Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 174 (2007) 117–137

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Store Atomicity for Transactional Memory

Jan-Willem Maessen[1]

Sun Microsystems Laboratories
UBUR02-311, 1 Network Dr.
Burlington, MA 01803 USA

Arvind[2]

MIT CSAIL
32-G866, 32 Vassar St.
Cambridge, MA 02139 USA

Abstract

We extend the notion of *Store Atomicity* [4] to a system with atomic transactional memory. This gives a fine-grained graph-based framework for defining and reasoning about transactional memory consistency. The memory model is defined in terms of thread-local *Instruction Reordering* axioms and *Store Atomicity*, which describes inter-thread communication via memory. A memory model with Store Atomicity is serializable: there is a unique global interleaving of all operations which respects the reordering rules and serializes all the operations in a transaction together. We extend Store Atomicity to capture this ordering requirement by requiring dependencies which cross a transaction boundary to point in to the initiating instruction or out from the committing instruction. We sketch a weaker definition of transactional serialization which accounts for the ability to interleave transactional operations which touch disjoint memory. We give a procedure for enumerating the behaviors of a transactional program—noting that a safe enumeration procedure permits only one transaction to read from memory at a time. We show that more realistic models of transactional execution require speculative execution. We define the conditions under which speculation must be rolled back, and give criteria to identify which instructions must be rolled back in these cases.

Keywords: computer architecture, multiprocessor memory consistency, cache coherence, transactional memory, instruction reordering, store atomicity

1 Introduction

In a recent paper [4] we describe *Store Atomicity*, a graph-based framework for reasoning about a large class of multiprocessor memory consistency models. We take the view that a system consists of a single *atomic memory* on which multiple *apparently-sequential threads* are operating. This paper follows up on that work by introducing a simple form of *transactional memory* akin to that described by

[1] Email: `JanWillem.Maessen@sun.com`

[2] Email: `arvind@csail.mit.edu`

Herlihy and Moss [11]. The present paper does not assume familiarity with Store Atomicity; we retain the explanation of the fundamentals of the original work.

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. Introducing transactions requires a more complex definition of serializability; all the operations in a transaction must be serialized as a group.

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 [14], or SC, remains the gold standard against which all other multiprocessor memory models are judged [12]. 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 proven to be sequentially consistent by showing that an appropriate serialization of program operations exists. The declarative view is most useful for determining that the behaviors exhibited by a particular memory system provide the appearance 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; we can in principle enumerate all the possible executions of a program simply by following the operational rules.

Store Atomicity provides a unifying framework in which SC and relaxed memory models with an atomic memory can be understood; however, the version given in [4] cannot account for memory transactions.

As a running example, we choose a relaxed model which permits aggressive instruction reordering. Our model is similar in spirit to a transactional version of the memory model of the PowerPC™ architecture [15] or the RMO model for the SPARC® architecture [19], but it differs from both models in minor respects. For the purposes of this paper we rule out nested transactions, the use of non-transactional memory operations within a transaction, and other extensions of the basic transaction mechanism. Our example model is otherwise flexible, and permits ambitious architectural features; it treats all threads uniformly, increasingly important when multiple threads share execution resources that were previously private; and it is simple.

In addition to the present surge in interest in transactional memory, there are several other reasons it is instructive to model transactions within the framework of Store Atomicity. First, we can use transactions to model existing atomic fetch-

and-operate instructions such as compare and swap. Second, transactions require us to model atomic updates which affect multiple memory locations simultaneously. For example, a complete memory model must deal with byte and word updates to the same storage. We might consider a word access to be an atomic group of byte-sized accesses; conversely we might instead model a byte store as an atomic read-modify-update of a word-sized location. In either case we might reasonably ask whether there are ordering constraints on the various components of these operations (the fetch, modify, and update of each of the locations involved).

Memory models with atomic storage are distinguished by varying rules for intra-thread 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. All communication between threads occurs through memory, which we discuss in Section 3; memory obeys the same rule (serializability) in all of the models we consider.

However, serializability alone gives very little intuition about the ordering dependencies between instructions in different threads. Store Atomicity (Section 3.4) describes the ordering constraints which must exist in serializable models. We reason about an execution, characterized by the instructions executed, the mapping *source* between each Load and the Store it observes, and the partial order $\prec$ of each thread's instructions. Note that there is no explicit memory in our formalism; Store Atomicity reasons directly in terms of the Load and Store operations and the *source* relation. Formally, there are three closely-related views of Store Atomicity which we make use of in this paper, and it is useful to spell them out:

First, given an execution along with a partial order (or equivalently a directed acyclic graph) of dependencies among instructions, we can say whether the execution obeys the conditions of Store Atomicity. The central claim of our work is that any execution for which there is a partial order which obeys Store Atomicity is serializable. The partial order here may correspond to a concrete implementation of a particular desired memory model; a violation of Store Atomicity indicates a problem in the implementation.

