Plan B: A Buffered Memory Model for Java

Delphine Demange[2] Vincent Laporte[2,1] Lei Zhao[1] Suresh Jagannathan[1] David Pichardie[3,1] Jan Vitek[1]

[1]Purdue University [2]ENS Cachan Bretagne - IRISA [3]INRIA Rennes

Abstract

Recent advances in verification have made it possible to envision trusted implementations of real-world languages. Java with its type-safety and fully specified semantics would appear to be an ideal candidate; yet, the complexity of the translation steps used in production virtual machines have made it a challenging target for verifying compiler technology. One of Java's key innovations, its memory model, poses significant obstacles to such an endeavor. The Java Memory Model is an ambitious attempt at specifying the behavior of multithreaded programs in a portable, hardware agnostic, way. While experts have an intuitive grasp of the properties that the model should enjoy, the specification is complex and not well-suited for integration within a verifying compiler infrastructure. Moreover, the specification is given in an axiomatic style that is distant from the intuitive reordering-based reasonings traditionally used to justify or rule out behaviors, and ill suited to the kind of operational reasoning one would expect to employ in a compiler. This paper takes a step back, and introduces a *Buffered Memory Model* (BMM) for Java. We choose a pragmatic point in the design space sacrificing generality in favor of a model that is fully characterized in terms of the reorderings it allows, amenable to formal reasoning, and which can be efficiently applied to a specific hardware family, namely x86 multiprocessors. Although the BMM restricts the reorderings compilers are allowed to perform, it serves as the key enabling device to achieving a verification pathway from bytecode to machine instructions. Despite its restrictions, we show that it is backwards compatible with the Java Memory Model and that it does not cripple performance.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]

Keywords Concurrency; Java; Memory Model; Verified Compilation

1. Introduction

Formally verified systems are becoming a reality [20, 22]. Recent successes have shown that it is possible to construct reasonably efficient compilers along with a machine-checked proof of correctness. Building on our experience with the Fiji real-time virtual machine [28] and verified compilation for relaxed memory architectures (CompcertTSO) [35], we have embarked on a project to produce a verified platform for a variant of the Java language that abides by the Safety Critical Specification for Java [13]. To build a verifying compiler requires starting from a formal semantics of the source language and, simultaneously, developing and proving the correctness of optimizations and transformations, with an associated formal operational semantics developed for each of the compiler's intermediate representations all the way through to the backend target. Although we expected Java to be a challenge because of the complexity of the transformations used in high-performance virtual machines like Fiji, the Java memory model [24] also complicates the task of devising a tractable formal semantics for the source and intermediate representations.

$$\begin{array}{l||l}
\multicolumn{2}{c}{x \leftarrow 0 \quad y \leftarrow 0 \quad z \leftarrow 0} \\
\hline
r_1 \leftarrow z & r_2 \leftarrow x \\
\texttt{if}\ (r_1 = 1)\ \{x \leftarrow 1; \quad y \leftarrow 1\} & r_3 \leftarrow y \\
\texttt{else} \qquad \{y \leftarrow 1; \quad x \leftarrow 1\} & \texttt{if}\ (r_2 = 1\ \&\ r_3 = 1)\ z \leftarrow 1
\end{array}$$

Figure 1. JMM. x, y and z are shared memory variables, r_i are local. In the JMM, no execution yields $r_1 = r_2 = r_3 = 1$. However, manually reordering the writes to x and y so that they performed in the same order in both branches allows the compiler to eliminate the conditional and hoist the assignments above the store to r_1, thus making such an execution permissible. Alternative definitions of the JMM [5] allow this execution, but the justification is complex, involving subtle notions of speculations and non-local reasoning over execution traces.

The Java Memory Model (JMM) was designed to specify the behavior of concurrent programs across all modern architectures and compiler optimizations currently in use in production virtual machines. The complexity underlying the JMM stems from the need to give a semantics to *all* programs, even racy ones, and the desire to be maximally portable. This degree of generality comes at the price of complexity. When explaining the semantics of racy programs, textbooks [12] elide key JMM notions such as the definition of a legal execution. This issue remains the subject of much active research [4, 5, 11, 16, 23]. Consider Fig. 1 which shows a program where, according to [24], no execution can yield $r_1 = r_2 = r_3 = 1$. But, if the programmer were to manually reorder the instructions in the else-branch, a seemingly insignificant refactoring, this result would be allowed. Such unintuitive behavior is unfortunate. From the verifying compiler writer's point of view, the situation is not much better – it is an open problem to write correctness proofs for a compiler with respect to the JMM's current definition, especially if the target platform enforces its own relaxed (or weak) memory semantics. Indeed, even in the absence of formal proofs, existing Java compilers are not JMM compliant [39]. (And in the process of writing this paper we discovered that our own virtual machine was not JMM compliant.)

Often the JMM is informally discussed in terms of allowed instruction reorderings combined with *sequentially consistent* (SC) interleavings. To illustrate, assume for a moment that reordering of independent, non-volatile, statements was allowed, then we could admit $r_1 = r_2 = 1$ as the value of the program on the left.

$$\begin{array}{r||l}
\multicolumn{2}{c}{x \leftarrow 0 \quad y \leftarrow 0} \\
\hline
r_1 \leftarrow x & r_2 \leftarrow y \\
y \leftarrow 1 & x \leftarrow 1
\end{array}
\quad \xrightarrow[\text{speculative loads}]{\text{reordering with}} \quad
\begin{array}{r||l}
\multicolumn{2}{c}{x \leftarrow 0 \quad y \leftarrow 0} \\
\hline
y \leftarrow 1 & x \leftarrow 1 \\
r_1 \leftarrow x & r_2 \leftarrow y
\end{array}$$

This would be true because after reordering, the result is an SC execution. With this approach, reasoning would be easy – determining the validity of an execution would boil down to considering combinations of permitted reorderings. Unfortunately, the JMM reorderings are complex, and reasoning about transformations is

POPL'13, January 23–25, 2013, Rome, Italy.

rarely intuitive. To rule an execution illegal requires showing that it is not an SC execution if the program is data-race free, or that it includes an *out-of-thin-air* read. Formalizing these notions leads to a complex definition involving race-committing sequences. This is a level of complexity that is challenging for most programmers to tackle. In [5], four test cases from Pugh et al. [30] were found to be flawed, but a later paper [37], using an automatic verification tool, contradicted this interpretation on two of the examples. Such a lack of clarity for programs less than 10 instructions long does not bode well for the viability of the model.

The complexity of the JMM arises in large part by its desire to be portable across all platforms and thus hardware agnostic. But, this is a challenging goal given the vastly different relaxed memory subsystems found on modern architectures, and the range of optimizations modern compilers perform, many of which are often ill-understood. For example, operational definitions of the Power architecture [33] are substantially different from those defined for x86 [36]; the former supports out-of-order unbounded speculative executions and subtle notions of partial coherence, while the latter is expressed in terms of more intuitive store buffers and limited memory reorderings. It is unclear how one might define a tractable formal language memory model that can be effectively tailored to both. Neither is there a clear specification of the optimizations that ought to be supported.

Of course, we could greatly simplify the problem by drastically limiting the behaviors admitted by Java with an easily understood albeit more restrictive model such as sequential consistency. Valid program executions would be limited to those that can be expressed purely in terms of a sequential interleaving of thread actions; but under SC, reordering of shared-memory reads and writes is prohibited. While simple to state and easy to understand, we choose to not follow this approach because SC would likely cripple performance of Java programs on all modern microprocessors due to the large number of fences needed to enforce SC on relaxed hardware; this would be especially true for racy programs implementing sophisticated lock-free algorithms of the kind found, for example, in optimized libraries such as `java.util.concurrent`.

Instead, we propose an alternative memory model that has a tractable semantics and designed to be mapped to x86 multiprocessors that support a Total Store Ordering relaxed memory semantics [32, 36] We introduce the *Buffered Memory Model* (BMM), a memory model for Java that can be fully characterized in terms of the memory reorderings it allows on top of sequentially consistent executions. The BMM comes in two forms. The first is an axiomatic specification that can be used to relate it to the JMM, and which provides an intuitive method to describe valid and invalid program executions. The second form is a fully operational abstract machine with write buffers attached to each thread. This semantics can be readily used by a verifying compiler infrastructure like CompcertTSO. BMM is backwards compatible: existing software that has been written against the JMM can be run directly since the JMM is a superset of the BMM (i.e., every legal BMM execution is legal under the JMM). This ensures that legacy software that has been validated and tested with the JMM in mind will remain correct. The contributions of this paper are thus as follows:

- An axiomatic definition of the BMM, an alternative memory model for Java programs, fully characterized in term of memory event reorderings, and a compliance proof with the JMM.
- A formalization of BMM_o, an operational definition of the semantics of Java programs, equipped with a simple mapping to the Total Store Order (TSO) memory model enabling it easily be used with the x86 family of multiprocessor architectures and with verified compilers like CompcertTSO targeting such platforms.
- A proof that BMM and BMM_o are equivalent. We prove the data race freedom (DRF) theorem for BMM and the legality of reordering optimizations under the BMM.
- An upper bound on the cost of preserving BMM semantics on a production virtual machine running on a TSO hardware, benchmark results measuring the cost and overhead of BMM enforcement, compared to JMM-compliant implementations, and an illustration of the subtle performance implications of the memory model with a case study on the implementation of biased locking.

We note that the design rationale for the BMM is very much dictated by our overarching goal of constructing a verifying compiler for Java. As a result, BMM is not intended to be general-purpose, and does not apply directly to all architectures and compiler optimizations. Instead, we view it as a starting point for further research on verified compilation and memory models.

The BMM, the statements of our results and key lemmas are expressed in Coq; our proofs are rigorous but non-mechanised. The details of both are available online, along with the version of Fiji and LLVM used for our benchmarks: `http://r.cs.purdue.edu/bmm`

2. Related Work

Language Memory Models. Languages like Java and C++ have sophisticated memory models to answer questions related to data visibility and updates for concurrent programs. The JMM does this using committing-sequences, that make it subtle, complex, and unsound [11, 39]. Huisman and Petri [16] proved its DRF guarantee. They tackled the inconsistencies of [24], related to memory initialization by adding hypotheses. Aspinall and Ševčík [5] proposed an alternative definition of the JMM that does not suffer from these issues. They restrict their definition to the finite case, as we do here. Recently, Lochbihler [23] extended the formalization by including infinite executions and dynamic allocations. This work proves type-safety for correctly synchronized programs. Such a mechanized proof is a *tour de force* since it covers a large fragment of Java. However, the proof structure is quite different from the simulation proof we need to perform within the scope of a verified optimizing compiler. There has also been recent work on memory models for C++. Boehm and Adve [7] provide a semantics for data-race free C++ programs, including a semantics for *low-level atomics*. In Java, it is mandatory to also give well-defined semantics to racy programs to avoid security breaches.

Weak Memory Model Formalizations. Work in this area has focussed primarily on characterizing hardware memory models. Early studies [1, 2, 14] outlined a range of hardware memory models, and attempted to rigorously formalize the vendor's documentations. Alglave et al. [3] defined a general framework for formalizing hardware models using partial orders. Operational characterizations have been examined in [27, 32, 33]. Burckhardt *et al.* [10] define an expressive denotational framework where a memory model is a set of dynamic reorderings, aggregation or splitting rewriting rules. TSO boils down to a store-load reordering and a store-load aggregation rule. The BMM follows this line of work by providing provably equivalent axiomatic and operational models that make it intuitive and suitable to verification.

Proofs, Verified Compilation, and Weak Memory Models. Defining multithreaded semantics in terms of reordering is also the approach taken by Miné [26]: the semantics of multithreaded C programs is defined as the interleavings of the programs possibly obtained by the syntactic transformations defining the memory model. From a certified compiler perspective, an operational definition of the JMM is desirable. There are several attempts to provide such a semantics [8, 9, 11, 17] but none of them has been used in a proof assistant. Our operational semantics is inspired by the TSO memory model proposed in [35] which has been formalized in Coq. Ševčík [38] identifies some trace transformations that are valid under the JMM. Transformations are however defined semantically. This gap is filled in [34] where the program transformations are proved to be correct, but this is done under a DRF assumption. [34] also identifies the need for characterizing memory models in terms of the transformations they permit, and this is the goal of the axiomatic formalization of BMM.

3. BMM: Two Models for the Price of One

While the BMM covers the whole of Java, the most challenging parts of reasoning about concurrent programs are those that manipulate shared memory in a potentially racy way. Following the JMM's design, we separate operations that are purely thread-local from those that deal with shared memory. The BMM is parametrized by an abstract notion of intra-thread semantics to deal with the former. We mention a few of the high-level concepts and our design choices before focusing on the memory model proper. (More details about this part of our model is in the online Coq development.)

Java Values. The JVM operates on two kinds of values: primitives and references. Primitives include `long` and `double` values that are 64-bits wide. Updates to these variables is not guaranteed to be atomic on a 32-bit machine. Our intra-thread semantics captures this. The treatment of references is delicate. Several instruction like `instanceof`, `checkcast`, `invokevirtual` require reading the class of an object pointed to by a reference. These operations are not relevant to the memory model because they refer to immutable meta-data. We explicitly avoid reading from shared memory in those cases since that would directly entail reasoning about the memory model. Instead, our semantics attaches type information to references in the spirit of formalizations of the bytecode verifier. As a consequence, the type of an object can be conceptually read without going to shared memory. We also give to each thread its own allocation pool. This matches the notion of field initialization that is advocated in the JMM. We do not need to initialize objects with a default value when we allocate them because each memory address has a well-typed initial value from the very beginning of the program execution.

