

# Plan B: A Buffered Memory Model for Java

Delphine Demange<sup>2</sup>, Vincent Laporte<sup>1,2</sup>, Lei Zhao<sup>1</sup>,  
David Pichardie<sup>1,3</sup>, Suresh Jagannathan<sup>1</sup>, and Jan Vitek<sup>1</sup>

<sup>1</sup> Purdue University

<sup>2</sup> ENS Cachan Bretagne / IRISA

<sup>3</sup> INRIA, Centre Rennes

**Abstract.** The Java Memory Model (JMM) is an ambitious attempt to provide a semantics for concurrent and, possibly, racy Java programs. It aims to provide a precise semantics that is portable across architectures and enables a variety of compiler optimizations. Unfortunately, the JMM has proven to be challenging for users to understand and for compiler writers to use. In fact, the formal statement of the model is flawed and existing Java compilers do not comply with it. We propose to investigate an alternative proposal that has a tractable definition and intuitive semantics, relates easily to existing architectures, while still enabling useful optimizations. To this end, we introduce a *Buffered Memory Model* for Java. Beyond its simplicity advantages for the programmer, the model is also amenable for formal reasoning and verification.

## 1 Introduction

Formally verified compilers are slowly becoming a reality [17, 20]. These efforts provide evidence that it is possible to construct reasonably efficient compilers along with a machine-checked proof of their correctness. This proof attests that every target program generated by the compiler will only exhibit behaviors that the semantics of the program source would allow. Programmers can thus reason about their programs at the source level, guaranteed that such reasoning will not be invalidated by the compiler.

Although these systems greatly relieve the burden of reasoning about compiler correctness, they still require a precise semantics for source programs that can be reasonably understood by programmers. For complex languages like Java, this is a tall order. In particular, in the presence of multiple concurrent threads of control, understanding a program execution must take into account notions of data visibility: when does an update to a location performed by one thread become visible to another? The Java Memory Model is an ambitious attempt to address this question; it aims to provide semantics for concurrent (potentially racy) programs that is portable across architectures and yet nonetheless enables useful compiler optimizations. Unfortunately, the JMM has proven to be challenging for users to understand. One could imagine a memory model that simply enforces *sequential consistency* (SC): the set of valid program execution are limited to those that can only be expressed as a sequential interleaving of

all thread actions. While simple to state and easy to understand, SC imposes fairly onerous restrictions on the kinds of optimizations a compiler might be allowed to perform, and is far removed from the kinds of behaviors permitted by modern microprocessors which take a much more relaxed view of memory consistency [27, 24].

The JMM’s attempt to define a relaxed memory semantics for Java (SC executions are only a strict subset of the executions permitted by the JMM) thus comes at the price of ease-of-understanding. Indeed, when it comes to explaining the semantics of a racy program, current textbooks [13] traditionally elide any attempt to expand on the JMM’s notion of a legal execution. The subtleties that arise in thinking about program executions within this model has been the subject of much research [5, 4]. From the compiler writer’s point of view, the situation is not better because there is no clear operational definition that captures the behaviors permitted by the JMM. Thus, it becomes difficult to write any correctness proof of a Java compiler with respect to the current definition. Indeed, previous work has shown that existing Java compilers are not JMM compliant [31].

To address these issues, we present an alternative memory model definition that has a tractable definition and intuitive semantics, easy for programmers to understand, but that also has a clear operational characterization, making it useful for compiler implementations. To this end, we introduce a *Buffered Memory Model* for Java (BMM). While not as rich as the JMM, it still permits a number of important optimizations, and has the advantage of being suitable for formal reasoning and is easier to validate. BMM comes in two forms whose equivalence we establish. The first,  $\text{BMM}_{\text{ro}}$  is an axiomatic specification close to the current JMM specification style, but which avoids the complex definition of JMM legal executions and provides a more constructive and intuitive method to describe its set of valid executions. It shares some design choices with the recent relaxed memory model proposed by Burckhardt *et al.* [11]. The second one,  $\text{BMM}_{\text{op}}$  is a fully operational abstract machine with write buffers attached to each thread. We lift this operational view up from the x86 buffer memory model proposed by Sewell *et al.* [27]. As pointed out in [5], the JMM was intricate or even flawed in the infinite case. Our work hence focuses on finite executions, as done in many recent work [4, 31].

We formally prove that both versions exhibit the same behaviors. All definitions (Section 3, 4 and 5) of this paper have been formalized with the Coq proof assistant. The companion development is available at

<http://r.cs.purdue.edu/BMM>

## 2 Motivation

The overarching goal of this work is to develop a verified compiler infrastructure for Java. Although there are many different ways we might go about this exercise, an obvious starting point is the construction of a precise source level semantics.

	SC	C++ (DRF model)	JMM	this work
DRF theorem for the programmer	✓	✓	✓	✓
Reordering memory accesses is legal	×	✓	✓	partially
Redundant memory accesses elimination/introduction	✓	✓	partially	partially
Operational semantics	✓	✓	×	✓
Gives a semantics to all programs	✓	×	✓	✓
Programmer can understand the semantics of racy programs	✓	×	×	✓
Sound with respect to existing JMM (does not break legacy Java code)	✓	×	✓	✓

Table 1: Expressivity and properties for different language-level memory models. They all support the DRF guarantee: any data-race free execution a sequentially consistent behavior. The models differ in the kinds of reordering they permit, how they are formalized, the set of programs they consider, and their support for legacy code. BMM is slightly 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. Note that reordering memory accesses is illegal under original JMM [21, 12] but legal under the alternative version proposed by [31].

The structure of this semantics should be both (a) relatively easy to follow to give programmers confidence that they understand the behavior of programs they write, and (b) match well with existing target platforms to give compilers the flexibility to exploit performance capabilities of modern CMPs. As we described above, the choice of memory model plays a critical role in this development.

As part of the fifth author previous work [26], we have developed a verifying compiler for a subset of C based on the Total Store Order (TSO) memory model found in Intel and Sparc processors. The key component of this model is a *store buffer*: each hardware thread effectively has a FIFO buffer of pending memory writes (avoiding the need to block while a write completes), so that reads performed on different processors can occur before the writes have propagated to main memory (see Fig. 1).

While reasonably convenient for a compiler and verifiers, realistic with respect to modern architectures, and perhaps suitable for low-level languages like C, TSO is too low-level and unwieldy as a *high-level* language-level memory model. An approach better suited for this purpose is to reason about data visibility axiomatically, in terms of properties (expressed on source-level behaviors) that partition executions into permissible and non-permissible ones.

While the JMM provides precisely this framework, its definition, which relies on complex and subtle notions of committed (i.e., speculative) actions, often leads to unintuitive explanations that weaken its utility; [5] provides a number

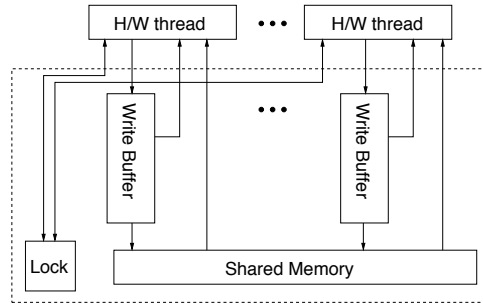


Fig. 1: x86-TSO block diagram.

of examples that illustrate these concerns. For example, the program shown in Fig. 2 does not permit executions in which  $r_1 = r_2 = r_3 = 1$ , but simply reordering the two writes in the else-branch would allow such a result. These subtleties arise because the JMM is defined in terms of a set of partial orders on read and write actions, combined with a causality order on data races; the outcome of a data race must be explained in terms of previously *committed* races.

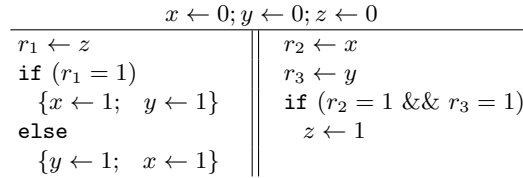


Fig. 2: Understanding the sets of valid and invalid program executions in the JMM can be counter-intuitive. In this example,  $x, y$  and  $z$  are shared memory variables, while  $r_1, r_2$  and  $r_3$  are thread-local variables or registers. The JMM prohibits  $r_1 = r_2 = r_3 = 1$ . However, simply reordering the writes to  $x$  and  $y$  in either of the branches (so both branches perform the writes in the same order), the conditional expression could be eliminated, and assignments could be hoisted above the store to  $r_1$ , making this execution permissible.

Our goals in this paper are thus three-fold: we wish to (a) exploit the operational simplicity of TSO in building a verified compiler infrastructure for Java, (b) reconcile and simplify the complexity of the JMM to better match this operational definition, and (c) ensure the model nonetheless enables useful optimizations. The result, and our primary contribution, is a new memory model for Java we call the Buffered Memory Model. The axiomatic characterization of this model, useful for programmers, is defined using vocabulary similar to what is used in the JMM, restricts the kinds of reorderings that it deems permissible, but avoids the complex notion of race committing sequence. The operational

formulation, useful for compilers, is defined in terms of per-thread store buffers and unbuffering actions on them found in a typical definition of TSO. However, the reordering restrictions imposed by this model, which mirror the behaviors allowed under TSO, are sufficiently strong to enable a simple operational formulation, and a proof of equivalence of the two formulations.

The set of allowed executions permitted by the model is easily stated. First, all SC executions are allowed. Second, any execution from which an SC execution can be derived only by a reordering of a read action with a write action that immediately precedes it (over disjoint locations) is also allowed<sup>4</sup>. Fig. 3 gives some examples of permissible and prohibited executions. These examples closely match the behavior intuitively expected from a TSO relaxed memory model.

$$\begin{array}{cc}
 \frac{x \leftarrow 0; y \leftarrow 0}{x \leftarrow 1 \parallel y \leftarrow 1} & \frac{x \leftarrow 0; y \leftarrow 0}{x \leftarrow 1 \parallel r_1 \leftarrow y} \\
 r_1 \leftarrow y \parallel r_2 \leftarrow x & y \leftarrow 1 \parallel r_2 \leftarrow x \\
 \text{(a) } r_1 = r_2 = 0 & \text{(b) } r_1 = 1, r_2 = 0 \\
 \text{is allowed under BMM} & \text{is illegal under BMM}
 \end{array}$$
  

$$\begin{array}{cc}
 \frac{x \leftarrow 0; y \leftarrow 0}{x \leftarrow 1 \parallel r_1 \leftarrow x \parallel r_2 \leftarrow y} & \frac{x \leftarrow 0; y \leftarrow 0}{x \leftarrow 1 \parallel r_1 \leftarrow x \parallel y \leftarrow 1 \parallel r_3 \leftarrow y} \\
 y \leftarrow 1 \parallel r_3 \leftarrow x & r_2 \leftarrow y \parallel r_4 \leftarrow x \\
 \text{(c) } r_1 = r_2 = 1, r_3 = 0 & \text{(d) } r_2 = r_4 = 0, r_1 = r_3 = 1 \\
 \text{is illegal under BMM} & \text{is illegal under BMM}
 \end{array}$$

Fig. 3: Example (a) shows a program and execution that corresponds to a behavior that may arise in the presence of a store buffer. The invalid executions involve reorderings on pairs of reads that are not permissible in the model.

*Generating Valid BMM Executions* Based on these examples, we can give an intuitive characterization of the set of valid executions that are permitted by BMM. We make these intuitions precise in the following sections. Consider a program  $P$ . To build the set of allowed executions, we first take the set of all well-formed executions of  $P$  (intuitively, well-formed executions maintain intra-thread program dependencies, are properly locked, and every read sees a write that is not overwritten along any *happens-before* path between the two actions). Now, for each execution  $E$  in that set, if it is SC, add it to the set of valid executions ; otherwise, generate all BMM reorderings (including reorderings of writes that precede reads of independent locations), and add  $E$  to the set of valid execution only if an SC execution can be reached this way.

We believe this methodology helps figuring out whether a given execution is allowed or not under BMM. In these previous examples, executions (b), (c) and

<sup>4</sup> This rule can be further generalized to deal with reorderings that involve multiple dependent reads.

(d) are BMM-invalid because they are SC-invalid and no reordering of a read before a write is possible. On the contrary, two such reorderings are possible in execution (a) and they lead to a program execution that is SC-valid. It shows that execution (a) is BMM-valid.

### 3 Background on Java Memory Model

The axiomatic view of BMM is formulated using some of the JMM notions. We recall these in this section, as well as some other preliminary definitions that we will use in BMM. A language memory model specifies formally the interactions between individual threads and central memory. It precisely describes what value can be read by each thread depending on what write is performed in this thread or another. These interactions are categorized using inter-thread actions.