Second, given a serializable execution, we can define the least partial order $\sqsubset$ which describes that execution and obeys Store Atomicity. Our second central claim is that any serializable execution possesses such an ordering. We can ask which pairs of instructions are ordered in every possible serialization of an execution. We claim that under the definition of Store Atomicity given in Section 3.4 two instructions are ordered in every serialization of an execution if and only if they are ordered by $\sqsubset$. However, this is not true for the relaxed rules for transactions given in Section 3.5.

Finally, given a program we can enumerate all possible executions of that program in one of the memory models of interest. In Section 4 we give a strategy for doing so. It relies crucially on constructing the $\sqsubset$ relation as the program executes. This permits us to identify *candidates*(L), the Store operations which can be observed by a Load L without leading to a violation of serializability.

Representing a program execution as a partial order or DAG has the advantage of capturing many indistinguishable serializations in a single, compact form. The

2^{nd} instr$\rightarrow$	+, *etc.*	Branch	L	S	Fence	Trans	Commit
1^{st} instr$\downarrow$			y	y, j			
+, *etc.*	*indep*	*indep*	*indep*	*indep*			
Branch		never		never			
L x	*indep*	*indep*	*indep*	$x \neq y$	never		never
S x, i			$x \neq y$	$x \neq y$	never		never
Fence			never	never			
Trans			never	never		N/A	never
Commit						never	N/A

Fig. 1. Weak Reordering Axioms. Entries indicate when instructions can be reordered. Instructions pairs marked N/A are illegal and should not occur.

operational view in Section 4 therefore provides a more compact way to reason about SC execution than the usual technique of enumerating all possible serializations. The sole source of non-determinism in our operational characterization is the choice of which Store is chosen as *source*(L) from among *candidatesL*.

In the presence of transactions we must either restrict parallel execution of transactions, or we must *speculate*—permit executions in which $\sqsubset$ would need to contain a cycle in order to obey Store Atomicity. This is the final reason for our interest in transactions: a transaction potentially running in parallel with a conflicting memory update fundamentally requires the use of speculative execution. The usual execution model for transactions *aborts* the transaction and re-executes it from the beginning if memory consistency is violated. Indeed, one reason for the recent surge in interest in transactional memory is that it can leverage existing techniques for speculative execution to provide a powerful and easy-to-use synchronization mechanism.

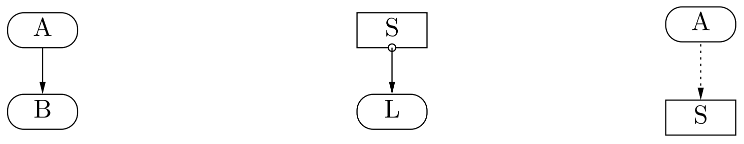
In Section 5 we formally define speculation as any execution which may result in a violation of Store Atomicity. Broadly, speculation must identify and break cycles in $\sqsubset$ before they form. Because such cycles span multiple threads, there are usually several choices of which instructions to roll back in order to recover from speculation failure. We describe how to identify minimal sets of instructions to roll back. It is not generally necessary to roll back entire transactions on speculation failure.

We conclude by discussing some of the most relevant related work, then look ahead at applications of the techniques we describe in this paper.

2 Instruction reordering

On a uniprocessor instructions can be reordered as long as the apparent sequential behavior of the program is preserved. On a shared memory system one has to be more careful because parallel threads may rely on the relative order in which Loads and Stores to several addresses are performed.

Figure 1 presents in tabular form one possible set of rules for reordering instructions. Table entries indicate when instruction reordering is permitted. Instruction



Local ordering, $A \prec B$ Observation, $S = source(L)$ Store Atomicity

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

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*. In this and subsequent figures w, x, y, z stand for arbitrary addresses, while i, j, k, l stand for arbitrary values. In either case a register value r_i may be used. 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.

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 [19,15,6] 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 speculative execution of memory transactions offers opportunities for additional behaviors in this vein.

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. At the highest level, any multithreaded execution must be *serializable modulo reordering*. We begin by giving a formal definition of serializability in a transactional setting; this definition is more complex than the definition of non-transactional serializability.

A formal definition of serializability gives very little insight into how programs behave in practice; our goal is to uncover the conditions under which pairs of instructions must be ordered in every serialization. These conditions define Store Atomicity. We say $A \sqsubset B$ ("A is before B") when the rules of Store Atomicity require instruction A to be ordered before instruction B. We explore the conditions which must be imposed by Store Atomicity by examining non-serializable behaviors. One shortcoming of our definition of transactional Store Atomicity is that it rigidly orders entire transactions when those transactions access overlapping state; we consider how we might go about relaxing this restriction.