Final Fields. The JMM provides complex rules for fields annotated `final`. Compilers are allowed to perform aggressive optimizations on reads. The surprising thing is that `final` fields may change between reads. This occurs if the programmer leaks an object during its construction. The result is a very non-intuitive semantics which stands more as a warning for the programmer who does not follow a safe publication pattern than a real semantics. Our position is to forbid unsafe publications [15] (which happens to be a restriction imposed by Safety Critical Java as well). This allows us to keep an intuitive semantics for `final` fields and let compilers aggressively optimize them.

Class Initialization. Lazy class loading is an important mechanism for concurrent programming; see for example the *initialization on demand holder idiom* [29]. Our semantic supports this by tracking the current initialization status of each class.

Synchronization Actions. Their treatment in the paper is slightly simplified. Our formal semantics needs to handle more synchronization actions than those explained here. For example, when spawning a thread, an exception `IllegalThreadStateException` can occur if the thread has already been started. It means that, for a given execution and a given thread, there can be only one successful spawn event but several other failed spawn events. A so-called JMM *synchronizes-with* edge must exist between the successful spawn and all failed spawn events.

3.1 A Reordering-based Memory Model

We propose a memory model that can be *fully* characterized by reorderings over SC executions and that has a simple specification. In the BMM, (1) all SC executions are allowed and, (2) any execution from which an SC execution can be derived by reordering a non-volatile read action with a preceeding non-volatile write action is allowed; both actions must operate over disjoint memory locations and the actions can be separated by a sequence of reads that see the write action. Thus, for instance in the following program

$$\begin{array}{c||c} \multicolumn{2}{c}{x \leftarrow 0 \quad y \leftarrow 0} \\ \hline x \leftarrow 1 & y \leftarrow 1 \\ r_1 \leftarrow y & r_2 \leftarrow x \end{array}$$

$r_1 = r_2 = 0$ is **allowed**

it suffices to reorder one of the threads for the result to be permissible under an SC execution. Fig. 2 gives some prohibited executions. Thanks to this reordering-based definition, all three examples are easily seen to be illegal since no reordering is applicable and no SC execution can exhibit such behaviors.

$$\begin{array}{c||c} \multicolumn{2}{c}{x \leftarrow 0 \quad y \leftarrow 0} \\ \hline x \leftarrow 1 & r_1 \leftarrow y \\ y \leftarrow 1 & r_2 \leftarrow x \end{array}$$

$r_1 = 1, r_2 = 0$ is **illegal**

$$\begin{array}{c||c||c} \multicolumn{3}{c}{x \leftarrow 0 \quad y \leftarrow 0} \\ \hline x \leftarrow 1 & r_1 \leftarrow x & r_2 \leftarrow y \\ & y \leftarrow 1 & r_3 \leftarrow x \end{array}$$

$r_1 = r_2 = 1, r_3 = 0$ is **illegal**

$$\begin{array}{c||c||c||c} \multicolumn{4}{c}{x \leftarrow 0 \quad y \leftarrow 0} \\ \hline x \leftarrow 1 & r_1 \leftarrow x & y \leftarrow 1 & r_3 \leftarrow y \\ & r_2 \leftarrow y & & r_4 \leftarrow x \end{array}$$

$r_2 = r_4 = 0, r_1 = r_3 = 1$ is **illegal**

Figure 2. BMM executions.

3.2 A Buffered Operational Memory Model

While more intuitive for the programmer, the reordering-based definition of BMM lacks an operational form. What we seek with an operational model is a companion proof technique to provide a formal correspondence proof between the different layers of a verified compiler. Operational semantics have been long advocated as a vehicle within which to conduct such proofs by facilitating the use of simulation diagrams. Thanks to their inductive form, they can be used to reason about global program behavior in terms of elementary single steps. Axiomatic models generally do not enjoy the same kind of proof technique. In [39], to prove validity of some program transformations, the authors had to reason on whole prefixes of traces. This uncomfortable situation comes from the fact that in the JMM, a trace of $n + 1$ execution steps is not easily defined in term of its n first steps. At the hardware level, operational semantics have been provided in [33, 36]. It is not surprising to see that no proof connects the JMM with any of these operational models. Thus, our second memory model, called BMM_o, introduces a *store buffer*: each hardware thread effectively has a FIFO buffer of pending memory writes, so that reads performed on different processors can occur before writes have propagated to main memory.

3.3 Reconciling the Two Models

While the axiomatic form of BMM has an intuitive definition and supports a subset of the executions permitted by the JMM, its operational variant has been specified to fit cleanly within the store buffering operations provided by x86-based relaxed memory hardware. The two models are reconciled with an equivalence proof. For this purpose, the BMM reorderings have been carefully chosen to align with the behaviors permitted under BMM_o. These two semantics have different uses. In Sec. 5.2 we use the reordering view to

	SC	C++	JMM	BMM
DRF theorem	✓	✓	✓	✓
Reordering memory accesses	×	✓	✓	⊗
Redundant memory accesses elimination/introduction	✓	✓	⊗	⊗
Operational semantics	✓	✓	×	✓
Semantics for all programs	✓	×	✓	✓
Programmer can understand the semantics of racy programs	✓	×	×	✓
Sound with respect to JMM (does not break legacy Java)	✓	×	✓	✓
Lock optimizations	✓	✓	⊗	⊗

Table 1. Expressivity and properties of memory models. ✓ if a property holds, × if it does not, and ⊗ if there are restrictions. The models differ in the reordering they permit, how they are formalized, the programs they consider, and their support for legacy code. BMM is weaker than the JMM in terms of the reorderings it allows, but its operational semantics is useful for verifying compiler optimizations, and its simpler axiomatic version is easier for programmers to understand. Reordering memory accesses is illegal under original JMM [11, 24] but legal under the alternative version of [39].

provide a DRF theorem. In Sec. 8 we use the operational semantics to study the validity of compiler-driven program transformations. Tab. 1 gives an overview of the properties of the BMM.

4. Background on the Java Memory Model

We introduce key notions from the JMM. A language memory model formally specifies what values can be read by each thread depending upon the writes performed by this thread or others. These interactions are categorized using inter-thread actions.

Inter-thread Actions. The shared memory of a program is split into a set of disjoint *addresses* which are instance fields, static fields or array positions but not local variables. For each address $x \in \mathbb{X}$, we can determine if it is volatile or not with the function $\mathsf{volatile} : \mathbb{X} \to \mathrm{bool}$. In the literature, external actions are distinguished from other memory actions. In this work, we model external actions using volatile writes [1], that can be identified with the function $\mathsf{external} : \mathbb{X} \to \mathrm{bool}$.[2] We assume a set $\mathbb{T}$ of threads, a set $\mathbb{L}$ of locks, and a set $\mathbb{V}$ of values. The set of inter-thread actions is given below, where superscript i denotes the unique identifier of memory actions.

$$
\begin{array}{rcll}
\mathbb{A} & ::= & \mathsf{w}_t^i x, v & \text{(thread } t \text{ writes value } v \text{ to address } x\text{)} \\
 & \mid & \mathsf{r}_t^i x & \text{(thread } t \text{ reads from address } x\text{)} \\
 & \mid & \mathsf{l}_t^i l & \text{(thread } t \text{ acquires a lock on monitor } l\text{)} \\
 & \mid & \mathsf{u}_t^i l & \text{(thread } t \text{ releases a lock on monitor } l\text{)} \\
 & \mid & \mathsf{s}_t t' & \text{(thread } t \text{ creates a new thread } t'\text{)} \\
 & \mid & \mathsf{b}_t & \text{(thread } t \text{ starts)} \\
 & \mid & \mathsf{j}_t^i t' & \text{(thread } t \text{ detects } t' \text{ has terminated)} \\
 & \mid & \mathsf{e}_t & \text{(thread } t \text{ ends)} \\
 & \mid & \mathsf{w}_0 x & \text{(default write action to address } x\text{)}
\end{array}
$$

$$x \in \mathbb{X} \qquad v \in \mathbb{V} \qquad l \in \mathbb{L} \qquad t, t' \in \mathbb{T} \qquad i \in \mathbb{N}$$

Action $\mathsf{w}_0 x$ initializes address x; it has no emitting thread.[3] Thread start (b_t) and end (e_t) can happen only once and thus do not require identifiers. For any action a that is not a default write action, we write $T(a)$ the emitting thread of this action. For any write action w, we write $V(w)$ the value written by that action; initialization actions write a default value according to the type of the related address. We introduce some notations for families of actions:

$$
\begin{array}{rll}
\mathbb{A}_{\mathrm{r}} & = \{\mathsf{r}_t^i x \mid t \in \mathbb{T},\ x \in \mathbb{X}\} & \text{(reads)} \\
\mathbb{A}_{\mathrm{w}} & = \{\mathsf{w}_t^i x, v; \mathsf{w}_0 x \mid t \in \mathbb{T},\ x \in \mathbb{X},\ v \in \mathbb{V}\} & \text{(writes)} \\
\mathbb{A}_{\mathrm{d}} & = \{\mathsf{w}_0 x \mid x \in \mathbb{X}\} & \text{(initializations)} \\
\mathbb{A}_{\mathrm{b}} & = \{\mathsf{b}_t \mid t \in \mathbb{T}\} & \text{(begins)} \\
\mathbb{A}_{\mathrm{s}} & = \{\mathsf{w}_t^i x, v; \mathsf{r}_t^i x \mid t \in \mathbb{T},\ x \in \mathbb{X},\ \mathsf{volatile}(x)\} & \\
 & \quad \cup \{\mathsf{l}_t^i l; \mathsf{u}_t^i l \mid t \in \mathbb{T},\ l \in \mathbb{L}\} & \text{(synchronizations)} \\
 & \quad \cup \{\mathsf{s}_t t'; \mathsf{b}_t; \mathsf{j}_t^i t'; \mathsf{e}_t \mid t, t' \in \mathbb{T}\} & \\
\mathbb{A}_{\mathrm{x}} & = \{\mathsf{w}_t^i x, v \mid t \in \mathbb{T},\ \mathsf{external}(x)\} & \text{(external actions)}
\end{array}
$$

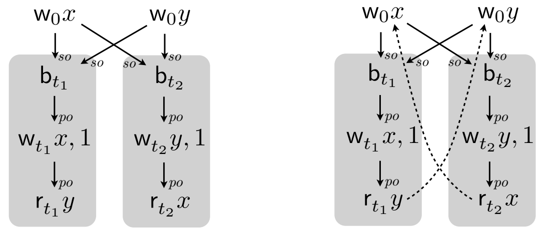
The JMM is based on a *happens-before* model [21]. An execution is described in terms of partial orders between memory actions. The same external behavior may be associated with many different interleavings of thread actions. An interleaving can be seen as a total order on actions: "this action occurs before that one according to global time". Such an interleaving is in fact a consistent extension of a partial order called "happens before" that precisely relates causal dependencies between actions. For example, the program Fig. 3a may exhibit an interleaving of thread-actions

$$\mathsf{b}_{t_1} :: \mathsf{b}_{t_2} :: \mathsf{w}_{t_1} x, 1 :: \mathsf{r}_{t_1} y :: \mathsf{w}_{t_2} y, 1 :: \mathsf{r}_{t_2} x$$

but there is no causal dependency between the read performed in t_1 and the one performed in t_2. Fig. 3b presents the causality relation behind such a linear presentation. Each gray region is dedicated to the actions owned by a same thread.

$$
\begin{array}{c}
x \leftarrow 0 \quad y \leftarrow 0 \\
\hline
\begin{array}{c||c}
x \leftarrow 1 & y \leftarrow 1 \\
r_1 \leftarrow y & r_2 \leftarrow x
\end{array}
\end{array}
$$

(a) Code of threads t_1, t_2.



(b) Causality relation (c) Axiomatic execution

Figure 3. A happens-before executions with and without write-seen arrows.

The unique identifier is omited here. Any sequence of solid arrows between actions a and b means *a should happen before b*. Apart from address initialization and thread starts, few actions are actually constrained in this example. We distinguish two kinds of arrows: $\xrightarrow{po}$ reflects program order between actions of a same thread; $\xrightarrow{so}$ reflects the synchronization relation between events. In more complex examples, it may relate an unlock of a monitor with its subsequent lock or the write of a volatile address with a subsequent read. Dotted arrows indicate which write was seen by each read action. These arrows form what we call an axiomatic execution. The write seen must satisfy some minimal constraints that we will make clear with the notion of well-formed execution.

[1] E.g., a call by a thread t to a function f with arguments args that returns value v is modeled as a volatile write to the abstract location $\mathsf{f}(\mathsf{args})$.

[2] We require that, $\forall x, \mathsf{external}(x) \Rightarrow \mathsf{volatile}(x)$

[3] Unlike [5, 24], we stick to the JMM: every address is virtually given a default value at the start of the program, even if the corresponding location is not allocated yet.

