Cooking the Books: Formalizing JMM Implementation Recipes*

Gustavo Petri[1], Jan Vitek[2], and Suresh Jagannathan[1]

1 **Purdue University**
2 **Northeastern University**

Abstract

The Java Memory Model (JMM) is intended to characterize the meaning of concurrent Java programs. Because of the model's complexity, however, its definition cannot be easily transplanted within an optimizing Java compiler, even though an important rationale for its design was to ensure Java compiler optimizations are not unduly hampered because of the language's concurrency features. In response, the *JSR-133 Cookbook for Compiler Writers* [16], an informal guide to realizing the principles underlying the JMM on different (relaxed-memory) platforms was developed. The goal of the cookbook is to give compiler writers a relatively simple, yet reasonably efficient, set of reordering-based recipes that satisfy JMM constraints.

In this paper, we present the first formalization of the cookbook, providing a semantic basis upon which the relationship between the recipes defined by the cookbook and the guarantees enforced by the JMM can be rigorously established. Notably, one artifact of our investigation is that the rules defined by the cookbook for compiling Java onto Power are *inconsistent* with the requirements of the JMM, a surprising result, and one which justifies our belief in the need for formally provable definitions to reason about sophisticated (and racy) concurrency patterns in Java, and their implementation on modern-day relaxed-memory hardware.

Our formalization enables simulation arguments between an architecture-independent intermediate representation of the kind suggested by [16] with machine abstractions for Power and x86. Moreover, we provide fixes for cookbook recipes that are inconsistent with the behaviors admitted by the target platform, and prove the correctness of these repairs.

1998 ACM Subject Classification D.1.3 Concurrent Programming. D.3.1 Formal Definitions and Theory. F.3.1 Specifying and Verifying and Reasoning about Programs.

Keywords and phrases Concurrency; Java; Memory Model; Relaxed-Memory

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.999

1 Introduction

A decade ago, the semantics of concurrent Java programs, the Java Memory Model (JMM), was revised and redefined [18]. This revision, which was adopted as part of the official Java specification [14] had multiple purposes. First, it was intended to replace the previous specification which disallowed many common architectural and compiler optimizations of Java programs that were found in many state-of-the-art JVMs. Second, it formalized, using a rather complicated axiomatic semantics, the possible behaviors of concurrent Java programs. Having a formalization, the *DRF-guarantee* [1] – establishing that programs that do not have data races (DRF) in their Sequentially Consistent (SC) semantics, can only exhibit SC behavior, even when executed on non-SC hardware – could be formally proved [5].

* This work is supported by the National Science Foundation under grants CCF-1216613 and CCF-1318227.

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 999–1048

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Unfortunately, due to the complexity of the formalism, many desirable properties of the semantics were not met, and many undesirable properties were not prevented [20]. In light of these shortcomings, there is currently a community effort to better understand and reconsider the definition of the JMM [13].

A testament to the complexity of the JMM specification is the *The JSR-133 Cookbook for Compiler Writers* [16], an informal guide to implementing the JMM in different computer architectures. This document is intended to aid Java compiler writers to provide safe, reasonably efficient implementations, that nonetheless satisfy the JMM requirements. Unlike the JMM, the high-level semantics of Java concurrency is described operationally, in terms of memory instruction reorderings, thus defining the relaxed behaviors a program may exhibit, in a form suitable for reasoning about the correctness of compiler optimizations.

One of the reasons why the current JMM specification is so complex is that it attempts to uniformly capture the set of memory relaxations induced by both relaxed-memory platforms as well as common compiler optimizations deemed necessary to provide performant Java implementations. A recent effort [9] has considered an alternative approach, namely giving a semantics to Java that captures only the relaxations permitted by the TSO (Total Store Ordering) memory model found on x86 architectures [22]. One could attempt to implement this flavor of Java in weaker architectures such as Power [25], but this is a substantially more challenging exercise; simply retrofitting the TSO-aware semantics developed in [9] for Power would incur a high performance cost, necessitating injection of low-level synchronization operations between normal variable memory accesses to ensure TSO behavior.

The following question thus presents itself: what is the strongest memory model that would be both (1) efficiently implementable – not requiring synchronization at the low level for non-volatile variables – in architectures as relaxed as Power, and (2) yet have a tractable formal semantics amenable to the rigorous proofs needed to demonstrate compiler correctness arguments à la CompcertTSO [21]? As a corollary, we also wish to understand the semantics of current *implementations* of JVMs with respect to the memory model they support. JVMs ensure their implementations are consistent with the JMM by making conservative decisions on synchronization and shared-memory accesses. We are interested in determining if there is a middle ground between the behaviors admitted by relaxed-memory architectures and the JMM, which provides a more tractable, perhaps stronger semantics than the JMM, but which provides nonetheless an acceptable performance for modern Java applications.

At first glance, it would appear that many of these questions are answered in [16]. However, given that [16] is an informal document, with no clear – let alone formal – semantic definitions, and no guarantees that the rules defined are correct, we consider a methodology to formalize the semantics induced by its "recipes", deriving as an important by-product, a provable validation that some of the minimal guarantees required by the JMM are satisfied. In this sense, our goals are broadly similar to [6], which provides a provably correct compilation strategy of C++11 into Power. However, operating as we do in the Java context, our challenges are substantially different; not only must our formalization cope uniformly with different architectures given the platform agnostic definition of the JMM, but it must also deal explicitly with a number of JMM-specific features such as its support for "roach-motel" reorderings, explicitly established as a requirement of the JMM [18]. These issues make it infeasible to seamlessly transplant the results from approaches like [6]. Unlike [6], we do not provide a concrete compilation strategy – indicating for example that a fence has to be emitted *immediately after* a volatile store – but rather indicate minimal constraints that must be satisfied by any such strategy – for example a fence must exists in between a volatile store and any subsequent memory action –. We do this to allow flexibility to capture systems

like Octet [7] where the fences might be added in garbage collection safe points for example. This follows the spirit of [16].

Perhaps surprisingly, the relation between [18] and [16] has not been considered formally before, and notably our results show that the rules implied by [16] for Power are at odds with the requirements of the JMM.[1] Concretely, while working on our proofs we found a counter-example to the DRF requirement of the JMM if the rules of [16] are used for Power. The example in question is the infamous IRIW litmus test – reproduced below – considering only *volatile* variables instead of normal variables. In Java, concurrent conflicting accesses to volatile variables are not considered to form a data races. We display the example below with each thread in a column, and we consider that the object o is shared among all threads, with volatile fields v and w. Variables starting with r are local to each thread.

$o.v = o.w = 0$ & both fields are volatile

$o.v = 1;$	$\|$	$o.w = 1;$	$\|$	$r0 = o.v;$	$\|$	$r2 = o.w;$
	$\|$		$\|$	$r1 = o.w;$	$\|$	$r3 = o.v;$

Is $r0 = r2 = 1$ & $r1 = r3 = 0$ allowed?

The behavior in question cannot be produced by an SC semantics. However, this behavior is possible in Power [25]. Moreover, inserting lwsync Power barriers in between the two reads in the reading threads would not prevent this behavior from happening as documented in [25, 8].[2] Unfortunately, lwsync was the barrier of choice recommended by [16] when our work was started to prevent this relaxation.[3] We tried this Java example in a Power 7 machine, and were able to reproduce the erroneous behavior in the two different JVM's we tested[4], indicating that this is not simply a theoretical inconvenience, but a critical dichotomy between desired semantics and implementations. Our discussions with several VM implementors indicate that (a) the cookbook was heavily used as a crucial reference, given the complexity of the official specification, and (b) some implementations are aware of the bug noted above, while others are not; given the subtlety and complexity of the JMM, and the lack of consensus among implementors on a proper implementation strategy, the anecdotal evidence makes clear that a cookbook-like document is quite necessary, with a provably correct version even more so. To highlight the subtlety of the issues involved, parts of the cookbook were in fact changed [6] in response to advances in the formalization of processor memory models (e.g., [17, 25]), but in the absence of a formal definition, those changes did not remediate the issues noted here.

The contributions of this paper are:

1. We formalize (operationally) the semantics of compiling concurrency features in Java as described by [16] into the x86 and Power relaxed-memory architectures.
2. Notably, our high-level semantics propagates the relaxations admitted by Power to normal Java variables. Our choice to propagate Power semantics for normal variables into a high-level semantics is motivated by the fact that any stronger semantics at the high-level would impose synchronization operations for normal variables in Power. This would most

[1] Of course, many of the architectures considered in [16] were not as well understood by the research community at the time it was published.
[2] The behavior manifests because lwsync imposes no constraints on when the stores performed by the first two threads become visible to the readers.
[3] At the time of this writing, December 2014, the cookbook has been updated based on our findings.
[4] The example failed on IBM's JVM and Jikes RVM. Similar examples failed in Fiji's realtime JVM implementation on ARM 7.

likely greatly degrade the performance of concurrent Java programs in Power, which is on the one hand unnecessary given the JMM definition, and on the other hand not required by [16]. We consider this to be a minimal performance requirement for any acceptably efficient implementation of the JMM on Power. Given that Power is one of the weakest architectural memory models yet studied, we consider that our high-level semantics serves as an upper bound of how strong a JMM could be, without penalizing weak architectures like Power.

3. [16] uses an intermediate representation to express memory operation reorderings. We formalize this intermediate representation, and prove a simulation argument between source-level programs and programs compiled to this IR, and establish an inclusion property between behaviors allowed by the target architectures (x86 and Power) and this IR.
4. We additionally formalize the different target architectures we consider in the same framework, and when the rules of [16] are correct we prove that they are so. Additionally, we identify the rules that *do not produce correct implementations*, and propose corrections, which we then prove sufficient to enforce the expected high-level semantics (e.g., volatile variables must exhibit SC semantics). Our findings have been propagated to the current revision of [16].
5. To the best of our knowledge, ours is the first formal attempt to relate the high-level semantics of the JMM with low-level architectural implementations as described in [16].

We emphasize that *it is not our aim to provide a new memory model for Java* – which presumably would be weaker than our IR to allow for additional relaxations –, instead we are simply using [16] as a harness for how existing implementations manifest the rules of the JMM. In short, we are documenting the ad-hoc model that implementors use.

The remainder of the paper is organized as follows. The next section provides additional motivation and gives an overview of our approach and proof structure. Section 3 presents the syntax and single-threaded semantics of the core language studied in terms of an abstract machine that admits weak memory behavior. Section 4 extends the semantics to deal with concurrency features found in *cookbook-high*, a language that supports normal references (with a Power-style relaxation) and volatile references, as well as locks. Section 5 describes a low-level intermediate representation (*cookbook-low*) that implements memory barriers, and does not support volatile references. We define the conditions necessary to compile cookbook-high into this IR, and provide a simulation result between executions in the low and high languages. Section 6 defines the semantics for x86 and Power in terms of our core language, and establishes a correspondence between cookbook-low programs and programs compiled to these architectures. Related work and conclusions are given in Sections 7 and 8, resp.

2 Overview

Consider the requirements of the JMM with respect to the implementation of synchronization operations, and its relation to the rules provided by the cookbook document. A driving principle of the JMM, dubbed the *roach motel semantics* [18], is that increasing the synchronization of a program cannot *add* new observable behaviors to it. The synchronization operations, formally defined in [18], include locking and volatile memory access operations.[5]

[5] Thread creation, termination, and object initialization are also synchronization operations, but they are not relevant for the ideas discussed here.

1st Op.\2nd Op.	Normal Load / Store	Volatile Load / Lock	Volatile Store / Unlock
Normal Load / Store			No
Volatile Load / Lock	No	No	No
Volatile Store / Unlock		No	No

Table 1 High-level Roach-Motel Semantics Rules

The roach motel principle implies that all program transformations which increase the *happens-before* [15] relation of the program – which captures the causality relation of a program enforced through its synchronization actions (locks and volatile accesses) – should be allowed by the memory model. Pragmatically, this means that normal memory operations following a volatile write can be reordered before it, since the resulting program imposes additional synchronization not required by the former. Similarly, normal memory operations preceding a volatile read can be reordered after it. An argument similar to the case of volatile writes applies to unlock operations (a monitorexit in Java bytecote), and the same is true for volatile reads with respect to lock operations (monitorenter). These observations justify the first table presented in the cookbook [16], that describes the reorderings possible at the highest-level considered in that document. We reproduce this table in Table 1. The table indicates that two operations can be reordered if the cell is empty, and that they cannot if the cell is marked "No"; the first operation is sampled from the rows and the second one from the columns. Data and control dependencies are assumed to be respected by the cookbook tables. Then, for instance two normal memory operations on different references can be freely reordered, but any two synchronization operations cannot.

Intermediate Representation

Before presenting the requirements for the implementation of these operations for a specific architecture, the cookbook introduces an intermediate low-level representation in which memory operations are not assumed to have inherent ordering semantics; instead, operation ordering is imposed through the use of additional barrier – or fence – instructions, that guard the kind of reordering permissible between two memory accesses. At this level, volatile memory operations are assumed to be "implemented" using normal memory operations – corresponding to the operations provided by the ISA of the target architectures –, and the ordering constraints of Table 1 have to be enforced rather than assumed. This intermediate representation assumes that there is a different barrier to prevent the reordering of any two kind of memory operations if the barrier is emitted by the code in between these two accesses. For example, two read operations can be prevented from being reordered if a *Load to Load* barrier (LoadLoad) is emitted in between them by the thread. Similar fences exist between stores and loads, loads and stores and two consecutive stores. Table 2 presents the kind of barriers that must be introduced in this intermediate representation to enforce the semantics of Java delineated by Table 1. This is the second table of [16].

Given the lack of a precise semantics for normal load and store instructions, it is difficult to formally establish the correspondence between the high- and low-level versions. Our first contribution (section 4) is the definition of a tractable semantics for these two layers that enables the correctness proof of the rules relating these two tables.

In [16], tables are presented which for each architecture relate the instructions from the corresponding ISA to implement each of the barriers described above. We postpone the discussion of how we establish the correspondence between low-level cookbook barriers

1st Op.\2nd Op.	Normal Load	Normal Store	Volatile Load/Lock	Volatile Store/Unlock
Normal Load				LoadStore
Normal Store				StoreStore
Volatile Load/Lock	LoadLoad	LoadStore	LoadLoad	LoadStore
Volatile Store/Unlock			StoreLoad	StoreStore

Table 2 Low-level Cookbook: Barriers Required

and architecture-specific instructions until section 6. Notably this final table provides only translations for the barrier instructions, leaving normal operations unrestricted.

Store-Atomicity Relaxation

A limitation of the cookbook document is that the argumentation is made in terms of operation reorderings, which disregards *store-atomicity* – or write-atomicity – which allows write operations to be propagated to different threads at different times, a relaxation permitted by some architectures, including Power and ARM [25, 3]. One could imagine providing a semantics which considers reordering of operations as the only source of relaxations in the style of the TSO, PSO and RMO [28] memory models. However, this would be insufficient to capture certain important relaxations that are permitted by architectures with weaker memory models; the following example (similar to the example WRC of [25]) illustrates this issue.

$$\begin{array}{c} o.f = o'.f = \texttt{NULL} \\ \hline o.f = o' \quad \| \quad (o.f).f = o \quad \| \quad r0 = o'.f; \\ \qquad\quad \| \qquad\qquad\quad \| \quad r1 = r0.f \\ \hline r0 = o \;\&\; r1 = \texttt{NULL}? \end{array} \tag{1}$$

This program has three threads, which share two objects o and o', each with a single field f initially `NULL`. We assume that the type of the field f is the same as the type of o and o'. In the result indicated at the end, we have that $r0 = o$, therefore it must be the case that the read of $o'.f$ in the third thread returns the object o. Indeed this is possible if the first thread executes first, then the second thread dereferences $o.f$ obtaining o' and after that it writes o into $o'.f$. Now we can fullfil the read of $r0$ in the third thread. It is obvious that the read of $r0.f$ in the third thread cannot happen before $r0$ has obtained its value through the previous read. Therefore these two reads cannot be reordered. In that case, if the only source of relaxation is reordering, the read $r0.f$ which in actuality is a read of $o.f$ must see the value o', since all reorderings are prevented through data dependencies. This final result cannot be produced by a *reordering-only* memory model. However, this is a possible behavior in Power, since a write-atomicity relaxation could mean that the write of the first thread is only propagated to the second, but not the third thread, allowing the third thread to read `NULL` for $r1$. To admit such behavior, it is then necessary to introduce write-atomicity relaxations existent in Power within the (low-level) cookbook semantics to avoid over-synchronizing normal memory accesses. This motivates the semantics we present in section 4.

Proof Structure

Figure 1 illustrates the overall proof structure that we follow in our work. At the top level, we have the semantics of the JMM as described in [18], or rather the improved version of [20].

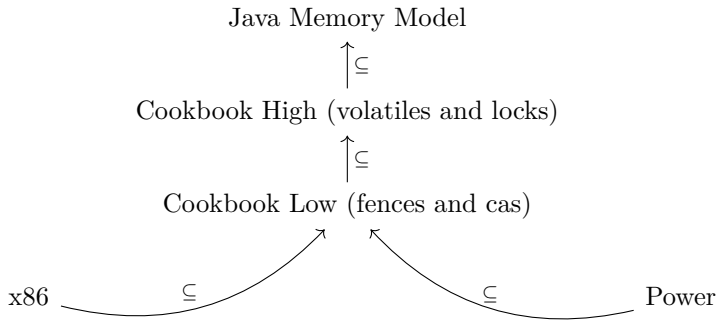

Figure 1 Models above PowerMM exhibit Write-Atomicity Relaxations

Below this level, we have a high-level, architecture-agnostic, operational semantics which adopts Power semantics for normal variables, and SC semantics for volatile variables and locks. We denote this semantics by cookbook-high. One level down, we have the intermediate representation that contains only normal memory accesses and barriers. Finally, at the bottom of the figure we have the semantics of the Power and x86 architectures, of which Power offers a more relaxed semantics. We establish a backwards simulation between the high and low-level definitions of the cookbook, show that high-level cookbook semantics respects the JMM, and that our low-level cookbook definition properly captures the behaviors admitted by x86 and Power.

In the next section we will introduce our unifying language and semantic artifacts that make our simulations and proofs possible.

3 A Language

We define an operational semantics for a relaxed memory framework inspired by [8]. We describe different memory models using the same basic syntax for the different languages discussed in the previous section. For example, the semantics of the top-level language that we consider (akin to a Java bytecode) includes a treatment of volatile variables; volatiles, however, do not appear in lower-level languages. Additionally, the languages that model specific architectures add barrier instructions, which are not present in higher-level languages.

We first introduce the main languages considered:

Cookbook-High: This is the top-level language, and is intended to model the memory-model related concerns of a Java-bytecode like language. As such it contains: **a.** *Normal references*, which are the normal fields and variables of a Java program; **b.** *Volatile references*, which are variables subject to strict ordering and visibility constraints as dictated by [18]. For example, volatile variables should have SC semantics when considered in isolation; and, **c.** *Locks* used to represent the mutually-exclusive lock in a Java monitor. As with volatile references, locks are subject to strict visibility and ordering constraints [18]. We do not concern ourselves with a proper definition of "object" in this work, since this notion is irrelevant for the memory-model issues being studied. We reiterate that this language, keeping in the spirit of Java bytecode, contains no barrier (or fence) operations.

Cookbook-Low: This is an intermediate representation used in [16] to establish the barriers needed to impose ordering at lower-levels to *implement* the semantics of cookbook-high. This language serves to bridge the gap between the cookbook-high language and multiple target architectures, each of which have their own ISA that implements different barriers

and memory models. The main difference between cookbook-low w.r.t. cookbook-high are: **a.** cookbook-low has no notion of volatile reference. References that are volatile at the top-level are considered as ordinary normal references in this level, and **b.** cookbook-low provides barrier instructions, to prevent the local reordering of certain type of instructions. For instance, the syntax $\langle \texttt{l}|\texttt{s} \rangle$ in this language is used to guarantee that a load memory accesses (i.e. a read) issued by a thread prior to the execution of the barrier cannot be performed later than any store operation (i.e. write) issued after the barrier by the same thread. This language includes all the barriers presented in [16]. This is a common intermediate representation that is subsequently compiled to each different target architecture.

POWER (PPC): This language, although expressed as a functional core, is intended to model the Power architecture as documented in [25, 8]. The main differences between this language and cookbook-low are: **a.** this language implements the actual barrier instructions of Power, that is lwsync and sync as opposed to the more abstract barriers of cookbook-low[6], and **b.** unlike the barriers of cookbook-low, these barriers have a global meaning (potentially involving more than one thread) as documented in [25, 8].

TSO (x86): This language represents the TSO memory model of x86 processors [22]. It has only one barrier instruction, namely mfence. It also has a cas() instruction, used in the implementation of locks.

3.1 Syntax

The syntax of our core language is a simple first-order language equipped with references, volatile references for cookbook-high, locks, conditionals and boolean values and operators. As mentioned earlier, we have different barrier instructions at different levels of our languages, which are part of the syntax. The syntax of our language is in ANF [11] to simplify the definition of evaluation contexts, and sequencing of operations, which is irrelevant for our purposes. Our source level syntax involves the following semantic categories:

$$x, y \in \mathcal{V}ar \qquad p \in \mathcal{P}tr \qquad \mathsf{p} \in \mathcal{V}ptr \qquad \ell \in \mathit{Locks}$$

$\mathcal{V}ar$ represent variables with substitution semantics, $\mathcal{P}tr$ represent normal pointers, $\mathcal{V}ptr$ represent *volatile* pointers, and $\mathcal{L}ocks$ represent locks. When presenting the semantics we will add another category for placeholder values, akin to the semantics of futures as in [10].

We present the syntax of our language in Figure 2. As customary, the values of the language, represented by the set $\mathcal{V}al$, include variables (a convenience to have our language in ANF), booleans, references, volatile references, locks and a special unit value () to represent termination. Expressions in our language, represented by the set $\mathcal{E}xpr$ contain all values, and boolean operators, which are implicit and ranged over by the metavariable $\oplus$. Commands include the standard skip, sequence and a simple let-binding construct to evaluate complex boolean expressions. Moreover, we have standard conditionals, reference creation, assignment, and dereferencing.