3.1 Serializability

An execution (or, equivalently, behavior) of a program is given by a partially ordered set of operations. We say $A \overset{a}{=} B$ 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 $\mathit{source}(L)$; clearly $\mathit{source}(L) \overset{a}{=} L$. In our definitions, we abbreviate Trans instructions by T and Commit instructions by C. We define the set of instructions in the transaction delimited by T and C, $\mathit{transaction}(T, C)$, as follows:

(i) $\emptyset$ if $T \not\prec C \vee \exists C'.T \prec C' \prec C \vee \exists T'.T \prec T' \prec C$ (T and C do not delimit a transaction)

(ii) $\{A \mid T \prec A \prec C\}$ otherwise.

We make use of the fact that the reordering rules in Figure 1 imply that $T \prec A \prec C$ exactly for instructions in the transaction.

A *serialization* of an execution is a total order $<$ on all operations obeying the following conditions:

(i) $A \prec B \Rightarrow A < B$: Local instruction ordering must be respected. This is what we mean by serialization *modulo instruction reordering.* Conventionally serialization is instead defined with respect to the original program order.

(ii) $\mathit{source}(L) < L$: A Store executes before any Load which observes it.

(iii) $\nexists S \overset{a}{=} L.\ \mathit{source}(L) < S < L$: Every load must obtain the value of the most recent store to the same address; there must be no intervening overwriting store.

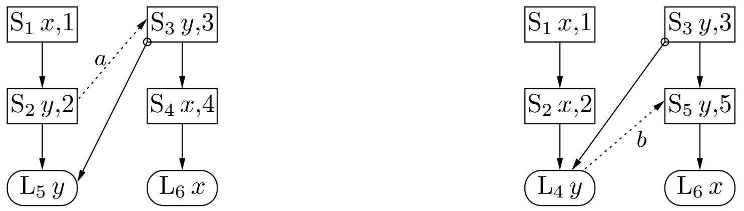
(iv) $T < A < C \Leftrightarrow (A \in \mathit{transaction}(T, C) \vee \mathit{transaction}(T, C) = \emptyset)$: For every transaction the instructions between T and C in the serialization are exactly the instructions in the transaction.

An execution $\langle \prec, \mathit{source}, \overset{a}{=} \rangle$ is serializable if there is an order $<$ satisfying the above conditions. In general an execution will have a set X of many possible serializations. Our goal is to define Store Atomicity such that $A \sqsubset B$ 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. We say two executions are *equivalent* when they have the same set of serializations.

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 in serialized executions prevent those violations. Those readers familiar with our previous paper [4] can safely skip this



Thread A	**Thread B**	**Thread A**	**Thread B**
$S_1\ x, 1$	$S_3\ y, 3$	$S_1\ x, 1$	$S_3\ y, 3$
Fence	Fence	$S_2\ x, 2$	$S_5\ y, 5$
$S_2\ y, 2$	$S_4\ x, 4$	Fence	Fence
$L_5\ y\ = \mathbf{3}$	$L_6\ x\ = \mathbf{1?}$	$L_4\ y\ = \mathbf{3}$	$L_6\ x\ = \mathbf{1?}$

Fig. 3. Left: When a Store to y is observed to have been overwritten, the stores must be ordered. Right: Observing a Store to y orders the Load before an overwriting Store.

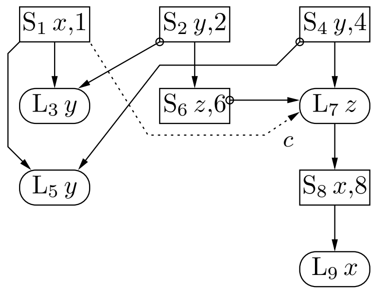
section, as the motivating examples have not changed; the next section describes similar examples of transactional executions which lead to violations of serializability. Each example shows a program fragment and a corresponding execution graph (the desired $\sqsubset$ relation). The meaning of the edges 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. These edges indicate ordering relationships which must always hold—that is, for which $A < B$ in every serialization.

Figure 3, left 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 Store S_k write its unique instruction number k to memory). Finally, the notation $L_5\ y\ = \mathbf{3}$ 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 $L_6\ x\ = \mathbf{1?}$ 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 could observe either S_1 or S_4. $\square$

Figure 3, right 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 $L_4 \sqsubset S_5$. Thus $S_1 \sqsubset S_2 \sqsubset L_6$, so L_6 cannot observe S_1, which was overwritten by 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_6 $z, 6$	L_7 z $=$ **6**
L_5 y $=$ **4**		Fence
		S_8 $x, 8$
		L_9 x $=$ **1?**

Fig. 4. Unordered operations on y may order other operations.