Notations. When a partial order is total we write it directly as a sequence of elements that uniquely characterizes it. When a partial order $\xrightarrow{o}$ is a disjoint union (indexed by $\mathbb{T}$) of orders, we write its restriction on thread t as $[\xrightarrow{o}]_t$. A list of elements can be thought of as a total order. We write $a \xrightarrow{tr} b$ when elements a, b are ordered with regards to a list tr. We call any irreflexive transitive relation an *order*. Two such relations R and R' are said to be *consistent* when they satisfy: $\forall x,\ y,\ \neg(xRy \wedge yR'x)$. We write $tr{\downarrow_A}$ for the sequence tr filtered to the elements of the set A.

DEFINITION 4.1 (Axiomatic execution). *An* execution E *is described by a tuple* $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ *where:*

- P *is a program and* $A \subseteq \mathbb{A} \setminus \mathbb{A}_\mathrm{d}$ *is a set of actions,*
- $\xrightarrow{po} \subseteq A \times A$ *is the* program order*, a disjoint union of total orders on actions of each thread,*
- $\xrightarrow{so} \subseteq (A \cup \mathbb{A}_\mathrm{d}) \times (A \cup \mathbb{A}_\mathrm{d})$ *is the* synchronization order*: the union of a total order on* $A \cap \mathbb{A}_\mathrm{s}$ *of all synchronization actions in A, and the cartesian product* $\mathbb{A}_\mathrm{d} \times (A \cap \mathbb{A}_\mathrm{s})$*,*
- $W \in \mathbb{A}_\mathrm{r} \rightharpoonup \mathbb{A}_\mathrm{w}$ *is a* write-seen *function that maps each read action r from A to a write action w of* $A \cup \mathbb{A}_\mathrm{d}$ *(r and w must operate on the same address).*

We now explain how to extract the happens-before relation from the program order and the synchronization order of an execution.

DEFINITION 4.2 (Synchronizes-with relation). *An action a* synchronizes-with *an action b (written* $a \xrightarrow{sw} b$*) in an execution* $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ *if* $a \xrightarrow{so} b$ *and a, b satisfy one of the following conditions:*

- $a \in \mathbb{A}_\mathrm{d}$ *and* $b \in A \cap \mathbb{A}_\mathrm{b}$ *(default writes synchronizes-with starts),*
- *a is a spawn of a thread t and b is the start of the thread t,*
- *a is a write to a volatile address x and b is a read from x,*
- *a is an unlock on monitor l and b is a lock on monitor l,*
- *a is the end of the thread t and b is a join action on t.*

DEFINITION 4.3 (Happens-before order). *The* happens-before *order of an execution is the transitive closure of the union of its synchronizes-with relation and its program order.*

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

Intra-thread Semantics. Our formalization requires an abstract notion of intra-thread semantic state $\mathrm{State}_\mathrm{intra}$ and an intra-thread labeled transition relation

$$\overset{\cdot}{\twoheadrightarrow} \subseteq \mathrm{State}_\mathrm{intra} \times \mathrm{Label}_\mathrm{intra} \times \mathrm{State}_\mathrm{intra}$$

given to each thread $t \in \mathbb{T}$. Transition labels belong to the set $\mathrm{Label}_\mathrm{intra} = (\mathbb{A} \setminus \mathbb{A}_\mathrm{r}) \cup (\mathbb{A}_\mathrm{r} \times \mathbb{V}) \cup \{\tau\}$: a thread can either take an action step, or a silent step (with label τ) that is memory irrelevant. For a read action step, the value read is paired with the action in the label. The requirements on the intra-thread semantics are:

- $\overset{\cdot}{\twoheadrightarrow}$ only relates states of the same thread,
- there is an *initial* state Ready: no transition leads to it and a thread t steps from it if and only if it emits the b_t action,
- non-silent labels are tagged with the emitting thread,
- there is a *final* state Done: a step of a thread t leads to it if and only if its last transition is labeled by e_t.

DEFINITION 4.4 (Intra-traces). *Let* $tr = a_1 :: \cdots :: a_n$ *be a sequence of actions in set A and let W be a write-seen function on A. Given a thread* $t \in \mathbb{T}$ *in program P, tr is an* intra-trace *of t if there exist* $s_0, s_1, \ldots, s_m \in \mathrm{State}_\mathrm{intra}$ $(m \geq n)$ *and* $l = l_1 :: \cdots :: l_m \in \mathrm{list}(\mathrm{Label}_\mathrm{intra})$ *such that:*

- *for all* $a \in \{a_1, \ldots, a_n\}$*,* $T(a) = t$*,*
- s_0 *is the initial intra-thread state* Ready*,*
- *for all* $i \in \{1, \ldots, m\}$*,* $s_{i-1} \overset{l_i}{\twoheadrightarrow} s_i$*,*
- *the projection* $b_1 :: \cdots :: b_n$ *of l to non-silent labels is such that* $b_i = (a_i, V(W(a_i)))$ *if* a_i *is a read action or* $b_i = a_i$ *otherwise.*

We write $P[t]$ *for the set of such pairs* (tr, W) *for P.*

DEFINITION 4.5 (Well-formed execution). *An execution* $\langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$*, is* well-formed *if*

- *A is finite,*
- $\xrightarrow{so}$ *is consistent with* $\xrightarrow{po}$*,*
- *Locking is proper: for all lock actions* $\mathsf{l}^i_t l \in A$ *and all threads* t' *different from the thread t, the number of lock actions on l emitted by* t' *before* $\mathsf{l}^i_t l$ *in* $\xrightarrow{so}$ *is the same as the number of unlock actions on l emitted by* t' *before* $\mathsf{l}^i_t l$ *in* $\xrightarrow{so}$*, and each unlock action* $\mathsf{u}^i_t l \in A$ *occurs after a matching lock action:*

$$\forall \mathsf{l}^i_t l, \forall t' \neq t, |\{\mathsf{l}^j_{t'} l \mid \mathsf{l}^j_{t'} l \xrightarrow{so} \mathsf{l}^i_t l\}| = |\{\mathsf{u}^j_{t'} l \mid \mathsf{u}^j_{t'} l \xrightarrow{so} \mathsf{l}^i_t l\}|$$
$$\forall \mathsf{u}^i_t l, |\{\mathsf{l}^j_t l \mid \mathsf{l}^j_t l \xrightarrow{po} \mathsf{u}^i_t l\}| > |\{\mathsf{u}^j_t l \mid \mathsf{u}^j_t l \xrightarrow{po} \mathsf{u}^i_t l\}|$$

- $\xrightarrow{po}$ *is intra-thread consistent: for all thread* $t \in \mathbb{T}$*,* $([\xrightarrow{po}]_t, W) \in P[t]$*,*
- $\xrightarrow{so}$ *is consistent with W: for every read r of a volatile address x we have* $W(r) \xrightarrow{so} r$ *and for any write w to x different from* $W(r)$*, either* $w \xrightarrow{so} W(r) \xrightarrow{so} r$ *or* $W(r) \xrightarrow{so} r \xrightarrow{so} w$*,*
- $\xrightarrow{hb}$ *is consistent with W: for all reads r of x,* $r \xrightarrow{hb} W(r)$ *does not hold and there is no intervening write w to x, i.e. such that* $W(r) \xrightarrow{hb} w \xrightarrow{hb} r$*.*

A distinguished subfamily of well-formed executions is the set of Sequentially Consistent axiomatic executions.

DEFINITION 4.6 (Sequentially Consistent (SC) execution). *A well-formed execution* $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ *is* SC *if there exists a total order* $\xrightarrow{to}$ *on A such that*

- $\xrightarrow{to}$ *is consistent with* $\xrightarrow{po}$ *and* $\xrightarrow{so}$*,*
- *for each read action* $r \in A$ *accessing address x,* $W(r)$ *is the last write on x before r in* $\xrightarrow{to}$*.*

The set of well-formed executions of a program forms the *Happens-Before* memory model. It is relatively easy to manipulate but it is not a satisfactory memory model because it allows *out-of-thin-air* values [24] and does not fulfill the DRF theorem. The JMM considers a subset of this model; these are known as *legal* executions. The exact definition of legal executions is subtle. In a nutshell, a well-formed execution E is legal if there exists a sequence $E_0, E_1, \ldots, E_n = E$ of well-formed executions such that in E_0, each read sees a write that it does not race with. Then progressively, each execution E_i allows some reads through data races but in a well-founded order until the execution E itself is reached. Thanks to this definition, one obtains almost directly that: in a DRF program all reads see writes that happen-before them and each execution is sequentially consistent; and no out-of-thin-air value can be read, by the causality order on races.

Whereas the DRF guarantee of the JMM is easily achieved, the complex definition of the race-committing sequence makes it hard to justify whether a given reordering is allowed or not. Our approach is to define the BMM with explicit reorderings, introduced in the next section.

5. BMM: An Axiomatic Memory Model

We now formally define the axiomatic view of our memory model. The semantics is built on top of two simple notions: sequential consistency and reordering of actions. The transformations of executions are expressed with the notion of local reordering.

DEFINITION 5.1 (Local reordering). *Given an execution* $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$, $E' = \langle P', A, \xrightarrow{po'}, \xrightarrow{so}, W\rangle$ *is a local reordering of* E *from an action list* l *to a list* l' *in thread* t *if*

- $[\xrightarrow{po}]_t = \alpha \cdot l \cdot \beta$ *and* $[\xrightarrow{po'}]_t = \alpha \cdot l' \cdot \beta$,
- $[\xrightarrow{po}]_{t'} = [\xrightarrow{po'}]_{t'}$ *for all threads* $t' \neq t$,
- *for all* $(tr, W) \in P'[t]$ *where* tr *is of the form* $\alpha \cdot l' \cdot \beta$*, there exists* $(\alpha \cdot l \cdot \beta, W) \in P[t]$,
- $P[t'] = P'[t']$ *for all thread* $t' \neq t$,
- l *and* l' *contain the same set of actions,*
- *no element of* l *or* l' *is a synchronization action.*

Such a reordering is written $E \xrightarrow{t:[l \longrightarrow l']} E'$.

Intuitively, we reorder the intra-thread trace $[\xrightarrow{po}]_t$ by transforming l into l'. BMM exposes two reorderings to the programmer. The first one is a *Write-Read* reordering which reorders a read before a previous adjacent write to a different address. Here is a simple example:

$$x{\leftarrow}1;\ r{\leftarrow}y \quad \xrightarrow{\text{WR}} \quad r{\leftarrow}y;\ x{\leftarrow}1$$

DEFINITION 5.2 (Write-Read reordering). *A* Write-Read *reordering,* $E \xrightarrow{\text{WR}} E'$*, of an execution* $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ *with respect to actions* w *and* r *in* t*, is a local reordering* E' *such that*

$$E \xrightarrow{t:[w::r \longrightarrow r::w]} E'$$

where w *and* r *operate on different addresses.*

Fig. 4 illustrates the use of the Write-Read reordering on a classical litmus test program. To understand the BMM semantics, a key observation must be made: the execution on the left is not sequentially consistent but after two WR reorderings we obtain a sequentially consistent execution. It is then tempting to ask if a BMM execution is any execution that can be transformed into an SC execution after some WR reorderings. Unfortunately, such a definition would not allow us to capture executions exhibited by TSO-hardware, so we need to work a bit harder.

The program on the top-left part of Fig. 5 illustrates this issue. In this program, the configuration $r_1 = 1, r_2 = 0, r_3 = 1; r_4 = 1, r_5 = 0$ is reachable under a TSO architecture, but it is not a SC execution and no WR reordering can be applied to this program. We introduce a second category of reorderings that is allowed in BMM that permits such executions.

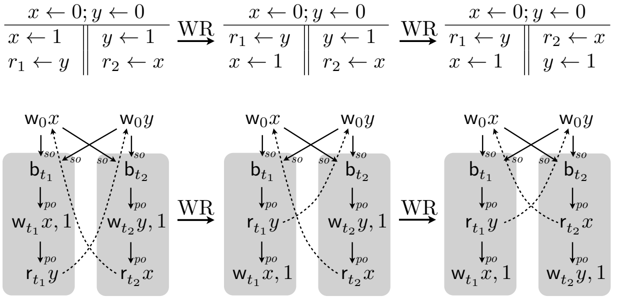


Figure 4. Write-Read reordering example

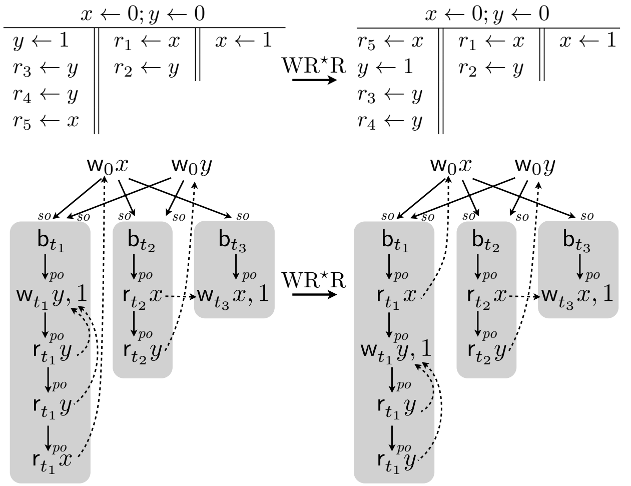


Figure 5. Write-Read-Read reordering example

DEFINITION 5.3 (Write-Read-Read reordering). *A* Write-Read-Read *reordering of* $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ *w.r.t. a tuple of action* $(w, \vec{r}, r')$ *of* A*, is a local reordering* E' *such that*

$$E \xrightarrow{t:[w::\vec{r}::r' \longrightarrow r'::w::\vec{r}]} E'$$

Reads in $\vec{r} = r_1, \ldots, r_n$ *have* w *as write-seen, and* r' *and* w *target different addresses. Such a reordering is written* $E \xrightarrow{\text{WR}^\star\text{R}} E'$.

