Compiling Volatile Correctly in Java

Shuyang Liu
University of California, Los Angeles, CA, USA

John Bender
Sandia National Laboratories, Albuquerque, NM, USA

Jens Palsberg
University of California, Los Angeles, CA, USA

Abstract

The compilation scheme for Volatile accesses in the OpenJDK 9 HotSpot Java Virtual Machine has a major problem that persists despite a recent bug report and a long discussion. One of the suggested fixes is to let Java compile Volatile accesses in the same way as C/C++11. However, we show that this approach is invalid for Java. Indeed, we show a set of optimizations that is valid for C/C++11 but invalid for Java, while the compilation scheme is similar. We prove the correctness of the compilation scheme to Power and x86 and a suite of valid optimizations in Java. Our proofs are based on a language model that we validate by proving key properties such as the DRF-SC theorem and by running litmus tests via our implementation of Java in Herd7.

2012 ACM Subject Classification Software and its engineering → Semantics

Keywords and phrases formal semantics, concurrency, compilation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.6

Related Version *Full Version*: `http://compilers.cs.ucla.edu/papers/compiling-volatile.pdf`

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.3`

Funding This material is based upon work supported by the National Science Foundation under Grant No. 1815496.

Acknowledgements We thank Doug Lea for the helpful insights on the Java language semantics and compilers; we thank Jade Alglave for her precious and detailed help on implementing Java architecture for Herd; we thank Ori Lahav, Anton Podkopaev and Viktor Vafeiadis for initially pointing out the issue of the Java Access Modes model; we thank all the reviewers of ECOOP'22 for their insightful feedback.

1 Introduction

In OpenJDK 9, the Java programming language introduced the VarHandle API with Access Modes to provide a standard set of operations that gives clear semantics to programs with shared object fields. Among the four available Access Modes (which we will explain in Section 3 in detail), programmers are allowed to use Volatile mode to ensure the consistency of updates on shared variables. Conceptually, the set of Volatile mode accesses in a program is totally ordered [9]. If all of the accesses in a program are in Volatile mode, then the program should only have sequentially consistent executions since all accesses in that program are totally ordered.

Sadly, this basic property of Volatile mode does not hold under the current implementation of the Java compiler in OpenJDK 9 HotSpot JVM. That is, marking all accesses as Volatile in a Java program can still result in behaviors that are not sequentially consistent when compiling to Power [14]. In particular, the C1 and the C2 compilers in HotSpot do not provide enough synchronization between a Volatile read and a Volatile write when compiling to the Power architecture. While we leave the details of their respective compilation schemes to Section 2, when a program includes a sequence of a Volatile read followed by a Volatile write, there is no `hwsync` instruction inserted in-between. Without the `hwsync`, it is possible for threads to disagree on the orders in which instructions are executed. As a consequence, the compilation schemes can still cause violations of sequential consistency in programs with all accesses marked Volatile. We have contacted the maintainers of the OpenJDK about this issue and a bug report has been filed [17].

One solution is to add the missing `hwsync` instruction to restore sequential consistency for Volatile. The resulting compilation scheme is similar to C/C++11 [4], which leads one to wonder whether Java compilers can simply handle Access Modes the same way as C/C++11 compilers handle atomic memory orders. However, there are significant differences in the semantics of Volatile access mode and the `seq_cst` memory order, which leads to differences in the valid compiler transformations applied to them respectively. In contrast to C/C++11 [6], Java does not allow certain compiler transformations to be applied to Volatile accesses. For example, register promotion cannot be applied to memory locations with Volatile accesses in Java while it can be applied in C/C++11. The differences provide Java programmers stronger synchronization guarantees and a more intuitive reasoning process: Volatile accesses (1) are equivalent to inserting `fullFence()`s, and (2) will not be optimized by the compiler in unexpected ways. We provide a detailed comparison along with soundness proofs and examples in Section 5.

While the change to the compilation scheme appears to be simple, the work of verifying its soundness is challenging. First, the formal language model JAM (hereafter JAM_{19}) [3] exhibits the same issue as the HotSpot compilers. That is, it cannot guarantee sequential consistency for programs with all accesses marked Volatile. Therefore, we revise the language model to fix this issue. To ensure the change to the model is valid we formally verify its key properties, such as the standard DRF-SC theorem, and leverage a set of empirical litmus tests via our implementation of Java in Herd7 [1] that keeps the model valid. We call the revised model JAM_{21} to distinguish from the original version. Second, the language model defines the semantics of `fullFence()` with a total order. However, many target-level architectures such as the Power memory model [14] only specify a partial observable order among their synchronization mechanisms (fence cumulativity). Therefore, we develop an intermediate language model, JAM'_{21}, to bridge JAM_{21} with the target level models. We show that JAM'_{21} yields the same observable program executions as JAM_{21} but does not specify a total order among `fullFence()`s, which simplifies the proof for compilation correctness.

1.1 Outline

The rest of the paper is structured as follows. Section 2 explains the bug in the current Java compiler to Power with an example. In Section 3, we explain the formal model that we use in this paper. Section 4 provides a correctness proof for our proposed compilation scheme to Power. Section 5 presents a set of program transformations that are valid/invalid for Java and a comparison with C/C++11. We include a discussion on expected performance impact in Section 6. Section 7 details some recent related work and finally, Section 8 concludes the paper.

1.2 Supplementary Material

The proofs of the theorems appear in this paper are available in the appendices (which are available in the full version of the paper). The following are also available as artifact of this paper at `https://github.com/ShuyangLiu/ECOOP22-Supplementary-Material`.

- The extended Herd7 tool suite with the Java architecture.
- The litmus tests that appear in this paper.
- The Coq proofs for some of the theorems in this paper.

2 The Problem of Compiling Volatile and How to Fix it

In this section we use an example to demonstrate that the approach implemented by the HotSpot JVM compilers does not provide sequentially consistent semantics even when all accesses use Volatile mode.

Consider the volatile-non-sc.4 example shown as an execution in Fig.1. In this example, there are four concurrent threads (`P1`, `P2`, `P3`, and `P4`) accessing two shared integer variables (x and y). The notation `Wx = 1` means "writing to variable x with value 1". The notation `Rx = 0` means "reading from variable x and the value returned is 0". In addition, each variable is initialized to 0 at the beginning before the threads start execution. The small superscript on each memory access denotes the access mode that the access uses. For example, $\texttt{Rx}^{\texttt{v}}$ means "reading with Volatile mode".

If all of the read and write accesses in this program use Volatile mode, would the reads ever return the values that are specified in the figure?

According to the specification [9], the program must exhibit sequentially consistent behavior because all accesses are marked Volatile:

> *"When all accesses use Volatile mode, program execution is sequentially consistent, in which case, for two Volatile mode accesses A and B, it must be that A precedes execution of B, or vice versa."*

Therefore, we are interested in whether the example in Fig. 1 is sequentially consistent. Sequential consistency, as first defined by [7], requires a total sequential order that preserves program order and the values returned by the reads are compatible with this total order. Following the definition, the execution in Fig. 1 does not satisfy sequential consistency. To see this, we demonstrate a contradiction under the guarantees of sequential consistency. Consider the following order constraints:

1. By program order, we know that (a) occurs before (b).
2. Since the value (b) gets is the initial value 0, it must occur before (c) writes to the location y.
3. Then, (d) reads the value written by (c), so (c) occurs before (d).
4. By program order, (d) occurs before (e).
5. Now, looking at `P4`, we know that the value of x changed from 1 to 2. Therefore, we can infer that (e) occurs before (a) since (e) is the only write to x with a value of 1 and (a) is the only write to x with a value of 2.

In this execution, we find a cycle: (a) $\longrightarrow$ (b) $\longrightarrow$ (c) $\longrightarrow$ (d) $\longrightarrow$ (e) $\longrightarrow$ (a) which appears in Fig. 1 with the "occurs before" relation represented as edges in the execution graph. Sequential consistency requires an irreflexive total order among all instructions. Therefore, the chain formed by the total order should be acyclic, i.e., a valid execution should not exhibit any cycle in its graph. Thus, this execution is inconsistent under sequential consistency and should be forbidden.

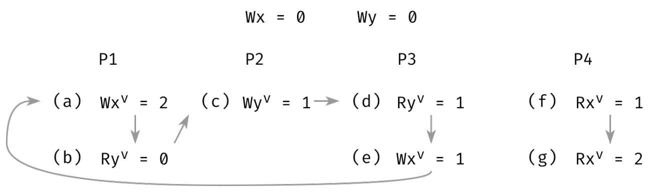

Figure 1 `volatile-non-sc.4` under the sequential consistency model, Forbidden.

```
                    Wx = 0     Wy = 0

       P1             P2             P3              P4

(a)  Wx = 2     (c)  Wy = 1     (d)  Ry = 1     (f)  Rx = 1
(B1) hwsync                     (B2) lwsync     (B3) hwsync
(b)  Ry = 0                     (e)  Wx = 1     (g)  Rx = 2
```

Figure 2 volatile-non-sc.4.ppc translated to Power by HotSpot C1, Allowed.

However, despite the promise of sequential consistency given by the source-level Volatile semantics, the compilation scheme found in the Java compilers for Power allows the example execution in Fig. 1. To see this, we present the compilation scheme from the C1 compiler which is the more conservative of HotSpot's two compilers. We then give a Power-consistent execution graph corresponding to the example in Fig. 1.

The Power architecture adopts a relaxed memory model and provides fence instructions to recover sequential consistency. Two main types of fence instructions, the stronger fence `hwsync` and the weaker fence `lwsync`, are usually used by the compilers to enforce synchronization guarantees. Using `lwsync` usually gives better performance but the synchronization guarantee of `lwsync` is weaker than `hwsync`. In particular, while both fence instructions carries a set of writes (Group A writes) when propagating to another thread, `lwsync` does not require an acknowledgement to continue executing the instructions after it. On the other hand, a `hwsync` requires an acknowledgment marking that it (along with its Group A writes) has propagated to all threads before proceeding to the next instruction.

The compilation to Power for Volatile accesses on C1 is the following [1]:

$$R^V \leadsto \texttt{hwsync ; lwz ; lwsync}$$
$$W^V \leadsto \texttt{lwsync ; stw ; hwsync}$$

A Volatile read is compiled to a `hwsync` instruction followed by a load instruction and a `lwsync` instruction; a Volatile write is compiled to a `lwsync` instruction followed by a store instruction and a `hwsync` instruction.

Fig. 2 shows the example from Fig. 1 according the compilation scheme in the C1 compiler[2].