*Inter-thread Actions* The shared memory of a program is split into a set of disjoint shared *addresses*. In practice addresses are instance fields, static fields or array positions but they are not local variables of a method (Java type safety forbids to manipulate their memory addresses). For each address  $x \in \mathbb{X}$ , we can determine if it is volatile or not with the function  $\text{isVolatile} : \mathbb{X} \rightarrow \text{bool}$ . In the literature, external actions are distinguished from other memory actions. In this work, external actions are simply a subset of volatile writes, that can be identified with the function  $\text{isExternal} : \mathbb{X} \rightarrow \text{bool}$ .<sup>5</sup> We believe this is not an unreasonable simplification for compilers to make.

We assume a set  $\mathbb{T}$  of dynamic threads, a set  $\mathbb{L}$  of memory locks. The set of inter-thread actions we consider is given below, where superscript  $i$  denotes the unique identifier of memory actions.

$\mathbb{A} \ni a ::=$	$W_t^i x$	(write action to the address $x$ , by thread $t$ )
	$R_t^i x$	(read action to the address $x$ , by thread $t$ )
	$L_t^i l$	(thread $t$ acquires a lock on monitor $l$ )
	$U_t^i l$	(thread $t$ releases a lock on monitor $l$ )
	$Spw_t t'$	(thread $t$ spawns a new thread $t'$ )
	$St_t$	(thread $t$ starts)
	$Join_t^i t'$	(thread $t$ detects that thread $t'$ has terminated)
	$End_t$	(thread $t$ ends)
	$W_0 x$	(default write action to the address $x$ )
		$x \in \mathbb{X}, l \in \mathbb{L}, t, t' \in \mathbb{T}, i \in \mathbb{N}$

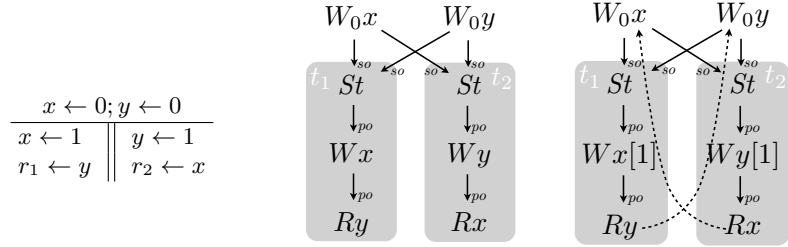
Action  $W_0 x$  is the default write action to the address  $x$ . It has no emitting thread: memory initialization is done somewhat implicitly in our formalization – no particular thread is in charge of initializing the memory<sup>6</sup>. Thread starting,

<sup>5</sup> We require that,  $\forall x, \text{isExternal}(x) \Rightarrow \text{isVolatile}(x)$

<sup>6</sup> This point diverges from [21, 4]: we believe this is both closer to the initial JMM proposal [16] and more suitable for an operational characterization. In Java, because the language is type safe, every address is virtually given a default value at the start of the program, even if the corresponding location is not allocated yet.

ending and spawning actions can happen only once during an execution and thus do not require any identifier. For any action  $a$  that is not a default write action, we note  $T(a)$  the emitting thread of this action. We introduce some notations for families of actions:

$$\begin{aligned}
\mathbb{A}_r &= \{R_t^i x \mid i \in \mathbb{N}, t \in \mathbb{T}, x \in \mathbb{X}\} && \text{(read actions)} \\
\mathbb{A}_w &= \{W_t^i x, W_0 x \mid i \in \mathbb{N}, t \in \mathbb{T}, x \in \mathbb{X}\} && \text{(write actions)} \\
\mathbb{A}_{dw} &= \{W_0 x \mid x \in \mathbb{X}\} && \text{(initialization actions)} \\
\mathbb{A}_{start} &= \{St_t \mid t \in \mathbb{T}\} && \text{(start actions)} \\
\mathbb{A}_s &= \{W_t^i x, R_t^i x \mid i \in \mathbb{N}, t \in \mathbb{T}, x \in \mathbb{X}, \text{isVolatile}(x)\} && \text{(synchronising actions)} \\
&\quad \cup \{L_t^i l, U_t^i l \mid i \in \mathbb{N}, t \in \mathbb{T}, l \in \mathbb{L}\} \\
&\quad \cup \{Spw_t t', St_t, Join_t t', End_t \mid t, t' \in \mathbb{T}, i \in \mathbb{N}\} \\
\mathbb{A}_x &= \{W_t^i x \mid i \in \mathbb{N}, t \in \mathbb{T}, x \in \mathbb{X}, \text{isExternal}(x)\} && \text{(external actions)}
\end{aligned}$$



(a) Code of threads  $t_1$  and  $t_2$  (b) Causality relation (c) Axiomatic execution

Fig. 4: An example of program and one of its happens-before execution.

Current JMM and several architecture memory models are based on a *happens-before* model [18]. An execution is described in terms of partial orders between memory actions. In a multithreaded program, each thread executes its own program and reads/writes in the memory according to it. A lot of different interleavings of thread-actions are generally possible for the same external behavior of a program. An interleaving can be seen as a total order on actions: “this action occurs before that one according to the global time of the interleaving”. Such an interleaving is in fact a consistent extension of a partial order called “happens before” that relates more precisely the causality dependencies between actions. For example, the program Fig. 4a may lead to the interleaving of thread-actions

$$St_{t_1} :: St_{t_2} :: W_{t_1}^3 x :: R_{t_1}^4 y :: W_{t_2}^5 y :: R_{t_2}^6 x$$

but they are no causality dependency between the read performed in  $t_1$  and the one performed in  $t_2$ . Fig. 4b presents the causality relation behind such a linear presentation. The owning thread of each action is implicit with gray regions. Unique identifier is useless here and we omit them. Any sequence of arrows

between an action  $a$  and an action  $b$  means  $a$  *should happens before*  $b$ . Such a program is poorly synchronised so, apart from address initialisation and thread starts, few actions are really constrained here. We distinguish two kinds of arrows in this example. The arrow  $\xrightarrow{po}$  reflects the program order between actions of a same thread. The arrow  $\xrightarrow{so}$  reflects the synchronisation relation between some events. Here it is reduced to a relation between address initialisations and thread starts but in more complex examples, it may relate an unlock of a monitor with a subsequent lock or the write of a volatile address with a subsequent read.

To complete this partial-order view, it is necessary to model the values read and written during an execution (we assume a set  $\text{Val}$  of dynamic values). In Fig. 4c we complete the picture with the value-seen of each write (in brackets<sup>7</sup>) and the write seen by each read action (dotted line). These arrows and this value seen/written information form what we call an axiomatic execution. The write seen by a read action must satisfy some minimal constraints that we will make clear with the notion of well-formed execution. We postpone until the next paragraph the precise requirements we demand on programs – a formal instantiation of this abstract notion is provided in Appendix D.

*Notations* When a partial order is total on a countable set of elements we sometimes note 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 total orders, we note  $[\xrightarrow{o}]_t$  its restriction on a thread  $t$ . Conversely, a list of elements can be thought of as a total order. We will write  $a \xrightarrow{tr} b$  when elements  $a, b$  are ordered with regards to a list  $tr$ . Notice that what is called an *order* in this paper is any irreflexive transitive relation. Two such relations  $P$  and  $Q$  are said *consistent* when they satisfy:  $\forall x, y, \neg(xPy \wedge yQx)$ . We write  $tr \downarrow_A$  for the sequence  $tr$  filtered to the elements of the set  $A$ .

**Definition 1 (Axiomatic Execution).** An execution  $E$  is described by a tuple

$$E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, V, W \rangle$$

where:

- $P$  is a program;
- $A \subseteq \mathbb{A} \setminus \mathbb{A}_{dw}$  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;
- $\xrightarrow{so} \subseteq (A \cup \mathbb{A}_{dw}) \times (A \cup \mathbb{A}_{dw})$  is the synchronisation order: the union of a total order on the set  $A \cap \mathbb{A}_s$  of all synchronisation actions in  $A$ , and the Cartesian product  $\mathbb{A}_{dw} \times (A \cap \mathbb{A}_s)$ ;
- $V \in \mathbb{A}_w \rightarrow \text{Val}$  is a value-seen function for  $A$  that assigns a value to each write action from  $A$ ;

<sup>7</sup> The value-seen of an initialisation write can be left implicit because it only depends on the type of the related address.

- $W \in \mathbb{A}_r \rightarrow \mathbb{A}_w$  is a write-seen function that maps each read action  $r$  from  $A$  to a write action  $w$  of  $A \cup \mathbb{A}_{dw}$  ( $r$  and  $w$  must operate on the same address).

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

**Definition 2 (Synchronises-with relation).** An action  $a$  synchronises-with an action  $b$  (written  $a \xrightarrow{sw} b$ ) in an execution  $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, V, W \rangle$  if  $a \xrightarrow{so} b$  and  $a, b$  satisfy one of the following conditions:

- $a \in \mathbb{A}_{dw}$  and  $b \in A \cap \mathbb{A}_{start}$ ,
- $a$  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 lock on monitor  $l$ ,
- $a$  is the end of the thread  $t$  and  $b$  is a join action on  $t$ .

**Definition 3 (Happens-before order).** The happens-before order of an execution is the transitive closure of the union of its synchronises-with relation and its program order.

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

*Intra-thread semantics* Our formalization tries to be as independent as possible of the concrete details of the underlying programming language (Java source or bytecode, intermediate representation of a Java compiler...). We give a concrete instantiation of this abstract program model in our companion Coq development. The only requirement we make on programs is an abstract notion of intra-thread semantic state  $\text{State}_{intra}$  and an intra-thread labelled transition relation

$$\dashrightarrow \subseteq \text{State}_{intra} \times \text{Label}_{intra} \times \text{State}_{intra}$$

that is given to each thread  $t \in \mathbb{T}$ . See Appendix D for a formal definition. Transition labels belong to the set  $\text{Label}_{intra} = (\mathbb{A} \times \text{Val}) \cup \{\tau\}$ : a thread can either make an action step, or a silent step that is memory irrelevant. For an action step, the value in the label is relevant only for read and write actions. The requirements on this intra-thread semantics are:

- $\dashrightarrow$  can only relate states of the same thread
- there is an *initial* state  $\text{Ready}$ : no transition leads to this state and a thread  $t$  can only step from it emitting the  $St_t$  action
- there is a *final* state  $\text{Done}$ : only transitions labelled by  $End_t$  lead to it and no transition steps from this state.

**Definition 4 (Intra traces).** Let  $tr = a_1 :: \dots :: a_n$  a sequence of actions in set  $A$ ,  $V$  a value-seen function on  $A$  and  $W$  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  $s_0, s_1, \dots, s_m \in \text{State}_{intra}$  ( $m \geq n$ ) and  $l = l_1 :: \dots :: l_m \in \text{list}(\text{Label}_{intra})$  such that:

- for all  $a \in \{a_1, \dots, a_n\}$ ,  $T(a) = t$

- $s_0$  is the initial intra-thread state Ready
- for all  $i \in \{1, \dots, m\}$ ,  $s_{i-1} \xrightarrow{l_i} s_i$  and
- $(a_1, v_1) :: \dots :: (a_n, v_n)$  is the projection of  $l$  to non-silent actions
- $v_i = V(a_i)$  if  $a_i \in \mathbb{A}_w$  and  $v_i = V(W(a_i))$  if  $a_i \in \mathbb{A}_r$

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

**Definition 5 (Well-formed execution).** An execution  $\langle P, A, \xrightarrow{po}, \xrightarrow{so}, V, W \rangle$  is well-formed if

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

$$\begin{aligned} \forall L_i^l \in A, \forall t' \neq t, \# \left\{ L_{t'}^j | L_{t'}^j \xrightarrow{so} L_i^l \right\} &= \# \left\{ U_{t'}^j | U_{t'}^j \xrightarrow{so} L_i^l \right\} \\ \forall U_i^l \in A, \# \left\{ L_{t'}^j \in A | L_{t'}^j \xrightarrow{po} U_i^l \right\} &> \# \left\{ U_{t'}^j \in A | U_{t'}^j \xrightarrow{po} U_i^l \right\} \end{aligned}$$

- $\xrightarrow{po}$  is intra-thread consistent: for all threads  $t \in \mathbb{T}$ ,  $([\xrightarrow{po}]_t, V, W) \in P[t]$ ;
- $\xrightarrow{so}$  is consistent with  $W$ : for every read  $r$  of a volatile variable  $x$  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 special subfamily of well-formed executions is the set of Sequentially Consistent axiomatic executions.

