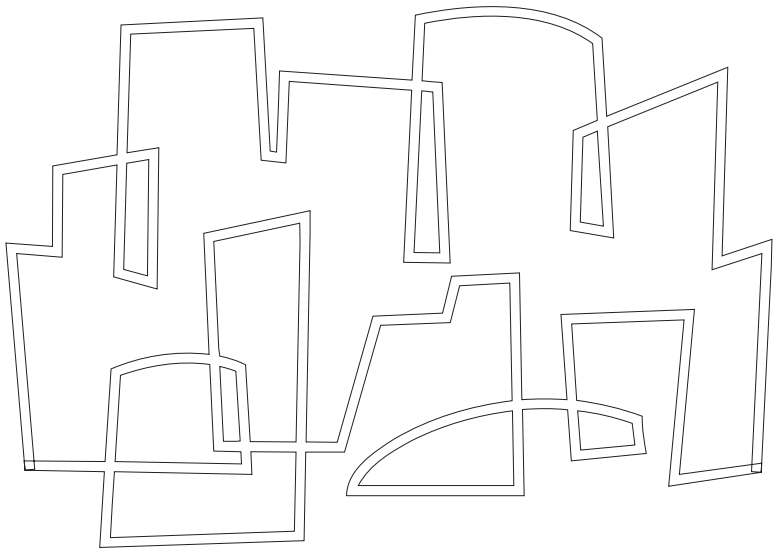
Defining Store Atomicity or Memory Operations without Memory

Computation Structures Group
Memo 477

July 19, 2004

Arvind (MIT), Jan-Willem Maessen (Sun)

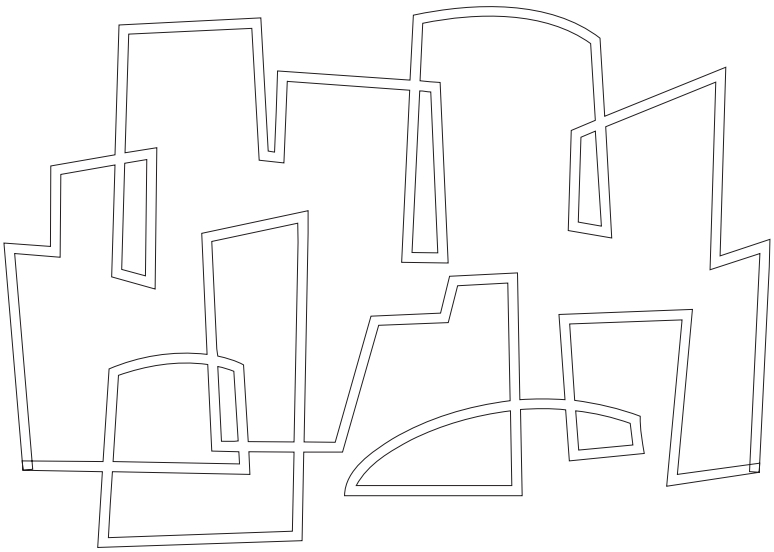


The Stata Center, 32 Vassar Street, Cambridge, Massachusetts 02139

Defining Store Atomicity or Memory Operations without Memory

Computation Structures Group
Memo 477

Arvind (MIT), Jan-Willem Maessen (Sun)



Defining Store Atomicity

or Memory Operations without Memory

Abstract

We characterize atomicity based on observations by Load and Store instructions; a Load observes the Store that supplies its data, and a Store observes itself. A memory model has *atomic stores* if, when two observers observe two stores to the same location to be in a particular order, that ordering relationship is respected by all observers. We capture program execution (including instruction reordering, aliasing, and branches) as a directed acyclic graph. There is no explicit memory—instead, in the execution graph a Load is linked to the Store which provides its value. We show that to maintain Store atomicity, Store-Store edges must be introduced whenever a Store-Load edge reveals a new ordering relationship. Some actions of cache coherence protocols can be understood as implementation of these edges. Store-Store edges have subtle implications for the correct implementation of memory fences. They also provide insight into the implementation requirements for synchronization primitives on machines with non-atomic Stores.

1 Introduction

A memory model specifies the values which can be obtained by each Load instruction in a program. In general, each such choice affects the values available to other instructions, including future Load operations. Precise and yet abstract specification of memory models has persisted as a problem both in computer systems and programming languages. It is not uncommon to bring in “language lawyers” to argue whether the behavior of a multithreaded program written in the

Java™ programming language conforms to the language specification [20, 9]. The situation in this regard is no better for multiprocessors built out of modern microprocessors.

Store atomicity (or a lack of it) is at the heart of the memory model definitional problem. Atomicity is typically defined by saying that, at any given point in time, every thread either sees a particular Store, or no thread sees it. This definition is not so difficult to understand if one imagines a system without caches. In such a system the memory model is defined using an explicit global store (a mapping from addresses to values). This inevitably serializes memory accesses at least to a particular memory address and makes the notion of "Store completion" precise and clear. However, we know that any multiprocessor, multithreaded computer system is actually distributed—there isn't a well-defined notion of simultaneity of events in different threads. Even within a thread, vagaries of compiler optimization and of instruction reordering by the processor make it difficult to pin down a well-founded notion of time.

In order to free the memory model from an explicit notion of time, we propose a *memoryless memory model* based on describing executions as directed acyclic graphs. Our graphs incorporate a complete processor semantics, including data and control flow. Our graphs also incorporate instruction reordering (or equivalently, compiler optimization) as specified by reordering axioms. We give a full treatment of all operations, including branches (with speculation) and aliasing. In doing so, we capture ordering constraints which have been widely ignored in prior formalizations of memory models. These include data dependencies between Stores and Loads and aliasing constraints among memory operations.

Between threads, the graph structure reflects the asynchronous nature of multithreaded computation. A inter-thread Store-Load edge is introduced whenever a Load observes a Store in another thread. We show that a memory model with store atomicity also requires the introduction of inter-thread Store-Store edges to enforce atomicity. Except for these edges (introduced by the memory model), graphs are constructed in a purely local fashion. The memory model treats inter-thread communication exactly as it would treat data dependency within the same thread.