For the cookbook-high language, we include a number of commands (in red) which operate on volatile variables and locks. Unlike Java, cookbook-high has special syntax to operate over volatile references. We do not consider programs that use the volatile syntax to access

[6] We only consider a subset (namely the ones needed by [16]) of available Power barriers in our development.

$$
\begin{array}{rcl}
v \in \mathcal{V}al & ::= & x \mid tt \mid ff \mid p \mid \mathtt{p} \mid \ell \mid () \\
e \in \mathcal{E}xpr & ::= & v \mid e \oplus e \\
\mathtt{c} \in \mathcal{L} & ::= & v \mid \mathtt{skip} \mid \mathtt{c}_0 \,;\, \mathtt{c}_1 \\
& \mid & \mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{c} \\
& \mid & \mathtt{if}\ v\ \mathtt{then}\ \mathtt{c}_0\ \mathtt{else}\ \mathtt{c}_1\ \mathtt{fi} \\
& \mid & \mathtt{let}\ x = \mathtt{new}\ v\ \mathtt{in}\ \mathtt{c} \\
& \mid & v_0 := v_1 \\
& \mid & \mathtt{let}\ x = !y\ \mathtt{in}\ \mathtt{c} \\
& \mid & \mathtt{let}\ x = \mathtt{new}_v\ v\ \mathtt{in}\ \mathtt{c} \\
& \mid & v_0 :=_v v_1 \\
& \mid & \mathtt{let}\ x = !_v y\ \mathtt{in}\ \mathtt{c} \\
& \mid & \mathtt{lock}\ x \mid \mathtt{unlock}\ x \mid b
\end{array}
$$

$$
\begin{array}{rcl}
b \in \mathcal{S}ync_{\mathrm{CBL}} & ::= & \langle \mathtt{s} | \mathtt{l} \rangle \mid \langle \mathtt{l} | \mathtt{l} \rangle \\
& \mid & \langle \mathtt{l} | \mathtt{s} \rangle \mid \langle \mathtt{s} | \mathtt{s} \rangle \\
b \in \mathcal{S}ync_{\mathrm{TSO}} & ::= & \mathsf{mfence} \\
& \mid & \mathtt{let}\ x = \mathtt{cas}(y)\ \mathtt{in}\ \mathtt{c} \\
b \in \mathcal{S}ync_{\mathrm{PPC}} & ::= & \mathsf{lwsync} \\
& \mid & \mathsf{sync}
\end{array}
$$

Figure 2 Syntax. The statements in red are present in Cookbook-High only. The synchronization statements b in blue depend on the language being considered.

normal references, nor the converse.[7] This assumption will be made precise when presenting the cookbook-high memory semantics. The first three commands in red create, store and read a volatile reference, respectively. The commands `lock` x and `unlock` x assume that the variable x will adopt a lock value at runtime and roughly represent the `monitorenter` and `monitorexit` bytecode instructions of Java.

Finally, the languages at levels lower than cookbook-high contain a number of barrier instructions. We add these to the syntax, and will restrict their usage according to the language being considered. In the case of `x86`, we consider a `cas`() instruction that atomically queries and updates a memory location. In our restricted language, the argument x to `cas`(x) is assumed to be a reference, which if it is initially *ff* will be set to *tt*, returning *tt*; otherwise, the operation has no effect, and returns *ff*.

3.2 Semantics

Our semantics follows [8] which models states in terms of configurations with rewriting rules that dictate how programs can reduce or perform effects on that state. The state is a triple, $(\sigma, \delta, \mathsf{T})$ comprising: **1.** a store σ which is a mapping from references (and volatile references) to their current value, **2.** a thread system T, which is a mapping from thread identifiers – sampled from the set $\mathcal{T}id$ and ranged with the metavariable t – to runtime commands (i.e. elements of $\mathcal{L}$ where some variables have been substituted by runtime values). These commands represent the continuation of the original command of thread t, and finally **3.** what we shall call a *temporary store*, which is represented by the metavariable δ. A temporary store is a sequence[8] of pending *memory operations* associated with their originating thread identifiers. They represent operations that have been issued by the threads, but not yet fully synchronized with the memory (σ), and the other threads. In essence, an operation in δ, is an operation that has been issued, perhaps has been partially executed, potentially made visible to some threads but not all, which the memory system will have to commit at a later

[7] In Java the distinction between volatile and normal memory accesses can be made through the *type* of a field. Here we assume a distinguished syntactic form, and implicitly consider only programs that make consistent assumptions in the syntax and runtime about volatile accesses.

[8] We use the notation $\delta \cdot \delta'$ for sequence concatenation, and ϵ to denote the empty sequence.

Memory Operations

$$\begin{array}{rcl} \mathsf{mo} \in \mathcal{O}pr & ::= & \mathtt{t}^{v}_{\rho} \mid \mathtt{e}^{v}_{\rho} \\ & \mid & \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\varrho,\mu} \mid \mathsf{rd}_{\varrho,\mu} \mid \overline{\mathsf{rd}_{\rho,\mu}} \\ & \mid & \mathbf{vwr}_{\varrho,\mu} \mid \mathbf{vrd}_{\varrho,\mu} \mid \overline{\mathbf{vrd}_{\rho,\mu}} \\ & \mid & \mathtt{lk}_{\mu} \mid \mathtt{ul}_{\mu} \mid b \end{array}$$

Single-Thread Step

$$\frac{[\![r]\!](\sigma,\delta,t) = (\mathsf{mo},\mathsf{c})}{(\sigma,\delta,(t,\mathbf{E}[r])\|\mathsf{T}) \xrightarrow[t]{\mathsf{mo}} (\sigma,\delta\cdot(t,\mathsf{mo}),(t,\mathbf{E}[\mathsf{c}'])\|\mathsf{T})}$$

$$[\![r]\!](\sigma,\delta,t) = \begin{cases} (\tau,()) & \text{if } r = \mathtt{skip} \\ (\tau,\mathsf{c}) & \text{if } r = v\,;\mathsf{c} \\ (\mathtt{t}_{\rho},\mathsf{c}_0) & \text{if } r = (\mathsf{if}\ \rho\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1) \\ (\mathtt{e}_{\rho},\mathsf{c}_1) & \text{if } r = (\mathsf{if}\ \rho\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1) \\ (\mathtt{t}^{tt},\mathsf{c}_0) & \text{if } r = (\mathsf{if}\ tt\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1) \\ (\mathtt{e}^{ff},\mathsf{c}_1) & \text{if } r = (\mathsf{if}\ ff\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1) \\ (\tau,\mathsf{c}[x\leftarrow\mu]) & \text{if } r = \mathtt{let}\ x = \mu\ \mathtt{in}\ \mathsf{c} \\ (\mathsf{rd}_{\varrho,\rho},\mathsf{c}[x\leftarrow\rho]) & \text{if } r = \mathtt{let}\ x = !\varrho\ \mathtt{in}\ \mathsf{c}\ \&\ \text{free } \rho \text{ in } \delta \\ (\mathbf{vrd}_{\varrho,\rho},\mathsf{c}[x\leftarrow\rho]) & \text{if } r = \mathtt{let}\ x = \mathbf{!}_{v}\varrho\ \mathtt{in}\ \mathsf{c}\ \&\ \text{free } \rho \text{ in } \delta \\ (\mathsf{wr}^{\{t\}}_{\varrho,\mu},()) & \text{if } r = (\varrho := \mu) \\ (\mathbf{vwr}_{\varrho,\mu},()) & \text{if } r = (\varrho :=_{v} \mu) \\ (\mathsf{wr}^{\{t\}}_{p,\mu},\mathsf{c}[x\leftarrow p]) & \text{if } r = \mathtt{let}\ x = \mathtt{new}\ \mu\ \mathtt{in}\ \mathsf{c}\ \&\ \text{free } p \text{ in } \delta,\sigma \\ (\mathbf{vwr}_{p,v},\mathsf{c}[x\leftarrow\mathsf{p}]) & \text{if } r = \mathtt{let}\ x = \mathtt{new}_{v}\ \mu\ \mathtt{in}\ \mathsf{c}\ \&\ \text{free } \mathsf{p} \text{ in } \delta,\sigma \\ (b,()) & \text{if } r = b \\ (\mathtt{lk}_{\mu},()) & \text{if } r = \mathtt{lock}\ \mu \\ (\mathtt{ul}_{\mu},()) & \text{if } r = \mathtt{unlock}\ \mu \end{cases}$$

Figure 3 Single thread semantics. Thread composition (no memory actions).

point in time. Different rules for committing the operations in the temporary store allow us to define different *ordering* and *visibility* components that collectively define a memory model.

The dynamic semantics captured by this framework thus allows (a) threads to contribute (by executing their code) memory operations into the temporary store δ; and (b) the memory system to take care of *committing* these operations in the main memory σ, and synchronizing the memory operations of all threads.

Intra-thread Semantics

The contribution of each thread to the temporary store is presented as a reduction semantics decomposing each command into a *reducible expression* (redex) and an evaluation context. Our semantics preserves the invariant that at runtime each command can be decomposed into a unique evaluation context and a redex, or it contains an error, in which case we disregard the computation. Below is our definition of evaluation contexts, and evaluation context application.

$$\mathbf{E} ::= [] \mid \mathbf{E}\,;\mathsf{c} \qquad\qquad \mathbf{E}[\mathsf{c}] = \begin{cases} \mathsf{c} & \text{if } \mathbf{E} = [] \\ \mathbf{E}'[\mathsf{c}]\,;\mathsf{c}' & \text{if } \mathbf{E} = \mathbf{E}'\,;\mathsf{c}' \end{cases}$$

To the set of values presented in Figure 2 we add a category of runtime *placeholder* values used to delay the effects of reads without blocking the execution of subsequent instructions

in the program (these are called *Identifiers* in [8]).

$$\rho \in \mathcal{P}lHold \qquad \varrho \in \mathcal{P}lHoldPtr \;::=\; p \mid \mathsf{p} \mid \rho \qquad \mu \in \mathcal{P}hold\mathcal{V}al \;::=\; v \mid \rho$$

The set $\mathcal{P}lHold$ contains an infinite set of values – ranged over by ρ – that will be used at runtime to stand in place of an actual value returned by a read (similar to the semantics of futures in [10]). We will use the metavariable ϱ to range over placeholder values or reference values (both normal and volatile), which can appear in the left hand side of an assignment, or in a dereferencing instruction. We will use the more general metavariable μ to range over placeholders or any other value, which can appear in the program anywhere a value is expected.

Using these placeholders, each time a read redex is to be reduced, we do not immediately query the memory, but instead generate a read operation in the temporary store, where a fresh placeholder takes the place of the value that will be queried at a later point in the execution. Hence, our semantics preserves the invariant that any placeholder value appearing in a program must have been generated by a prior read operation, which is still uncommitted (i.e. the memory system has not returned a value for it yet). Once the read operation is committed, all placeholders are replaced with the appropriate value. Finally, we remark that *placeholders are not storable values*. Thus, writes to memory can only be committed if they contain an actual value in $\mathcal{V}al$. The top left of Figure 3 defines the contributions of each thread to the temporary store.

Our language permits a light form of branch speculation, achieved by predicting a branch when a placeholder value is the condition of an `if` instruction. An operation $\mathtt{t}^{v}_{\rho}$ represent the speculation of a *then* branch (where any of v or ρ could be absent). This operation is contributed by a thread taking a *then* branch, where the condition of the branch is a placeholder (ρ). If the conditional is evaluated on a value (v) instead, $\mathtt{t}^{v}$ is produced, which is not really a speculation, since the value of the condition is known. In the case of a real speculation, at a later point, the value of placeholder ρ will be substituted (say by v), and the operation will be substituted by $\mathtt{t}^{v}_{\rho}$ (we need to keep track of the placeholder to enforce ordering constraints w.r.t. speculations). Clearly, if v is a *ff* this will be a mis-speculation since we are considering a then branch, and we will simply disregard the miss-predicted trace. Note that when the condition of a branch is a placeholder, a branch can always proceed given that the placeholder is later substituted with a value that matches the prediction (i.e. $\mathtt{t}^{tt}_{\rho}$). Similarly $\mathtt{e}^{v}_{\rho}$ represents the speculation of an *else* branch.

The second line of operations corresponds to memory operations on *normal* references. For the time being, we disregard the action $\overline{\mathsf{rd}_{\rho,\mu}}$ which will be considered when presenting the semantics of the memory system. The first action, $\mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\varrho,\mu}$ is a write action emitted by a thread. The components ϱ and μ are the reference (or placeholder) that is being written, and the value (or placeholder) that is being written into it. Additionally, to capture the semantics of atomicity relaxations of Power, each write operation in the temporary store contains a set $\mathcal{W}$ of *thread identifiers*, indicating which thread can currently see this write – even before the write is executed in the memory. For instance, a write event $\mathsf{wr}^{\mathcal{W}\cup\{t\}}_{p,v}$ can be seen by reads of thread t before reaching the memory (σ). When threads emit write events the set $\mathcal{W}$ contains only the current thread $\{t\}$ (simulating store-buffers à la TSO). Similarly, the set $\mathcal{I}$ represents a set of placeholder values whose originating read has been fulfilled by this write. This component is only used to give semantics to Power barriers. Throughout the paper, whenever any of these sets are empty we will omit them for readability. Read memory operations $\mathsf{rd}_{\varrho,\mu}$ correspond to the issuance of a read operation, on reference (or placeholder) ϱ whose return value (or placeholder) is μ. Initially, μ will always be a fresh

$$
\begin{array}{rcll}
(\sigma, (t, \mathtt{t}_{tt}) \cdot \delta, \mathsf{T}) & \underset{\text{SC}}{\hookrightarrow} & (\sigma, \delta, \mathsf{T}) & \\
(\sigma, (t, \mathsf{e}_{ff}) \cdot \delta, \mathsf{T}) & \underset{\text{SC}}{\hookrightarrow} & (\sigma, \delta, \mathsf{T}) & \\
(\sigma, \delta_0 \cdot (t, \mathsf{wr}_{p,v}) \cdot \delta_1, \mathsf{T}) & \underset{\text{SC}}{\hookrightarrow} & (\sigma[p \leftarrow v], \delta_0 \cdot \delta_1, \mathsf{T}) & \left[\delta_0 \overset{\curvearrowleft}{\text{SC}} (t, \mathsf{wr}_{p,\rho})\right] \\
(\sigma, \delta_0 \cdot (t, \mathsf{rd}_{p,\rho}) \cdot \delta_1, \mathsf{T}) & \underset{\text{SC}}{\hookrightarrow} & (\sigma, \delta_0 \cdot (\delta_1[\rho \leftarrow v]), \mathsf{T}[\rho \leftarrow v]) & \left[\delta_0 \overset{\curvearrowleft}{\text{SC}} (t, \mathsf{rd}_{p,\rho}) \;\&\; \sigma(p) = v\right] \\
(\sigma, \delta_0 \cdot (t, \mathbf{vwr}_{p,v}) \cdot \delta_1, \mathsf{T}) & \underset{\text{SC}}{\hookrightarrow} & (\sigma[p \leftarrow v], \delta_0 \cdot \delta_1, \mathsf{T}) & \left[\delta_0 \overset{\curvearrowleft}{\text{SC}} (t, \mathbf{vwr}_{p,\rho})\right] \\
(\sigma, \delta_0 \cdot (t, \mathbf{vrd}_{p,\rho}) \cdot \delta_1, \mathsf{T}) & \underset{\text{SC}}{\hookrightarrow} & (\sigma, \delta_0 \cdot (\delta_1[\rho \leftarrow v]), \mathsf{T}[\rho \leftarrow v]) & \left[\delta_0 \overset{\curvearrowleft}{\text{SC}} (t, \mathbf{vrd}_{p,\rho}) \;\&\; \sigma(p) = v\right]
\end{array}
$$

Figure 4 Sequential Consistent Memory: Load, Store and Conditionals. (Side conditions included between brackets.)

placeholder value(and indeed, each placeholder value in the temporary store must come as the result of a read event). Later, the memory system will substitute this placeholder by a concrete value read from memory.

The third line presents exactly the same operations, this time generated by instructions that operate on *volatile* references. Importantly, the action $\mathbf{vwr}_{\varrho,\mu}$ does not share the components $\mathcal{W}$, and $\mathcal{I}$ with its normal counterpart, a consequence of the fact that volatile variables only have SC semantics, and therefore will be processed directly from the memory.

Finally, the operations $\mathtt{lk}_\mu$ and $\mathtt{ul}_\mu$ are contributions of lock and unlock instructions on the lock value μ. In the dynamics of the program, μ needs to be a lock value (ℓ), or a placeholder value that will be eventually substituted by a lock value. The last operation, b, represents barrier instructions, and corresponds to the synchronization actions of Figure 2. With this definition of memory operations, we can precisely define temporary stores as being sequences of pairs of a thread identifier and a memory operation: $\delta \in (\mathcal{T}id \times \mathcal{O}pr)^*$.

The semantics of intra-thread (sequential) computation is given in the rest of Figure 3. At the bottom of Figure 3 we presents the definition of $[\![r]\!](\sigma, \delta, t)$, a function that takes as input a redex r, a store σ, a temporary store δ, and a thread id t returning a pair containing: **1.** the memory operation to be added to the temporary store (where τ represents a reduction with no memory operation), and **2.** the continuation of the command to be executed. Note that the only place where this definition uses the store σ and the temporary store δ is in choosing a *fresh placeholder* value for reads and a *free location* for reference creation.

The rule Single-Thread Step at the top right of the figure, shows how each thread contributes to the temporary store in a full configuration. We take $(t, \mathsf{c}) \parallel \mathsf{T}$ to be the extension of T to $\mathsf{T}' = \mathsf{T}[t \leftarrow \mathsf{c}]$.

Before proceeding with the semantics of the memory system, we remark that for lack of space, we defer the treatment of locks to Appendix B of the extended version of the paper [23]. Importantly, the treatment of locks is similar to the treatment of volatile variables as hinted by Table 1. Therefore, most arguments that apply to a volatile load apply to a lock instruction, and similarly a volatile store with an unlock. We only consider locks in the paper in the tables, to show that their treatment corresponds with the respective volatile operations.[9]

[9] The treatment of locks can be found in the extended version of this paper [23].

Sequential Consistency

We illustrate how to encode sequential consistency (SC) using this semantics in Figure 4. We indicate between square brackets "[]" side conditions that apply to each of the rules on the right hand side of the figure. To describe the semantics, we define a conflict relation between memory operations which is necessary to describe when two memory operations are (in-)dependent, and therefore cannot be reordered. For SC, the conflict relation includes all pairs.

$$\forall \mathsf{mo}, \mathsf{mo}',\ \ \mathsf{mo} \#_{\text{SC}} \mathsf{mo}'$$

The memory semantics uses a *commutativity predicate* between a temporary store and a pair of a thread and a memory operation. This predicate states when a memory operation can bypass the operations in the given temporary store. For SC, commutativity is characterized by the following requirement stating that actions of the same thread cannot bypass each other in the temporary store:

$$\delta \overset{\curvearrowleft}{\text{SC}} (t, \mathsf{mo}) \iff \forall (t, \mathsf{mo}') \in \delta,\ \neg((t, \mathsf{mo}') \#_{\text{SC}} (t, \mathsf{mo}))$$

In other words, in δ there are no other operations by thread t. Other memory models will have different commutativity predicates, and we anticipate that only in the case of Power, this predicate will involve more than the simple conflict relation (#). Notice that since SC does not have write atomicity relaxations we have no rules that can extend the set $\mathcal{W}$ of writes, nor the set $\mathcal{I}$ of placeholders, which we omitted. It is not hard to see that this semantics imposes SC since all read operations are performed, in-order, from the store σ.

4 Cookbook Semantics

Table 3 captures the conflict relation induced by [16] (extending Table 1 with our memory operations) and we will use it to parameterize the semantics of the cookbook-high language. That table does not impose ordering constraints between two normal memory accesses (and indeed no memory barriers are systematically added between these), which are then considered to be as relaxed as the target architecture allows for normal memory accesses. Of all the target architectures that we consider in this work, the weakest is Power. We will then assume that at the cookbook-high semantics, the behaviors allowed by Power memory operations are propagated for memory operations on normal Java references. We notice that while it is possible to enforce a stricter semantics by adding fences in between normal memory accesses, this would likely severely impact the performance of even sequential programs, a clearly undesirable result.

The resulting semantics then adheres to Power behavior for normal memory accesses (cf. [8]), and SC for volatiles. This is what we refer to as cookbook-high, and it is what we use to prove our correctness results. We concede that a weaker semantics for non-volatile variables could be considered, as it is indeed the case in the JMM [18], at the expense of more complicated and subtle reasoning to prove soundness of low-level implementations.

In this work, we consider a strict interpretation of the rules of [16]. We use relational notation to capture the information found in these tables. We write $\mathsf{mo} \#_{\text{CH}} \mathsf{mo}'$ to signify that a pair of memory operations mo and mo' have a conflict, if and mo defines a row and mo' defines a column, and the matching entry in the table has a conflict symbol #, or the condition in the entry is met by the memory operations (up to the obvious substitutions of formal parameters; for example $\mathsf{wr}_p \#_{\text{CH}} \mathsf{wr}_p$). This conflict relation will be used to know

1st\2nd	$\mathsf{rd}_{p'}$	$\mathsf{rd}_{\rho'}$	$\overline{\mathsf{rd}_{\rho'}}$	$\mathsf{wr}_{p'}$	$\mathsf{wr}_{\rho'}$	$\mathbf{vrd}_{\mathsf{p}'}$	$\overline{\mathbf{vrd}_{\mathsf{p}'}}$	$\mathbf{vwr}_{\mathsf{p}'}$	$\mathtt{lk}_{\ell'}$	$\mathtt{ul}_{\ell'}$
rd_{p}				$p = p'$	#			#		#
rd_{ρ}				#	#			#		#
$\overline{\mathsf{rd}_{\rho}}$								#		#
$\mathsf{wr}_{\overline{p}}^{\mathcal{I}}$	$p = p'$	#	$\rho' \in \mathcal{I}$	$p = p'$	#			#		#
wr_{ρ}	#	#	#	#	#			#		#
$\mathbf{vrd}_{\mathsf{p}}$	#	#	#	#	#	#	#	#	#	#
$\overline{\mathbf{vrd}_{\mathsf{p}}}$	#	#	#	#	#	#	#	#	#	#
$\mathbf{vwr}_{\mathsf{p}}$						#	#	#	#	#
$\mathtt{lk}_{\ell}$	#	#	#	#	#	#	#	#	#	#
$\mathtt{ul}_{\ell}$						#	#	#	#	#

Table 3 Cookbook-High Conflict Relation ($\#_{\mathrm{CH}}$).

when two actions of the same thread can commute in the temporary store with each other, denoted $(t, \mathsf{mo}) \overset{\curvearrowleft}{\mathrm{CH}} (t, \mathsf{mo}')$.