**Definition 6 (Sequentially Consistent (SC<sub>ax</sub>) execution).** A well-formed execution  $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, V, W \rangle$  is SC<sub>ax</sub> 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 Java memory model because it allows *out-of-thin-air* values [21] and breaks basic principles of the Java security. The JMM [21] considers then a subset of this model (but still containing SC executions): the *legal* executions. The exact definition of legal executions is somewhat technical and convoluted. In a nutshell,

a well-formed execution  $E$  is legal if there exists a sequence  $E_0, E_1, \dots, E_n$  of well-formed executions such that in  $E_0$ , each read sees a write that is not in race with it. 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 the two important properties of the JMM: 1) in a data race free program all reads see writes that happen before them and each execution is sequentially consistent 2) No out-of-thin-air value can be read, by the causality order on races imposed by the JMM.

*Local transformations* Compilers or architectures may want to optimize some individual thread executions. We finish this section by formalizing the notion of local reordering we will consider in this work.

**Definition 7 (Local reordering).** Let  $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, V, W \rangle$  an execution.  $E' = \langle P', A, \xrightarrow{po'}, \xrightarrow{so}, V, W \rangle$  is a local reordering of  $E$  from a list  $b$  of initial actions to a list  $a$  of final actions in thread  $t_0$  if

- $[\xrightarrow{po}]_{t_0} = \alpha_0 \cdot b \cdot \beta_0$  and  $[\xrightarrow{po'}]_{t_0} = \alpha_0 \cdot a \cdot \beta_0$  for some sequences  $\alpha_0$  and  $\beta_0$ ;
- $[\xrightarrow{po}]_t = [\xrightarrow{po'}]_t$  for all threads  $t \neq t_0$ ;
- for all  $(tr, V, W) \in P'[t_0]$  where  $tr$  is of the form  $\alpha \cdot a \cdot \beta$ , there exists  $(\alpha \cdot b \cdot \beta, V, W) \in P[t_0]$ ;
- $P[t] = P'[t]$  for all thread  $t \neq t_0$ ;
- $b$  and  $a$  contain the same set of actions;
- neither elements of  $b$  or  $a$  are synchronisation actions.

Such a reordering is written  $E \xrightarrow{t_0:[b \rightarrow a]} E'$ .

Intuitively, we make a permutation of the intra-trace  $[\xrightarrow{po}]_{t_0}$  of thread  $t_0$  by transforming the sequence  $b$  into the sequence  $a$ .

## 4 Axiomatic memory model: $\text{BMM}_{\text{ro}}$

We define formally a first formal view of our memory model. The semantics is built on top of two notions that programmers should arguably well understand: sequential consistency and instruction reordering.  $\text{BMM}_{\text{ro}}$  exposes two<sup>8</sup> fundamental instruction reorderings to the programmer. The first one is *Write-Read* reordering. Its basic effect is to reorder a read before a previous adjacent write if both actions target different addresses. Here is a simple example of Write-Read reordering of a program:

$$\begin{array}{ccc} x \leftarrow 1 & \xrightarrow{\text{WR}} & r_1 \leftarrow y \\ r_1 \leftarrow y & & x \leftarrow 1 \end{array}$$

<sup>8</sup> *Write-Read* reordering is in fact a special case of the forthcoming *Write-Read-Read* reordering but we believe it is easier to reason first on this simpler case before generalizing.

**Definition 8 (Write-Read reordering).** A Write-Read reordering of an execution  $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, V, W \rangle$  with respect to a couple of write/read actions  $(w, r)$  of  $A$  in a thread  $t$ , is a local reordering  $E'$  such that

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

$w$  and  $r$  must not operate on the same address. Such a reordering is written  $E \xrightarrow{WR} E'$ .

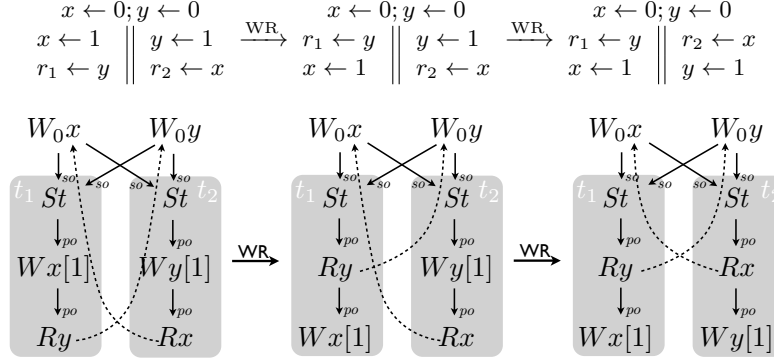


Fig. 5: Write-Read reordering example

Fig. 5 illustrates the use of the Write-Read reordering on a classical litmus test program. To understand the  $BMM_{ro}$  semantics, a key observation must be made here: 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  $BMM_{ro}$  execution to be any execution that we can transform into an SC execution after some WR reordering. Unfortunately such a definition would not allow to capture some execution exhibited by TSO-hardware.

The program on top-left part of Fig. 6 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 there is no way to apply any WR reordering on this program. We introduce then a second category of reorderings that is allowed in  $BMM_{ro}$ .

**Definition 9 (Write-Read-Read reordering).** A Write-Read-Read reordering of an execution  $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, V, W \rangle$  w.r.t. a tuple of write/reads/read action  $(w, r_1 :: \dots :: r_n, r')$  of  $A$ , is a local reordering  $E'$  such that

$$E \xrightarrow{t:[w::r_1::\dots::r_n::r' \rightarrow r'::w::r_1::\dots::r_n]} E'$$

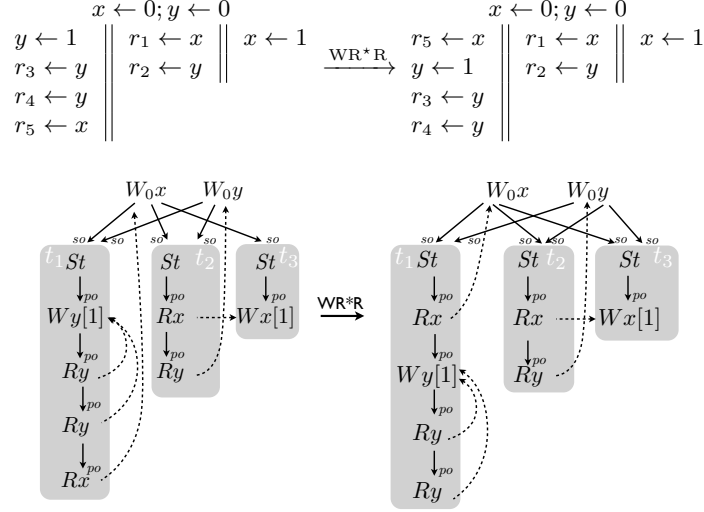


Fig. 6: Write-Read-Read reordering example

All reads  $r_1, \dots, r_n$  must have  $w$  as write-seen, and  $r'$  and  $w$  must target different addresses. Such a reordering is written  $E \xrightarrow{\text{WR}^*\text{R}} E'$ .

In Fig. 6, we apply this transformations to the previous program. This time, a reordering is possible and leads to an SC execution. We prove in Section 6 that these two reorderings capture exactly a TSO operational semantics.

Formally, a  $\text{BMM}_{\text{ro}}$  execution is any execution that can be successively transformed using WR and  $\text{WR}^*\text{R}$  reordering until reaching a SC execution.

**Definition 10** ( $\text{BMM}_{\text{ro}}$  executions). *The set of  $\text{BMM}_{\text{ro}}$  executions is defined:*

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

where  $\xrightarrow{\text{RO}} = (\xrightarrow{\text{WR}} \cup \xrightarrow{\text{WR}^*\text{R}})^*$ .

In the sequel, we will write  $\text{BMM}_{\text{ro}}(P)$  for the set of executions of a program  $P$ . The  $\text{BMM}_{\text{ro}}$  observable behaviors of a program  $P$  is then defined as the set of sequences of external action - value pairs:

$$\text{Beh}_{\text{ro}}(P) = \left\{ (\xrightarrow{\text{so}} \downarrow_{\mathbb{A}_x}, V, W) \mid \langle P, A, \xrightarrow{\text{po}}, \xrightarrow{\text{so}}, V, W \rangle \in \text{BMM}_{\text{ro}}(P) \right\}$$

#### 4.1 $\text{BMM}_{\text{ro}}$ is a least post-fixpoint

Every time we need to prove that  $\text{BMM}_{\text{ro}}$  is included in a given set of well-formed executions, we can rely on the following post-fixpoint characterisation.

**Lemma 1 (BMM<sub>ro</sub> least post-fixpoint characterisation).** *BMM<sub>ro</sub> is the least set  $S$  that satisfies*

- all  $SC_{ax}$  executions are in  $S$ ;
- $S$  is backward-closed by BMM<sub>ro</sub> transformations: for any well-formed execution  $E, E'$  such that  $E \xrightarrow{RO} E'$ , if  $E' \in S$  then  $E \in S$

We will use this lemma to show that BMM<sub>ro</sub> is a subset of the JMM executions and for one of the double inclusion between BMM<sub>ro</sub> and the operational semantics given in Section 5.

## 4.2 BMM<sub>ro</sub> 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:

- the JMM accepts all sequentially consistent executions
- WR is a valid execution transformation in the JMM
- WR\*R is a valid execution transformation in the JMM (by composition of WR and redundant read after write elimination)

The validity of the Write-Read reordering and redundant-read-after-write elimination have been formally proved in [31].

**Theorem 1.** *Let JMM the set of all JMM legal executions.  $BMM_{ro} \subseteq JMM$ .*

*Proof.* By [31] we know that JMM contains all  $SC_{ax}$  executions and that the transformations WR and WR\*R are valid in the JMM: it means JMM is backward-closed by these transformations. We can hence conclude with the previous lemma.  $\square$

## 5 Operational memory model : BMM<sub>op</sub>

In this section, we provide an operational view of the Java Buffered Memory Model: BMM<sub>op</sub>. The reorderings allowed in its axiomatic version can be somewhat implemented by a BMM<sub>op</sub> machine, that attaches a write-buffer to each running thread. The BMM<sub>op</sub> machine semantics will also be parametrized by an intra-thread semantics as specified in Section 3. Hence, we need to consider an extra set of actions: the silent actions in  $\mathbb{A}_{sil}$ .

$$\begin{array}{l} \mathbb{A}_{sil} \ni sa ::= B(a) \quad (\text{unbuffering of an action } a \in \mathbb{A}_w \setminus \mathbb{A}_{dw} \text{ by thread } T(a)) \\ \quad \quad \quad | \tau_t \quad (\text{silent step by thread } t) \end{array}$$

The idea behind BMM<sub>op</sub> is to provide a generative, operational machine that produces executions in a format that is very close to what is specified in

Section 3. Given an input operational execution, the machine executes, modifying a memory state that is made of thread buffers and a shared memory. Along its execution, the machine also builds all the ingredients of an axiomatic executions.