[1] This compilation scheme was found in the OpenJDK 13 HotSpot compiler and it follows from a previously inaccurate description in the documentation [9] regarding the semantics of Volatile accesses. We have contacted the author and the documentation has been corrected in the latest version while the compiler bug (although reported) is still not fixed at the time of writing.

[2] The C2 compiler yields a slightly different compilation scheme for Volatile reads: Instead of inserting a `lwsync` fence after the load instruction, it emits a control dependency followed by an `isync` instruction, which we denote as `ctrlisync`. But in this example, the resulting execution graph is effectively the same

```
                   Wx = 0      Wy = 0

      P1             P2             P3               P4

(a)  Wx = 2    (c)  Wy = 1    (d)  Ry = 1     (f)  Rx = 1
(B1) hwsync                   (B2) hwsync     (B3) hwsync
(b)  Ry = 0                   (e)  Wx = 1     (g)  Rx = 2
```

Figure 3 volatile-non-sc.4.ppc translated to Power using the revised compilation scheme, Forbidden.

The Power memory model [14] allows the behavior annotated in Fig. 2. The full trace of the execution can be found in the full version of this paper. Here we give a brief explanation. First note that a write operation is split into multiple steps and can be propagated to foreign threads in different orders if not properly synchronized. Furthermore, the `lwsync` in `P3` is not sufficient in this case. In particular, the `lwsync` does not require an acknowledgement before proceeding to the next instructions and it only requires (c) `Wy = 1` to be propagated when itself needs to be propagated to the thread (the cumulativity of `lwsync`). Since `P4` needs to read from (e) `Wx = 1`, which is subsequent to (B2), (B2) needs to be propagated to `P4` before (e) `Wx = 1` is propagated to `P4`. The propagation of (B2) `lwsync` makes sure that (c) `Wy = 1` is propagated to `P4` before it can read x (even though it doesn't really need to read the value of y). On the other hand, `P1` does not have any instructions reads from an instruction of `P3` that comes after (in program order) (B2). Therefore, it does not require (c) and (B2) to be propagated to it when it executes (b). As a result, (c) can be propagated to `P1` long after reaching `P3` and hence letting `P3` and `P1` have different views of the memory during the execution. When `P1` tries to read the value of y, it can only get an initial value of 0 since the newer value has not been propagated to `P1` yet. Consequently, this non-SC execution is allowed (consistent) under the Power memory model, despite that the semantics of the "all-Volatile" source program requires it to be forbidden.

The solution to fix this issue is quite straightforward. Instead of letting Volatile read be translated using "leading fence" while Volatile write be translated using "trailing fence", they should both use the same fence inserting strategy (both leading fence or both trailing fence).[3] Therefore, the correct compiler scheme for Volatile should be:

$$\mathtt{R}^{\mathtt{V}} \rightsquigarrow \texttt{hwsync ; lwz ; lwsync}$$
$$\mathtt{W}^{\mathtt{V}} \rightsquigarrow \texttt{hwsync ; stw}$$

With the revised compilation scheme we can demonstrate that the example of Fig. 1 is forbidden in accordance with the required SC semantics. The resulting execution graph is shown in Fig. 3. While most of this example matches Fig. 2, (B2) now is a `hwsync` instruction. As an effect of this change, (B2) is now required to be propagated to every thread and get

as C1's because the effect of `ctrlisync` is subsumed into the `lwsync` or the `hwsync` instruction that it follows. In addition, we have simplified the compiled code (such as eliminating the fence instructions at the beginning or end of the threads and merging consecutive fence instructions) without changing its semantics for clarity here.

[3] Here we choose to show the leading fence strategy for simplicity. However, the trailing fence strategy is symmetric to leading fence and the same correctness proof works for both conventions given it's used consistently (more details can be found in Section 4.1). In practice, it is usually preferable to use trailing fence strategy for better performance.

acknowledged before start executing (e). As a result, at the time when (c) is propagated to `P4` (as a result of the cumulative effect of (B2) just like in Section. 2), it must also have propagated to `P1` due to the acknowledgement required by the `hwsync` at (B2). Therefore, it becomes impossible for (b) to read the value 0 because Power requires reads to always read from the latest value that has been propagated to the thread. That is, this execution is now forbidden by Power, aligning with the sequentially consistent semantics promised by the Java Volatile mode. Note that the reasoning is the same if we use a "trailing fence" scheme. The key is to deploy a fence insertion strategy such that there is a `hwsync` fence inserted between every pair of Volatile accesses.

Interestingly, we found similar compilation schemes applied to other architectures in HotSpot as well. This is not an accident. The source of this compiling behavior stems from the IR phase of the compiler. At the IR (called the Ideal Graph IR in HotSpot) level, a Volatile read is translated to a `fullFence()` followed by an Acquire read; a Volatile write is translated to a Release write followed by a `fullFence()`. Then each compiler back end translates the code further using the corresponding template file that maps the IR to specific architecture instructions. In the case of Power, a `fullFence()` is mapped to the `hwsync` instruction and Release-Acquire accesses are implemented using the `lwsync` instruction. While the example we provide here focuses on the compilation to Power, the more fundamental issue here is a lack of `fullFence()` between a Volatile read and a Volatile write at the IR encoding level. JAM_{19} aligns with this encoding when specifying the semantics of Volatile memory operations. As a result, JAM_{19} also exhibits the same problem. That is, when all memory accesses are Volatile, JAM_{19} does not guarantee sequential consistency.

3 Formal Model

In this section we present the revised model JAM_{21}, which we use as our theoretical foundation for proving compiler correctness in the rest of the paper. We begin by introducing the basic syntax (Section 3.1) used in the rest of the paper. Then we give the formal definition of JAM_{21} in Section 3.2.

3.1 Basic Syntax

We adopt the syntax of [3] and the `cat` language [1] in addition to some utility functions.

Given a program $P \in \mathbb{P}$, there is a set of executions (run-time traces) associated with P. We call the executions *histories* of P and use H to denote a single history. Each execution history consists of sets of memory access events specified by P. In particular:

- H.`E` denotes the whole set of memory events of H.
- H.`F` denotes the whole set of fence events of H.
- H.`IW` denotes the set of initialization writes of H.
- H.`FW` denotes the set of final writes of H
- H.`W` denotes the set of write events in H.
- H.`R` denotes the set of read events in H.
- H.`RMW` denotes the set of read-modify-write events in H.

Note that we treat each `RMW` events as a single event and $H.\mathtt{RMW} \subseteq H.\mathtt{W}$ and $H.\mathtt{RMW} \subseteq H.\mathtt{R}$. In addition, for `RMW` operations such as *compare-and-swap* (CAS), we assume the operation is on its success comparison path. They are sometimes implemented using LL/SC instructions on hardware, which cannot guarantee atomicity if the comparison fails. We assume each write event to the same memory location has an unique value for simplicity.

For each memory event i, we define the following utility functions to extract memory event attributes:

- $H.AccessMode(i)$ returns the Access Mode of event i in H.
- $H.val(i)$ returns the value of event i in H.
- $H.loc(i)$ returns the shared memory location of event i in H.
- $H.Tid(i)$ returns the thread identifier of which i is executed from

Finally, we use the symbol $\mathbb{H}$ to denote the set of all execution histories.

The memory events of each H are related by order relations.

- The program order (po) is a partial order relation ($\texttt{po} \subseteq H.\texttt{E} \times H.\texttt{E}$) specified by P. We use the notation $i_1 \xrightarrow{\texttt{po}} i_2$ to denote the pair of events $\langle i_1, i_2 \rangle$ related by po and $H.\texttt{po}$ to denote the set of all pairs relates by po in H.
- The reads-from (rf) order is a partial order relation ($\texttt{rf} \subseteq H.\texttt{W} \times H.\texttt{R}$). For each read event i_2, there exists a unique write event i_1 such that $H.val(i_1) = H.val(i_2)$ and $H.loc(i_1) = H.loc(i_2)$. We use the notation $i_1 \xrightarrow{\texttt{rf}} i_2$ to denote the pair of events $\langle i_1, i_2 \rangle$ related by rf and $H.\texttt{rf}$ to denote the set of all pairs relates by rf in H.
- Model-Specific relations. There are sets of relations that are specifically defined by the memory model. They are derived from the event attributes, po, and rf using the semantic rules of the memory model. We will detail them in the next few sections. We use the notation $i_1 \xrightarrow{\texttt{R}} i_2$ to denote the pair of events $\langle i_1, i_2 \rangle \in H.\texttt{R}$.

We also use operations on relations: given relations $\texttt{R}_1$ and $\texttt{R}_2$, we use composition $\texttt{R}_1\texttt{;}\texttt{R}_2$, union $\texttt{R}_1\texttt{|}\texttt{R}_2$, intersection $\texttt{R}_1\texttt{\&}\texttt{R}_2$, complement $\sim\texttt{R}_1$, transitive closure $\texttt{R}_1^+$, and inversion $\texttt{R}_1^{-1}$.

We may present an execution history H as a graph. An execution graph consists of a set of nodes labeled with unique identifiers, and a set of labeled edges. Each labeled node refers to an executed memory access.

Lastly, we use the notation acyclic($\xrightarrow{\texttt{R}}$) to denote that R is acyclic in the execution history.

3.2 The JAM_{21} Model

In this section, we present the JAM_{21} model. The full definition of the relations in JAM_{21} can be found in the full version of this paper. We explain several excerpts of the formal model.

There are five available access modes in JAM_{21}: Plain mode, Opaque mode, Release mode, Acquire mode, and Volatile mode. The synchronization effect of the access modes are partially ordered using $\sqsubseteq$:

$$\text{Plain} \sqsubseteq \text{Opaque} \sqsubseteq \{\text{Release}, \text{Acquire}\} \sqsubseteq \text{Volatile}.$$

3.2.1 Visibility

At the center of JAM_{21} is the notion of *visibility orders* (vo). The most basic form of visibility, vo includes the reads-from (rf) relation. Intuitively, a read has certainly seen the effects of the write it takes its value from. Otherwise, visibility comes from synchronization[4]. Both

[4] Here, we use the high-level term "synchronization" for any memory consistency guarantee among instructions. We noticed that the usage of this term might differ outside of this paper. Therefore, we try to avoid using this term ambiguously to avoid confusion.