The main contributions of this paper are a simple definition of store atomicity for both hard-

ware and software systems, and a method of maintaining store atomicity as a multithreaded computation unfolds. This combination of declarative and operational specifications makes it easy to see that 1. our definition is realizable and 2. it can be used to understand how various implementations in the past have enforced atomicity. This work may also lead to some new ways of enforcing atomicity, especially in large systems with coarse-grained synchronization primitives and minimal global cache coherence.

Paper organization: We begin with an informal discussion of the issues associated with store atomicity using some simple examples. We also motivate our formal definition and offer an outline of a solution to maintaining store atomicity as computation unfolds (Section 2). In Section 2.5 we connect this definition of atomicity with typical memory consistency protocols. In Section 3 we define execution graphs and give a formal definition of the $\sqsubset$ ordering, which is needed to define atomicity. We discuss some properties of our model in Section 3.5. In Section 4 we discuss related work. We offer our conclusions in Section 5.

2 Informal Definition of Store Atomicity

2.1 The Problem

In the absence of atomicity, we can obtain executions such as the following:

$\mathrm{Ld}\ x$ **= 6**	$\mathrm{Ld}\ x$ **= 7**	$\mathrm{St}\ x, 6$	$\mathrm{St}\ x, 7$
$\mathrm{Fence}_{\mathrm{LL}} x, x$	$\mathrm{Fence}_{\mathrm{LL}} x, x$		
$\mathrm{Ld}\ x$ **= 7**	$\mathrm{Ld}\ x$ **= 6?**		
$\mathrm{Fence}_{\mathrm{LL}} x, x$			
$\mathrm{Ld}\ x$ **= 6?**			

(2.1)

In this example there are two independent (and hence, unordered) Stores to x from two different threads. In the absence of atomicity, each Load is free to obtain its result from either one; the result is that the apparent value of x can alternate between the two, and that different threads can see the stores in an inconsistent order. (The boldface after each Load indicates the value loaded. A question mark indicates a questionable result, which should not happen.)

Suppose the first thread observes first 6, then 7. Having done so, it should no longer be

$a, x, y, z \in$ **Address** $= \mathbb{N}$

$r \in$ **Register** ::= $r_1 \ldots r_n$

$c \in$ **Constant** ::= $\mathbb{Z}$

Instruction ::= $r_d = c$
| $r_d = r_a + r_b \mid \ldots$
| $\mathrm{Br}\ r_c\ l$
| $r_d = \mathrm{Ld}\ r_a$
| $\mathrm{St}\ r_a, r_v$
| $\mathrm{Fence}_{\mathrm{LL}}\ r_x, r_y \mid \mathrm{Fence}_{\mathrm{LS}}\ r_x, r_y \mid \mathrm{Fence}_{\mathrm{SL}}\ r_x, r_y \mid \mathrm{Fence}_{\mathrm{SS}}\ r_x, r_y$

$l \in$ **Instruction-Label** ::= $l_1 \mid l_2 \mid ...$

Thread ::= a sequence of labeled instructions

Figure 1: Syntax of source programs

legal for the first thread to once again observe 6. Similarly, the second thread, after the first two observations of the first thread, should be able to load respectively, 6 followed by 6, or 6 followed by 7 or 7 followed by 7 but not 7 followed by 6—the observed ordering must be consistent across all threads.

2.2 Preliminaries

In this paper we consider programs written in the simple assembly language described in Figure 1. For readability, we take a few liberties with this syntax. We assume there are enough registers—none of our examples requires more than four or five in total. We omit the register destination of a Load when it is not subsequently used. The only operation which takes a non-register operand is the constant move $r_d = c$; however, for conciseness we use constant operands wherever it is convenient to do so. Finally, we take x and y to be distinct addresses, store a distinct value in each Store operation, and assume that there are no other Stores to the locations in question.

Within a thread, instructions may be freely reordered (modulo data dependency) with a few exceptions, summarized in Figure 2. Branch operations are ordered, and stores cannot occur before a preceding branch. Stores to the same address must occur in order, and Loads to the same address may not pass them.

$2^{nd} \rightarrow$ $1^{st} \downarrow$	$+, c$	Br	Ld y	St y, w	$\text{Fence}_{\text{R}-}$ y, w	$\text{Fence}_{\text{W}-}$ y, w
$+, c$	*dep.*	*dep.*	*dep.*	*dep.*	*dep.*	*dep.*
Br		X		X		
Ld x	*dep.*	*dep.*	*dep.*	$x \neq y$	$x \neq y$	*dep.*
St x, v			$x \neq y$	$x \neq y$		$x \neq y$
$\text{Fence}_{-\text{R}}$ v, x			$x \neq y$			
$\text{Fence}_{-\text{W}}$ v, x				$x \neq y$		

Figure 2: Reordering Axioms. The table specifies the permitted reordering of instructions. The first instruction in program order is listed on the left, the second is listed along the top. Blank entries may always be reordered; no data dependency can exist. Entries marked with an 'X' indicate reordering is prohibited. Pairs marked *dep.* must respect data dependencies. Entries of the form $x \neq y$ must also respect data dependencies, and may be reordered only if the addresses x and y do not alias. Finally, note that register name dependencies (write-after-read and write-after-write) are not accounted for in this reordering table; our graph-based model does away with them.

To control reordering, we make use of fine-grained Fence operations in the style of CRF [19, 16]. In Example 2.1, the Loads in the first two threads are separated by a Load/Load Fence—if these Fences are removed, the Loads can be freely reordered and we learn nothing about the relative order of the Store operations in the remaining two threads.

2.3 The interaction of multiple memory locations

Example 2.1 involved only a single memory location, x. But it must be possible to order memory accesses to different locations, or a Fence operation will have very little meaning. For example:

$$\begin{array}{ll} \text{St } x, 1 & \text{Ld } y \text{ } \mathbf{= 3} \\ \text{St } x, 2 & \text{Fence}_{\text{LL}} y, x \\ \text{Fence}_{\text{SS}} x, y & \text{Ld } x \text{ } \mathbf{= 1?} \\ \text{St } y, 3 & \end{array} \tag{2.2}$$

Here, the Store-Load dependency y should prevent Ld x from seeing 1 (since it was overwritten by 2). It seems that a Fence must have a non local effect in the sense that the Stores it guards must be "completed" or made visible to other threads before those threads can observe anything from the post fence region.

