# An Extended Buffered Memory Model With Full Reorderings

Gurvan Cabon, David Cachera, David Pichardie

# An Extended Buffered Memory Model With Full Reorderings*

Gurvan Cabon Inria/IRISA
David Cachera ENS Rennes/IRISA/Inria
David Pichardie ENS Rennes/IRISA/Inria

**Abstract**

Modern multicore processor architectures and compilers of shared-memory concurrent programming languages provide only weak memory consistency guarantees. A *memory model* specifies which write action can be seen by a read action between concurrent threads. The most well known memory model is the sequentially consistent (SC) model but, to improve performance, modern architectures and languages employ *relaxed* memory models where a read may not see the most recent write that has been performed by other threads. These models come in different formalization styles (axiomatic, operational) and have their own advantages and disadvantages.

In a POPL'13 paper, Demange et al [12], proposed an alternative style that is fully characterized in terms of the reorderings it allows. This Buffered Memory Model (BMM) targets the Java programming language. It is strictly less relaxed than the Java Memory Model. It is shown equivalent to an operational model but is restricted to TSO relaxations.

This paper extends the BMM in order to allow more reorderings. We present the new set of memory event reorderings rules that fully characterize the model and an alternative operational model that is again shown equivalent.

## 1 Introduction

Writing correct programs requires a deep understanding of the underlying programming language semantics, *i.e.* how programs of that language are executed. The techniques and tools used for providing clear semantics to sequential programming languages are now well established. The rise of concurrent software, enabled by current multicore architectures, has called for new models of programming. For the shared memory parallel programming paradigm, the traditional way of defining the semantics of concurrent programs is the sequential consistency (SC) model, introduced by Lamport in 1979 [15]: executions of all threads are interleaved and accesses to shared memory apparently occur in a global-time linear order.

However, mainly for efficiency reasons, multi-threaded programming languages do not follow this simple and intuitive semantics, and rely on weaker memory models that allow more efficient implementations at the expense of exhibiting sometimes additional counter-intuitive behaviours. These weak memory models are used at a software level in high-level languages (e.g. Java or C++) to leverage the current multi-processors architectures such as x86 or PowerPC, on which ensuring sequential consistency would represent too much of a cost. Besides, this allows for aggressive optimizations in the compilation chain of these languages. However, these optimizations are not well understood or justified in a formal semantics. Even if formalizations do exist for software weak memory models [5, 19], realistic compiler optimizations (*e.g.* memory aware common subexpression elimination or redundant lock elimination) are, to this date, out of the reach of these formalization efforts.

*This work was supported by Agence Nationale de la Recherche, grant number ANR-14-CE28-0004 DISCOVER.

The landscape of weak memory semantics is highly diversified. On one hand, there are many ways of expressing the semantics of such models. One can adopt an operational point of view, and write the semantics as an abstract machine. This presents the advantage of being close to the hardware model, but is not necessarily well suited for proving compiler optimizations because of the amount of low-level actions that are involved for each memory action. Axiomatic memory models [4] in contrast rely on partial orders on events. They are more abstract and concise, and they have been used to model a large hierarchy of hardware semantics. They are amenable to computer-aided program verification, but as far as we know they did not capture yet the intuitive correctness proofs that are performed by compiler designers when inventing new aggressive optimizations. Those experts generally explain and justify their program analyses in terms of action reorderings. In standard axiomatic memory models, reorderings are properties of the model, but they do not characterize it.

Demange et al [12], proposed an alternative axiomatic style that is fully characterized in terms of the reorderings it allows. This Buffered Memory Model (BMM) targets the Java programming language, with the overarching goal of constructing a verified compiler for Java. The model provides also an intuitive method to describe valid and invalid program executions. Let us take as an example the following program running two threads in parallel, where the $r_i$ represent local variables, and $C.f, C.g$ shared variables (static fields in Java).

| $C.f = 0;\ C.g = 0$ | |
|---|---|
| $C.f = 1$ | $C.g = 1$ |
| $r_0 = C.g$ | $r_1 = C.f$ |