Volatile (`V`) and Release(`REL`)-Acquire(`ACQ`), (`RA` as the union) accesses provide synchronization and thus visibility. Note that Volatile accesses are also included in the set of accesses that are considered Release-Acquire by the model. Further, `vo` can be derived from `ra` or `svo` orders, which captures the synchronization effects of Release-Acquire memory events or fences, `spush` or `volint` orders, which capture the synchronization effects of Volatile memory events or `fullFence()`s. In addition, the `pushto` order is trace order (`to`) restricted to the domain of `spush` and `volint`. Composing `pushto` with `spush` or `volint` emulates the cross-thread total order among `fullFence()`s, which is also part of the `vo` order. Finally, `po` to the same location is also included as part of the `vo` definition.

$$
\begin{aligned}
\texttt{ra} &\triangleq \texttt{po ; [REL | V] | [ACQ | V] ; po}\\
\texttt{svo} &\triangleq \texttt{po ; [F \& REL] ; po ; [W] | [R] ; po ; [F \& ACQ] ; po}\\
\texttt{spush} &\triangleq \texttt{po ; [F \& V] ; po}\\
\texttt{volint} &\triangleq \texttt{[V] ; po ; [V]}\\
\texttt{vvo} &\triangleq \texttt{rf | svo | ra | spush | volint | pushto ; (spush | volint)}\\
\texttt{vo} &\triangleq \texttt{vvo}^{+}\texttt{ | po-loc}
\end{aligned}
$$

Note that the definition of `volint` has been corrected from JAM_{19} to ensure sequential consistency for Volatile.

3.2.2 Coherence

The coherence order, `co-jom`, is an order among writes to the same location. Coherence order edges can be derived using the `vo` order and the `po` order among memory accesses.

$$
\begin{aligned}
\texttt{WWco(rel)} &\triangleq \{\langle i_1, i_2\rangle \mid \langle i_1, i_2\rangle \in H.\texttt{rel} \wedge i_1, i_2 \in H.\texttt{W} \wedge H.loc(i_1) = H.loc(i_2) \wedge i_1 \neq i_2\}\\
\texttt{coww} &\triangleq \texttt{WWco(vo)}\\
\texttt{cowr} &\triangleq \texttt{WWco(vo ; rf}^{-1}\texttt{)}\\
\texttt{corw} &\triangleq \texttt{WWco(vo ; po)}\\
\texttt{corr} &\triangleq \texttt{[O | RA | V] ; WWco(rf ; po ; rf}^{-1}\texttt{) ; [O | RA | V]}\\
\texttt{co-jom} &\triangleq \texttt{coww | cowr | corw | corr}
\end{aligned}
$$

Note that `co-jom` is different from the definition of `co` in other memory models such as Power and x86-TSO. Instead of enumerating all possible total coherence order to check the consistency of a given execution history, JAM_{21} derives coherence order `co-jom` among memory events from their known relations. Therefore, `co-jom` is a partial order among writes to the same location in JAM_{21}. We use the notation $i_1 \xrightarrow{\texttt{co-jom}} i_2$ to denote the pair of events $\langle i_1, i_2\rangle$ related by `co-jom` and $H.$`co-jom` to denote the set of all pairs relates by `co-jom` in H. We use the simpler name `co` to denote `co-jom` when the context is clear.

In addition, different from JAM_{19}, Plain mode reads to the same location ordered by `po` can be reordered by compiler and therefore cannot be used to derive `co-jom` order.

3.2.3 Execution Consistency

Axiomatic models define program semantics as the set of allowed executions. We adopt the same definition of *candidate execution* from [1].

▶ **Definition 1** (Consistent Candidate Execution). *Given a program P and a memory model M, an execution history H is a **M-consistent candidate execution of P** if and only if:*

- *H is a candidate execution of P (specified by the architecture of the programming language of which P is written in).*
- *H is M-consistent.*

We denote the set of all M-consistent candidate executions of P by $Histories_M(P)$.

We now have all the definitions needed to define execution consistency under JAM_{21}.

▶ **Definition 2** (JAM_{21}-consistency). *An execution history H is JAM_{21}-**consistent** if it is trace coherent and satisfies the following two requirements:*

1. *No-Thin-Air: po / rf is acyclic.* acyclic($\xrightarrow{\texttt{po} | \texttt{rf}}$)
2. *Coherence.: co-jom is acyclic,* acyclic($\xrightarrow{\texttt{co-jom}}$)

*We say such an execution history H is **allowed** by JAM_{21}. Otherwise, it is **forbidden**.*

For the JAM_{21} model, we use $Histories_{JAM_{21}}(P)$ to denote the set of all JAM_{21}-consistent execution histories of P.

JAM_{21} satisfies a set of properties such as the DRF-SC Theorem. We show the theorems and the proofs in the full version of this paper.

3.2.4 Validation with Litmus Tests

The experimental validation of the JAM_{21} model includes two parts.

First, we implement the Java *architecture* in Herd7. Herd7 [1] was developed to simulate program executions with user-defined memory models. An *architecture* in Herd7 provides the parser for litmus tests written in the language corresponding to the architecture and an operational semantics of the instructions that appear in litmus tests. Herd7 uses the parser and the instruction semantics from the architecture to form an internal representation of the input litmus test and generate the set of all possible executions. Then, Herd7 checks the consistency of the executions using memory models written in the `cat` language. As of today, several mainstream architectures, such as C/C++11 [6], x86 [15], ARM [2], and Power [14], have been implemented and included in Herd7's official repository. Unfortunately, Java is not. JAM_{19} [3] validated its formalization by mapping memory events to other architectures' events that exists in the Herd7 repository and run the litmus tests in the architecture's language. The mapping roughly captures part of the compilation scheme but it is neither complete nor proven sound. For example, in its mapping to ARMv8, Volatile accesses are ignored and not mapped to any memory event. Hence this approach is invalid and the results cannot be trusted though they show intentions on how JAM_{19} was expected to behave. Therefore, we extend the Herd7 tool suite with the Java architecture and translate the set of litmus tests used for testing JAM_{19} to Java[5]. A detailed description of each supported instruction is shown in the full version of this paper.

Second, we validate the JAM_{21} model using the Java translation of the set of litmus tests that was originally used to validate JAM_{19} and compare their outcomes. The results are mostly the same as the results from JAM_{19} except for three cases that are relevant to the inconsistency issue discussed earlier in this paper because we wish to fix the issue while keeping other parts of the model unchanged. The three exceptions reveal another

[5] Note that not all tests are translatable. For example, for the cases that test address dependencies, there is no corresponding Java version since the notion of address dependency does not exist in Java. We drop a small set of litmus tests due to this reason.

aspect of the change, accommodating both the leading fence convention and the trailing fence convention, whereas JAM_{19} forced the compiler to choose a particular (problematic) convention. Since the compiler is free to choose either convention, a full synchronisation is only guaranteed to appear between a pair of Volatile accesses. In effect, certain executions that was forbidden by JAM_{19} are allowed by JAM_{21} since it is no longer guaranteed that Volatile writes are *followed* by a full synchronisation and Volatile reads are *prepended* with a full synchronisation. In addition, we have added new litmus tests for showing the change in the semantics of Volatile, volatile-non-sc.4 and volatile-non-sc.5. While JAM_{19} allows the non-sequentially consistent behavior, JAM_{21} correctly forbids them. We further translated the examples to Power using the problematic compilation scheme, volatile-non-sc.4.ppc and volatile-non-sc.5.ppc, and the tests are indeed allowed by the Power memory model. Please see the full version of this paper for a detailed report.

4 Compilation Correctness to Power

In this section, we show that the revised compilation scheme for Power is correct with respect to the Power memory model [14]. We use an intermediate model for the Java Access Modes that is observationally equivalent to JAM_{21}, which we call JAM'_{21}. We include the detailed definition of JAM'_{21} and the proofs for their observational equivalence in the full version of this paper. We use JAM'_{21} to prove that the revised compilation scheme to Power is correct.

4.1 The Power Memory Model

We use the Power memory model defined in Herd7 [1], which consists of the following basic order definitions (Please see the full version of this paper for the full semantics):

- `po` and `rf` follows the same definitions as in JAM_{21} (as described in Section. 3).
- `co` is the union of total orders among writes to the same location. Additionally, if i_1 and i_2 are events on different threads and $i_1 \xrightarrow{\texttt{co}} i_2$, then $i_1 \xrightarrow{\texttt{coe}} i_2$.
- `ctrl` is the control dependency between memory accesses.
- `ppo` is the set of preserved program orders. The detailed definition can be found in the full version of this paper.
- $\texttt{chapo} \triangleq \texttt{rfe} \mid \texttt{fre} \mid \texttt{coe} \mid (\texttt{fre} \ ; \ \texttt{rfe}) \mid (\texttt{coe} \ ; \ \texttt{rfe})$
- $\texttt{com} \triangleq \texttt{rf} \mid \texttt{fr} \mid \texttt{co}$
- `po-loc` is a subset of `po` that relates accesses to the same locations.
- `rmw` relates the read and the write access from the same `RMW` memory event.
- $\texttt{hb} \triangleq \texttt{ppo} \mid (\texttt{sync} \mid \texttt{lwsync}) \mid \texttt{rfe}$
- $\texttt{propbase} \triangleq ((\texttt{sync} \mid \texttt{lwsync}) \mid (\texttt{rfe} \ ; \ (\texttt{sync} \mid \texttt{lwsync}))) \ ; \ \texttt{hb}^*$
- $\texttt{prop} \triangleq \texttt{propbase} \,\&\, (\texttt{W} * \texttt{W}) \mid (\texttt{chapo}? \ ; \ \texttt{propbase}^* \ ; \ \texttt{sync} \ ; \ \texttt{hb}^*)$
- Additional order definitions can be found in the full version of this paper.