The input of the  $\text{BMM}_{\text{op}}$  machine is an operational execution, made of a program, an action trace, and two value/write seen functions:

**Definition 11 (Operational Execution).** *An operational execution is a tuple  $(P, tr, V, W)$  where  $P$  is a program,  $V$  is a value-seen function,  $W$  is a write-seen function and  $tr \in \text{list}(\mathbb{A} \cup \mathbb{A}_{\text{sil}})$  is finite and such that no action appear more than once in  $tr$ .*

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

$$\langle \begin{array}{ll} ts \in \mathbb{T} \mapsto \text{State}_{\text{intra}}; & \text{(intra thread state of threads)} \\ b \in \mathbb{T} \mapsto \text{list}(\mathbb{A}_{\text{w}} \setminus \mathbb{A}_{\text{dw}}); & \text{(one buffer per thread)} \\ m \in \mathbb{X} \mapsto \mathbb{A}_{\text{w}}; & \text{(one write action per location)} \\ ws \in \mathbb{A}_{\text{r}} \mapsto \mathbb{A}_{\text{w}}; & \text{(write-seen function)} \\ vs \in \mathbb{A}_{\text{w}} \mapsto \text{Val}; \end{array} \rangle$$

The semantics state first keeps track of each thread intra state in  $\text{State}_{\text{intra}}$ . Each thread is given a write buffer, through which must go all non-volatile write actions performed by any thread. When unbuffered, these writes are committed to the shared memory  $m$ , that maps addresses to write actions. Note that the value associated to each location in  $m$  can be retrieved through the function  $V$ . Given a memory state (buffers  $b$  and memory  $m$ ), the  $\text{BMM}_{\text{op}}$  machine specifies the write action a thread  $t$  can read when accessing the address  $x$ :

$$\text{read}_t(b, m, x) = \begin{cases} w & \text{if } w \text{ is the first write to address } x \text{ in } b[t] \\ m(x) & \text{if no write to } x \text{ is in } b[t] \\ W_0x & \text{if } x \notin \text{dom}(m) \end{cases}$$

The  $\text{BMM}_{\text{op}}$  machine is defined as a labelled transition system where steps are either labelled by an action in  $\mathbb{A}$ , or silent steps labelled by silent actions in  $\mathbb{A}_{\text{sil}}$ . Semantic rules are given Fig. 7. In all rules (except  $\text{BUFF}$ ) the  $\text{BMM}_{\text{op}}$  machine makes a step that the intra-thread semantics can match. Rule  $\text{TAU}$  corresponds to an intra-thread silent step (when e.g. the thread is manipulating his local registers). Then  $\text{BMM}_{\text{op}}$  machine does not change state in this case. On a non-volatile writing step, the write action is put into the buffer of the thread (rule  $\text{WRITE}$ ). A write action can be unbuffered at any time, in which case the write action is committed into shared memory. The value-seen is extended at every writing step (rules  $\text{WRITE}$  and  $\text{WRITEW}$ ), and the write-seen during reading steps (rules  $\text{READ}$ , and  $\text{READW}$ ), according to the function  $\text{read}$  defined above. Notice however that accesses to volatile locations require the buffer of the thread to be empty, ensuring that all threads have a consistent view of volatile locations – they are always directly read from or written to the shared memory. Similarly all synchronising actions are emitted by threads whose buffers are

$$\begin{array}{c}
\frac{ts(t) \xrightarrow{-\tau} s}{\langle ts, b, m, ws, vs \rangle \xrightarrow{\tau} \langle ts[t \mapsto s], b, m, ws, vs \rangle} \text{[TAU]} \\
\frac{b(t) = l \cdot [W_t^i x]}{\langle ts, b, m, ws, vs \rangle \xrightarrow{B(W_t^i x)} \langle ts, b[t \mapsto l], m[x \mapsto W_t^i x], ws, vs \rangle} \text{[BUFF]} \\
\frac{ts(t) \xrightarrow{W_t^i x, v} s \quad \text{isVolatile}(x)}{\langle ts, b, m, ws, vs \rangle \xrightarrow{W_t^i x} \langle ts[t \mapsto s], b[t \mapsto W_t^i x :: b(t)], m, ws, vs[W_t^i x \mapsto v] \rangle} \text{[WRITE]} \\
\frac{ts(t) \xrightarrow{R_t^i x, v} s \quad w = \text{read}_t(b, m, x) \quad v = vs(w) \quad \text{isVolatile}(x)}{\langle ts, b, m, ws, vs \rangle \xrightarrow{R_t^i x} \langle ts[t \mapsto s], b, m, ws[R_t^i x \mapsto w], vs \rangle} \text{[READ]} \\
\frac{ts(t) \xrightarrow{W_t^i x, v} s \quad \text{isVolatile}(x) \quad b(t) = []}{\langle ts, b, m, ws, vs \rangle \xrightarrow{W_t^i x} \langle ts, b, m[x \mapsto W_t^i x], ws, vs[W_t^i x \mapsto v] \rangle} \text{[WRITEV]} \\
\frac{ts(t) \xrightarrow{R_t^i x, v} s \quad w = \text{read}_t(b, m, x) \quad v = vs(w) \quad \text{isVolatile}(x) \quad b(t) = []}{\langle ts, b, m, ws, vs \rangle \xrightarrow{R_t^i x} \langle ts[t \mapsto s], b, m, ws[R_t^i x \mapsto w], vs \rangle} \text{[READV]} \\
\frac{ts(t) \xrightarrow{L_t^i l} s \quad b(t) = []}{\langle ts, b, m, ws, vs \rangle \xrightarrow{L_t^i l} \langle ts[t \mapsto s], b, m, ws, vs \rangle} \text{[LOCK]} \\
\frac{ts(t) \xrightarrow{U_t^i l} s \quad b(t) = []}{\langle ts, b, m, ws, vs \rangle \xrightarrow{U_t^i l} \langle ts[t \mapsto s], b, m, ws, vs \rangle} \text{[UNLOCK]} \\
\frac{ts(t) \xrightarrow{Spw_t t'} s \quad t' \notin \text{dom}(b) \quad t' \notin \text{dom}(ts) \quad b(t) = []}{\langle ts, b, m, ws, vs \rangle \xrightarrow{Spw_t t'} \langle ts[t \mapsto s][t' \mapsto \text{Ready}], b[t' \mapsto \varepsilon], m, ws, vs \rangle} \text{[SPAWN]} \\
\frac{ts(t) = \text{Ready} \quad \text{Ready} \xrightarrow{St_t} s \quad b(t) = []}{\langle ts, b, m, ws, vs \rangle \xrightarrow{St_t} \langle ts[t \mapsto s], b, m, ws, vs \rangle} \text{[START]} \\
\frac{ts(t) \xrightarrow{End_t} \text{Done} \quad b(t) = []}{\langle ts, b, m, ws, vs \rangle \xrightarrow{End_t} \langle ts[t \mapsto \text{Done}], b, m, ws, vs \rangle} \text{[END]} \\
\frac{ts(t') = \text{Done} \quad ts(t) \xrightarrow{Join_t t'} s \quad b(t) = []}{\langle ts, b, m, ws, vs \rangle \xrightarrow{Join_t t'} \langle ts[t \mapsto s], b, m, ws, vs \rangle} \text{[JOIN]}
\end{array}$$

Fig. 7: BMM<sub>op</sub> machine (labelled transition system)

empty. When a thread ends, its state is kept in the  $\text{BMM}_{\text{op}}$  state, enabling other threads to join it, and preventing it to be restarted.

We then define a  $\text{BMM}_{\text{op}}$  execution as a constrained operational execution that is accepted by the  $\text{BMM}_{\text{op}}$  machine: the input trace can be executed by the machine, and the machine has been able to rebuild the same value-seen and write-seen functions (held in the last semantic state), meaning in some sense, that the input execution was intra-thread consistent.

**Definition 12** ( $\text{BMM}_{\text{op}}$  execution). *An operational execution  $(P, tr, V, W)$  is in  $\text{BMM}_{\text{op}}(P)$  if there exist states  $s_0, s_1, \dots, s_n \in \text{State}$  satisfying:*

- $tr$  is properly locked (see Definition 5, using  $\xrightarrow{tr}$  instead of  $\xrightarrow{po}$  and  $\xrightarrow{so}$ );
- $s_0$  is an initial state: memory, buffers and the value-seen and write-seen functions are empty and the  $s_0.ts$  is defined for exactly one thread, mapping it to the Ready state;
- $tr = a_1 :: \dots :: a_n \in \text{list}(\mathbb{A} \cup \mathbb{A}_{\text{sil}})$
- for all  $i \in \{1, \dots, n\}$ ,  $s_{i-1} \xrightarrow{a_i} s_i$
- $V = s_n.vs$  and  $W = s_n.ws$ ;

The  $\text{BMM}_{\text{op}}$  behaviors of a program  $P$  are the external action traces obtained by projecting on  $\mathbb{A}_x$  all executions of  $P$  accepted by the  $\text{BMM}_{\text{op}}$  machine:

$$\text{Beh}_{\text{op}}(P) = \{(tr \downarrow_{\mathbb{A}_x}, V, W) \mid (P, tr, V, W) \in \text{BMM}_{\text{op}}(P)\}$$

## 6 $\text{BMM}_{\text{ro}}$ and $\text{BMM}_{\text{op}}$ are equivalent

In this section, we show that  $\text{BMM}_{\text{ro}}$  and  $\text{BMM}_{\text{op}}$  are equivalent relaxed memory models: they allow the exact same set of behaviors for any program. In Section 7, we will show they both enjoy a DRF guarantee, meaning that, on data-race free programs, they both coincide with a sequential consistent memory model. The remainder of this section is dedicated to the proof of Theorem 2:

**Theorem 2.** *For all program  $P$ ,  $\text{Beh}_{\text{op}}(P) = \text{Beh}_{\text{ro}}(P)$ .*

The proof is somewhat technical, and relies on auxiliary lemmas. Hence, we explain here the main ideas, giving some insight about the technical arguments, and provide a full proof in Appendix A. We define an operator  $\rho$  that, given an operational execution, builds an axiomatic execution.

**Definition 13** ( $\rho$  operator). *Let  $E_{\text{op}} = (P, tr, V, W)$  an operational execution. The operator  $\rho$  is defined as  $\rho(E_{\text{op}}) = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, V, W \rangle$  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  $a, b \in \mathbb{A}_s$  and  $a \xrightarrow{tr} b$ .

Based on this operator, we define auxiliary notions that we use in the proof (and nowhere else): an operational execution  $E_{op}$  is well-formed if  $\rho(E_{op})$  is well-formed, and it is  $SC_\alpha$  if  $\rho(E_{op})$  is  $SC_{ax}$ . Similarly, we define operational reorderings relying on  $\rho$ : an execution  $E'_{op}$  is an operational local reordering  $\phi$  of  $E_{op}$  if  $\rho(E_{op}) \xrightarrow{\phi} \rho(E'_{op})$ , with  $\phi \in \{WR, WR^*R\}$ . We will abuse notations and write it  $E_{op} \xrightarrow{\phi} E'_{op}$ , and also lift this notion to trace reorderings  $\xrightarrow{RO}$ .

The operator  $\rho$  bridges the gap between  $BMM_{ro}$  and  $BMM_{op}$ : we will show that  $\rho(BMM_{op}) = BMM_{ro}$ . Each inclusion is proved, in its own subsection, and Theorem 2 follows nicely thanks to the following lemma, which trivially holds by definition of  $\rho$ :

**Lemma 2.** *Let  $E_{op} = (P, tr, V, W)$  and  $E_{ro} = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, V, W \rangle$  two executions such that  $E_{ro} = \rho(E_{op})$ . Then  $(tr \downarrow_{\mathbb{A}_x}, V, W) = (\xrightarrow{so} \downarrow_{\mathbb{A}_x}, V, W)$ .*

### 6.1 $\rho(BMM_{op}) \subseteq BMM_{ro}$

At the heart of this proof is the following lemma, stating that the reordering specification  $BMM_{ro}$  can be simulated in the operational world:

**Lemma 3.** *Let  $E_{op} = (P, tr, V, W) \in BMM_{op}(P)$ . Then, there exists  $P', tr'$  such that  $E_{op} \xrightarrow{RO} (P', tr', V, W)$ , with  $(P', tr', V, W) \in BMM_{op}(P')$  is  $SC_\alpha$ .*

Before going into more details in the proof of Lemma 3, let us show how it helps proving the first inclusion.

**Corollary 1.**  $\rho(BMM_{op}) \subseteq BMM_{ro}$ .

*Proof.* Let  $E_{op} = (P, tr, V, W) \in BMM_{op}(P)$ . By Lemma 3, we have that  $E_{op} \xrightarrow{RO} (P', tr', V, W)$ , with  $E'_{op} = (P', tr', V, W) \in BMM_{op}(P')$  is  $SC_\alpha$ . By definition,  $\rho(E_{op}) \xrightarrow{RO} \rho(E'_{op})$  and  $\rho(E'_{op})$  is  $SC_{ax}$ . Hence,  $\rho(E_{op}) \in BMM_{ro}(P)$ .  $\square$

Let us now briefly sketch how the proof of Lemma 3 is conducted. It heavily relies on a reordering scheme performed on operational execution in  $BMM_{op}$ :

**Lemma 4.** *Let  $E_{op} = (P, tr, V, W) \in BMM_{op}(P)$  with  $tr = \alpha \cdot [W_t^i x] \cdot \beta$  and  $B(W_t^i x) \notin \beta$ . We note:*

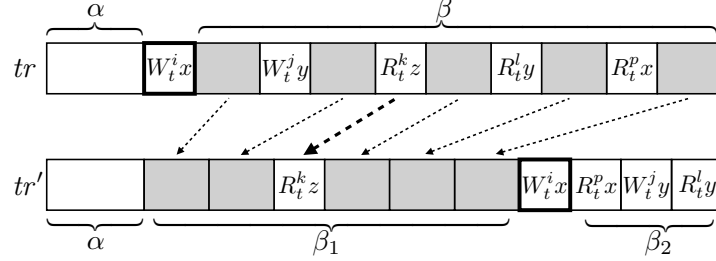
- $\mathcal{W}^t = \{W_t^j y \in \beta \mid y \in \mathbb{X}, j \in \mathbb{N}\}$  the set of write actions in  $\beta$  from thread  $t$ .
- $\mathcal{R}^t = \{R_t^j y \in \beta \mid W_t^i x \cdot \beta = \beta_1 \cdot W_t^k y \cdot \beta_2 \cdot R_t^j y \cdot \beta_3, y \in \mathbb{X}, j, k \in \mathbb{N}, \beta_1, \beta_2, \beta_3 \in \text{list}(\mathbb{A} \cup \mathbb{A}_{sil})\}$  the read actions in  $\beta$  that see a write from thread  $t$  in  $[W_t^i x] \cdot \beta$ .
- $\beta \setminus (\mathcal{W}^t \cup \mathcal{R}^t)$  the remaining actions in  $\beta$ .