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

Figure 4 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 in the execution. Here S_1 is succeeded by two loads of y, L_3 and L_5. Meanwhile, L_7 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 4 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 Store Atomicity: it lets us show not just that an execution is serializable, but also that execution can continue without future violations of serializability.

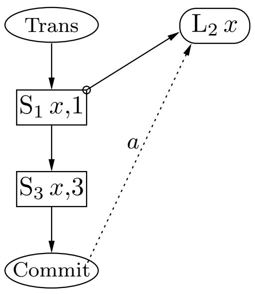
3.3 Violations of Transactional Atomicity

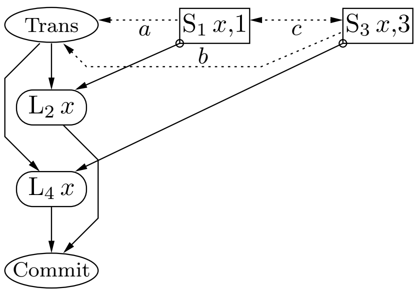
In this section we examine two transactional programs which violate the atomicity of transactions. Our goal is to understand how to enforce the dependencies described in the previous section when a transaction is involved.

Figure 5, left shows that if an instruction in a transaction precedes an instruction outside that transaction, then the Commit operation of the transaction must also precede that instruction. Here S_1 is followed within the same transaction by S_3. If L_2 observes S_1, then $S_1 < S_3 <$ Commit $< L_2$ in every serialization; we insert the additional dependency a to reflect this fact. But this edge shows that S_1 has been overwritten and this observation is impossible.

Notice that reasoning transactionally has a somewhat different flavor from the examples in the previous section: The contradictory edge a is only inserted as a result of the attempt to read S_1, it did not previously exist in the graph. □

Figure 5, right shows that if an instruction in a transaction follows an instruction





Thread A	**Thread B**
Trans	$L_2\ x\ = \mathbf{1?}$
$S_1\ x, 1$	
$S_3\ x, 3$	
Commit	

Thread A	**Thread B**	**Thread C**
Trans	$S_1\ x, 1$	$S_3\ x, 3$
$L_2\ x\ = \mathbf{1}$		
$L_4\ x\ = \mathbf{3?}$		
Commit		

Fig. 5. Left: Violation of atomicity of reads in a transaction. Right: Violation of atomicity of writes in a transaction.

outside the transaction, then the Trans operation of the transaction must also follow that instruction. Here L_2 observes S_1. By the definition of Linearizability, $S_1 <$ Trans $< L_2$ in every serialization; we insert the edge a to reflect this fact. If L_4 observes S_3, we must similarly conclude that $S_3 <$ Trans $< S_4$ in every serialization, reflected by inserting the edge b. But now both Stores precede both Loads; reasoning as in Figure 3 we are obliged to insert both edges c—these two Store operations cannot be serialized, and we conclude that L_4 cannot observe S_3 given that $\mathit{source}(L_2) = S_1$. □

3.4 The Store Atomicity property

In this section we define the Store Atomicity property formally. Given an execution $\langle \prec, \mathit{source}, \overset{a}{=} \rangle$, we define $\sqsubset$ as the least partial order (that is, the order with the fewest dependencies) obeying the rules given in this section. Any execution for which a valid ordering $\sqsubset$ exists is said to obey Store Atomicity. The key conjecture of Store Atomicity is that an execution is serializable if and only if it obeys Store Atomicity.

Definition of Store Atomicity:
The definition of serialization directly tells us the following important facts about the $\sqsubset$ relation:

(i) $A \prec B \Rightarrow A \sqsubset B$: local ordering is respected.

(ii) $\mathit{source}(L) \sqsubset L$: a Load happens after the Store it observes.

(iii) $\nexists S \overset{a}{=} L.\ \mathit{source}(L) \sqsubset S \sqsubset L$: A load cannot observe a Store which is certain to be overwritten.

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).

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

b. Successor Stores of an observed Store are ordered after its observers: When L sees $\mathit{source}(L)$, L must occur before any subsequent S to the same location (see example in Figure 3).

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

c. Mutual ancestors of Loads are ordered before mutual successors of the distinct Stores they observe: Store/load pairs to the same address impose an order on other nodes, even if they themselves are not ordered (see example in Figure 4).

$$\begin{gathered} L \overset{a}{=} L' \ \wedge\ A \sqsubset L \ \wedge\ A \sqsubset L' \ \wedge\ \mathit{source}(L) \neq \mathit{source}(L') \ \wedge \\ \mathit{source}(L) \sqsubset B \ \wedge\ \mathit{source}(L') \sqsubset B \ \Rightarrow\ A \sqsubset B \end{gathered}$$

d. Predecessor operations precede the start of a transaction: A predecessor operation must either occur in the same transaction, or must precede the transaction itself.

