Cooking the Books: Formalizing JMM Implementation Recipes*

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

1 Purdue University, USA
2 Northeastern University, USA

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* [15], 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 [15] 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.2015.445

1 Introduction

A decade ago, the semantics of concurrent Java programs, the Java Memory Model (JMM), was revised and redefined [17]. This revision, which was adopted as part of the official Java specification [13] 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. 445–469
Leibniz International Proceedings in Informatics
LIPICS 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 [19]. In light of these shortcomings, there is currently a community effort to better understand and reconsider the definition of the JMM [12].

A testament to the complexity of the JMM specification is the *The JSR-133 Cookbook for Compiler Writers* [15], 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 [21]. One could attempt to implement this flavor of Java in weaker architectures such as Power [24], 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 [20]? 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 [15]. However, given that [15] 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 [17]. 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 [15].

Perhaps surprisingly, the relation between [17] and [15] has not been considered formally before, and notably our results show that the rules implied by [15] 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 [15] 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 [24]. Moreover, inserting lwsync Power barriers in between the two reads in the reading threads would not prevent this behavior from happening as documented in [24, 8].[2] Unfortunately, lwsync was the barrier of choice recommended by [15] 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., [16, 24]), 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 [15] 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 likely greatly degrade the performance of concurrent Java programs in Power, which is on

[1] Of course, many of the architectures considered in [15] 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.

the one hand unnecessary given the JMM definition, and on the other hand not required by [15]. 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. [15] 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 [15] 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 [15].
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 [15].

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 [15] 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* [17], is that increasing the synchronization of a program cannot *add* new observable behaviors to it. The synchronization operations, formally defined in [17], 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.

Table 1 High-level Roach-Motel Semantics Rules.

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

The roach motel principle implies that all program transformations which increase the *happens-before* [14] 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 [15], 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 [15].

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 [15], 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

Table 2 Low-level Cookbook: Barriers Required.

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

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 [24, 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 [27] 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 [24]) illustrates this issue.

$$\begin{array}{c}
o.f = o'.f = \mathtt{NULL} \\
\hline
\begin{array}{ccccc}
o.f = o' & \parallel & (o.f).f = o & \parallel & r0 = o'.f; \\
 & \parallel & & \parallel & r1 = r0.f
\end{array} \\
\hline
r0 = o \ \& \ r1 = \mathtt{NULL}?
\end{array} \tag{1}$$

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 [17], or rather the improved version of [19].

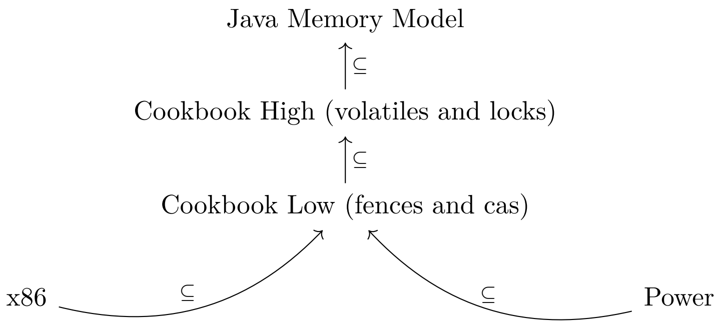
Java Memory Model
⊆
Cookbook High (volatiles and locks)
⊆
Cookbook Low (fences and cas)
x86
⊆
⊆
Power

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 [17]. 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 [17]. 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 [15] 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 [15]. 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 [24, 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 [24, 8].

TSO (x86): This language represents the TSO memory model of x86 processors [21]. 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:

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

$\mathcal{V}ar$ represent variables with substitution semantics, $\mathcal{P}tr$ represent normal pointers, $\mathcal{V}ptr$ represent *volatile* pointers, and $\mathcal{L}ocks$ represent locks.

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 normal references, nor the converse.[7] This assumption will be made precise when presenting

[6] We only consider a subset (namely the ones needed by [15]) of available Power barriers in our development.

[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.

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

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

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.

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 $\texttt{cas}(x)$ is assumed to be a reference, which if it is initially $f\!f$ will be set to tt, returning tt; otherwise, the operation has no effect, and returns $f\!f$.

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 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.

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

Memory Operations

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

Single-Thread Step

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

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

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

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]).

$$\rho \in \mathcal{P}l\mathcal{H}old \qquad \varrho \in \mathcal{P}l\mathcal{H}old\mathcal{P}tr \ ::= \ p \mid \mathsf{p} \mid \rho \qquad \mu \in \mathcal{P}hold\mathcal{V}al \ ::= \ v \mid \rho$$

The set $\mathcal{P}\mathit{lHold}$ 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 $\mathcal{V}\mathit{al}$. 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 placeholder value. Later, the memory system will substitute this placeholder by a concrete value read from memory.

