To appear in PACT 2017

Weak Memory Models: Balancing Definitional Simplicity and Implementation Flexibility

Sizhuo Zhang Muralidaran Vijayaraghavan Arvind
{szzhang, vmurali, arvind}@csail.mit.edu
MIT CSAIL

Abstract

The memory model for RISC-V, a newly developed open source ISA, has not been finalized yet and thus, offers an opportunity to evaluate existing memory models. We believe RISC-V should not adopt the memory models of POWER or ARM, because their axiomatic and operational definitions are too complicated. We propose two new weak memory models: WMM and WMM-S, which balance definitional simplicity and implementation flexibility differently. Both allow all instruction reorderings except overtaking of loads by a store. We show that this restriction has little impact on performance and it considerably simplifies operational definitions. It also rules out the out-of-thin-air problem that plagues many definitions. WMM is simple (it is similar to the Alpha memory model), but it disallows behaviors arising due to shared store buffers and shared write-through caches (which are seen in POWER processors). WMM-S, on the other hand, is more complex and allows these behaviors. We give the operational definitions of both models using Instantaneous Instruction Execution *(*I^{2}E*), which has been used in the definitions of SC and TSO. We also show how both models can be implemented using conventional cache-coherent memory systems and out-of-order processors, and encompasses the behaviors of most known optimizations.*

1. Introduction

A memory model for an ISA is the specification of all legal multithreaded program behaviors. If microarchitectural changes conform to the memory model, software remains compatible. Leaving the meanings of corner cases to be implementation dependent makes the task of proving the correctness of multithreaded programs, microarchitectures and cache protocols untenable. While strong memory models like SC and SPARC/Intel-TSO are well understood, weak memory models of commercial ISAs like ARM and POWER are driven too much by microarchitectural details, and inadequately documented by manufacturers. For example, the memory model in the POWER ISA manual [33] is "defined" as reorderings of events, and an event refers to performing an instruction with respect to a processor. While reorderings capture some properties of memory models, it does not specify the result of each load, which is the most important information to understand program behaviors. This forces the researchers to formalize these weak memory models by empirically determining the allowed/disallowed behaviors of commercial processors and then constructing models to fit these observations [8, 10, 7, 39, 11, 49, 48, 9, 25].

The newly designed open-source RISC-V ISA [1] offers a unique opportunity to reverse this trend by giving a clear definition with understandable implications for implementations. The RISC-V ISA manual only states that its memory model is weak in the sense that it allows a variety of instruction reorderings [58]. However, so far no detailed definition has been provided, and the memory model is not fixed yet.

In this paper we propose two weak memory models for RISC-V: WMM and WMM-S, which balance definitional simplicity and implementation flexibility differently. The difference between the two models is regarding store atomicity, which is often classified into the following three types [38]:

- *Single-copy atomic*: a store becomes visible to all processors at the same time, e.g., in SC.
- *Multi-copy atomic*: a store becomes visible to the issuing processor before it is advertised simultaneously to all other processors, e.g., in TSO and Alpha [4].
- *Non-atomic* (or non-multi-copy-atomic): a store becomes visible to different processors at different times, e.g., in POWER and ARM.

Multi-copy atomic stores are caused by the store buffer or write-through cache that is private to each processor. Non-atomic stores arise (mostly) because of the sharing of a store buffer or a write-through cache by multiple processors, and such stores considerably complicate the formal definitions [49, 25]. WMM is an Alpha-like memory model which permits only multi-copy atomic stores and thus, prohibits shared store buffers or shared write-through caches in implementations. WMM-S is an ARM/POWER-like memory model which admits non-atomic stores. We will present the implementations of both models using out-of-order (OOO) processors and cache-coherent memory systems. In particular, WMM and WMM-S allow the OOO processors in multicore settings to use all speculative techniques which are valid for uniprocessors, including even the *load-value speculation* [37, 43, 26, 45, 46], *without* additional checks or logic.

We give operational definitions of both WMM and WMM-S. An operation definition specifies an abstract machine, and the legal behaviors of a program under the memory model are those that can result by running the program on the abstract ma-

	Definition		Model properties / Implementation flexibility			
	Operational model	Axiomatic model	Store atomicity	Allow shared write-through cache/shared store buffer	Instruction reorderings	Ordering of data-dependent loads
SC	Simple; I^{2}E [35]	Simple [35]	Single-copy atomic	No	None	Yes
TSO	Simple; I^{2}E [51]	Simple [56]	Multi-copy atomic	No	Only St-Ld reordering	Yes
RMO	Doesn't exist	Simple; needs fix [59]	Multi-copy atomic	No	All four	Yes
Alpha	Doesn't exist	Medium [4]	Multi-copy atomic	No	All four	No
RC	Doesn't exist	Medium [28]	Unclear	*No*	All four	Yes
ARM and POWER	Complex; non I^{2}E [49, 25]	Complex [39, 11]	Non-atomic	Yes	All four	Yes
WMM	Simple; I^{2}E	Simple	Multi-copy atomic	No	All except Ld-St reordering	No
WMM-S	Medium; I^{2}E	Doesn't exist	Non-atomic	Yes	All except Ld-St reordering	No

Figure 1: Summary of different memory models

chine. We observe a growing interest in operational definitions: memory models of x86, ARM and POWER have all been formalized operationally [44, 51, 49, 48, 25], and researchers are even seeking operational definitions for high-level languages like C++ [34]. This is perhaps because all possible program results can be derived from operational definitions mechanically while axiomatic definitions require guessing the whole program execution at the beginning. For complex programs with dependencies, loops and conditional branches, guessing the whole execution may become prohibitive.

Unfortunately, the operational models of ARM and POWER are too complicated because their abstract machines involve microarchitectural details like reorder buffers (ROBs), partial and speculative instruction execution, instruction replay on speculation failure, etc. The operational definitions of WMM and WMM-S are much simpler because they are described in terms of *Instantaneous Instruction Execution* (I^{2}E), which is the style used in the operational definitions of SC [35] and TSO [44, 51]. An I^{2}E abstract machine consists of n atomic processors and a n-ported atomic memory. The atomic processor executes instructions instantaneously and in order, so it always has the up-to-date architectural (register) state. The atomic memory executes loads and stores instantaneously. Instruction reorderings and store atomicity/non-atomicity are captured by including different types of buffers between the processors and the atomic memory, like the store buffer in the definition of TSO. In the background, data moves between these buffers and the memory asynchronously, e.g., to drain a value from a store buffer to the memory.

I^{2}E definitions free programmers from reasoning partially executed instructions, which is unavoidable for ARM and POWER operational definitions. One key tradeoff to achieve I^{2}E is to forbid a store to overtake a load, i.e., disallow Ld-St reordering. Allowing such reordering requires each processor in the abstract machine to maintain multiple unexecuted instructions in order to see the effects of future stores, and the abstract machine has to contain the complicated ROB-like structures. Ld-St reordering also complicates axiomatic definitions because it creates the possibility of "out-of-thin-air" behaviors [20], which are impossible in any real implementation and consequently must be disallowed. We also offer evidence, based on simulation experiments, that disallowing Ld-St reordering has no discernible impact on performance.

For a quick comparison, we summarize the properties of common memory models in Figure 1. SC and TSO have simple definitions but forbid Ld-Ld and St-St reorderings, and consequently, are not candidates for RISC-V. WMM is similar to RMO and Alpha but neither has an operational definition. Also WMM has a simple axiomatic definition, while Alpha requires a complicated axiom to forbid out-of-thin-air behaviors (see Section 5.2), and RMO has an incorrect axiom about data-dependency ordering (see Section 10).

ARM, POWER, and WMM-S are similar models in the sense that they all admit non-atomic stores. While the operational models of ARM and POWER are complicated, WMM-S has a simpler I^{2}E definition and allows competitive implementations (see Section 9.2). The axiomatic models of ARM and POWER are also complicated: four relations in the POWER axiomatic model [11, Section 6] are defined in a fixed point manner, i.e., their definitions mutually depend on each other.

Release Consistency (RC) are often mixed with the concept of "SC for data-race-free (DRF) programs" [6]. It should be noted that "SC for DRF" is inadequate for an ISA memory model, which must specify behaviors of *all* programs. The original RC definition [28] attempts to specify all program behaviors, and are more complex and subtle than the "SC for DRF" concept. We show in Section 10 that the RC definition fails a litmus test for non-atomic stores and forbids shared write-through caches in implementation.

This paper makes the following contributions:

1. WMM, the *first* weak memory model that is defined in I^{2}E and allows Ld-Ld reordering, and its axiomatic definition;
2. WMM-S, an extension on WMM that admits non-atomic stores and has an I^{2}E definition;
3. WMM and WMM-S implementations based on OOO processors that admit all uniprocessor speculative techniques (such as load-value prediction) without additional checks;
4. Introduction of *invalidation buffers* in the I^{2}E definitional framework to model Ld-Ld and other reorderings.

Paper organization: Section 2 presents the related work. Section 3 gives litmus tests for distinguishing memory models. Section 4 introduces I^{2}E. Section 5 defines WMM. Section 6 shows the WMM implementation using OOO processors. Section 7 evaluates the performance of WMM and the influence of forbidding Ld-St reordering. Section 8 defines WMM-S. Section 9 presents the WMM-S implementations with non-atomic stores. Section 10 shows the problems of RC and RMO. Section 11 offers the conclusion.

2. Related Work

SC [35] is the simplest model, but naive implementations of SC suffer from poor performance. Although researchers have proposed aggressive techniques to preserve SC [27, 47, 32, 29, 23, 60, 18, 53, 36, 31], they are rarely adopted in commercial processors perhaps due to their hardware complexity. Instead the manufactures and researchers have chosen to present weaker memory models, e.g. TSO [56, 44, 51, 50], PSO [59], RMO [59], Alpha [4], Processor Consistency [30], Weak Consistency [24], RC [28], CRF [52], Instruction Reordering + Store Atomicity [14], POWER [33] and ARM [13]. The tutorials by Adve et al. [5] and by Maranget et al. [42] provide relationships among some of these models.