$$B \in \mathit{transaction}(T, C) \ \wedge\ A \sqsubset B \ \Rightarrow\ A \in \mathit{transaction}(T, C) \vee A \sqsubseteq T$$

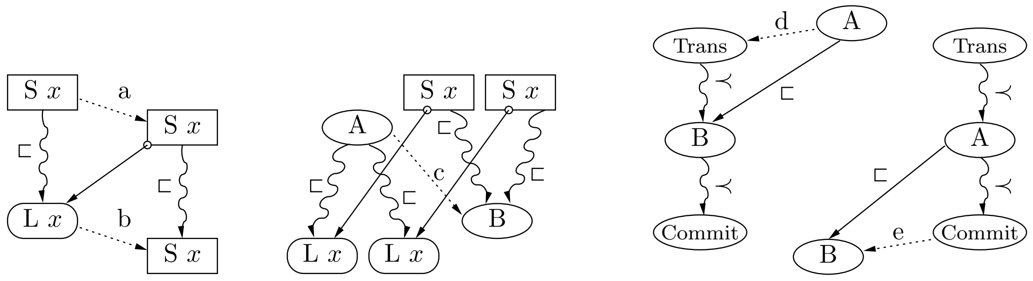
e. Successor operations follow the end of a transaction: A successor operation must either occur in the same transaction, or must succeed the transaction as a whole.

$$A \in \mathit{transaction}(T, C) \ \wedge\ A \sqsubset B \ \Rightarrow\ B \in \mathit{transaction}(T, C) \vee C \sqsubseteq B$$

□

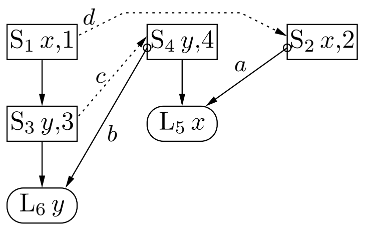
Collectively, we refer to **a–e** as the *Store Atomicity Conditions*. The last two properties (which talk about the relationship between the $\sqsubset$ relation as a whole and the transactional structure of the program) subsume any conditions we might set about specific relationships required by the *source* relation or by the Store Atomicity rules. Note in particular that together they imply that if $A \sqsubset B$ in different transactions, then the Commit operation of the first transaction will precede the Trans operation of the second transaction (thus the careful use of $\sqsubseteq$ in **d** and **e**). Note also that they imply in practice that any operation in the same thread as a transaction is ordered either before the Trans operation or after the Commit operation in spite of the reorderings permitted by Figure 1. In Section 3.5 we consider how these two conditions might be relaxed while still guaranteeing serializability.

These rules describe Store Atomicity as a *declarative* property—we can check an arbitrary execution graph and say whether or not it obeys Store Atomicity. An



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.
d. Predecessor operations precede the start of a transaction.
e. Successor operations follow the end of a transaction.

Fig. 6. Store Atomicity in brief. Wavy edges are arbitrary $\sqsubset$ relationships. When an operation occurs inside a transaction, incoming atomicity edges must point to the Trans operation and outgoing atomicity edges must start at the Commit operation.



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

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

execution graph is any graph which obeys the rules given above (in particular, unlike $\sqsubset$ it need not be the least such graph). 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 4 we can insert an edge from L_5 to S_2. Doing so rules out any execution in which $S_2 < L_3 < S_4 < L_5$. Real systems make use of this fact to simplify implementation (see Section 4.2). Because $\sqsubset$ is the least ordering which obeys store atomicity, it ensures that legal program behaviors are not ruled out prematurely in this way.

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. 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.

3.5 Relaxing Serializability for Transactions

If two transactions touch entirely disjoint memory locations, then the dependencies inserted by Store Atomicity permit these instructions to run in parallel—the result will be indistinguishable from serialized executions in which one transaction or the other is serialized first. However, conditions (d) and (e) in Section 3.4 have the effect of strongly ordering operations which interact with a transaction. This obviously satisfies the definition of transactional serializability from Section 3.1. However, it is possible to relax this condition while still preserving the *existence* of a serializable execution for transactions which share resources in common. We see examples of this in practice: A transaction begins, another thread writes to a particular location, and then the transaction reads the value written. So long as the write *could have* been reordered before the start of the transaction, this behavior is consistent. The only thing that matters is that potentially-conflicting instructions aren't *required* to overlap. We therefore conjecture that the following definitions are sufficient to guarantee serializability:

d'. Operations must not interleave with a transaction: An operation which is ordered (in the sense of $\sqsubset$) with respect to both the beginning and end of a transaction must be part of that transaction.

$$\mathit{transaction}(T,C) \neq \emptyset \;\wedge\; T \sqsubset A \sqsubset C \Rightarrow A \in \mathit{transaction}(T,C)$$

e'. Unique source for each address loaded: A Store which acts as the source for a Load is either in the same transaction as that Load, or precedes all Loads of the same location in that transaction.