However, to obtain ordering guarantees, Store-Store dependencies uncovered by observation

must be respected as well:

$$\begin{array}{ll} \mathrm{St}\ x, 1 & \mathrm{St}\ y, 2 \\ \mathrm{Fence_{SS}} x, y & \mathrm{Fence_{SS}} y, x \\ \mathrm{St}\ y, 3 & \mathrm{St}\ x, 4 \\ \mathrm{Ld}\ y\ \textbf{= 2} & \mathrm{Ld}\ x\ \textbf{= 1?} \end{array} \tag{2.3}$$

Here $\mathrm{Ld}\ y$ in the first thread proves that $\mathrm{St}\ y, 3$ occurs before $\mathrm{St}\ y, 2$. We expect the fences to imply that $\mathrm{St}\ x, 1$ occurs before $\mathrm{St}\ x, 4$, and thus that it is impossible for the $\mathrm{Ld}\ x$ to be 1.

2.4 Store Atomicity and its Enforcement

Atomicity may be defined rigorously, but informally, as follows:

> A memory model has *atomic stores* if, when two observers observe two stores to the same location to be in a particular order, that ordering relationship is respected by all observers.

To formalize this definition, we must define the notion of an observer and of ordering between observers.

Definition (Observer): The only way to observe a memory location is to either Store to it (in which case we observe the Store itself, *observee*$(S) = S$), or to Load from it (in which case we observe the Store S which supplied the value loaded, *observee*$(L) =$ *source*$(L) = S$).

For the moment, we assume a partial order $A \sqsubset B$ (read "A before B") between operations. In our semantics (Section 3) this instruction reordering is captured by generating a directed acyclic graph of dependencies between instructions.

Loads L and Stores S to a single location must comply with the following two axioms:

Axiom (Causality): $L \not\sqsubset$ *source*(L)

Axiom (Overwrite): $S_1 \sqsubset S_2 \sqsubset L \Rightarrow$ *source*$(L) \neq S_1$ (only a recent Store may be loaded.)

After a Load L has executed, the following properties must hold:

Definition (Dependency): *source*$(L) \sqsubset L$, that is, there is a dependency between the source of the data loaded and the Load itself.

Definition (Store Atomicity):

$$A \sqsubset B, \textit{address}(A) = \textit{address}(B),$$
$$\textit{observee}(A) \neq \textit{observee}(B) \Rightarrow \textit{observee}(A) \sqsubset \textit{observee}(B)$$

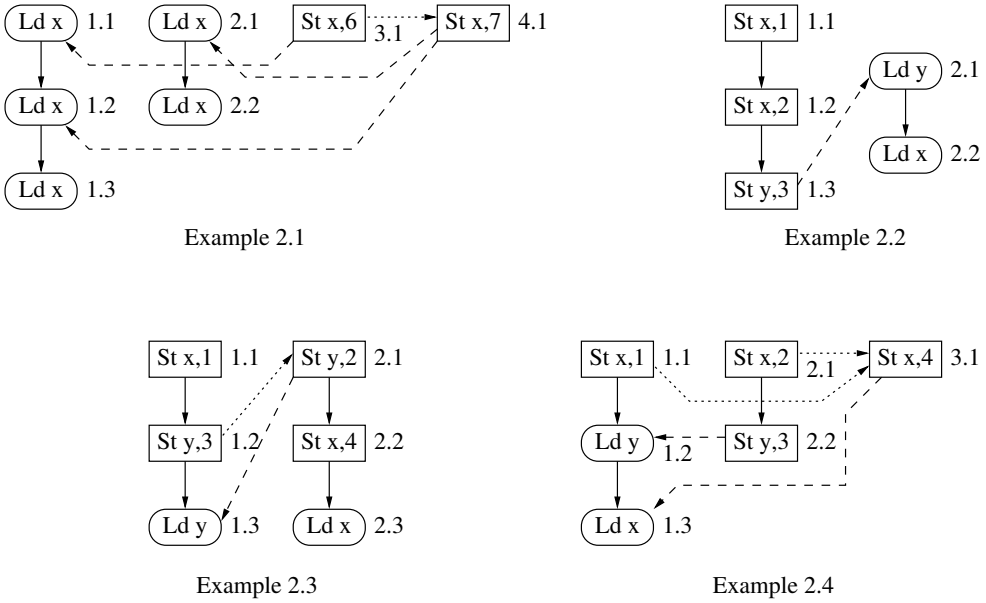


Figure 3: Load-Store graphs for the examples in the text. The solid lines result from the reordering semantics (including Fences). The dashed lines indicate Store-Load dependencies. The dotted line indicates a Store-Store edge.

In our semantics, when a Load observes a Store in another thread, we draw an edge from that Store to the observing Load. But this Store-Load edge together with other intra-thread data and control flow edges may not be sufficient to capture the ordering between the Stores as implied by the atomicity definition. Hence, in our semantics, when a Load observes a Store in another thread, we also draw a Store-Store edge (the dotted line) from every Store which precedes the Load to the observed Store. Coupled with the overwrite rule, these Store-Store edges enforce atomicity while keeping the graph acyclic. We explain this by drawing simplified graphs for the examples in this Section. (We erase the Fence instructions, but preserve the connections they make between Loads and Stores).

So in Example 2.1 in Figure 3, when the Load 1.2 observes the Store 4.1, we insert the edge from 3.1 (previously observed by 1.1) to 4.1. This edge, together with the overwrite rule, prevents 1.3 and 2.2 from observing 3.1. Similarly, in Example 2.3 the Load 1.3 observes the Store 2.1, and so we must add an edge from the prior Store 1.2 to 2.1. This will prevent the Load 2.3 from observing the Store 1.1 (since Load 2.2 is later).