▶ **Definition 3** (Power Consistency). *An execution history H is Power-**consistent** if it is trace coherent and satisfies the following six requirements:*

1. *SC-PER-LOCATION:* $\texttt{po-loc} \mid \texttt{com}$ *is acyclic.*
2. *ATOMICITY:* $\texttt{rmw} \,\&\, (\texttt{fre} \ ; \ \texttt{coe})$ *is empty.*
3. *NO-THIN-AIR:* `hb` *is acyclic.*
4. *PROPAGATION:* $(\texttt{co} \mid \texttt{prop})$ *is acyclic.*
5. *OBSERVATION:* $\texttt{fre};\texttt{prop};\texttt{hb}^*$ *is irreflexive.*
6. *SCXX:* $\texttt{co} \mid (\texttt{po} \,\&\, (\texttt{X} * \texttt{X}))$ *is acyclic (where* `X` *denotes atomic accesses)*

*We say such an execution history H is **allowed** by Power. Otherwise, it is **forbidden**.*

```
      getOpaque() ⇝ lwz ; cmp ; bc
      setOpaque() ⇝ stw
     getAcquire() ⇝ lwz ; lwsync
     setRelease() ⇝ lwsync ; stw
    getVolatile() ⇝ hwsync ; lwz ; lwsync
(Or getVolatile() ⇝ lwz ; hwsync for trailing fence convention)
    setVolatile() ⇝ hwsync ; stw
(Or setVolatile() ⇝ lwsync ; stw ; hwsync for trailing fence convention)
   AcquireFence() ⇝ lwsync
   ReleaseFence() ⇝ lwsync
      fullFence() ⇝ hwsync
      getAndAdd() ⇝ hwsync ; _1:  ldarx ; add ; stdcx.  ; bne _1 ; lwsync
  (Or getAndAdd() ⇝ lwsync ; _1:  ldarx ; add ; stdcx.  ; bne _1 ; hwsync for trailing
                                                              fence convention)
getAndAddAcquire() ⇝ _1:  ldarx ; add ; stdcx.  ; bne _1 ; lwsync
getAndAddRelease() ⇝ lwsync ; _1:  ldarx ; add ; stdcx.  ; bne _1
```

Figure 4 Compilation to Power.

4.2 Compilation Scheme

We use the compilation scheme in Fig. 4. Note that this is slightly different from the compilation scheme found in OpenJDK HotSpot compiler in that each Opaque mode read is translated to a load instruction followed by a conditional branch. This enables us to ensure the No-Thin-Air property as it is not guaranteed in the Power memory model. The problem of Out-of-Thin-Air in axiomatic models has been an active research area for a long time and there exists various ways to use weaker compilation schemes while still ruling out thin-air reads. However, it is out of the scope of this paper and here we adopt the stronger scheme for Opaque mode to simplify the proofs. Additionally, we fix the compilation scheme for Volatile as suggested in Section 2. Note that both leading fence and trailing fence conventions ensure a hwsync instruction is inserted between each pair of Volatile mode accesses as long as they are used consistently (use the same convention for Volatile writes and reads). Therefore, the proof for the trailing fence convention can be carried out in a very similar way as the proof for the leading fence convention.

We start our proof by defining a *CompilesTo* relation over execution histories that relates source level executions to target level executions. Intuitively, the process of compilation can be seen as a transformation function on executions from source level to target level. With the *CompilesTo* relation, we can characterize a subset of target level executions that are constructed particularly through the compilation (following a given compilation scheme) from the source level. Note that at this step we do not check whether the resulting execution is consistent under the target level memory model, since the consistency of an execution is checked after the execution is constructed in axiomatic memory models.

▶ **Definition 4** (Compilation of an Execution). *We define the "CompilesTo" relation $\rightsquigarrow \subseteq \mathbb{H} \times \mathbb{H}$ for the compilation from Java to Power as the following: Given a Java program P_{src}, let P_{tgt} be the target-level program compiled from P_{src} using the compilation scheme in Fig. 4 (using the leading fence convention). Let H_{src} be a candidate execution history of P_{src} and H_{tgt} be a candidate execution history of P_{tgt}. We say $H_{src} \rightsquigarrow H_{tgt}$ if:*

- $H_{tgt}.IW = H_{src}.IW$

- $H_{tgt}.\mathit{FW} = H_{src}.\mathit{FW}$
- $H_{tgt}.E = H_{src}.E$
- $H_{tgt}.\mathtt{rf} = H_{src}.\mathtt{rf}$
- $H_{tgt}.\mathtt{po} = H_{src}.\mathtt{po}$
- $H_{tgt}.\mathtt{co} \subseteq H_{src}.to$
- *If* $i_1 \in H_{src}.\mathtt{E}$, $i_{rmw} \in H_{src}.\mathtt{RMW}$ *and* $i_{rmw} \xrightarrow{\mathtt{po}} i_1$*, then* $i_{rmw} \xrightarrow{\mathtt{ctrl}} i_1$ *in* H_{tgt}
- *If* $i_R^{\sqsupseteq O} \in H_{src}.\mathtt{R}$, $i_1 \in H_{src}.\mathtt{E}$ *and* $i_R^{\sqsupseteq O} \xrightarrow{\mathtt{po}} i_1$*, then* $i_R \xrightarrow{\mathtt{ctrl}} i_1$ *in* H_{tgt}
- *If* $i_1, i_2 \in H_{src}.\mathtt{E}$ *and* $i_1 \xrightarrow{\mathtt{push}} i_2$*, then* $i_1 \xrightarrow{\mathtt{sync}} i_2$ *for* $i_1, i_2 \in H_{tgt}.\mathtt{E}$
- *If* $i_1, i_2 \in H_{src}.\mathtt{E}$ *and* $i_1 \xrightarrow{\mathtt{ra}} i_2$*, then* $i_1 \xrightarrow{\mathtt{lwsync}} i_2$ *for* $i_1, i_2 \in H_{tgt}.\mathtt{E}$

Once we have the source level and target level execution histories, we use the memory model to check for consistency. A correct compilation, intuitively, should not introduce any new program behavior. In this context, it means there should not be any execution H_{src} that is forbidden by the source level memory model being related (by the "CompilesTo" relation) with a H_{tgt} that is allowed by the target level memory model. That is, if H_{tgt} is consistent under the target level memory model, then H_{src} should also be consistent under source level memory model. Formally, we have the following definition (recall that we use $Histories_M(P)$ to denote the set of consistent execution histories if a program P under a memory model M).

▶ **Definition 5** (Compilation Correctness)**.** *Let* P_{src} *be a source program and* S *be a memory model that supports the source language,* P_{tgt} *be the target program compiled from* P_{src} *using a compilation scheme and* T *be a memory model that supports the target language. We say a compiler that compiles* P_{src} *to* P_{tgt} *is* ***correct*** *if for all* $H_{tgt} \in Histories_T(P_{tgt})$ *there exists a* $H_{src} \in Histories_S(P_{src})$ *such that* $H_{src} \rightsquigarrow H_{tgt}$*.*

4.3 Proof of Compilation Correctness

We leverage an intermediate memory model, JAM'_{21}, to prove the compilation correctness to Power. While the complete definition of JAM'_{21} can be found in the full version of this paper, it is important to note that JAM'_{21} is *observationally equivalent* to JAM_{21}, which means they allow the same visible program behaviors given the same program. Intuitively, each consistent execution under JAM_{21} has a corresponding consistent execution under JAM'_{21} with the same set of events and the same observable value on each event. Formally, we give the following definitions for observational equivalence.

▶ **Definition 6** (Observational Equivalence of Execution Histories)**.** *Given a program* P*, let* H *and* H' *be two execution histories of* P*. We say* H *and* H' *are* ***observationally equivalent*** *if:*

- $H.\mathtt{IW} = H'.\mathtt{IW}$
- $H.\mathtt{FW} = H'.\mathtt{FW}$
- $H.\mathtt{E} = H'.\mathtt{E}$
- $H.\mathtt{po} = H'.\mathtt{po}$
- $H.\mathtt{rf} = H'.\mathtt{rf}$
- $\forall i \in H.\mathtt{E}, H.AccessMode(i) = H'.AccessMode(i)$

▶ **Definition 7** (Observational Equivalence of Memory Models)**.** *Given a program* P*, let* M_1 *and* M_2 *be two memory models that support the architecture of the programming language that* P *is written in. Let* $Histories_{M_1}(P)$ *be the set of all* M_1*-consistent candidate executions of* P*; let* $Histories_{M_2}(P)$ *be the set of all* M_2*-consistent candidate executions of* P*. We say* M_1 *and* M_2 *are* ***observationally equivalent*** *if:*

- *(⇒) For all $H_1 \in Histories_{M_1}(P)$, there exists $H_2 \in Histories_{M_2}(P)$ such that H_1 is observationally equivalent to H_2.*
- *(⇐) For all $H_2 \in Histories_{M_2}(P)$, there exists $H_1 \in Histories_{M_1}(P)$ such that H_2 is observationally equivalent to H_1.*

Then we prove the compilation correctness from JAM'_{21} to Power.

▶ **Lemma 8** (JAM'_{21} to Power)**.** *Let P_{src} be a Java program, P_{tgt} be the Power program compiled from P_{src} using the compilation scheme in Fig. 4 (with the leading fence convention). For all $H_{tgt} \in Histories_{Power}(P_{tgt})$ there exists a $H_{src} \in Histories_{JAM'}(P_{src})$ such that $H_{src} \rightsquigarrow H_{tgt}$.*

Please see the full version of this paper for the proof.

Finally, we associate JAM_{21} with JAM'_{21} through the notion of observational equivalence and prove the compilation correctness from JAM_{21} to Power.

▶ **Theorem 9** (Compilation Correctness to Power (Leading Fence Convention))**.** *The compilation from Java to Power following the compilation scheme in Fig. 4 (using the leading fence convention) is correct. That is, let P_{src} be a Java program, P_{tgt} be the Power program compiled from P_{src} using the compilation scheme in Fig. 4 (using the leading fence convention). For all $H_{tgt} \in Histories_{Power}(P_{tgt})$ there exists a $H_{src} \in Histories_{JAM}(P_{src})$ such that $H_{src} \rightsquigarrow H_{tgt}$.*

Please see the full version of this paper for the proof.

▶ **Corollary 10** (Compilation Correctness to Power (Trailing Fence Convention))**.** *The compilation from Java to Power following the compilation scheme in Fig. 4 (using the trailing fence convention) is correct. That is, let P_{src} be a Java program, P_{tgt} be the Power program compiled from P_{src} using the compilation scheme in Fig. 4 (using the trailing fence convention). For all $H_{tgt} \in Histories_{Power}(P_{tgt})$ there exists a $H_{src} \in Histories_{JAM}(P_{src})$ such that $H_{src} \rightsquigarrow H_{tgt}$.*

Please see the full version of this paper for the proof.

5 Compiler Transformations

One important aspect of compilers is the program transformations that they apply to the program. A correct compiler transformation should not introduce any new program behavior. While this is relatively simple for sequential programs, it can yield subtle issues when applying the same transformations to concurrent programs. A memory model's task is then to accommodate a set of common program transformations while still provide intuitive synchronization guarantees to the programmers. In Section 4 we show that Java and C/C++11 can use the same compilation scheme to Power (and x86, please see the full version of this paper). However, Java has a stronger semantics for Volatile comparing to `seq_cst` in C/C++11 and can adopt only a strict subset of the transformations that are valid for C/C++11.

In this section, we use the set of compiler transformations detailed by [6] and compare their soundness in Java with C/C++11. We provide formal proofs for the sound transformations and counter-examples for invalid transformations. We conclude this section by discussing the implications of our results.