However, the fact that this semantics has write-atomicity relaxations as explained before implies that to preserve a consistent semantics we need to impose constraints between different threads on the commutativity of operations. For instance, a read action that sees a write before the latter has been made visible to all threads – a *read-early* action – cannot be accepted as performed by the issuing thread – *committed* – before the write is performed made visible to all threads. Hence, we overload the conflict relation ($\#$) to pairs of a thread and a memory operation. The minimal requirements for this relation are presented using the notation $\#_{\blacktriangleleft}$:

$$\varrho = \varrho' \text{ or } \varrho \in \mathcal{P}lHold \text{ and } t' \in \mathcal{W} \text{ or } \mathcal{I} \neq \emptyset \neq \mathcal{I}' \Rightarrow \begin{array}{c}(t, \mathsf{wr}_{\varrho}^{\mathcal{W},\mathcal{I}}) \ \#_{\blacktriangleleft} \ (t', \mathsf{wr}_{\varrho}^{\mathcal{W}',\mathcal{I}'}) \\ (t, \mathsf{wr}_{\varrho}^{\mathcal{W},\mathcal{I}}) \ \#_{\blacktriangleleft} \ (t', \mathsf{rd}_{\varrho'})\end{array} \tag{2}$$

$$(t, \mathsf{wr}^{\mathcal{W},\mathcal{I}\cup\{\rho\}}) \ \#_{\blacktriangleleft} \ (t', \overline{\mathsf{rd}_{\rho}}) \tag{3}$$

$$\varrho = \varrho' \text{ or } \varrho \in \mathcal{P}lHold \Rightarrow (t, \mathsf{rd}_{\varrho}) \ \#_{\blacktriangleleft} \ (t, \mathsf{wr}_{\varrho'}) \tag{4}$$

$$\varrho = \varrho' \text{ or } \varrho \in \mathcal{P}lHold \text{ and } t' \in \mathcal{W} \cup \{t\} \Rightarrow (t, \mathsf{wr}_{\varrho}^{\mathcal{W}}) \ \#_{\blacktriangleleft} \ (t', \mathsf{wr}_{\varrho'}) \tag{5}$$

$$\begin{array}{c}(t, \overline{\mathsf{rd}_{\rho}}) \ \#_{\blacktriangleleft} \ (t, \mathsf{t}_{\overline{\rho}}) \text{ and } (t, \overline{\mathsf{rd}_{\rho}}) \ \#_{\blacktriangleleft} \ (t, \mathsf{e}_{\overline{\rho}}) \\ (t, \mathsf{t}_{\overline{\rho}}) \ \#_{\blacktriangleleft} \ (t, \mathsf{wr}) \text{ and } (t, \mathsf{e}_{\overline{\rho}}) \ \#_{\blacktriangleleft} \ (t, \mathsf{wr})\end{array} \tag{6}$$

Condition (2) implies the obvious data dependencies between a write action and a subsequent action on the same reference by the same thread. Notice that a placeholder can potentially represent any reference, and hence, when the target of a write is undefined, any action by the same thread potentially conflicts with it. Moreover, the condition states that if a write action by t has been made visible to t', then this write action will conflict with subsequent actions on the same reference (with a conservative over-approximation for placeholders) by thread t'. Finally, it establishes that a write that has been seen early ($\mathcal{I} \neq \emptyset$) conflicts with any other action on the same reference (cf. placeholder), either after or before ($\mathcal{I}' \neq \emptyset$). Condition 3 establishes a natural constraint between an early write, and any early read that used this write. Condition 4 is similar to the first constraint established by 2, except that it operates on a read followed by a write of the same thread. Condition 5 requires that writes that potentially target the same reference be kept in order if the first has been made visible to the thread issuing the second. Finally, condition 6 requires that a speculation action be ordered

(SC) Speculation Commit
$$(\sigma, (t, \mathsf{mo}) \cdot \delta, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\sigma, \delta, \mathsf{T}) \qquad \left[\mathsf{mo} \in \{\mathsf{t}_\rho^{tt}, \mathsf{e}_\rho^{ff}\}\right]$$

(VW) Volatile Write
$$(\sigma, \delta_0 \cdot (t, \mathbf{vwr}_{\mathsf{p},v}) \cdot \delta_1, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\sigma[\mathsf{p} \leftarrow v], \delta_0 \cdot \delta_1, \mathsf{T}) \qquad \left[\delta_0 \overset{\curvearrowleft}{\text{CH}} (t, \mathbf{vwr}_{\mathsf{p},v})\right]$$

(VR) Volatile Read
$$(\sigma, \delta_0 \cdot (t, \mathbf{vrd}_{\mathsf{p},\rho}) \cdot \delta_1, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\sigma, \delta_0 \cdot (t, \overline{\mathbf{vrd}_{\rho,v}}) \cdot (\delta_1[\rho \leftarrow v]), \mathsf{T}[\rho \leftarrow v]) \qquad \left[\begin{array}{c}\sigma(\mathsf{p}) = v \\ \delta_0 \overset{\curvearrowleft}{\text{CH}} (t, \mathbf{vrd}_{\mathsf{p},\rho})\end{array}\right]$$

(VRC) Volatile Read Commit
$$(\sigma, \delta_0 \cdot (t, \overline{\mathbf{vrd}_{\rho,v}}) \cdot \delta_1, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\sigma, \delta_0 \cdot \delta_1, \mathsf{T}) \qquad \left[\delta_0 \overset{\curvearrowleft}{\text{CH}} (t, \mathbf{vrd}_{\rho,v})\right]$$

(NW) Normal Write
$$(\sigma, \delta_0 \cdot (t, \mathsf{wr}_{p,v}^{\mathcal{W},\mathcal{I}}) \cdot \delta_1, \mathsf{T}) \underset{\text{MM}}{\hookrightarrow} (\sigma[p \leftarrow v], \delta_0 \cdot \delta_1, \mathsf{T}) \qquad \left[\begin{array}{c}\delta_0 \overset{\curvearrowleft}{\text{MM}} (t, \mathsf{wr}_{p,v}^{\mathcal{W},\mathcal{I}}) \\ \text{MM} \in \{\text{CH}, \text{CL}, \text{PPC}, \text{TSO}\}\end{array}\right]$$

(NR) Normal Read
$$(\sigma, \delta_0 \cdot (t, \mathsf{rd}_{p,\rho}) \cdot \delta_1, \mathsf{T}) \underset{\text{MM}}{\hookrightarrow} (\sigma, \delta_0 \cdot (\delta_1[\rho \leftarrow v]), \mathsf{T}[\rho \leftarrow v]) \qquad \left[\begin{array}{c}\sigma(p) = v \\ \delta_0 \overset{\curvearrowleft}{\text{MM}} (t, \mathsf{rd}_{p,\rho}) \\ \text{MM} \in \{\text{CH}, \text{CL}, \text{PPC}, \text{TSO}\}\end{array}\right]$$

(EW) Write Early
$$(\sigma, \delta_0 \cdot (t, \mathsf{wr}_{\varrho,v}^{\mathcal{W},\mathcal{I}}) \cdot \delta_1, \mathsf{T}) \underset{\text{MM}}{\hookrightarrow} (\sigma, \delta_0 \cdot (t, \mathsf{wr}_{\varrho,v}^{\mathcal{W}',\mathcal{I}}) \cdot \delta_1, \mathsf{T}) \qquad \left[\begin{array}{c}\mathcal{W} \subset \mathcal{W}' \\ \text{MM} \in \{\text{CH}, \text{CL}, \text{PPC}, \text{TSO}\}\end{array}\right]$$

(ER) Read Early
$$(\sigma, \delta_0 \cdot (t, \mathsf{wr}_{\varrho,\mu}^{\mathcal{W},\mathcal{I}}) \cdot \delta_1 \cdot (t', \mathsf{rd}_{\varrho,\rho}) \cdot \delta_2, \mathsf{T}) \underset{\text{MM}}{\hookrightarrow} (\sigma, \delta_0 \cdot (t, \mathsf{wr}_{\varrho,\mu}^{\mathcal{W},\mathcal{I}\cup\{\rho\}}) \cdot \delta_1 \cdot (t', \overline{\mathsf{rd}_{\rho,\mu}}) \cdot (\delta_2[\rho \leftarrow \mu]), \mathsf{T}[\rho \leftarrow \mu])$$
$$\left[\begin{array}{c}t' \in \mathcal{W} \\ \delta_1 \overset{\curvearrowleft}{\text{MM}} (t', \mathsf{rd}_{\varrho,\rho}) \\ \delta_0 \overset{\curvearrowleft}{\text{MM}\mathcal{S}ync} (t', \mathsf{rd}_{\varrho,\rho}) \\ \text{MM} \in \{\text{CH}, \text{CL}, \text{PPC}, \text{TSO}\}\end{array}\right]$$

(ERC) Commit Read Early
$$(\sigma, \delta_0 \cdot (t, \overline{\mathsf{rd}_{\rho,v}}) \cdot \delta_1, \mathsf{T}) \underset{\text{MM}}{\hookrightarrow} (\sigma, \delta_0 \cdot \delta_1, \mathsf{T}) \qquad \left[\begin{array}{c}\delta_0 \overset{\curvearrowleft}{\text{MM}} (t, \overline{\mathsf{rd}_{\rho,v}}) \\ \text{MM} \in \{\text{CH}, \text{CL}, \text{PPC}, \text{TSO}\}\end{array}\right]$$

(FN) Fence
$$(\sigma, \delta_0 \cdot (t, b) \cdot \delta_1, \mathsf{T}) \underset{\text{MM}}{\hookrightarrow} (\sigma, \delta_0 \cdot \delta_1, \mathsf{T}) \qquad \left[\begin{array}{c}\delta_0 \overset{\curvearrowleft}{\text{MM}} (t, b) \\ \text{MM} \in \{\text{CL}, \text{PPC}, \text{TSO}\}\end{array}\right]$$

Figure 5 A high-level semantics for the Java cookbook (induced by Power).

w.r.t. the read action that originated the value of the conditional; and that write actions (following [25]) cannot bypass prior speculative branching actions. Since volatile references are not subject to write-atomicity relaxations, their constraints are fully defined by Table 3. In particular, many of the constraints in $\#_{\blacktriangleleft}$ take a conservative approach when relating placeholder values, whose target references are not known (e.g. Equation 2). However, when relating a normal memory access and a volatile access in cookbook-high, even if one or both of them have placeholder values we know that they could not conflict since $\mathcal{P}tr \cap \mathcal{V}ptr = \emptyset$. Therefore, all the conditions requiring $\varrho = \varrho'$ or $\varrho \in \mathcal{P}lHold$ as a precondition do not apply if one memory access is volatile and the other is not. The commutativity predicate of the cookbook-high language is thus:

$$\delta \overset{\curvearrowleft}{\text{CH}} (t, \mathsf{mo}) \iff \forall\ (t', \mathsf{mo}') \in \delta,\ \neg(t', \mathsf{mo}')\ \#_{\blacktriangleleft}\ (t, \mathsf{mo}) \text{ and } t = t' \Rightarrow \neg(\mathsf{mo}'\ \#_{\text{CH}}\ \mathsf{mo})$$

Figure 5 shows the rules capturing memory model behavior for cookbook-high. These rules are only concerned with the memory, and follow a judgment of the form

$$(\sigma, \delta, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\sigma', \delta', \mathsf{T}')$$

where the arrow is annotated with the memory model relation (defining a corresponding commutativity) being considered. Notice that many of the rules in Figure 5 concern multiple memory models. Now we are only concerned with the ones that refer to CH (i.e. all but (FN)). The rule (SC) simply establishes that a conditional speculation operation, whose condition has been validated (i.e. $\mathtt{t}_\rho^{tt}$ or e_ρ^{ff}) can be removed from the temporary store when it reaches the front. Note that the placeholder substitution operation does not eliminate the placeholder (i.e. $\mathtt{t}_\rho[\rho \leftarrow v] = \mathtt{t}_\rho^v$ as opposed to $\mathtt{t}_v$) since we need to keep record of the placeholder value for the definition of commutativity. The rule (VW) and which refers to a volatile write is identical to the write rule for SC of Figure 4. It reflects the fact that volatile writes have SC semantics in cookbook-high. The rules (VR) and (VRC), which can always be applied immediately one after the other, mimic the read rule of Figure 4, again reflecting the fact that volatile reads are always satisfied from the memory, and have SC semantics. However, unlike in Figure 4 the (VR) rule does not directly remove the read operation $(t, \mathbf{vrd}_{\mathsf{p},\rho})$, instead replacing it with the "read mark" memory operation $(t, \overline{\mathbf{vrd}_{\rho,v}})$ where v is the current value of p in the store. This read mark memory operation can then (perhaps immediately) be removed by rule (VRC) to mimic the SC read rule. Thus, the read mark operation serves as a marker indicating that a read operation, say $\mathbf{vrd}_{\mathsf{p},\rho}$ has been performed (i.e. the value has been queried from the memory), and the placeholder ρ has been substituted by the value v, but the operation has not yet been removed from the temporary store, instead simply replaced for $\overline{\mathbf{vrd}_{\rho,v}}$ which through the commutativity predicate might limit the applicability of subsequent memory operations in the temporary store. Clearly, this marker is unnecessary for volatile references. We include it to simplify the simulation argument between cookbook-high and cookbook-low (presented in the next section). On the contrary, for normal references $\overline{\mathsf{rd}_{\rho,v}}$ is used to limit the commutativity of subsequent operations.[10]

Rules(NW) and (NR) represent the commitment of a normal reference write and read, resp. These rules resemble their SC counterparts, except that the write operation $\mathsf{wr}_{p,v}^{\mathcal{W},\mathcal{I}}$ might have been propagated to other threads (the threads in the $\mathcal{W}$ set) and it might have already been used to satisfy some reads in the temporary store early (the reads whose placeholders are in $\mathcal{I}$). Otherwise this rule is identical to the SC version up to the new definition of the commutativity predicate ($\overset{\frown}{\text{CH}}$). Similarly, a read that has not yet been performed (through a read-early action (ER)) can be performed, querying the store, as long as it commutes with all other operations in the temporary store before it.

Perhaps the most interesting rules are (EW) and (ER) for early-writes and early-reads. Recall that a write can be prematurely propagated to certain threads (i.e. before reaching the store). The component $\mathcal{W}$ of a memory write operation is a set of thread IDs that can immediately see this write in the temporary store. The (EW) rule extends the set to new threads. To be able to use these writes that are propagated through the temporary store, the rule (ER) can reduce a read action $\mathsf{rd}_{\varrho,\rho}$ by substituting the placeholders ρ that it generated by the value (or placeholder) written by the propagated write. As it is the case in (VR) the action does not immediately disappear from the temporary store, but a marker is added, with the placeholder that was substituted and the value (or placeholder) that it was substituted with. This marker is important because a read action can prevent the commutativity of subsequent operations, and if we were to simply remove it we lose that information. Moreover we note that the placeholder of the original read (ρ) is added to the set $\mathcal{I}$ of reads that were fulfilled early by the matching write. This again, is to preserve commutativity constraints as

[10] In this sense, it plays a fundamental role in the definition of Power barriers.

discussed above. To end this case we notice that one of the side-conditions of (ER) requires that $\delta_0 \ {}_{\text{MM}}\overset{\curvearrowleft}{\mathcal{S}ync}\ (t', \mathsf{rd}_{\varrho,\rho})$ which considers only the synchronization operations of δ_0 and for CH is defined as:

$$\delta_0 \ {}_{\text{CH}}\overset{\curvearrowleft}{\mathcal{S}ync}\ (t', \mathsf{rd}_{\varrho,\rho}) \iff \forall (t', \mathsf{mo}) \in \delta_0, \neg(\mathsf{mo} \in \{\mathbf{vrd}, \overline{\mathbf{vrd}}, \mathtt{lk}_\ell\})$$

Thus, the read action $(t', \mathsf{rd}_{\varrho,\rho})$ should not bypass synchronization actions by t' in δ_0 (i.e., in the temporary store before the write). This is because synchronization actions in δ_0 performed by t' could prevent the execution of the read (for example a pending volatile read by t'). The final rule (ERC) is similar to (NR) and serves to eliminate read markers from the temporary store.

The final judgment below is the obvious composition between the intra-thread semantics of Figure 3 and the memory semantics defined in this section.

$$\frac{\textsc{Composed Semantics} \quad (\sigma, \delta, \mathsf{T}) \xrightarrow[t]{\mathsf{mo}} (\sigma', \delta', \mathsf{T}') \quad \text{or} \quad (\sigma, \delta, \mathsf{T}) \underset{\text{MM}}{\hookrightarrow} (\sigma', \delta', \mathsf{T}')}{(\sigma, \delta, \mathsf{T}) \overset{\text{MM}}{\Longrightarrow} (\sigma', \delta', \mathsf{T}')}$$

▶ **Definition 1** (Cookbook-high Execution). A cookbook-high thread system T_H reaches a configuration $(\sigma, \delta, \mathsf{T}'_H)$ if starting from a default store σ_0, and the empty temporary store ϵ

$$(\sigma_0, \epsilon, \mathsf{T}_H) \overset{\text{CH}}{\Longrightarrow}^* (\sigma, \delta, \mathsf{T}'_H)$$

where $\overset{\text{CH}}{\Longrightarrow}^*$ denotes the transitive closure of the semantics $\overset{\text{CH}}{\Longrightarrow}$ considered as a relation.

As an example, let us reconsider the program motivating store-atomicity that we saw in section 2 in the syntax of cookbook-high where we use tuples as possible values.

$$p := p' \quad \| \quad \mathtt{let}\ x = !p\ \mathtt{in}\ x := p \quad \| \quad \begin{array}{l}\mathtt{let}\ x = !p'\ \mathtt{in} \\ \mathtt{let}\ y = !x\ \mathtt{in}\ (x, y)\end{array}$$

If we use thread names t_0, t_1 and t_2 for these threads we observe that by executing them in order we can reach a configuration with a temporary store of the form

$$(t_0, \mathsf{wr}_{p,p'}) \cdot (t_1, \mathsf{rd}_{p,\rho}) \cdot (t_1, \mathsf{wr}_{\rho,p'}) \cdot (t_2, \mathsf{rd}_{p',\rho'}) \cdot (t_2, \mathsf{rd}_{\rho',\rho''})$$

Then by a (EW) rule in the write of t_0 we can extend the visibility to t_1, and use (ER) in the read by t_1 obtaining

$$(t_0, \mathsf{wr}_{p,p'}^{\{t_0,t_1\},\{\rho\}}) \cdot (t_1, \overline{\mathsf{rd}_{\rho,p'}}) \cdot (t_1, \mathsf{wr}_{p',p}) \cdot (t_2, \mathsf{rd}_{p',\rho'}) \cdot (t_2, \mathsf{rd}_{\rho',\rho''})$$

We can now repeat these steps for the write of t_1 extending their visibility to t_2.

$$(t_0, \mathsf{wr}_{p,p'}^{\{t_0,t_1\},\{\rho\}}) \cdot (t_1, \overline{\mathsf{rd}_{\rho,p'}}) \cdot (t_1, \mathsf{wr}_{p',p}^{\{t_1,t_2\},\{\rho'\}}) \cdot (t_2, \overline{\mathsf{rd}_{\rho',p}}) \cdot (t_2, \mathsf{rd}_{p,\rho''})$$

And now we can see that the last read $(t_2, \mathsf{rd}_{p,\rho''})$ can proceed with a (NR) rule, and since the write $(t_0, \mathsf{wr}_{p,p'}^{\{t_0,t_1\},\{\rho\}})$ has not been made visible to t_2, this read will return the default value in memory (assumed to be NULL). This is a behavior that is possible in Power, and we propagate for normal variables in cookbook-high.

We can define the notion of *well-behaved* cookbook-high program to which we restrict our attention in the subsequent sections.

▶ **Definition 2** (Well-Behaved Cookbook-High). We say a command c of the cookbook-high language is well-behaved if when composed with any concurrent context, including other well-behaved threads, execution never leads to a temporary store δ with actions of the form:

- rd_p or wr_p where p is a volatile reference, or
- $\mathbf{vrd}_p$ or $\mathbf{vwr}_p$ where p is a normal reference.

JMM Guarantees

The semantics of the JMM [18, 20] is defined in terms of a *justification procedure* which commits actions of a hypothetical execution one by one. This semantic style is very different from the operational one introduced in this section. However, we have proved (in Appendix C of the extended version of this paper [23]) that every execution of cookbook-high corresponds to a legal execution of the JMM as redefined by [20].

▶ **Theorem 3.** *All the executions of cookbook-high as per the semantics presented in this section can be justified as per the formalization of the* JMM *of [20].*

Proof. (Sketch) Space limitations preclude us from presenting the formal definition of the JMM as presented in [20].[11] While the semantics of Cookbook-High is operational, and events are generated as the program is executed, the semantics of the JMM is axiomatic. In the JMM, assuming a hypothetical execution ξ one justifies the execution with a series of steps that – using other executions – justify each of the actions in ξ. Our proof consists of a process to justify a final execution ξ, but in our treatment, the final execution is not known ahead of time. Instead, we justify steps as they happen, and show that if the operational semantics of Cookbook-High makes progress, all of the generated steps can be justified. We show that axioms that each time a step of Cookbook-High happens, it can be committed by the axioms of the JMM, meaning that all the Cookbook-High actions are permisible actions of the JMM. This argument extended to full traces guarantees the statement of the theorem. ◀

This result is not surprising since the semantics of cookbook-high is much more restrictive than the intended semantics of the JMM. As a corollary we obtain that the guarantees that are respected by the JMM [5, 20] also hold for us.

▶ **Corollary 4** (JMM properties)**.** *The following properties hold for Cookbook-High:*

- *Cookbook-High respects the DRF guarantee.*
- *Cookbook-High prevents* out-of-thin-air *reads.*
- *The projection of the semantics of cookbook-high to volatile variables and locks respects the* SC *semantics.*

Proof. (Sketch) This proof is an immediate consequence of the previous theorem, and the fact that all of these properties hold for the JMM [20]. ◀

This is a consequence of the theorem above and [20]. Moreover, cookbook-high provides SC semantics for volatile variables and locks (with respect to each other). The results above are some fundamental requirements that the JMM should satisfy [18]. We emphasize that the non-trivial proof of the theorem above can be found in Appendix C of the extended version of the paper [23].

5 Cookbook-Low: Definition, Compilation, Simulation

Following [16] we present an intermediate representation to mediate the compilation and proofs between cookbook-high, and its implementations in the different architectures. To that end we define the cookbook-low language in Figure 2. Our first result is that the semantics of

[11] The definition and full proof can be found in the appendix of the extended version of this paper [23].

1st\2nd	$\mathsf{rd}_{p'}$	$\mathsf{rd}_{\rho'}$	$\overline{\mathsf{rd}_{\rho'}}$	$\mathsf{wr}_{p'}$	$\mathsf{wr}_{\rho'}$
rd_{p}				$p = p'$	#
rd_{ρ}				#	#
$\overline{\mathsf{rd}_{\rho}}$					
$\mathsf{wr}_{\overline{p}}^{\mathcal{I}}$	$p = p'$	#	$\rho' \in \mathcal{I}$	$p = p'$	#
wr_{ρ}	#	#	#	#	#

$$\mathsf{mo} \in \{\mathsf{rd}, \overline{\mathsf{rd}}\} \Rightarrow \begin{cases} \mathsf{mo} \ \# \ \langle \mathtt{ld}|\_\rangle \ \& \\ \langle \_|\mathtt{ld}\rangle \ \# \ \mathsf{mo} \end{cases}$$
$$\mathsf{wr} \ \# \ \langle \mathtt{st}|\_\rangle \ \& \ \langle \_|\mathtt{st}\rangle \ \# \ \mathsf{wr}$$
$$b \ \# \ b'$$

Table 4 Cookbook-Low Conflict Relation ($\#_{\text{CL}}$).