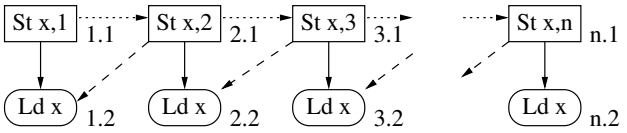


Figure 4: A chain of dependencies. A Load in the chain cannot observe any Store to its left due to the insertion of Store-Store edges by the atomicity rule.

Sometimes no Store-Store edges are required, as seen in the graph for Example 2.2. The following example demonstrates that more than one Store-Store edge may need to be drawn:

$$\begin{array}{lll} \mathrm{St}\ x, 1 & \mathrm{St}\ x, 2 & \mathrm{St}\ x, 4 \\ \mathrm{Fence}_{\mathrm{SL}} x, y & \mathrm{Fence}_{\mathrm{SS}} x, y & \\ \mathrm{Ld}\ y \ \textbf{= 3} & \mathrm{St}\ y, 3 & \\ \mathrm{Fence}_{\mathrm{LL}} y, x & & \\ \mathrm{Ld}\ x \ \textbf{= 4} & & \end{array} \tag{2.4}$$

Here, the Load of y creates a dependency between the second thread and the first thread. Thus, the Store to x in both threads occurs before the Load of x. When $\mathrm{St}\ x, 4$ is observed, we know it must occur after the other Stores to x, but we still do not know their relative order.

Adding Store-Store edges ensures that chains of observations are respected. For example, in Figure 2.4 we see a chain of stores and loads. The Store-Store edges guarantee that the rightmost Load in the chain cannot observe the leftmost Store (or any Store to its left).

2.5 Connection to implementation

In order to understand the implications of our atomicity condition for implementations, it is important to grasp the theoretical flexibility offered by our model:

1. A Store-Store edge orders two Stores. It is always safe to draw a Store-Store edge between two unordered Stores, as we prove in Section 3.5.

2. The Store-Store edges that must be drawn to preserve atomicity when a Load observes a Store in another thread can be drawn any time prior to the observation, i.e., the drawing of the Store-Load edge. In our model we draw Store-Store edges as lazily as possible, i.e., as late, but no later, than the Store-Load edge.

Implementations invariably draw Store-Store edges (i.e., order Stores) *much* earlier than the Loads that require them are performed. Furthermore, implementations often impose Store-Store edges not required by our semantics. Both of these actions are safe according to our model.

Implementations achieve the effect of a Store-Store edge on the same address within a thread simply by overwriting the old value in the cache or memory. Across threads, this effect is achieved by cache coherence; each Store causes invalidations of old copies in the other threads [2]. Keeping clean or read-only copies of the latest Stored value in multiple caches is safe because the coherence protocol guarantees the destruction of these copies when a Store-Store edge is drawn.

In the CRF model [19] the effect of a Store-Store edge is achieved via the "commit" operation, which forces a dirty value in a cache to be written to the global memory, displacing the previous value. A Load is guaranteed to see this new value after it performs the "reconcile" operation.

From the implementation point of view the role of Fences in inter-thread communication is the most difficult thing to understand. As shown in Example 2.3, when Stores to two different addresses x and y are separated by a Store-Store fence, any observation of address y can potentially affect the observations of address x. This effect cannot be achieved by merely overwriting values—x and y represent different locations. Given $\mathrm{Ld}\ y$ **= 2** (node 1.3 in Figure 3), node 2.1 must have overwritten the value of node 1.2. This much can be achieved by cache-coherence protocols. But any observation of x which comes after this event must be cognizant of the ordering between nodes 1.1 and 2.1.

Informally, the semantics of a store fence is often described as "all the stores before the fence must *complete* before the fence is discharged". This effect can be achieved by writing back all the dirty copies before a fence and indeed this is what most systems do. This non-local effect makes fences quite expensive to use in modern microprocessors. To further understand this cost of Fences (really of Store atomicity), imagine that the first thread in Example 2.3 has exclusive copies of x and y. Variable y will be written back only when another thread Stores to y but x has to be written back just to get past the Fence, even if no other thread ever requests x.

One way of alleviating this cost is to resort to coarse-grain synchronization primitives (e.g., locks) and non-atomic stores. In a "properly synchronized program" [3] stores commit just before leaving a locked region (or even later, after taking a lock [12]). The implementation underlying a model like Location Consistency [8] must either provide the necessary synchronization primitives, or have a strong enough memory model to permit them to be coded in terms of Load, Store, and Fence. For example, a model which respects only Store-Load dependencies allows synchronization between n threads in $O(n)$ space and time using the algorithm of Dijkstra [4].

3 Execution Graphs

In the remainder of this paper, we present a detailed semantics for an atomic memory in the sense of Section 2. We break program execution into three concurrent tasks:

1. Graph Generation: constructs the local dependency graph for a thread, based on the reordering semantics summarized in Figure 2. Graph generation suspends under some conditions and can only be resumed by the actions of Graph Execution or Memory Action.

2. Graph Execution: performs the computations represented by the generated graph. A node whose inputs are available is *resolved* to a value. This process suspends if there are no nodes which can be resolved, or if a Memory Action is required to resolve a node.

3. Memory Action: connects a Store instruction to a Load instruction. This process may also require the introduction of inter-thread Store-Store edges. Memory action is the only aspect of our semantics which affects the behavior of multiple threads.

After introducing the graph syntax, we describe of each of these tasks. In Section 3.5 we show that our procedure indeed generates a partial order and is consistent with properties discussed in Section 2.

