instruction which provides the *address* of each previous potentially-aliasing memory operation. In Figure 1 L_6 is the source of the address of S_7, which potentially aliases L_8, so $L_6 \prec L_8$, as shown in the left-most execution in Figure 9. On an actual non-speculative machine, we do not reorder S_7 and L_8 until L_6 is complete. This is true even though there is no data dependency between L_6 and L_8. This dependency means that $S_2 \sqsubset S_4 \sqsubset L_8$, so L_8 cannot observe S_2.

5.2. Speculative address disambiguation

The center and right-hand diagrams in Figure 9 show the speculative behavior of Figure 8 in the same

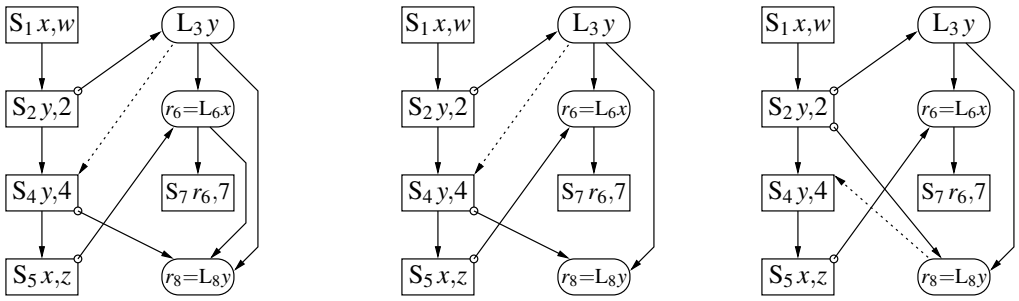


Figure 9. The behaviors of Figure 8 in which *source*$(L_3) = S_2$ and *source*$(L_6) = S_5$.

Thread A	**Thread B**
$S_1\ x,w$	$L_3\ y$ **= 2**
Fence	Fence
$S_2\ y,2$	$r_6 = L_6\ x$ **= z**
$S_4\ y,4$	$S_7\ r_6,7$
Fence	$r_8 = L_8\ y$
$S_5\ x,z$	

Figure 8. Example in which speculative execution of L_8 alters program behavior.

situation. Address aliasing speculation allows us to predict that S_7 and L_8 will not alias. L_8 can be speculatively reordered before both L_6 and S_7; there is no need to wait for r_6 to be loaded. Once r_6 has been loaded, if its value is equal to y we attempt to insert an edge between S_7 and L_8. If *source*$(L_8) \sqsubset S_7$ and $r_6 = y$, L_8 is observing a value overwritten by S_7, violating Store Atomicity. L_8 and any instructions which depend upon it must be thrown away and re-tried.

Speculatively assuming two instructions do not alias eliminates the need to wait for the addresses of potentially-aliasing operations to be resolved. Consequently, the local graph for speculative execution (on the right in Figure 8) omits the edge from L_6 to L_8. The consequence of all this is that L_8 can be reordered before L_4, observing L_2 as shown in the rightmost graph. This behavior was forbidden by the dependency $S_6 \prec L_8$ in the non-speculative execution. The original non-speculative behavior remains valid in a speculative setting (middle graph).

In effect, speculation drops the dependency $S_6 \prec L_8$ required by alias checking. The price is that we may later resolve L_8, then discover that $S_7 \prec L_8$ and that L_8 was not on the frontier of the graph.

The differences between speculative and non-speculative models is often quite subtle. To our surprise, all the behaviors permitted by aliasing speculation appear to be consistent with the reordering rules in Figure 1, but some are nonetheless impossible in a non-speculative model. This is because checking for aliasing introduces a subtle ordering dependency: it is not safe to attempt to reorder two memory operations until both their addresses have been computed. Omitting this dependency leads to *simpler* rules. To our knowledge, no extant processor memory model accounts for the additional dependencies required to resolve aliasing non-speculatively.