Under an SC semantics, the outcome $r_0 = 0, r_1 = 0$ is not possible, as an enumeration of the 6 possible interleavings may show. Under BMM (and any other similar TSO-compliant model [21]) new behaviours are allowed for this program. Taking the operational point of view of BMM, we can think of an architecture where each thread owns a private FIFO store buffer. This design follows closely the CompCertTSO [20] semantics. When a thread writes a shared variable with a value, this pair (variable,value) is stored in its buffer rather than directly committed to the shared memory. Stored values may then be unbuffered at any time (but before any synchronization point as lock/unlock operations or volatile accesses). When a thread reads a shared variable, it first searches for the newest reference to that variable in its private buffer, before looking in the global memory if no such exists. In this example, the two writes actions $C.f = 1$ and $C.g = 1$ fill each thread buffers with a store action. If no unbuffering is performed before the read actions, the shared memory is unchanged and still associates 0 to $C.g$ and $C.f$. As a consequence, the outcome $r_0 = 0, r_1 = 0$ is possible under BMM operational semantics.

BMM allows an equivalent but more programming oriented view of the same phenomenon. It consists in considering reorderings of instructions within each thread: a read can be shifted before a write if performed on a different memory address ($WR$ reordering rule). If the composition of reordered threads produces a legal SC execution, then the initial program provides a BMM-compliant execution with the same observable behaviours. On the previous example, the program can be reordered with two $WR$ rule applications in order to obtain the following program.

| $C.f = 0;\ C.g = 0$ | |
|---|---|
| $r_0 = C.g$ | $r_1 = C.f$ |
| $C.f = 1$ | $C.g = 1$ |

This program allows the final result $r_0 = 0, r_1 = 0$ under a SC model, thus the original program allows the same result under BMM, by definition of the reordering view of BMM.

BMM is strictly less relaxed than the Java Memory Model: every program executions that are legal under BMM, are also legal under JMM. Its operational form and its reordering form have been shown equivalent [12].

A major limitation of BMM is that it is not enough relaxed for modern JVM implementation. At the architecture level, implementing a JVM with BMM semantics should be efficient, if we only target x86 architectures that implement TSO relaxations. But at a compiler level, aggressive memory optimizations as common subexpression elimination on memory loads are not allowed by TSO relaxations [20]. Such optimizations require more reordering rules. The *Relaxed-Memory Order* (RMO), described in the SPARC architecture manual V9 [22] and formalized in an axiomatic manner by Higman and Kawash [13], allows for the four reorderings $RR$, $WW$, $RW$ and $WR$ between read ($R$) and write ($W$) actions. $RR$ swaps two reads that operate on different variables. $WW$ does a similar transformation on write actions. $RW$ swaps a write that appears just after a read, on a different variable. $WR$ perform the inverse transformation.

This paper extends BMM with the four reorderings $RR$, $WW$, $RW$ and $WR$. We first define a new multibuffer operational semantics with write/read buffers ($\text{RMO}_{\text{op}}$). We then introduce the $\text{RMO}_{\text{ro}}$ model that characterizes RMO using its four reorderings rules. We then establish the equivalence between $\text{RMO}_{\text{op}}$ and $\text{RMO}_{\text{ro}}$.

## 2 Preliminary definitions and notations

We assume a set $\mathbb{X}$ of variables, a set $\mathbb{T}$ of threads and a set $\mathbb{V}$ of values. A variable is said to be *volatile* when it cannot be subject to local modifications. The formal difference between volatile variables and classic variables will be given when defining the models. The set of inter-thread actions we consider is listed here:

$$\begin{array}{rcll} \mathbb{A} & := & \mathrm{w}_t^i x, v & (t \text{ writes the value } v \text{ to address } x) \\ & | & \mathrm{r}_t^i x & (t \text{ reads the value from address } x) \\ & | & \mathrm{w}_0 x & (\text{default write to address } x) \end{array}$$

We use the following notation for distinguishing different kinds of actions: $\mathbb{A}_w$ for all the write actions, $\mathbb{A}_r$ for the read actions, $\mathbb{A}_d$ for the default write actions, $\mathbb{A}_s$ for the actions on volatile variable (synchronization actions). The default write actions are only used at the beginning of a program to initialize the variables in the memory. If $w \in \mathbb{A}_w$ is a write action of the form $\mathrm{w}_t^i x, v$, we note $V(w)$ the written value $v$.