In Fig. 5, we apply this transformations to the previous program. This time, a reordering is possible and leads to an SC execution. Note that $\text{WR}^\star\text{R}$ is a generalization of the previous WR reordering. We prove in Section 7 that this new reordering exactly captures a TSO operational semantics. Formally, a BMM execution is any execution that can be transformed using $\text{WR}^\star\text{R}$ reorderings until reaching an SC execution.

DEFINITION 5.4 (BMM executions). BMM *executions are:*

$$\text{BMM} = \left\{ E \mid \exists E', E \xrightarrow{RO} E' \text{ and } E' \text{is SC} \right\}$$

with $\xrightarrow{RO} = (\xrightarrow{\text{WR}^\star\text{R}})^*$.

We write BMM(P) for the set of executions of a program P. The BMM observable behaviors of a program P is then defined as the set of sequences of external actions:

$$Obs(P) = \left\{ \xrightarrow{so}\downarrow_{\mathbb{A}_{\mathsf{x}}} \mid \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle \in \text{BMM}(P) \right\}$$

The definition of BMM is a least post-fixpoint. Every time we must prove that BMM is included in some set of well-formed executions, we can rely on the following characterization.

LEMMA 5.1 (BMM least post-fixpoint characterisation). BMM *is the least set* S *of well-formed executions such that*

- *all* SC *executions are in* S,
- S *is backward-closed by* BMM *reorderings: for any well-formed executions* E, E' *such that* $E \xrightarrow{RO} E'$*, if* $E' \in S$ *then* $E \in S$.

Proof. First, BMM satisfies the two above properties. Now, let S be a set of well-formed executions satisfying both properties. We show that $\mathsf{BMM} \subseteq S$. Let $E \in \mathsf{BMM}$. If E is SC, then $E \in S$. Otherwise, there exists an SC execution E' such that $E \xrightarrow{RO} E'$. But E' is in S, so E is in S too. □

We use this lemma to show that BMM is a subset of the JMM executions and to show the equivalence between BMM and $\mathsf{BMM}_\circ$.

5.1 BMM is a Subset of JMM

The current Java Memory Model defines the set of legal executions as a subset of all well-formed executions that are justifiable using a sequence of intermediate justifications. In order to connect our model with the JMM, rather than unfolding the details of this formal definition, we rely on the following JMM properties:

- JMM accepts all sequentially consistent executions,
- JMM allows reordering of non-volatile memory accesses hitting different locations [39].

THEOREM 5.2. *Let* JMM *be the set of all legal executions permitted by the* JMM*. Then,* $\mathsf{BMM} \subseteq \mathsf{JMM}$.

Proof. We use here Lemma 5.1. We first know that JMM contains all SC executions. Then, suppose that $E \xrightarrow{RO} E'$ with $E' \in \mathsf{JMM}$. In the JMM, reordering non volatile memory accesses hitting different addresses is allowed [39]. We use this property to un-transform E' into E. Hence, E is also in JMM, meaning that JMM is backward-closed by $\mathrm{WR}^\star\mathrm{R}$. □

5.2 DRF Guarantee

We establish that BMM enjoys the important property that any reasonable memory model should have, namely a *data-race-free guarantee* - data-race free programs have SC executions only. We first define the standard notions of concurrent conflicting memory action, data-race, and data-race-free programs:

DEFINITION 5.5 (Conflicting actions). *Two non-volatile actions* $a, b \in \mathbb{A}_\mathrm{r} \cup \mathbb{A}_\mathrm{w}$ *are* conflicting *if they target the same address and* $T(a) \neq T(b)$ *and at least one of them is a write.*

DEFINITION 5.6 (Data-races). *Let* $\langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ *be a* BMM SC *execution. Two conflicting actions* a, b *form a* data-race *if they are not ordered by* $\xrightarrow{hb}$*.*

DEFINITION 5.7 (DRF). *A program* P *is* data-race free*, written* $DRF(P)$*, if all of its* SC *executions are free of data-race.*

THEOREM 5.3 (DRF guarantee). *For all* P*, if* $DRF(P)$*, then for all execution* $E \in \mathsf{BMM}(P)$*,* E *is* SC.

Proof. Let P be such that $DRF(P)$, and $E \in \mathsf{BMM}(P)$. By Theorem 5.2, $E \in \mathsf{JMM}(P)$. But JMM satisfies the DRF guarantee [38], so E is sequentially consistent. □

6. $\mathsf{BMM}_\circ$: An Operational Memory Model

In this section, we provide an operational view of the BMM. The reorderings allowed in its axiomatic version can be implemented by a $\mathsf{BMM}_\circ$ machine that attaches a write-buffer to each running thread. The $\mathsf{BMM}_\circ$ semantics is also parametrized by an intra-thread semantics. Hence, we need to consider an extra set of actions: the silent actions in $\mathbb{A}_\mathrm{sil}$ that are either the unbuffering $\overline{\mathrm{B}}(a)$ of a write action $a \in \mathbb{A}_\mathrm{w} \setminus \mathbb{A}_\mathrm{d}$ by thread $T(a)$ or a silent step τ_t by thread t.

$$\mathbb{A}_\mathrm{sil} ::= \overline{\mathrm{B}}(a) \mid \tau_t$$

The idea behind $\mathsf{BMM}_\circ$ is to provide a generative, operational machine that executes an input operational execution, modifying a memory state, made of thread buffers and a shared memory. The input of the $\mathsf{BMM}_\circ$ machine is an operational execution, made of a program and a trace of *operational actions*. An operational action $a \in \mathbb{A}_\mathrm{op}$ is either an action in $\mathbb{A} \setminus (\mathbb{A}_\mathrm{d} \cup \mathbb{A}_\mathrm{r})$, or a pair in $\mathbb{A}_\mathrm{r} \times \mathbb{A}_\mathrm{w}$ (for each read action we record the write action that it sees, and refer to it as its *write-seen*), or a silent action in $\mathbb{A}_\mathrm{sil}$.

$$\frac{ts(t) \xrightarrow{\tau}\!\!\!\!\rightarrow s}{ts, b, m \xrightarrow{\tau_t} ts[t \mapsto s], b, m}\,[\textsc{Tau}]$$

$$\frac{ts(t) \xrightarrow{\mathsf{r}^i_t x | V(w)}\!\!\!\!\rightarrow s \quad w = \mathrm{rd}_t(b, m, x) \quad \neg\mathsf{volatile}(x)}{ts, b, m \xrightarrow{\mathsf{r}^i_t x | w} ts[t \mapsto s], b, m}\,[\textsc{Read}]$$

$$\frac{ts(t) \xrightarrow{\mathsf{w}^i_t x, v}\!\!\!\!\rightarrow s \quad \neg\mathsf{volatile}(x)}{ts, b, m \xrightarrow{\mathsf{w}^i_t x, v} ts[t \mapsto s], b[t \mapsto (\mathsf{w}^i_t x, v) :: b(t)], m}\,[\textsc{Write}]$$

$$\frac{b(t) = l \cdot [\mathsf{w}^i_t x, v]}{ts, b, m \xrightarrow{\overline{\mathrm{B}}(\mathsf{w}^i_t x, v)} ts, b[t \mapsto l], m[x \mapsto (\mathsf{w}^i_t x, v)]}\,[\textsc{UnBuff}]$$

$$\frac{ts, m \xrightarrow{\lambda}_{\mathsf{synch}} ts', m' \quad b(t) = []}{ts, b, m \xrightarrow{\lambda} ts', b, m'}\,[\textsc{Synch}]$$

Figure 6. $\mathsf{BMM}_\circ$ machine (labelled transition system).

DEFINITION 6.1 (Operational execution). *An operational execution is a pair* (P, tr) *where* P *is a program and* $tr \in \mathrm{list}(\mathbb{A}_\mathrm{op})$ *is finite and such that no action appear more than once in* tr*.*

The $\mathsf{BMM}_\circ$ machine is then defined by a transition system, parametrized by an intra-thread semantics. We now describe its states and transitions. A $\mathsf{BMM}_\circ$ state $\in \mathrm{State}$ is a record

$$\begin{array}{rcll} ts & \in & \mathbb{T} \rightharpoonup \mathrm{State}_\mathrm{intra}; & \text{(intra-thread state of threads)} \\ b & \in & \mathbb{T} \rightharpoonup \mathrm{list}(\mathbb{A}_\mathrm{w} \setminus \mathbb{A}_\mathrm{d}); & \text{(one buffer per thread)} \\ m & \in & \mathbb{X} \to \mathbb{A}_\mathrm{w} & \text{(one write action per address)} \end{array}$$

The state first keeps track of each intra-thread state in $\mathrm{State}_\mathrm{intra}$. Each thread is given a write buffer; all non-volatile write actions are first written to this buffer. When unbuffered, these writes are committed to the shared memory m, that maps addresses to write actions. Given a memory state (buffers b and memory m), the $\mathsf{BMM}_\circ$ machine specifies the write action a thread t can read when accessing the address x:

$$\mathrm{rd}_t(b, m, x) = \begin{cases} w & \text{if } w \text{ is the first write to } x \text{ in } b(t) \\ m(x) & \text{if there is no write to } x \text{ in } b(t) \end{cases}$$

If a pending write to this address appears in the buffer of t, we take the most recent in the execution (*i.e.* the first in the buffer). Otherwise we consult the memory. If no write has been performed yet at this address we will retrieve the default value for this address.

The $\mathsf{BMM}_\circ$ machine is defined as a labeled transition system where steps are labeled by operational actions. The salient semantic rules are given Fig. 6. In all rules (except BUFF) the $\mathsf{BMM}_\circ$ machine makes a step that the intra-thread semantics can match. Rule TAU corresponds to a intra-thread silent step (when e.g. the thread operates on registers). The $\mathsf{BMM}_\circ$ memory state does not change in this case. On a non-volatile read (READ) the value is obtained from the memory state using the rd function. For this action, the intra-thread event is a pair $\mathsf{r}^i_t x \mid v$ composed of a read and a value. The intra-thread semantics accepts any value here but the purpose of the rule is to constrain it using thread-local buffers and shared memory. On a non-volatile write (WRITE), the write action is put onto the thread's buffer. A write action can be unbuffered at any time, in which case the write is committed into shared memory (UNBUFF).

$$\dfrac{ts(t) \overset{\mathsf{r}^i_t x | V(w)}{\twoheadrightarrow} s \quad w = m(x) \quad \mathsf{volatile}(x)}{ts, m \xrightarrow{\mathsf{r}^i_t x | w}_{\mathsf{synch}} ts[t \mapsto s], m}\,[\textsc{ReadV}] \qquad \dfrac{ts(t) \overset{\mathsf{w}^i_t x, v}{\twoheadrightarrow} s \quad \mathsf{volatile}(x)}{ts, m \xrightarrow{\mathsf{w}^i_t x, v}_{\mathsf{synch}} ts[t \mapsto s], m[x \mapsto (\mathsf{w}^i_t x, v)]}\,[\textsc{WriteV}]$$

$$\dfrac{ts(t) \overset{\mathsf{b}_t}{\twoheadrightarrow} s}{ts, m \xrightarrow{\mathsf{b}_t}_{\mathsf{synch}} ts[t \mapsto s], m}\,[\textsc{Begin}] \qquad \dfrac{ts(t) \overset{\mathsf{s}_t t'}{\twoheadrightarrow} s \quad t' \notin \mathrm{dom}(ts)}{ts, m \xrightarrow{\mathsf{s}_t t'}_{\mathsf{synch}} ts[t \mapsto s][t' \mapsto \mathrm{Ready}], m}\,[\textsc{Spawn}] \qquad \dfrac{ts(t) \overset{\mathsf{l}^i_t l}{\twoheadrightarrow} s}{ts, m \xrightarrow{\mathsf{l}^i_t l}_{\mathsf{synch}} ts[t \mapsto s], m}\,[\textsc{Lock}]$$

$$\dfrac{ts(t) \overset{\mathsf{e}_t}{\twoheadrightarrow} s}{ts, m \xrightarrow{\mathsf{e}_t}_{\mathsf{synch}} ts[t \mapsto s], m}\,[\textsc{End}] \qquad \dfrac{ts(t') = \mathrm{Done} \quad ts(t) \overset{\mathsf{j}^i_t t'}{\twoheadrightarrow} s}{ts, m \xrightarrow{\mathsf{j}^i_t t'}_{\mathsf{synch}} ts[t \mapsto s], m}\,[\textsc{Join}] \qquad \dfrac{ts(t) \overset{\mathsf{u}^i_t l}{\twoheadrightarrow} s}{ts, m \xrightarrow{\mathsf{u}^i_t l}_{\mathsf{synch}} ts[t \mapsto s], m}\,[\textsc{Unlock}]$$

Figure 7. $\mathsf{BMM}_\circ$ machine (synchronization actions).