*Then, there exist  $P', \beta_1, \beta_2$  such that:  $E'_{op} = (P', tr', V, W) \in BMM_{op}(P')$ ,  $E_{op} \xrightarrow{RO} E'_{op}$  and  $tr' = \alpha \cdot \beta_1 \cdot [W_t^i x] \cdot \beta_2$  with:*

- $\beta_1 = \beta \downarrow_{\beta \setminus (\mathcal{W}^t \cup \mathcal{R}^t)}$

- $\beta_2$  contains the elements of  $\mathcal{W}^t \cup \mathcal{R}^t$
- $[W_t^i x] \cdot \beta_2$  matches the pattern  $(W_t^i x_1; (R_t^i x_1)^*) \cdot \dots \cdot (W_t^i x_n; (R_t^i x_n)^*)$

We will not detail the proof here (see Appendix A), but rather give an intuition about the reordering scheme. This reordering scheme can be applied on a part of an execution during which a given write action, performed by say thread  $t$ , keeps in its buffer. This is illustrated, with the notations of the lemma, by the following figure, where the grey regions denote subsequences of action whose owning thread is not  $t$ . The bold stroke action  $W_t^i x$  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 WR\* $\mathcal{R}$  reordering rule. First, write actions of  $t$  cannot be moved so are kept to the right of the pivot.



Then, read actions of  $t$  are more involved: either they see a write to a variable that occurred (necessarily in thread  $t$ ) after the pivot in  $tr$ , and they are accumulated in a pattern  $(W_t^i x_1; (R_t^i x_1)^*) \cdot \dots \cdot (W_t^i x_n; (R_t^i x_n)^*)$ , or they see a write that occurs before the pivot, and WR\* $\mathcal{R}$  is applied repeatedly on this pattern, until they are swapped before  $W_t^i x$ . Proof of Lemma 3 is then conducted by induction on the length of the operational execution, and uses intensively the reordering lemma.

## 6.2 $\text{BMM}_{\text{ro}} \subseteq \rho(\text{BMM}_{\text{op}})$

We reuse here the post-fixpoint characterization of  $\text{BMM}_{\text{ro}}$  (Lemma 1).

We first show that  $\rho(\text{BMM}_{\text{op}})$  contains all  $\text{SC}_{\text{ax}}$  axiomatic executions. Let  $E_{\text{ro}} = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, V, W \rangle \in \text{BMM}_{\text{ro}}$  be an  $\text{SC}_{\text{ax}}$  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 variable w.r.t.  $\xrightarrow{to}$ . We claim that  $E_{\text{op}} = (P, tr, V, W)$  can be build, with  $tr \downarrow_{\mathbb{A}} = \xrightarrow{to}$ , and that  $E_{\text{op}} \in \text{BMM}_{\text{op}}(P)$ . Silent actions are inserted in  $\xrightarrow{to}$  so that each write action is immediately unbuffered, and that  $tr$  in intra-thread consistent – an equivalent condition was required for  $E_{\text{ro}} \in \text{BMM}_{\text{ro}}(P)$ . Finally  $\rho(E_{\text{op}}) = E_{\text{ro}}$ .

Let us show that  $\rho(\text{BMM}_{\text{op}})$  is backward-closed by WR and WR\* $\mathcal{R}$ . Let  $E_{\text{ro}}$  and  $E_{\text{ro}}$  two well-formed axiomatic executions such that  $E'_{\text{ro}} \in \rho(\text{BMM}_{\text{op}})$

and  $E_{ro} \xrightarrow{\Phi} E'_{ro}$ , with  $\Phi \in \{\text{WR}, \text{WR}^*\text{R}\}$ .  $E'_{ro} \in \rho(\text{BMM}_{\text{op}})$  so there exists  $E'_{op} \in \text{BMM}_{\text{op}}$  such that  $E'_{ro} = \rho(E'_{op})$ . In Section 8, we show that WR and WR\* $\text{R}$  are valid transformation under  $\text{BMM}_{\text{op}}$  (Lemmas 5 and 6), meaning that, in both cases, there exists  $E_{op} \in \text{BMM}_{\text{op}}$  such that  $\rho(E_{op}) = E_{ro}$ . Hence,  $E_{ro} \in \rho(\text{BMM}_{\text{op}})$ .

## 7 DRF guarantee of BMM

This section establishes that BMM enjoys the property that any reasonable memory model should have, namely a *data-race-free guarantee*, meaning that data-race free programs only have sequentially consistent executions. The programmer can therefore only think about his programs in terms of interleavings, provided they are correctly synchronised, in this interleaving setting.

Sequential consistent behaviors can be defined in a more intuitive way than in the axiomatic Definition 6. We define them operationally. Following the same approach as  $\text{BMM}_{\text{op}}$ , we define<sup>9</sup> an  $\text{SC}_i$  machine (interleaving SC). Informally, its semantic states are the same than  $\text{BMM}_{\text{op}}$ , except it does not use any buffer: all threads actions are interleaved and performed directly on a shared memory. Thus, the  $\text{SC}_i$  machine ensures that, in any accepted execution  $(P, tr, V, W)$ , all read actions in  $tr$  see the last write to their variable, w.r.t.  $\xrightarrow{tr}$ . We write  $\mathbb{A}_\tau = \mathbb{A} \cup \{\tau_t \mid t \in \mathbb{T}\}$ .

**Definition 14 (SC<sub>i</sub> execution and behavior).** *An operational execution  $E = (P, tr, V, W)$  is  $\text{SC}_i$  if  $tr$  is a properly locked sequence of actions with no unbuffering:  $tr \in \text{list}(\mathbb{A}_\tau)$ , and  $E$  is accepted by the  $\text{SC}_i$  machine. The  $\text{SC}_i$  observable behaviors of a program  $P$  is defined:*

$$\text{Beh}_{sc}(P) = \{(tr \downarrow_{\mathbb{A}_x}, V, W) \mid (P, tr, V, W) \text{ is } \text{SC}_i\}$$

We reuse the  $\text{SC}_i$  definition to define what it means for an operational execution in  $\text{BMM}_{\text{op}}$  to be sequentially consistent:  $E_{op} = (P, tr, V, W) \in \text{BMM}_{\text{op}}$  is said to be  $\text{SC}_{\text{op}}$  if  $(P, tr \downarrow_{\mathbb{A}_\tau}, V, W)$  is  $\text{SC}_i$ , i.e. when forgetting all unbuffering actions in  $tr$  leads to an  $\text{SC}_i$  execution. On top of this  $\text{SC}_i$  memory model, we define as is standard the notions of concurrent conflicting memory action (data-race), and data-race-free programs:

**Definition 15 (Conflicting actions – Data-race – DRF program).**

- Two non-volatile actions  $a, b \in \mathbb{A}_r \cup \mathbb{A}_w$  are conflicting if they target the same address and  $T(a) \neq T(b)$  and at least one of them is a write.
- In an  $\text{SC}_i$  execution  $(P, tr, V, W)$ , two conflicting actions  $a, b$  form a data-race if  $tr$  is of the form  $tr_1 \cdot a :: b \cdot tr_2$ .
- Program  $P$  is data-race free, written  $\text{DRF}(P)$ , if its  $\text{SC}_i$  executions are free of data-race.

<sup>9</sup> A formal definition of the  $\text{SC}_i$  machine can be found in the Coq development.

The DRF guarantee of BMM is stated by the following theorem, the full proof of which is in Appendix B.

**Theorem 3 (DRF guarantee).** *For all  $P$ ,  $DRF(P) \Rightarrow Beh_{op}(P) = Beh_{sc}(P)$ .*

*Proof Sketch.* Let  $P$  be a program such that  $DRF(P)$ . We focus on  $Beh_{op}(P) \subseteq Beh_{sc}(P)$ , and show that  $Beh_{ro}(P) \subseteq Beh_{sc}(P)$ . Let  $b \in Beh_{ro}(P)$ , and  $E_{ro} = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, V, W \rangle \in BMM_{ro}(P)$  the emitting trace. Using Theorem 1, we know that this execution is legal.

We now rely on two theorems of [30]<sup>10</sup>: a DRF guarantee for the JMM and proof of equivalence between the DRF notion this theorem uses and the more intuitive version of it expressed on interleavings. We thus obtain that  $E_{ro}$  is  $SC_{ax}$ , which gives us a total order  $\xrightarrow{to}$  on actions that can be used to build an  $SC_i$  operational execution of  $P$  ( $P, tr', V, W$ ). But  $\xrightarrow{to}$  is consistent with  $\xrightarrow{so}$ , and the value-seen and write-seen are not modified. Hence  $(P, tr', V, W)$  has the same observational behavior than  $E$ .

## 8 Validity of transformations in $BMM_{op}$

One of the objectives of any Java memory model is to take into account the reordering performed by the hardware and to allow compilers to perform some program transformations that deal directly with memory accesses. Starting from the initial work by Ševčík [31, 30] we list in Table 2 various transformations. In each column we indicate their validity under a Sequential Consistent model (SC), and the JMM (revised version proposed in [31]). In the last column, we give the corresponding result for BMM. These results are as expected under a TSO setting, but they demonstrate that, despite its restricted set of reorderings (JMM is more permissive on normal memory accesses reorderings), BMM is relaxed enough to allow most of the transformations listed in the table.

Transformation	SC	JMM	BMM
Trace preserving transformation	✓	✓	✓
Reordering normal memory accesses		✓	(only WR)
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	✓		

Table 2: Validity of transformation in memory models

<sup>10</sup> The precise statements of theorems we use can be found in Appendix B

### 8.1 Valid transformations

We formally show here the validity of the local reorderings WR and WR<sup>\*</sup>R. We refer the reader to the Appendix C for the proof of validity of the other transformations.

**Definition 16 (Valid reordering).** *A local reordering  $\xrightarrow{\Phi}$  between axiomatic executions is said to be valid with respect to  $\text{BMM}_{\text{op}}$  if for all axiomatic execution  $E$  and all operational execution  $E'$ ,  $E \xrightarrow{\Phi} \rho(E')$  and  $E' \in \text{BMM}_{\text{op}}$  implies that there exists  $E'' \in \text{BMM}_{\text{op}}$  such that  $E = \rho(E'')$ .*

**Lemma 5.**  $\xrightarrow{\text{WR}}$  is valid.

*Proof.* Let  $E = \langle P, A, \xrightarrow{p^o}, \xrightarrow{s^o}, V, W \rangle$  be an axiomatic execution and let  $E' = (P', tr', V, W)$  be an operational execution such that  $E \xrightarrow{\text{WR}} \rho(E')$  and  $E' \in \text{BMM}_{\text{op}}$ . By hypothesis, the trace  $tr'$  is of the form  $tr' = \beta \cdot [R_t^j x] \cdot \gamma \cdot [W_t^i y] \cdot \delta$  and  $\gamma$  does not contain any action owned by  $t$  except some unbuffering actions.

The write action  $W_t^i y$  can be performed just after  $R_t^j x$  since the unbuffering in  $\gamma$  are independent of it and no read action in  $\gamma$  can see  $W_t^i y$  (it is still in its buffer). Hence the trace  $\beta \cdot [R_t^j x] \cdot [W_t^i y] \cdot \gamma \cdot \delta$  is still in  $\text{BMM}_{\text{op}}$  for the same value-seen and write-seen functions.