6. Total Store Order: A non-atomic model

The Total Store Order (TSO) memory model is a non-atomic memory model which is in wide use in the SPARC architecture [29]. It is of particular interest because it is reasonably well-understood and violates memory atomicity. The only reordering permitted in TSO is that a later Load can bypass an earlier Store operation. Local Load operations are permitted to obtain values from the Store pipeline before they have been committed to memory. In effect, a Load which obtains its value from a local Store must be treated specially.

Figure 10 (based on a similar example in [1]) shows that simple globally-applicable reordering rules cannot precisely capture Store Atomicity. Both threads Store to the flag variable z and then Load from it. These Loads are satisfied from the Store buffer before the Stores become globally visible. This permits the subsequent L_6 and L_{10} to be reordered very early in execution, and to observe S_5 and S_1 instead of S_7 and S_2.

In Figure 11 we see graphs for this particular execution under three different models. First, observe that the leftmost execution is consistent with the rules from Figure 1; these rules are very lenient and permit any TSO execution along with many executions which violate TSO (for this reason, TSO programs require fewer fences than the equivalent programs under RMO).

Second, note that if we simply allow Store/Load reordering as permitted in TSO we obtain the inconsistent execution shown in the center of Figure 11. Here we

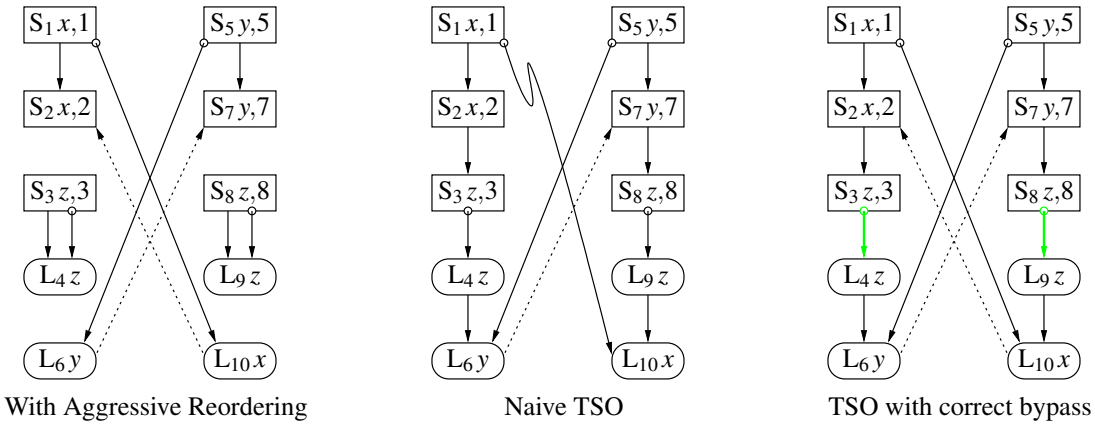


Figure 11. Graphs for Figure 10 in several different models. TSO dependencies are gray.

Thread A	**Thread B**
S_1 x, 1	S_5 y, 5
S_2 x, 2	S_7 y, 7
S_3 z, 3	S_8 z, 8
L_4 z **= 3**	L_9 z **= 8**
L_6 y **= 5**	L_{10} x **= 1**

Figure 10. An execution which obeys TSO but violates memory atomicity.

assume that $source(L_6) = S_5$; according to Store Atomicity, $L_6 \sqsubset S_7$. This means that $S_1 \sqsubset S_2 \sqsubset L_{10}$, so L_{10} obtains a value which has been overwritten.