1st\2nd	rd	wr	**vrd**	**vwr**	$\mathtt{lk}_{\ell'}$	$\mathtt{ul}_{\ell'}$
rd				$\langle \mathtt{l}|\mathtt{s}\rangle$		$\langle \mathtt{l}|\mathtt{s}\rangle$
wr				$\langle \mathtt{s}|\mathtt{s}\rangle$		$\langle \mathtt{s}|\mathtt{s}\rangle$
vrd	$\langle \mathtt{l}|\mathtt{l}\rangle$	$\langle \mathtt{l}|\mathtt{s}\rangle$	$\langle \mathtt{l}|\mathtt{l}\rangle$	$\langle \mathtt{l}|\mathtt{s}\rangle$	$\langle \mathtt{l}|\mathtt{l}\rangle$	$\langle \mathtt{l}|\mathtt{s}\rangle$
vwr			$\langle \mathtt{s}|\mathtt{l}\rangle$	$\langle \mathtt{s}|\mathtt{s}\rangle$	$\langle \mathtt{s}|\mathtt{l}\rangle$	$\langle \mathtt{s}|\mathtt{s}\rangle$
$\mathtt{lk}_{\ell}$	$\langle \mathtt{l}|\mathtt{l}\rangle$	$\langle \mathtt{l}|\mathtt{s}\rangle$	$\langle \mathtt{l}|\mathtt{l}\rangle$	$\langle \mathtt{l}|\mathtt{s}\rangle$	$\langle \mathtt{l}|\mathtt{l}\rangle$	$\langle \mathtt{l}|\mathtt{s}\rangle$
$\mathtt{ul}_{\ell}$			$\langle \mathtt{s}|\mathtt{l}\rangle$	$\langle \mathtt{s}|\mathtt{s}\rangle$	$\langle \mathtt{s}|\mathtt{l}\rangle$	$\langle \mathtt{s}|\mathtt{s}\rangle$

Table 5 Cookbook-High to Cookbook-Low Barriers (H-L).

cookbook-low simulates the semantics of cookbook-high if the rules presented in Table 5 are respected when compiling from cookbook-high to cookbook-low. These rules indicate that in-between any two memory accesses by the same thread at the cookbook-high level, the first of the kind indicated in the rows and the second one of the kind indicated in the columns, there must be a barrier between the corresponding memory accesses in the cookbook-low level as indicated by that cell in the table mediating them. We omit the reference name, and other parameters of memory operations since they are unnecessary. Similarly, operations $\overline{\mathsf{rd}}$ and $\overline{\mathbf{vrd}}$, which are not directly emitted by threads are omitted, but their constraints are similar to those of rd and **vrd**, resp.

The main differences between cookbook-high and cookbook-low are that:

1. In cookbook-low there are no *volatile* references. Hence, we collapse the set $\mathcal{V}ptr$ of cookbook-high into $\mathcal{P}tr$. If $\mathcal{P}tr_H$ denotes the set of normal references at the cookbook-high level, and $\mathcal{P}tr_L$ the set of normal references at cookbook-high, then $\mathcal{V}ptr \cup \mathcal{P}tr_H \subseteq \mathcal{P}tr_L$. Recall cookbook-high require that $\mathcal{P}tr_H \cap \mathcal{V}ptr = \emptyset$, and therefore in cookbook-low there should be no confusion when considering $\mathcal{V}ptr$ as being part of $\mathcal{P}tr_L$.
2. The cookbook-low semantics includes barrier operations defined by $\mathcal{S}ync_{\text{CBL}}$ as given in Figure 2. These barriers impose the obvious conflict relation with respect to other memory accesses of the same thread.

The semantics of the memory component of cookbook-low uses the conflict relation $\#_{\text{CL}}$ of Table 4 to define commutativity. All the rules of Figure 5 not involving volatile references (that is (VW), (VR) and (VRC)) apply to cookbook-low. The only rule that was not explained in the previous section is (FN), which simply dictates when a barrier operation can be removed from the temporary store.

▶ **Definition 5** (Cookbook-Low Execution). A thread system T_L of cookbook-low reaches a configuration $(\sigma, \delta, \mathsf{T}'_L)$ if starting from a default store σ_0, and the empty temporary store ϵ

$$(\sigma_0, \epsilon, \mathsf{T}_L) \overset{\text{CL}}{\Longrightarrow}^{*} (\sigma, \delta, \mathsf{T}'_L)$$

5.1 Compilation

Following [16] we specify sufficient conditions to enforce the cookbook-high semantics, which in turn is a conservative approximation of the JMM. Table 5 indicates for each cell which barrier – if any – has to be inserted in between the cookbook-low memory accesses that implement the corresponding cookbook-high memory accesses. We use Table 5 as a function with two arguments corresponding to the first and second memory operation, with the result depicted within the corresponding cell, which we will write as $\text{H-L}\ (\mathsf{mo}, \mathsf{mo}') = b$. Therefore we write $\text{H-L}\ (\mathsf{mo}, \mathsf{mo}') = b$ if the cell in row mo and column mo' contains a barrier b.

Below we present an intuitive statement of our main theorem relating programs of Cookbook-High and Cookbook-Low. The formal statement, and its proof are relegated to Appendix A.

▶ **Theorem 6** (Cookbook-Low simulates Cookbook-High). *Given thread systems, T_H of Cookbook-High and T_L of Cookbook-Low related by related by the function induced by Table 5. Each time that a thread in the system T_L can take a step by the composed semantic, the thread system T_H can take a similar step leading to related configurations.*

Proof. (Sketch) The proof is based on the construction of a simulation relation between configurations of Cookbook-High and Cookbook-Low. In a nutshell, normal variables in Cookbook-High correspond to identical variables in Cookbook-Low, whereas volatile variables in Cookbook-High are mapped into normal variables of Cookbook-Low. The stores of both configurations are the same, and the temporary stores are strongly related, where the relation takes into account the effects that the barriers imposed by Table 5 have on the shape of the Cookbook-Low temporary store w.r.t. the shape of the corresponding Cookbook-High temporary store. Finally, the program code of Cookbook-High and Cookbook-Low are related by an auxiliary *well-compiled* relation which enforces that fences have been added to the Cookbook-Low programs in accordance with Table 5.

As in any backward simulation, the proof strategy consists in a case analysis of all the possible Cookbook-Low step, showing that starting in related Cookbook-Low and Cookbook-High configurations, a new Cookbook-High configuration can be reached by executing zero or more steps in the Cookbook-High semantics. ◀

This means that any behavior produced by the Cookbook-Low configuration can also be produced by the Cookbook-High configuration, proving that Table 5 induces a correct compilation from Cookbook-High to Cookbook-Low.

6 x86 and Power Simulation

In this section, we present the semantics of the x86 and Power architectures in our core language. These definitions are inspired by [22, 25].

6.1 x86

To define the semantics of x86 it suffices to consider Figure 5 where the commutativity relation variable $\overset{\curvearrowleft}{\text{MM}}$ is instantiated with $\overset{\curvearrowleft}{\text{TSO}}$ induced by the conflict relation $\#_{\text{TSO}}$ below.

$$\mathsf{mo} \#_{\text{TSO}} \mathsf{mo}' \iff (\mathsf{mo} \neq \mathsf{wr}) \ \vee \ (\mathsf{mo} \neq \overline{\mathsf{rd}}) \ \vee \ (\mathsf{mo} = \mathsf{wr}_p \ \wedge \ \mathsf{mo}' \neq \mathsf{rd}_p)$$

Moreover, we require that the write-early rule (EW) cannot propagate a write $(t, \mathsf{wr}_\varrho^{\mathcal{W},\mathcal{I}})$ to a set $\mathcal{W}$ larger than $\{t\}$. In other words, writes can only be read early by the thread that

$$\begin{array}{lcllcl} \texttt{x86}(\langle \texttt{l}|\texttt{l}\rangle) & = & \texttt{skip} & \texttt{x86}(\langle \texttt{s}|\texttt{s}\rangle) & = & \texttt{skip} \\ \texttt{x86}(\langle \texttt{l}|\texttt{s}\rangle) & = & \texttt{skip} & \texttt{x86}(\langle \texttt{s}|\texttt{l}\rangle) & = & \texttt{mfence} \end{array}$$

Figure 6 x86 Compilation Strategy.

$$\begin{array}{lcllcl} \texttt{PPC}(\langle \texttt{l}|\texttt{l}\rangle) & = & \texttt{sync} & \texttt{PPC}(\langle \texttt{s}|\texttt{s}\rangle) & = & \texttt{lwsync} \\ \texttt{PPC}(\langle \texttt{l}|\texttt{s}\rangle) & = & \texttt{lwsync} & \texttt{PPC}(\langle \texttt{s}|\texttt{l}\rangle) & = & \texttt{sync} \end{array}$$

Figure 7 PPC Compilation Strategy.

emitted them (in essence modeling the store-buffers of TSO [22]). Finally, we have only one barrier instruction that imposes the following orderings, where the last one is inconsequential, and one of the fences can be eliminated.

$$\mathsf{wr} \ \#_{\text{TSO}} \ \mathsf{mfence} \qquad \mathsf{mfence} \ \#_{\text{TSO}} \ \mathsf{rd} \qquad \mathsf{mfence} \ \#_{\text{TSO}} \ \mathsf{mfence}$$

The implementation of the cookbook-low barrier operations in x86 is given in Figure 6. We say that an x86 command c_x is well-compiled w.r.t. to a cookbook-low source command c_L if c_x is obtained from c_L by substituting all the barrier operations according to Figure 6. A similar definition relates a x86 temporary store δ_x and a cookbook-low temporary store δ_L.

▶ **Lemma 7.** *If an x86 temporary store δ_x is related by Figure 6 to a cookbook-low temporary store δ_L and there is a memory operation (t, mo) such that $\delta_x \overset{\curvearrowleft}{\text{TSO}} (t, \mathsf{mo})$, then it is also the case that $\delta_x \overset{\curvearrowleft}{\text{CL}} (t, \mathsf{mo})$*

Proof. Since all memory operations but barriers are the same in x86 and cookbook-low, we only need to consider if there exists a case where $(t', b) \overset{\curvearrowleft}{\text{TSO}} (t, \mathsf{mo})$ and $\neg((t', b') \overset{\curvearrowleft}{\text{CL}} (t, \mathsf{mo}))$ where b and b' are related by Figure 6. The only case where these condition could be matched is the one of mfence at the x86 level w.r.t. a $\langle \texttt{s}|\texttt{l}\rangle$ at the cookbook-low level. It is then evident, by considering the cases of Table 4 and the definition of $\#_{\text{TSO}}$ conflict above, that this case is not possible since $\langle \texttt{s}|\texttt{l}\rangle$ and mfence impose the exact same conflict relation. ◀

▶ **Theorem 8** (Cookbook-Low to x86 Simulation)**.** *Given a thread system T_L in cookbook-low, an x86 thread system obtained from T_L by substituting the barriers according to Figure 6. This substitution establishes a simulation between their x86 and cookbook-low semantics.*

Proof. Obvious consequence of Theorem 7 In combination with Theorem 3 and Theorem 6 we obtain an end to end argument for the rules of the cookbook.

6.2 Power

Power is the weakest architecture considered in this paper, and the motivation for the write atomicity relaxation in the cookbook-high language semantics. The semantics of Power is defined in Figure 5 where we instantiate the memory model to PPC (i.e. we consider the $\overset{\curvearrowleft}{\text{PPC}}$ reordering relation).

We allow all normal memory operations of the same thread to commute in Power as long as they respect the constraints imposed by the minimal conflict $\#_{\blacktriangleleft}$, with the additional constraint that read operations to the same reference cannot be reordered (i.e. $\mathsf{rd}_p \ \#_{\text{PPC}} \ \mathsf{rd}_p$). Unlike the barriers we have considered thus far, Power barriers can impose global constraints that order and add visibility restrictions on operations of different threads. This is referred to in [24] and in [25] as cumulativity effects.

We first introduce the constraints imposed on $\overset{\curvearrowleft}{\text{PPC}}$ by the sync barrier of Power encoded in the conflict relation $\#_{\text{PPC}}$, where we implicitly assume that barrier operations conflict with each other.

$$(t, \mathsf{wr}) \#_{\text{PPC}} (t, \mathsf{sync}) \#_{\text{PPC}} (t, \mathsf{wr}) \quad (7) \qquad (t, \overline{\mathsf{rd}}) \#_{\text{PPC}} (t, \mathsf{sync}) \#_{\text{PPC}} (t, \overline{\mathsf{rd}}) \quad (9)$$

$$(t, \mathsf{rd}) \#_{\text{PPC}} (t, \mathsf{sync}) \#_{\text{PPC}} (t, \mathsf{rd}) \quad (8) \qquad (t, \mathsf{wr}^{\mathcal{W}\cup\{t'\},\mathcal{I}}) \#_{\text{PPC}} (t', \mathsf{sync}) \quad (10)$$

Notice that Equation 9 imposes strong ordering constraints between sync operations and reads that perhaps have been performed early. More importantly, Equation 10 reflects a conflict between a write in thread t, which is visible to thread t', and a subsequent sync by t'. In this sense, sync is a very strong barrier, because it imposes ordering between any two operations of the same thread, and moreover, imposes ordering between the write operations that have been made visible to a thread, and the thread's own operations.

A much weaker barrier is lwsync, whose conflict and commutativity relations we define below. Unlike sync, the commutativity of lwsync might depend on a number of prior operations performed by the thread before (in particular w.r.t. the action $\overline{\mathsf{rd}}$ as we shall see). Therefore, the last rule below (14) is simply stated in terms of commutativity ($\overset{\curvearrowleft}{\text{PPC}}$) instead of conflict ($\#_{\text{PPC}}$)[12].

$$(t, \mathsf{wr}) \#_{\text{PPC}} (t, \mathsf{lwsync}) \#_{\text{PPC}} (t, \mathsf{wr}) \quad (11)$$

$$(t, \mathsf{rd}) \#_{\text{PPC}} (t, \mathsf{lwsync}) \;\&\; (t, \overline{\mathsf{rd}}) \#_{\text{PPC}} (t, \mathsf{lwsync}) \quad (12)$$

$$t = t' \text{ or } t' \in \mathcal{W} \Rightarrow (t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}) \#_{\text{PPC}} (t', \mathsf{lwsync}) \quad (13)$$

$$\left.\begin{array}{r} \delta = \delta' \cdot (t', \mathsf{lwsync}) \cdot \delta_3 \;\& \\ \delta' = \delta_0 \cdot (t, \mathsf{wr}_{\varrho,\mu}^{\mathcal{W}\cup\{t'\},\mathcal{I}\cup\{\rho'\}}) \cdot \delta_1 \cdot (t', \overline{\mathsf{rd}_{\rho',\mu}}) \cdot \delta_2 \;\& \\ \neg(\delta_0 \overset{\curvearrowleft}{\text{PPC}} (t, \mathsf{wr}_{\varrho,\mu}^{\mathcal{W}\cup\{t'\},\mathcal{I}\cup\{\rho'\}})) \end{array}\right\} \Rightarrow \neg(\delta \overset{\curvearrowleft}{\text{PPC}} (t', \mathsf{rd}_{\varrho',\mu'})) \quad (14)$$

Notice that lwsync *does* allow the commutativity of $(t, \mathsf{wr}_p) \cdot (t, \mathsf{lwsync}) \overset{\curvearrowleft}{\text{PPC}} (t, \mathsf{rd}_{p'})$ if $p \neq p'$. In other words, it does not prevent write-read reorderings (which is a typical capability of the TSO memory model). Secondly, we remark that Equation 13 is slightly stronger than the formalization of lwsync proposed in [25] (we follow [8]). However, as has been argued in [8], the behaviors that are not considered by this strengthening of lwsync have not been observed in the actual machines as reported by [25, 8, 3]. Finally, and perhaps the most complicated rule is given in Equation 14. This rules relates the constraints of an lwsync in between a $(t', \overline{\mathsf{rd}})$ operation and a (t', rd) operation. The rule states that the second read action can only commute with the preceding temporary store if the write that the early-read action saw (i.e. $(t, \mathsf{wr}_{\varrho,\mu}^{\mathcal{W}\cup\{t'\},\mathcal{I}\cup\{\rho'\}})$) is in condition to be immediately performed (the rule is stated as a contrapositive). It is precisely this behavior of lwsync that enables IRIW behavior even if lwsync barriers are present between the reads of the reader threads (see [8]).

We have presented the compilation rules for the barriers in cookbook-low shown in Figure 7. We notice that compared to [16] we have replaced the compilation of the barrier $\langle \mathtt{l}|\mathtt{l} \rangle$ from lwsync to sync. To see why this is necessary, consider the compiled version of the IRIW example, where all references are volatile. We obtain the following program.

$$\left[\, \mathsf{p} := tt \,\right] \parallel \left[\, \mathsf{p}' := tt \,\right] \parallel \left[\begin{array}{l} \mathtt{let}\ x = !p\ \mathtt{in} \\ \quad \mathsf{lwsync}\,; \\ \quad \mathtt{let}\ y = !p'\ \mathtt{in}\ (x, y) \end{array}\right] \parallel \left[\begin{array}{l} \mathtt{let}\ x = !p'\ \mathtt{in} \\ \quad \mathsf{lwsync}\,; \\ \quad \mathtt{let}\ y = !p\ \mathtt{in}\ (x, y) \end{array}\right]$$

[12] To ease our simulation proofs we have slightly changed the semantics of lwsync and the (ERC) rules as presented in [8]. These changes are of no consequence, and only instrumental in proving our simulations.

This program produces the relaxed behavior resulting in (tt, ff) for both threads, assuming that initially we have that p and p' hold ff in the store. We have tested a litmus Java example program, which contains only volatiles, in a Power V7 machine, and had been able to reproduce the relaxed behavior, which is clearly unacceptable according to [18], since volatiles have SC semantics. Moreover, this is a *data-race-free* program that violates the *DRF-guarantee*.

▶ **Lemma 9.** *For any well-formed temporary cookbook-low store* $\delta_{cl} \cdot (t, \mathsf{mo})$ *such that it is transformed into the Power temporary store* $\delta_{ppc} \cdot (t, \mathsf{mo})$ *by the translation of Table 7, we have that* $\delta_{cl} \overset{\curvearrowright}{\text{CL}} (t, \mathsf{mo})$ *implies* $\delta_{ppc} \overset{\curvearrowright}{\text{PPC}} (t, \mathsf{mo})$.

Proof. As it was the case with x86 (Theorem 7) we only need to consider the barrier memory operations since all other operations are the same. Again we assume by contradiction that it might be the case that $\delta_{ppc} \overset{\curvearrowright}{\text{PPC}} (t, \mathsf{mo})$ and $\neg(\delta_{cl} \overset{\curvearrowright}{\text{CL}} (t, \mathsf{mo}))$. Since sync enforces ordering between all possible memory accesses, and moreover, over memory operations of different threads, we have that the offending operations cannot have been compiled to sync. (We notice here that if we had compiled $\langle \mathtt{l}|\mathtt{l} \rangle$ to lwsync as in the original [16], we could have had to consider the relaxation allowed by Equation 14 discussed above, which leads to the volatile IRIW counter-example.) We are left with the $\langle \mathtt{l}|\mathtt{s} \rangle$ and $\langle \mathtt{s}|\mathtt{s} \rangle$ barriers. By Equation 11, Equation 12 and Equation 13 above, we have that lwsync enforces at least constraints as strong as those of $\langle \mathtt{l}|\mathtt{s} \rangle$ and $\langle \mathtt{s}|\mathtt{s} \rangle$ of cookbook-low, meaning that it must be the case that $\delta_{cl} \overset{\curvearrowright}{\text{CL}} (t, \mathsf{mo})$ and therefore concluding our proof. ◀

▶ **Theorem 10** (Cookbook-Low to Power Simulation)**.** *Given a thread system* T_L *in cookbook-low, a Power thread system is obtained from* T_L *by substituting the barriers according to Figure 7. This substitution establishes a simulation between their Power and cookbook-low semantics.*

Proof. Obvious consequence of Theorem 9.

About Power's lwsync

The semantics of lwsync considered in [25] is slightly weaker that the one considered here. In particular, an example where these two differ is presented in [25] under the name R01, which uses lwsync. Under that weaker interpretation of lwsync, Figure 7 would have to compile $\langle \mathtt{s}|\mathtt{s} \rangle$ into a sync for Power instead of the weaker lwsync. Unsurprisingly, that is the recommended implementation of sequentially consistent loads and stores in [6]. We clarify that this is *only* required for the implementation of *volatile* memory accesses (which are a lot less prevalent than normal memory accesses). Therefore this is unlikely to degrade the performance dramatically, and we could adopt it as a safe default. Theorem 9 is evidently true under this modification. This may or may not be considered an error in [16], according to the interpretation of lwsync chosen. According to [25] it is; however, the relaxation that is source of the error has not been observed in practice [25], making hard to convince compiler writers that it is necessary.

About ARM

We are tempted to make the argument that ARM is similar to Power leveraging [25]. Unfortunately [3] has found [25] to be inaccurate w.r.t. ARM. In [3] a different model is proposed, but it is claimed that some current ARM architectures suffer from a bug, which simply stated allows reads on the same reference to be reordered. Moreover, the new ARMv8

relaxed memory model is not yet quite well understood (see [13]). We note however that most of the behaviors discussed in [3] w.r.t. ARM are sound for the JMM, and could easily be incorporated into cookbook-high without affecting our proofs. Moreover, the conservative strategy of [16], which compiles all barriers to the sync-like `dmb` is guaranteed to satisfy our simulations as shown in Theorem 10.

7 Related Work

Our work is closely based on the intuitions provided by [16], whose structure and rules we try to follow as close as possible. We only depart from the informal description of [16] to remediate errors. Similar to [16], [4] presents an operational definition of a low-level language agnostic memory model, describing how the model, equipped with a notion of store atomicity and permissible instruction reorderings, can be used to capture various kinds of weak memory behavior.

Also related to our development is [6], where a compilation strategy for C++11 to Power is defined and verified correct. Unlike [6], however, we do not attempt to provide a concrete compilation strategy, instead verifying the minimal conditions required for compiling to architectures considered in [16]. In particular, this means we that need a *lingua franca* to relate the JMM and the architectures: we use the cookbook-low language for this purpose. Moreover, the roach motel semantics of the JMM is a fundamental property that we preserve, whereas this is not a concern in [6].

Recent efforts consider program analyses to insert fences [26, 2] to guarantee SC. We do not consider implementing SC for Java here which would be prohibitively inefficient for architectures like Power; recent work [19, 27] argues that the cost of ensuring end-to-end SC may be modest, assuming particular non-standard hardware support. Other works consider the elimination of redundant fences in weak memory systems (e.g., TSO [29, 21]). Since [16] enforces a conservative implementation of the JMM, we believe the cookbook-low formalization could be a starting point to consider similar results for Java (as informally argued at the end of [16]).

8 Conclusion

We present the first formal study of the minimal conditions necessary to guarantee the correct compilation of the JMM in different architectures as advocated by [16]. In doing so, we identify errors in the recommended implementation of volatiles for Power, and we propose *provably correct* repairs. We also define the semantics of the cookbook-low language, which we propose as an upper bound on how strong a memory model for Java can be, while not needing additional synchronization for the implementation of normal variables. Our work thus puts the "cookbook for compiler writers" [16] on a sound formal footing, a much needed exercise considering the current ongoing conversations about repairing the JMM.

Acknowledgements. We would like to thank Luc Maranget who provided us with access to a Power 7 machine to conduct some of our experiments. We are also grateful to Peter Sewell and Doug Lea who provided insightful comments on an early draft of our work. Finally we thank the anonymous reviewers whose recommendations have improved the quality of the paper.