$$\begin{array}{l} L \in \mathit{transaction}(T,C) \;\;\Rightarrow \\ \quad (\mathit{source}(L) \in \mathit{transaction}(T,C) \;\vee \\ \quad\; \forall L' \in \mathit{transaction}(T,C).\; L \overset{\mathrm{a}}{=} L' \Rightarrow \mathit{source}(L) \sqsubset L') \end{array}$$

□

Note that when stores to the same location are ordered (replacing the L/L entry in Figure 1 with $x \neq y$), condition **d'** implies condition **e'**. This is true for most of the architectural memory models in use, though it is generally not true of the memory model exposed by compilers.

Note also that under condition d' it is no longer the case that $A < B$ in every serialization implies $A \sqsubset B$: Operations outside a transaction are no longer strongly ordered with respect to that transaction's Trans and Commit instructions. However, they are ordered with respect to all operations on the same address within the transaction. We conjecture that there is a weakening of the definition of transactional serialization given in Section 3.1 which does preserve this property, and which permits exactly the same Load/Store behaviors as the present definition.

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 produced 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 non-deterministic, 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 $S \overset{\text{a}}{=} L$ such that:

(i) All prior Loads $L' \sqsubset S$ and Stores $S' \sqsubset S$ have been resolved.

(ii) $\nexists S' \overset{\text{a}}{=} L.\ S \sqsubset S' \sqsubset L$: S has not been overwritten.

(iii) If S is in a transaction, and L is not in the same transaction, that transaction must have committed and S must be the most recent store $S \overset{\text{a}}{=} L$ in that transaction:

$$\begin{aligned} &S \in \mathit{transaction}(T, C) \ \wedge \ L \notin \mathit{transaction}(T, C) \ \Rightarrow \\ &\quad \nexists S' \in \mathit{transaction}(T, C). S' \overset{\text{a}}{=} S \ \wedge \ S \prec S' \end{aligned}$$

(iv) If L is in a transaction, and S is not in the same transaction, then any load $L' \overset{\text{a}}{=} L$ obtains its value either from S or from a store in the same transaction:

$$\begin{aligned} &L \in \mathit{transaction}(T, C) \ \wedge \ \mathit{source}(L) \notin \mathit{transaction}(T, C) \ \Rightarrow \\ &\quad \forall L' \in \mathit{transaction}(T, C). L' \overset{\text{a}}{=} L \ \Rightarrow \\ &\qquad \mathit{sourceof}(L') = S \vee \mathit{sourceof}(L') \in \mathit{transaction}(T, C) \end{aligned}$$

It should be evident from the last two conditions above that transactional memory substantially complicates the definition of *candidates*(L).

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: resolving a Load 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. A 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.

If there is any transaction with a mixture of unresolved and resolved loads, we only consider the unresolved loads in this transaction during the Load Resolution step. In effect we run this transaction exclusively until all its loads have been resolved. This rules out executions such as the one in Figure 8 where two different transactions make mutually irreconcilable observations. We discuss this issue further in Section 5. □

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, S, Trans, and Commit, 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 (one starting point is to avoid unfolding or execution 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.

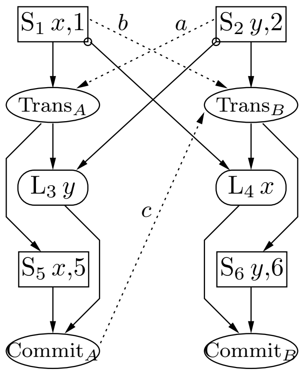
4.2 Enforcing Store Atomicity in real systems

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, typically imposing a well-defined order upon 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.

Transactional memory protocols typically build on this notion of ownership, preventing cache lines touched during the transaction from being changed by another thread until the transaction commits. This has the effect of ordering transactions and of ensuring all Loads see consistent data. Because writes are not shared until a transaction commits, only the final Store to a given location in the transaction is observed. In effect, other processors which use data written by the transaction are ordered after the Commit operation. However, this protocol can deadlock on examples such as the one in Figure 8 unless transactions are serialized. In practice speculation is used to resolve such conflicts: one of the transactions is rolled back.

Within a processor, an ordering relationship between two instructions requires the earlier to complete before the later instruction performs any visible action.



Thread A	**Thread B**
$S_1\ x, 1$	$S_2\ y, 2$
$Trans_A$	$Trans_B$
$L_3\ y\ = \mathbf{2}$	$L_4\ x\ = \mathbf{1?}$
$S_5\ x, 5$	$S_6\ y, 6$
$Commit_A$	$Commit_B$

Fig. 8. Reading ahead results in violations of read atomicity.