All synchronizing actions are emitted by threads whose buffers are empty. They are gathered under the SYNCH rule whose definition (relying on a relation $\xrightarrow{\cdot}_{\mathsf{synch}}$) is in Fig. 7. Reads from and writes to volatile locations directly access the memory so that all threads have a consistent view (READV and WRITEV). When a thread ends (END), its state is kept in the $\mathsf{BMM}_\circ$ state, enabling other threads to join it (JOIN), and preventing it to from being restarted (SPAWN).

We then define a $\mathsf{BMM}_\circ$ execution as a constrained operational execution that is accepted by the $\mathsf{BMM}_\circ$ machine: the input trace is properly locked and can be executed by the machine, with the intended meaning that the input execution is intra-thread consistent.

DEFINITION 6.2 ($\mathsf{BMM}_\circ$ execution). *An operational execution (P, tr) is in $\mathsf{BMM}_\circ(P)$ if there exists states $s_0, s_1, \ldots, s_n$ in* State *satisfying the following:*

- *tr is properly locked (see Definition 4.5, using $\xrightarrow{tr}$ instead of $\xrightarrow{po}$ and $\xrightarrow{so}$),*
- *s_0 is an initial state: the memory maps every address to the corresponding default write ($\forall x, m(x) = \mathsf{w}_0 x$), buffers are empty and $s_0.ts$ is defined for exactly one thread, mapping it to the* Ready *state,*
- *$tr = a_1 :: \cdots :: a_n \in \mathrm{list}(\mathbb{A}_{\mathrm{op}})$,*
- *for all $i \in \{1, \ldots, n\}$, $s_{i-1} \xrightarrow{a_i} s_i$.*

The $\mathsf{BMM}_\circ$ behaviors of program P are external action traces obtained by projecting all executions of P accepted by the $\mathsf{BMM}_\circ$ machine on $\mathbb{A}_{\mathrm{x}}$:

$$Obs_\circ(P) = \{tr{\downarrow_{\mathbb{A}_{\mathrm{x}}}} \mid (P, tr) \in \mathsf{BMM}_\circ(P)\}$$

7. Equivalence of BMM and $\mathsf{BMM}_\circ$

We show that BMM and $\mathsf{BMM}_\circ$ are equivalent relaxed memory models: they allow the exact same set of behaviors for any program.

THEOREM 7.1. *For all program P, $\quad Obs_\circ(P) = Obs(P)$.*

The proof relies on an operator ρ that bridges the gap between BMM and $\mathsf{BMM}_\circ$ by building axiomatic executions from operational ones.

DEFINITION 7.1 (ρ operator). *Let $E_\circ = (P, tr)$ be an execution, ρ is defined as $\rho(E_\circ) = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ where*

- *A is the set of non-silent actions in tr,*
- *for all $a, b \in A$, $a \xrightarrow{po} b$ iff $T(a) = T(b)$ and $a \xrightarrow{tr} b$,*
- *for all $a, b \in A$, $a \xrightarrow{so} b$ iff $a, b \in \mathbb{A}_{\mathrm{s}}$ and $a \xrightarrow{tr} b$,*
- *for all pairs $\mathsf{r}^i_t x \mid w$ in tr, $W(\mathsf{r}^i_t x) = w$.*

We show that $\rho(\mathsf{BMM}_\circ) = \mathsf{BMM}$. Each inclusion is proved and Theorem 7.1 follows from the following lemma.

LEMMA 7.2. *Let $E_\circ = (P, tr)$ be an operational execution and $\langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle = \rho(E_\circ)$, then $tr{\downarrow_{\mathbb{A}_{\mathrm{x}}}} = (\xrightarrow{so}{\downarrow_{\mathbb{A}_{\mathrm{x}}}})$.*

Proof. By definition of ρ, we know that for all $a, b \in A$, $a \xrightarrow{so} b$ iff $a, b \in \mathbb{A}_{\mathrm{s}}$ and $a \xrightarrow{tr} b$. □

We define auxiliary notions needed for the proof. An operational execution $E_\circ$ is well-formed if $\rho(E_\circ)$ is well-formed, and it is SC_ρ if $\rho(E_\circ)$ is SC. Similarly, we define operational reorderings relying on ρ: an execution $E'_\circ$ is an operational reordering of $E_\circ$ if $\rho(E_\circ) \xrightarrow{\mathrm{WR}^\star\mathrm{R}} \rho(E'_\circ)$. We abuse notations and write it $E_\circ \xrightarrow{\mathrm{WR}^\star\mathrm{R}} E'_\circ$, and lift this notion to trace reorderings $\xrightarrow{RO}$. To lighten the notations in this section, we keep implicit the unique identifier of actions and write w^t_x for $\mathsf{w}^i_t x, v$ (the value written is omitted). When considering operational actions in a trace we generally omit the write action w attached to a read action.

7.1 $\rho(\mathsf{BMM}_\circ) \subseteq \mathsf{BMM}$

We prove that every $\mathsf{BMM}_\circ$ execution trace can be reordered into a SC trace. The proof relies on a reordering scheme explained in the following lemma.

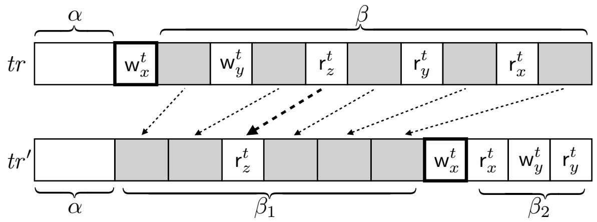
LEMMA 7.3. *Let $E_\circ = (P, tr) \in \mathsf{BMM}_\circ(P)$, with $tr = \alpha \cdot [\mathsf{w}^t_x] \cdot \beta$ an execution such that $\overline{\mathrm{B}}(\mathsf{w}^t_x) \notin \beta$. We write*

- *$\mathcal{W}^t = \{\mathsf{w}^t_y \in \beta \mid y \in \mathbb{X}\}$, write actions belonging to thread t,*
- *$\mathcal{R}^t = \{\mathsf{r}^t_y \in \beta \mid \mathsf{w}^t_x \cdot \beta = \gamma_1 \cdot \mathsf{w}^t_y \cdot \gamma_2 \cdot \mathsf{r}^t_y \cdot \gamma_3,\ y \in \mathbb{X},\ \gamma_1, \gamma_2, \gamma_3 \in \mathrm{list}(\mathbb{A}_{\mathrm{op}})\}$, read actions seeing a write performed by t in $[\mathsf{w}^t_x] \cdot \beta$,*
- *$\beta \setminus (\mathcal{W}^t \cup \mathcal{R}^t)$ the remaining actions in β.*

Then, there exist P', β_1, β_2 such that $E'_\circ = (P', tr') \in \mathsf{BMM}_\circ(P')$, $E_\circ \xrightarrow{RO} E'_\circ$ and $tr' = \alpha \cdot \beta_1 \cdot [\mathsf{w}^t_x] \cdot \beta_2$ where,

- *$\beta_1 = \beta{\downarrow_{\beta \setminus (\mathcal{W}^t \cup \mathcal{R}^t)}}$,*
- *β_2 contains the elements of $\mathcal{W}^t \cup \mathcal{R}^t$,*
- *$[\mathsf{w}^t_x] \cdot \beta_2$ matches the pattern $(\mathsf{w}^t_{x_1}; (\mathsf{r}^t_{x_1})^*) \cdot \ldots \cdot (\mathsf{w}^t_{x_n}; (\mathsf{r}^t_{x_n})^*)$,*
- *for all δ, $(P, tr \cdot \delta) \in \mathsf{BMM}_\circ(P)$ then $(P', tr' \cdot \delta) \in \mathsf{BMM}_\circ(P')$.*

Before going into the detail of the proof, we give an intuition on how we use it. This lemma is applied to a part of an execution in which a write action, performed by t, stays in its buffer. This is illustrated in the following figure, where the grey regions denote subsequences of actions whose owning thread is not t. The bold stroke action w^t_x is the write action that remains in t's buffer until the end of tr. We will use this action as a pivot on all the actions performed in the rest of tr, so that the resulting trace tr' is as illustrated: (a) all grey actions are shifted before the pivot, remaining in the same relative order, by changing the interleaving; (b) actions of thread t are handled with the $\mathrm{WR}^\star\mathrm{R}$ reordering rule. Because write actions of t cannot be moved, they are kept to the right of the pivot.



Handling read actions of t is more involved: either they see a write that occurred (necessarily in thread t) after the pivot in tr, and they are accumulated in a pattern $(\mathsf{w}_{x_1}^t; (\mathsf{r}_{x_1}^t)^*) \cdot \ldots \cdot (\mathsf{w}_{x_n}^t; (\mathsf{r}_{x_n}^t)^*)$, or they see a write that occurs before the pivot, and $\mathrm{WR}^\star\mathrm{R}$ is applied repeatedly on this pattern, until they are swapped before w_x^t.

Proof. Let $E_\circ = (P, tr) \in \mathsf{BMM}_\circ(P)$, with $tr = (\alpha \cdot [\mathsf{w}_x^t] \cdot \beta)$ and $\overline{\mathrm{B}}(\mathsf{w}_x^t) \notin \beta$. The sub-trace $[\mathsf{w}_x^t] \cdot \beta$ can be decomposed as $[\mathsf{w}_x^t] \cdot \beta = \beta_p \cdot \beta_t$ with $\beta_p = [\mathsf{w}_y^t; (\mathsf{r}_y^t)^*]^+$ (we take the shortest β_t). We now proceed by induction on the length of β_t.

- Base case: β_t is empty. We don't need any reordering transformation to reach the expected pattern.
- Inductive case. We proceed by case analysis on the first element of $\beta_t = a :: \beta'_t$ and show that a can be either (i) integrated inside the pattern β_p or (ii) be moved before this pattern:
  - If $a = \mathsf{r}_k^t$, when k is one of the addresses of β_p. Here, we integrate in β_p by applying an $\mathrm{WR}^\star\mathrm{R}$ as many times as needed to make it part of the right $\mathsf{w}_k^t(\mathsf{r}_k^t)^*$ pattern such that $\beta_p = \beta_{p_1} \cdot (\mathsf{w}_k^t(\mathsf{r}_k^t)^*) \cdot \beta_{p_2}$. Note $\mathrm{WR}^\star\mathrm{R}$ can be applied because the pattern only concerns thread t, and the visibility conditions required by $\mathrm{WR}^\star\mathrm{R}$ are fulfilled. The right-most such pattern before a, i.e. such that there is no w_k^t in β_{p_2}, is the only one that can ensure the write-seen of a to be preserved, according to $\mathsf{BMM}_\circ$. Thus the resulting trace $\left(P', \alpha \cdot \beta_{p_1} \cdot (\mathsf{w}_k^t(\mathsf{r}_k^t)^* a) \cdot \beta_{p_2} \cdot \beta'_t\right)$ is $\mathsf{BMM}_\circ$.
  - If $a = \mathsf{r}_k^t$, when k is not any of the addresses of β_p. We apply the same reasoning as in the previous case, rewriting the trace with $\mathrm{WR}^\star\mathrm{R}$, but in this case, this simply amounts to put a before w_x^t, because $\mathrm{WR}^\star\mathrm{R}$ will be applied on the whole β_p (and the write-seen is trivially kept valid).
  - If $a = \overline{\mathrm{B}}(a')$, then $a' \in \alpha$ because there is no unbuffering action in β_p. This unbuffering could have been done just before w_x^t. This unbuffering does not modify the visibility constraints of the new trace, as it cannot overwrite any of the write actions in β_p. For any trace δ, if $(P, \alpha \cdot \beta_p \cdot [a] \cdot \beta'_t \cdot \delta) \in \mathsf{BMM}_\circ(P)$ then the trace $(P, \alpha \cdot [a] \cdot \beta_p \cdot \beta'_t \cdot \delta)$ is also in $\mathsf{BMM}_\circ$ (P): any read in δ that sees a' in the first trace can still see it in the second trace because no write action in β_p is unbuffered before δ.
  - For all other cases ($a = \mathsf{r}_k^{t''}$, where $t'' \neq t$, $a = \mathsf{w}_k^{t''}$, with $t'' \neq t$, and $a \in \mathbb{A}_\mathrm{s}$), we can just change the interleaving to move a before β_p and conclude easily.

  For each resulting trace, we show that we are under the induction hypothesis premises. Hence, we conclude by induction.

□

This reordering scheme is extensively used in the proof of Lemma 7.4, stating that the reordering interpretation of BMM can be simulated in the operational world.

LEMMA 7.4. *Let $E_\circ = (P, tr) \in \mathsf{BMM}_\circ(P)$. Then there exist P', tr' such that $E_\circ \xrightarrow{RO} (P', tr')$, with $(P', tr') \in \mathsf{BMM}_\circ(P')$ is SC_ρ.*

In order to prove the lemma by induction, we need to prove a stronger result, that includes an additional property on the shape of the two operational traces tr and tr'. This so-called *write-swapping* property is formally defined as follows.