References

1. Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, (ISCA 1990), Seattle, WA, June 1990*, pages 2–14, 1990.
2. Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't Sit on the Fence - A Static Analysis Approach to Automatic Fence Insertion. In *Computer Aided Verification, (CAV 2014), Vienna, Austria, July 18-22, 2014. Proceedings*, pages 508–524, 2014.
3. Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7, 2014.
4. Arvind and Jan-Willem Maessen. Memory Model = Instruction Reordering + Store Atomicity. In *33rd International Symposium on Computer Architecture (ISCA 2006), June 17-21, 2006, Boston, MA, USA*, pages 29–40, 2006.
5. David Aspinall and Jaroslav Ševčík. Formalising Java's Data Race Free Guarantee. In *Theorem Proving in Higher Order Logics, 20th International Conference, (TPHOLs 2007), Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, pages 22–37, 2007.
6. Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 2012), Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 509–520, 2012.
7. Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. OCTET: Capturing and Controlling Cross-thread Dependences Efficiently. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, (OOPSLA 2013), Indianapolis, IN, USA, October 26-31, 2013*, pages 693–712, 2013.
8. Gérard Boudol, Gustavo Petri, and Bernard P. Serpette. Relaxed Operational Semantics of Concurrent Programming Languages. In *Proceedings Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics, (EXPRESS/SOS 2012), Newcastle upon Tyne, UK, September 3, 2012.*, pages 19–33, 2012.
9. Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. Plan B: a buffered memory model for Java. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 2013), Rome, Italy - January 23 - 25, 2013*, pages 329–342, 2013.
10. Cormac Flanagan and Matthias Felleisen. The Semantics of Future and Its Use in Program Optimizations. In *The 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 1995), San Francisco, California, USA, January 23-25, 1995*, pages 209–220, 1995.
11. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation, (PLDI 1993), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993.
12. Marieke Huisman and Gustavo Petri. The Java Memory Model: a Formal Explanation. *Verification and Analysis of Multi-threaded Java-like Programs (VAMP'07)*, 2007.
13. JMM Mailing list: Developing the JEP 188: Java Memory Model Update. `http://mail.openjdk.java.net/mailman/listinfo/jmm-dev`, 2014.
14. Java Memory Model and Thread Specification, 2004.
15. Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

16 Doug Lea. The JSR-133 Cookbook for Compiler Writers. `http://g.oswego.edu/dl/jmm/cookbook.html`.

17 Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An Axiomatic Memory Model for POWER Multiprocessors. In *Computer Aided Verification - 24th International Conference, (CAV 2012), Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 495–512, 2012.

18 Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *Special POPL Issue (DRAFT)*, 2005.

19 Daniel Marino, Abhayendra Singh, Todd D. Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A Case for an SC-preserving Compiler. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI 2011), San Jose, CA, USA, June 4-8, 2011*, pages 199–210, 2011.

20 Jaroslav Ševčík and David Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, pages 27–51, 2008.

21 Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM*, 60(3):22, 2013.

22 Scott Owens, Susmit Sarkar, and Peter Sewell. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics, 22nd International Conference, (TPHOLs 2009), Munich, Germany, August 17-20, 2009. Proceedings*, pages 391–407, 2009.

23 Gustavo Petri, Jan Vitek, and Suresh Jagannathan. Cooking the Books: Formalizing JMM Implementation Recipes (Extended Version), 2015. `https://www.cs.purdue.edu/homes/gpetri/publis/CtB-long.pdf`.

24 PowerPC ISA. Version 2.06 Revision B. IBM, 2010.

25 Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER Multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI 2011), San Jose, CA, USA, June 4-8, 2011*, pages 175–186, 2011.

26 Dennis Shasha and Marc Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

27 Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd D. Millstein, and Madanlal Musuvathi. End-to-end Sequential Consistency. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, pages 524–535, 2012.

28 SPARC Corporation. *The SPARC Architecture Manual (V. 9)*. Prentice-Hall, Inc., 1994.

29 Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying Fence Elimination Optimisations. In *Static Analysis - 18th International Symposium, (SAS 2011), Venice, Italy, September 14-16, 2011. Proceedings*, pages 146–162, 2011.

A Simulation Proofs

This appendix provides the theory and proofs for theorem Theorem 6 whose intuitive meaning was presented in section 5.

To capture the meaning of Table 5 in the program semantics we will define when a cookbook-low program is *well-compiled* with respect to Table 5 and a program expressed in cookbook-high. To that end we need the following definitions.

▶ **Definition 11** (Fence flattening)**.** We will denote by $\mathsf{c}\Downarrow$ to the program c (in this case a cookbook-low program) where all the fence instructions have been removed. If c is a cookbook-high program $\mathsf{c}\Downarrow$ substitutes all volatile instructions for their normal counterpart instruction. We elide the recursive definition on the syntax, which is self-evident. Similarly, and abusing the notation, we will denote by $\delta\Downarrow$ to the temporary store δ where all fence memory operations have been removed. This definition is again self-evident and defined by recursion on the temporary store.

Before formalizing the notion of a well-compiled program, we need to define when a cookbook-low temporary store δ_L is *well-synchronized* with respect to Table 5, meaning that the memory operations therein are separated by barrier operations according to that table. We use the notation $\mathsf{mo}\uparrow$ to denote the casting of cookbook-low memory operations corresponding to volatiles at the cookbook-high level, to their corresponding cookbook-low memory operations. For example, we write $\mathsf{rd_p}\uparrow = \mathbf{vrd_p}$ if the cookbook-low operation $\mathsf{rd_p}$ on a volatile pointer $\mathsf{p} \in \mathcal{V}ptr$ when cast is a cookbook-high volatile operation $\mathbf{vrd_p}$. Applying this cast on an operation over a normal reference leaves it unaltered. We remark that since casts are just used to query Table 5, the values and other components of the memory operations are unimportant.

▶ **Definition 12** (Well-Synchronized Cookbook-Low Temp. Store)**.** We say that a cookbook-low generated temporary store δ_L is *well-synchronized* according to Table 5 if whenever δ_L can be decomposed as: $\delta_L = \delta_{L0} \cdot (t, \mathsf{mo}) \cdot \delta_{L1} \cdot (t, \mathsf{mo}') \cdot \delta_{L2}$ and $\text{H-L}\ (\mathsf{mo}\uparrow, \mathsf{mo}'\uparrow) = b$ then there exists a memory operation (t, b) in δ_{L1}.

▶ **Definition 13** (Well-Compiled Cookbook-Low Program)**.** We say that a cookbook-low thread system T_L is well-compiled from the cookbook-high thread system T_H if it satisfies:

1. For each configuration $(\sigma, \delta'_L, \mathsf{T}'_L)$ such that $(\sigma_0, \epsilon, \mathsf{T}_L) \overset{\text{CL}}{\Longrightarrow}^* (\sigma, \delta'_L, \mathsf{T}'_L)$, where σ_0 is some initial store and ϵ is the empty initial temporary store, we have that δ'_L is a well-synchronized temporary store according to Table 5, and
2. $\mathsf{dom}(\mathsf{T}_L) = \mathsf{dom}(\mathsf{T}_H)$ and for each thread $t \in \mathsf{dom}(\mathsf{T}_H)$ we have that $\mathsf{T}_H(t)\Downarrow = \mathsf{T}_L(t)\Downarrow$. In other words, removing fences from T_L and treating volatile accesses as normal in T_H, yields the same programs.

A.1 Simulation

We now define the simulation relation between configurations of cookbook-low and cookbook-high, which will be used to prove the adequacy of Theorem 13.

While Theorem 13 is simple and intuitive, it is insufficient to relate intermediate configurations at runtime. This is because what are volatile memory accesses at runtime for cookbook-high are normal memory accesses for cookbook-low. Unsurprisingly then, they are subject to different operational rules as defined by the semantics ($\overset{\text{CH}}{\Longrightarrow}$) and ($\overset{\text{CL}}{\Longrightarrow}$) resp. In particular, while volatile writes and reads are not subject to write-early or read-early behaviors in cookbook-high, the corresponding accesses will be subject to these rules in

cookbook-low. Practically this means that there will be placeholder values that are substituted in the temporary store and thread system earlier in cookbook-low than cookbook-high. This complicates our definition which will now have to take care of comparing the configurations up-to: **1.** the substitution of high-level memory operations for low-level memory operations, **2.** the substitution of placeholders that have been read early, and **3.** the fact that some substitutions could enable the generation of new operations at the low-level – through redex-creation after substitution – earlier than at the high-level. We now present definitions that enable the expression of our simulation relation.

Given a mapping f with domain A and co-domain B ($f : A \to B$) we use the generic notation $\mathsf{c}[f]$ and $\delta[f]$ to denote the substitution of all occurrences of elements in $\mathsf{dom}(f)$ with their respective image in f. This is the standard recursive definition on the structure of the command for c, and the length of the temporary store for δ. We will sometimes, by abuse of notation, apply the substitution of commands to an entire thread system $\mathsf{T}[f]$, with the obvious definition.

To capture the asynchrony between the time at which reads are performed in cookbook-low vs. cookbook-high, we parameterize the simulation with a placeholder map, mapping placeholders to either values, or another placeholder as might be the case with read-early actions in cookbook-low. We use the metavariable $\varphi : \mathcal{P}lHold \to \mathcal{P}holdVal$ to denote this mapping. redexes. Our first observation is that placeholder substitution preserves

▶ Remark. Given a redex r and a placeholder map φ, the substitution $r[\varphi^{-1}]$ is also a redex.

Proof. Simple case analysis on the definition of redexes in Figure 2.

▶ **Definition 14** (Placeholder Map Substitution)**.** We define the placeholder substitution of a cookbook-high temporary store δ_H with placeholder map φ, denoted $\delta_H[\varphi]$, thus: **1.** each occurrence of a placeholder $\rho \in \mathsf{dom}(\varphi)$ other than in an action $\mathbf{vrd}_{\_,\rho}$, $\mathtt{t}^{\rho}_{v}$ or e^{ρ}_{v} is immediately substituted by $\varphi(\rho)$, and **2.** any action of the form $\mathbf{vrd}_{\_,\rho}$ (there is at most one such action, because this is the action originating the placeholder ρ and generating a new one requires freshness), is replaced by the action $\overline{\mathsf{rd}_{\rho,\varphi(\rho)}}$, any action of the form $\mathtt{t}^{\rho}$ is substituted for $\mathtt{t}^{\rho}_{\varphi(\rho)}$ and similarly for e^{ρ} . We overload the notation to $\mathsf{c}[\varphi]$, denoting the obvious substitution of placeholders for their images in φ in the command c.

We need to be able to compare cookbook-high temporary stores with cookbook-low temporary stores. To that end, we use the following notion.

▶ **Definition 15** (Equality up to Volatile Compilation)**.** We say that two memory operations mo and mo' are low-to-high similar, denoted $\mathsf{mo} \simeq \mathsf{mo}'$ and defined as the symmetric relation satisfying (with free variable existentially quantified):

$$\mathsf{mo} \simeq \mathsf{mo}' \iff \begin{cases} \mathsf{mo} = \mathsf{mo}', \text{ or} \\ \mathsf{mo} = \mathbf{vwr}_{\rho,\mu} \quad \& \ \mathsf{mo}' = \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\rho,\mu}, \text{ or} \\ \mathsf{mo} = \mathbf{vwr}_{\mathsf{p},\mu} \quad \& \ \mathsf{mo}' = \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\mathsf{p},\mu}, \text{ or} \\ \mathsf{mo} = \mathbf{vrd}_{\rho,\mu} \quad \& \ \mathsf{mo}' = \mathsf{rd}_{\rho,\mu}, \text{ or} \\ \mathsf{mo} = \mathbf{vrd}_{\mathsf{p},\mu} \quad \& \ \mathsf{mo}' = \mathsf{rd}_{\mathsf{p},\mu}, \text{ or} \\ \mathsf{mo} = \mathbf{vrd}_{\rho,\rho'} \quad \& \ \mathsf{mo}' = \overline{\mathsf{rd}_{\rho',\mu}}, \text{ or} \\ \mathsf{mo} = \mathbf{vrd}_{\mathsf{p},\rho'} \quad \& \ \mathsf{mo}' = \overline{\mathsf{rd}_{\rho',\mu}}, \text{ or} \\ \mathsf{mo} = \overline{\mathbf{vrd}_{\rho,\mu}} \quad \& \ \mathsf{mo}' = \overline{\mathsf{rd}_{\rho,\mu}}. \end{cases}$$

This definition can be trivially extended to traces of memory operations, denoted $\delta_H \simeq \delta_L$. We adopt the same notation to compare commands, where we for example write $(x :=_v v) \simeq (v := v)$ to denote that the volatile assignment in cookbook-high is similar *up to volatile compilation* to the normal assignment in cookbook-low.

Before presenting the full simulation definition, we consider a restriction to the possible placeholder maps that we consider in our proofs. Indeed, by observing the conflict relation in Table 3 we can see that a volatile read action (any of $\mathbf{vrd}_\varrho$ or $\overline{\mathbf{vrd}_\varrho}$) conflicts with any possible subsequent memory operation of the same thread. Consequently, in Table 5 volatile read operations need to be followed by a barrier of the form $\langle \mathsf{ld}|\_\rangle$ before any subsequent memory operation by the same thread. With this in mind, we know that any operation in a temporary store of cookbook-low which is the target of a volatile load operation in cookbook-high, is blocked by the $\overset{\curvearrowleft}{\text{CL}}$ commutativity definition in the temporary store until the earlier volatile load (and its barrier) are committed. Notably, the only operations that are delayed in cookbook-high w.r.t. cookbook-low (which justifies the need for the placeholder map) are volatile reads. But then, by the rule (ER) of Figure 5 for the commutativity relation $\overset{\curvearrowleft}{\text{CL}}$, we know that any read-early action can only appen if there are no $\langle \mathtt{l}|\mathtt{l}\rangle$ operations by the same thread before it in the temporary store. Hence, for each thread there is at most one early-read operation in the temporary store of cookbook-low w.r.t. the same thread in cookbook-high. Moreover, this operation has to be the first such volatile read by the thread in cookbook-high (otherwise a $\langle \mathtt{l}|\mathtt{l}\rangle$ would have prevented the second one). We will encode this intuition in the *valid placeholder map* w.r.t. a cookbook-low temporary store definition.

▶ **Definition 16** (Valid placeholder map)**.** A placeholder map φ is valid w.r.t. the cookbook-low configuration $(\sigma, \delta_L, \mathsf{T}_L)$ if for every placeholder $\rho \in \mathsf{dom}(\varphi)$ we have:

- the first memory operation in δ_L in which ρ occurs is an early-read action. That is, $\delta_L = \delta_{L0} \cdot (t, \overline{\mathsf{rd}_{\rho,\mu}}) \cdot \delta_{L1}$, where ρ does not occur in δ_{L0} in any action other than the $\mathcal{I}$ component of the write seen by the read (cf. rule (ER)),
- any action of thread t in δ_{L1} is separated of this early-read by a barrier that prevents their commutativity according to $\overset{\curvearrowleft}{\text{CL}}$, and
- the thread t does not satisfy these conditions for any other placeholder ρ' such that $\rho' \in \mathsf{dom}(\varphi)$.

In essence, this definition is saying that for each thread, there is at most one early-read in cookbook-low, which corresponds to a volatile read in cookbook-high, and this early-read action must block any memory access by thread t after it from being committed.

We can finally define the simulation between a cookbook-high configuration and a cookbook-low one. Our simulation is parameterized by **1.** the source cookbook-high program, and its cookbook-low candidate compilation, and **2.** the current placeholder map relating placeholders substituted early in cookbook-low w.r.t. cookbook-high.

▶ **Definition 17** (Low to High Simulation Relation)**.** We say that a cookbook-high configuration $(\sigma, \delta_H, \mathsf{T}_H)$ simulates a cookbook-low configuration $(\sigma, \delta_L, \mathsf{T}_L)$ with respect to the valid placeholder map φ, and initial programs T_{H0} and T_{L0}, which we shall denote by $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}\ (\sigma, \delta_L, \mathsf{T}_L)$, iff:

1. $(\sigma_0, \epsilon, \mathsf{T}_{H0}) \overset{\text{CH}}{\Longrightarrow}^* (\sigma, \delta_H, \mathsf{T}_H)$, and
2. $(\sigma_0, \epsilon, \mathsf{T}_{L0}) \overset{\text{CL}}{\Longrightarrow}^* (\sigma, \delta_L, \mathsf{T}_L)$, and
3. T_{L0} is a well-compiled thread system from T_{H0} with respect to Table 5, and
4. $\delta_L \Downarrow \simeq (\delta_H[\varphi])\Downarrow$, and

5. for all $t \in \mathsf{dom}(\mathsf{T}_L)$ we have $\mathsf{T}_L(t)\Downarrow \simeq\ (\mathsf{T}_H(t)[\varphi])\Downarrow$.

▶ Remark. For any two operations mo and mo′ whose target reference are not a placeholders, and such that they are not data-dependent on each other (i.e. they do not operate on the same pointer), if the operations are conflicting according to Table 3 ($\mathsf{mo} \#_{\text{CH}} \mathsf{mo}'$), then Table 5 requires that a barrier separating them to be inserted when compiling to cookbook-low, i.e. $\text{H-L}\ (\mathsf{mo}, \mathsf{mo}') = b$ with $\mathsf{mo}\ \text{CL}\ b$ and $b\ \text{CL}\ \mathsf{mo}$.

Proof. Simple observation of Table 3 and Table 5.

We now establish our simulation result between cookbook-low and cookbook-high.

▶ **Theorem 18** (Cookbook-Low simulates Cookbook-High)**.** *Given a cookbook-high and a cookbook-low states related by the simulation relation* $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}\ (\sigma, \delta_L, \mathsf{T}_L)$ *such that* $(\sigma, \delta_L, \mathsf{T}_L)$ *can take a step:* $(\sigma, \delta_L, \mathsf{T}_L) \overset{\text{CL}}{\Longrightarrow} (\sigma', \delta'_L, \mathsf{T}'_L)$ *then there exists* φ' *such that either*

$$(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi'}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}\ (\sigma', \delta'_L, \mathsf{T}'_L)$$

or there exists δ'_H *and* T'_H *with* $(\sigma, \delta_H, \mathsf{T}_H) \overset{\text{CH}^*}{\Longrightarrow} (\sigma', \delta'_H, \mathsf{T}'_H)$ *and*

$$(\sigma', \delta'_H, \mathsf{T}'_H)\ \mathcal{R}^{\varphi'}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}\ (\sigma', \delta'_L, \mathsf{T}'_L)$$

Proof Sketch. The proof proceeds by cases on the cookbook-low step taken. The case in which the step is an intra-thread step from Figure 3, we have by item A.1 that there must be a similar redex up to placeholder substitution. The most difficult and important cases are (ER) and (NW) when they are the compilation target of a volatile variable. In the case of (ER) the volatile read cannot be performed at top level, since there are no volatile early-reads at top level. At this step the placeholder map is extended to capture the delay between cookbook-high and cookbook-low. For the case of (NW), if the reference being written is not the target of a volatile reference in cookbook-high then both semantics can take similar steps. Otherwise, if it is the target of a volatile reference in cookbook-high all the early-reads that have seen this write early (i.e. all the reads that generate a placeholder in the $\mathcal{I}$ component of this write) have to be immediately performed. It is not hard to see that all the conditions in cookbook-high are met to immediately perform these reads (by a combination of the constraints of (ER)) that produced the early reads and (NW) which is the rule being applied. Once all these steps are performed in cookbook-high their placeholders are substituted in cookbook-high as dictated by the placeholder map φ, which evidently preserves the simulation with the updated placeholder map $\varphi/\mathcal{I}$. The full details of this proof are included as additional material. ◀

The remark below states that the only requirement for two *initial* configurations to be *similar* (according to Theorem 17) is that the thread systems be *well-compiled* related. This definition captures the notion that compilation implies similarity at static time.

▶ Remark. For any default store σ_0 and a cookbook-low thread system T_L such that it is a well-compiled program from the cookbook-high thread system T_H we have that

$$(\sigma_0, \epsilon, \mathsf{T}_H)\ \mathcal{R}^{\emptyset}_{(\mathsf{T}_H, \mathsf{T}_L)}\ (\sigma_0, \epsilon, \mathsf{T}_L)$$

Proof. Obvious by Theorem 17.

Another observation that we make is that the contribution of operations of each thread to the temporary store is independent of the temporary store, and the store (other than by the freshness requirements).

▶ Remark. The rules of Figure 3 are defined independent of the initial temporary store (δ), and the resulting temporary store (δ') is uniquely determined by the memory operation (mo) being performed.

Proof. Obvious by cases on the steps of rule (Single-Thread Step) defined in Figure 3.

We can now establish the simulation of intra-thread steps. That is, if a cookbook-low configuration is similarity related to a configuration of cookbook-high, then if cookbook-low takes a step of the intra-thread semantics, a step in cookbook-high can be taken to preserve the simulation. This lemma will be used to prove the overall simulation argument between cookbook-low and cookbook-high.

▶ **Lemma 19** (Intra-Thread Weak Simulation). *Assuming states of cookbook-high and cookbook-low related by simulation* $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}\ (\sigma, \delta_L, \mathsf{T}_L)$ *such that* $(\sigma, \delta_L, \mathsf{T}_L) \xrightarrow[t]{\mathsf{mo}} (\sigma, \delta'_L, \mathsf{T}'_L)$ *then either:*

1. mo *is a barrier operation (b), in which case by Theorem 17 we know that* $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}$ $(\sigma, \delta'_L, \mathsf{T}'_L)$*, or*
2. *there exists a cookbook-high configuration* $(\sigma, \delta'_H, \mathsf{T}'_H)$ *such that it can take the step* $(\sigma, \delta_H, \mathsf{T}_H) \xrightarrow[t]{\mathsf{mo}} (\sigma, \delta'_H, \mathsf{T}'_H)$ *and furthermore simulation is preserved:* $(\sigma, \delta'_H, \mathsf{T}'_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}$ $(\sigma, \delta'_L, \mathsf{T}'_L)$.

Proof. If the memory operation mo is a barrier operation the case is obvious, since by Theorem 17 we have that T_{L0} is a well-compiled program w.r.t. T_{H0} and therefore the temporary store δ'_L which includes a barrier more than δ_L is well-synchronized, and once more by Theorem 17 it is the case that $\delta'_L\Downarrow \simeq (\delta_H[\varphi])\Downarrow$. We notice that by the definition of fence-flattening ($\Downarrow$) removing a fence operation does not change the relation $\mathsf{T}_L(t)\Downarrow = \ (\mathsf{T}_H(t)[\varphi])\Downarrow$ for every thread $t \in \mathsf{dom}(\mathsf{T}_H)$ which applied before, and so it similarly applies to T'_L.

If the memory operation is not a barrier operation we have to consider the cases (which are all similar). Firstly we note that by condition 5 of Theorem 17, and item A.1 we have that for each thread t either they coincide in their redexes up to the placeholder map (φ) substitution and equality up to volatile compilation (Theorem 15), or the cookbook-lowthread contains a barrier operation not present in the cookbook-high thread (since they are erased by flattening). The latter case was considered above, therefore we have that $\mathsf{T}_L(t) = \mathbf{E}[r]$ and $\mathsf{T}_H(t) = \mathbf{E}'[r']$ where $r'[\varphi] \simeq r$.

If we have that $r'[\varphi] = r$ we are done, since the memory operation generated by both semantics will be related after placeholder map substitution, which is easy to check. Similarly if the actions are not identical, they will generate memory operations that are similar up to volatile compilation as defined in Theorem 15. Let us consider the case where we have at the cookbook-low a *normal assignment* of a *volatile pointer*, which evidently has to be a volatile assignment at the cookbook-high level. Then we have $r = (\mathsf{p} := v)$ and $r' = (\mathsf{p} :=_v v)$. By considering the rules of Figure 3 we see that cookbook-low produces a $\mathsf{wr}_{\mathsf{p},v}$ operation and cookbook-high produces a $\mathbf{vwr}_{\mathsf{p},v}$ operation. These operations are related by Theorem 15. All other cases are similar. We notice that the choice of free placeholder values and reference names for *new* operations can always be matched by cookbook-high because we keep the stores σ in synchrony and the temporary stores eliminate their placeholder values at the same time (as it will be shown in Theorem 23).