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

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

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 [22]. 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]

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

[9] The treatment of locks can be found in the extended version of this paper [22].

(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:

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

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 [15] (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 [17], 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 [15]. 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 when two actions of the same thread can commute in the temporary store with each other, denoted $(t, \mathsf{mo}) \overset{\frown}{\scriptstyle\text{CH}} (t, \mathsf{mo}')$.

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

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

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	#	#	#	#	#			#		#
$\mathbf{vrd}_{\mathsf{p}}$	#	#	#	#	#	#	#	#	#	#
$\overline{\mathbf{vrd}_{\mathsf{p}}}$	#	#	#	#	#	#	#	#	#	#
$\mathbf{vwr}_{\mathsf{p}}$						#	#	#	#	#
$\mathtt{lk}_\ell$	#	#	#	#	#	#	#	#	#	#
$\mathtt{ul}_\ell$						#	#	#	#	#

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 $\#_{\blacktriangleleft}$:

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

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

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

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

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

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 w.r.t. the read action that originated the value of the conditional; and that write actions (following [24]) 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

(SC) Speculation Commit

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

(VW) Volatile Write

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

(VR) Volatile Read

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

(VRC) Volatile Read Commit

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

(NW) Normal Write

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

(NR) Normal Read

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

(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

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

(ERC) Commit Read Early

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

(FN) Fence

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

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

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 $\varrho \in \mathcal{P}l\mathcal{H}old$ 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:

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

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

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

Thus, the read action $(t', \mathsf{rd}_{\varrho,\rho})$ should not bypass synchronization actions by t' in δ_0 (i.e.,

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

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.

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

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.

JMM Guarantees

The semantics of the JMM [17, 19] 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 [22]) that every execution of cookbook-high corresponds to a legal execution of the JMM as redefined by [19].

▶ **Theorem 1.** *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 [19].*

Proof. (Sketch) Space limitations preclude us from presenting the formal definition of the JMM as presented in [19].[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

[11] The definition and full proof can be found in the appendix of the extended version of this paper [22].

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

1st\2nd	$\mathsf{rd}_{p'}$	$\mathsf{rd}_{\rho'}$	$\overline{\mathsf{rd}}_{\rho'}$	$\mathsf{wr}_{p'}$	$\mathsf{wr}_{\rho'}$
rd_p				$p = p'$	$\#$
rd_ρ				$\#$	$\#$
$\overline{\mathsf{rd}}_\rho$					
$\mathsf{wr}_{p}^{\_,\mathcal{I}}$	$p = p'$	$\#$	$\rho' \in \mathcal{I}$	$p = p'$	$\#$
wr_ρ	$\#$	$\#$	$\#$	$\#$	$\#$

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

$$\mathsf{wr} \ \# \ \langle \mathtt{st} | \_ \rangle \ \& \ \langle \_ | \mathtt{st} \rangle \ \# \ \mathsf{wr}$$

$$b \ \# \ b'$$

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, 19] also hold for us.

▶ **Corollary 2** (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 [19]. ◀

This is a consequence of the theorem above and [19]. 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 [17]. We emphasize that the non-trivial proof of the theorem above can be found in Appendix C of the extended version of the paper [22].

5 Cookbook-Low: Definition, Compilation, Simulation

We now present the intermediate representation of [15] as the cookbook-low language of Figure 2. Our first result is that the semantics of 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, 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 $\mathbf{vrd}$, resp.

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

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

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

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 $\mathcal{S}ync$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.

5.1 Compilation

Following [15] 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 H-L $(\mathsf{mo}, \mathsf{mo}') = b$. Therefore we write 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 the Appendix A of the extended version of this paper [22].

▶ **Theorem 3** (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

$$\begin{array}{rclcrcl} \mathtt{x86}(\langle \mathtt{l}|\mathtt{l}\rangle) & = & \mathtt{skip} & \quad & \mathtt{x86}(\langle \mathsf{s}|\mathsf{s}\rangle) & = & \mathtt{skip} \\ \mathtt{x86}(\langle \mathtt{l}|\mathsf{s}\rangle) & = & \mathtt{skip} & \quad & \mathtt{x86}(\langle \mathsf{s}|\mathtt{l}\rangle) & = & \mathsf{mfence} \end{array}$$

Figure 6 x86 Compilation Strategy.

$$\begin{array}{rclcrcl} \mathtt{PPC}(\langle \mathtt{l}|\mathtt{l}\rangle) & = & \mathsf{sync} & \quad & \mathtt{PPC}(\langle \mathsf{s}|\mathsf{s}\rangle) & = & \mathsf{lwsync} \\ \mathtt{PPC}(\langle \mathtt{l}|\mathsf{s}\rangle) & = & \mathsf{lwsync} & \quad & \mathtt{PPC}(\langle \mathsf{s}|\mathtt{l}\rangle) & = & \mathsf{sync} \end{array}$$

Figure 7 PPC Compilation Strategy.