Because we want to compare models, we have to know what their observable behaviors are. We will give a precise definition later for each considered model, but they all rely on a set of variables that we will call *external*. We assume that any external variable is a volatile variable. We also define the set $\mathbb{A}_{ex}$ as the set of the write actions on the external variables, $\mathbb{A}_{ex} = \{\mathrm{w}_t^i x, v \mid x \text{ is external}\}$.

We could expand our language adding other synchronization actions. For example a fence action that would wait for the thread's buffers to be empty before being able to continue to execute the program. For the sake of simplicity and clarity of this paper, we do not consider such actions, but once correctly defined, they should not cause problem in the proofs.

When a partial order is total, we sometimes write it directly as a sequence; conversely, we will write $a \xrightarrow{1} b$ when $a$ and $b$ are two elements of a list $l$ and they appear in that order in $l$. When a partial order $\xrightarrow{\text{o}}$ is a disjoint union of orders indexed by a set $S$, we will write $[\xrightarrow{\text{o}}]_s$ its restriction on the element $s \in S$. When a list $l$ contains elements from a set $S$, we will write $l \downarrow_T$ the projection of $l$ on a subset $T$ of $S$. When writing down a list or a trace, we will use the "$\cdot$" separator; for example the list containing the integers 1, 2 and 3 is denoted $1 \cdot 2 \cdot 3$. The set of all functions from a set $I$ to a set $J$ is written $I \mapsto J$. For a function f, the notation $f[x \mapsto v]$ represents a function equal to f, excepted for $x$ for which the value returned is $v$.

As in [12], we will not give a formal syntax to programs here. Let us consider that a program consists of a first list of default write actions to initialize the memory, and an ordered list of actions per thread.

We rather start our study directly at a semantic level. Every thread is given an intra-thread semantics as a labelled transition relation

$$\dot{\longrightarrow\!\!\!\!\!\!\rightarrow} \subseteq \mathrm{State}_{\mathrm{intra}} \times \mathrm{Label}_{\mathrm{intra}} \times \mathrm{State}_{\mathrm{intra}}$$

that operates over intra-thread states in $\mathrm{State}_{\mathrm{intra}}$. An example of intra-thread state could be the current call stack of a thread where each stack element is a frame that contains a method $m$, its current program counter in $m$ and the current set of values for each local variable in $m$.

The transition labels belong to the set $\mathrm{Label}_{\mathrm{intra}} = (\mathbb{A}\backslash\mathbb{A}_r) \cup (\mathbb{A}_r \times \mathbb{V}) \cup \{\tau\}$ and each step corresponds to an action in the program. For a step labelled by a read action, the read action is paired with the value that is read. The label $\tau$ is used for actions that are memory irrelevant (manipulating the registers for example). A transition $s \xrightarrow{a}\!\!\!\!\!\!\rightarrow s'$ can be taken if and only if the program can perform action $a$ in state $s$ and the step leads to $s'$. In the set of states $\mathrm{State}_{\mathrm{intra}}$, there is a particular initial state `Ready` with no transition leading to it.

# 3 MultiBuffer operational model

The purpose of the RMO [22] model is to authorize the four basic reorderings mentioned in the introduction: $RW$, $RR$, $WW$ and $WR$. To explain them intuitively, we can take the first one as an example. An $RW$ reordering allows a write action to be executed before a read action preceding it in the sequential order if they target different variables. The meaning of the other reorderings can be found by analogy.

The operational view of this model relies on a multibuffer machine, using FIFO buffers for write and read actions. We aim to build a proof of the equivalence between this operational model ($\mathrm{RMO}_{\mathrm{op}}$) and the reordering model ($\mathrm{RMO}_{\mathrm{ro}}$) that will be described in the next section.

The multibuffer model relies on the hardware architecture illustrated on Figure 1. The threads can store the read and write actions in buffers, one for each *non volatile* variable, in order to execute them later. These actions thus take effect only when they are unbuffered, *i.e.* when the machine takes the action out of the buffer. Read and write actions on volative variables are directly performed on the shared memory and require all buffers of the considered thread to be empty.