DEFINITION 7.2 (Write-swapping). *Let tr and tr' two sequences of actions. The* write-swapping *property holds on (tr, tr') if for any write actions w_1, w_2 to the same address,*

- *if $w_1 \xrightarrow{tr} w_2$ and $w_2 \xrightarrow{tr'} w_1$ (write actions have been swapped) then the trace tr is of the form $tr = \alpha \cdot [w_1] \cdot \beta \cdot [w_2] \cdot \gamma$ and $\overline{\mathrm{B}}(w_1) \notin \beta$.*
- *if $w_1 \xrightarrow{tr} w_2$ and $w_1 \xrightarrow{tr'} w_2$ (write actions have not been swapped) then if $\overline{\mathrm{B}}(w_1)$ occurs before w_2 in tr, it is also the case in tr'.*

Proof. We prove Lemma 7.4 by strong induction on the size of the execution $|tr| = n$. Assume the property holds for any integer $k < n$. Let $E_\circ = (P, tr) \in \mathsf{BMM}_\circ(P)$ an execution of size n.

We assume $n > 0$ (the case $n = 0$ holds trivially). So tr is of the form $tr = tr_1 \cdot [a]$. By induction on tr_1, we get P_2 and tr_2 such that $(P, tr_1) \xrightarrow{RO} (P_2, tr_2)$, with $E_\circ^2 = (P_2, tr_2) \in \mathsf{BMM}_\circ(P_2) \cap \mathrm{SC}_\rho$, plus a write-swapping on (tr_1, tr_2).

We extend $E_\circ^2$ to $(P_2, tr_2 \cdot [a])$. If action a is not a read action, we can conclude directly. Otherwise, $a = \mathsf{r}_x^{t_3}$, the extended trace is in $\mathsf{BMM}_\circ(P_2)$, and the write-swapping property holds. It remains to show that it is SC_ρ. If it is not, we proceed by case analysis:

Case 1: There is a thread $t \neq t_3$ whose buffer is not empty at the end of tr_2. Formally, there is a write action w_y^t in tr_2 such that $\overline{\mathrm{B}}(\mathsf{w}_y^t) \notin tr_2$. $tr_2.[a]$ is of the form $\alpha \cdot [\mathsf{w}_y^t] \cdot \beta \cdot [\mathsf{r}_x^{t_3}]$. Applying Lemma 7.3 on w_y^t, we get P_3 and tr_3 such that $(P_2, tr_2.[a]) \xrightarrow{RO} (P_3, tr_3)$ with $E_\circ^3 = (P_3, tr_3) \in \mathsf{BMM}_\circ(P_3)$ and $tr_3 = \alpha \cdot \beta_1 \cdot [\mathsf{r}_x^{t_3}] \cdot [\mathsf{w}_y^t] \cdot \beta_2$ with $[\mathsf{w}_y^t] \cdot \beta_2$ matching the pattern $[\mathsf{w}_\cdot^t; (\mathsf{r}_\cdot^t)^*]^+$. By induction on $\alpha \cdot \beta_1 \cdot [\mathsf{r}_x^{t_3}]$, we get P_4 and tr_4 such that $\left(P_3, \alpha \cdot \beta_1 \cdot [\mathsf{r}_x^{t_3}]\right) \xrightarrow{RO} (P_4, tr_4)$, with $E_\circ^4 = (P_4, tr_4) \in \mathsf{BMM}_\circ(P_4) \cap \mathrm{SC}_\rho$, plus a write-swapping property on the traces. We concatenate the suffix $[\mathsf{w}_y^t] \cdot \beta_2$ to extend tr_4 to an execution in $\mathsf{BMM}_\circ(P_4) \cap \mathrm{SC}_\rho$. The sequential consistency holds thanks to the pattern of $[\mathsf{w}_y^t] \cdot \beta_2$. The write-swapping is preserved by the concatenation.

Case 2: All threads distinct from t_3 have their buffer empty at the end of tr_2. Formally, for every write action $\mathsf{w}_y^t \in tr_2$ such that $t \neq t_3$, $\overline{\mathrm{B}}(\mathsf{w}_y^t) \in tr_2$. Let $\mathsf{w}_x^{t_1}$ be the write seen by $\mathsf{r}_x^{t_3}$.

- If $\overline{\mathrm{B}}(\mathsf{w}_x^{t_1}) \notin tr_2$, then it means that $t_1 = t_3$. The trace $tr_2.[a]$ is of the form: $\alpha \cdot [\mathsf{w}_x^{t_1}] \cdot \beta \cdot [\mathsf{r}_x^{t_3}]$. We apply Lemma 7.3 on $\mathsf{w}_x^{t_1}$. The trace $\alpha \cdot \beta_1 \cdot [\mathsf{w}_x^{t_1}] \cdot \beta_2$ we obtain is in $\mathsf{BMM}_\circ$ and looking at the shape of β_1 and β_2 it is even in SC_ρ. We must show that $\mathsf{w}_x^{t_1}$ is now the most recent write to x in β_2. But any other write there would be from thread t_1 and $\mathsf{r}_x^{t_3}$ could thus not see $\mathsf{w}_x^{t_1}$.
- If $\overline{\mathrm{B}}(\mathsf{w}_x^{t_1}) \in tr_2$, then the trace $tr_2.[a]$ is of the form $\alpha \cdot [\mathsf{w}_x^{t_1}] \cdot \beta \cdot [\overline{\mathrm{B}}(\mathsf{w}_x^{t_1})] \cdot \gamma \cdot [\mathsf{r}_x^{t_3}]$. No $\mathsf{w}_x^{t_2}$ more recent than $\mathsf{w}_x^{t_1}$ can appear in γ: either it is unbuffered in γ and it would overwrite $\mathsf{w}_x^{t_1}$ or it is not unbuffered but then $t_2 = t_3$ and $\mathsf{w}_x^{t_1}$ could not be seen by $\mathsf{r}_x^{t_3}$. No unbuffering $\overline{\mathrm{B}}(\mathsf{w}_x^{t_2})$ can either appear in γ since it would overwrite $\mathsf{w}_x^{t_1}$.

  By Lemma 7.3 on $\mathsf{w}_x^{t_1}$ and β, we get a trace tr_3 such that $tr_3 \cdot [\overline{\mathrm{B}}(\mathsf{w}_x^{t_1})] \cdot \gamma \cdot [\mathsf{r}_x^{t_3}] = \alpha \cdot \beta_1 \cdot [\mathsf{w}_x^{t_1}] \cdot \beta_2 \cdot [\overline{\mathrm{B}}(\mathsf{w}_x^{t_1})] \cdot \gamma \cdot [\mathsf{r}_x^{t_3}]$. In this trace, the most recent write to x is now $\mathsf{w}_x^{t_1}$. The subtrace $\alpha \cdot \beta_1 \cdot [\mathsf{w}_x^{t_1}] \cdot \beta_2 \cdot [\overline{\mathrm{B}}(\mathsf{w}_x^{t_1})]$ is SC because $\alpha \cdot [\mathsf{w}_x^{t_1}] \cdot \beta \cdot [\overline{\mathrm{B}}(\mathsf{w}_x^{t_1})]$ was. It remains to show that all reads in γ still see the most recent writes. By induction on $tr_3 \cdot [\overline{\mathrm{B}}(\mathsf{w}_x^{t_1})] \cdot \gamma$, an SC execution is rebuilt, that keeps the same write-seen.

□

Finally, we obtain the first inclusion as a corollary of Lemma 7.4.

COROLLARY 7.5. $\rho(\mathsf{BMM}_\circ) \subseteq \mathsf{BMM}$.

Proof. Let $E_\circ = (P, tr) \in \mathsf{BMM}_\circ(P)$. By Lemma 7.4, we have $E_\circ \xrightarrow{RO} E'_\circ$, with $E'_\circ = (P', tr') \in \mathsf{BMM}_\circ(P')$ is SC_ρ. By definition, $\rho(E_\circ) \xrightarrow{RO} \rho(E'_\circ)$ and $\rho(E'_\circ)$ is SC. Hence, $\rho(E_\circ) \in \mathsf{BMM}(P)$. □

7.2 $\mathsf{BMM} \subseteq \rho(\mathsf{BMM}_\circ)$

We use here the post-fixpoint characterization of BMM (Lemma 5.1). We first show that $\rho(\mathsf{BMM}_\circ)$ contains all SC axiomatic executions. Let $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle \in \mathsf{BMM}$ be an SC execution. Then there exists a total order $\xrightarrow{to}$ on A, compatible with $\xrightarrow{po}$ and $\xrightarrow{so}$ such that all read actions in A see the last write to their address w.r.t. $\xrightarrow{to}$. We claim that $E_\circ = (P, tr)$ can be build, with $tr\downarrow_{\mathbb{A}} = \xrightarrow{to}$, and that $E_\circ \in \mathsf{BMM}_\circ(P)$. Silent actions are inserted in $\xrightarrow{to}$ so that each write action is immediately unbuffered, and that tr is intra-thread consistent – an equivalent condition was required for $E \in \mathsf{BMM}(P)$. Finally $\rho(E_\circ) = E$. We also show that $\rho(\mathsf{BMM}_\circ)$ is backward-closed by $\mathrm{WR}^\star\mathrm{R}$. Let E and E' two well-formed axiomatic executions such that $E' \in \rho(\mathsf{BMM}_\circ)$ and $E \xrightarrow{\mathrm{WR}^\star\mathrm{R}} E'$. $E' \in \rho(\mathsf{BMM}_\circ)$ so there exists $E'_\circ \in \mathsf{BMM}_\circ$ such that $E' = \rho(E'_\circ)$. $\mathrm{WR}^\star\mathrm{R}$ is a valid transformation under $\mathsf{BMM}_\circ$ (see Section 8.1), meaning that there exists $E_\circ \in \mathsf{BMM}_\circ$ such that $\rho(E_\circ) = E$. Hence, $E \in \rho(\mathsf{BMM}_\circ)$.

8. Validity of Transformations

One of the objectives of any memory model is to take into account the reorderings performed by the hardware and to allow compilers to perform some program transformations that deal directly with memory accesses or locks. Tab. 2 gives standard transformations and their validity under various memory models [38, 39].

For a proof of validity we rely on the operational model: we consider a $\mathsf{BMM}_\circ$ trace of a transformed program and show there exists a valid $\mathsf{BMM}_\circ$ trace of the original program with the same behavior. For a proof of invalidity, we provide a counter-example and use the intuitive reordering memory model of BMM: given a program P and a transformed program P', we show that there exists an execution that is valid for P' but invalid for P (both under BMM). This table demonstrate that, despite its restricted set of reorderings, BMM allows useful transformations.

8.1 Validity of WR and $\mathrm{WR}^\star\mathrm{R}$

Among the set of transformations, the validity of the local reordering $\mathrm{WR}^\star\mathrm{R}$ is crucial for the memory model inclusion.

DEFINITION 8.1 (Valid reordering). *A local reordering $\xrightarrow{\Phi}$ between axiomatic executions is said to be valid with respect to* $\mathsf{BMM}_\circ$ *if for all axiomatic execution E and all operational execution $E_\circ$, $E \xrightarrow{\Phi} \rho(E_\circ)$ and $E_\circ \in \mathsf{BMM}_\circ$ implies that there exists $E'_\circ \in \mathsf{BMM}_\circ$ such that $E = \rho(E'_\circ)$.*

We can show that both WR and $\mathrm{WR}^\star\mathrm{R}$ are valid.

LEMMA 8.1. $\xrightarrow{\mathrm{WR}}$ *is valid.*

Proof. Let $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ be an axiomatic execution and $E_\circ = (P', tr')$ an operational execution with $E \xrightarrow{\mathrm{WR}} \rho(E_\circ)$ and $E_\circ \in \mathsf{BMM}_\circ$. By hypothesis, we know that $tr' = \beta \cdot [\mathsf{r}^j_t x] \cdot \gamma \cdot [\mathsf{w}^i_t y, v] \cdot \delta$ and γ does not contain any action owned by t except some unbuffering actions. The write action $\mathsf{w}^i_t y, v$ can be performed just after $\mathsf{r}^j_t x$ since the unbuffering in γ are independent of it and no read action in γ can see $\mathsf{w}^i_t y, v$ (it is still in its buffer). Hence the trace $\beta \cdot [\mathsf{r}^j_t x] \cdot [\mathsf{w}^i_t y, v] \cdot \gamma \cdot \delta$ is still in $\mathsf{BMM}_\circ$ for the same value-seen and write-seen information. After a swap we obtain a trace $tr'' = \beta \cdot [\mathsf{w}^i_t y, v] \cdot [\mathsf{r}^j_t x] \cdot \gamma \cdot \delta$ that belongs to $\mathsf{BMM}_\circ(P)$ (by definition of $\xrightarrow{\mathrm{WR}}$) and $\rho(P, tr'') = E$. □

Transformation	SC	JMM	BMM
Trace preserving transformation	✓	✓	✓
Reordering normal memory accesses	×	✓	⊗
Redundant read after read elimination	✓	×	✓
Redundant read after write elimination	✓	✓	✓
Irrelevant read elimination	✓	✓	✓
Irrelevant read introduction	✓	×	✓
Redundant write before write elimination	✓	✓	✓
Redundant write after read elimination	✓	×	×
Roach motel reordering	✓	×	⊗

We write ✓ for a valid transformation and × when it is generally wrong, ⊗ for limited applicability: only $\mathrm{WR}^\star\mathrm{R}$ applies to normal memory accesses; a read can be delayed past a lock and a write can take over an unlock.

Table 2. Validity of transformations in memory models.