A large amount of research has also been devoted to specifying the memory models of high-level languages: C++ [54, 19, 17, 15, 34], Java [41, 22, 40], etc. We will provide compilation schemes from C++ to WMM and WMM-S.

Recently, Lustig et al. have used Memory Ordering Specification Tables (MOSTs) to describe memory models, and proposed a hardware scheme to dynamically convert programs across memory models described in MOSTs [38]. MOST specifies the ordering strength (e.g. locally ordered, multi-copy atomic) of two instructions from the same processor under different conditions (e.g. data dependency, control dependency). Our work is orthogonal in that we propose new memory models with operational definitions.

3. Memory Model Litmus Tests

Here we offer two sets of litmus tests to highlight the differences between memory models regarding store atomicity and instruction reorderings, including enforcement of dependency-ordering. All memory locations are initialized to 0.

3.1. Store Atomicity Litmus Tests

Figure 2 shows four litmus tests to distinguish between these three types of stores. We have deliberately added data dependencies and Ld-Ld fences ($\mathsf{FENCE_{LL}}$) to these litmus tests to prevent instruction reordering, e.g., the data dependency between I_2 and I_3 in Figure 2a. Thus the resulting behaviors can arise only because of different store atomicity properties. We use $\mathsf{FENCE_{LL}}$ for memory models that can reorder data-dependent loads, e.g., I_5 in Figure 2b would be the MB fence for Alpha. For other memory models that order data-dependent loads (e.g. ARM), $\mathsf{FENCE_{LL}}$ could be replaced by a data dependency (like the data dependency between I_2 and I_3 in Figure 2a). The Ld-Ld fences only stop Ld-Ld reordering; they do not affect store atomicity in these tests.

Proc. P1	Proc. P2
$I_1 : \mathsf{St}\ a\ 1$	$I_4 : \mathsf{St}\ b\ 1$
$I_2 : r_1 = \mathsf{Ld}\ a$	$I_5 : r_3 = \mathsf{Ld}\ b$
$I_3 : r_2 = \mathsf{Ld}\ (b + r_1 - 1)$	$I_6 : r_4 = \mathsf{Ld}\ (a + r_3 - 1)$
SC forbids but TSO allows: $r_1 = 1, r_2 = 0, r_3 = 1, r_4 = 0$	

(a) SBE: test for multi-copy atomic stores

Proc. P1	Proc. P2	Proc. P3
$I_1 : \mathsf{St}\ a\ 2$	$I_2 : r_1 = \mathsf{Ld}\ a$	$I_4 : r_2 = \mathsf{Ld}\ b$
	$I_3 : \mathsf{St}\ b\ (r_1 - 1)$	$I_5 : \mathsf{FENCE_{LL}}$
		$I_6 : r_3 = \mathsf{Ld}\ a$
TSO, RMO and Alpha forbid, but RC, ARM and POWER allow: $r_1 = 2,\ r_2 = 1,\ r_3 = 0$		

(b) WRC: test for non-atomic stores [49]

Proc. P1	Proc. P2	Proc. P3
$I_1 : \mathsf{St}\ a\ 2$	$I_2 : r_1 = \mathsf{Ld}\ a$	$I_4 : r_2 = \mathsf{Ld}\ b$
	$I_3 : \mathsf{St}\ b\ (r_1 - 1)$	$I_5 : \mathsf{St}\ a\ r_2$
TSO, RMO, Alpha and *RC* forbid, but ARM and POWER allow: $r_1 = 2,\ r_2 = 1,\ m[a] = 2$		

(c) WWC: test for non-atomic stores [42, 3]

Proc. P1	Proc. P2	Proc. P3	Proc. P4
$I_1 : \mathsf{St}\ a\ 1$	$I_2 : r_1 = \mathsf{Ld}\ a$	$I_5 : \mathsf{St}\ b\ 1$	$I_6 : r_3 = \mathsf{Ld}\ b$
	$I_3 : \mathsf{FENCE_{LL}}$		$I_7 : \mathsf{FENCE_{LL}}$
	$I_4 : r_2 = \mathsf{Ld}\ b$		$I_8 : r_4 = \mathsf{Ld}\ a$
TSO, RMO and Alpha forbid, but RC, ARM and POWER allow: $r_1 = 1,\ r_2 = 0,\ r_3 = 1,\ r_4 = 0$			

(d) IRIW: test for non-atomic stores [49]

Figure 2: Litmus tests for store atomicity

SBE: In a machine with single-copy atomic stores (e.g. an SC machine), when both I_2 and I_5 have returned value 1, stores I_1 and I_4 must have been globally advertised. Thus r_2 and r_4 cannot both be 0. However, a machine with store buffers (e.g. a TSO machine) allows P1 to forward the value of I_1 to I_2 locally without advertising I_1 to other processors, violating the single-copy atomicity of stores.

WRC: Assuming the store buffer is private to each processor (i.e. multi-copy atomic stores), if one observes $r_1 = 2$ and $r_2 = 1$ then r_3 must be 2. However, if an architecture allows a store buffer to be shared by P1 and P2 but not P3, then P2 can see the value of I_1 from the shared store buffer before I_1 has updated the memory, allowing P3 to still see the old value of a. A write-through cache shared by P1 and P2 but not P3 can cause this *non-atomic store* behavior in a similar way, e.g., I_1 updates the shared write-through cache but has not invalidated the copy in the private cache of P3 before I_6 is executed.

WWC: This litmus test is similar to WRC but replaces the load in I_6 with a store. The behavior is possible if P1 and P2 share a write-through cache or store buffer. However, RC forbids this behavior (see Section 10).

IRIW: This behavior is possible if P1 and P2 share a write-through cache or a store buffer and so do P3 and P4.

3.2. Instruction Reordering Litmus Tests

Although processors fetch and commit instructions in order, speculative and out-of-order execution causes behaviors as if instructions were reordered. Figure 3 shows the litmus tests on these reordering behaviors.

Proc. P1	Proc. P2
I_1 : St a 1	I_3 : St b 1
$I_2 : r_1 = $ Ld b	$I_4 : r_2 = $ Ld a
SC forbids, but TSO allows: $r_1 = 0, r_2 = 0$	

(a) SB: test for St-Ld reordering [42]

Proc. P1	Proc. P2
I_1 : St a 1	$I_3 : r_1 = $ Ld b
I_2 : St b 1	$I_4 : r_2 = $ Ld a
TSO forbids, but Alpha and RMO allow: $r_1 = 1, r_2 = 0$	

(b) MP: test for Ld-Ld and St-St reorderings [49]

Proc. P1	Proc. P2
$I_1 : r_1 = $ Ld b	$I_3 : r_2 = $ Ld a
I_2 : St a 1	I_4 : St b 1
TSO forbids, but Alpha, RMO, RC, POWER and ARM allow: $r_1 = r_2 = 1$	

(c) LB: test for Ld-St reordering [49]

Proc. P1	Proc. P2
I_1 : St a 1	$I_4 : r_1 = $ Ld b
I_2 : FENCE	I_5 : if$(r_1 \neq 0)$ exit
I_3 : St b 1	$I_6 : r_2 = $ Ld a
Alpha, RMO, RC, ARM and POWER allow: $r_1 = 1, r_2 = 0$	

(d) MP+Ctrl: test for control-dependency ordering

Proc. P1	Proc. P2
I_1 : St a 1	$I_4 : r_1 = $ Ld b
I_2 : FENCE	I_5 : St $(r_1 + a)$ 42
I_3 : St b 100	$I_6 : r_2 = $ Ld a
Alpha, RMO, RC, ARM and POWER allow: $r_1 = 100, r_2 = 0$	

(e) MP+Mem: test for memory-dependency ordering

Proc. P1	Proc. P2
I_1 : St a 1	$I_4 : r_1 = $ Ld b
I_2 : FENCE	$I_5 : r_2 = $ Ld r_1
I_3 : St b a	
RMO, RC, ARM and POWER forbid, but Alpha allows: $r_1 = a, r_2 = 0$	

(f) MP+Data: test for data-dependency ordering

Figure 3: Litmus tests for instruction reorderings

SB: A TSO machine can execute I_2 and I_4 while I_1 and I_3 are buffered in the store buffers. The resulting behavior is as if the store and the load were reordered on each processor.
MP: In an Alpha machine, I_1 and I_2 may be drained from the store buffer of P1 out of order; I_3 and I_4 in the ROB of P2 may be executed out of order. This is as if P1 reordered the two stores and P2 reordered the two loads.
LB: Some machines may enter a store into the memory before all older instructions have been committed. This results in the Ld-St reordering shown in Figure 3c. Since instructions are committed in order and stores are usually not on the critical path, the benefit of the eager execution of stores is limited. In fact we will show by simulation that Ld-St reordering does not improve performance (Section 7).
MP+Ctrl: This test is a variant of MP. The two stores in P1 must update memory in order due to the fence. Although the execution of I_6 is conditional on the result of I_4, P2 can issue I_6 speculatively by predicting branch I_5 to be not taken. The execution order $I_6, I_1, I_2, I_3, I_4, I_5$ results in $r_1 = 1$ and $r_2 = 0$.
MP+Mem: This test replaces the control dependency in MP+Ctrl with a (potential) memory dependency, i.e., the unresolved store address of I_5 may be the same as the load address of I_6 before I_4 is executed, However, P2 can execute I_6 speculatively by predicting the addresses are not the same. This results in having I_6 overtake I_4 and I_5.
MP+Data: This test replaces the control dependency in MP+Ctrl with a data dependency, i.e., the load address of I_5 depends on the result of I_4. A processor with *load-value prediction* [37, 43, 26, 45, 46] may guess the result of I_4 before executing it, and issue I_5 speculatively. If the guess fails to match the real execution result of I_4, then I_5 would be killed. But, if the guess is right, then essentially the execution of the two data-dependent loads (I_4 and I_5) has been reordered.