3.1 Graph Syntax

An execution graph is a directed, acyclic graph which is built incrementally as execution proceeds. The rules for generating the graph are designed to preserve the atomicity property of the generated

$L \in$ **Execution Tag** ::= $\mathbb{N}$

$P \in$ **Thread Tag** ::= $P_1 \mid P_2 \mid ...$

$V \in$ **Value** ::= $\mathbb{Z} \mid$ *done*

$R \in$ **Node Tag** ::= $P.L$

$C \in$ **Control Edges** ::= $\{P.L_1, P.L_2, \ldots\}$

$W \in$ **Store-Load Edges** ::= $\{P_1.L_1, P_2.L_2, \ldots\}$

$O \in$ **Store-Store Edges** ::= $\{P_1.L_1, P_2.L_2, \ldots\}$

$$
\begin{array}{rcl}
\textbf{Node} & ::= & (c,\ R_a,\ V,\ \{\}) \\
& \mid & (+,\ R_a,\ R_b,\ V,\ \{\}) \\
& \mid & (\ldots,\ R_a,\ R_b,\ V,\ \{\}) \\
& \mid & (\mathrm{Br},\ R_c,\ V,\ C) \\
& \mid & (\mathrm{St}_1,\ R_a,\ V,\ \{\}) \\
& \mid & (\mathrm{St}_2,\ R_{\mathrm{St}_1},\ R_v,\ V,\ C,\ O) \\
& \mid & (\mathrm{Ld}_1,\ R_a,\ V,\ \{\}) \\
& \mid & (\mathrm{Ld}_2,\ R_{\mathrm{Ld}_1},\ R_g,\ V,\ C) \\
& \mid & (\mathrm{PreFenceL}_1,\ R_a,\ V,\ \{\}) \\
& \mid & (\mathrm{PreFenceL}_2,\ R_{PFL_1},\ V,\ C) \\
& \mid & (\mathrm{PreFenceS}_1,\ R_a,\ V,\ \{\}) \\
& \mid & (\mathrm{PreFenceS}_2,\ R_{PFSA},\ V,\ C) \\
& \mid & (\mathrm{PostFenceL}_1,\ R_b,\ V,\ C) \\
& \mid & (\mathrm{PostFenceL}_2,\ R_{\mathrm{PostFenceL}_1},\ R_{\mathrm{PreFenceX}_2},\ V,\ C) \\
& \mid & (\mathrm{PostFenceS}_1,\ R_b, V,\ C) \\
& \mid & (\mathrm{PostFenceS}_2,\ R_{\mathrm{PostFenceL}_1},\ R_{\mathrm{PreFenceX}_2},\ V,\ C)
\end{array}
$$

Figure 5: Graph node syntax

graph at all times. Each node in the graph is uniquely identified by a node tag, which is a pair of a thread name and an execution tag written $P.L$. Execution tags L (not to be confused with instruction labels in the source program) are integers that are assigned to each node in a thread in increasing program order. Nodes are connected by three kinds of directed edges: data edges, control edges, and memory edges. A data edge is used to indicate the passing of a value between two nodes. A control edge is used to indicate ordering within a thread of execution (due to branches or memory fences). A memory edge is either a Store-Load edge, used to indicate the flow of data from a Store to a Load, or a Store-Store edge, used to enforce atomicity.

Node syntax is summarized in Figure 5. Nodes contain an *operation* with one or more *operands* and a *value*. The final entry in each node is the set of control edges. Many instruc-

tions do not require control edges; this is indicated by the empty set $\{\}$ in the grammar. Note that instructions involving addresses are split into multiple phases (nodes) to resolve aliasing; this split-phase operation will be described in the Graph Generation process.

Each thread has an *execution state* consisting of a program counter (PC) to keep track of the current instruction, a node counter (NC) to keep track of the next available execution tag, and a register table (RT) to record which node is responsible for computing the value of each register. We update the mapping for register r by writing $RT[r] = L$. Because nodes are uniquely tagged, we use tags and nodes interchangeably in the semantics.

Two operators express the relationship between nodes: $<$ and $\sqsubset$. For $P.L_1$ and $P.L_2$ in the same thread P, $P.L_1 < P.L_2$ is read as "L_1 is before L_2 in program order in thread P." Program order is determined by integer comparison between the execution tags, i.e., $L_1 < L_2$. Nodes in different threads are not ordered by $<$.

The $\sqsubset$ operator indicates *execution order*. $P_1.L_1 \sqsubset P_2.L_2$ indicates that there is a path from $P_1.L_1$ to $P_2.L_2$ in the graph. Thus, $\sqsubset$ is simply transitive closure of the edge relation. Unlike $<$, $\sqsubset$ can be applied to nodes from different threads of execution. Execution order and program order are only loosely related. In particular, reordering may combine with interprocessor communication to result in a situation where $P.L \sqsubset P.L'$, but $P.L' < P.L$. We say in this case that these instructions have been *reordered*. In the following example the instructions in the first thread are reordered before execution:

$$\begin{array}{ll} \mathrm{Ld}\ x \ \textbf{= 3} & \mathrm{Ld}\ y \ \textbf{= 5} \\ \mathrm{St}\ y, 5 & Fence_{LS} y, x \\ & \mathrm{St}\ x, 3 \end{array} \tag{3.1}$$

Note that two or more threads must interact in order for reordering to be visible to the program in this way.

3.2 Graph Generation

To start graph generation for a thread we initialize the thread state:

1. Create n nodes (one for each register) with operation $r_i = 0$ and tag i.

2. Set $RT[r_i] = i$
3. Set PC to 0.
4. Set NC to n.

We initialize memory by constructing a separate *initialization thread* which traverses all of memory writing a 0 into each memory location. This thread contains no Fences or Loads.

During graph generation, an instruction is fetched at the current value of the PC and both NC and PC are incremented by one unless the instruction is a split-phase or branch. For Loads and Stores NC and PC are incremented by two and for Fences by 4. A branch resets the PC if it is taken. The value in a node is initially $\perp$.

We describe graph generation for each instruction as follows:

Constant move: $r_d = c$:

1. Create a node $(c,\ \perp,\ \{\})$ with tag $P.NC$.
2. Set $RT[r_d] = NC$.

Addition: $r_d = r_a + r_b$:

1. Create a node $(+,\ RT[r_a],\ RT[r_b],\ \perp,\ \{\})$ with tag P.NC.
2. Set $RT[r_d] = NC$.

Branch: $\mathrm{Br}\ r_c\ l$ branches to the static target l if r_c is less than or equal to zero:

1. Create a node $(\mathrm{Br},\ RT[r_c],\ \perp,\ \mathit{PriorBr}(P.NC))$ with tag P.NC,
2. Block on this node until $RT[r_c]$ is resolved by graph execution and memory action.
3. Once resolved, if $r_c \leq 0$, then $PC = l$, otherwise PC is unchanged.