To prove a transformation is valid, intuitively, we show that there does not exist a H_{src} of P_{src} such that it is forbidden by JAM_{21} but the corresponding H_{tgt} of P_{tgt} is allowed.

Transformation		C/C++11	Java
Strengthening	[Sec. 5.1]	✓	✓
Sequentialisation	[Sec. 5.2]	✓	✓
Reordering	[Sec. 5.3]	See Fig. 6	
Merging	[Sec. 5.4]	See Fig. 7	
Register Promotion	[Sec. 5.5]	✓	For locations that does not have Volatile access

Figure 5 Compiler Transformations in C/C++11 and Java.

▶ **Definition 11** (Valid Program Transformation). *Let P_{src} be a Java program which has a set of candidate executions, $Histories(P_{src})$. Let $T : \mathbb{H} \to \mathbb{H}$ be a program transformation and $H_{tgt} = T(H_{src})$ for each candidate execution H_{src} of P_{src}. Then we say T is* ***valid*** *under JAM_{21} if and only if for each H_{tgt}, if H_{tgt} is JAM_{21}-consistent, then H_{src} is also JAM_{21}-consistent.*

The results for Java comparing them C/C++11 [6] are summarized in Fig. 5.

5.1 Strengthening

Strengthening transforms the access mode of accesses to stronger access modes. It is supported by JAM_{21} due to the monotonicity property of the memory model. The formal theorem is the following:

▶ **Theorem 12** (Strengthening). *Let H_{tgt} an execution of P_{tgt}, which is obtained from applying Strengthening to a program P_{src}. There exists an execution H_{src} of P_{src} such that:*

- $H_{src}.E = H_{tgt}.E$
- $H_{src}.\mathtt{po} = H_{tgt}.\mathtt{po}$
- $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf}$
- $\forall i \in H_{src}.E, H_{src}.AccessMode(i) \sqsubseteq H_{tgt}.AccessMode(i)$

If H_{tgt} is JAM_{21}-consistent, then H_{src} is JAM_{21}-consistent.

Proof. By Monotonicity of JAM_{21}, all the constraints in H_{src} are preserved in the strengthened execution H_{tgt}. Therefore, if H_{tgt} is JAM_{21}-consistent, so is H_{src}. ◀

5.2 Sequentialisation

Sequentialisation transforms two concurrent accesses into accesses in a single sequential process. It is natually supported by JAM_{21} because sequentialisation does not remove any synchronization from the program.

▶ **Theorem 13** (Sequentialisation). *Let P_{src} be a Java program and P_{tgt} be a Java program obtained by performing a sequentialisation operation on a pair of accesses a and b. Let H_{tgt} be an execution of P_{tgt}. Then there exists an execution H_{src} of P_{src} such that*

- $H_{src}.\mathtt{po} \cup \{\langle a, b\rangle\} = H_{tgt}.\mathtt{po}$ *where* $\langle a, b\rangle \notin H_{src}.\mathtt{po}$ *and* $\langle b, a\rangle \notin H_{src}.\mathtt{po}$
- $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf}$
- $H_{src}.\mathtt{E} = H_{tgt}.\mathtt{E}$
- $H_{src}.\mathtt{to} = H_{tgt}.\mathtt{to}$
- $H_{src}.\mathtt{IW} = H_{tgt}.\mathtt{IW}$
- $\forall i \in H_{src}.\mathtt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$

and if H_{tgt} is JAM_{21}-consistent, then H_{src} is JAM_{21}-consistent.

	$\mathtt{R}_y^{m_2}$	$\mathtt{W}_y^{m_2}$	$\mathtt{RMW}_y^{m_2}$	$\mathtt{F}^{m_2}$
$\mathtt{R}_x^{m_1}$	$m_1 \sqsubseteq \text{Opaque}$	$m_1, m_2 \sqsubseteq \text{Opaque} \wedge (m_1 = \text{Plain} \vee m_2 = \text{Plain})$	$m_1 = \text{Plain} \wedge m_2 \sqsubseteq \text{Acquire}$	$(m_1 \sqsubseteq \text{Opaque} \wedge m_2 = \text{Release} \wedge \forall i, \mathtt{F}^{m_2} \xrightarrow{\text{po}} i \Rightarrow i \notin H.\mathtt{W}) \vee (m_1 = \text{Acquire} \wedge m_2 = \text{Acquire}) \vee (m_1 = \text{Acquire} \wedge m_2 = \text{Release})$
$\mathtt{W}_x^{m_1}$	$m_1 \neq \text{Volatile} \vee m_2 \neq \text{Volatile}$	$m_2 \sqsubseteq \text{Opaque}$	$m_2 \sqsubseteq \text{Acquire}$	$(m_2 = \text{Acquire}) \vee (m_2 = \text{Release} \wedge \forall i, \mathtt{F}^{m_2} \xrightarrow{\text{po}} i \Rightarrow i \notin H.\mathtt{W}) \vee (m_2 = \text{Release} \wedge \forall i, \mathtt{F}^{m_2} \xrightarrow{\text{po}} i \wedge i \in H.\mathtt{W} \Rightarrow AccessMode(i) = \text{Release})$
$\mathtt{RMW}_x^{m_1}$	$m_1 \sqsubseteq \text{Release}$	$m_1 \sqsubseteq \text{Release} \wedge m_2 = \text{Plain}$	-	$(m_1 \sqsupseteq \text{Acquire} \wedge m_2 = \text{Acquire}) \vee (m_2 = \text{Release} \wedge \forall i, \mathtt{F}^{m_2} \xrightarrow{\text{po}} i \Rightarrow (i \in H.\mathtt{R} \vee (i \in H.\mathtt{W} \wedge AccessMode(i) = \text{Release})))$
$\mathtt{F}^{m_1}$	$(m_1 = \text{Release}) \vee (m_1 = \text{Acquire} \wedge \forall i, i \xrightarrow{\text{po}} \mathtt{F}^{m_1} \Rightarrow i \notin H.\mathtt{R})$	$m_1 = \text{Release} \wedge m_2 \sqsupseteq \text{Release} \vee (m_1 = \text{Acquire} \wedge \forall i, i \xrightarrow{\text{po}} \mathtt{F}^{m_1} \Rightarrow i \notin H.\mathtt{R})$	$m_1 = \text{Release} \wedge m_2 \sqsupseteq \text{Release} \vee (m_1 = \text{Acquire} \wedge \forall i, i \xrightarrow{\text{po}} \mathtt{F}^{m_1} \Rightarrow i \notin H.\mathtt{R})$	$(m_1 = \text{Release} \wedge m_2 = \text{Acquire}) \vee (m_1 = \text{Acquire} \wedge \forall i, i \xrightarrow{\text{po}} \mathtt{F}^{m_1} \Rightarrow i \notin H.\mathtt{R}) \vee (m_2 = \text{Release} \wedge \forall i, \mathtt{F}^{m_2} \xrightarrow{\text{po}} i \Rightarrow i \notin H.\mathtt{W})$

Figure 6 Allowed Deordering Pairs in JAM_{21}.

Proof. Assume towards contradiction that H_{src} is not JAM_{21}-consistent. Then there are two cases: either there is a $(\mathtt{po} \mid \mathtt{rf})^+$ cycle or a $\mathtt{co}$ cycle in H_{src}. Whether or not a and b are included in this cycle, adding a $\mathtt{po}$ edge between a and b cannot eliminate this cycle (although it might introduces new cycles). Therefore, H_{tgt} is also not JAM_{21}-consistent, contradicting to our assumption. ◀

5.3 Reordering

The operation of *reordering* can be seen as composing *deordering* with *sequentialisation*. Since we know that sequentialisation is sound in JAM_{21}, we only need to show that deordering is sound in order to show reordering is sound in JAM_{21}.

Deordering

Deordering is a transformation that turns a pair of accesses related by a $\mathtt{po}$ relation into a pair of concurrent accesses. In effect, it removes an $\mathtt{po}$ edge in the execution graph.

First, we adopt the same definition of adjacent events from [6]:

▶ **Definition 14** (Adjacent Events). *Two events a and b are* ***adjacent*** *in a partial order* $\mathtt{R}$ *if for all c, we have:*

- $c \xrightarrow{\mathtt{R}} a \Rightarrow c \xrightarrow{\mathtt{R}} b$
- $b \xrightarrow{\mathtt{R}} c \Rightarrow a \xrightarrow{\mathtt{R}} c$

For Java, the table of allowed reordering two adjacent events (with each row as the first event and column as the second event) is shown in Fig. 6 (some of the cases are different from C11 [6] and we have marked them in red). Intuitively, the sound deorderable pairs are ordered by the $\mathtt{po}$ edges that does not impose any synchronization in the program. Therefore, deordering (removing the $\mathtt{po}$ edge) does not introduce new program behavior.

Name	C/C++11	Java
Read-read Merging	$\mathtt{R}^{m}; \mathtt{R}^{m} \rightsquigarrow \mathtt{R}^{m}$	$\mathtt{R}^{m \sqsubseteq \mathsf{Acq}}; \mathtt{R}^{m \sqsubseteq \mathsf{Acq}} \rightsquigarrow \mathtt{R}^{m}$
Write-write Merging	$\mathtt{W}^{m}; \mathtt{W}^{m} \rightsquigarrow \mathtt{W}^{m}$	$\mathtt{W}^{m \sqsubseteq \mathsf{Rel}}; \mathtt{W}^{m \sqsubseteq \mathsf{Rel}} \rightsquigarrow \mathtt{W}^{m}$
Write/RMW-read Merging	$\mathtt{W}^{m}; \mathtt{R}^{\mathsf{acq}} \rightsquigarrow \mathtt{W}^{m}$	$\mathtt{W}^{m}; \mathtt{R}^{m \sqsubseteq \mathsf{Opq}} \rightsquigarrow \mathtt{W}^{m}$
	$\mathtt{W}^{\mathsf{sc}}; \mathtt{R}^{\mathsf{sc}} \rightsquigarrow \mathtt{W}^{\mathsf{sc}}$	✗
	$\mathtt{RMW}^{m}; \mathtt{R}^{m_r \sqsubseteq m} \rightsquigarrow \mathtt{RMW}^{m}$	$\mathtt{RMW}^{m}; \mathtt{R}^{m \sqsubseteq \mathsf{Opq}} \rightsquigarrow \mathtt{RMW}^{m}$
Write-RMW Merging	$\mathtt{W}^{m_w \sqsubseteq m}; \mathtt{RMW}^{m} \rightsquigarrow \mathtt{W}^{m_w}$	$\mathtt{W}^{m_w \sqsubseteq \mathsf{Rel}}; \mathtt{RMW}^{m \sqsubset \mathsf{Vol}} \rightsquigarrow \mathtt{W}^{m_w}$
RMW-RMW Merging	$\mathtt{RMW}^{m}; \mathtt{RMW}^{m} \rightsquigarrow \mathtt{RMW}^{m}$	$\mathtt{RMW}^{m \sqsubset \mathsf{Vol}}; \mathtt{RMW}^{m \sqsubset \mathsf{Vol}} \rightsquigarrow \mathtt{RMW}^{m}$
Fence-fence Merging	$\mathtt{F}^{m}; \mathtt{F}^{m} \rightsquigarrow \mathtt{F}^{m}$	$\mathtt{F}^{m}; \mathtt{F}^{m} \rightsquigarrow \mathtt{F}^{m}$

Figure 7 Mergable Pairs in C/C++11 [6] and Java.