3.3. Miscellaneous Tests

All programmers expect memory models to obey *per-location SC* [21], i.e., all accesses to a single address appear to execute in a sequential order which is consistent with the program order of each thread (Figure 4).

Proc. P1	Proc. P2
$I_1 : r_1 = $ Ld a	I_3 : St a 1
$I_2 : r_2 = $ Ld a	
Models with per-location SC forbid: $r_1 = 1, r_2 = 0$	

Figure 4: Per-location SC

Proc. P1	Proc. P2
$I_1 : r_1 = $ Ld b	$I_3 : r_2 = $ Ld a
I_2 : St a r_1	I_4 : St b r_2
All models forbid: $r_1 = r_2 = 42$	

Figure 5: Out-of-thin-air read

Out-of-thin-air behaviors (Figure 5) are impossible in real implementations. Sometimes such behaviors are permitted by axiomatic models due to incomplete axiomatization.

4. Defining Memory Models in I^{2}E

Figure 6 shows the I^{2}E abstract machines for SC, TSO/PSO and WMM models. All abstract machines consist of n atomic processors and a n-ported atomic memory m. Each processor contains a register state s, which represents all architectural registers, including both the general purpose registers and special purpose registers, such as PC. The abstract machines for TSO/PSO and WMM also contain a *store buffer sb* for each processor, and the one for WMM also contains an *invalidation buffer ib* for each processor as shown in the figure. In the abstract machines all buffers are unbounded. The operations of these buffers will be explained shortly.

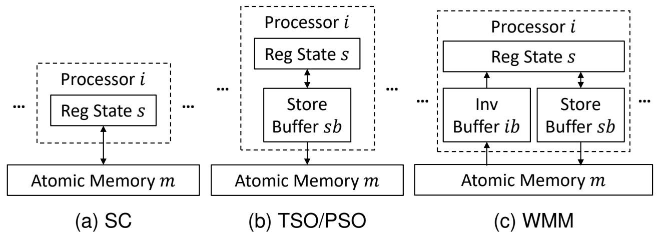

Figure 6: I^{2}E abstract machines for different models

The operations of the SC abstract machine are the simplest: in one step we can select any processor to execute the next instruction on that processor *atomically*. That is, if the instruction is a non-memory instruction (e.g. ALU or branch), it just modifies the register states of the processor; if it is a load, it reads from the atomic memory instantaneously and updates the register state; and if it is a store, it updates the atomic memory instantaneously and increments the PC.

4.1. TSO Model

The TSO abstract machine proposed in [44, 51] (Figure 6b) contains a store buffer *sb* for each processor. Just like SC, any processor can execute an instruction atomically, and if the instruction is a non-memory instruction, it just modifies the local register state. A store is executed by inserting its $\langle \text{address}, \text{value} \rangle$ pair into the local *sb* instead of writing the data in memory. A load first looks for the load address in the local *sb* and returns the value of the *youngest* store for that address. If the address is not in the local *sb*, then the load returns the value from the atomic memory. TSO can also perform a *background* operation, which removes the *oldest* store from a *sb* and writes it into the atomic memory. As we discussed in Section 3, store buffer allows TSO to do St-Ld reordering, i.e., pass the SB litmus test (Figure 3a).

In order to enforce ordering in accessing the memory and to rule out non-SC behaviors, TSO has a fence instruction, which we refer to as Commit. When a processor executes a Commit fence, it gets blocked unless its *sb* is empty. Eventually, any *sb* will become empty as a consequence of the background operations that move data from the *sb* to the memory. For example, we need to insert a Commit fence after each store in Figure 3a to forbid the non-SC behavior in TSO.

We summarize the operations of the TSO abstract machine in Figure 7. Each operation consists of a *predicate* and an *action*. The operation can be performed by taking the action only when the predicate is true. Each time we perform only one operation (either instruction execution or *sb* dequeue) atomically in the whole system (e.g., no two processors can execute instructions simultaneously). The choice of which operation to perform is nondeterministic.

Enabling St-St reordering: We can extend TSO to PSO by changing the background operation to dequeue the oldest store for *any* address in *sb* (see the PSO-DeqSb operation in Figure 7). This extends TSO by permitting St-St reordering.

5. WMM Model

WMM allows Ld-Ld reordering in addition to the reorderings allowed by PSO. Since a reordered load may read a stale value, we introduce a conceptual device called *invalidation buffer*, *ib*, for each processor in the I^2E abstract machine (see Figure 6c). *ib* is an unbounded buffer of $\langle \text{address}, \text{value} \rangle$ pairs, each representing a stale memory value for an address that can be observed by the processor. Multiple stale values for an address in *ib* are kept ordered by their staleness.

TSO-Nm (non-memory execution)
Predicate: The next instruction of a processor is a non-memory instruction.
Action: Instruction is executed by local computation.
TSO-Ld (load execution)
Predicate: The next instruction of a processor is a load.
Action: Assume the load address is a. The load returns the value of the *youngest* store for a in *sb* if a is present in the *sb* of the processor, otherwise, the load returns $m[a]$, i.e. the value of address a in the atomic memory.
TSO-St (store execution)
Predicate: The next instruction of a processor is a store.
Action: Assume the store address is a and the store value is v. The processor inserts the store $\langle a, v \rangle$ into its *sb*.
TSO-Com (Commit execution)
Predicate: The next instruction of a processor is a Commit and the *sb* of the processor is empty.
Action: The Commit fence is executed simply as a NOP.
TSO-DeqSb (background store buffer dequeue)
Predicate: The *sb* of a processor is not empty.
Action: Assume the $\langle \text{address}, \text{value} \rangle$ pair of the oldest store in the *sb* is $\langle a, v \rangle$. Then this store is removed from *sb*, and the atomic memory $m[a]$ is updated to v.

PSO-DeqSb (background store buffer dequeue)
Predicate: The *sb* of a processor is not empty.
Action: Assume the value of the oldest store for *some* address a in the *sb* is v. Then this store is removed from *sb*, and the atomic memory $m[a]$ is updated to v.

Figure 7: Operations of the TSO/PSO abstract machine

The operations of the WMM abstract machine are similar to those of PSO except for the background operation and the load execution. When the background operation moves a store from *sb* to the atomic memory, the original value in the atomic memory, i.e. the stale value, enters the *ib* of every other processor. A load first searches the local *sb*. If the address is not found in *sb*, it either reads the value in the atomic memory or any stale value for the address in the local *ib*, the choice between the two being nondeterministic.

The abstract machine operations maintain the following invariants: once a processor *observes* a store, it cannot observe any staler store for that address. Therefore, (1) when a store is executed, values for the store address in the local *ib* are purged; (2) when a load is executed, values staler than the load result are flushed from the local *ib*; and (3) the background operation does not insert the stale value into the *ib* of a processor if the *sb* of the processor contains the address.

Just like introducing the Commit fence in TSO, to prevent loads from reading the stale values in *ib*, we introduce the Reconcile fence to clear the local *ib*. Figure 8 summarizes the operations of the WMM abstract machine.

5.1. Properties of WMM

Similar to TSO/PSO, WMM allows St-Ld and St-St reorderings because of *sb* (Figures 3a and 3b). To forbid the be-

WMM-Nm (non-memory execution): Same as TSO-Nm.
WMM-Ld (load execution)
Predicate: The next instruction of a processor is a load.
Action: Assume the load address is a. If a is present in the sb of the processor, then the load returns the value of the *youngest* store for a in the local sb. Otherwise, the load is executed in either of the following two ways (the choice is arbitrary):

1. The load returns the atomic memory value $m[a]$, and all values for a in the local ib are removed.
2. The load returns some value for a in the local ib, and all values for a older than the load result are removed from the local ib. (If there are multiple values for a in ib, the choice of which one to read is arbitrary).

WMM-St (store execution)
Predicate: The next instruction of a processor is a store.
Action: Assume the store address is a and the store value is v. The processor inserts the store $\langle a, v\rangle$ into its sb, and removes all values for a from its ib.
WMM-Com (Commit execution): Same as TSO-Com.
WMM-Rec (execution of a Reconcile fence)
Predicate: The next instruction of a processor is a Reconcile.
Action: All values in the ib of the processor are removed.
WMM-DeqSb (background store buffer dequeue)
Predicate: The sb of a processor is not empty.
Action: Assume the value of the oldest store for *some* address a in the sb is v. First, the stale $\langle$address, value$\rangle$ pair $\langle a, m[a]\rangle$ is inserted to the ib of every other processor whose sb does not contain a. Then this store is removed from sb, and $m[a]$ is set to v.

Figure 8: Operations of the WMM abstract machine

havior in Figure 3a, we need to insert a Commit followed by a Reconcile after the store in each processor. Reconcile is needed to prevent loads from getting stale values from ib. The I^{2}E definition of WMM automatically forbids Ld-St reordering (Figure 3c) and out-of-thin-air behaviors (Figure 5).

Ld-Ld reordering: WMM allows the behavior in Figure 3b due to St-St reordering. Even if we insert a Commit between the two stores in P1, the behavior is still allowed because I_4 can read the stale value 0 from ib. This is as if the two loads in P2 were reordered. Thus, we also need a Reconcile between the two loads in P2 to forbid this behavior in WMM.

No dependency ordering: WMM does not enforce any dependency ordering. For example, WMM allows the behaviors of litmus tests in Figures 3d, 3e and 3f (I_2 should be Commit in case of WMM), because the last load in P2 can always get the stale value 0 from ib in each test. Thus, it requires Reconcile fences to enforce dependency ordering in WMM. In particular, WMM can reorder the data-dependent loads (i.e. I_4 and I_5) in Figure 3f.