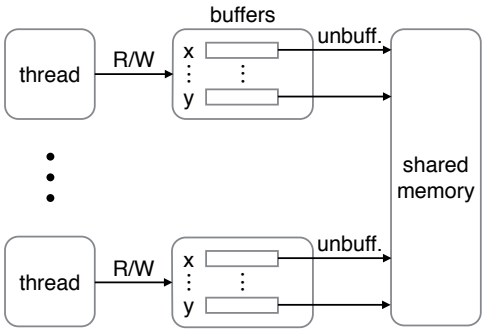



Figure 1: $\mathrm{RMO}_{\mathrm{op}}$ architecture

The machine is defined by a labelled transition system

$$\dot{\rightarrow} \subseteq \mathrm{S}_{\mathrm{MB}} \times \mathrm{LabelMB}_{\mathrm{inter}} \times \mathrm{S}_{\mathrm{MB}}$$

whose transitions are displayed on Figures 2 and 3.

A state $s \in \mathrm{S_{MB}}$ is a triple $(ts, b, m)$ recording the local state of each thread, one buffer per thread and per variable, and the global memory, mapping each address to one write action, as formalized below.

$$\begin{array}{lcl} ts & \in & \mathbb{T} \mapsto \mathrm{State_{intra}} \\ b & \in & (\mathbb{T} \times \mathbb{X}) \mapsto \mathrm{List}(\mathbb{A}_r \times \mathbb{V} \cup \mathbb{A}_w \backslash \mathbb{A}_d) \\ m & \in & \mathbb{X} \mapsto \mathbb{A}_w \end{array}$$

Each step corresponds either to an action generated by the intra-thread semantics, or to the unbuffering of a write or read action, or to an intra-thread silent action. We thus define the set of silent actions, denoted by $\mathbb{A}_{sil}$, as intra-thread silent steps $\tau$ and unbufferings $\overline{\mathrm{B}}(a)$ of a write action $a \in \mathbb{A}_w \backslash \mathbb{A}_d$ or a read action $a \in \mathbb{A}_r$.

$$\mathbb{A}_{sil} = \{\tau\} \cup \{\overline{\mathrm{B}}(a) | a \in \mathbb{A}_w \backslash \mathbb{A}_d\} \cup \{\overline{\mathrm{B}}(a) | a \in \mathbb{A}_r\}$$

The labels on steps are simply the intra-thread actions or the silent actions, except in the case of a read action on a volatile variable, in which we label $(r \mid w)$ meaning that $r$ reads the value written by $w$.

$$\mathrm{LabelMB_{inter}} = \mathbb{A} \cup \mathbb{A}_{sil} \cup ((\mathbb{A}_r \cap \mathbb{A}_s) \times \mathbb{A}_w)$$

Rule [Tau] allows a thread $t$ to do a silent step, nothing is changed in the shared memory but the intra-thread state of thread $t$ is modified. When a thread wants to perform a read action on a non-volatile variable and read the value $v$, rule [R] makes the machine insert the read action in the buffer, paired with the value that will be read later on unbuffering. In a similar way, rule [W] puts the write action on a non-volatile variable in the buffer, without modifying the shared memory $m$. When comes the possibility to get a read action out of the buffer, *i.e.,* when the value that appears in the buffer corresponds to the value in the memory, the [UnBuffR] rule reduces the buffer and emits the label $\overline{\mathrm{B}}(\mathrm{r}_t^i x) \mid w$, where $w$ is the write action that previously committed the considered value to the memory. Getting a write action out of the buffer [UnBuffW] only consists in updating the global memory with the given write action. Finally, the synchronization actions require all the buffers of the considered thread to be empty: a write or read on a volatile variable ([RS] or [WS]) directly performs the action on the memory.

In this operational model, an execution is a trace produced by the labels of the $\mathrm{RMO_{op}}$ machine. The definition below formalizes the constraints on this trace.