After a swap we obtain a trace  $tr'' = \beta \cdot [W_t^i y] \cdot [R_t^j x] \cdot \gamma \cdot \delta$  that belongs to  $\text{BMM}_{\text{op}}(P)$  (by definition of  $\xrightarrow{\text{WR}}$ ) and  $\rho(P, tr'', V, W) = E$ .  $\square$

**Lemma 6.**  $\xrightarrow{\text{WR}^*\text{R}}$  is valid.

*Proof.* Let  $E = \langle P, A, \xrightarrow{p^o}, \xrightarrow{s^o}, V, W \rangle$  be an axiomatic execution and let  $E' = (P', tr', V, W)$  be an operational execution such that  $E \xrightarrow{\text{WR}^*\text{R}} \rho(E')$  and  $E' \in \text{BMM}_{\text{op}}$ . By hypothesis, the traces  $tr'$  is of the form  $tr' = \beta \cdot [R_t^j x] \cdot \gamma \cdot [W_t^i y] \cdot \gamma_1 \cdot [R_t^{i_1} y] \cdots \gamma_n \cdot [R_t^{i_n} y] \cdot \delta$ . Each  $\gamma, \gamma_1, \dots, \gamma_n$  does not contain any action owned by  $t$  except some unbuffering actions.

As in the previous proof, the write action  $W_t^i y$  can be performed just after  $R_t^j x$ . All read actions  $R_t^{i_1} y, \dots, R_t^{i_n} y$  see the write  $W_t^i y$  in  $tr'$  and then they can also be put forward either. The trace  $\beta \cdot [R_t^j x] \cdot [W_t^i y] \cdot [R_t^{i_1} y] \cdots [R_t^{i_n} y] \cdot \gamma \cdot \gamma_1 \cdots \gamma_n \cdot \delta$  is still in  $\text{BMM}_{\text{op}}(P')$  for the same value-seen and write-seen functions (the moved reads see  $W_t^i y$  directly from  $t$ 's buffer).

We then conclude with a swap: the trace  $tr'' = \beta \cdot [W_t^i y] \cdot [R_t^j x] \cdots [R_t^{i_n} y] \cdot [R_t^j x] \cdot \gamma \cdot \gamma_1 \cdots \gamma_n \cdot \delta$  belongs to  $\text{BMM}_{\text{op}}(P)$  and  $\rho(P, tr'', V, W) = E$ .  $\square$

### 8.2 Invalid transformations

We argue that the reordering nature of BMM also helps understanding why certain transformations are invalid under a TSO hardware memory model. The following example demonstrates this point, in the case of redundant write after read elimination which is invalid [11]. Here  $v$  is a volatile address.

$$\begin{array}{ccc}
\frac{x \leftarrow 0; y \leftarrow 0}{\begin{array}{l} r_1 \leftarrow x \\ y \leftarrow 1 \\ x \leftarrow r_1 \\ r_3 \leftarrow x \end{array}} \parallel \frac{x \leftarrow 1 \\ v \leftarrow 0 \\ r_2 \leftarrow y}{\begin{array}{l} x \leftarrow 1 \\ v \leftarrow 0 \\ r_2 \leftarrow y \end{array}} & \xrightarrow[\text{after read elim.}]{\text{redundant write}} & \frac{x \leftarrow 0; y \leftarrow 0}{\begin{array}{l} r_1 \leftarrow x \\ y \leftarrow 1 \\ r_3 \leftarrow x \end{array}} \parallel \frac{x \leftarrow 1 \\ v \leftarrow 0 \\ r_2 \leftarrow y}{\begin{array}{l} x \leftarrow 1 \\ v \leftarrow 0 \\ r_2 \leftarrow y \end{array}} \\
r_1 = r_2 = 0 \\ \text{and } r_3 = 1 \text{ is invalid} & & r_1 = r_2 = 0 \\ \text{and } r_3 = 1 \text{ is valid}
\end{array}$$

On the left, we can easily understand that the execution is not valid because it is not sequentially consistent and no reordering WR or even WR\*R is possible.

After the redundant write elimination, the execution is valid because the read  $r_3 \leftarrow x$  can be reordered with the write  $y \leftarrow 1$  to give a sequentially consistent execution. This transformation thus introduced new behaviours (it is invalid).

## 9 Related Work

Relaxed memory model has been studied for decades. Early surveys [1, 2] outlines a range of hardware memory models and some of them have been formalized [14, 25, 23]. However, higher level memory models are required for high level programming languages, taking into account both hardware memory models and compiler optimization reordering.

Language level memory models, on the other hand, have been proposed with the purpose of helping programmers in reasoning about possible program executions and directing compilers in exploring valid program transformations. The first proposed DRF model [3] guarantees sequential consistency for data-race free programs while leaving the semantics of the racy ones undefined. The C++ memory model [8, 7], known as C++11, inherits DRF. Paper [6] formally proves the correctness of the compilation scheme from C++11 model to POWER hardware memory model.

Although easy to understand, the DRF model leaves the semantics of racy programs completely undefined, and this is unacceptable for the languages that promise strong safety, such as Java. The JMM [21] extends DRF with the intention for providing a full semantics, while still allowing as many optimizations as possible. However, the current JMM definition is subtle, complex, and formally broken. Inspired by the counter example found in [12], Ševčík *et al.* [31] confirm that the JMM is incapable of justifying some of the program transformations that it intends to allow. They propose in [4] an alternative definition of the JMM that does not suffer from these issues.

Moreover, for a compiler verification perspective, an operation definition of the JMM is desirable. There are several attempts to provide such *weak operational semantics* [12, 9, 10, 15] for programming languages but none of them has ever been used in a proof assistant. Our operational  $\text{BMM}_{\text{op}}$  semantics is inspired by the C-light TSO memory model proposed in [26] which has been formalized in Coq and used to prove non trivial program transformations.

Another line of work is to capture the valid set of program transformations under given memory models. Papers [19, 29] adapt traditional sequential program optimization algorithms to be memory model aware. Shasha *et al.* [28] describe a set of SC-allowable reordering rules. Ševčík [30] focuses on DRF model. They prove that a number of program transformations are safe under data-race-free assumption in the sense that they cannot introduce the out-of-the-thin-air error. Marino *et al.* [22] propose a program transformation that enables program speculative execution and dynamic recovery upon SC-violation such that all conventional optimizations remain applicable. Burckhardt *et al.* [11] design a technique that can verify the validity of given local program transformation under given memory model.

## 10 Conclusion and Future Work

This work presents BMM, an alternative memory model for Java. The axiomatic characterization of this model, useful for programmers, is defined using vocabulary similar to what is used in the JMM but avoids the complex notion of race committing sequence. While being not as rich as the JMM, it still permits a number of important optimizations and it has a tractable definition and intuitive semantics, easy for programmers to understand. Last but not least, it has a clear operational formulation, useful for compilers, defined in terms of per-thread store buffers as found in a typical definition of TSO. We also provide a proof of equivalence between the axiomatic and the operational definitions.

This semantics is the first step to achieve a further goal: developing a verified compiler infrastructure for Java. The main remaining step is to provide a compiler bridge between the Java BMM and TSO hardware semantics. Our model prevents the compiler to perform some memory access reorderings, and one could fear a performance overhead on generated programs. But, as Marino *et al.* [22] experimentally demonstrated, such an overhead is already very reasonable when restricting a compiler to preserve SC. Moreover, critical systems reliability requires to find the right balance between performance of the generated code and tractability of a machine-checked proof of the correctness of the compiler.

## References

1. S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, dec 1996.
2. S. V. Adve and M. Hill. A unified formalization of four shared-memory models. *Parallel and Distributed Systems, IEEE Transactions on*, 4(6):613–624, jun 1993.
3. S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th annual international symposium on Computer Architecture, ISCA '90*, 1990.
4. D. Aspinall and J. Ševčík. Formalising Java’s data race free guarantee. In *Proc. of TPHOLs'07*, pages 22–37. Springer, 2007.
5. D. Aspinall and J. Ševčík. Java Memory Model Examples: Good, Bad and Ugly. In *Proc. of VAMP'07*, 2007.
6. M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to power. In *Proc. of POPL'12*, 2012.

7. M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proceedings of POPL '11*, 2011.
8. H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. *SIGPLAN Not.*, 43:68–78, 2008.
9. G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *Proc. of POPL 2009*, pages 392–403. ACM, 2009.
10. G. Boudol and G. Petri. A theory of speculative computation. In *Proc. of ESOP 2010*, volume 6012 of *LNCS*, pages 165–184. Springer, 2010.
11. S. Burckhardt, M. Musuvathi, and V. Singh. Verifying local transformations on relaxed memory models. In *Proc. of CC 2010*, 2010.
12. P. Cenciarelli, A. Knapp, and E. Sibilio. The Java Memory Model: Operationally, denotationally, axiomatically. In *Proc. of ESOP '07*, 2007.
13. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley Longman, 2006.
14. L. Higham, J. Kawash, and N. Verwaaland. Defining and comparing memory consistency models. In *Proc. of PDCS 1997*, pages 349–356, 1997.
15. R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. volume 6012 of *LNCS*, pages 307–326. Springer, 2010.
16. Jsr. Jsr-133: Java memory model and thread specification, 2004.
17. 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):619–695, 2006.
18. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
19. J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of PPOPP '99*, 1999.
20. X. Leroy. A formally verified compiler back-end. *JAR*, 43(4), 2009.
21. J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proc. of POPL'05*, pages 378–391. ACM, 2005.
22. D. Marino, A. Singh, T. D. Millstein, M. Musuvathi, and S. Narayanasamy. A case for an sc-preserving compiler. In *Proc. of PLDI 2011*, pages 199–210. ACM, 2011.
23. S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In *Proceedings of TPHOLs '09*, 2009.
24. S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding power multiprocessors. In *Proc. of PLDI 2011*, 2011.
25. 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 *Proceedings of POPL '09*, 2009.
26. J. Ševčík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *Proc. of POPL 2011*, 2011.
27. 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):89–97, 2010.
28. D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10:282–312, April 1988.
29. Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler techniques for high performance sequentially consistent java programs. In *Proceedings of PPOPP '05*, 2005.
30. J. Ševčík. *Program Transformations in Weak Memory Models*. PhD thesis, The University of Edinburgh, 2009.
31. J. Ševčík and D. Aspinall. On validity of program transformations in the Java Memory Model. In *Proceedings of ECOOP'08*, pages 27–51. Springer, 2008.

## A $\text{BMM}_{\text{op}}$ and $\text{BMM}_{\text{ro}}$ are equivalent

In this section, we change slightly the notation convention for actions. To lighten the notation, we keep implicit the unique identifier of actions, as it is clear from the context that they are all unique, and we will write  $W_x^t$  for  $W_t^i x$ .

**Lemma 7.** *Let  $E_{\text{op}} = (P, tr, V, W) \in \text{BMM}_{\text{op}}(P)$ , with*

$$tr = \alpha \cdot [W_x^t] \cdot \beta$$

*an execution such that  $B(W_x^t) \notin \beta$ .*

*We note*

- $\mathcal{W}^t = \{W_y^t \in \beta \mid y \in \mathbb{X}\}$  *the set of write actions in  $\beta$  that belong to thread  $t$ .*
- $\mathcal{R}^t = \{R_y^t \in \beta \mid W_x^t \cdot \beta = \beta_1 \cdot W_y^t \cdot \beta_2 \cdot R_y^t \cdot \beta_3, y \in \mathbb{X}, \beta_1, \beta_2, \beta_3 \in \text{list}(\mathbb{A} \cup \mathbb{A}_{\text{sil}})\}$  *the set of read actions in  $\beta$  that see a write performed by thread  $t$  in  $[W_x^t] \cdot \beta$ .*
- $\beta \setminus (\mathcal{W}^t \cup \mathcal{R}^t)$  *the remaining actions in  $\beta$ .*

*Then, there exist  $P', \beta_1, \beta_2$  such that  $E'_{\text{op}} = (P', tr', V, W) \in \text{BMM}_{\text{op}}(P')$ ,  $E_{\text{op}} \xrightarrow{\text{RO}} E'_{\text{op}}$  and*

$$tr' = \alpha \cdot \beta_1 \cdot [W_x^t] \cdot \beta_2$$

- $\beta_1 = \beta \downarrow_{\beta \setminus (\mathcal{W}^t \cup \mathcal{R}^t)}$
- $\beta_2$  *contains the elements of  $\mathcal{W}^t \cup \mathcal{R}^t$*
- $[W_x^t] \cdot \beta_2$  *matches the pattern  $(W_{x_1}^t; (R_{x_1}^t)^*) \cdot \dots \cdot (W_{x_n}^t; (R_{x_n}^t)^*)$*
- *for all trace  $\delta$ , if  $(P, tr \cdot \delta, V, W) \in \text{BMM}_{\text{op}}(P)$  then  $(P', tr' \cdot \delta, V, W) \in \text{BMM}_{\text{op}}(P')$ .*

*Proof.* Let  $E_{\text{op}} = (P, tr, V, W) \in \text{BMM}_{\text{op}}(P)$ , with  $tr = (\alpha \cdot [W_x^t] \cdot \beta)$  and  $B(W_x^t) \notin \beta$ . The sub-trace  $[W_x^t] \cdot \beta$  can be decomposed as  $[W_x^t] \cdot \beta = \beta_p \cdot \beta_t$  with  $\beta_p = [W_y^t; (R_y^t)^*]^+$  (we take the shortest  $\beta_t$ ). We now proceed by induction on the length of  $\beta_t$ .