Since none of these rules modify δ_L and δ_H other than by adding an operation at the end (given by Theorem A.1) we conclude that the resulting configurations again similar. This concludes the proof. ◀

The following lemma establishes a strict relation between the actions in a cookbook-low temporary store, and a cookbook-high temporary store if they for part of configurations related by similarity.

▶ **Lemma 20** (Temporary Store Decompose). *Assuming that the following relation holds* $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}\ (\sigma, \delta_L, \mathsf{T}_L)$ *and also* $\delta_L = \delta_{L0} \cdot (t, \mathsf{mo}) \cdot \delta_{L1}$. *Then if* mo *is not a cookbook-low barrier instruction we have that there exist* δ_{H0}, δ_{H1} *and* mo' *such that:*

1. $\mathsf{mo} \simeq \mathsf{mo}'[\varphi]$,
2. $\delta_{L0}\Downarrow \simeq (\delta_{H0}[\varphi])\Downarrow$, *and*
3. $\delta_{L1}\Downarrow \simeq (\delta_{H1}[\varphi])\Downarrow$.

Moreover, if mo *is a barrier operation, conditions 2. and 3. above are still satisfied.*

Proof. By $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}\ (\sigma, \delta_L, \mathsf{T}_L)$ and Theorem 17 we have that $\delta_L\Downarrow \simeq\ (\delta_H[\varphi])\Downarrow$. We proceed by induction on δ_{L0} which is obvious. ◀

The following lemma and corollary state that when a cookbook-low and cookbook-high configurations are related by simulation, and a memory operation mo in cookbook-high is related to an operation mo' in cookbook-low – which has more values (i.e. a placeholder of mo has been substituted in mo') – then there must be fence operations that prevent the execution of mo' in cookbook-low. This is what justifies our definition of valid placeholder map Theorem 16.

▶ **Lemma 21.** *Suppose that we have the following states related by simulation:* $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}$ $(\sigma, \delta_L, \mathsf{T}_L)$. *Moreover suppose that by of Theorem 20 we have:*

1. $\delta_L = \delta_{L0} \cdot (t, \mathsf{wr}_{\overline{p},\_}) \cdot \delta_{L1}$,
2. $\delta_H = \delta_{H0} \cdot (t, \mathsf{wr}_{\overline{\rho},\_}) \cdot \delta_{H1}$ *with*
3. $\delta_{L0}\Downarrow \simeq (\delta_{H0}[\varphi])\Downarrow$ *and* $\delta_{L1}\Downarrow \simeq (\delta_{H1}[\varphi])\Downarrow$, *and*
4. $(t, \mathsf{wr}_{\overline{p},\_}) \simeq (t, \mathsf{wr}_{\overline{\rho},\_})[\varphi]$, *meaning that* $\varphi(\rho) = p$.

it must then be the case that

$$\delta_{L0} = \delta'_{L0} \cdot (t', \mathsf{wr}^{\mathcal{W}\cup\{t\},\mathcal{I}\cup\{\rho\}}_{\mathsf{p},p}) \cdot \delta''_{L0} \cdot (t, \overline{\mathsf{rd}_{\rho,p}}) \cdot \delta'''_{L0}$$

and moreover, δ'''_{L0} *can be decomposed as:*

$$\delta'''_{L0} = \delta^0_{L0} \cdot (t, \langle \mathtt{l}|\mathtt{s}\rangle) \cdot \delta^1_{L0}$$

meaning that if we have a write $(t, \mathsf{wr}_{\overline{p},\_})$ *in* δ_L *whose corresponding write in* δ_H *contains a placeholder value, then there must be a barrier* $\langle \mathtt{l}|\mathtt{s}\rangle$ *operation by thread* t *(with a preceding early-read by* t*) before the write in* δ_L.

Proof. The proof simply considers the Theorem 17, Theorem 16 and Theorem 13. From Theorem 17 we have that φ is a valid placeholder map. From Theorem 16 we have that if $\rho \in \mathsf{dom}(\varphi)$, which must be true by item 4 above, there must be a read-early action in δ_L of a volatile pointer p, which is a regular volatile read in δ_H and that read action must come in δ_L before the write action in question (i.e. $(t, \mathsf{wr}_{\overline{p},\_})$). Finally, by the definition of Theorem 13 and the rules of Table 5 we know that in between this early-read and the following write of the same thread there must be a $\langle \mathtt{l}|\mathtt{s}\rangle$ operation by the same thread (otherwise we would have a contradiction to Theorem 13). This concludes our proof. ◀

▶ **Corollary 22.** *Suppose that we have the following states related by simulation:* $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}\ (\sigma, \delta_L, \mathsf{T}_L)$*. Moreover suppose that by of Theorem 20 we have that:*

1. $\delta_L = \delta_{L0} \cdot (t, \overline{\mathsf{rd}_{p,\_}}) \cdot \delta_{L1}$,
2. $\delta_H = \delta_{H0} \cdot (t, \mathsf{rd}_{\rho,\_}) \cdot \delta_{H1}$ *(or* $\mathbf{vrd}_{\rho,\_}$ *with* p *in the previous item instead),*
3. $\delta_{L0}\Downarrow \simeq (\delta_{H0}[\varphi])\Downarrow$ *and* $\delta_{L1}\Downarrow \simeq (\delta_{H1}[\varphi])\Downarrow$*, and*
4. $(t, \mathsf{rd}_{p,\_}) \simeq (t, \mathsf{rd}_{\rho,\_})[\varphi]$*, meaning that* $\varphi(\rho) = p$ *(similarly for volatiles).*

it must then be the case that

$$\delta_{L0} = \delta'_{L0} \cdot (t', \mathsf{wr}^{\mathcal{W}\cup\{t\},\mathcal{I}\cup\{\rho\}}_{\mathsf{p},p}) \cdot \delta''_{L0} \cdot (t, \overline{\mathsf{rd}_{\rho,p}}) \cdot \delta'''_{L0}$$

and moreover, δ'''_{L0} *can be decomposed as:*

$$\delta'''_{L0} = \delta^0_{L0} \cdot (t, \langle \mathtt{l}|\mathtt{l}\rangle) \cdot \delta^1_{L0}$$

meaning that if we have a read $(t, \mathsf{rd}_{p,\_})$ *(resp. read of a volatile* $(t, \mathsf{rd}_{\mathsf{p},\_})$*) in* δ_L *whose corresponding read (resp. volatile) in* δ_H *contains a placeholder value, then there must be a barrier* $\langle \mathtt{l}|\mathtt{l}\rangle$ *operation by thread* t *(with a preceding early-read by* t*) before the read in* δ_L*.*

Proof. The proof is identical to that of Theorem 21

The following lemma is the counterpart of Theorem 19 for memory operations. This lemma has to take into account that the read-early operations on volatiles that are delayed in cookbook-high w.r.t. cookbook-low have to be performed when the write that justifies the early-reads (i.e. it is an early-write) is committed in memory. This is considered in the case (NR) for cookbook-low.

▶ **Lemma 23** (Memory Weak-Simulation)**.** *Given cookbook-high and cookbook-low states related by the simulation relation* $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}\ (\sigma, \delta_L, \mathsf{T}_L)$ *such that* $(\sigma, \delta_L, \mathsf{T}_L)$ *can take a memory step:* $(\sigma, \delta_L, \mathsf{T}_L) \underset{\text{CL}}{\hookrightarrow} (\sigma', \delta'_L, \mathsf{T}'_L)$ *then there exists* φ' *such that either*

$$(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi'}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}\ (\sigma', \delta'_L, \mathsf{T}'_L)$$

or there exists δ'_H *and* T'_H *such that* $(\sigma, \delta_H, \mathsf{T}_H) \underset{\text{CH}}{\hookrightarrow}^* (\sigma', \delta'_H, \mathsf{T}'_H)$ *and*

$$(\sigma, \delta'_H, \mathsf{T}'_H)\ \mathcal{R}^{\varphi'}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}\ (\sigma, \delta'_L, \mathsf{T}'_L)$$

Proof. We proceed by case analysis on the step taken $(\sigma, \delta_L, \mathsf{T}_L) \underset{\text{CL}}{\hookrightarrow} (\sigma', \delta'_L, \mathsf{T}'_L)$. Evidently the step taken cannot be any of (SC), (VW) or (VR), since these rules do not apply to the semantics of cookbook-low (CL). Let us consider the possible steps:

(NW) By $(\sigma, \delta_L, \mathsf{T}_L) \underset{\text{CL}}{\hookrightarrow} (\sigma', \delta'_L, \mathsf{T}'_L)$ we know that we can decompose δ_L as $\delta_L = \delta_{L0} \cdot (t, \mathsf{mo}) \cdot \delta_{L1}$ where $\mathsf{mo} \in \{\mathsf{wr}^{\mathcal{W},\mathcal{I}}_{p,v}, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\mathsf{p},v}\}$ (notice the difference between a normal pointer p and a volatile pointer p). At CL this difference is inconsequential, but the step taken in CH changes and it is subject to a different set of constraints as per $\overset{\curvearrowleft}{\text{CH}}$. Let us consider the two cases:

- We consider volatile references first ($\mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\mathsf{p},v}$). Given the memory step $(\sigma, \delta_L, \mathsf{T}_L) \underset{\text{CL}}{\hookrightarrow} (\sigma', \delta'_L, \mathsf{T}'_L)$ we have that $\sigma' = \sigma[\rho \leftarrow v]$, $\delta'_L = \delta_{L0} \cdot \delta_{L1}$, $\mathsf{T}'_L = \mathsf{T}_L$, and $\delta_{L0} \overset{\curvearrowleft}{\text{CL}} (t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\mathsf{p},v})$. By Theorem 20 and Theorem 2 we must have that $\delta_H = \delta_{H0} \cdot (t, \mathbf{vwr}_{\mathsf{p},v}) \cdot \delta_{H1}$. We do not need to consider the case with $(t, \mathbf{vwr}_{\rho,\mu})$ instead of $(t, \mathbf{vwr}_{\mathsf{p},v})$ because it contradicts Theorem 21. We argue now that a (VW) step can be take in the

cookbook-high semantics for $(\sigma, \delta_H, \mathsf{T}_H)$. We need to check that $\delta_{H0} \overset{\curvearrowleft}{\text{CH}} (t, \mathbf{vwr}_{\mathsf{p},v})$. Assume by contradiction that $\neg(\delta_{H0} \overset{\curvearrowleft}{\text{CH}} (t, \mathbf{vwr}_{\mathsf{p},v}))$. It must then be the case that $\delta_{H0} = \delta'_{H0} \cdot (t', \mathsf{mo}') \cdot \delta''_{H0}$ where the action (t', mo') conflicts with $(t, \mathbf{vwr}_{\mathsf{p},v})$. Let us consider the following cases:

- If $t \neq t'$, by $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}\ (\sigma, \delta_L, \mathsf{T}_L)$ and Theorem 20 we have that at the cookbook-low level we must have $\delta_{L0} = \delta'_{L0} \cdot (t', \mathsf{mo}'') \cdot \delta''_{L0}$. Since we had assumed that $\delta_{L0} \overset{\curvearrowleft}{\text{CL}} (t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\mathsf{p},v})$, and $\mathsf{mo}' \simeq \mathsf{mo}''[\varphi]$ (again by Theorem 20) we have a contradiction, meaning that $\delta_{H0} \overset{\curvearrowleft}{\text{CH}} (t, \mathbf{vwr}_{\mathsf{p},v})$ in which case a (VW) is possible. We notice that the final stores are identical. While we have performed the write both at cookbook-low and cookbook-high level, we will perform at cookbook-high all the read actions that happened to see this write early (i.e. all reads generating placeholder values in $\mathcal{I}$). We postpone this argument momentarily.
- If $t = t'$ we must have that $\mathsf{mo}' \#_{\text{CH}} \mathbf{vwr}_{\mathsf{p},v}$ by the definition of $\overset{\curvearrowleft}{\text{CH}}$ and Table 3. Again, we consider the shape of δ_{H0} for the corresponding operation to mo' in cookbook-low. By $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}\ (\sigma, \delta_L, \mathsf{T}_L)$ and Theorem 20 we know that it must be the case that $\delta_{L0} = \delta'_{L0} \cdot (t, \mathsf{mo}'') \cdot \delta''_{L0}$, where $\mathsf{mo}'' \simeq \mathsf{mo}'[\varphi]$. But then, by the definition of equality up to volatile compilation Theorem 15, and Table 3 we have that it must be the case that $\mathsf{mo}'' \#_{\text{CL}} \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\mathsf{p},v}$, which contradicts $\delta_{L0} \overset{\curvearrowleft}{\text{CL}} (t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\mathsf{p},v})$, a necessary condition for the initial (NW) step to be taken: a contradiction. Similarly to the case $t \neq t'$ we conclude that the (VW) step can be take in cookbook-high rendering similar states.

Delayed Volatile Reads. We will now consider the executions in cookbook-high of volatile reads that had already been read-early from the write $(t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\mathsf{p},v})$ being performed. We notice that this is the first time where we can be sure that the value written by $(t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\mathsf{p},v})$ (i.e. v) is in the current memory ($\sigma[\mathsf{p} \leftarrow v]$). We will then in cookbook-high execute all the reads getting a value from memory (hence implying SC-only behaviors) which have been read-early in cookbook-low for this write. These reads are all operations which in δ_L which are of the form $(t', \overline{\mathsf{rd}_{\rho,v}})$ where $\rho \in \mathcal{I}$ by the semantics of early-reads (ER). Let us say for instance that we could decompose δ_{L1} as $\delta_{L1} = \delta'_{L1} \cdot (t', \overline{\mathsf{rd}_{\rho,v}})\delta''_{L1}$. Then, once more by the simulation of the initial states and Theorem 20 we have that $\delta_{H1} = \delta'_{L1} \cdot (t', \mathbf{vrd}_{\mathsf{p},\rho})\delta''_{H1}$ with all the additional condition of that lemma. By $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}\ (\sigma, \delta_L, \mathsf{T}_L)$ and Theorem 16 we know that since p is a volatile pointer it must be the case that for each placeholder value $\rho \in \mathcal{I}$ it must be the case that $\varphi(\rho) = v$. We clearly have then that $\overline{\mathsf{rd}_{\rho,v}} \simeq \mathbf{vrd}_{\mathsf{p},\rho}[\varphi]$ (recall the particular definition of placeholder map substitution in Theorem 14).

We need now prove that a (VR) operation is permissible for the corresponding read $(t', \mathbf{vrd}_{\mathsf{p},\rho})$ in δ_{H1} rendering a new operation $(t', \overline{\mathbf{vrd}_{\rho,v}})$ in its place. If we can do this, it is evident that the resulting states are again similar by equality up to volatile implementation Theorem 15. The only obstacle we could have is if $\neg(\delta_{H0} \cdot \delta'_{H1} \overset{\curvearrowleft}{\text{CH}} (t', \mathsf{rd}_{\mathsf{p},\rho}))$. Again, we proceed by contradiction to show that this is not possible. Since volatile writes at the cookbook-high level are not subject to write-atomicity relaxations (there is no $\mathcal{W}$ component for such writes), the only possibility for the read operation to not be able to commute is if there is a conflicting operation by the same thread in $\delta_{H0} \cdot \delta'_{H1}$. That is, $\delta_{H0} \cdot \delta'_{H1} = \delta'_{H3} \cdot (t', \mathsf{mo}') \cdot \delta''_{H3}$ with $\mathsf{mo}' \#_{\text{CH}} \mathsf{rd}_{\mathsf{p},\rho}$ according to Table 3. If mo' is not a volatile memory operation, by considering Table 3 we have a contradiction. Therefore mo' must be on a volatile. Otherwise, if this action is on a volatile variable we have once more by Theorem 20 that there must exists

a corresponding normal memory access in δ_{L0} which is separated from the $(t', \overline{\mathsf{rd}_{\rho,v}})$ action in δ_L by an appropriate barrier ($\langle \mathtt{l}|\mathtt{l}\rangle$, or $\langle \mathtt{s}|\mathtt{l}\rangle$) by Theorem 17 and the fact that the temporary store δ_L is well-synchronized (Theorem 12). But this would mean that the cookbook-low (ER) step that generated the operation $(t', \overline{\mathsf{rd}_{\rho,v}})$ in δ_L was not possibile, since the condition $\delta_{L0} \cdot \delta'_{L1} \overset{\curvearrowright}{\text{CL}}_{Sync} (t', \mathsf{rd}_{\varrho,\rho})$ was not met by virtue of the preceding barrier. We reach then a contradiction meaning that $(\delta_{H0} \cdot \delta'_{H1} \overset{\curvearrowright}{\text{CH}} (t', \mathsf{rd}_{\mathsf{p},\rho}))$ and the (VR) operation can be performed.

We conclude this part of the proof noticing that since the value v is in the store $\sigma[\mathsf{p} \leftarrow v]$ this (VR) operation substitutes the placeholder value ρ for v and we can therefore erase it from the placeholder map, obtaining a simulation where φ is replaced by $\varphi/\{\rho\}$. We perform this for all read operations whose placeholder appears in $\mathcal{I}$ for the original write $(t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\mathsf{p},v})$ and conclude the case.

This step makes clear that on volatile references one has the illusion of SC. This is the only step that reduces the placeholder map φ.

- If the reference in the write operation is not volatile (i.e. $(t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{p,v})$) we reason as before arguing that a normal write step (NW) can be taken in cookbook-high. By Theorem 20 we have that $\delta_{H0} \cdot (t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{p,v}) \cdot \delta_{H1}$ and if we can make a step $(\sigma, \delta_{H0} \cdot (t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{p,v}) \cdot \delta_{H1}, \mathsf{T}_H) \underset{\text{CH}}{\hookrightarrow} (\sigma[p \leftarrow v], \delta_{H0} \cdot \delta_{H1}, \mathsf{T}_H)$ we have that $(\sigma[p \leftarrow v], \delta_{H0} \cdot \delta_{H1}, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}\ (\sigma[p \leftarrow v], \delta_{L0} \cdot \delta_{L1}, \mathsf{T}_L)$. We only need to prove that $\delta_{H0} \overset{\curvearrowright}{\text{CH}} (t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{p,v})$. We assume by contradiction that $\neg(\delta_{H0} \overset{\curvearrowright}{\text{CH}} (t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{p,v}))$. It must then be the case that there is a conflicting action mo in δ_{H0} with $(t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{p,v})$. Let us say that $\delta_{H0} = \delta'_{H0} \cdot (t', \mathsf{mo}) \cdot \delta''_{H0}$ where if $t = t'$ then $\mathsf{mo} \#_{\text{CH}} \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{p,v}$, and if $t' \neq t$ we reason as in the previous case of the proof. If $t = t'$ we have as we have reasoned before a contradiction to $\delta_{H0} \overset{\curvearrowright}{\text{CL}} (t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{p,v})$, which is necessary for the original (NW) step in cookbook-low. And again, if the conflicting is w.r.t. a previous volatile variable of the same thread, by Theorem 20 and well-synchronized temporary store δ_L (given by Theorem 17) we would have either a $(t, \langle \mathtt{l}|\mathtt{s}\rangle)$ or a $(t, \langle \mathtt{s}|\mathtt{s}\rangle)$ barrier in δ_{L0}, which invalidates the applicability of the (NW) rule, implying once more a contradiction. Therefore, we have that $\delta_{H0} \overset{\curvearrowright}{\text{CH}} (t, \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{p,v})$ which means that the (NW) step is possible at the cookbook-high level and renders related states as output. We notice that the placeholder map needs not be changed in this case.

(NR) We again, consider the cases whether the reference being accessed is normal or volatile:

- If it is a volatile reference we have by that $\delta_L = \delta_{L0} \cdot (t, \mathsf{rd}_{\mathsf{p},\rho}) \cdot \delta_{L1}$, $\sigma = \sigma'$, $\delta_{L0} \overset{\curvearrowright}{\text{CL}} (t, \mathsf{rd}_{\mathsf{p},\rho})$, $\delta'_L = \delta_{L0} \cdot (\delta_{L1}[\rho \leftarrow \sigma(\mathsf{p})])$ and $\mathsf{T}'_L = \mathsf{T}_{L\rho \leftarrow \sigma(\mathsf{p})}$. By the hypothesis $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}\ (\sigma, \delta_L, \mathsf{T}_L)$ and Theorem 20 we have that $\delta_H = \delta_{H0} \cdot (t, \mathbf{vrd}_{\mathsf{p},\rho}) \cdot \delta_{H1}$. We do not need to consider the possibility of having an operation of the form $(t, \mathbf{vrd}_{\rho',\rho})$ in δ_H (where $\varphi(\rho') = \mathsf{p}$ by Theorem 15) by virtue of Theorem 22 which implies that we would have a contradiction in that case. Evidently, if we can perform a (VR) step reducing $(t, \mathbf{vrd}_{\mathsf{p},\rho})$ we arrive at states related by $\mathcal{R}^{\rho}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}$, since the substitutions are identical in both states. We only need to prove that $\delta_{H0} \overset{\curvearrowright}{\text{CH}} \mathbf{vrd}_{p,\rho}$.

  Assume by contradiction that $\neg(\delta_{H0} \overset{\curvearrowright}{\text{CH}} \mathbf{vrd}_{p,\rho})$. As we reasoned before there must be a memory operation (t, mo) in δ_{H0} such that $\mathsf{mo} \#_{\text{CH}} \mathbf{vrd}_{p,\rho}$ (since operations on other threads cannot prevent a read from being performed in this thread). On inspection of Table 3 we find that mo must be another volatile memory access (or a lock action). But then, by Theorem 20 and Theorem 17 there must be a barrier b in δ_{L0} such

that $b \#_{\text{CL}} \mathsf{rd}_{p,\rho}$ contradicting $\delta_{L0} \overset{\curvearrowright}{\text{CL}} (t, \mathsf{rd}_{\mathsf{p},\rho})$ which is required for the cookbook-low memory step to happen. We conclude that $(\delta_{H0} \overset{\curvearrowright}{\text{CH}} \mathbf{vrd}_{p,\rho})$ which ends this case.