**Definition 1 (Multibuffer operational execution)** *A multibuffer operational execution is a pair $(P, tr)$ where $P$ is the program and $tr = a_1 \cdots a_n$ is a finite list of labels in $\mathrm{LabelMB_{inter}}$ such that*

- *no action appears more than once in $tr$,*
- *there exist states $s_0, \ldots, s_n$ of the $\mathrm{RMO_{op}}$ machine, such that $\forall i \in \{1, \ldots, n\}, s_{i-1} \xrightarrow{\mathrm{a_i}} s_i$ and $s_0$ is an initial state:*
  - *in state $s_0$, $ts$ is defined for all the threads, mapping them to the Ready state,*
  - *all the buffers are empty,*
  - *each variable points to the value given by $\mathbb{A}_d$.*

We note $\mathrm{RMO_{op}}$ the set of all the multibuffer operational executions, $\mathrm{RMO_{op}}(P)$ the executions concerning a program $P$ and $\mathcal{O}_{\mathrm{op}}(P)$ the observable behaviors of the program $P$ under $\mathrm{RMO_{op}}$.

$$\mathcal{O}_{\mathrm{op}}(P) = \{tr \downarrow_{\mathbb{A}_{ex}} | \ (P, tr) \in \mathrm{RMO_{op}}(P)\}$$

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

$$\frac{ts(t) \xrightarrow{\mathsf{r}_t^i x|v} \!\!\!\!\rightarrow\, s}{ts, b, m \xrightarrow{\mathsf{r}_t^i x} ts[t \mapsto s], b[(t,x) \mapsto (\mathsf{r}_t^i x \mid v) \cdot b(t,x)], m}[\mathrm{R}]$$

$$\frac{b(t,x) = l \cdot [(\mathsf{r}_t^i x \mid V(w))] \quad w = m(x)}{ts, b, m \xrightarrow{\overline{\mathrm{B}}(\mathsf{r}_t^i x)|w} ts, b[(t,x) \mapsto l], m}[\textsc{UnBuffR}]$$

$$\frac{ts(t) \xrightarrow{\mathsf{w}_t^i x,v} \!\!\!\!\rightarrow\, s}{ts, b, m \xrightarrow{\mathsf{w}_t^i x,v} ts[t \mapsto s], b[(t,x) \mapsto (\mathsf{w}_t^i x, v) \cdot b(t,x)], m}[\mathrm{W}]$$

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

$$\frac{ts, m \xrightarrow{\lambda_t}_{\mathsf{synch}} ts', m' \quad \forall x \in \mathbb{X}, b(t,x) = [\,]}{ts, b, m \xrightarrow{\lambda_t} ts', b, m'}[\textsc{Synch}]$$

Figure 2: $\mathrm{RMO_{op}}$ machine (labelled transition system)

# 4 Reordering memory model

We now provide an alternative model $\mathrm{RMO_{ro}}$, based on reorderings of axiomatic executions.

**Definition 2 (Axiomatic execution)** *An axiomatic execution $\mathcal{E}$ of a program $P$ is a tuple $\langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ such that:*

- *$P$ is the program,*
- *$A \subset \mathbb{A} \backslash \mathbb{A}_d$ is the finite set of actions of the program,*
- *$\xrightarrow{\mathrm{po}} \in A \times A$ is the* program order*, the union of the total orders on actions of each thread,*
- *$\xrightarrow{\mathrm{so}} \in (A \cup \mathbb{A}_d) \times (A \cup \mathbb{A}_d)$ is the* synchronization order*, the union of a total order on $(A \cap \mathbb{A}_s)$ (the synchronization actions in A), and the cartesian product $\mathbb{A}_d \times (A \cap \mathbb{A}_s)$,*
- *$W$ is a write-seen function that maps each read action to any write action operating on the same address.*

*Intuitively, $W$ returns the write action associated to the value read in the memory by the read action. $\xrightarrow{\mathrm{po}}$ is simply an order giving the order in which the actions can be executed, $\xrightarrow{\mathrm{so}}$ fixes the order in which the synchronization actions can be executed (the total order on $(A \cap \mathbb{A}_s)$), but it also represents the fact that the default actions will be done before the start actions of the threads.*