Loads and Stores are split into two phases, the *address phase* and the *operation phase*, such that the latter is dependent upon the former. In order to guarantee that memory action respects local execution order, a Load must stall until the address phase of preceding Store and $\mathrm{Fence}_{\mathrm{XL}}$ operations are complete. Similarly, a Store must stall until the address steps of preceding Load, Store, or $\mathrm{Fence}_{\mathrm{XS}}$ operations are complete. Control edges are added to the operation phase to indicate any aliasing occurs.

$$
\begin{aligned}
\mathit{PriorBr}(P.L) &= \{P.L' < P.L \mid \mathit{operation}(P.L') = \mathrm{Br}\} \\
\\
\mathit{PriorLd}_1(P.L) &= \{P.L' < P.L \mid \mathit{operation}(P.L') = \mathrm{Ld}_1\} \\
\mathit{PriorSt}_1(P.L) &= \{P.L' < P.L \mid \mathit{operation}(P.L') = \mathrm{St}_1\} \\
\mathit{PriorLdSt}_1(P.L) &= \mathit{PriorLd}_1(P.L) \cup \mathit{PriorSt}_1(P.L) \\
\mathit{PriorPostFenceL}_1(P.L) &= \{P.L' < P.L \mid \mathit{operation}(P.L') = \mathrm{PostFenceL}_1\} \\
\mathit{PriorPostFenceS}_1(P.L) &= \{P.L' < P.L \mid \mathit{operation}(P.L') = \mathrm{PostFenceS}_1\} \\
\\
\mathit{PriorLd}_2(P.L) &= \{P.L' < P.L \mid \mathit{operation}(P.L') = \mathrm{Ld}_2 \wedge \mathit{address}(P.L') = \mathit{address}(P.L)\} \\
\mathit{PriorSt}_2(P.L) &= \{P.L' < P.L \mid \mathit{operation}(P.L') = \mathrm{St}_2 \wedge \mathit{address}(P.L') = \mathit{address}(P.L)\} \\
\mathit{PriorLdSt}_2(P.L) &= \mathit{PriorLd}_2(P.L) \cup \mathit{PriorSt}_2(P.L) \\
\mathit{PriorPostFenceL}_2(P.L) &= \{P.L' < P.L \mid \mathit{operation}(P.L') = \mathrm{PostFenceL}_2 \wedge \mathit{address}(P.L') = \mathit{address}(P.L)\} \\
\mathit{PriorPostFenceS}_2(P.L) &= \{P.L' < P.L \mid \mathit{operation}(P.L') = \mathrm{PostFenceS}_2 \wedge \mathit{address}(P.L') = \mathit{address}(P.L)\}
\end{aligned}
$$

Figure 6: Prior functions for computing control dependencies. We write $\mathit{operation}(P.L)$ and $\mathit{address}(P.L)$ for the operation and address parts of node $P.L$ respectively.

To simplify the definition of control dependencies, we define *prior* functions on a node tag $P.L$, as shown in Figure 6.

Store: $\mathrm{St}\ r_a, r_v$:

1. Create a node $(\mathrm{St}_1,\ RT[r_a],\ \bot,\ C)$ with tag $T = P.NC - 1$, where $C = \mathit{PriorLdSt}_1(T) \cup \mathit{PriorPostFenceS}_1(T)$
2. Block until node T is resolved.
3. Create a node $(\mathrm{St}_2,\ T,\ RT[r_v],\ \bot,\ C')$ with tag $P.NC$, where $C' = \mathit{PriorLdSt}_2(T) \cup \mathit{PriorPostFenceS}_2(T) \cup \mathit{PriorBr}(T)$

Load: $r_d = \mathrm{Ld}\ r_a$:

1. Create a node $(\mathrm{Ld}_1,\ RT[r_a],\ \bot,\ C)$ with tag $T = P.NC - 1$, where $C = \mathit{PriorSt}_1(T) \cup \mathit{PriorPostFenceL}_1(T)$.
2. Set $RT[r_d] = NC$.
3. Block until node T is resolved.
4. Create a node $(\mathrm{Ld}_2,\ T,\ \bot,\ \bot,\ C')$ with tag $P.NC$, where $C' = \mathit{PriorSt}_2(T) \cup \mathit{PriorPostFenceL}_2(T)$. Note that the memory edge L_g is initially absent ($\bot$); it must be filled in by a memory action.

The fence instruction imposes an ordering constraint between memory operations involving the pre-address r_l and the post-address r_s. In addition to the split phases described above, we split a Fence instruction into two halves with a control dependency between them, as shown in Figure 7. The PreFence handles ordering with respect to prior instructions; the PostFence handles ordering with respect to subsequent instructions. Each half of the fence operation is split-phase,

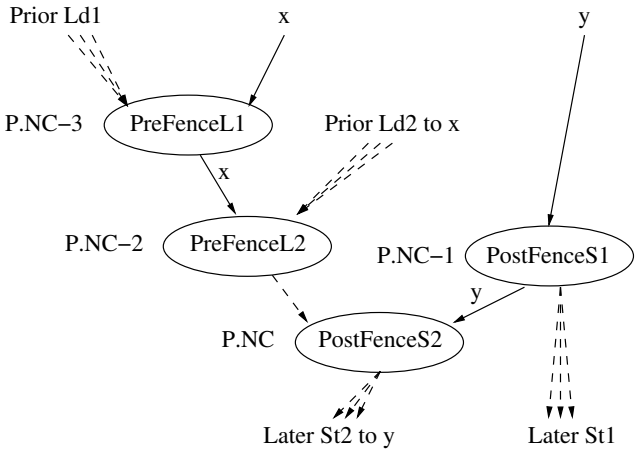


Figure 7: Graph unfolding for $\mathrm{Fence_{LS}}\ r_l, r_s$. Dotted edges indicate control arcs. Control arcs to $\mathrm{PreFenceL_2}$ are determined by x. Control arcs from $\mathrm{PostFenceS_2}$ are determined by y.