Multi-copy atomic stores: Stores in WMM are multi-copy atomic, and WMM allows the behavior in Figure 2a even when Reconcile fences are inserted between Ld-Ld pairs $\langle I_2, I_3\rangle$ and $\langle I_5, I_6\rangle$. This is because a store can be read by a load from the same processor while the store is in sb. However, if the store is ever pushed from sb to the atomic memory, it becomes visible to all other processors simultaneously. Thus WMM forbids the behaviors in Figures 2b, 2c and 2d (FENCE$_{\text{LL}}$ should be Reconcile in these tests).

Per-location SC: WMM enforces per-location SC (Figure 4), because both sb and ib enforce FIFO on same address entries.

5.2. Axiomatic Definition of WMM

Based on the above properties of WMM, we give a simple axiomatic definition for WMM in Figure 9 in the style of the axiomatic definitions of TSO and Alpha. A True entry in the order-preserving table (Figure 9b) indicates that if instruction X precedes instruction Y in the program order ($X <_{po} Y$) then the order must be maintained in the global memory order ($<_{mo}$). The notation $S \xrightarrow{rf} L$ means a load L reads from a store S. The notation $\max_{<_{mo}}\{\text{set of stores}\}$ means the youngest store in the set according to $<_{mo}$. The axioms are self-explanatory: the program order must be maintained if the order-preserving table says so, and a load must read from the youngest store among all stores that precede the load in either the memory order or the program order. (See Appendix A for the equivalence proof of axiomatic and I^{2}E definitions.)

These axioms also hold for Alpha with a slightly different order-preserving table, which marks the (Ld,St) entry as $a = b$. (Alpha also merges Commit and Reconcile into a single fence). However, allowing Ld-St reordering creates the possibility of out-of-thin-air behaviors, and Alpha uses an additional complicated axiom to disallow such behaviors [4, Chapter 5.6.1.7]. This axiom requires considering all possible execution paths to determine if a store is ordered after a load by dependency, while normal axiomatic models only examine a single execution path at a time. Allowing Ld-St reordering also makes it difficult to define Alpha operationally.

Axiom Inst-Order (preserved instruction ordering):

$$X <_{po} Y \wedge \text{order}(X,Y) \Rightarrow X <_{mo} Y$$

Axiom Ld-Val (the value of a load):

$$\text{St } a\ v \xrightarrow{rf} \text{Ld } a \Rightarrow$$
$$\text{St } a\ v = \max_{<_{mo}}\{\text{St } a\ v' \mid \text{St } a\ v' <_{mo} \text{Ld } a \vee \text{St } a\ v' <_{po} \text{Ld } a\}$$

(a) Axioms for WMM

X \ Y	Ld b	St $b\ v'$	Reconcile	Commit
Ld a	$a = b$	True	True	True
St $a\ v$	False	$a = b$	False	True
Reconcile	True	True	True	True
Commit	False	True	True	True

(b) WMM order-preserving table, i.e. order(X,Y) where $X <_{po} Y$

Figure 9: Axiomatic definition of WMM

5.3. Compiling C++11 to WMM

C++ primitives [54] can be mapped to WMM instructions in an efficient way as shown in Figure 10. For the purpose of comparison, we also include a mapping to POWER [16].

C++ operations	WMM instructions	POWER instructions
Non-atomic Load	Ld	Ld
Load Relaxed	Ld	Ld
Load Consume	Ld; Reconcile	Ld
Load Acquire	Ld; Reconcile	Ld; cmp; bc; isync
Load SC	Commit; Reconcile; Ld; Reconcile	sync; Ld; cmp; bc; isync
Non-atomic Store	St	St
Store Relaxed	St	St
Store Release	Commit; St	lwsync; St
Store SC	Commit; St	sync; St

Figure 10: Mapping C++ to WMM and POWER

The Commit; Reconcile sequence in WMM is the same as a sync fence in POWER, and Commit is similar to lwsync. The cmp; bc; isync sequence in POWER serves as a Ld-Ld fence, so it is similar to a Reconcile fence in WMM. In case of Store SC in C++, WMM uses a Commit while POWER uses a sync, so WMM effectively saves one Reconcile. On the other hand, POWER does not need any fence for Load Consume in C++, while WMM requires a Reconcile.

Besides the C++ primitives, a common programming paradigm is the well-synchronized program, in which all critical sections are protected by locks. To maintain SC behaviors for such programs in WMM, we can add a Reconcile after acquiring the lock and a Commit before releasing the lock.

For any program, if we insert a Commit before every store and insert a Commit followed by a Reconcile before every load, then the program behavior in WMM is guaranteed to be sequentially consistent. This provides a conservative way for inserting fences when performance is not an issue.

6. WMM Implementation

WMM can be implemented using conventional OOO multiprocessors, and even the most aggressive speculative techniques cannot step beyond WMM. To demonstrate this, we describe an OOO implementation of WMM, and show simultaneously how the WMM model (i.e. the I^2E abstract machine) captures the behaviors of the implementation. The implementation is described abstractly to skip unrelated details (e.g. ROB entry reuse). The implementation consists of n OOO processors and a coherent write-back cache hierarchy which we discuss next.

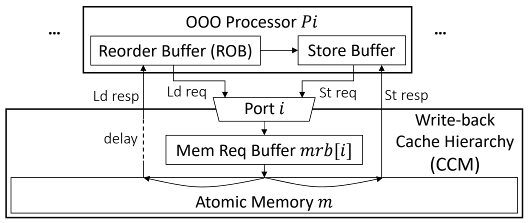

Figure 11: CCM+OOO: implementation of WMM

6.1. Write-Back Cache Hierarchy (CCM)

We describe CCM as an abstraction of a conventional write-back cache hierarchy to avoid too much details. In the following, we explain the function of such a cache hierarchy, abstract it to CCM, and relate CCM to the WMM model.

Consider a real n-ported write-back cache hierarchy with each port i connected to processor Pi. A request issued to port i may be from a load instruction in the ROB of Pi or a store in the store buffer of Pi. In conventional coherence protocols, all memory requests can be *serialized*, i.e., each request can be considered as taking effect at some time point within its processing period [57]. For example, consider the non-stalling MSI directory protocol in the Primer by Sorin et al. [55, Chapter 8.7.2]. In this protocol, a load request takes effect immediately if it hits in the cache; otherwise, it takes effect when it gets the data at the directory or a remote cache with M state. A store request always takes effect at the time of writing the cache, i.e., either when it hits in the cache, or when it has received the directory response and all invalidation responses in case of miss. We also remove the requesting store from the store buffer when a store request takes effect. Since a cache cannot process multiple requests to the same address simultaneously, we assume requests to the same address from the same processor are processed in the order that the requests are issued to the cache.

CCM (Figure 11) abstracts the above cache hierarchy by operating as follows: every new request from port i is inserted into a memory request buffer $mrb[i]$, which keeps requests to the same address in order; at any time we can remove the oldest request for an address from a mrb, let the request access the atomic memory m, and either send the load result to ROB (which may experience a delay) or immediately dequeue the store buffer. m represents the coherent memory states. Removing a request from mrb and accessing m captures the moment when the request takes effect.

It is easy to see that the atomic memory in CCM corresponds to the atomic memory in the WMM model, because they both hold the coherent memory values. We will show shortly that how WMM captures the combination of CCM and OOO processors. Thus any coherent protocol that can be abstracted as CCM can be used to implement WMM.

6.2. Out-of-Order Processor (OOO)

The major components of an OOO processor are the ROB and the store buffer (see Figure 11). Instructions are fetched into and committed from ROB in order; loads can be issued (i.e., search for data forwarding and possibly request CCM) as soon as its address is known; a store is enqueued into the store buffer only when the store commits (i.e. entries in a store buffer cannot be killed). To maintain the per-location SC property of WMM, when a load L is issued, it kills younger loads which have been issued but do not read from stores younger than L. Next we give the correspondence between OOO and WMM.

Store buffer: The state of the store buffer in OOO is represented by the sb in WMM. Entry into the store buffer when a store commits in OOO corresponds to the WMM-St operation. In OOO, the store buffer only issues the oldest store for some address to CCM. The store is removed from the store buffer when the store updates the atomic memory in CCM. This corresponds to the WMM-DeqSb operation.

ROB and eager loads: Committing an instruction from ROB corresponds to executing it in WMM, and thus the architectural register state in both WMM and OOO must match at the time of commit. Early execution of a load L to address a with a return value v in OOO can be understood by considering where $\langle a, v\rangle$ resides in OOO when L *commits*. Reading from *sb* or atomic memory m in the WMM-Ld operation covers the cases that $\langle a, v\rangle$ is, respectively, in the store buffer or the atomic memory of CCM when L commits. Otherwise $\langle a, v\rangle$ is no longer present in CCM+OOO at the time of load commit and must have been overwritten in the atomic memory of CCM. This case corresponds to having performed the WMM-DeqSb operation to insert $\langle a, v\rangle$ into *ib* previously, and now using the WMM-Ld operation to read v from *ib*.

Speculations: OOO can issue a load speculatively by aggressive predictions, such as branch prediction (Figure 3d), memory dependency prediction (Figure 3e) and even load-value prediction (Figure 3f). As long as all predictions related to the load eventually turn out to be correct, the load result got from the speculative execution can be preserved. *No further check is needed*. Speculations effectively reorder dependent instructions, e.g., load-value speculation reorders data-dependent loads. Since WMM does not require preserving any dependency ordering, speculations will neither break WMM nor affect the above correspondence between OOO and WMM.