The semantics of the reordering model is based on the notion of Sequential Consistency (SC) introduced by Lamport in his seminal paper [15]. The actions within the threads are executed in order but can be interleaved with actions of other threads. Lamport describes a method to interconnect correct

$$\frac{ts(t) \xrightarrow{\mathsf{r}_t^i x|V(w)} \!\!\!\!\rightarrow\!\!\!\!\rightarrow\ s \quad w = m(x) \quad \mathsf{volatile}(x)}{ts, m \xrightarrow{\mathsf{r_t^i x|w}}_{\mathsf{synch}} ts[t \mapsto s], m} [\text{RS}]$$

$$\frac{ts(t) \xrightarrow{\mathsf{w}_t^i x, v} \!\!\!\!\rightarrow\!\!\!\!\rightarrow\ s \quad \mathsf{volatile}(x)}{ts, m \xrightarrow{\mathsf{w_t^i x,v}}_{\mathsf{synch}} ts[t \mapsto s], m[x \mapsto (\mathsf{w}_t^i x, v)]} [\text{WS}]$$

Figure 3: $\text{RMO}_{\text{op}}$ machine (synchronization actions)

non-concurrent processors and gives two requirements ensuring the sequential consistency of any execution. We do not focus on this requirements here, but rather on the definition of sequential consistency. A correct processor is a processor for which the result of a program is the same as if it would have been sequentially executed. An execution is sequentially consistent if we can execute the actions of the program in any order (but respecting the order for the actions within each thread) and obtain the same *write-seen* relation. More formally, we have the following definition.

**Definition 3 (**SC **execution)** *An axiomatic execution* $\mathcal{E} = \langle P, A, \xrightarrow{\text{po}}, \xrightarrow{\text{so}}, W\rangle$ *is sequentially consistent (*SC*) if there exists a total order* $\xrightarrow{\text{to}}$ *on* $A$ *such that*

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

In a reordering model, the goal is to apply transformations to a program until an $SC$ execution is found. We formalize this notion of transformation by introducing local reorderings.

**Definition 4 (Local reordering)** *A local reordering of an execution* $\langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ *with respect to a list of actions* $l$ *in a thread* $t$*, is an execution* $\langle P', A, \xrightarrow{po'}, \xrightarrow{so}, W\rangle$ *if:*

- $[\xrightarrow{po}]_t = \alpha \cdot l \cdot \beta$ *and* $[\xrightarrow{po'}]_t = \alpha \cdot l' \cdot \beta$ *with* $l'$ *a permutation of* $l$,
- $[\xrightarrow{po}]_{t'} = [\xrightarrow{po'}]_{t'}$ *for all threads* $t' \neq t$,
- *for all traces* $\alpha$ *and* $\beta$*, if* $(\alpha \cdot l' \cdot \beta, W)$ *is a possible intra-trace for the thread* $t$ *in the program* $P'$*; then* $(\alpha \cdot l \cdot \beta, W)$ *is a possible intra-trace for the thread* $t$ *in the program* $P$.
- *no element of* $l$ *is a synchronization action.*

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

The reordering model allows the program to reorder the consecutive read and write actions if they do not access the same variable. We give a more formal definition using this notion of local reordering.

**Definition 5 (WR reordering)** *Let* $E_a$ *be an axiomatic execution* $\langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$.

*A Write-Read (WR) reordering of* $E_a$ *with respect to a pair* $(w, r)$ *in a thread* $t$*, is a execution E' such that*

- $w$ *and* $r$ *do not access the same address,*

- $E \xrightarrow{t:[w::r \to r::w]} E'$.

*We similarly define Read-Read (RR), Write-Write (WW) and Read-Write (RW) reorderings.*

When such reorderings exist, we will respectively write $E \xrightarrow{\text{WR}} E'$, $E \xrightarrow{\text{RR}} E'$, $E \xrightarrow{\text{WW}} E'$ and $E \xrightarrow{\text{RW}} E'$. We define the $\xrightarrow{\text{RMO}}$ relation by $\xrightarrow{\text{RMO}} = (\xrightarrow{\text{WR}} \cup \xrightarrow{\text{RR}} \cup \xrightarrow{\text{WW}} \cup \xrightarrow{\text{RW}})^*$. The $\text{RMO}_{\text{ro}}$ executions then are all the axiomatic executions reachable with this relation from an SC execution.