To capture the behavior of TSO, we use a different kind of edge to represent Store-Load order within a single thread. These dependencies, between S_3 and L_4 and between S_8 and L_9, are shown in grey in the rightmost execution in Figure 11. Because these operations are in the same thread, there is *no* ordering dependency of any kind between them—the grey edges do not figure in to the $\sqsubset$ ordering in any way. By contrast, if (say) L_4 had obtained its value from a S_8, we would consider $S_3 \prec L_4$. Store Atomicity would then require that $S_3 \sqsubset S_8 \sqsubset L_4$. In general if $S \stackrel{a}{=} L$ and S is before L in program order, $S \not\sqsubset L$ when $S = source(L)$ and $S \prec L$ otherwise. $\square$

We believe that models which treat the local thread specially are shortsighted in the modern era of multicore processors. For example, it might seem reasonable for multiple threads running on the same core to use the same bypass optimization, so that a Load from one thread can be satisfied by an outstanding Store from another thread. We advocate the use of memory models which treat all processors symmetrically. We can bracket TSO on either side by models which treat every thread the same way. However, it remains to be seen if there is a store atomic model which incorporates all of the behaviors permitted by TSO, but does not require additional barriers to be inserted into TSO code in order to guarantee program correctness.

7. Related work

The literature on memory models is a study in the tension between elegant, simple specification and efficient implementation. Collier [6] is a standard reference on the subject for computer architects, and established the tradition of reasoning from examples which we have continued. The tutorial by Adve and Gharachorloo [1] is an accessible introduction to the foundations of memory consistency.

The use of graphs or partial orders to represent temporal ordering constraints for memory consistency has a long history. In virtually all work on the subject, cycles in the graph indicate violations of memory consistency. Shasha and Snir [27] take a program and discover which local orderings are involved in potential cycles and are therefore actually necessary to preserve SC behavior; the remaining edges can be dropped, permitting the use of a more weakly-ordered memory system. Condon and Hu [7] use graphs very similar to ours to represent executions of a decidable SC variant. The computation-centric memory models of Frigo and Luchangco [9, 8, 20] use DAGs to capture ordering dependencies between memory operations. Synchronization is implicit in the graph structure—several of the models explored are not sufficiently strong in themselves to encode synchronization using load and store operations. The specification of TSO [29] is given in terms of several partial orders on program instructions in a single thread. However, the inter-thread behavior is defined by serialization. TSOtool [12] constructs a graph representing an observed execution, and uses properties **a** and **b** from Store Atomicity to check for violations of Total Store Order. They do not formalize or check property **c**; indeed, they give

an example similar to Figure 5 which they accept even though it violates TSO.

The post-facto nature of memory semantics creates problems in verifying the correctness of consistency protocols, even for Sequential Consistency. Model checking of SC was shown to be undecidable [13, 4, 26] due to the difficulty of producing a witnessing order for all possible executions. Our model, too, lacks a well-defined notion of time. We avoid some of the resulting problems by establishing a clear mapping between L and $source(L)$ (rather than between L and the value loaded); this is akin to Qadeer's [25] notion of data independence. In practice, however, $candidates(L)$ is limited by considering the temporal order of protocol actions. Work on decidable subsets of SC [3, 13, 5] should thus extend gracefully to weaker models with Store Atomicity.

For programmers, the compiler and runtime can have an enormous influence on memory model guarantees. The idea of properly synchronized programs [2] and of release consistency [11, 18] is to present a programming model which, if obeyed, appears to be sequentially consistent, even with a comparatively weak underlying memory system.

The revised specification of the memory model for the Java™ Programming Language [17, 22] sets an ambitious goal: to completely specify the behavior of every program in such a way that no compiler optimizations are invalidated, and programmers can reason in terms of a high-level model similar to release consistency. The model must encompass an open-ended set of speculative behaviors while forbidding a small class of bad behaviors—those which permit data (such as passwords) to be pulled "out of thin air." Every single load requires a *justification*, a distinct execution from that program point in which the load obtains the value loaded. The justification is discarded; only the value loaded is kept. Thus, it is likely that enumerating the legal behaviors of Java code is at least NP-complete and may be undecidable. The JMM is best viewed purely as a declarative specification which can be applied to executions after the fact to determine their legality.