- If the reference is normal we again have that $\delta_L = \delta_{L0} \cdot (t, \mathsf{rd}_{p,\rho}) \cdot \delta_{L1}$, $\sigma = \sigma'$, $\delta_{L0} \overset{\curvearrowright}{\text{CL}} (t, \mathsf{rd}_{p,\rho})$, $\delta'_L = \delta_{L0} \cdot (\delta_{L1}[\rho \leftarrow \sigma(p)])$ and $\mathsf{T}'_L = \mathsf{T}_{L\rho \leftarrow \sigma(p)}$. Moreover since we have $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}\ (\sigma, \delta_L, \mathsf{T}_L)$, and using Theorem 20 and Theorem 22 we have that $\delta_H = \delta_{H0} \cdot (t, \mathsf{rd}_{p,\rho}) \cdot \delta_{H1}$. We need to prove that we can apply the (NR) rule to reduce $(t, \mathsf{rd}_{p,\rho})$ in cookbook-high to conclude the case. Once more, we need to prove that $(\delta_{H0} \overset{\curvearrowright}{\text{CH}} \mathsf{rd}_{p,\rho})$ by contradiction. It must then be the case that there is a memory operation mo that conflicts with $\mathsf{rd}_{p,\rho}$ in δ_{H0}. The only possibility considering that $\#_{\text{CH}}$ and $\#_{\text{CL}}$ are similar for all memory actions at this level is that there might be in mo be an operation on a place-holder ρ that is in the domain of φ (notice that place-holder operations impose more constraints that normal operations). But then, by the definition of valid place-holder map for φ and Theorem 22 there must be a prior volatile read action originating this place-holder, on which a read-early action has been performed in δ_{L0}. We conclude as we did for the $\mathsf{wr}^{\mathcal{W},\mathcal{I}}_{p,\rho}$ case that there must be a $\langle \mathtt{l} | \mathtt{l} \rangle$ in δ_{L0} contradicting $\delta_{L0} \overset{\curvearrowright}{\text{CL}} (t, \mathsf{rd}_{p,\rho})$. Henceforth it must be the case that $(\delta_{H0} \overset{\curvearrowright}{\text{CH}} \mathsf{rd}_{p,\rho})$ and the step can be performed rendering similar states.

(EW) In the case where the variable being accessed is a volatile, no step is taken in the cookbook-high level. The states remain related by $\mathcal{R}^{\varphi}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}$ because by 15 the atomicity set of cookbook-low writes is ignored for volatile accesses at the cookbook-high level. In the case of a normal reference, we need to prove that $\delta_{s0} \overset{\curvearrowright}{\text{CH}} \mathsf{wr}^{\mathcal{W},\mathcal{I}}_{\varrho,\mu}$ for which we use the exact same argument as used in the normal reference case of rule (NW) above.

(ER) We know then that δ_L can be decomposed as: $\delta_L = \delta_{L0} \cdot (t, \mathsf{wr}^{\mathcal{W} \cup \{t'\},\mathcal{I}}_{\varrho,\mu}) \cdot \delta_{L1} \cdot (t', \mathsf{rd}_{\varrho,\rho}) \cdot \delta_{L2}$. We then have two cases corresponding on whether the memory operations are on volatile or normal variables.

- If the memory access is volatile we have as we have reasoned above that: $\delta_H = \delta_{H0} \cdot (t, \mathbf{vwr}_{\varrho,\mu}) \cdot \delta_{H1} \cdot (t', \mathbf{vrd}_{\varrho,\rho}) \cdot \delta[H2]$. Since this is a volatile memory access, which at cookbook-high level can only be performed on memory (i.e. after the write $\mathbf{vwr}_{\varrho,\mu}$ has committed, as argued in the case (NW) of this proof), this will be a no-step in the semantics of cookbook-high. However, we need to show that the resulting configurations are similar, and more importantly that the resulting placeholder map is valid. The first condition is easy to check using the updated placeholder map $\varphi[\rho \mapsto \mu]$. This placeholder keeps track of the substitution in $\delta_{L2}[\rho \leftarrow \mu]$ and $\mathsf{T}_{L\rho \leftarrow \mu}$ that happens in this semantic step at the cookbook-low level. We show now that $\varphi[\rho \mapsto \mu]$ is a valid place-holder map as per Theorem 16. By the validity of φ we know that if $\varphi[\rho \mapsto \mu]$ is invalid it must be due to ρ. We evidently have the first condition of validity, that is that the placeholder ρ appears first in δ'_L (the resulting temporary store) as a $\overline{\mathsf{rd}_{\rho,\mu}}$. Similarly we have that it corresponds to a volatile access in δ_H. It remains to see that it is the first such access for thread t' in δ'_L. Assume on the contrary that it is not. Therefore, as we have argued before there should be a $(t', \langle \mathtt{l} | \mathtt{l} \rangle)$ in $\delta_{L0} \cdot (t, \mathsf{wr}^{\mathcal{W} \cup \{t'\},\mathcal{I}}_{\varrho,\mu}) \cdot \delta_{L1}$ by the rules of Table 3 and Table 4. This however would disable the application of the rule (ER) which specifically requires that $\delta_{L1} \overset{\curvearrowright}{\text{CL}} (t', \mathsf{rd}_{\varrho,\rho})$ and $\delta_{L0}\ {}_{\text{CL}}\overset{\curvearrowright}{\mathcal{S}_{sync}}\ (t', \mathsf{rd}_{\varrho,\rho})$. Therefore the resulting place-holder map is valid and we conclude the case.
- If the memory access is non-volatile we reason as above that the conditions imposed for the rule (ER) at the cookbook-low level imply that the rule can similarly be applied to the non-volatile reference at the cookbook-high level. We recall here that only volatile

references have SC semantics at the cookbook-high level, and therefore this case is much simpler than the previous one, not requiring to update the placeholder map φ.

(ERC) These rules apply equivalently at the cookbook-low and cookbook-high levels. We simply notice that at the point where a $\overline{\mathsf{rd}_{\_,\_}}$ operation is being performed at the cookbook-low level, the write operation that it read from has already been committed, both at the cookbook-low and cookbook-high level (see case (NR)), and for the case of volatile reads, all the read actions involving the write's volatile reference p have been replaced from $(t, \mathbf{vrd}_{\mathsf{p},\rho})$ to $(t, \overline{\mathbf{vrd}_{\rho,v}})$ during a (NR) step.

(FN) This rule does not affect the simulation relation. We rely on Lemma 19 to establish that the resulting cookbook-low configuration remains related to the cookbook-high configuration by $\mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}$.

◀

We can now present the final simulation theorem, which is simply a formal rewriting of the theorem Theorem 6 presented in section 5.

▶ **Theorem 24** (Cookbook-Low simulates Cookbook-High – Formal statement of Theorem 6)**.** *Provided two states from cookbook-high and cookbook-low respectively related by the simulation relation* $(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}\ (\sigma, \delta_L, \mathsf{T}_L)$ *such that* $(\sigma, \delta_L, \mathsf{T}_L)$ *can take a step:* $(\sigma, \delta_L, \mathsf{T}_L) \overset{\text{CL}}{\Longrightarrow} (\sigma', \delta'_L, \mathsf{T}'_L)$ *then there exists* φ' *such that either*

$$(\sigma, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi'}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}\ (\sigma', \delta'_L, \mathsf{T}'_L)$$

or there exists δ'_H *and* T'_H *with* $(\sigma, \delta_H, \mathsf{T}_H) \overset{\text{CH}}{\Longrightarrow}^{*} (\sigma', \delta'_H, \mathsf{T}'_H)$ *and*

$$(\sigma, \delta'_H, \mathsf{T}'_H)\ \mathcal{R}^{\varphi'}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}\ (\sigma, \delta'_L, \mathsf{T}'_L)$$

Proof. The proof considers the cases of the step being taken in $(\sigma, \delta_L, \mathsf{T}_L) \overset{\text{CL}}{\Longrightarrow} (\sigma', \delta'_L, \mathsf{T}'_L)$. If the step is an intra-thread step (as per Figure 3) we use Theorem 19 to conclude. If on the other hand the step taken is a memory step (as per Figure 5) we apply Theorem 23 to conclude the proof. ◀

(LK) Lock

$$(\mathcal{O}, \sigma, \delta_0 \cdot (t, \mathtt{lk}_\ell) \cdot \delta_1, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\mathcal{O}[\ell \leftarrow t], \sigma, \delta_0 \cdot \delta_1, \mathsf{T}) \quad \left[\begin{array}{c} \mathcal{O}(\ell) = \bot \\ (t, \mathtt{lk}_\ell) \overset{\curvearrowleft}{\text{CH}} \delta_0 \end{array} \right]$$

(UL) Unlock

$$(\mathcal{O}, \sigma, \delta_0 \cdot (t, \mathtt{ul}_\ell) \cdot \delta_1, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\mathcal{O}[\ell \leftarrow \bot], \sigma, \delta_0 \cdot \delta_1, \mathsf{T}) \quad \left[\begin{array}{c} \mathcal{O}(\ell) = t \\ (t, \mathtt{ul}_\ell) \overset{\curvearrowleft}{\text{CH}} \delta_0 \end{array} \right]$$

Figure 8 Implementation of Locks in cookbook-high (CH).

B Locks

To implement locks at the cookbook-high level we will extend the configurations of cookbook-high with a lock map. This is simply a mapping from lock values to thread IDs or a default value $\bot$ which is not a thread ID ($\bot \notin \mathcal{T}id$). We will denote lock maps with the metavariable $\mathcal{O}$ and we assume the standard notation for substitution to update the lock map. We will assume that when a thread t holds the lock ℓ we will have $\mathcal{O}(\ell) = t$ and when the lock is free we will have $\mathcal{O}(\ell) = \bot$. The default state, which we use for initial states, has $\bot$ for all locks.

B.1 Cookbook-High Locks

The semantics of lock memory operations for cookbook-high is given in Figure 8 which extends the rules of Figure 5, and where the commutativity relation is derived from the conflict relation presented in Table 3.

We need to extend the minimal conflict relation ($\#_\blacktriangleleft$) that we defined in section 4 to capture the fact that a lock operation issued on a placeholder value does indeed conflict with the read operation of the same thread that generated the placeholder

$$\mathsf{mo} \in \{\mathsf{rd}_{\varrho,\rho}, \mathbf{vrd}_{\varrho,\rho}\} \;\&\; \mathsf{mo}' \in \{\mathtt{lk}_\rho, \mathtt{ul}_\rho\} \Rightarrow (t, \mathsf{mo}) \#_\blacktriangleleft (t, \mathsf{mo}')$$

The commutativity relation of cookbook-high is consistent with this extension to the conflict relation as defined in Table 3. Moreover, while Table 3 is for simplicity defined using lock and unlock operations operating on lock values ($\mathtt{lk}_\ell$, $\mathtt{ul}_\ell$) the same constraints apply for lock and unlock operations operating on placeholder values ($\mathtt{lk}_\rho$, $\mathtt{ul}_\rho$). Finally, and as we did before with volatile variables, we consider only executions of cookbook-high that respect the syntax. That is, no lock values can be used as the target of load or store operations, and no normal references can be used as the target for lock and unlock operations. Commands attempting such operations are considered erroneous[13].

B.2 Cookbook-Low Locks

At cookbook-low level we have that locks are implemented with normal variables, as we did with volatile variables before. Then, at cookbook-low we will consider that lock variables are part of the normal references. If we consider the set of $\mathcal{P}tr_H$ to the set of cookbook-high normal references and $\mathcal{P}tr_L$ to the set of cookbook-low references, we will have that $\mathcal{P}tr_H \cup \mathcal{V}ptr \cup \mathcal{L}ocks \subseteq \mathcal{P}tr_L$. Once more, since we assumed in cookbook-high that $\mathcal{L}ocks \cap$

[13] In fact this attempting that would be imposible in Java

(LK) Lock

$$(\sigma, \delta_0 \cdot (t, \mathtt{lk}_\ell) \cdot \delta_1, \mathsf{T}) \underset{\text{CL}}{\hookrightarrow} (\sigma[\ell \leftarrow t], \delta_0 \cdot \delta_1, \mathsf{T}) \quad \left[\begin{array}{c} \sigma(\ell) = \bot \\ (t, \mathtt{lk}_\ell) \overset{\curvearrowleft}{\text{CL}} \delta_0 \end{array} \right]$$

(UL) Unlock

$$(\sigma, \delta_0 \cdot (t, \mathtt{ul}_\ell) \cdot \delta_1, \mathsf{T}) \underset{\text{CL}}{\hookrightarrow} (\sigma[\ell \leftarrow \bot], \delta_0 \cdot \delta_1, \mathsf{T}) \quad \left[\begin{array}{c} \sigma(\ell) = t \\ (t, \mathtt{ul}_\ell) \overset{\curvearrowleft}{\text{CL}} \delta_0 \end{array} \right]$$

Figure 9 Implementation of Locks in cookbook-low (CL).

$\mathcal{P}tr_H = \emptyset$ there will be no confusion between normal references at cookbook-low and references implementing locks.

We will make a further assumption at cookbook-low level that references implementing lock operations in cookbook-high cannot be manipulated in an unconstrained fashion at cookbook-low. That is, accesses to references in $\mathcal{L}ocks \subseteq \mathcal{P}tr_L$ will only be accessed through lock and unlock operations. This is a direct consequence of the assumption that we made at cookbook-high level. This implies that the case of a read-early memory access we considered for volatile references when compiled into cookbook-low is not an issue for locks, which therefore do not need to be considered in the placeholder map discussed for the definition of Theorem 17. This simplifies the arguments about locks w.r.t. volatile variables significantly. Otherwise considering the conflict relation of Table 4 where we equate lock operations with normal reads (not subject to the $\overline{\mathsf{rd}}$ operation) and unlock operations with normal writes, we obtain the cookbook-low extended conflict relation for locks, which defines our semantics. Moreover the compilation requirements of Table 5 imply that this assumption is correct. We remark here that in [16] additional barriers for monitorenter and monitorexit are considered for the cases in which lock is not implemented by reading and/or unlock is not directly implemented using a writing operation. In our work we will consider that a lock operation is implemented some variation of test-and-set (a form of `cas` for x86 and a form of `LL`/`SC` for Power) in which case we can ignore this extension.

We now present the extension of Theorem 18 to consider lock and unlock operations. We need to extend the definition of the simulation relation Theorem 17 to capture the lock map component at the cookbook-high level.

▶ **Definition 25.** We say a configuration of the cookbook-high $(\mathcal{O}, \sigma_H, \delta_H, \mathsf{T}_H)$, is *similar* to a configuration of cookbook-low $(\sigma_L, \delta_L, \mathsf{T}_L)$ for the initial thread cookbook-high thread system T_{H0} which is well-compiled to T_{L0} if there exists a placeholder map φ such that:

1. $\mathsf{dom}(\sigma_L) = \mathsf{dom}(\sigma_H) \cup \mathsf{dom}(\mathcal{O})$ and
2. $(\sigma_H[\mathcal{O}], \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}\ (\sigma_L, \delta_L, \mathsf{T}_L)$

We will overload the notation $\mathcal{R}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}$, which can easily be disambiguated by the cookbook-high configuration at the left. We will then write:

$$(\mathcal{O}, \sigma_H, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}\ (\sigma_L, \delta_L, \mathsf{T}_L)$$

We notice here that the only modification between the previous simulation relation of Theorem 17 and the new one in Theorem 25 is on the store and the lock map. Hence, all results that applied to the temporary store (e.g. Theorem 20) are still valid with this new definition.

▶ **Corollary 26.** *Suppose that we have the following states related by simulation:* $(\mathcal{O}, \sigma_H, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}$ $(\sigma_L, \delta_L, \mathsf{T}_L)$*. Moreover suppose that by Theorem 20 we have that:*

1. $\delta_L = \delta_{L0} \cdot (t, \mathsf{mo}) \cdot \delta_{L1}$ *where* $\mathsf{mo} \in \{\mathtt{lk}_\ell, \mathtt{ul}_\ell\}$,
2. $\delta_H = \delta_{H0} \cdot (t, \mathsf{mo}') \cdot \delta_{H1}$ *where* $\mathsf{mo}' \in \{\mathtt{lk}_\rho, \mathtt{ul}_\rho\}$ *accordingly with the previous item,*
3. $\delta_{L0}\Downarrow \simeq (\delta_{H0}[\varphi])\Downarrow$ *and* $\delta_{L1}\Downarrow \simeq (\delta_{H1}[\varphi])\Downarrow$*, and*
4. $(t, \mathsf{mo}) \simeq (t, \mathsf{mo}')[\varphi]$*, meaning that* $\varphi(\rho) = \ell$*.*

it must then be the case that

$$\delta_{L0} = \delta'_{L0} \cdot (t', \mathsf{wr}^{\mathcal{W}\cup\{t\},\mathcal{I}\cup\{\rho\}}_{\mathsf{p},p}) \cdot \delta''_{L0} \cdot (t, \overline{\mathsf{rd}_{\rho,p}}) \cdot \delta'''_{L0}$$

and moreover, δ'''_{L0} *can be decomposed as:*

$$\delta'''_{L0} = \delta^0_{L0} \cdot (t, \langle \mathtt{l}|\mathtt{l}\rangle) \cdot \delta^1_{L0} \text{ if } \mathsf{mo} = \mathtt{lk}_\ell \text{ and}$$

$$\delta'''_{L0} = \delta^0_{L0} \cdot (t, \langle \mathtt{l}|\mathtt{s}\rangle) \cdot \delta^1_{L0} \text{ if } \mathsf{mo} = \mathtt{ul}_\ell$$

meaning that if we have a locking operation (t, mo) *in* δ_L *whose corresponding operation in* δ_H *contains a placeholder value, then there must be a barrier b operation by thread t (with a preceding early-read by t) before the locking operation in* δ_L*.*

Proof. The proof is identical to that of Theorem 21

▶ **Lemma 27.** *We can reproduce Theorem 23 to the extended simulation relation* $(\mathcal{O}, \sigma_H, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}$ $(\sigma_L, \delta_L, \mathsf{T}_L)$*, and considering locks.*

Proof. We need to consider the case of a (LK) and (UL) at the cookbook-low level, all the other cases are an immediate consequence of Theorem 23.

(LK) This case is similar to the (NR) case where the reference is volatile. We have by that $\delta_L = \delta_{L0} \cdot (t, \mathtt{lk}_\ell) \cdot \delta_{L1}$, $\sigma'_L = \sigma_L[\ell \leftarrow t]$, $\delta_{L0} \overset{\curvearrowleft}{\text{CL}} (t, \mathtt{lk}_\ell)$, $\delta'_L = \delta_{L0} \cdot \delta_{L1}$ and $\mathsf{T}'_L = \mathsf{T}_L$. By the hypothesis $(\mathcal{O}, \sigma_H, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}$ $(\sigma_L, \delta_L, \mathsf{T}_L)$, Theorem 20 and Theorem 26 we have $\delta_H = \delta_{H0} \cdot (t, \mathtt{lk}_\ell) \cdot \delta_{H1}$. (Notice that if we had a placeholder $\rho \in \mathsf{dom}(\varphi)$ with $\varphi\rho = \ell$ and instead of $(t, \mathtt{lk}_\ell)$ in δ_H we had $(t, \mathtt{lk}_\rho)$ we would have a contradiction to Theorem 26 since a (LK) step is being taken at the cookbook-low level, and the corollary implies that there should be a barrier in δ_{L0} to prevent it.) Evidently, if we can perform a (LK) step reducing $(t, \mathtt{lk}_\ell)$ in cookbook-high we arrive at states related by $\mathcal{R}^{\rho}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}$, since the modification in the store σ'_L, corresponds to the modification of the lock map at cookbook-high ($\mathcal{O}[\ell \leftarrow t]$). We only need to prove that $\delta_{H0} \overset{\curvearrowleft}{\text{CH}} \mathtt{lk}_\ell$. Assume then by contradiction that $\neg(\delta_{H0} \overset{\curvearrowleft}{\text{CH}} \mathtt{lk}_\ell)$. Then there must exist a memory operation (t, mo) in δ_{H0} such that $\mathsf{mo} \#_{\text{CH}} \mathtt{lk}_\ell$ (operations on other threads cannot conflict with a lock from this thread). On inspection of Table 3 we find that mo must be another volatile memory access or lock action. But then, by Theorem 20 and Theorem 17 there must be a barrier b in δ_{L0} such that $b \#_{\text{CL}} \mathsf{rd}_{p,\rho}$ contradicting $\delta_{L0} \overset{\curvearrowleft}{\text{CL}} (t, \mathtt{lk}_\ell)$ which is required for the cookbook-low memory step to happen. We conclude that $(\delta_{H0} \overset{\curvearrowleft}{\text{CH}} \mathtt{lk}_\ell)$ which ends this case.

(UL) We have by that $\delta_L = \delta_{L0} \cdot (t, \mathtt{ul}_\ell) \cdot \delta_{L1}$, $\sigma'_L = \sigma_L[\ell \leftarrow \bot]$, $\delta_{L0} \overset{\curvearrowleft}{\text{CL}} (t, \mathtt{ul}_\ell)$, $\delta'_L = \delta_{L0} \cdot \delta_{L1}$ and $\mathsf{T}'_L = \mathsf{T}_L$. By the hypothesis $(\mathcal{O}, \sigma_H, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}$ $(\sigma_L, \delta_L, \mathsf{T}_L)$, Theorem 20 and Theorem 26 we have $\delta_H = \delta_{H0} \cdot (t, \mathtt{ul}_\ell) \cdot \delta_{H1}$ as before. Evidently, if we can perform a (UL) step reducing $(t, \mathtt{lk}_\ell)$ in cookbook-high we arrive at states related by $\mathcal{R}^{\rho}_{(\mathsf{T}_{H0},\mathsf{T}_{L0})}$, since the modification in the store σ'_L, corresponds to the modification of the lock map

at cookbook-high ($\mathcal{O}[\ell \leftarrow \bot]$). We only need to prove that $\delta_{H0} \overset{\curvearrowleft}{\text{CH}} \texttt{ul}_\ell$. Assume then by contradiction that $\neg(\delta_{H0} \overset{\curvearrowleft}{\text{CH}} \texttt{ul}_\ell)$. Then there must exist a memory operation (t, mo) in δ_{H0} such that $\mathsf{mo} \#_{\text{CH}} \texttt{ul}_\ell$ (once more, operations on other threads cannot conflict with a lock from this thread). On inspection of Table 3 we find that any other memory operation by the same thread conflicts with $\texttt{ul}_\ell$. But then, by Theorem 20 and Theorem 17 there must be a barrier b in δ_{L0} such that $b \#_{\text{CL}} \mathsf{rd}_{p,\rho}$ contradicting $\delta_{L0} \overset{\curvearrowleft}{\text{CL}} (t, \texttt{lk}_\ell)$ which is required for the cookbook-low memory step to happen. We conclude that $(\delta_{H0} \overset{\curvearrowleft}{\text{CH}} \texttt{lk}_\ell)$ which ends the whole proof.