yielding a total of four graph nodes per fence instruction. The graph and associated constructs for a Load-Store fence ($\mathrm{Fence_{LS}}$) will be shown; other fences are similar:

Fence: $\mathrm{Fence_{LS}}\ r_l, r_s$:

1. Create a node $(\mathrm{PreFenceL_1},\ RT[r_l],\ \perp,\ C)$ with tag $T = P.NC - 3$, where $C = \mathit{PriorLd}_1(T)$
2. Block until node $P.NC - 3$ is resolved.
3. Create a node $(\mathrm{PreFenceL_2},\ L,\ \perp,\ C')$ with tag $P.NC - 2$, where $C' = \mathit{PriorLd}_2(T)$ to node $L + 1$.
4. Create a node $(\mathrm{PostFenceS_1},\ RT[r_s],\ \perp,\ \{\})$ with tag $P.NC - 1$.
5. Create a node $(\mathrm{PostFenceS_2},\ P.NC - 1,\ \perp,\ \{P.NC - 2\})$ with tag $P.NC$.

Simplifying Alias Dependencies: Our graph generation process introduces many redundant edges. For example, every branch operation depends on the outcome of all previous branches in the same thread; this means that there is a total order on branches in a single thread. As a result, we need only insert a dependency between a branch instruction and the most recent branch in the program. We can capture this by redefining *PriorBr* as follows (Here max selects the maximal element of the set with respect to $<$):

$$\mathit{PriorBr}(P.L) = max\{P.L' < P.L \mid \mathit{operation}(P.L') = \mathrm{Br}\}$$

St_1 operations are totally ordered in a similar fashion. Furthermore, St_1 operations are also dependent on prior Ld_1 operations as well. When considering both sets of dependencies, we need only consider Ld_1 operations since the previous St_1:

$$\begin{aligned}\textit{PriorLdSt}_1(P.L) = \{P.L' < P.L \mid \textit{operation}(P.L') = \mathrm{Ld}_1 \wedge \textit{PriorSt}_1(P.L) < P.L'\} \\ \cup \{\textit{PriorSt}_1(P.L)\}\end{aligned}$$

Finally, if we consider Ld_1 operations alone (as we must for $\mathrm{PreFence L}_1$), we must look back in time to the last Load. That Load will be dependent on a prior St_1, which will depend on prior Ld_1 operations. The important observation here is that the St_1 operation of interest is not the most recent one, but the one which was most recent when the last Store occurred (in program order).

$$\begin{aligned}&\textit{PriorLd}_1(P.L) = \{P.L' < P.L \mid \textit{operation}(P.L') = \mathrm{Ld}_1 \wedge \textit{PriorSt}_1(P.L_m) < P.L'\} \\ &\quad where\ P.L_m = max\{P.L' < P.L \mid \textit{operation}(P.L') = \mathrm{Ld}_1\}\end{aligned}$$

Similar arguments apply to the *PriorSt*$_2$, *PriorLdSt*$_2$, and *PriorLd*$_2$ functions.

3.3 Graph Execution

After graph generation, a node has the value $\perp$. When all its inputs (including its control edges) become available, the value of node $P.L$ is *resolved* based on the values of its data inputs $P.L_i$ as follows:

Constant move: A constant move of constant c has no inputs and no control dependencies and can execute immediately. We set *value*$(P.L) = c$.

Addition: *value*$(P.L) =$ *value*$(P.L_a) +$ *value*$(P.L_b)$.

Branch: When input node $P.L_c$ has been resolved, *value*$(P.L) =$ *done*.

St$_1$: *value*$(P.L) =$ *value*$(P.L_a)$. Similar rules apply to the other address phase operations Ld_1, $\mathrm{PreFence L}_1$, $\mathrm{PreFence S}_1$, $\mathrm{PostFence L}_1$, $\mathrm{PostFence S}_1$.

St$_2$: *value*$(P.L) =$ *value*$(P.L_v)$. Note that the Store-Store edges need not be computed yet. The value is only used by Load operations; no instruction is directly data dependent upon a Store.

Ld$_2$: *value*$(P.L) =$ *value*$(P'.L_g)$. Note that $P'.L_g$ must be instantiated by the memory action rules. This is the only non-local data dependency in instruction execution.

PreFenceL$_2$: *value*$(P.L) =$ *done*.

Similar execution rules apply to $\mathrm{PreFenceS}_2$, $\mathrm{PostFenceL}_2$, and $\mathrm{PostFenceS}_2$. Pre- and Post-Fence instructions are distinguished by their data dependencies.

3.4 Memory Action

The memory action phase is guided by the rules identified in Section 2. The phase matches each Load operation L to a corresponding Store instruction S as follows:

1. Identify a Store S to the same address as L satisfying:
   Causality:: $L \not\sqsubset S$
   Overwrite:: $S' \sqsubset L \Rightarrow S \not\sqsubset S'$
2. To preserve atomicity, for all $S' \sqsubset L$ where $S' \neq S$, add Store-Store edge $S' \sqsubset S$.
3. Then add Store-Load edge *source*$(L) = S$

3.5 Properties

There is a standard algorithm to generate all possible executions of a given program. Note that, with the exception of Ld operations, every instruction in a thread may be resolved purely locally. Therefore, we can interleave a step of unfolding and execution with a single step of Load resolution as follows:

1. Perform the graph generation and execution steps on every thread until no further unfolding or execution is possible. At this point, all threads will be awaiting the outcome of some Load instruction.
2. Generate a new execution for each possible valid combination of a Ld instruction and a corresponding St instruction.
3. For each such execution, return to the first step.

To show that execution graphs are well founded, we prove that they must be acyclic, and that they cannot get stuck—one of graph generation, graph execution, and memory action is always possible. We also show that execution graphs are linearizable [15]. We can obtain sequentially consistent execution [13] by placing an appropriate Fence between each adjacent pair of memory operations.