**Definition 6 ($\text{RMO}_{\text{ro}}$ execution)** *The set of* $\text{RMO}_{\text{ro}}$ *executions is defined as* $\text{RMO}_{\text{ro}} = \{E | \exists E', E \xrightarrow{\text{RMO}} E'$ *and* $E'$ *is SC*$\}$.

*We write* $\text{RMO}_{\text{ro}}(P)$ *the set of executions of a program* $P$.

We finally define the $\text{RMO}_{\text{ro}}$ observable behavior of a program as the set of all the sequences of external actions:

$$\mathcal{O}_{\text{ro}}(P) = \{\xrightarrow{so}\downarrow_{\mathbb{A}_{ex}} | \ \langle P, A, \xrightarrow{\text{po}}, \xrightarrow{\text{so}}, W\rangle \in \text{RMO}_{\text{ro}}(P)\}.$$

An important property of $\text{RMO}_{\text{ro}}$ is its post-fixpoint characterization, as expressed by the following lemma.

**Lemma 1** $\text{RMO}_{\text{ro}}$ *is the least set* $S$ *that satisfies:*

*i all* SC *executions are in* $S$,

*ii* $S$ *is backward closed by* $\xrightarrow{\text{RMO}}$.

It follows directly from the definition of $\text{RMO}_{\text{op}}$ executions.

# 5 Semantic equivalence between $\text{RMO}_{\text{op}}$ and $\text{RMO}_{\text{ro}}$

We now want to prove that both semantics are observationally equivalent, namely that the following theorem holds.

**Theorem 1** *For all program* $P$,

$$\mathcal{O}_{\text{op}}(P) = \mathcal{O}_{\text{ro}}(P).$$

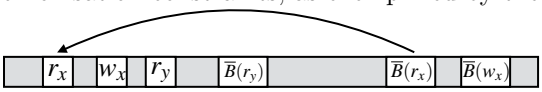
The main idea is to define a function $\rho : \text{RMO}_{\text{op}} \to \text{RMO}_{\text{ro}}$ transforming the operational executions into axiomatic executions.

**Definition 7** *Let* $E_o = (P, tr) \in \text{RMO}_{\text{op}}$, $\rho$ *is defined as* $\rho(E_o) = \langle P, A, \xrightarrow{\text{po}}, \xrightarrow{\text{so}}, W\rangle$ *where*

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

Then, since the function $\rho$ preserves behaviors, we have to prove $\rho(\mathrm{RMO}_{\mathrm{op}}) = \mathrm{RMO}_{\mathrm{ro}}$.

Proving inclusion $\rho(\mathrm{RMO}_{\mathrm{op}}) \subseteq \mathrm{RMO}_{\mathrm{ro}}$ intuitively consists in rearranging the initial operational execution by making unbuffering actions "jump" over other actions, while respecting the FIFO and synchronisation constraints, as exemplified by the following schema.




These jumps are matched on the axiomatic side by reorderings. When all possible jumps have been made to collapse together unbuffering actions and their corresponding original actions, we get the wanted SC execution, as formalized by the following lemma.

**Lemma 2** *Let $E_o = (P, tr) \in \mathrm{RMO}_{\mathrm{op}}(P)$. Then there exists $(P', tr') \in \mathrm{RMO}_{\mathrm{op}}(P')$ such that $\rho(E_0) \xrightarrow{\mathrm{RMO}} \rho(P', tr')$ and all read action in $tr'$ sees the last write action in the trace.*

The other inclusion is obtained using the fixpoint characterization of $\mathrm{RMO}_{\mathrm{ro}}$, that is proving that $\rho(\mathrm{RMO}_{\mathrm{op}})$ contains the SC executions and is backward closed by $\xrightarrow{\mathrm{RMO}}$. Thus, $\mathrm{RMO}_{\mathrm{ro}}$ being the least such set, $\mathrm{RMO}_{\mathrm{ro}} \subseteq \rho(\mathrm{RMO}_{\mathrm{op}})$.