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. In practice, of course, identifying every possible source of ordering dependencies in a particular cache coherence protocol is the chief challenge of protocol verification; too few dependencies, and the memory model is violated, too many and the protocol will deadlock.

5 Speculation

The use of speculative execution is fundamental to hardware transactional memory implementations. The founding principle of transactional memory was to permit lock-free parallel updates to shared data structures [11]. As we showed in Figure 8, it is possible for two simultaneously-executing transactions to end up in a mutually inconsistent state when, as part the transaction, each one overwrites state previously read by the other transaction. One or the other of the transactions must be *rolled back*.

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 perform value speculation, guessing values and verifying them later; we defer this to future work. Second, we can resolve instructions early, before all of their dependencies have been satisfied. This can result in violations of Store Atomicity. In [4] we discuss a particular example, address aliasing speculation, arguing that while it permits new behaviors compared with a non-speculative model, it leads to a simpler and easier-to-understand memory model.

Figure 8 demonstrates that running multiple transactions in parallel can lead to violations of serializability. Here L_3 observes S_2, which implies that $S_2 < Trans_A < L_3$

(edge a). In parallel, L_4 observes S_1, which in turn implies that $S_1 < Trans_B < L_4$ (edge b). When S_5 subsequently executes, we must following the same reasoning as in Figure 3, and conclude that $L_3 < S_6$ and thus that $L_3 < Commit_A < Trans_B < S_6$ (edge c). Now we can observe that in any serialization $S_1 < S_5 < L_4$ and thus that it was illegal for L_4 to have observed S_1. The execution thus far cannot be serialized; one of the transactions must be aborted and rolled back. Running multiple transactions in parallel requires speculative execution. □

The operational framework given in Section 4.1 limits Load Resolution to a transaction which contains a mix of resolved and unresolved Loads. Conceptually, it is simple to permit Load Resolution to occur in any thread: simply discard a behavior if it is ever discovered to contradict the rules of Store Atomicity. An actual system, however, only works with one behavior at a time. When an inconsistency occurs, we must decide which instructions to roll back.

It is straightforward to emulate the behavior of current transactional memory implementations: with each Trans instruction we associate a *snapshot* of the thread state at the point where the instruction was issued. If a violation of Store Atomicity is discovered after resolving a Store in a particular transaction, all instructions in the transaction can be erased; we then restart with the snapshot saved at the Trans instruction.

This approach is somewhat restrictive, however. First, it chooses a fixed strategy for conflict resolution—always roll back the transaction which most recently resolved a Load. Second, it does not account for more aggressive forms of speculation. For example, we might consider a model in which transactional Store operations can be observed before the transaction commits, in a manner similar to the execution shown in Figure 5. If these Store operations are later overwritten (as in S_3), it causes speculation failure in any observers (such as L_2 and any subsequent instructions in thread B). Finally, it may often be possible to perform *partial* rollback of transactions.

If we can describe the *minimal* sets of instructions which must be discarded when speculation fails, it is then easy to judge the correctness of any particular rollback mechanism: it must roll back a superset of a minimal set of instructions.

Violations of Store Atomicity lead to dependency cycles. This is true even for examples such as the one in Figure 8 in which the violation can first be detected as an observation of an overwritten value. In the figure, rule **b** from Section 3.4 requires us to insert an additional edge $L_4 \sqsubset S_5$, and thus by rules **d** and **e** we must also insert an edge $Commit_B \sqsubset Trans_A$. This results in a cycle involving both transactions. Any minimal set of instructions must begin with a Load, since this represents the first place in the execution where a different choice can be made which would lead to a consistent outcome. We can choose to roll back any Load instruction which breaks the cycle, erasing it and any instructions which depend upon it (either in the same thread or via *source* edges). Because the cycle in Figure 8 encompasses both transactions, we are free to choose to erase either L_3 and $Commit_A$ or L_4 and $Commit_B$. In the right-hand example Figure 5, the operations L_2 and L_4 do not lie on the cycle; they simply cause the insertion of an edge involved in the cycle. We are free to roll back either one.

In practice, however, we can obtain conflicts between transactions for which it makes sense to roll back a Store instruction rather than a Load. This is because our model has infinite recall—if we roll back a Load, we are free to choose a new candidate Store which might be an ancestor or descendant of the original Store chosen. In practice, there is typically only one value available for a Load to observe. By rolling back a Store operation, we make an older, overwritten value available for observation.

6 Related work

The literature on memory models is a study in the tension between elegant, simple specification and efficient implementation. Collier [5] 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.