To prove that JAM_{21} supports the reordering shown in this table, we need to prove each cell shown in the table is valid for JAM_{21}.

▶ **Theorem 15** (Deordering). *Let P_{src} be a Java program and P_{tgt} be a Java program obtained by performing a deordering operation on a pair of accesses a and b according to Fig. 6. Let H_{tgt} be an execution of P_{tgt}. Then there exists an execution H_{src} of P_{src} such that*

- $H_{src}.\mathtt{po} = H_{tgt}.\mathtt{po} \cup \{\langle a, b\rangle\}$ *where a and b are* po*-adjacent*
- $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf}$
- $H_{src}.\mathtt{E} = H_{tgt}.\mathtt{E}$
- $H_{src}.\mathtt{to} = H_{tgt}.\mathtt{to}$
- $H_{src}.\mathtt{IW} = H_{tgt}.\mathtt{IW}$
- $\forall i \in H_{src}.\mathtt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$

and if H_{tgt} is JAM_{21}-consistent, then H_{src} is JAM_{21}-consistent.

Please see the full version of this paper for the proof.

Reordering, as mentioned previously, can be decomposed into two steps: deordering and sequentialisation. Since we have already shown the soundness of the two transformations, the soundness of reordering follows naturally.

▶ **Corollary 16** (Reordering). *JAM_{21} supports the reordering transformation for pairs of adjacent accesses shown in Fig. 6.*

5.4 Merging

Merging transforms two adjacent accesses into one single equivalent access to reduce the number of memory accesses in the program. We have grouped all types of merging transformations appeared in C/C++11 [6] here in one section. A summarized result of mergable pairs comparing with C/C++11 can be found in Fig. 7. The results are mostly similar except for Volatile. Many merging transformation are invalid for Volatile because they remove the cross-thread synchronization of Volatile.

5.4.1 Read-Read Merging

Read-read merging is sometimes done when the compiler is optimizing redundant loads in the same thread. When we are encountering two consecutive reads to the same location, the first read is unchanged but the second read becomes a local read without accessing the memory.

Let a' and b be two adjacent read accesses reading from the same write access a. $a \xrightarrow{\texttt{rf}} a'$ and $a \xrightarrow{\texttt{rf}} b$, and $a' \xrightarrow{\texttt{po}} b$. Assuming $AccessMode(a') = AccessMode(b)$, then

- $\forall i, a' \xrightarrow{\texttt{po}} i \Rightarrow b \xrightarrow{\texttt{po}} i$
- $\forall i, a' \xrightarrow{\texttt{ra}} i \Rightarrow b \xrightarrow{\texttt{ra}} i$
- $\forall i, a' \xrightarrow{\texttt{push}} i \Rightarrow b \xrightarrow{\texttt{push}} i$
- $\forall j, j \xrightarrow{\texttt{po}} b \Rightarrow j \xrightarrow{\texttt{po}} a'$

For executions, this corresponds to the following transformation in the execution graph: since the value of $r1$ and $r2$ are guaranteed to have the same value in P_{tgt}, we know that this corresponds to the execution of P_{src} where the two read accesses read from the same write access. Then we want to show that, if H_{tgt} is JAM_{21}-consistent, H_{src} is also JAM_{21}-consistent.

▶ **Theorem 17** (Read-Read Merging)**.** *Let H_{tgt} be an JAM_{21}-consistent execution. Let $a \in H_{tgt}.R \backslash RMW$ and let $a' \in H_{tgt}.\texttt{E}$ such that $a \xrightarrow{\texttt{rf}} a'$. Let $b \notin H_{tgt}.\texttt{E}$. There exists a H_{src} such that:*

- $H_{src}.\texttt{po} = H_{tgt}.\texttt{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, b\rangle \mid i \xrightarrow{\texttt{po}} a\} \cup \{\langle b, j\rangle \mid a \xrightarrow{\texttt{po}} j\}$
- $H_{src}.\texttt{rf} = H_{tgt}.\texttt{rf} \cup \{\langle a', b\rangle\}$
- $H_{src}.\texttt{E} = H_{tgt}.\texttt{E} \cup \{b\}$
- $H_{src}.to = H_{tgt}.to \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{to} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{to} j\}$
- $H_{src}.\texttt{IW} = H_{tgt}.\texttt{IW}$
- $\forall i \in H_{tgt}.\texttt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$
- $b \in H_{src}.R$
- $H_{src}.AccessMode(b) = H_{src}.AccessMode(a) \sqsubseteq$ *Acquire*

and H_{src} is JAM_{21}-consistent.

Please see the full version of this paper for the proof.

Note that JAM_{21} does not allow read-read merging if the two read accesses are both Volatile mode reads. We provide an example of this in the full version of this paper.

5.4.2 Write-Write Merging

The write-write merge transformation refers to the program transformation that merges two consecutive write operations into one by removing the former one. JAM_{21} support write-write merge when the access modes of the two writes are the same and they are not Volatile mode accesses.

Let a and b be the two adjacent writes such that $a \xrightarrow{\texttt{po}} b$. We once again have the properties:

- $\forall i, i \xrightarrow{\texttt{po}} a \Rightarrow i \xrightarrow{\texttt{po}} b$
- $\forall j, b \xrightarrow{\texttt{po}} j \Rightarrow a \xrightarrow{\texttt{po}} j$
- $\forall i, i \xrightarrow{\texttt{ra}} a \Rightarrow i \xrightarrow{\texttt{ra}} b$

We have the following theorem.

▶ **Theorem 18** (Write-Write Merging)**.** *Let H_{tgt} be an JAM_{21}-consistent execution. Let $b \in H_{tgt}.W \backslash RMW$ and let $a \notin H_{tgt}.\texttt{E}$ and $loc(a) = loc(b) \wedge \forall i \in H_{tgt}.W, loc(i) = loc(b) \Rightarrow val(a) \neq val(i)$. There exists a H_{src} such that:*

- $H_{src}.\texttt{po} = H_{tgt}.\texttt{po} \cup \{\langle a,b\rangle\} \cup \{\langle i,a\rangle \mid i \xrightarrow{\texttt{po}} b\} \cup \{\langle a,j\rangle \mid b \xrightarrow{\texttt{po}} j\}$
- $H_{src}.\texttt{rf} = H_{tgt}.\texttt{rf}$
- $H_{src}.\texttt{E} = H_{tgt}.\texttt{E} \cup \{a\}$
- $H_{src}.\textit{to} = H_{tgt}.\textit{to} \cup \{\langle a,b\rangle\} \cup \{\langle i,a\rangle \mid i \xrightarrow{to} b\} \cup \{\langle a,j\rangle \mid b \xrightarrow{to} j\}$
- $H_{src}.\texttt{IW} = H_{tgt}.\texttt{IW}$
- $\forall i \in H_{tgt}.\texttt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$
- $a \in H_{src}.W$
- $H_{src}.AccessMode(a) = H_{src}.AccessMode(b) \sqsubseteq \mathit{Release}$

and H_{src} is JAM_{21}-consistent.

Please see the full version of this paper for the proof.

Note that write-write merging is not valid for Volatile mode writes. We provide an example of this in the full version of this paper.

5.4.3 Write/RMW-read Merging

The Write/RMW-read merging refers to the program transformation that merges a write/RMW and a read into a single write/RMW and a local access.

Similarly, the transformation with an RMW operation and a read operation optimizes the latter read operation to read locally and in effect removes a memory load operation in the execution graph.

JAM_{21} only support this transformation when the read operation is (or is weaker than) Opaque mode which is different from RC11 [6]'s result for C/C++11. We provide a counter-example in the full version of this paper to show that write/RMW-read merging is invalid when the read is (or is stronger than) Acquire mode.

▶ **Theorem 19** (Write/RMW-Read Merging). *Let H_{tgt} be a JAM_{21}-consistent execution. Let $a \in H_{tgt}.\texttt{W}$ and $b \notin H_{tgt}.\texttt{E}$. There exists a H_{src} such that:*

- $H_{src}.\texttt{E} = H_{tgt}.\texttt{E} \cup \{b\}$
- $b \in H_{src}.\texttt{R}$
- $H_{src}.loc(b) = H_{src}.loc(a)$
- $H_{src}.val(b) = H_{src}.val(a)$
- $H_{src}.\texttt{po} = H_{tgt}.\texttt{po} \cup \{\langle a,b\rangle\} \cup \{\langle i,a\rangle \mid i \xrightarrow{\texttt{po}} b\} \cup \{\langle a,j\rangle \mid b \xrightarrow{\texttt{po}} j\}$
- $H_{src}.\texttt{rf} = H_{tgt}.\texttt{rf} \cup \{\langle a,b\rangle\}$
- $H_{src}.\textit{to} = H_{tgt}.\textit{to} \cup \{\langle a,b\rangle\} \cup \{\langle i,a\rangle \mid i \xrightarrow{to} b\} \cup \{\langle a,j\rangle \mid b \xrightarrow{to} j\}$
- $H_{src}.\texttt{IW} = H_{tgt}.\texttt{IW}$
- $\forall i \in H_{tgt}.\texttt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$
- $H_{src}.AccessMode(b) \sqsubseteq \mathit{Opaque}$

Please see the full version of this paper for the proof.

5.4.4 Write-RMW Merging

The write-RMW merging refers to the program transformation that merges a write and a consecutive RMW operation into a write with the value of the RMW. For example, if we have the following pattern in a program:

```
x = 1;
x.getAndSet(1,2);
```

It can be tranformed to:

```
x = 2;
```

Similar to write-write merging, JAM_{21} supports write-RMW merging when the access mode of the write is {Opaque, Release} and the access mode of the RMW is {Acquire, Release}.