We then conclude that $\mathrm{RMO}_{\mathrm{ro}} = \rho(\mathrm{RMO}_{\mathrm{op}})$ and obtain a proof for Theorem 1.

# 6 Related work

Formalizing concurrency semantics for mainstream languages like Java and C++ has drawn attention without leading to a definitive good solution. The Java Memory Model (JMM) [17] relies on a so-called committing-sequences, that make it very hard to reason on. It also been criticized for soundness problem by Cenciarelli at al. [10] and Sevcík *et al.* [19] because its formal definition does no allow some program transformations that are performed by major Java execution platforms. Lochbihler [16] provides a extensive formal specification of JMM in Isabelle/HOL and proves type-safety for correctly synchronized programs.

There has also been work on memory models for C++. Boehm and Adve [7] provide a semantics for data-race free C++ programs, including a semantics for *low-level atomics*. Batty *et al.* complete this work with a formal specification in HOL [6]. Such a formal model is a prerequisite for further proofs. For example, Sarkar *et al.* proved correctness of a compilation scheme from C++ to Power [18], using this model.

A substantial effort has been made on on characterizing hardware memory models. [1, 2, 14] studied a large range of hardware memory models, and attempted to rigorously formalize the vendor's documentations. Alglave *et al.* [3] defined a general framework for formalizing hardware models using partial orders. Burckhardt *et al.* [9] propose an original semantic framework based on a notion of dynamic reorderings. This is close to the spirit of the BMM of Demange *et al.* [12].

Boudol and Petri [8] give a framework to describe different kinds of operational models. They provide an alternative operational model for RMO. Instead of using read/write buffers, it relies on an *action pipeline* where actions can be reordered. Compared to BMM and its extensions presented in the current paper, Boudol's model is not directly characterized in terms of execution reorderings.

# 7 Conclusion

We have presented two provably equivalent models for the RMO model, extending our previous work on the TSO compliant BMM model. The first model is an operational model with multiple buffers that is a closed abstraction of modern hardware. The second model is directly defined in terms of reordering of axiomatic executions.

In our future work we would like to use the reordering model in soundness proofs. We believe we could find elegant proof techniques for the $\text{RMO}_{\text{ro}}$ model. It would be interesting to use this model to prove standard aggressive memory optimizations that are performed by modern compilers, as common subexpression elimination on memory loads, or partial redundancy elimination on memory accesses [11].

# References


[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12), 1996.

[2] S. V. Adve and M. Hill. A Unified Formalization of Four Shared-Memory Models. *Par. and Distr. Systems, IEEE Transactions on*, 1993.

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

[4] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[5] 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*, pages 509–520, 2012.

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

[7] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. *SIGPLAN Not.*, 43, 2008.

[8] G. Boudol, G. Petri, and B. P. Serpette. Relaxed operational semantics of concurrent programming languages. In *EXPRESS/SOS*, volume 89, pages 19–33, 2012.

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

[10] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java Memory Model: Operationally, denotationally, axiomatically. In *Proc. of ESOP*, 2007.

[11] C. Click. Global code motion / global value mumbering. In *Proc. of PLDI*, pages 246–257. ACM, 1995.

[12] D. Demange, V. Laporte, L. Zhao, S. Jagannathan, D. Pichardie, and J. Vitek. Plan B: a buffered memory model for Java. In *Proc. of POPL*, pages 329–342. ACM, 2013.

[13] L. Higham and J. Kawash. Memory consistency and process coordination for SPARC multiprocessors. In *HiPC*, volume 1970 of *Lecture Notes in Computer Science*, pages 355–366. Springer, 2000.

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

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

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

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

[18] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *Proc. of PLDI*, pages 311–322. ACM, 2012.

[19] J. Sevcík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *Proc. of ECOOP*, pages 27–51, 2008.

[20] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *Proc. of POPL*, pages 43–54. ACM, 2011.

[21] SPARC International Inc. The SPARC architecture manual: version 8. Prentice-Hall, 1992.

[22] SPARC International Inc. The SPARC architecture manual: version 9. Prentice-Hall, 1994.