An alternative to SC is a weak but easily-understood memory model—the goal of location consistency [9, 10], and the goal of our relaxed memory model. The CRF memory model [28] takes this idea to its logical conclusion, by presenting a model rich enough to capture other consistency models, including SC, location consistency, and a proposed memory model for the Java Programming Language [21]. Like the present work, CRF focuses its attention on a set of reordering rules governing the behavior of instructions within a thread. However, it uses a cache-based, rather than a graph-based, model for interprocessor memory consistency, making it harder to reason abstractly about program behavior.

8. Conclusions and future work

In this paper we have examined the behavior of memory models with an atomic memory. These models generalize SC to a setting in which instructions in each thread are partially ordered rather than totally ordered. Our technique is parameterized by a set of reordering rules; it is easy to experiment with a broad range of memory models simply by changing the requirements for instruction reordering.

We have defined Store Atomicity as a property that captures which instructions must be ordered in *any* serialization of an execution. Furthermore, we have given an operational meaning to Store Atomicity, permitting us to enumerate the possible behaviors of a multithreaded program as a set of execution graphs. Such a procedure has long existed for Sequential Consistency; to our knowledge this is the first such procedure for a relaxed memory model. Our enumeration procedure allows us to verify by execution that a program fragment will have the desired behavior in all cases. This can be used by architects to verify memory consistency protocols, but it can also be used by programmers to guarantee that a program actually behaves as expected (for example, to check that a locking algorithm meets its specification).

We have deliberately glossed over some details which are important on a real machine. We assumed all reads and writes accessed fixed-size, aligned words; in practice, loads and stores occur at many granularities from a single byte to whole cache blocks. A faithful model can potentially match a Load up with several Store operations, each providing a portion of the data being read. Real architectures also provide atomic memory primitives such as Compare and Swap which atomically combine Load and Store actions. None of these details is particularly difficult to capture, but as a whole they complicate the presentation of the underlying idea of Store Atomicity.

By drawing a clear boundary between legal and illegal behaviors for a particular memory model, it is easy to judge the safety of speculation using our framework. It is not well-understood how to determine when speculation violates a relaxed memory model. We learned that simple relaxed memory model specifications may permit behaviors which can *only* be obtained through address aliasing speculation. This is because resolving address aliasing non-speculatively introduces subtle ordering dependencies which have been ignored in the past.

In the following paragraphs we discuss several directions in which this work can be extended or applied.

Effect on programming: Previously unseen behaviors are a cause for concern for programmers—programs tested on non-speculative machines may fail in a difficult-to-debug fashion when moved to a speculative machine. Our goal in this paper was a *descriptive* specification, which could enumerate all possible behaviors of a program's execution. Application programmers are better served by a *prescriptive* programming discipline, describing how to write programs to obtain particular behaviors. We are interested in using our model to define and check various prescriptive disciplines. For example, we can say a program is *well synchronized* if for every load of a non-synchronization variable there is exactly one eligible store which can provide its value according to Store Atomicity. This idea generalizes the notion of Proper Synchronization [2] to arbitrary synchronization mechanisms, rather than just locks. When programs obey such a discipline, they can be run using much weaker memory models such as (Lazy) Release Consistency [11, 18].

Tools for verifying memory model violations: It should be relatively easy to take a program execution and demonstrate that it is correct according to a given memory model without the need to compute serializations. Graph-based approaches such as TSOtool [12] have already demonstrated their effectiveness in this area.

Transactional memory: Transactional memory [14] is an appealing programming discipline which has gained increasing intellectual traction in recent years. One may view a transaction as an atomic group of Load and Store operations, where the addresses involved in the group are not necessarily known a priori. It is worth exploring if the *big-step*, "all or nothing", semantics of currently used to describe atomic transactions can be explained in terms of *small-step* semantics using the framework provided in this paper.

Speculative execution: The immediate reaction of many of our colleagues on hearing of the discrepancy between speculative execution and non-speculative execution was "The speculative execution must be wrong." Yet similar behaviors are currently possible on many machines regardless of the specification of the memory model. It is important for designers to understand the implications of allowing or disallowing speculation when specifying a memory model. For implementors of speculation, it must be clear how to decide that failure has occurred, and how to roll back execution when it does so. We are working to formalize a general framework for speculation based on Store Atomicity. This will allow us to treat unusual forms of speculation such as cross-thread speculation; multithreaded architectures provide a fertile ground for exploiting such techniques.