Fences: Fences never go into store buffers or CCM in the implementation. In OOO, a Commit can commit from ROB only when the local store buffer is empty. Reconcile plays a different role; at the time of commit it is a NOP, but while it is in the ROB, it stalls all younger loads (unless the load can bypass directly from a store which is younger than the Reconcile). The stall prevents younger loads from reading values that would become stale when the Reconcile commits. This corresponds to clearing *ib* in WMM.

Summary: For any execution in the CCM+OOO implementation, we can operate the WMM model following the above correspondence. Each time CCM+OOO commits an instruction I from ROB or dequeues a store S from a store buffer to memory, the atomic memory of CCM, store buffers, and the results of committed instructions in CCM+OOO are exactly the same as those in the WMM model when the WMM model executes I or dequeues S from *sb*, respectively.

7. Performance Evaluation of WMM

We evaluate the performance of implementations of WMM, Alpha, SC and TSO. All implementations use OOO cores and coherent write-back cache hierarchy. Since Alpha allows Ld-St reordering, the comparison of WMM and Alpha will show whether such reordering affects performance.

7.1. Evaluation Methodology

We ran SPLASH-2x benchmarks [61, 2] on an 8-core multiprocessor using ESESC simulator [12]. We ran all benchmarks except ocean_ncp, which allocates too much memory and breaks the original simulator. We used sim-medium inputs except for cholesky, fft and radix, where we used sim-large inputs. We ran all benchmarks to completion without sampling.

The configuration of the 8-core multiprocessor is shown in Figures 12 and 13 . We do not use load-value speculation in this evaluation. The Alpha implementation can mark a younger store as committed when instruction commit is stalled, as long as the store can never be squashed and the early commit will not affect single-thread correctness. A committed store can be issued to memory or merged with another committed store in WMM and Alpha. SC and TSO issue loads speculatively and monitor L1 cache evictions to kill speculative loads that violate the consistency model. We also implement store prefetch as an optional feature for SC and TSO; We use **SC-pf** and **TSO-pf** to denote the respective implementations with store prefetch.

Cores	8 cores (@2GHz) with private L1 and L2 caches
L3 cache	4MB shared, MESI coherence, 64-byte cache line 8 banks, 16-way, LRU replacement, max 32 req per bank 3-cycle tag, 10-cycle data (both pipelined) 5 cycles between cache bank and core (pipelined)
Memory	120-cycle latency, max 24 requests

Figure 12: Multiprocessor system configuration

Frontend	fetch + decode + rename, 7-cycle pipelined latency in all 2-way superscalar, hybrid branch predictor
ROB	128 entries, 2-way issue/commit
Function units	2 ALUs, 1 FPU, 1 branch unit, 1 load unit, 1 store unit 32-entry reservation station per unit
Ld queue	Max 32 loads
St queue	Max 24 stores, containing speculative and committed stores
L1 D cache	32KB private, 1 bank, 4-way, 64-byte cache line LRU replacement, 1-cycle tag, 2-cycle data (pipelined) Max 32 upgrade and 8 downgrade requests
L2 cache	128KB private, 1 bank, 8-way, 64-byte cache line LRU replacement, 2-cycle tag, 6-cycle data (both pipelined) Max 32 upgrade and 8 downgrade requests

Figure 13: Core configuration

7.2. Simulation Results

A common way to study the performance of memory models is to monitor the commit of instructions at the commit slot of ROB (i.e. the oldest ROB entry). Here are some reasons why an instruction may not commit in a given cycle:

- **empty**: The ROB is empty.
- **exe**: The instruction at the commit slot is still executing.
- **pendSt**: The load (in SC) or Commit (in TSO, Alpha and WMM) cannot commit due to pending older stores.
- **flushLS**: ROB is being flushed because a load is killed by another older load (only in WMM and Alpha) or older store (in all models) to the same address.

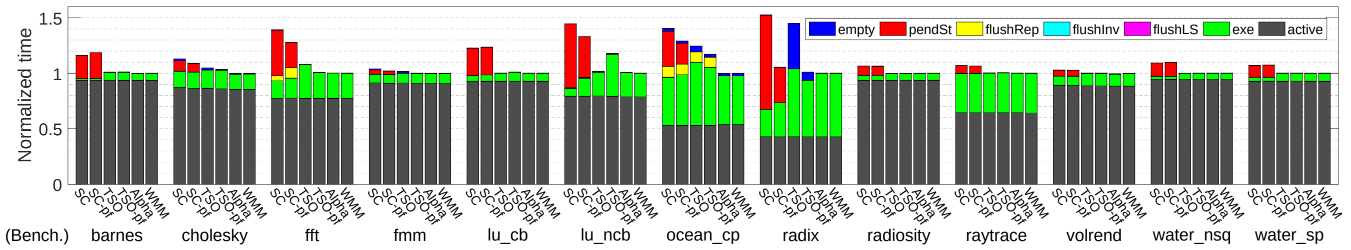

Figure 14: Normalized execution time and its breakdown at the commit slot of ROB

- **flushInv**: ROB is being flushed after cache invalidation caused by a remote store (only in SC or TSO).
- **flushRep**: ROB is being flushed after cache replacement (only in SC or TSO).

Figure 14 shows the execution time (normalized to WMM) and its breakdown at the commit slot of ROB. The total height of each bar represents the normalized execution time, and stacks represent different types of stall times added to the active committing time at the commit slot.

WMM versus SC: WMM is much faster than both SC and SC-pf for most benchmarks, because a pending older store in the store queue can block SC from committing loads.

WMM versus TSO: WMM never does worse than TSO or TSO-pf, and in some cases it shows up to 1.45× speedup over TSO (in radix) and 1.18× over TSO-pf (in lu_ncb). There are two disadvantages of TSO compared to WMM. First, load speculation in TSO is subject to L1 cache eviction, e.g. in benchmark ocean_cp. Second, TSO requires prefetch to reduce store miss latency, e.g., a full store queue in TSO stalls issue to ROB and makes ROB empty in benchmark radix. However, prefetch may sometimes degrade performance due to interference with load execution, e.g., TSO-pf has more commit stalls due to unfinished loads in benchmark lu_ncb.

WMM versus Alpha: Figure 15 shows the average number of cycles that a store in Alpha can commit before it reaches the commit slot. However, the early commit (i.e. Ld-St reordering) does not make Alpha outperform WMM (see Figure 14), because store buffers can already hide the store miss latency. Note that ROB is typically implemented as a FIFO (i.e. a circular buffer) for register renaming (e.g., freeing physical registers in order), precise exceptions, etc. Thus, if the early committed store is in the middle of ROB, its ROB entry cannot be reused by a newly fetched instruction, i.e. the effective size of the ROB will not increase. In summary, the Ld-St reordering in Alpha does not increase performance but complicates the definition (Section 5.2).

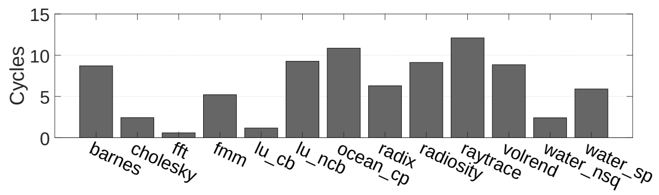

Figure 15: Avg. cycles to commit stores early in Alpha

8. WMM-S Model

Unlike the multi-copy atomic stores in WMM, stores in some processors (e.g. POWER) are non-atomic due to shared write-through caches or shared store buffers. If multiple processors share a store buffer or write-through cache, a store by any of these processors may be seen by all these processors before other processors. Although we could tag stores with processor IDs in the store buffer, it is infeasible to separate values stored by different processors in a cache.

In this section, we introduce a new I^{2}E model, WMM-S, which captures the non-atomic store behaviors in a way independent from the sharing topology. WMM-S is derived from WMM by adding a new background operation. We will show later in Section 9 why WMM-S can be implemented using memory systems with non-atomic stores.

8.1. I^{2}E Definition of WMM-S

The structure of the abstract machine of WMM-S is the same as that of WMM. To model non-atomicity of stores, i.e., to make a store by one processor readable by another processor before the store updates the atomic memory, WMM-S introduces a new background operation that copies a store from one store buffer into another. However, we need to ensure that all stores for an address can still be put in a total order, i.e. the coherence order, and the order seen by any processor is consistent with this total order (i.e. per-location SC).

To identify all the copies of a store in various store buffers, we assign a unique tag t when a store is executed (by being inserted into *sb*), and this tag is copied when a store is copied from one store buffer to another. When a background operation dequeues a store from a store buffer to the memory, all its copies must be deleted from all the store buffers which have them. This requires that all copies of the store are the oldest for that address in their respective store buffers.

All the stores for an address in a store buffer can be strictly ordered as a list, where the *youngest* store is the one that entered the store buffer *last*. We make sure that all ordered lists (of all store buffers) can be combined transitively to form a partial order (i.e. no cycle), which has now to be understood in terms of the tags on stores because of the copies. We refer to this partial order as the *partial coherence order* ($<_{co}$), because it is consistent with the coherence order.

Consider the states of store buffers shown in Figure 16 (primes are copies). A, B, C and D are different stores to the same address, and their tags are t_A, t_B, t_C and t_D, respectively. A' and B' are copies of A and B respectively created by the background copy operation. Ignoring C', the partial coherence order contains: $t_D <_{co} t_B <_{co} t_A$ (D is older than B, and B is older than A' in P2), and $t_C <_{co} t_B$ (C is older than B' in P3). Note that t_D and t_C are not related here.

At this point, if we copied C in P3 as C' into P1, we would add a new edge $t_A <_{co} t_C$, breaking the partial order by introducing the cycle $t_A <_{co} t_C <_{co} t_B <_{co} t_A$. Thus copying of C into P1 should be forbidden in this state. Similarly, copying a store with tag t_A into P1 or P2 should be forbidden because it would immediately create a cycle: $t_A <_{co} t_A$. In general, the background copy operation must be constrained so that the partial coherence order is still acyclic after copying.