▶ **Theorem 20** (Write-RMW Merging). *Let H_{tgt} be a JAM_{21}-consistent execution. Let $b \in H_{tgt}.\mathtt{W} \backslash H_{tgt}.\mathtt{RMW}$, $a \notin H_{tgt}.\mathtt{E}$ and $v \in \mathtt{Val}$. There exists a H_{src} such that:*

- $H_{src}.\mathtt{E} = H_{tgt}.\mathtt{E} \cup \{a\}$
- $\forall i \in H_{tgt}.\mathtt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$
- $H_{src}.AccessMode(a) \in \{\mathit{Opaque},\ \mathit{Release}\}$
- $H_{src}.AccessMode(b) \in \{\mathit{Acquire},\ \mathit{Release}\}$
- $H_{src}.loc(b) = H_{src}.loc(a)$
- $b \in H_{src}.\mathtt{RMW}$
- $H_{src}.val(b) = (H_{src}.val(a), v)$
- $H_{src}.\mathtt{po} = H_{tgt}.\mathtt{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\mathtt{po}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\mathtt{po}} j\}$
- $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf} \cup \{\langle a, b\rangle\}$
- $H_{src}.\mathit{to} = H_{tgt}.\mathit{to} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{to} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{to} j\}$
- $H_{src}.\mathtt{IW} = H_{tgt}.\mathtt{IW}$

and H_{src} is JAM_{21}-consistent.

Please see the full version of this paper for the proof.

5.4.5 RMW-RMW Merging

The RMW-RMW merging transformation refers to the program transformation that merges two consecutive RMW operations into one such that it has the first RMW's (expected) read value and the second RMW's write value. For example, if we have the following pattern in a program:

```
x.getandSet(1,2);
x.getandSet(2,3);
```

then it might be transformed into:

```
x.getAndSet(1,3);
```

The RMW-RMW merging transformation is essentially the same as write-write merging and read-read merging described previously. Therefore, the set of constraints on valid access modes for merging is the intersection of the two. That is, two RMWs are mergeable if they are both Acquire mode or Release mode. For the counter-examples showing this transformation is invalid for Volatile accesses, please see the examples for write-write and read-read merging.

▶ **Theorem 21** (RMW-RMW Merging). *Let H_{tgt} be a JAM_{21}-consistent execution. Let x be a memory location and $a \in H_{tgt}.\mathtt{E}$ with $H_{tgt}.val(a) = (v_r, v_w)$, $H_{tgt}.loc(a) = x$, and $H_{tgt}.AccessMode(a) \in \{\mathit{Release},\ \mathit{Acquire}\}$. Let $b \notin H_{tgt}.\mathtt{E}$, there exists a H_{src} such that:*

- $H_{src}.\mathtt{E} = H_{tgt}.\mathtt{E} \cup \{b\}$
- $\forall i \in H_{tgt}.\mathtt{E}, H_{src}.AccessMode(i) = H_{tgt}.AccessMode(i)$
- $H_{src}.val(a) = (v_r, v)$
- $H_{src}.val(b) = (v, v_w)$
- $H_{src}.loc(b) = x$
- $H_{src}.AccessMode(b) = H_{src}.AccessMode(a) \in \{\mathit{Release},\ \mathit{Acquire}\}$
- $H_{src}.\mathtt{po} = H_{tgt}.\mathtt{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, b\rangle \mid i \xrightarrow{\mathtt{po}} a\} \cup \{\langle b, j\rangle \mid a \xrightarrow{\mathtt{po}} j\}$
- $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf} \cup \{\langle a, b\rangle\}$
- $H_{src}.\mathit{to} = H_{tgt}.\mathit{to} \cup \{\langle a, b\rangle\} \cup \{\langle i, b\rangle \mid i \xrightarrow{to} a\} \cup \{\langle b, j\rangle \mid a \xrightarrow{to} j\}$
- $H_{src}.\mathtt{IW} = H_{tgt}.\mathtt{IW}$

and H_{src} is JAM_{21}-consistent.

Please see the full version of this paper for the proof.

5.4.6 Fence-fence Merging

The Fence-fence merging refers to the program transformation that merges two consecutive fences of the same access mode into one. For example, if we have:

```
VarHandle.fullFence();
VarHandle.fullFence();
```

then it can be optimized to:

```
VarHandle.fullFence();
```

Since JAM_{21} is fence-based such that each fence is converted into an edge between memory accesses, this is trivially supported since the execution graph before and after the transformation is exactly the same.

5.5 Register Promotion for Non-shared Variable

Register Promotion promotes memory accesses of a non-shared memory location to local registers. It has the effect of removing memory accesses for thread-local variables. JAM_{21} only supports register promotion for variables without any Volatile accesses in the program. For non-Volatile accesses, since the variable is not shared across threads, it is safe to remove them without worrying about removing synchronization from the program. In contrast, Volatile accesses impose cross-thread synchronizations with Volatile accesses for other variables, so removing such accesses can potentially remove important synchronization in the program and introduce new behaviors that were previously forbidden by the memory model. We provide a counter-example in this section showing that we cannot promote Volatile accesses to local register accesses even if the location is only accessed by one thread.

Suppose all accesses to a memory location are in the same thread, the transformation can be seen as two steps:

1. Weakening the accesses to Plain mode accesses
2. Removing the Plain mode accesses

▶ **Theorem 22** (Weakening for non-shared variable). *Let H_{tgt} be a JAM_{21}-consistent execution such that, for all accesses i and j in H_{tgt}.E, $loc(i) = loc(j) = x \Rightarrow Tid(i) = Tid(j)$ for some memory location x. In addition, $\forall i \in H_{tgt}.\mathtt{E}, loc(i) = x \Rightarrow AccessMode(i) =$ Plain. There exists an execution H_{src} such that:*

- $H_{src}.\mathtt{E} = H_{tgt}.\mathtt{E}$
- $H_{src}.\mathtt{po} = H_{tgt}.\mathtt{po}$
- $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf}$
- $H_{src}.\mathit{to} = H_{tgt}.\mathit{to}$
- $H_{src}.\mathtt{IW} = H_{tgt}.\mathtt{IW}$
- $\forall i \in H_{src}.\mathtt{E}, loc(i) = x \Rightarrow AccessMode(i) \in$ {Release, Acquire}

and H_{src} is JAM_{21}-consistent.

Please see the full version of this paper for the proof.

▶ **Theorem 23** (Removing Plain accesses for non-shared variable). *Let H_{tgt} be a JAM_{21}-consistent execution. Let x be a memory location and for all $i \in H_{tgt}.\mathtt{E}$ such that $loc(i) = x$, $Tid(i) = t$ for some t. Let $a \notin H_{tgt}.\mathtt{E}$. There is a H_{src} such that:*

- $H_{src}.\mathtt{E} = H_{tgt}.\mathtt{E} \cup \{a\}$
- $H_{src}.loc(a) = x$
- $H_{src}.AccessMode(a) = \mathsf{Plain}$
- $H_{src}.\mathtt{po} \supset H_{tgt}.\mathtt{po}$
- *for all* $i \in H_{src}.\mathtt{E}$ *such that* $H_{src}.loc(i) = x$, $i \xrightarrow{\mathtt{po}} a$ *or* $a \xrightarrow{\mathtt{po}} i$
- $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf}$ *if* $a \in H_{src}.W \backslash RMW$, *otherwise,* $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf} \cup \{\langle i, a\rangle\}$ *such that* $(i \in H_{src}.W) \wedge (loc(i) = x) \wedge (i \xrightarrow{\mathtt{po}} a) \wedge (\forall j \in H_{src}.\mathtt{E}, (loc(j) = x) \wedge (j \xrightarrow{\mathtt{po}} a) \Rightarrow (j \xrightarrow{\mathtt{po}} i))$.
- $H_{src}.\mathtt{to} = H_{tgt}.\mathtt{to}$
- $H_{src}.\mathtt{IW} = H_{tgt}.\mathtt{IW}$

and H_{src} *is* JAM_{21}*-consistent.*

Please see the full version of this paper for the proof.

Counter Example

We now show a counter example for invalid register promotion on locations with Volatile accesses. Consider the following program:

```
Thread0 {
  int r1 = X.getOpaque(); // 1
  int r2 = X.getOpaque(); // 2
}

Thread1 {
  int r3 = Y.getOpaque(); // 1
  int r4 = Y.getOpaque(); // 2
}

Thread2 {
  X.setOpaque(2);
  Z.setVolatile(1);
  Y.setVolatile(1);
}

Thread3 {
  Y.setVolatile(2);
  X.setVolatile(1);
}
```

An execution with the annotated values in this program is not allowed by JAM_{21}. The execution graph before the transformation is shown in Fig. 8. First note that the Volatile access on z also has Release semantics due to the monotonicity of access modes, which yields the `ra` edge in Thread 2. The total order among `push` edges gives use two cases:

1. `Wz = 1` $\xrightarrow{\mathtt{vvo}}$ `Wx = 1`. Since `Wx = 2` $\xrightarrow{\mathtt{ra}}$ `Wz = 1` and $\mathtt{ra} \subseteq \mathtt{vvo}$ and $\mathtt{vvo}^+ \subseteq \mathtt{vo}$, we have `Wx = 2` $\xrightarrow{\mathtt{vo}}$ `Wx = 1`, which contradict with the `co` edge established by the observation from Thread 0.
2. `Wy = 2` $\xrightarrow{\mathtt{vvo}}$ `Wy = 1`. This contradict with the `co` edge established by the observation from Thread 1.

In both cases there is a contradiction (a `co` cycle). Therefore, this execution is forbidden by JAM_{21}.

In this example, the memory location z is only accessed by Thread 2. It maybe tempting to promote z to a local register on Thread 2 to reduce the number of memory instructions, which yields the following program:

```
Thread0 {
  int r1 = X.getOpaque();
  int r2 = X.getOpaque();
}

Thread1 {
  int r3 = Y.getOpaque();
  int r4 = Y.getOpaque();
}

Thread2 {
  X.setOpaque(2);
  int z = 1
  Y.setVolatile(1);
}

Thread3 {
  Y.setVolatile(2);
  X.setVolatile(1);
}
```

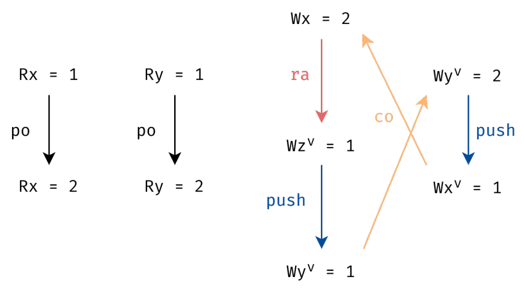

Figure 8 Before Register Promotion on Volatile access (Forbidden).