◀

▶ **Theorem 28** (Cookbook-Low simulates Cookbook-High). *Given states of cookbook-high and cookbook-low related by the simulation relation extended with locks*

$$(\mathcal{O}, \sigma_H, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}\ (\sigma_L, \delta_L, \mathsf{T}_L)$$

such that $(\sigma_L, \delta_L, \mathsf{T}_L)$ *can take a step:*

$$(\sigma_L, \delta_L, \mathsf{T}_L) \overset{\text{CL}}{\Longrightarrow} (\sigma'_L, \delta'_L, \mathsf{T}'_L)$$

then there exists φ' *such that either*

$$(\mathcal{O}, \sigma_H, \delta_H, \mathsf{T}_H)\ \mathcal{R}^{\varphi'}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}\ (\sigma'_L, \delta'_L, \mathsf{T}'_L)$$

or there exists δ'_H *and* T'_H *with*

$$(\mathcal{O}, \sigma_H, \delta_H, \mathsf{T}_H) \overset{\text{CH}}{\Longrightarrow}^{*} (\mathcal{O}', \sigma'_H, \delta'_H, \mathsf{T}'_H)$$

and

$$(\mathcal{O}', \sigma'_H, \delta'_H, \mathsf{T}'_H)\ \mathcal{R}^{\varphi'}_{(\mathsf{T}_{H0}, \mathsf{T}_{L0})}\ (\sigma'_L, \delta'_L, \mathsf{T}'_L)$$

Proof. The proof is identical to Theorem 18 using Theorem 27. ◀

B.3 Implementation of Locks

As we mentioned in the introduction, we will not provide an implementation for locks, but rather assume that the are a number of low-level steps that implement lock and unlock operations. We will assume linearizable implementations of these operations, and we will reduce the simulation argument between the low-level architectural implementation to make the cookbook-low lock step at the linearization point of the implementation. In particular, for x86 we assume that locks are implemented using (cas), upon whose successful execution we will have the linearization point. For unlock, we simply assume a low-level store operation as being the linearization point. Any implementation that has these or more operations in the implementations of low-level locks will satisfy the properties required by our proofs. A similar argument is made for Power, where we assume that a store conditional (strex) operation is the linearization point of lock, and a store is the linearization point of unlock.

x86

To cope with the (cas) instruction of x86 we will simply assume that it is an atomic load/store operation followed by an mfence[14]. It is trivial to observe that, since mfence prevents the only

[14] We could further require to have an mfence before the (cas) but since store operations cannot bypass any other operation in x86 this would be redundant.

source of reordering in x86, and a store operation (for the successful (cas)) cannot bypass any other operations in x86 [22], this operation implies a full barrier w.r.t. any preceding and subsequent operation, in which case we have that the fences added in cookbook-low by grace of Table 5 are evidently fulfilled by this (cas) operation. This implies by Theorem 7 that the simulation can be immediately realized.

Power

For will not encode the semantics of LL/SC in Power, but simply assume that they can perform an atomic read and store on the reference that implements the lock. Since no assumptions are made other than that LL/SC performs a load, and that the implementation of unlock performs a store, we have to add all the barriers as indicated by Table 5 and translated according to Figure 7. It is therefore evident that the same argument that we used in Theorem 9 applies to locks, and we conclude that the simulation can be extended with to lock operations immediately.

C Cookbook-High – JMM conformance

C.1 From operational Runs to JMM Executions

The key insight of this proof consists on a strategy to build a JMM trace using the steps generated by the semantics presented in Figure 5. To that end, we will firstly define in succinct terms the semantics of the JMM as presented by [20] and we will then instrument the semantics of Figure 5 with a JMM execution. Let us firstly refresh the definition of the JMM as presented in [20].

JMM Executions

The definition below captures the notion of actions similar to the one presented in [20]. We diverge slightly from the notation to reuse the notion of memory operation introduced in Figure 3.

▶ **Definition 29** (JMM Actions). A JMM memory action – ranged over with the variable a – is a triple $a = (\mathsf{mo}, t, u)$ comprising:

1. a memory operation mo is restricted to be a read or write operation, either on a normal or a volatile reference, or a lock or unlock operation[15]. Moreover, these operations cannot contain placeholder values,
2. a thread identifier t, which corresponds to the thread that generated the memory operation, and finally,
3. a unique identifier u for the event. We will not show how such unique identifiers are generated, but we point out that it suffices to add to the state a global counter, which is incremented each time an operation is added into the temporary store by the intra-thread semantics of Figure 3 and it is attached to each memory operation generated. This is just a proof artifact that does not affect the semantics in any way. We will hence consider that memory operations in the temporary store (δ) are uniquely identified by u throughout the life time of the program. These identifiers will be used to match memory operations in the temporary store with JMM actions being committed (see below).

▶ **Definition 30** (Execution). A JMM execution of program $P = \mathsf{c}_0 \| \dots \| \mathsf{c}_n$ is a 6-tuple – ranged over by ξ – of the form $\xi = \langle \mathsf{A}, P, \xrightarrow{po}, \xrightarrow{so}, \mathcal{WS}, \mathcal{VW} \rangle$ comprising the following components:

1. a set A of JMM memory actions as in Theorem 29,
2. the text of the program P,
3. a partial relation $\xrightarrow{po}$, relating the actions of A representing the program order of P. This relation relates only actions originating from the same thread.
4. a partial relation relation $\xrightarrow{so}$ (total on the synchronization actions of the set A) relating all synchronization operations. All JMM actions other than normal reference reads and writes are considered synchronization actions (i.e. volatile and lock actions).
5. A partial function $\mathcal{WS} : \mathsf{A} \to \mathsf{A}$ which for each read action (normal or volatile) maps to a unique write action, from which the value read was generated, and finally

[15] The full JMM considers more operations including thread start and end, which are unimportant in the core calculus considered here.

6. a partial function $\mathcal{VW} : \mathsf{A} \to \mathcal{V}al$ wich for each write action returs the value that was written. Notice that our encoding of actions includes the value in the action, which means that this function is not necessary and will not be used. We add it for completeness.

The following is the standard definition of the synchronizes-with order of the JMM.

▶ **Definition 31** (Synchronizes-With Order)**.** We define the *synchronizes-with* order, which is derived from the synchronization-order ($\xrightarrow{so}$), if the following conditions are met on a pair of actions $a_0 \xrightarrow{sw} a_1$:

1. a_0 and a_1 are both operations on the same volatile reference p or on the same lock ℓ, and
2. a_0 is a volatile write or an unlock operation, and a_1 is a volatile read on the same reference as a_0 or a lock operation on the same lock as a_0, and
3. $a_0 \xrightarrow{so} a_1$.

The following is the standard definition of the happens-before order of the JMM.

▶ **Definition 32** (Happens-Before Order)**.** We define the *happens-before* to be the irreflexive transitive closure of the union of the program-order and the synchronizes-with order

$$\xrightarrow{hb} = (\xrightarrow{po} + \xrightarrow{so})^+$$

For clarity we omit the definiton of sequential validity of the JMM as presented in [20], since this is obviously true in our Cookbook-High semantics. Notice that since in the JMM, since there is no operational semantics, these notions have to be axiomatized in an ad-hoc way.

Similarly, the following definition imposes constraints on the posible JMM executions. We just present the definition here, and point out that our Cookbook-High semantics respects them trivially. We shall refrain from proving this proposition which although straightforward is tedious to prove.

▶ **Definition 33** (Well-Formed Execution)**.** We say a JMM execution $\xi = \langle \mathsf{A}, P, \xrightarrow{po}, \xrightarrow{so}, \mathcal{WS}, \mathcal{VW} \rangle$ is *well-formed* if the following conditions are met:

1. A is finite,
2. $\xrightarrow{po}$ is a total order on the actions of a single thread, and two actions of different threads are incomparable,
3. $\xrightarrow{so}$ is total on the synchronization actions of A,
4. $\mathcal{WS}$ is well typed,
5. Locking is proper: At any point a lock is held by at most one thread, and each lock is released by the thread that acquires it as many times as it acquired it,
6. $\xrightarrow{po}$ is consistent with the semantics of each of the commands c of program P. (This is evidently true in Cookbook-High since the actions in A are generated by the intra-thread semantics of Figure 3.)
7. $\mathcal{WS}$ is consistent with $\xrightarrow{so}$. That is, each volatile read sees exactly the last volatile write before it in $\xrightarrow{so}$ (notice that since we assume initialization actions, and $\xrightarrow{so}$ is total, exactly one such write exists). A similar condition is established for locks,
8. $\mathcal{WS}$ is consistent with $\xrightarrow{hb}$. That is, for each read action r it is not the case that $r \xrightarrow{hb} \mathcal{WS}(r)$ nor that there exists a write action w on the same reference as r such that $\mathcal{WS}(r) \xrightarrow{hb} w \xrightarrow{hb} r$.

While the semantics of Cookbook-High is operational, and events are generated as the program is executed, the semantics of the JMM is axiomatic. In the JMM, assuming a hypothetical execution ξ one justifies the execution with a series of steps that – using other executions – justify each of the actions in ξ. We will present here the process to justify a final execution ξ, but in our treatment, we will consider that we are justifying an execution that is not known ahead of time. Instead, we will justify steps as they happen, and show that if the operational semantics of Cookbook-High makes progress, all of the generated steps can be justified. This means that we are considering a hypothetical final execution ξ. We have then to show that axioms that relate the final execution with the committing executions are preserved by the semantics.

The following is a slight modification the definition of legal executions of the JMM as presented by [20].

▶ **Definition 34** (Legal Execution)**.** We say that a JMM well-formed execution $\xi = \langle \mathsf{A}, P, \xrightarrow{po}, \xrightarrow{so}, \mathcal{WS}, \mathcal{VW} \rangle$ is *legal* if there is a sequence of executions-commit set pairs (ξ_i, C_i) where the last element in the sequence is identical to ξ and all the actions of ξ are in the last commit set, and each $\xi_i = \langle \mathsf{A}_i, P, \xrightarrow{po}_i, \xrightarrow{so}_i, \mathcal{WS}_i, \mathcal{VW}_i \rangle$ is a well formed execution and the following conditions are met for each of them[16]:

1. $C_{i-1} \subseteq C_i$,
2. $C_i \subseteq \mathsf{A}_i$,
3. for each read action $r \in C_i$ we have $\mathcal{WS}(r) \xrightarrow{hb} r \iff \mathcal{WS}(r) \xrightarrow{hb}_i r$, and it is never the case that $r \xrightarrow{hb}_i \mathcal{WS}(r)$,
4. $\mathcal{VW}_i|_{C_i} = \mathcal{VW}|_{C_i}$,
5. $\mathcal{WS}i|_{C_{i-1}} = \mathcal{WS}|_{C_{i-1}}$,
6. for each read $r \in \mathsf{A}_i/C_{i-1}$ we have $\mathcal{WS}_i(r) \xrightarrow{hb}_i r$, and
7. for each read $r \in C_i/C_{i-1}$ we have $\mathcal{WS}(r) \in C_i$.

C.2 Extending Cookbook-High Runs

To relate Cookbook-High executions with JMM executions, we will extend the notion of a configuration considered in section 4 (in fact the extension with locks considered in Figure 8) to include a current committing JMM execution. Notably this JMM execution will have as committed actions all the memory operations that the current Cookbook-High execution has committed so far. That is, when removing a memory operation mo from the temporary store δ in a Cookbook-High configuration, say $(\mathcal{O}, \sigma, \delta, \mathsf{T})$, we will add an action a corresponding to the memory operation mo (i.e., they have the same unique identifier) into the commit set of the current JMM execution ξ. Therefore, Cookbook-High executions are now a 6-tupe $(\xi, C, \mathcal{O}, \sigma, \delta, \mathsf{T})$, where ξ is the current committing execution, generated by the last commit rule (from Figure 5), and C is the set of committed actions in ξ so far.

Committing procedure

We now describe a committing procedure to justify actions à la JMM as they are executed by the semantics of Cookbook-High. This procedure will modify the ξ and C components of the extended configurations considered before in accordance with the Cookbook-High step being executed.

[16] We omit legality rules for external actions which are treated as synchronization actions.

▶ **Definition 35.** Consider the execution of a memory action

$$(\mathcal{O}, \sigma, \delta, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\mathcal{O}', \sigma', \delta', \mathsf{T}')$$

as given by the semantics of Figure 5 and Figure 8. We can construct a step

$$(\xi, C, \mathcal{O}, \sigma, \delta, \mathsf{T}) \underset{\text{CH}}{\hookrightarrow} (\xi', C', \mathcal{O}', \sigma', \delta', \mathsf{T}')$$

of the semantics extended with JMM justifications based on the following cases:

- If the rule being executed is (SC), (EW), (ER) or (VRC) we just propagate the ξ and C unmodified[17].
- If the action is any of (VW), (VR), (LK) or (UL) we extend the current JMM execution ξ and the committed set C in the following way:
  - Notice that all of these rules can happen only if we have specific volatile pointers p or lock ℓ depending on the rule being used.
  - Notice that for all all the rules except (VR) all the values of the memory operation being committed are known. For the particular case of (VR) we have a memory operation of the form $\mathbf{vrd}_{\mathsf{p},\rho}$ that is being committed. Since this rule is deterministic, it will replace the placeholder ρ by the current value in $\sigma(\rho)$.
  - We then generate the JMM action that is exactly the current memory operation (say (u, t, mo)) for all cases but for (VR), and for this latter case, we generte the action $(u, t, \mathbf{vrd}_{\mathsf{p},\sigma(\mathsf{p})})$ (the unique identifier is supposed to be part of the memory operation as we mentioned above). We then add the generated action to the actions of the execution $\mathsf{A}' = \mathsf{A} \cup \{(u, t, \mathsf{mo})\}$ and to the committed actions set (i.e. $C' = C \cup \{(u, t, \mathsf{mo})\}$)) and extend the $\xrightarrow{po}$ and $\xrightarrow{so}$ of the current execution accordingly. Evidently we extend the $\mathcal{WS}$ and $\mathcal{VW}$ functions in the obvious way. We will later show that adding these operations always respects the legality conditions of the JMM.
- If the rule being executed is a normal write action (NW), the conditions stated above hold similarly, where the memory operation being committed is of the form $\mathsf{wr}_{p,v}^{\mathcal{W},\mathcal{T}}$ where importantly the action contains no placeholder in the target pointer, or the value being written. We proceed as in the item above, except that there is no need to update the $\xrightarrow{so}$ order.
- If the rule being executed is a of a normal read action (NR) we have two cases.
  1. If the target write (that is the write that stored the value presently at $\sigma(p)$) is related by the happens-before ($\xrightarrow{hb}$) relation considering the execution ξ extended with the read, we just add it as in the cases above.
  2. If however, the read is not related by happens-before (recall that adding a normal read does not modify the $\xrightarrow{so}$ order), we have to do more work, since when a read is committed it has to immediately see a value that is $\xrightarrow{hb}$ related to it (see item 6 of Theorem 34). Since one such write must exists in any JMM computation, we choose any, add the read to the commit set C and immediately modify the generated trace to have the read point to the write that it actually sees (that is the write that store the value in $\sigma(p)$).

[17] Note that the rule (VRC) was added just as a technical convenience, but it does not have any effect in the semantics.

- If the rule is a early read commit action (ERC) we proceed as in the case above. The only difference being that the target write might not be the one that stored the current value at $\sigma(p)$. However, as we shall see it must be the case that the target write is already committed in C and moreover, it satisfies the conditions of legality of Theorem 34.

As we mentioned before, we assume there is an action unique identifier generated for each memory operation that is added into the temporary store by the semantics of Figure 3. This shall serve both to relate committed actions with the memory operation that generated them, and as the unique identifier for actions as prescribed by the JMM formalization. We shall not say more about these identifiers, and we shall freely use the fact that actions can be uniquely identified, and mapped to a prior state where the memory operation that generated the action was present in the temporary store as justified by the following lemma.

▶ **Lemma 36** (Memory Action – Memory Operation)**.** *For every action a in the current execution of a composite state $(\xi, C, \mathcal{O}, \sigma, \delta, \mathsf{T})$ there exists a pior state in the Cookbook-High execution leading to $(\xi, C, \mathcal{O}, \sigma, \delta, \mathsf{T})$ such that there was a memory operation in the temporary store of that state with the same identifier as a.*

Proof. Obvious by the committing procedure and the generation of unique identifiers in the semantics. We use this lemma as a basis to map actions and memory operations throughout this proof.

▶ Remark (Operations in δ appear in program order). It is clear from generation of memory operations in the intra-thread semantics of Figure 3 and the committing semantics of Figure 5 and Figure 8 that all operations in the temporary store are ordered according to the program order of the participating threads.

We now argue, as we have anticipated in the racy-read commit case of Theorem 35 that we can commit the read seeing a happens-before related write, and then modify it to match the read actually seen in the Cookbook-High run.

In the JMM, while justifying the legality of an execution, read actions that are not committed are only allowed to see writes that are related to them by happens-before (again see item (6) of Theorem 34). We note here that forcing reads to see happens-before related writes *only* implies that the trace might not even be sequentially consistent (SC). The following example illustrates this point.

$$\begin{gathered} x = y = 0 \\ \left[\begin{array}{l} x := 1; \\ r_1 = y \end{array}\right] \quad \| \quad \left[\begin{array}{l} y := 1; \\ r_2 = y \end{array}\right] \\ r_1 = r_2 = 1 \end{gathered}$$

While the example above is clearly CS. To justify it we have to start with an execution where both reads, r_1 and r_2 see a happens-before related value (in this case the initial value 0). However, this initial execution where both reads see the default value of 0 will only be used to justify the writes, and the initial commitment of the reads, but the final execution will make both reads see the other thread's write action, therefore these initial reads will be discarded. While in the JMM committing process one can remap the write seen for a read, in Cookbook-High reads can only see a unique and final value during their execution. We therefore must forge a read to commit read actions as required by rule (6), which will immediately be discarded and the read will be made to see the write of the final trace. To do so, we will argue that for any read that must be committed there exists a prior visible

write that is happens-before related. Since the value of this write will not be the final value seen by the read, we do not care about it. Moreover, the immediately subsequent committed execution will have the read pointing to the final in the execution being justified. Since we commit actions one at a time, we will commit read seeing any prior write action (we will leave it unspecified since we know one such write exists), and will immediately map it to the correct write it sees.

▶ Remark. For any read action being justified in the JMM there exists a previously committed write action that is happens-before related to it.

Proof. This lemma is an immediate consequence of the fact that the initialization writes are happens-before related to any action on the same reference, and committing more writes can only extend the happens-before relation, but not reduce it.

This remark guarantees that the in process used for committing reads in Theorem 35 there is always a way to commit reads, and then point the write-seen function ($\mathcal{WS}$) to the seen write right after the commit.

To facilitate some further arguments, we will firstly prove that the much more constraining semantics of Cookbook-High w.r.t. the JMM guarantees that a slightly weaker notion than the happens-before order – which we shall name the *synchronized-before* order is respected by our committing procedure indicated in Theorem 35.

▶ **Definition 37** (Synchronized-Before Order)**.** Consider the *synchronized-before* order to be defined by the following equation:

$$\xrightarrow{\mathsf{sb}} = (\xrightarrow{po}{}^{*} \xrightarrow{sw} \xrightarrow{po}{}^{*})^{+}$$

That is, the irreflexive transitive closure of the symchronizes-with order extended with the program order before and after.

The definition above relates to each action, every other that action that is happens-before related to it, except if the latter action is only related by program-order to the former, with no synchronization actions in-between.

▶ **Lemma 38** (Cookbook-High respects Synchronized-Before)**.** *Given two actions a_0 and a_1 in* A *such that they are related as: $a_0 \xrightarrow{\mathsf{sb}} a_1$. Then, it must be the case that the memory operation that generated a_0 was committed before the memory operation that generated a_1 in the committing procedure of Theorem 35 (i.e. they were committed in order by the Cookbook-High trace).*

Proof. This proof is immediate by Theorem 35. Let us consider firs the example of two synchronization actions related by $\xrightarrow{so}$ in the resulting JMM trace. By the construction of Theorem 35 they must be committed in that order. The same applies to volatile memory actions. A simple observation of the ordering conditions of Cookbook-High presented in section 4 and the conflict relation $\#_{\text{CH}}$ extends the argument above to the full synchronized-before order.

The following lemma establishes that the semantics of Cookbook-High commits actions from the temporary store that are related by causality in order. By causality we mean that the result of the first operation is necessary for the second operation to happen. An example is a read for which the value read will be used in the second operation (e.g. a write to a reference read in the first action). Another example is two operations on the same reference.

▶ **Lemma 39** (Cookbook-High respects Data-Dependencies)**.** *Assume two actions in the* JMM *resulting execution such that they are in data-dependency relation. Suppose an action a can only result as a consequence of a read seeing a particular value v, or the case where the two actions operate on the same reference. Then, the these actions are committed in order by Cookbook-High.*

Proof. We observe first that two memory operations on the temporary store that are related by causality either share a placeholder value, or operate on similar references. Then, by inspection of the relation $\#_{\text{CH}}$ and the minimal conflict relation $\#_{\blacktriangleleft}$ of section 4 we can observe that it will never be the case that the second operation – say mo' – can be reordered w.r.t. the first operation – say mo – i.e. it is not the case that $\mathsf{mo} \overset{\curvearrowleft}{\text{CH}} \mathsf{mo}'$ which then by the rules of Figure 5 implies that mo must be removed from the temporary store before mo' is, and ultimately by the procedure of Theorem 35 means that the action generated by mo is justified before mo'.

It only remains to see that the procedure described in Theorem 35 guarantees the legality of the produced execution ξ according to the rules of Theorem 34. This is the purpose of our final theorem.

▶ **Theorem 40** (Formal statement of Theorem 3)**.** *The* JMM *execution produced by running the Cookbook-High semantics equipped with procedure delineated in Theorem 35 is guaranteed to be a legal* JMM *execution according to the axioms of Theorem 34, and furthermore it produces the same behaviors. In essence, the execution produced by Cookbook-High is a legal* JMM *execution.*

Proof. We argue about the elements of Theorem 34 one by one. We consider that the different commit sets (and JMM executions) that are produced by Theorem 35 are the different justifications of the final execution.

1. Item (1) is trivially true, since Theorem 35 only adds actions to the committed set.
2. Item (2) is also trivially true, since the actions that are added to C are also added into A.
3. Item (3) refers only to the rule for committing a read action. The first condition of this rule requires that if the final write a read sees is happens-before related to it, it must be so in the trace that justifies the read. Since for us, the case where the write is happens-before related is done in a single commit (as per Theorem 35) this is trivially true. The second condition requires that never a read that sees a write contradict the happens-before relation. This is true in our semantics as established by Theorem 38 and Theorem 39.
4. Condition (4) is trivially true because the value written by write actions never changes in Cookbook-High, let alone for committed writes.
5. Condition (5) refers to read actions. As we devised Theorem 35, care has been taken to guarantee this requirement. Reads that do not see a write through a data race need not ever change their write-seen write after being committed. Reads that see a racy-write are committed seing a happens-before related write, and are immediately made see their final write ($\mathcal{WS}(r)$).
6. Again, this rule has been carefully taken into account in Theorem 35 as explained above, and furthermore, any other read (not yet committed) can see any happens-before write since we do not care at this point about their execution (Theorem C.2).
7. The final condition (7) requires that whenever a read action is committed the write that it sees has to have been committed before. We argue that this is true by means of Theorem 39 for the case when a read sees a write of the same thread. If it sees a write of

a different thread that is related to it by happens-before relation, we argue by means of Theorem 38. Finally, if the two actions form a race, it could be the case that the write is committed when the read is committed, therefore validating the rule, and we are done. Otherwise, if it isn't the only possibility would be that of an early-read commit (since normal reads with placeholders cannot be committed). Finally, if the read is indeed a read early action, we know by the conflict relation $\#_{\blacktriangleleft}$ and the definition of $\overset{\curvearrowleft}{\text{CH}}$ that the read can only be committed if the write that it saw (early) has already been committed (condition (3) of $\#_{\blacktriangleleft}$).

We conclude hence that the procedure to justify legal JMM executions from a Cookbook-High run of Theorem 35 is sufficient concluding our theorem. ◀

▶ **Corollary 41.** *Cookbook-High respects the Data Race Free Guarantee It has been argued elsewhere [5, 12] that the* JMM *satisfies the DRF-guarantee.*