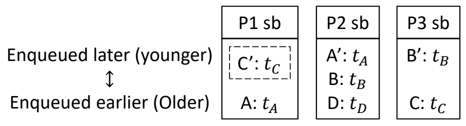

Figure 16: Example states of store buffers

Figure 17 shows the background operations of the WMM-S abstract machine. The operations that execute instructions in WMM-S are the same as those in WMM, so we do not show them again. (The store execution operation in WMM-S needs to also insert the tag of the store into *sb*).

WMM-DeqSb (background store buffer dequeue)
Predicate: There is a store S in a store buffer, and all copies of S are the oldest store for that address in their respective store buffers.
Action: Assume the $\langle$address, value, tag$\rangle$ tuple of store S is $\langle a, v, t\rangle$. First, the stale $\langle$address, value$\rangle$ pair $\langle a, m[a]\rangle$ is inserted to the *ib* of every processor whose *sb* does not contain a. Then all copies of S are removed from their respective store buffers, and the atomic memory $m[a]$ is updated to v.
WMM-S-Copy (background store copy)
Predicate: There is a store S that is in the *sb* of some processor i but not in the *sb* of some other processor j. Additionally, the partial coherence order will still be acyclic if we insert a copy of S into the *sb* of processor j.
Action: Insert a copy of S into the *sb* of processor j, and remove all values for the store address of S from the *ib* of processor j.

Figure 17: Background operations of WMM-S

Binding background copy with load execution: If the WMM-S-Copy operation is restricted to always happen right before a load execution operation that reads from the newly created copy, it is not difficult to prove that the WMM-S model remains the same, i.e., legal behaviors do not change. In the rest of the paper, we will only consider this "restricted" version of WMM-S. In particular, all WMM-S-Copy operations in the following analysis of litmus tests fall into this pattern.

8.2. Properties of WMM-S

WMM-S enforces per-location SC (Figure 4), because it prevents cycles in the order of stores to the same address. It also allows the same instruction reorderings as WMM does (Figure 3). We focus on the store non-atomicity of WMM-S.
Non-atomic stores and cumulative fences: Consider the litmus tests for non-atomic stores in Figures 2b, 2c and 2d (FENCE_{LL} should be Reconcile in these tests). WMM-S allows the behavior in Figure 2b by copying I_1 into the *sb* of P2 and then executing I_2, I_3, I_4, I_5, I_6 sequentially. I_1 will not be dequeued from *sb* until I_6 returns value 0. To forbid this behavior, a Commit is required between I_2 and I_3 in P2 to push I_1 into memory. Similarly, WMM-S allows the behavior in Figure 2c (i.e., we copy I_1 into the *sb* of P2 to satisfy I_2, and I_1 is dequeued after I_5 has updated the atomic memory), and we need a Commit between I_2 and I_3 to forbid the behavior. In both litmus tests, the inserted fences have a cumulative global effect in ordering I_1 before I_3 and the last instruction in P3.

WMM-S also allows the behavior in Figure 2d by copying I_1 into the *sb* of P2 to satisfy I_2, and copying I_5 into the *sb* of P4 to satisfy I_6. To forbid the behavior, we need to add a Commit right after the first load in P2 and P4 (but before the FENCE_{LL}/Reconcile that we added to stop Ld-Ld reordering). As we can see, Commit and Reconcile are similar to release and acquire respectively. Cumulation is achieved by globally advertising observed stores (Commit) and preventing later loads from reading stale values (Reconcile).
Programming properties: WMM-S is the same as WMM in the properties described in Section 5.3, including the compilation of C++ primitives, maintaining SC for well-synchronized programs, and the conservative way of inserting fences.

9. WMM-S Implementations

Since WMM-S is strictly more relaxed than WMM, any WMM implementation is a valid WMM-S implementation. However, we are more interested in implementations with non-atomic memory systems. Instead of discussing each specific system one by one, we explain how WMM-S can be implemented using the ARMv8 flowing model, which is a general abstraction of non-atomic memory systems [25]. We first describe the adapted flowing model (FM) which uses fences in WMM-S instead of ARM, and then explain how it obeys WMM-S.

9.1. The Flowing Model (FM)

FM consists of a tree of segments $s[i]$ rooted at the atomic memory m. For example, Figure 19 shows four OOO processors (P1...P4) connected to a 4-ported FM which has six segments ($s[1\dots6]$). Each segment is a list of memory requests, (e.g., the list of blue nodes in $s[6]$, whose head is at the bottom and the tail is at the top).

OOO interacts with FM in a slightly different way than CCM. Every memory request from a processor is appended to the tail of the list of the segment connected to the processor (e.g., $s[1]$ for P1). OOO no longer contains a store buffer; after a store is committed from ROB, it is directly sent to FM and there is no store response. When a Commit fence reaches the commit slot of ROB, the processor sends a Commit request to FM, and the ROB will not commit the Commit fence until FM sends back the response for the Commit request.

Inside FM, there are three background operations: (1) Two requests in the same segment can be reordered in certain cases; (2) A load can bypass from a store in the same segment; (3) The request at the head of the list of a segment can flow into the parent segment (e.g., flow from $s[1]$ into $s[5]$) or the atomic memory (in case the parent of the segment, e.g. $s[6]$, is m). Details of these operations are shown in Figure 18.

FM-Reorder (reorder memory requests)
Predicate: The list of segment $s[i]$ contains two consecutive requests r_{new} and r_{old} (r_{new} is above r_{old} in $s[i]$); and *neither* of the following is true:

1. r_{new} and r_{old} are memory accesses to the same address.
2. r_{new} is a Commit and r_{old} is a store.

Action: Reorder r_{new} and r_{old} in the list of $s[i]$.
FM-Bypass (store forwarding)
Predicate: The list of segment $s[i]$ contains two consecutive requests r_{new} and r_{old} (r_{new} is above r_{old} in $s[i]$). r_{new} is a load, r_{old} is a store, and they are for the same address.
Action: we send the load response for r_{new} using the store value of r_{old}, and remove r_{new} from the segment.
FM-Flow (flow request)
Predicate: A segment $s[i]$ is not empty.
Action: Remove the request r which is the head of the list of $s[i]$. If the parent of $s[i]$ in the tree structure is another segment $s[j]$, we append r to the tail of the list of $s[j]$. Otherwise, the parent of $s[i]$ is m, and we take the following actions according to the type of r:

- If r is a load, we send a load response using the value in m.
- If r is a store $\langle a, v\rangle$, we update $m[a]$ to v.
- If r is a Commit, we send a response to the requesting processor and the Commit fence can then be committed from ROB.

Figure 18: Background operations of FM

It is easy to see that FM abstracts non-atomic memory systems, e.g., Figure 19 abstracts a system in which P1 and P2 share a write-through cache while P3 and P4 share another.

Two properties of FM+OOO: First, FM+OOO enforces per-location SC because the segments in FM never reorder requests to the same address. Second, stores for the same address, which lie on the path from a processor to m in the tree structure of FM, are strictly ordered based on their distance to the tree root m; and the combination of all such orderings will not contain any cycle. For example, in Figure 19, stores in segments $s[3]$ and $s[6]$ are on the path from P3 to m; a store in $s[6]$ is older than any store (for the same address) in $s[3]$, and stores (for the same address) in the same segment are ordered from bottom to top (bottom is older).

9.2. Relating FM+OOO to WMM-S

WMM-S can capture the behaviors of any program execution in implementation FM+OOO in almost the same way that WMM captures the behaviors of CCM+OOO. When a store updates the atomic memory in FM+OOO, WMM-S performs a WMM-S-DeqSb operation to dequeue that store from store buffers to memory. When an instruction is committed from a ROB in FM+OOO, WMM-S executes that instruction. The following invariants hold after each operation in FM+OOO and the corresponding operation in WMM-S:

1. For each instruction committed in FM+OOO, the execution results in FM+OOO and WMM-S are the same.
2. The atomic memories in FM+OOO and WMM-S match.
3. The *sb* of each processor *Pi* in WMM-S holds exactly all the stores in FM+OOO that is *observed by the commits of Pi* but have not updated the atomic memory. (A store is observed by the commits of *Pi* if it has been either committed by *Pi* or returned by a load that has been committed by *Pi*).
4. The order of stores for the same address in the *sb* of any processor in WMM-S is exactly the order of those stores on the path from *Pi* to *m* in FM+OOO.

It is easy to see how the invariants are maintained when the atomic memory is updated or a non-load instruction is committed in FM+OOO. To understand the commit of a load L to address a with result v in processor *Pi* in FM+OOO, we still consider where $\langle a, v\rangle$ resides when L commits. Similar to WMM, reading atomic memory m or local *ib* in the load execution operation of WMM-S covers the cases that $\langle a, v\rangle$ is still in the atomic memory of FM or has already been overwritten by another store in the atomic memory of FM, respectively. In case $\langle a, v\rangle$ is a store that has not yet updated the atomic memory in FM, $\langle a, v\rangle$ must be on the path from *Pi* to m. In this case, if $\langle a, v\rangle$ has been observed by the commits of *Pi* before L is committed, then L can be executed by reading the local *sb* in WMM-S. Otherwise, on the path from *Pi* to m, $\langle a, v\rangle$ must be younger than any other store observed by the commits of *Pi*. Thus, WMM-S can copy $\langle a, v\rangle$ into the *sb* of *Pi* without breaking any invariant. The copy will not create any cycle in $<_{co}$ because of invariants 3 and 4 as well as the second property of FM+OOO mentioned above. After the copy, WMM-S can have L read v from the local *sb*.