- Base case:  $\beta_t$  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  $\beta_p$  or (ii) be moved before this pattern:
  - If  $a = R_k^{t''}$ , where  $t'' \neq t$ . It could not see any of the write actions in  $\beta_p$ , because they are all in  $t$ 's buffer. Thus, only changing the interleaving, we can obtain  $(P, \alpha \cdot [a; W_x^t] \cdot \beta_p \cdot \beta'_t, V, W)$  that is also in  $\text{BMM}_{\text{op}}(P)$ .
  - If  $a = R_k^t$ , when  $k$  is one of the variable of  $\beta_p$ . Here, we integrate in  $\beta_p$  by applying an  $\text{WR}^*\text{R}$  as many times as needed to make it part of the right  $W_k^t(R_k^t)^*$  pattern such that  $\beta_p = \beta_{p_1} \cdot (W_k^t(R_k^t)^*) \cdot \beta_{p_2}$ . Note  $\text{WR}^*\text{R}$  can be applied because the pattern only concerns thread  $t$ , and the visibility conditions required by  $\text{WR}^*\text{R}$  are fulfilled. The right-most such pattern before  $a$ , i.e. such that there is no  $W_k^t$  in  $\beta_{p_2}$ , is the only

one that can ensure  $W$  to be preserved, according to  $\text{BMM}_{\text{op}}$ . Thus the resulting trace  $(P', \alpha \cdot \beta_{p_1} \cdot (W_k^t(R_k^t)^*a) \cdot \beta_{p_2} \cdot \beta'_t, V, W)$  is  $\text{BMM}_{\text{op}}$ .

Being able to apply  $\text{WR}^*\text{R}$  requires some extra precautions. We must make sure that there exists a program  $P'$  whose intra-thread semantics accepts the reordering and we must prove the resulting trace is still in  $\text{BMM}_{\text{op}}$ . The first point is an assumption we have to made on the abstract notion of program: after some local variable renaming and loop enrolling it is always possible to perform such a reordering on independent write/read accesses. The second point is easy to prove since the read we move can keep the same write-seen<sup>11</sup>.

- If  $a = R_k^t$ , when  $k$  is not any of the variable of  $\beta_p$ . We apply the same reasoning as in the previous case, rewriting the trace with  $\text{WR}^*\text{R}$ , but in this case, this simply amounts to put  $a$  before  $W_x^t$ , because  $\text{WR}^*\text{R}$  will be applied on the whole  $\beta_p$  (and the value-seen  $W$  is trivially kept valid).
- If  $a = W_k^{t''}$ , with  $t'' \neq t$ . Then changing the interleaving between  $t''$  and  $t$  leads to  $(P, \alpha \cdot [a] \cdot \beta_p \cdot \beta'_t, V, W)$  that is easily shown to be in  $\text{BMM}_{\text{op}}(P)$ :
  - \* if  $k$  is not any of the variables involved in the pattern, the write-seen function  $W$  keeps trivially valid in  $\text{BMM}_{\text{op}}$
  - \* otherwise, the write-seen function is still preserved, as any write-action in  $\beta_p$  cannot overwrite  $a$ : in fact, all of them are in  $t$ 's buffer all along  $\beta$ .

For any trace  $\delta$ , if  $(P, \alpha \cdot \beta_p \cdot [a] \cdot \beta'_t \cdot \delta, V, W) \in \text{BMM}_{\text{op}}(P)$  then the trace  $(P, \alpha \cdot [a] \cdot \beta_p \cdot \beta'_t \cdot \delta, V, W) \in \text{BMM}_{\text{op}}(P)$  holds too: any read in  $\delta$  that sees  $a$  in the first trace can still see it in the second trace because no write action in  $\beta_p$  is unbuffered before  $\delta$ .

- If  $a = \text{B}(a')$ , then  $a' \in \alpha$  because there is no unbuffering action in  $\beta_p$ . 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  $\beta_p$ .

For any trace  $\delta$ , if  $(P, \alpha \cdot \beta_p \cdot [a] \cdot \beta'_t \cdot \delta, V, W) \in \text{BMM}_{\text{op}}(P)$  then the trace  $(P, \alpha \cdot [a] \cdot \beta_p \cdot \beta'_t \cdot \delta, V, W) \in \text{BMM}_{\text{op}}(P)$  holds too: any read in  $\delta$  that sees  $a'$  in the first trace can still see it in the second trace because no write action in  $\beta_p$  is unbuffered before  $\delta$ .

- If  $a \in \mathbb{A}_s$ , then  $T(a) \neq t$ , otherwise,  $t$ 's buffer would be empty, which contradicts the hypothesis on  $\text{B}(W_x^t)$ . In this case, we simply change the interleaving between  $T(a)$  and  $t$  and conclude as in previous cases.

In each of the resulting traces, we can easily show that we are under the induction hypothesis premises. Hence, we conclude the proof by induction.

<sup>11</sup> It should not be confused with the fact the a  $RW$  transformation is invalid under BMM: here we pick an interleaving trace  $\alpha \cdot [W; R] \cdot \gamma$  where the read and the write are adjacent. To prove the (wrong) validity of a  $RW$  reordering we would start from a trace  $\alpha \cdot [W] \cdot \beta \cdot [R] \cdot \gamma$  where some interleaved actions of other threads occur between the write and the read. It would not be possible to prove then that  $\alpha \cdot [R] \cdot \beta \cdot [W] \cdot \gamma$  is still in  $\text{BMM}_{\text{op}}$ .

□

**Definition 17 (Write-Swapping).** *Let  $tr$  and  $tr'$  two sequences of actions. The write-swapping property holds on  $(tr, tr')$  if for any write action  $w_1, w_2$  to the same address,*

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

**Lemma 8.** *Let  $E_{op} = (P, tr, V, W) \in \text{BMM}_{op}(P)$ . Then there exist  $P', tr'$  such that  $E_{op} \xrightarrow{RO} (P', tr', V, W)$ , with  $(P', tr', V, W) \in \text{BMM}_{op}(P')$  is  $\text{SC}_\alpha$  and the write-swapping property holds on  $(tr, tr')$ .*

*Proof.* We prove the lemma by strong induction on the size of the execution  $|tr| = n$ . We assume the property holds for any integer  $k$  such that  $k < n$ . Let  $E_{op} = (P, tr, V, W) \in \text{BMM}_{op}(P)$  an execution of size  $n$ . We can assume  $n > 0$  because the case  $n = 0$  holds trivially when there is no action in the trace.

$tr$  is of the form  $tr = tr_1 \cdot [a]$ . We can apply the induction hypothesis on  $tr_1$  of length  $n-1$ . It gives us a program  $P_2$  and a trace  $tr_2$  such that  $(P, tr_1, V, W) \xrightarrow{RO} (P_2, tr_2, V, W)$ , with  $E_{op}^2 = (P_2, tr_2, V, W) \in \text{BMM}_{op}(P_2) \cap \text{SC}_\alpha$ , plus a write-swapping property on  $(tr_1, tr_2)$ .

If action  $a$  is not a read action,  $E_{op}^2$  can be extended to a trace  $(P_2, tr_2 \cdot [a], V, W)$  that is still in  $\text{BMM}_{op}(P_2) \cap \text{SC}_\alpha$  and the write-swapping property holds on  $(tr_1 \cdot [a], tr_2 \cdot [a])$ .

Otherwise  $tr$  is of the form  $tr = tr_1 \cdot [R_x^{t_3}]$ .  $(P_2, tr_2 \cdot [R_x^{t_3}], V, W)$  is in  $\text{BMM}_{op}(P_2)$  but we don't know yet if it is in  $\text{SC}_\alpha$  (the write-swapping property holds at this stage anyway). If  $W(R_x^{t_3})$  is the last write action to  $x$  in  $tr_2$  we can extend the  $\text{SC}_\alpha$  property from  $tr_2$  to  $tr_2 \cdot [R_x^{t_3}]$  and conclude directly.

Otherwise, we continue with a split case:

**Case 1:** there exists a thread  $t$ , distinct from  $t_3$  whose buffer is not empty at the end of  $tr_2$ . Formally, it means that there exists a write action  $W_y^t$  in  $tr_2$  such that  $B(W_y^t) \notin tr_2$ . Let us write  $tr_2$  as:

$$tr_2 = \alpha \cdot [W_y^t] \cdot \beta \cdot [R_x^{t_3}]$$

Here, we can apply lemma 7 on  $W_y^t$  and  $\beta \cdot [R_x^{t_3}]$ . We obtain then a program  $P_3$ , a trace  $tr_3$  such that  $E_{op}^2 \xrightarrow{RO} (P_3, tr_3, V, W)$  with  $E_{op}^3 = (P_3, tr_3, V, W) \in \text{BMM}_{op}(P_3)$  and  $tr_3$  is of the form

$$tr_3 = \alpha \cdot \beta_1 \cdot [R_x^{t_3}] \cdot [W_y^t] \cdot \beta_2$$

with  $[W_y^t] \cdot \beta_2$  which matches the pattern  $[W^t; (R^t)^*]^+$ .

Since  $\alpha \cdot \beta_1 \cdot [R_x^{t_3}]$  is a trace of length strictly less than  $n$  we can apply our induction hypothesis on it. We obtain a program  $P_4$  and a trace  $tr_4$  such that  $(P_3, \alpha \cdot \beta_1 \cdot [R_x^{t_3}], V, W) \xrightarrow{RO} (P_4, tr_4, V, W)$ , with  $E_{op}^4 = (P_4, tr_4, V, W) \in \text{BMM}_{\text{op}}(P_4) \cap \text{SC}_\alpha$ , plus a write-swapping property on  $(\alpha \cdot \beta_1 \cdot [R_x^{t_3}], tr_4)$ . We can concatenate the suffix  $[W_y^t] \cdot \beta_2$  to extend  $tr_4$  to an execution  $(P_4, tr_4 \cdot [W_y^t] \cdot \beta_2, V, W) \in \text{BMM}_{\text{op}}(P_4) \cap \text{SC}_\alpha$ . The sequential consistency holds because of the special pattern of  $[W_y^t] \cdot \beta_2$ . The write-swapping is also preserved, by concatenation, between  $(tr_3, tr_4 \cdot [W_y^t] \cdot \beta_2)$ .

**Case 2:** for all thread  $t$ , distinct from  $t_3$ , its buffer is empty at the end of  $tr_2$ . Formally, it means that for every write action  $W_y^t \in tr_2$  such that  $t \neq t_3$ ,  $B(W_y^t) \in tr_2$ .

Let note  $W_x^{t_1} = W(R_x^{t_3})$  the write seen by  $R_x^{t_3}$ .

**Case 2.1:**  $B(W_x^{t_1}) \notin tr_2$ . Under our current assumptions, this implies that  $t_1 = t_3$ . Let us write  $tr_2$  like the following:

$$tr_2 = \alpha \cdot [W_x^{t_1}] \cdot \beta \cdot [R_x^{t_3}]$$

We can apply lemma 7 on  $W_x^{t_1}$  and  $\beta$ . The trace  $\alpha \cdot \beta_1 \cdot [W_x^{t_1}] \cdot \beta_2$  we obtain is in  $\text{BMM}_{\text{op}}$  and looking at the shape of  $\beta_1$  and  $\beta_2$  it is even in  $\text{SC}_\alpha$ . To conclude we must show that  $W_x^{t_1}$  is now the most recent write to  $x$  in  $\beta_2$ . But any other write there would be a write from thread  $t_1$  and  $R_x^{t_3}$  could then not see  $W_x^{t_1}$ .