Reference specification of a computer family: It will be worth while to write an ISA specification which permits maximum flexibility in implementation and yet provides an easy to understand memory model. Manuals for current computer systems fall woefully short of this ideal.

Acknowledgments

Krste Asanovic played a vital role in targeting our work to computer architects. The comments of the anonymous reviewers on three different versions of this paper over two years had a major impact on our presentation of semantics to this audience. We've also benefited enormously from discussions with Victor Luchangco, Sarita Adve, Matteo Frigo, and Maurice Herlihy. Jan-Willem Maessen would like to thank the Fortress team, which is part of the Sun Hero HPCS project. Arvind was supported in part by the IBM PERCS project. Both projects are funded in part by the DARPA HPCS program.

References

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, pages 66–76, Dec. 1996.

[2] S. V. Adve and M. D. Hill. Weak Ordering – A New Definition. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 2–14. ACM, May 1990.

[3] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Trans. Program. Lang. Syst.*, 15(1):182–205, 1993.

[4] R. Alur, K. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 219, Washington, DC, USA, 1996. IEEE Computer Society.

[5] J. D. Bingham, A. Condon, and A. J. Hu. Toward a decidable notion of sequential consistency. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 304–313, New York, NY, USA, 2003. ACM Press.

[6] W. W. Collier. *Reasoning about parallel architectures*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[7] A. E. Condon and A. J. Hu. Automatable verification of sequential consistency. In *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 113–121, New York, NY, USA, 2001. ACM Press.

[8] M. Frigo. The weakest reasonable memory model. Master's thesis, MIT, Oct. 1997.

[9] M. Frigo and V. Luchangco. Computation-centric memory models. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, June/July

1998.

[10] G. R. Gao and V. Sarkar. Location Consistency – A New Memory Model and Cache Coherence Protocol. Technical Memo 16, CAPSL Laboratory, Department of Electrical and Computer Engineering, University of Delaware, Feb. 1998.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26. ACM, May 1990.

[12] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu. Tsotool: A program for verifying memory systems using the memory consistency model. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 114, Washington, DC, USA, 2004. IEEE Computer Society.

[13] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 301–315, London, UK, 1999. Springer-Verlag.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[15] M. D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, 31(8):28–34, 1998.

[16] Intel, editor. *Intel IA-64 Architecture Software Developer's Manual*. Intel Corporation, Jan. 2000.

[17] JSR 133. Java memory model and thread specification revision. http://jcp.org/jsr/detail/133.jsp, Sept. 2004.

[18] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13–21. ACM, May 1992.

[19] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[20] V. Luchangco. *Memory Consistency Models for High Performance Distributed Computing*. PhD thesis, MIT, Cambridge, MA, Sep 2001.

[21] J.-W. Maessen, Arvind, and X. Shen. Improving the Java memory model using CRF. In *Proceedings of the 15th AnnualConference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–12, Minneapolis, MN, Oct 2000.

[22] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 378–391, Long Beach, CA, Jan. 2005. ACM SIGPLAN.

[23] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 328–337, Washington, DC, USA, 2001. IEEE Computer Society.

[24] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for A New Family of RISC Processors*. Morgan Kaufmann, 1994.

[25] S. Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. *IEEE Trans. Parallel Distrib. Syst.*, 14(8):730–741, 2003.

[26] A. Sezgin and G. Gopalakrishnan. On the decidability of shared memory consistency validation. In *MEMOCODE '2005: Proceedings of the Third ACM-IEEE Conference on Formal Methods and Models for Codesign*, 2005.

[27] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[28] X. Shen, Arvind, and L. Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999. ACM.

[29] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual (Version 9)*. Prentice-Hall, 1994.