3.1 Theorem (Acyclic) An execution graph is acyclic.

Proof: Causality ensures that a Store-Load edge does not introduce a cycle. A Store-Store edge $S' \sqsubset S$ introduces a cycle only if $S \sqsubset S' \sqsubset L$. This would violate the overwrite rule. All other edges connect to a newly created node later in program order.

3.2 Lemma (Load Progress) It is possible for memory action to find a Store to match any Load operation in an arbitrary acyclic program graph with an initialization thread.

Proof: Call the Load L. The initialization thread guarantees there will be at least one Store S to the same location. If $S \sqsubset L$, there is at least one most recent store; this store will be legal according to the overwrite rule. Otherwise we choose an unrelated S.

3.3 Corollary (Store-Store Insertion) If $S' \not\sqsubset S$ are Stores to the same location, it is always valid to insert the Store-Store edge $S \sqsubset S'$.

Proof: Inserting the extra edge will not prevent Load Progress; Store-Store edges are ignored by Graph Execution.

3.4 Theorem (Progress) It is always possible to make progress in one of the three phases.

Proof: Given load progress and acyclicity, progress follows from the definition of Graph Execution, and in particular the fact that store-store edges are ignored.

3.5 Theorem (Linearizability) Executions are *linearizable*: Nodes can be totally ordered respecting $\sqsubset$ such that each Load receives the value of the immediately preceding Store to the same location.

Proof: The only thing preventing such a linearization would be a path from *source*(L) to L containing another store S to the same location. This contradicts the overwrite rule. Edges which do not lie on such a path can be ordered immediately before S or after L.

We can reduce any execution graph to a *Load-Store* graph, similar to the ones shown in Figure 3, as follows:

1. **Completion:** Transitively close the graph.

2. **Erasure:** Erase all nodes except Ld_2 and St_2 nodes.

3. **Minimization:** Remove all redundant edges, preferring data flow edges to control edges to memory edges. For example, in Example 2.3 in Figure 3 the Store-Store edge from 1.1 to 2.1 is redundant (given the edges from 1.1 to 1.2 and 1.2 to 2.1) and has been erased.

4 Related work

The literature on memory models (see the tutorial by Adve and Gharachorloo [2] for an introduction) is a study in the tension between elegant, simple specification and efficient implementation. As Adve et. al. [1] note, it is sufficient for a compiler or an architecture to *appear* to uphold the rules of a particular memory model; the underlying coherence mechanism is unimportant.

Sequential Consistency [13] remains the standard against which memory models are compared; it is arguably [10] the only model which is widely understood by programmers. The idea of properly synchronized programs [3] and of release consistency [8, 12] is to present a programming model which, if obeyed, appears to be sequentially consistent, even with a comparatively weak underlying memory system.

An alternative is to attempt to provide a weak but easily-understood memory model, as in location consistency [6, 7]. The CRF memory model [19] takes this to its logical conclusion, by presenting a model rich enough to capture other consistency models, including SC, location consistency, and a proposed memory semantics for the Java programming language [16].

The struggle to formalize a memory model for the Java virtual machine is a case study in the difficulty of writing a model which permits aggressive compiler optimization. The initial model [9, 14] had a number of shortcomings, described in detail by Pugh [18]. Five years of discussion [11] have finally resulted in a preliminary specification [17] which makes use of at least five different ordering relations in defining the memory model.

Like the present work, the computation-centric memory models of Frigo and Luchangco [6,

5, 15] use DAGs to capture ordering dependencies between memory operations. However, no semantics is given for graph generation. 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.

5 Conclusions

The main contributions of this paper are the declarative specification of atomicity in the presence of instruction reordering and the necessary conditions for maintaining atomicity operationally. The original motivation for this work came from two different sources:

1. A desire to find a simple operational model—a way of enumerating all the legal behaviors of a program—buried in the declarative specification of the memory model for the Java Programming Language [17]. This problem is quite difficult in view of the commonly accepted compiler optimizations that permit reordering.

2. A desire to define precisely the property of store atomicity in the presence of caches. For example, in the CRF memory model [19] a clear distinction is made between the global "commit" operator and a weaker version which commits data to a specific processor. The latter is needed to define non-atomic behaviors. In spite of this clear difference, a definition of atomicity was never given.

In an attempt to give an operational semantics for a simple multithreaded language with instruction reordering and fences we discovered that memory actions could not be defined properly without a declarative specification of atomicity. This fact would not have emerged had we used a global memory in our operational model. The resulting model of atomicity has proved surprisingly easy to reason about.

It came as a surprise to us that the minimal set of dependencies required to implement an atomic memory with Fences gives us a model that is strong enough to be linearizable, i.e., a model in which all Loads and Stores to a location can be totally ordered. It may be that linearizability

is the property we want from a low-level memory model, just as Sequential Consistency is the property programmers want.

We also gained some insight into the behavior of fences—they're not as local as we previously thought. The fact that fences are expensive to implement provides us motivation to look for alternatives. A system with non-atomic stores but with suitable synchronization primitives that enforce atomicity at a transaction level may prove to be more useful.

References

[1] S. Adve, V. Pai, and P. Ranganathan. Recent Advances in Memory Consistency Models for Hardware Shared Memory Systems. *Proceedings of the IEEE*, 87(3), March 1999.

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

[3] 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.

[4] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

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

[6] M. Frigo and V. Luchangco. Computation-centric memory models. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, June/July 1998.

[7] 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.

[8] 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.

[9] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Menlo Park, CA, 1996.

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

[11] Java memory model mailing list. http://www.cs.umd.edu/ pugh/java/memoryModel.

[12] 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.

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

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Menlo Park, CA, 1997.

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

[16] 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. ACM SIGPLAN. Also available as MIT LCS Computation Structures Group Memo 428.

[17] J. Manson, W. Pugh, and S. Adve. The unified memory model proposal for Java. http://www.cs.umd.edu/ pugh/java/memoryModel/unifiedProposal/, Apr. 2004.

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

[19] 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.

[20] G. L. Steele Jr. Memory models for programming languages. Invited talk, Harvard University., Mar. 2004.