**Case 2.2:**  $B(W_x^{t_1}) \in tr_2$ . Let us write  $tr_2$  like the following:

$$tr_2 = \alpha \cdot [W_x^{t_1}] \cdot \beta \cdot [B(W_x^{t_1})] \cdot \gamma \cdot [R_x^{t_3}]$$

Note that any conflicting  $W_x^{t_2}$  that would be more recent than  $W_x^{t_1}$  can't appear in  $\gamma$  because either it is unbuffered in  $\gamma$  and it would overwrite  $W_x^{t_1}$  or it is not unbuffered but then  $t_2 = t_3$  and  $W_x^{t_1}$  could not be seen by  $R_x^{t_3}$ . Note also that any unbuffering of the form  $B(W_x^{t_2})$  can neither appear in  $\gamma$  since it would overwrite  $W_x^{t_1}$ . At last, note that any  $W_x^{t_2}$  in  $\beta$  must satisfy  $t_1 \neq t_2$ . Otherwise since such a write can't be unbuffered in  $\gamma$  (previous remark) we would again have  $t_2 = t_3$  and  $W_x^{t_1}$  could not be seen by  $R_x^{t_3}$ .

Now, if we use lemma 7 on  $W_x^{t_1}$  and  $\beta$ , it gives us a trace  $tr_3$  of the form

$$tr_3 \cdot [B(W_x^{t_1})] \cdot \gamma \cdot [R_x^{t_3}] = \alpha \cdot \beta_1 \cdot [W_x^{t_1}] \cdot \beta_2 \cdot [B(W_x^{t_1})] \cdot \gamma \cdot [R_x^{t_3}]$$

In this trace, the most recent write to  $x$  is now  $W_x^{t_1}$ . Any other write to  $x$  in  $\beta_2$  would be a write to  $x$  performed by  $t$  in  $\beta$  (see the previous remark). The subtrace  $\alpha \cdot \beta_1 \cdot [W_x^{t_1}] \cdot \beta_2 \cdot [B(W_x^{t_1})]$  is sequentially consistent because  $\alpha \cdot [W_x^{t_1}] \cdot \beta \cdot [B(W_x^{t_1})]$  was. Still, some reads in  $\gamma$  may not see now the most recent write to their target address. The proof can however be concluded by using the induction hypothesis on  $tr_3 \cdot [B(W_x^{t_1})] \cdot \gamma$ . This rebuilds a SC execution. We may fear such reordering invalidates the fact that  $W_x^{t_1}$  is the most recent write to  $x$  but the write-swapping property ensures it can't. Indeed, if any  $W_x^{t_0}$  action has been swapped with  $W_x^{t_1}$

during this reordering, the unbuffering  $B(W_x^{t_0})$  should appear after  $W_x^{t_1}$  in  $tr_3$ : in  $\beta_2$  or in  $\gamma$ . But we have already observed that such an action can't appear in  $\gamma$  and there is no unbuffering in  $\beta_2$ .

□

## B BMM is a DRF model

**Theorem 4 (DRF guarantee).** *For all  $P$ ,  $DRF(P) \Rightarrow Beh_{op}(P) = Beh_{sc}(P)$ .*

*Proof.*

- Let us show that  $Beh_{sc}(P) \subseteq Beh_{op}(P)$ . Let  $(P, tr \downarrow_{\mathbb{A}_x}, V, W) \in Beh_{sc}(P)$ . We show that there exists  $tr'$  such that  $tr' \downarrow_{\mathbb{A}_x} = tr \downarrow_{\mathbb{A}_x}$  and  $(P, tr', V, W) \in BMM_{op}(P)$  is  $SC_{op}$ . We can build  $tr' \in \text{list}(\mathbb{A} \cup \mathbb{A}_{sil})$  from  $tr$  by inserting after each action in  $tr$  an unbuffering action from the same owner thread. The resulting trace is still properly locked and coherent with the intra-thread semantics. Finally,  $V$  and  $W$  are those obtained by running the  $BMM_{op}$  machine on  $tr'$ . In fact, unbufferings has no impact on the intra-thread semantics and the interleaving. Hence  $V$  can also be built by  $BMM_{op}$ . Moreover, the systematic unbuffering ensures that all buffers are empty at each step, thus implying that all threads have a coherent view of the memory (there is no ambiguity when reading an address). Finally  $tr \downarrow_{\mathbb{A}_x} = tr' \downarrow_{\mathbb{A}_x}$  because the interleaving and  $W$  are preserved. This proves the first inclusion.
- It remains to show that  $Beh_{op}(P) \subseteq Beh_{sc}(P)$ . Let  $P$  be a program such that  $DRF(P)$ , we show that  $Beh_{op}(P) \subseteq Beh_{sc}(P)$ . We will in fact show that  $Beh_{ro}(P) \subseteq Beh_{sc}(P)$ . Let  $b \in Beh_{ro}(P)$ , and  $E_{ro} = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, V, W \rangle \in BMM_{ro}(P)$  the emitting trace. Using Theorem 1, this execution is legal. We now rely on theorems of [30]. The first one is the DRF guarantee of JMM:

*Let  $P$  be a program such that for all of its  $SC_{ax}$  executions, the execution order  $\xrightarrow{hb}$  is total on conflicting actions. Then any legal execution of  $P$  is  $SC_{ax}$ .*

The second one (Corollary 2.14 in [30]) shows the equivalence between the DRF notion this theorem uses and the more intuitive version of it expressed on interleavings – the so-called weakly data-race freeness :

*Let  $P$  be a program. Then  $P$  is weakly data-race-free if and only if for all of its JMM  $SC_{ax}$  executions  $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, V, W \rangle$ , for*

*all pairs  $a, b$  of conflicting actions, either  $a \xrightarrow{hb} b$  or  $b \xrightarrow{hb} a$ .*

We then show that the notion of DRF expressed on interleaving coincides with Definition 15.

Using these results, we obtain that  $E_{ro}$  is  $SC_{ax}$ , i.e. there exists a total order  $\xrightarrow{to}$  on  $A$ , that is consistent with  $\xrightarrow{po}$  and  $\xrightarrow{so}$ , and for each read action  $r \in A$  accessing an address  $x$ ,  $W(r)$  is the last write on  $x$  before  $r$  in  $\xrightarrow{to}$ .

From this total order, we build an  $\text{SC}_i$  operational execution  $(P, tr', V, W)$ . The action trace  $tr'$  is such that for all  $a, b$ ,  $a \xrightarrow{tr'} b$  if  $a \xrightarrow{to} b$  and some  $\tau$  actions are inserted so that  $(P, tr', V, W)$  is  $\text{SC}_i$  (the proper locking is satisfied thanks to the consistency of  $\xrightarrow{to}$  with  $\xrightarrow{so}$  and to the well-formedness of  $E_{ro}$ ). But  $\xrightarrow{to}$  is consistent with  $\xrightarrow{so}$ , and the value-seen and write-seen are not modified:  $(P, tr', V, W)$  and  $E_{ro}$  have the same observational behavior.

□

## C Valid reordering

Let  $E' = (P', tr', V, W)$  an operational execution. For each transformation to be proved valid we observe the shape of  $tr'$  and provide a corresponding  $\text{BMM}_{\text{op}}$  trace of an untransformed program. All read and write actions mentioned in this section operate on non-volatile addresses.

*Redundant read after read elimination*

$$tr' = \beta \cdot [R_t^i x] \cdot \gamma$$

The trace  $\beta \cdot [R_t^i x] \cdot [R_t^j x] \cdot \gamma$  is still in  $\text{BMM}_{\text{op}}$ , assuming that the intra-thread semantics allows a redundant read to be inserted. The read  $R_t^j x$  can see the same write as  $R_t^i x$ .

*Redundant read after write elimination*

$$tr' = \beta \cdot [W_t^i x] \cdot \gamma$$

The trace  $\beta \cdot [W_t^i x] \cdot [R_t^j x] \cdot \gamma$  is still in  $\text{BMM}_{\text{op}}$ , assuming that the intra-thread semantics allows a redundant read to be inserted. The read  $R_t^j x$  can directly see the write  $W_t^i x$  that is still in its buffer.

*Irrelevant read elimination*

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

The trace  $\beta \cdot [R_t^i x] \cdot \gamma$  is still in  $\text{BMM}_{\text{op}}$ , assuming that the intra-thread semantics allows a redundant read to be inserted. The read  $R_t^j x$  can see any write.

*Irrelevant read introduction*

$$tr' = \beta \cdot [R_t^i x] \cdot \delta$$

The trace  $\beta \cdot \gamma$  is still in  $\text{BMM}_{\text{op}}$ , assuming that the intra-thread semantics allows to remove the read  $R_t^j x$ . It does not affect visibility of the remaining actions.

*Redundant write before write elimination*

$$tr' = \beta \cdot [W_t^j y] \cdot \gamma$$

We have to distinguish two cases.

1.  $\gamma = \gamma_1 \cdot B(W_t^j y) \cdot \gamma_2$   
 The trace  $\beta \cdot [W_t^i y] \cdot [W_t^j y] \cdot \gamma_1 \cdot B(W_t^i y) \cdot B(W_t^j y) \cdot \gamma_2$  is still in  $\text{BMM}_{\text{op}}$ , assuming that the intra-thread semantics allows to add a redundant write. The inserted write is given the same value-seen that  $W_t^j y$ . The added unbuffering allow to maintain the visibility of  $W_t^j y$  since  $W_t^i y$  is never seen.
2.  $B(W_t^j y) \notin \gamma$   
 The trace  $tr' = \beta \cdot [W_t^i y] \cdot [W_t^j y] \cdot \gamma$  is still in  $\text{BMM}_{\text{op}}$ , assuming that the intra-thread semantics allows to add a redundant write. The inserted write is given the same value-seen that  $W_t^j y$ .  $W_t^i y$  is never seen.

## D Formal Definition of a Program

Here is the formal definition of a program that we use in the Coq development. It summarizes all the properties that are required about a program and its intra-thread semantics. The other definitions we have formalized can be found at <http://r.cs.purdue.edu/BMM/> along with a concrete instantiation of this abstract data-type. The concrete definition of a program relies on an intermediate representation suitable for a Java compiler.

**Module Type PROGRAM.**

*Threads* We assume a type  $TT$  of threads.

**Parameter  $TT$  : Type.**

And a map structure with threads as keys.

**Declare Module TTMAP : TREE with Definition elt := TT.**

*Memory* We assume a type  $XX$  of addresses.

**Parameter  $XX$  : Type.**

And a map structure with addresses as keys.

**Declare Module XXMAP : TREE with Definition elt := XX.**

For each address, we can determine if it is volatile.

**Parameter isVolatile : XX → bool.**

Some volatile addresses are *external*. A write action to such an address models an external action.

**Parameter isExternal : XX → bool.**

**Axiom external\_is\_volatile : ∀ x:XX, isExternal x → isVolatile x.**

*Locks* We assume a type  $LL$  of locks.

Parameter  $LL$  : Type.

And a map structure with locks as keys.

Declare Module  $LLMAP$  : TREE with Definition  $elt := LL$ .

*Values* We assume a non empty type  $val$  of dynamic values.

Parameter  $val$  : Type.

Parameter  $anyval$ :  $val$ .

*Program semantics* We assume a type  $program$  of programs.

Parameter  $program$ : Type.

Such a program has an intra-thread semantics that relates states of a thread. We thus assume a type  $intra\_state$  of states of running threads.

Parameter  $intra\_state$  : Type.

A thread is either ready to run, in some running state or has completed its execution.

Inductive  $state :=$

| Ready

| Running ( $s$ :  $intra\_state$ )

| Done.

Labelled transition relation of intra-thread semantics states.  $step\ p\ t\ s\ l\ s'$  holds whenever in program  $p$ , thread  $t$  can step from state  $s$  to state  $s'$  through a transition labelled by  $l$ .

Parameter  $step$  :  $program \rightarrow TT \rightarrow$

$state \rightarrow (label\ XX\ TT\ LL\ val) \rightarrow state \rightarrow Prop$ .

No transition leads to *Ready*.

Axiom  $thread\_leaves\_ready$  :  $\forall p\ this, \forall s\ l, \neg step\ p\ this\ s\ l\ Ready$ .

Any transition from *Ready* must be labelled by  $St$ .

Axiom  $thread\_ready\_start$  :  $\forall p\ this, \forall l\ s',$

$step\ p\ this\ Ready\ l\ s' \rightarrow \exists v, l = lbl\ (TThreadStart\ this)\ v$ .

Any transition to *Done* must be labelled by  $End$ .

Axiom  $thread\_done\_end$  :  $\forall p\ this, \forall l\ s,$

$step\ p\ this\ s\ l\ Done \rightarrow \exists v, l = lbl\ (TThreadEnd\ this)\ v$ .

No transition steps from *Done*.

Axiom  $thread\_reaches\_done$  :  $\forall p\ this, \forall l\ s', \neg step\ p\ this\ Done\ l\ s'$ .

End PROGRAM.