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 [21, 24].

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 emitted them (in essence modeling the store-buffers of TSO [21]). Finally, we have only one barrier instruction that imposes the following orderings.

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

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.

▶ **Theorem 4** (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.*[12]

[12] The proofs of this section are embedded in the text in the extended version [22].

In combination with Theorem 1 and Theorem 3 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 $\textsc{ppc}$ (i.e. we consider the $\overset{\curvearrowleft}{\textsc{ppc}}$ 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. $\mathsf{rd}_p \;\#_{\textsc{ppc}}\; \mathsf{rd}_p$). 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 [23] and in [24] as cumulativity effects.

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

$$(t, \mathsf{wr}) \;\#_{\textsc{ppc}}\; (t, \mathsf{sync}) \;\#_{\textsc{ppc}}\; (t, \mathsf{wr}) \tag{7}$$

$$(t, \overline{\mathsf{rd}}) \;\#_{\textsc{ppc}}\; (t, \mathsf{sync}) \;\#_{\textsc{ppc}}\; (t, \overline{\mathsf{rd}}) \tag{9}$$

$$(t, \mathsf{rd}) \;\#_{\textsc{ppc}}\; (t, \mathsf{sync}) \;\#_{\textsc{ppc}}\; (t, \mathsf{rd}) \tag{8}$$

$$(t, \mathsf{wr}^{\mathcal{W}\cup\{t'\},\mathcal{I}}) \;\#_{\textsc{ppc}}\; (t', \mathsf{sync}) \tag{10}$$

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 ($\overset{\curvearrowleft}{\textsc{ppc}}$) instead of conflict ($\#_{\textsc{ppc}}$).

$$(t, \mathsf{wr}) \;\#_{\textsc{ppc}}\; (t, \mathsf{lwsync}) \;\#_{\textsc{ppc}}\; (t, \mathsf{wr}) \tag{11}$$

$$(t, \mathsf{rd}) \;\#_{\textsc{ppc}}\; (t, \mathsf{lwsync}) \;\&\; (t, \overline{\mathsf{rd}}) \;\#_{\textsc{ppc}}\; (t, \mathsf{lwsync}) \tag{12}$$

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

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

Notice that lwsync *does* allow the commutativity of $(t, \mathsf{wr}_p) \cdot (t, \mathsf{lwsync}) \overset{\curvearrowleft}{\textsc{ppc}} (t, \mathsf{rd}_{p'})$ 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 [24] (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 [24, 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 [15] 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.

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

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 [17], since volatiles have SC semantics. Moreover, this is a *data-race-free* program that violates the *DRF-guarantee.*

▶ **Theorem 5** (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.*

About Power's lwsync

The semantics of lwsync considered in [24] is slightly weaker that the one considered here. In particular, an example where these two differ is presented in [24] under the name R01, which uses lwsync. Under that weaker interpretation of lwsync, Figure 7 would have to compile $\langle \mathsf{s}|\mathsf{s}\rangle$ 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 5 is evidently true under this modification. This may or may not be considered an error in [15], according to the interpretation of lwsync chosen. According to [24] it is; however, the relaxation that is source of the error has not been observed in practice [24], 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 [24]. Unfortunately [3] has found [24] 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 [12]). 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 [15], which compiles all barriers to the sync-like `dmb` is guaranteed to satisfy our simulations as shown in Theorem 5.

7 Related Work

Our work is closely based on the intuitions provided by [15], whose structure and rules we try to follow as close as possible. We only depart from the informal description of [15] to remediate errors. Similar to [15], [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 [15]. 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 [25, 2] to guarantee SC. We do not consider implementing SC for Java here which would be prohibitively inefficient for architectures like Power; recent work [18, 26] 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 [28, 20]). Since [15] 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 [15]).

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 [15]. 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" [15] 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 JMM Mailing list: Developing the JEP 188: Java Memory Model Update. `http://mail.openjdk.java.net/mailman/listinfo/jmm-dev`, 2014.

13 Java Memory Model and Thread Specification, 2004.

14 Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

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

16 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.

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

18 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.

19 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.

20 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.

21 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.

22 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`.

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

24 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.

25 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.

26 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.

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

28 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.