Performance comparison with ARM and POWER: As we have shown that WMM-S can be implemented using the generalized memory system of ARM, we can turn an ARM multicore into a WMM-S implementation by stopping Ld-St reordering in the ROB. Since Section 7 already shows that Ld-St reordering does not affect performance, we can conclude qualitatively that there is no discernible performance difference between WMM-S and ARM implementations. The same arguments apply to the comparison against POWER and RC.

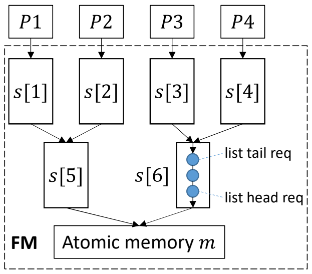

Figure 19: OOO+FM

Proc. P1	Proc P2
I_1 : St a 1	$I_4 : r_1 =$ Ld b
I_2 : MEMBAR	I_5 : if$(r_1 \neq 1)$ exit
I_3 : St b 1	I_6 : St c 1
	$I_7 : r_2 =$ Ld c
	$I_8 : r_3 = a + r_2 - 1$
	$I_9 : r_4 =$ Ld r_3
RMO forbids: $r_1 = 1,\ r_2 = 1$ $r_3 = a,\ r_4 = 0$	

Figure 20: Dependency ordering in RMO

10. Problems of RC and RMO

Here we elaborate the problems of RC (both RC_{sc} and RC_{pc}) and RMO, which have been pointed out in Section 1.

RC: Although the RC definition [28] allows the behaviors of WRC and IRIW (Figures 2b and 2d), it disallows the behavior of WWC (Figure 2c). In WWC, when I_2 reads the value of store I_1, the RC definition says that I_1 is performed with respect to (w.r.t) P2. Since store I_5 has not been issued due to the data dependencies in P2 and P3, I_1 must be performed w.r.t P2 before I_5. The RC definition says that "all writes to the same location are serialized in some order and are performed in that order with respect to any processor" [28, Section 2]. Thus, I_1 is before I_5 in the serialization order of stores for address a, and the final memory value of a cannot be 2 (the value of I_1), i.e., RC forbids the behavior of WWC and thus forbids shared write-through caches in implementations.

RMO: The RMO definition [59, Section D] is incorrect in enforcing dependency ordering. Consider the litmus test in Figure 20 (MEMBAR is the fence in RMO). In P2, the execution of I_6 is conditional on the result of I_4, I_7 loads from the address that I_6 stores to, and I_9 uses the results of I_7. According the definition of dependency ordering in RMO [59, Section D.3.3], I_9 depends on I_4 transitively. Then the RMO axioms [59, Section D.4] dictate that I_9 must be after I_4 in the memory order, and thus forbid the behavior in Figure 20. However, this behavior is possible in hardware with speculative load execution and store forwarding, i.e., I_7 first speculatively bypasses from I_6, and then I_9 executes speculatively to get 0. Since most architects will not be willing to give up on these two optimizations, RISC-V should not adopt RMO.

11. Conclusion

We have proposed two weak memory models, WMM and WMM-S, for RISC-V with different tradeoffs between definitional simplicity and implementation flexibility. However RISC-V can have only one memory model. Since there is no obvious evidence that restricting to multi-copy atomic stores affects performance or increases hardware complexity, RISC-V should adopt WMM in favor of simplicity.

12. Acknowledgment

This work was done at CSAIL, MIT as part of the Proteus project, which is partially funded by DARPA BRASS grant number 6933274 (2015-2019).

A. Proof of the Equivalence of the Axiomatic and I^{2}E Models of WMM

A.1. WMM I^{2}E Model $\subseteq$ WMM Axiomatic Model

We begin with proving that the I^{2}E model is contained by the axiomatic model, i.e., all WMM axioms are satisfied for every execution allowed by the I^{2}E model. For proof purposes, we first state the following two properties of the I^{2}E model (L_1, L_2, S_1, S_2 denote loads and stores to the same address):

- **CoRR** (Read-Read Coherence): $L_1 <_{po} L_2 \wedge S_1 \xrightarrow{rf} L_1 \wedge S_2 \xrightarrow{rf} L_2 \Longrightarrow S_1 = S_2 \vee S_1 <_{co} S_2$.
- **CoWR** (Write-Read Coherence): $S_2 \xrightarrow{rf} L_1 \wedge S_1 <_{po} L_1 \Longrightarrow S_1 = S_2 \vee S_1 <_{co} S_2$.

In the above properties, $<_{co}$ is the coherence order, which is a total order of all stores for the same address. $<_{co}$ is defined in the I^{2}E model as the order of dequeuing stores to the atomic memory. The CoRR property says that for the stores (S_1, S_2) read by two loads (L_1, L_2) for the same address and in the same processor, either they are the same store or they maintain the program order of the loads in the coherence order. The CoWR property says that if a load L_1 is younger than a store S_1 for the same address in the same processor, then L_1 reads from either S_1 or another store S_2 which is younger than S_1 in the coherence order. With the above two properties, we can prove the following theorem.

Theorem 1. *WMM* I^{2}E *model* $\subseteq$ *WMM axiomatic model.*

Proof. An execution of the I^{2}E model gives a total order E of all the operations. We select the operations from E that execute loads, execute fences, and dequeue stores to form a total order M of loads, fences and stores. Notice that M satisfies the Inst-Order axiom but does not satisfy the Ld-Val axiom. In the following, we will move some loads towards the beginning of total order M to make M the memory order ($<_{mo}$) in the axiomatic model which satisfy both axioms. For each processor Pi, we modify total order M by examining each load L of Pi in program order (from oldest to youngest) using the following way (assuming that L accesses address a and reads from store S in execution E):

1. L reads from the store buffer or atomic memory in E: We do not change M.
2. L reads from the invalidation buffer in E: Assume that S is overwritten by another store S_o in the atomic memory in E. We move L to be right before S_o in total order M.

After each step of the examination of loads, we can prove that the Inst-Order axiom still holds and that the Ld-Val axiom holds for all examined loads. It should be noted that moving a load will not affect the Ld-Val axiom for all other loads, so we only need to show that the Ld-Val axiom holds for the newly examined load L, and that the Inst-Order axiom holds between L and other instructions.

It is easy to prove for the first case, so we only focus on the second case. Consider M before we make the change. First note that

- S precedes S_o in M, and there is no other store to a in between.

Since L reads from the invalidation buffer, we have the following facts:

- S_o precedes L in M.
- There is no Reconcile from Pi which sits between S_o and L in M.

The CoWR property of the I^{2}E model indicates that

- Any store (other than S), which accesses a and precedes L in the program order, must precede S in M.

The above facts guarantee that after moving L to be right before S_o, the Ld-Val axiom is satisfied, and the Inst-Order axiom holds between L and all stores and fences.

For each load L' to address a that precedes L in the program order and reads from store S', the CoRR property of the I^{2}E model indicates that

- Either S' is just S or S' precedes S in M.

According to the way we adjust positions of loads in M, we know L' must precede S_o in M. Thus L' should still precede L in M after moving L, and the Inst-Order axiom also holds between L and all older loads in the program order. □

A.2. WMM Axiomatic Model $\subseteq$ WMM I^{2}E Model

Now we show that any program behavior allowed by the axiomatic model can be produced by the I^{2}E model.

Theorem 2. *WMM axiomatic model $\subseteq$ WMM* I^{2}E *model.*

Proof. We first describe an algorithm to operate the I^{2}E model to produce the program behavior of any combination of memory order, program order and read-from relations (i.e. $\langle <_{mo}, <_{po}, \xrightarrow{rf} \rangle$) that satisfies the axiomatic model. The algorithm begins with an empty set Z, and a queue Q which contains all memory instructions (including fences) in the memory order (i.e., the head of Q is the oldest instruction in $<_{mo}$). In each step of the algorithm, we perform the following actions:

- If the next instruction of some processor in the I^{2}E model is a non-memory instruction, then we perform the WMM-Nm operation to execute it.
- If the next instruction of some processor in the I^{2}E model is a store, then we perform the WMM-St operation to execute that store.
- Otherwise, if the next instruction of some processor in the I^{2}E model is a load in set Z, then we perform the WMM-Ld operation to execute that load.
- If neither of the above conditions applies, then we pop out instruction I from the head of Q and process it in the following way:
  - If I is a store, then we dequeue that store from the store buffer in the I^{2}E model.
  - If I is a fence, then we execute that fence in the I^{2}E model.
  - I must be a load in this case. If I is the next instruction to execute on its processor in the I^{2}E model, then we execute the load. Otherwise, we add I into set Z.

The algorithm terminates when both P and Q are empty, and there is no instruction to execute on any processor in the I^{2}E model. We will prove that in each step of the algorithm, we can indeed perform the operation in the I^{2}E model, and if we execute a load, the load gets the same result as indicated by the $\xrightarrow{rf}$ relation. It is easy to prove in case we execute non-memory instructions, stores and fences, so we only focus on the cases in which we execute loads.

First consider the case that we pop out a load instruction I from Q, which is the next instruction to execute on processor Pi. In this case, the algorithm will execute I in the I^{2}E model using the WMM-Ld operation. Assume I reads from store S in the $\xrightarrow{rf}$ relations. If S precedes I in $<_{mo}$, then I can read from the atomic memory in the I^{2}E model (because all stores preceding I in $<_{mo}$ have been processed in previous steps of the algorithm and written to the atomic memory, and the Ld-Val axiom ensures that there is no other store to the address of I between I and S in $<_{mo}$). Otherwise, S should precede I in the program order of processor Pi according to the Ld-Val axiom, and thus I can read the value of S from the store buffer in the I^{2}E model.