LEMMA 8.2. $\xrightarrow{\mathrm{WR}^\star\mathrm{R}}$ *is valid.*

Proof. Let $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ be an axiomatic execution and let $E_\circ = (P', tr')$ be an operational execution such that $E \xrightarrow{\mathrm{WR}^\star\mathrm{R}} \rho(E_\circ)$ and $E_\circ \in \mathsf{BMM}_\circ$. By hypothesis, the traces tr' is of the form $tr' = \beta\cdot[\mathsf{r}^j_t x]\cdot\gamma\cdot[\mathsf{w}^i_t y, v]\cdot\gamma_1\cdot[\mathsf{r}^{i_1}_t y]\cdots\gamma_n\cdot[\mathsf{r}^{i_n}_t y]\cdot\delta$. Each $\gamma, \gamma_1, \ldots, \gamma_n$ does not contain any action owned by t except some unbuffering actions. As in the previous proof, the action $\mathsf{w}^i_t y, v$ can be performed just after $\mathsf{r}^j_t x$. All read actions $\mathsf{r}^{i_1}_t y, \ldots, \mathsf{r}^{i_n}_t y$ see the write $\mathsf{w}^i_t y, v$ in tr' and can thus also be performed earlier. The trace $\beta\cdot[\mathsf{r}^j_t x]\cdot[\mathsf{w}^i_t y, v]\cdot[\mathsf{r}^{i_1}_t y]\cdots[\mathsf{r}^{i_n}_t y]\cdot\gamma\cdot\gamma_1\cdots\gamma_n\cdot\delta$ is still in $\mathsf{BMM}_\circ(P')$ for the same value-seen and write-seen (the moved reads see $\mathsf{w}^i_t y, v$ directly from t's buffer). We then conclude with a swap: the trace $tr'' = \beta \cdot [\mathsf{w}^i_t y, v] \cdot [\mathsf{r}^{i_1}_t y] \cdots [\mathsf{r}^{i_n}_t y] \cdot [\mathsf{r}^j_t x] \cdot \gamma \cdot \gamma_1 \cdots \gamma_n \cdot \delta$ belongs to $\mathsf{BMM}_\circ(P)$ and $\rho(P, tr'') = E$. □

8.2 Other Valid Transformations

Given $E' = (P', tr')$, for each transformation we observe the shape of tr' and provide a corresponding $\mathsf{BMM}_\circ$ trace of an untransformed program. We assume that the intra-thread semantics accepts the transformation. All read and write actions are non-volatile.

Redundant Read after Read Elimination.

$$tr' = \beta \cdot [\mathsf{r}^i_t x] \cdot \gamma$$

Trace $\beta \cdot [\mathsf{r}^i_t x] \cdot [\mathsf{r}^j_t x] \cdot \gamma$ is in $\mathsf{BMM}_\circ$. $\mathsf{r}^j_t x$ and $\mathsf{r}^i_t x$ see the same write.

Redundant Read after Write Elimination.

$$tr' = \beta \cdot [\mathsf{w}^i_t x, v] \cdot \gamma$$

Trace $\beta \cdot [\mathsf{w}^i_t x, v] \cdot [\mathsf{r}^j_t x] \cdot \gamma$ is in $\mathsf{BMM}_\circ$. $\mathsf{r}^j_t x$ sees $\mathsf{w}^i_t x, v$ in its buffer.

Irrelevant Read Elimination.

$$tr' = \beta \cdot \delta$$

Trace $\beta \cdot [\mathsf{r}^i_t x] \cdot \gamma$ is in $\mathsf{BMM}_\circ$. $\mathsf{r}^i_t x$ sees the first write to x in the buffer or pick the current write attached to x in memory.

Irrelevant Read Introduction.

$$tr' = \beta \cdot [\mathsf{r}^i_t x] \cdot \delta$$

Trace $\beta \cdot \gamma$ is in $\mathsf{BMM}_\circ$. The visibility of the other actions is preserved.

Redundant Write before Write Elimination.

$$tr' = \beta \cdot [\mathsf{w}_t^j y, v] \cdot \gamma$$

We distinguish two cases. If $\gamma = \gamma_1 \cdot \overline{\mathrm{B}}(\mathsf{w}_t^j y, v) \cdot \gamma_2$. The trace $\beta \cdot [\mathsf{w}_t^i y, v'] \cdot [\mathsf{w}_t^j y, v] \cdot \gamma_1 \cdot \overline{\mathrm{B}}(\mathsf{w}_t^i y, v') \cdot \overline{\mathrm{B}}(\mathsf{w}_t^j y, v) \cdot \gamma_2$ is in $\mathsf{BMM}_\circ$. Adding the unbuffering preserves the visibility of $\mathsf{w}_t^j y, v$ since $\mathsf{w}_t^i y, v'$ is never seen. Otherwise, $\overline{\mathrm{B}}(\mathsf{w}_t^j y, v) \notin \gamma$. The trace $tr' = \beta \cdot [\mathsf{w}_t^i y, v'] \cdot [\mathsf{w}_t^j y, v] \cdot \gamma$ is in $\mathsf{BMM}_\circ$. $\mathsf{w}_t^i y, v'$ is never seen.

Roach-motel: Read-lock Reordering. A read before a lock can be reordered (no action emitted by the thread t in γ).

$$tr' = \beta \cdot [\mathsf{l}_t^i l] \cdot \gamma \cdot [\mathsf{r}_t^j x] \cdot \delta$$

We assume tr' is properly locked so no synchronization action on l is in γ. Hence the following interleaving is a valid trace of the transformed program: $\beta \cdot \gamma \cdot [\mathsf{l}_t^i l] \cdot [\mathsf{r}_t^j x] \cdot \delta$. Then the trace $\beta \cdot \gamma \cdot [\mathsf{r}_t^j x] \cdot [\mathsf{l}_t^i l] \cdot \delta$ is in $\mathsf{BMM}_\circ$.

Roach-motel: Unlock-write Reordering. A write after an unlock can be reordered. Since the buffer must be empty, the transformed trace has the following shape, where $\mathsf{w}_t^i x, v$ is the last non silent action of thread t.

$$tr' = \beta \cdot [\mathsf{w}_t^i x, v] \cdot \gamma \cdot [\overline{\mathrm{B}}(\mathsf{w}_t^i x, v)] \cdot \delta \cdot [\mathsf{u}_t^j l] \cdot \eta$$

There is no action emitted by thread t in γ except some unbufferings. No action in γ emitted by the other threads can see the write $\mathsf{w}_t^i x, v$. Then, changing the scheduling, we can obtain the following $\mathsf{BMM}_\circ$ traces: $\beta \cdot \gamma \cdot [\mathsf{w}_t^i x, v] \cdot [\overline{\mathrm{B}}(\mathsf{w}_t^i x, v)] \cdot \delta \cdot [\mathsf{u}_t^j l] \cdot \eta$. By hypothesis, there is no action emitted by the thread t in δ; besides there is no lock/unlock action on l in δ. Therefore changing the scheduling again can lead to the following $\mathsf{BMM}_\circ$ (properly locked) trace: $\beta \cdot \gamma \cdot [\mathsf{w}_t^i x, v] \cdot [\overline{\mathrm{B}}(\mathsf{w}_t^i x, v)] \cdot [\mathsf{u}_t^j l] \cdot \delta \cdot \eta$. Finally, the following trace is a $\mathsf{BMM}_\circ$ execution of the untransformed program: $\beta \cdot \gamma \cdot [\mathsf{u}_t^j l] \cdot [\mathsf{w}_t^i x, v] \cdot [\overline{\mathrm{B}}(\mathsf{w}_t^i x, v)] \cdot \delta \cdot \eta$.

8.3 Invalid Transformations

The BMM helps understand why some transformations are invalid.

Redundant Write after Read Elimination. Here v is volatile.

$$\begin{array}{c} x\leftarrow 0 \quad y\leftarrow 0 \\ \hline \begin{array}{c||l} r_1\leftarrow x & x\leftarrow 1 \\ y\leftarrow 1 & v\leftarrow 0 \\ x\leftarrow r_1 & r_2\leftarrow y \\ r_3\leftarrow x & \end{array} \\ r_1=r_2=0, r_3=1 \text{ invalid} \end{array} \xrightarrow[\text{after read elim.}]{\text{redund. write}} \begin{array}{c} x\leftarrow 0 \quad y\leftarrow 0 \\ \hline \begin{array}{c||l} r_1\leftarrow x & x\leftarrow 1 \\ y\leftarrow 1 & v\leftarrow 0 \\ & r_2\leftarrow y \\ r_3\leftarrow x & \end{array} \\ r_1=r_2=0, r_3=1 \text{ valid} \end{array}$$

On the left, we see why the execution is not valid: it isn't SC and no reordering is possible. After the redundant write elimination, the execution becomes valid because the read $r_3\leftarrow x$ can be reordered with the write $y\leftarrow 1$ to give a SC execution. This transformation thus introduced new behaviors.

Roach-motel: Unlock-read Reordering. Reordering reads and unlocks is not allowed. Here, the reads cannot both see the default writes, whereas they could if one of them was hoisted into the critical section and reordered with the preceeding write.

$$\begin{array}{c} x \leftarrow 0 \quad y \leftarrow 0 \\ \hline \begin{array}{l||l} \mathsf{lock}\ l & \mathsf{lock}\ l' \\ x \leftarrow 1 & y \leftarrow 1 \\ \mathsf{unlock}\ l & \mathsf{unlock}\ l' \\ r_1\leftarrow y & r_2\leftarrow x \end{array} \end{array}$$

Roach-motel: Write-lock Reordering. Reordering writes and locks is not allowed. The reads cannot both see the default writes, whereas they could if one of the writes was hoisted into the critical section and reordered with the next read.

$$\begin{array}{c} x \leftarrow 0 \quad y \leftarrow 0 \\ \hline \begin{array}{l||l} x \leftarrow 1 & y \leftarrow 1 \\ \mathsf{lock}\ l & \mathsf{lock}\ l' \\ r_1\leftarrow y & r_2\leftarrow x \\ \mathsf{unlock}\ l & \mathsf{unlock}\ l' \end{array} \end{array}$$

9. Empirical Evaluation of the BMM

We have shown that the BMM is more restrictive than the JMM. It is, therefore, natural to ask how severe these restrictions are in practice, i.e. what is the performance impact imposed by BMM when incorporated within a production virtual machine running on a TSO architecture. In this section, we present results from a preliminary study that provides a coarse upper-bound approximation of the overheads incurred by BMM conformance. Our experiment is as follows. Starting with a production virtual machine, we switch the backend from a non-verifying optimizing compiler to one that preserves TSO semantics. Then, we modify any Java-level optimizations performed by the VM to be BMM compliant. We expect that the performance results are going to be an upper bound on the costs of BMM, since we limited the optimizations performed by the VM and ensured that they precisely enforce the reorderings allowed by BMM. This enforcement is realized through the injection of *memory fences*, operations that effectively flush the contents of store buffers; we make no attempt to optimize or verify the placement of such fences [40].

Specifically, we start with the Fiji open-source real-time JVM [28]. We selected Fiji because it has competitive performance, we understand the optimizations it performs well, and it is representative of a real-world system. The Fiji compiler takes bytecode as input and, in the configuration we use here, transforms it into ANSI C code, which is fed to `gcc`. The Fiji compiler includes a variety of classic optimizations as well as Java-specific techniques. To achieve BMM compliance in the back-end, we replace `gcc` with $\mathrm{LLVM}_{\mathsf{TSO}}$, an LLVM branch with optimizations either modified or disabled to preserve compliance with the TSO memory model [25]. This is a close approximation of what we would write to support the BMM in the backend. Within Fiji, we carefully examined all optimization passes to ensure compliance with BMM; redundant code elimination (RCE) is the only optimization for which compliance could not be guaranteed; RCE is performed over local operations as well as heap loads; as a result, the optimization permits write-read or read-read operations to be reordered, violating BMM semantics. We modified the optimization to disallow processing any heap loads.

We evaluated the modified system on a variety of benchmarks, including SPECjvm98, SPECjbb2000, and a subset of DaCapo2006 (2006-10-MR2 Release) and DaCapo2009 (9.12 Bach Release). The whole benchmark suite is a mix of concurrent and sequential programs, which we think are well suited to exhibit the throughput overheads of missed optimization opportunities and superfluous fences. (The concurrent programs we consider are `avora9`, `jbb2000`, `lusearch6`, and `lusearch9`). The experiments were run on an 8-Core Xeon 3.16 GHz processors and 8 GB memory machine installed with Fedora 13 Linux (kernel 2.6.34). SPECjvm98 was executed for 15 iterations with the first 5 iterations being warm-up. We report the mean of the last 10 iterations. The standard deviation was negligible. The SPECjbb2000 and DaCapo benchmarks were all executed with their default configurations, except that the workload was set to the maximum wherever possible and that the number of warehouses (for SPECjbb2000) and the number of threads (for DaCapo2009) were all set to 8 to exercise the maximum concurrency.