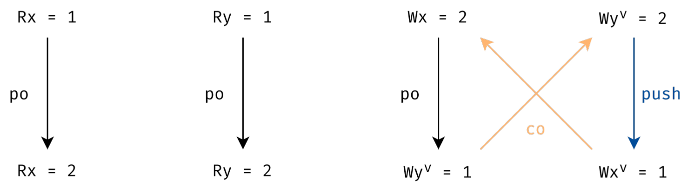

Figure 9 After Register Promotion on Volatile access (Allowed).

The execution graph after the transformation is shown in Fig. 9.

The annotated program behavior becomes allowed by JAM_{21} after the transformation. As the execution graph shows, since Volatile accesses also have cross-thread synchronization effect, we cannot simply weaken it to a Plain access without introducing new program behaviors.

5.6 Why are many transformations invalid for Volatile?

As we have shown, many local transformations are invalid for Volatile accesses under JAM_{21}. This is not a surprise and is intended to provide programmers a more intuitive semantics for Volatile accesses.

First, as we have confirmed with the author of [9], Java's Access Modes intend equivalent semantics for Volatile mode and `fullFence()`. In this way, the programmers can easily understand the semantics of both once they understand `fullFence()`. To accurately capture this intention, JAM_{21} used a fence-based approach with `push` order to model Volatile mode. As we described in Section 3, `fullFence()` in Java has cross-thread synchronization effects. As a result, any local program transformation that removes a Volatile access from the execution graph may also remove its cross-thread synchronization, and might introduce new program behavior after the transformation. Therefore, those transformations on Volatile accesses are mostly not allowed by JAM_{21}. On the other hand, the `sc` fence in C/C++11 [6] has slightly stronger synchronization effect than `sc` accesses so that they can be used to restore sequential consistency when inserted between every pair of accesses. Some of the transformations are allowed to apply to `sc` accesses but not to the fence version of the program.

In addition, restricting the set of possible transformations that is allowed to apply to Volatile variables can keep the coding process simple for programmers. From the programmers' perspective, one of the biggest challenges of developing and debugging concurrent programs comes from the compiler transformations that introduces surprising program behaviors that are not observable under sequential consistency. Therefore, restricting the set of possible transformations on Volatile accesses can restrict the set of surprising program behaviors that can happen when using Volatile mode, making the development process simpler. From this perspective, JAM_{21} provides more synchronization guarantees for Volatile mode than C/C++11 for `sc` mode atomic accesses.

Lastly, as we have confirmed with the author of [9], the current implementation of OpenJDK JVM does not apply those transformations on Volatile accesses.

6 Performance Implications

At the time of writing, the compiler bug [17] has been reported but still not resolved. The main argument against fixing the bug by inserting the missing fence instruction is that it may slow down the performance significantly. In this section, we argue that this is not the case.

The reason we only translated our `volatile-non-sc` example to Power instructions is that we only expect changes in the implementation of compilers targeting Power architectures. There is no need to change the Java compilers for x86 [15] and ARMv8 [13] all thanks to a property called *write atomicity*. Write atomicity, or *multicopy atomicity*, ensures that, when a write issued by a thread becomes observable by any other thread, it is observable by all other threads in the system. The issue that we demonstrate in this paper is caused by a write operation becoming visible to some threads before some other threads. Therefore, this violation of sequential consistency may only be observed when compiling to non-multicopy atomic architectures. If the underlying architecture ensures multicopy atomicity, then we can be sure that all writes are committed in a broadcast style and Release-Acquire semantics is sufficient. Since x86 [15] and ARMv8 [13] are multicopy atomic, we do not expect the incorrect program behavior to appear on those architectures. Therefore, no change is needed in compilers targeting multicopy-atomic architectures. In fact, we give a correctness proof for x86 in the full version of this paper to concretely show that the current compilation scheme to x86 is correct with respect to the x86-TSO memory model. Furthermore, the fence instruction that compilers use to compile to ARMv7 is the `DMB SY` instruction [8], which captures the same effects of a `fullFence()`. The only change that needs to be made is when compiling to Power instructions. This change might slow down some programs. However, relative to all other major factors that affect the performance of Java programs, we expect the impact by this change in compilers to be small.

Furthermore, symmetric to "leading fence" scheme, the "trailing fence" scheme is also valid. A correct compiler may choose to either of the schemes. Usually one may wish to choose the "trailing fence" scheme for better performance. In this case, comparing to the original compilation scheme, the fix only changes the compilation scheme for each Volatile read:

1. Remove the `hwsync` in front of the `lwz` instruction
2. Change the `lwsync` following the `lwz` instruction to `hwsync`

It is easy to see that this fix only requires, in effect, moving the `hwsync` instructions that were originally inserted before the `lwz` instruction, but does not add more. In addition, it removes the `lwsync` instructions. Therefore, we do not expect this change to the compilation scheme to have much performance impact as argued in the discussions in the bug report [17].

On the other hand, the impact of this change for compiler optimizations is unclear. That is, whether this revised compilation scheme disables some of the compiler optimizations is still a question. However, since C/C++11 compilers has long adopted this compilation scheme and performance has always been the first priority in their implementations, the possibility of disabling optimisations is unlikely. We leave a detailed empirical study for future work.

7 Related Work

7.1 Sequential Consistency Issue in C/C++11

A similar but different issue in C/C++11 memory model for atomic operations with sequentially consistent memory order was pointed out by Manerkar, et al. [11] and Lahav, et al. [6]. In particular, when using the "trailing fence" convention for compiling to Power and ARMv7 on GCC, the intended sequentially consistent semantics for certain atomic accesses can be lost due to the different placement of fences in the programs. In other words, the previous C/C++11 memory model was not able to support the two existing compilation schemes on GCC. On the other hand, JAM_{19} did not have the same problem. Since JAM_{19} defined the semantics of Volatile mode in terms of `push` orders, which emulates the effect of a full fence, it already supports and aligned with the existing compilation scheme found on OpenJDK JVMs.

The problem, however, was that the existing compilation scheme does not give sufficient synchronization to some programs with all accesses marked as Volatile. Since JAM_{19} models the problematic compilation scheme, it is necessary to repair the problem for both the compiler and the formal model.

7.2 Using Volatile to Restore Sequential Consistency in Java

Due to the complexity of the original Java Memory Model (JMM) [12], a class of bugs caused by missing "`volatile`" annotations on certain shared variables, called *missing-annotation bugs*, is found across real-world Java applications [10]. Aiming to improve the safety guarantees of the Java language, volatile-by-default JVM was proposed and developed by [10] to advocate the idea that variables should have `volatile` semantics by default and relaxed semantics by choice. Following their idea, the correctness of volatile (or Volatile mode, as they are equivalent) semantics become especially important. After all, if we cannot restore sequential consistency by annotating every variable as `volatile` (or use Volatile mode for every access), then volatile-by-default JVM would not be able to ensure intuitive program behaviors either. As of today, we are not aware of any `volatile`-by-default JVM for versions of Java after JDK9. Thus, we suggest that researchers carefully ensure the correctness of the `volatile` (or Volatile mode) implementations when implementing such JVM for Java versions after JDK9.

7.3 Memory Fairness and Compiler Transformations

Recently a declarative definition of *memory fairness* was proposed for axiomatic relaxed memory models [5]. As an improvement to the existing definition of *thread fairness*, the declarative memory fairness property can be easily integrated into axiomatic models with the No-Thin-Air restriction and can be used to prove the termination of concurrent algorithms. We noticed that the original JAM model [3] was published before this definition was proposed and therefore did not make any assertions regarding memory fairness. We leave it as our future work to verify whether memory fairness preserves the correctness of the compiler transformations and the compilation schemes.

8 Conclusion

In this paper, we have demonstrated that Java can use a compilation scheme that is similar to C/C++11. On the other hand, one should not simply compile Java's Access Modes the same way as C/C++11 compiles atomic memory orders since the formal memory models supports different compiler optimizations. In the future, we hope the bug can be resolved soon and the examples in this paper can be added to the Java Concurrency Stress Tests *jcstress* [16] tool suite to aid in maintaining the correctness of the OpenJDK HotSpot implementations.

References

1 Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2), July 2014. `doi:10.1145/2627752`.

2 ARM ARM. Architecture reference manual-armv8, for armv8-a architecture profile. *ARM Limited, Dec*, 2017.

3 John Bender and Jens Palsberg. A formalization of java's concurrent access modes. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. `doi:10.1145/3360568`.

4 Peter Sewell Jaroslav Sevcik. C/C++11 mappings to processors. Technical report, University of Cambridge, October 2016. URL: `https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`.

5 Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. Making weak memory models fair. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. `doi:10.1145/3485475`.

6 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 618–632, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3062341.3062352`.

7 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979. `doi:10.1109/TC.1979.1675439`.

8 Doug Lea. The jsr-133 cookbook for compiler writers. `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`, 2011. Last modified: Tue Mar 22 07:11:36 2011.

9 Doug Lea. Using jdk 9 memory order modes. `http://gee.cs.oswego.edu/dl/html/j9mm.html`, 2018. Last Updated: Fri Nov 16 08:46:48 2018.

10 Lun Liu, Todd Millstein, and Madanlal Musuvathi. A volatile-by-default jvm for server applications. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. `doi:10.1145/3133873`.

11 Yatin A Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Counterexamples and proof loophole for the c/c++ to power and armv7 trailing-sync compiler mappings. *arXiv preprint arXiv:1611.01507*, 2016.

12 Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. *SIGPLAN Not.*, 40(1):378–391, January 2005. `doi:10.1145/1047659.1040336`.

13 Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying arm concurrency: Multicopy-atomic axiomatic and operational models for armv8. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. `doi:10.1145/3158107`.

14 Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 175–186, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/1993498.1993520`.

15 Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010. `doi:10.1145/1785414.1785443`.

16 Aleksey Shipilev. jcstress - the java concurrency stress tests. `https://wiki.openjdk.java.net/display/CodeTools/jcstress`, 2017. Last Updated: Wed Dec 05 13:55 2018.

17 Aleksey Shipilev. [JDK-8262877] PPC sequential consistency problem: volatile stores are too weak. Technical report, OpenJDK Bug System, March 2021. URL: `https://bugs.openjdk.java.net/browse/JDK-8262877`.