Hardware Transactional Memory was first proposed by Herlihy and Moss over a decade ago [11]. There has been a surge in interest in recent years, in part because of close connections between the transactional model and ideas such as speculative lock elision [17] and speculative loop parallelization, culminating in efforts such as TCC [8]. However, there is relatively little work presenting formal semantics for transactional memory; existing formalizations [10] take a "big step" view of transactional execution in which a transaction is thought of as a single, all-or-nothing step. Our hope is that our *small step* view of transactional execution helps clarify the fine-grained behavior of actual transactional memory mechanisms.

The work described in this paper stands on the shoulders of the work presented in our prior ISCA paper [4]. This work, in turn, rests upon an enormous body of work on memory consistency starting with Lamport's [14] characterization of sequential consistency in 1979. We continue to be heavily influenced by the CRF memory model [18], which reduces memory consistency to a set of primitive instructions which can be composed in different ways to obtain particular memory models. In common with CRF and UMM [20], we maintain a clear separation of local and global aspects of the model and capture reordering constraints in a simple table. However, the prior models take an architectural view of memory consistency (using caches in CRF and buffers in UMM). Of graph-based approaches to memory consistency, our technique has the greatest in common with TSOtool [9]. We avoid some of the worst problems with undecidability [3] by establishing a clear mapping between a Load L and its source $source(L)$, something which cannot in general be done simply by observing the values read and written—many Stores may write the same value to the same location. This gives us something similar to the data independence of Qadeer [16].

For programmers, the compiler and runtime can have an enormous influence on memory model guarantees. One hope of transactional memory is to provide an easier-to-use and yet more-efficient programming model than was possible using locks. However, data races are still possible in a transactional setting if shared data is manipulated outside a transaction. The idea of properly synchronized programs [2] will continue to be relevant, albeit under simplified assumptions. The community is just beginning to formulate transactional consistency protocols comparable to release consistency [7,13].

7 Conclusions and future work

In this paper we have built upon the Store Atomicity framework of [4] to examine the behavior of memory models with an atomic memory and atomic memory transactions. These models generalize SC to a setting in which there are atomic memory transactions, and 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.

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 argued that violations of Store Atomicity indicate violations of serializability. A realistic model of transactional memory requires the use of speculation, and we argued that it was not necessary to roll back whole transactions when conflict occurs; instead, we can roll back any Load which breaks a cycle, along with the instructions which depend upon it. In practice, of course, it may be necessary to roll back a Store operation in order to make a new value available to be loaded. An interesting aspect of transactional execution is that there is usually a *choice* in this matter—when transactions conflict, either one of them can be rolled back.

Better understanding of transactional serialization: Store Atomicity is a property that captures which instructions must be ordered in *any* serialization of an execution. In this respect, our semantics for transactional memory are not yet completely satisfactory: in practice many implementations interleave the instructions of multiple transactions without harm. The conditions outlined in Section 3.5 are a starting point, but further refinement of these conditions is undoubtedly possible.

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 [9] have already demonstrated their effectiveness in this area. Techniques similar to those described here (most notably routing inter-thread dependencies through Trans and Commit operations) have been suggested for checking transactional memory models. Similarly, it would not be difficult to adapt the techniques of UMM [20] to perform exhaustive model checking in a transactional setting.

Reference specification of a computer family: It is worthwhile to write an ISA specification which permits maximum flexibility in implementation and yet provides an easy to understand memory model. It is our hope that transactional models are simpler to understand, particularly for programmers. It has been conjectured [8] that transactional techniques (particularly batched updates) can scale well even when a relatively strong memory model is chosen. It remains to be seen how well these claims stand up on very large systems (those with tens, hundreds, or even thousands of multi-core CPUs).

Acknowledgement

Krste Asanovic played a vital role in targeting our work to computer architects. Our work has benefited greatly from numerous comments and conversations over the years. For our treatment of transactions in particular we would like to thank Maurice Herlihy, Victor Luchangco, Yossi Lev, and the rest of the Scalable Synchronization Research Group at Sun Microsystem Laboratories. Jan-Willem Maessen would like to thank the Fortress team. Arvind was supported in part by the IBM PERCS project. Both projects were funded in part by the DARPA HPCS program.

References

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

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

[3] Rajeev Alur, Ken McMillan, and Doron 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.

[4] Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. In *ISCA '06: Proceedings of the 33rd annual International Symposium on Computer Architecture*, 2006.

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

[6] Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual.* January 2000.

[7] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John 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.

[8] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, and Juin-Yeu Joseph 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.

[10] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[11] Maurice Herlihy and J. Eliot 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.

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

[13] P. Keleher, A.L. 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.

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

[15] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for A New Family of RISC Processors.* Morgan Kaufmann, 1994.

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

[17] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[18] Xiaowei Shen, Arvind, and Larry 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.

[19] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual (Version 9).* Prentice-Hall, 1994.

[20] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. UMM: an operational memory model specification framework with integrated model checking capability. *Concurr. Comput. : Pract. Exper.*, 17(5-6):465–487, 2005.