We evaluate how BMM impacts performance by examining performance relative to a system that uses an unmodified version of Fiji and LLVM; in this baseline configuration, there are no optimizations that are either modified or disabled within either Fiji or LLVM; specifically, the baseline implementation allows Fiji to perform RCE transformations, and LLVM to perform non-TSO compliant reorderings. The results are shown in Fig. 8 with column BMM. The numbers are normalized with respect to this baseline. First, observe that adding TSO support to LLVM and crippling RCE has almost no impact on performance on most of benchmarks with the exception of `jack` and `avrora9`; for the former, BMM is

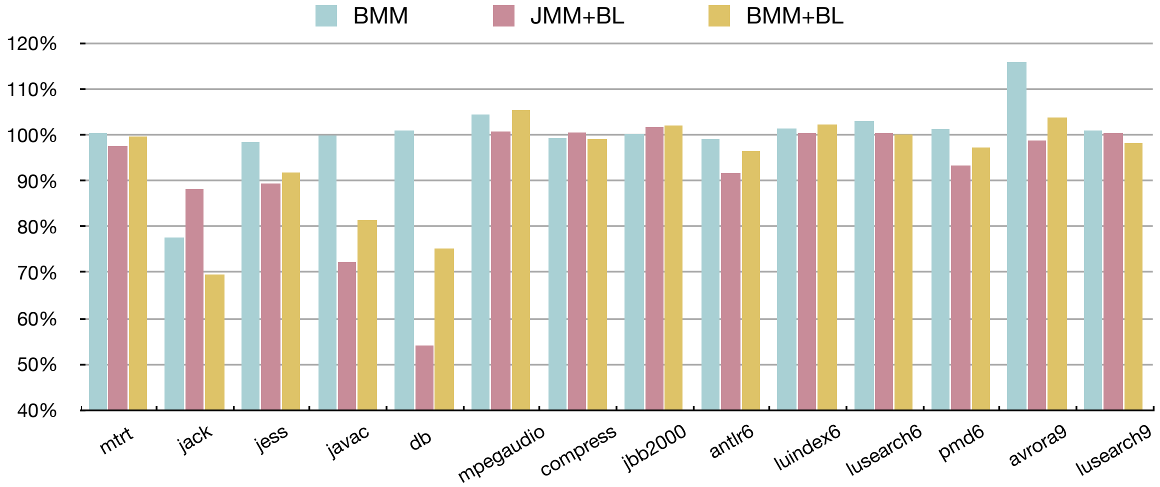


Figure 8. Fiji VM. Execution time normalized to JMM configuration. See Tab. 3 for configurations.

Name	LLVM	RCE	Biased-locking
JMM (baseline)	regular	regular	off
BMM	tso	modified	off
JMM+BL	regular	regular	on with no fence
BMM+BL	tso	modified	on with 2 fences

Table 3. Evaluation configurations

22% faster while 16% slower for the latter. The average slowdown is 1%. In general, these results provide strong anecdotal evidence that BMM is essentially performance neutral. On more relaxed architectures the story is, of course, likely to be different.

Biased Locking. While inspecting the code of the Fiji VM, we stumbled upon an interesting optimization. Fiji has several implementations of the Java concurrency control primitives. One optimization that is supported is called biased locking [19]. Biased locking is an implementation of locking operations that is efficient when one thread repeatedly acquires and releases the same monitor [31], a pattern that is frequent in Java. Roughly, a monitor can be *biased* to one thread that can then acquire and release it without any synchronization. However, when a thread needs a lock that is biased to another thread, that bias needs to be *revoked* using synchronizing instructions. As shown in Fig. 8 under column JMM+BL, turning on biased locking in Fiji results in a maximum speed up of roughly 45% (and an average of 8%). This optimization thus appears to be important to performance.

Fiji biased locking is implemented without any CAS or fences. Although we believe this is what most of modern Java VM do, its correctness is unclear given the current formal definition of JMM. Biased locking implies a roach motel reordering and this kind of reordering has been shown to be invalid by Sevčik [39]. Indeed, according to the “JSR–133 cookbook” and other folklore,[4] *biased locking is permitted under JMM*, but to the best of our knowledge, there is no formal proof on the number of fences required to ensure correctness.

The BMM requires two fences, as we will prove next, one at the lock and one at the unlock. With two fences, the benefits of biased locking is 30% at best (and on average 6%).

Validity of Biased Locking. We explain how to reason about biased locking under the BMM. Since the implementation of the locking mechanism cannot be described with Java constructs only, we introduce two actions available only at a lower-level language in the compilation chain: a *memory fence* f_t and a *fake unlock* $\tilde{\mathsf{u}}_t^i l$. The memory fence can be emitted by a thread if and only if its buffer is empty. The effect of the action is to ensure the buffer is empty prior to any subsequent action being performed.

$$\frac{ts(t) \xrightarrow{\mathsf{f}_t}\!\!\!\twoheadrightarrow s}{ts, m \xrightarrow{\mathsf{f}_t}_{\mathsf{synch}} ts[t \mapsto s], m}\,[\textsc{Fence}]$$

The fake unlock models the bias revocation; this action is taken into account as a regular unlock in what makes a trace properly locked. However, it imposes no restrictions on the state of buffers.

$$\frac{ts(t) \xrightarrow{\tilde{\mathsf{u}}_t^i l}\!\!\!\twoheadrightarrow s}{ts, b, m \xrightarrow{\tilde{\mathsf{u}}_t^i l} ts[t \mapsto s], b, m}\,[\textsc{FakeUnlock}]$$

At this level of abstraction we do not describe *how* the biased locking is to be implemented, but the intuition is that the conjunction of a fence and a fake unlock corresponds to an unlock. The general case of a thread silently reacquiring the same lock can be seen as a trace transformation (there is no $\mathsf{l}_t l$ in β and no $\mathsf{l}_t l$ nor $\mathsf{u}_t l$ in γ):

$$\alpha \cdot [\mathsf{l}_t^i l] \cdot \beta \cdot [\mathsf{u}_t^j l] \cdot \gamma \cdot [\mathsf{l}_t^k l] \cdot \delta \xrightarrow{\text{BL}} \alpha \cdot [\mathsf{l}_t^i l] \cdot \beta \cdot [\mathsf{f}_t] \cdot \gamma \cdot [\mathsf{f}_t] \cdot \delta$$

This transformation is indeed valid: consider a $\mathsf{BMM}_\circ$ execution (P', tr') with $tr' = \alpha \cdot [\mathsf{l}_t^i l] \cdot \beta \cdot [\mathsf{f}_t] \cdot \gamma \cdot [\mathsf{f}_t] \cdot \delta$. If the intra-thread semantics allows to replace the first fence by an action $\mathsf{u}_t^j l$ and the second one by an action $\mathsf{l}_t^k l$, then the untransformed trace $tr = \alpha \cdot [\mathsf{l}_t^i l] \cdot \beta \cdot [\mathsf{u}_t^j l] \cdot \gamma \cdot [\mathsf{l}_t^k l] \cdot \delta$ is also $\mathsf{BMM}_\circ$: the Fence rule ensures that when the f_t action is emitted the buffer of thread t is empty so the Unlock/Lock rules can be applied. Moreover the trace tr is properly locked. The revocation of a biased lock corresponds to the following trace transformation.

$$\alpha \cdot [\mathsf{l}_t^i l] \cdot \beta \cdot [\mathsf{u}_t^j l] \cdot \gamma \cdot \delta \xrightarrow{\text{BL}'} \alpha \cdot [\mathsf{l}_t^i l] \cdot \beta \cdot [\mathsf{f}_t] \cdot \gamma \cdot [\tilde{\mathsf{u}}_t^{j'} l] \cdot \delta$$

Since there is no lock on monitor l in γ, if the transformed trace is properly locked, then so is the initial one. The action f_t ensures t's buffer is empty after β. The Unlock rule can thus be applied.

[4] https://blogs.oracle.com/dave/entry/biased_locking_in_hotspot

10. Conclusions

This work presents BMM, a memory model that has been designed as part of a broader undertaking to build a verifying compiler for multithreaded safety-critical Java. The key characteristics required for such a memory model are ease-of-understanding both for programmers and compiler writers, and a practical realization within a compiler framework that does not impose onerous restrictions on important program optimizations. We believe the BMM is a promising step in this direction. Its axiomatic definition is expressed using intuitive and simple memory reordering notions, making it suitable for reasoning about program transformations, while its operational instantiation can conveniently serve as a basis for a verifying compiler infrastructure. Its backward compatibility with the JMM entails that we can use BMM on legacy code. It thus provides a key missing piece for a verified infrastructure for Java on the x86 processor family.

The question of how to obtain a more relaxed model, one that would allow efficient implementations on architectures such as Power and ARM, while at the same time remaining amenable to incorporation within verified compilers, remains a subject for future research. Although these more relaxed platforms still guarantee coherence (i.e., there is a single linear order of writes that all threads must respect), they allow other non-intuitive behaviors such as out-of-order writes, speculative execution, etc., as well as having a substantially more complex notion of dependence; these complexities make formalization and axiomatic reasoning challenging [33]. We believe that more experience with BMM, with respect to both applications and optimizations, is necessary before we can transplant the intuitions underlying our development here to these other environments.

Acknowledgements. We would like to thank Hans Boehm, Cliff Click, Doug Lea, Gustavo Petri, Filip Pizlo, Jaroslav Ševčík and Peter Sewell for their useful feedback on this work. We thank Dan Marino for his help with LLVM$_{\mathsf{TSO}}$.

References

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: a Tutorial. *Computer*, 29(12), 1996.

[2] S. V. Adve and M. Hill. A Unified Formalization of Four Shared-Memory Models. *Trans. on Par. and Distributed Systems*, 4(6), 1993.

[3] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in Weak Memory Models. In *Proc. of CAV*, 2010.

[4] D. Aspinall and J. Ševčík. Java Memory Model Examples: Good, Bad and Ugly. In *Proc. of VAMP*, 2007.

[5] D. Aspinall and J. Ševčík. Formalising Java's Data Race Free Guarantee. In *Proc. of TPHOLs*, 2007.

[6] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ Concurrency. In *Proc. of POPL*, 2011.

[7] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. *SIGPLAN Not.*, 43, 2008.

[8] G. Boudol and G. Petri. Relaxed Memory Models: an Operational Approach. In *Proc. of POPL*, 2009.

[9] G. Boudol and G. Petri. A Theory of Speculative Computation. In *Proc. of ESOP*, 2010.

[10] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying Local Transformations on Relaxed Memory Models. In *Proc. of CC*, 2010.

[11] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java Memory Model: Operationally, Denotationally, Axiomatically. In *Proc. of ESOP*, 2007.

[12] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[13] T. Henties, J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for Safety-Critical Applications. In *Certification of Safety-Critical Software Controlled Systems (SafeCert)*, March 2009.

[14] L. Higham, J. Kawash, and N. Verwaaland. Defining and Comparing Memory Consistency Models. In *Proc. of PDCS*, 1997.

[15] L. Hubert, T. Jensen, V. Monfort, and D. Pichardie. Enforcing Secure Object Initialization in Java. In *Proc. of ESORICS*, 2010.

[16] M. Huisman and G. Petri. The Java Memory Model: a Formal Explanation. In *Proc. of VAMP*, 2007.

[17] R. Jagadeesan, C. Pitcher, and J. Riely. Generative Operational Semantics for Relaxed Memory Models. In *Proc. of ESOP*, 2010.

[18] JSR. Jsr-133: Java Memory Model and Thread Specification, 2004.

[19] K. Kawachiya, A. Koseki, and T. Onodera. Lock Reservation: Java Locks can mostly do without atomic operations. In *Proc. of OOPSLA*, 2002.

[20] G. Klein and T. Nipkow. A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.*, 28(4), 2006.

[21] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7), 1978.

[22] X. Leroy. A Formally Verified Compiler Back-End. *J. Autom. Reasoning*, 43(4), 2009.

[23] A. Lochbihler. Java and the Java Memory Model – A Unified, Machine-Checked Formalisation. In *Proc. of ESOP*, 2012.

[24] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proc. of POPL*, 2005.

[25] D. Marino, A. Singh, T. D. Millstein, M. Musuvathi, and S. Narayanasamy. A Case for an SC-preserving Compiler. In *Proc. of PLDI*, 2011.

[26] A. Miné. Static Analysis of Run-Time Errors in Embedded Critical Parallel C Programs. In *Proc. of ESOP*, 2011.

[27] S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In *Proc. of TPHOLs*, 2009.

[28] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek. High-level Programming of Embedded Hard Real-Time Devices. In *Proc. of EuroSys*, 2010.

[29] W. Pugh. The initialization on demand holder idiom, 2004. `http://www.cs.umd.edu/~pugh/java/memoryModel`.

[30] W. Pugh. Causality Test Cases for the Java Memory Model, 2004. `http://www.cs.umd.edu/~pugh/java/memoryModel`.

[31] K. Russell and D. Detlefs. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *Proc. of OOPSLA*, 2006.

[32] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The Semantics of x86-CC Multiprocessor Machine Code. In *Proc. of POPL*, 2009.

[33] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding Power Multiprocessors. In *Proc. of PLDI*, 2011.

[34] J. Ševčík. Safe Optimisations for Shared-Memory Concurrent Programs. In *Proc. of PLDI*, 2011.

[35] J. Ševčík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory Concurrency and Verified Compilation. In *Proc. of POPL*, 2011.

[36] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM*, 53(7), 2010.

[37] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: Checking Axiomatic Specifications of Memory Models. In *Proc. of PLDI*, 2010.

[38] J. Ševčík. *Program Transformations in Weak Memory Models*. PhD thesis, The University of Edinburgh, 2009.

[39] J. Ševčík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *Proc. of ECOOP*, 2008.

[40] V. Vafeiadis and F. Z. Nardelli. Verifying Fence Elimination Optimizations. Model. In *Proc. of SAS*, 2011.