Next consider the case that we execute a load L for address a of processor Pi and L is in set Z. (L must have already been popped out from Q in a previous step of the algorithm.) Assume L reads from store S in the $\xrightarrow{rf}$ relations. If S is after L in $<_{mo}$ (i.e. $L <_{mo} S$), then L can read the value of S from the store buffer in the I^{2}E model. Otherwise, S must have been popped out from Q and have been written into the atomic memory of the I^{2}E model. In this case, consider the store S_o to a that is right after S in $<_{mo}$. If S_o has not been popped out from Q, then L reads the value of S from the atomic memory in the I^{2}E model. Otherwise, S_o must have overwritten S in the atomic memory of the I^{2}E model. In this case, L is able to read the value of S from the invalidation buffer for the following reasons:

1. L should precede S_o in $<_{mo}$ (otherwise L will read from S_o instead of S according to the Ld-Valaxiom).
2. Any store that precedes L in $<_{po}$ cannot be after S in $<_{mo}$ (otherwise L will not read from S according to the Ld-Valaxiom). Thus, in the I^{2}E model, the store buffer of Pi is always empty from the time that S is written to the atomic memory until L is executed. Therefore, the value of S is inserted into the invalidation buffer of Pi when S_o overwrites S in the atomic memory of the I^{2}E model, and no store older than L in Pi can remove S from the invalidation buffer.
3. Any Reconcile that precedes L in $<_{po}$ also precedes S_o in the memory order, so no Reconcile older than L in Pi

cannot remove S from the invalidation buffer.

4. Consider a load L' that precedes L in $<_{po}$ and reads from store S' in the $\xrightarrow{rf}$ relations. S' cannot be after S in memory order (otherwise L will read S' instead of S according to the Ld-Val axiom). Thus, in the I^{2}E model, S' cannot write the atomic memory after S does (i.e., the load result of L' is not more up-to-date than S), and the execution of L' cannot remove S from the invalidation buffer.

$\square$

By combining Theorems 1 and 2, we have proven the equivalence between the I^{2}E model and the axiomatic model of WMM.

Theorem 3. *WMM* I^{2}E *model* $\equiv$ *WMM axiomatic model.*

References

[1] "The risc-v instruction set," https://riscv.org/.

[2] "Splash-2x benchmarks," http://parsec.cs.princeton.edu/parsec3-doc.htm#splash2x.

[3] "Wwc+addrs test result in power processors," http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/ppc051.html#toc11.

[4] *Alpha Architecture Handbook, Version 4*. Compaq Computer Corporation, 1998. [Online]. Available: http://www.ece.cmu.edu/~ece447/s14/lib/exe/fetch.php?media=alphaahb.pdf

[5] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *computer*, vol. 29, no. 12, pp. 66–76, 1996.

[6] S. V. Adve and M. D. Hill, "Weak ordering a new definition," in *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI. ACM, 1990, pp. 2–14.

[7] J. Alglave, "A formal hierarchy of weak memory models," *Formal Methods in System Design*, vol. 41, no. 2, pp. 178–210, 2012.

[8] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli, "The semantics of power and arm multiprocessor machine code," in *Proceedings of the 4th workshop on Declarative aspects of multicore programming*. ACM, 2009, pp. 13–24.

[9] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig, "Software verification for weak memory via program transformation," in *Programming Languages and Systems*. Springer, 2013, pp. 512–532.

[10] J. Alglave and L. Maranget, *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ch. Stability in Weak Memory Models, pp. 50–66. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_6

[11] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 36, no. 2, p. 7, 2014.

[12] E. K. Ardestani and J. Renau, "Esesc: A fast multicore simulator using time-based sampling," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 448–459.

[13] ARM, *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, 2013.

[14] Arvind and J.-W. Maessen, "Memory model = instruction reordering + store atomicity," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 29–40.

[15] M. Batty, A. F. Donaldson, and J. Wickerson, "Overhauling sc atomics in c11 and opencl," *SIGPLAN Not.*, vol. 51, no. 1, pp. 634–648, Jan. 2016. Available: http://doi.acm.org/10.1145/2914770.2837637

[16] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell, "Clarifying and compiling c/c++ concurrency: from c++ 11 to power," in *ACM SIGPLAN Notices*, vol. 47, no. 1. ACM, 2012, pp. 509–520.

[17] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing c++ concurrency," in *ACM SIGPLAN Notices*, vol. 46, no. 1. ACM, 2011, pp. 55–66.

[18] C. Blundell, M. M. Martin, and T. F. Wenisch, "Invisifence: performance-transparent memory ordering in conventional multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 233–244.

[19] H.-J. Boehm and S. V. Adve, "Foundations of the c++ concurrency memory model," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 68–78.

[20] H.-J. Boehm and B. Demsky, "Outlawing ghosts: Avoiding out-of-thin-air results," in *Proceedings of the Workshop on Memory Systems Performance and Correctness*, ser. MSPC '14. New York, NY, USA: ACM, 2014, pp. 7:1–7:6. Available: http://doi.acm.org/10.1145/2618128.2618134

[21] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "The complexity of verifying memory coherence," in *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2003, pp. 254–255.

[22] P. Cenciarelli, A. Knapp, and E. Sibilio, "The java memory model: Operationally, denotationally, axiomatically," in *Programming Languages and Systems*. Springer, 2007, pp. 331–346.

[23] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: bulk enforcement of sequential consistency," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 278–289.

[24] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 14, no. 2. IEEE Computer Society Press, 1986, pp. 434–442.

[25] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell, "Modelling the armv8 architecture, operationally: Concurrency and isa," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL 2016. New York, NY, USA: ACM, 2016, pp. 608–621. Available: http://doi.acm.org/10.1145/2837614.2837615

[26] W. J. Ghandour, H. Akkary, and W. Masri, "The potential of using dynamic information flow analysis in data value prediction," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 431–442.

[27] K. Gharachorloo, A. Gupta, and J. L. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *Proceedings of the 1991 International Conference on Parallel Processing*, 1991, pp. 355–364.

[28] 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*. ACM, 1990, pp. 15–26.

[29] C. Gniady and B. Falsafi, "Speculative sequential consistency with little custom storage," in *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*. IEEE, 2002, pp. 179–188.

[30] J. R. Goodman, *Cache consistency and sequential consistency*. University of Wisconsin-Madison, Computer Sciences Department, 1991.

[31] D. Gope and M. H. Lipasti, "Atomic sc for simple in-order processors," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 404–415.

[32] C. Guiady, B. Falsafi, and T. N. Vijaykumar, "Is sc+ ilp= rc?" in *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*. IEEE, 1999, pp. 162–171.

[33] IBM, *Power ISA, Version 2.07*, 2013.

[34] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer, "A promising semantics for relaxed-memory concurrency," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: ACM, 2017, pp. 175–189. Available: http://doi.acm.org/10.1145/3009837.3009850

[35] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 690–691, 1979.

[36] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram, "Efficient sequential consistency via conflict ordering," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1. ACM, 2012, pp. 273–286.

[37] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," *ACM SIGOPS Operating Systems Review*, vol. 30, no. 5, pp. 138–147, 1996.

[38] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi, "Armor: defending against memory consistency model mismatches in heterogeneous architectures," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015, pp. 388–400.

[39] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. Martin, P. Sewell, and D. Williams, "An axiomatic memory model for power multiprocessors," in *Computer Aided Verification*. Springer, 2012, pp. 495–512.

[40] J.-W. Maessen, Arvind, and X. Shen, "Improving the java memory model using crf," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 1–12, 2000.

[41] J. Manson, W. Pugh, and S. V. Adve, "The java memory model," in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. New York, NY, USA: ACM, 2005, pp. 378–391. Available: http://doi.acm.org/10.1145/1040305.1040336

[42] L. Maranget, S. Sarkar, and P. Sewell, "A tutorial introduction to the arm and power relaxed memory models," http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf, 2012.

[43] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti, "Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 34. Washington, DC, USA: IEEE Computer Society, 2001, pp. 328–337. Available: http://dl.acm.org/citation.cfm?id=563998.564039

[44] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: x86-tso," in *Theorem Proving in Higher Order Logics*. Springer, 2009, pp. 391–407.

[45] A. Perais and A. Seznec, "Eole: Paving the way for an effective implementation of value prediction," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014, pp. 481–492.

[46] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 428–439.

[47] P. Ranganathan, V. S. Pai, and S. V. Adve, "Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models," in *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1997, pp. 199–210.

[48] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams, "Synchronising c/c++ and power," in *ACM SIGPLAN Notices*, vol. 47, no. 6. ACM, 2012, pp. 311–322.

[49] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding power multiprocessors," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 175–186.

[50] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave, "The semantics of x86-cc multiprocessor machine code," *SIGPLAN Not.*, vol. 44, no. 1, pp. 379–391, Jan. 2009. Available: http://doi.acm.org/10.1145/1594834.1480929

[51] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-tso: a rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, 2010.

[52] X. Shen, Arvind, and L. Rudolph, "Commit-reconcile and fences (crf): A new memory model for architects and compiler writers," in *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*. IEEE, 1999, pp. 150–161.

[53] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-end sequential consistency," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 524–535.

[54] R. Smith, Ed., *Working Draft, Standard for Programming Language C++*. http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4527.pdf, May 2015.

[55] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[56] SPARC International, Inc., *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., 1992.

[57] M. Vijayaraghavan, A. Chlipala, Arvind, and N. Dave, *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. Cham: Springer International Publishing, 2015, ch. Modular Deductive Verification of Multiprocessor Hardware Designs, pp. 109–127. Available: http://dx.doi.org/10.1007/978-3-319-21668-3_7

[58] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The risc-v instruction set manual. volume 1: User-level isa, version 2.1," *Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley*, May 2014. Available: https://people.eecs.berkeley.edu/~krste/papers/riscv-spec-v2.1.pdf

[59] D. L. Weaver and T. Gremond, *The SPARC architecture manual (Version 9)*. PTR Prentice Hall Englewood Cliffs, NJ 07632, 1994.

[60] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 266–277.

[61] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 24–36.