Compiling Volatile Correctly in Java

Shuyang Liu
University of California, Los Angeles, United States

John Bender
Sandia National Laboratories, United States

Jens Palsberg
University of California, Los Angeles, United States

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.2022.23

Acknowledgements We thank Doug Lea for the helpful insights on the Java language semantics and compilers; 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 useful feedback. This material is based upon work supported by the National Science Foundation under Grant No. 1815496.

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

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:71
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

all accesses marked Volatile. We have contacted the maintainers of the OpenJDK about this issue and a bug report has been filed [18].

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, $\mathtt{Rx}^{\mathtt{v}}$ 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) ⟶ (b) ⟶ (c) ⟶ (d) ⟶ (e) ⟶ (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.

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`

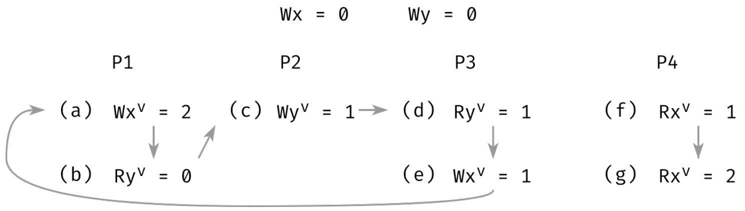


Figure 1 `volatile-non-sc.4` under the sequential consistency model, Forbidden

```
                      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

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]:

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

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].

The Power memory model [14] allows the behavior annotated in Fig. 2. The full trace of the execution can be found in Appendix M. 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

[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 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.

```
                 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

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:

$$
\begin{aligned}
&\mathtt{R}^{\mathtt{V}} \rightsquigarrow \mathtt{hwsync\ ;\ lwz\ ;\ lwsync} \\
&\mathtt{W}^{\mathtt{V}} \rightsquigarrow \mathtt{hwsync\ ;\ stw}
\end{aligned}
$$

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 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.

[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.

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 $H.\texttt{RMW} \subseteq H.\texttt{W}$ and $H.\texttt{RMW} \subseteq H.\texttt{R}$. 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 $\texttt{R}_1|\texttt{R}_2$, intersection $\texttt{R}_1\&\texttt{R}_2$, 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 Appendix A. 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 $\sqsubseteq$:

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

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 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.

```
ra ≜ po ; [REL | V] | [ACQ | V] ; po
```

[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.

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

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($\xrightarrow{\texttt{po | rf}}$)

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 Appendix H and Appendix I.

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 Appendix K.1.

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 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

[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.

volatile-non-sc.5.ppc, and the tests are indeed allowed by the Power memory model. Please see Appendix K.2 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 to Appendix D.1 and its full definition in Appendix B. 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 Appendix C for the full semantics):

- $\mathtt{po}$ and $\mathtt{rf}$ follows the same definitions as in JAM_{21} (as described in Section. 3).
- $\mathtt{co}$ 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 $i_1 \xrightarrow{\mathtt{co}} i_2$, then $i_1 \xrightarrow{\mathtt{coe}} i_2$.
- $\mathtt{ctrl}$ is the control dependency between memory accesses.
- $\mathtt{ppo}$ is the set of preserved program orders. The detailed definition can be found in Appendix C.
- $\mathtt{chapo} \triangleq \mathtt{rfe} \mid \mathtt{fre} \mid \mathtt{coe} \mid (\mathtt{fre}\ ;\ \mathtt{rfe}) \mid (\mathtt{coe}\ ;\ \mathtt{rfe})$
- $\mathtt{com} \triangleq \mathtt{rf} \mid \mathtt{fr} \mid \mathtt{co}$
- $\mathtt{po\text{-}loc}$ is a subset of $\mathtt{po}$ that relates accesses to the same locations.
- $\mathtt{rmw}$ relates the read and the write access from the same $\mathtt{RMW}$ memory event.
- $\mathtt{hb} \triangleq \mathtt{ppo} \mid (\mathtt{sync} \mid \mathtt{lwsync}) \mid \mathtt{rfe}$
- $\mathtt{propbase} \triangleq ((\mathtt{sync} \mid \mathtt{lwsync})\ \mid\ (\mathtt{rfe}\ ;\ (\mathtt{sync} \mid \mathtt{lwsync})))\ ;\ \mathtt{hb}^*$
- $\mathtt{prop} \triangleq \mathtt{propbase} \,\&\, (\mathtt{W} * \mathtt{W}) \mid (\mathtt{chapo}?\ ;\ \mathtt{propbase}^*\ ;\ \mathtt{sync}\ ;\ \mathtt{hb}^*)$
- Additional order definitions can be found in Appendix C.

▸ 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: $\mathtt{po\text{-}loc}\ \mid\ \mathtt{com}$ is acyclic.
2. Atomicity: $\mathtt{rmw}\ \&\ (\mathtt{fre}\ ;\ \mathtt{coe})$ is empty.
3. No-Thin-Air: $\mathtt{hb}$ is acyclic.
4. Propagation: $(\mathtt{co}\ \mid\ \mathtt{prop})$ is acyclic.
5. Observation: $\mathtt{fre} ; \mathtt{prop} ; \mathtt{hb}^*$ is irreflexive.
6. SCXX: $\mathtt{co}\ \mid\ (\mathtt{po} \,\&\, (\mathtt{X} * \mathtt{X}))$ is acyclic (where $\mathtt{X}$ denotes atomic accesses)

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

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

```
          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

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:

- $H_{tgt}.\mathtt{IW} = H_{src}.\mathtt{IW}$
- $H_{tgt}.\mathtt{FW} = H_{src}.\mathtt{FW}$
- $H_{tgt}.\mathtt{E} = H_{src}.\mathtt{E}$
- $H_{tgt}.\mathtt{rf} = H_{src}.\mathtt{rf}$
- $H_{tgt}.\mathtt{po} = H_{src}.\mathtt{po}$
- $H_{tgt}.\mathtt{co} \subseteq H_{src}.\mathtt{to}$
- 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 $i_R^{\sqsupseteq \mathsf{O}} \in H_{src}.\mathtt{R}$, $i_1 \in H_{src}.\mathtt{E}$ and $i_R^{\sqsupseteq \mathsf{O}} \xrightarrow{\mathtt{po}} i_1$, 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 Appendix D, 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 1 (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 Appendix D 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 1 (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 Appendix D for the proof.

▸ Corollary 1 (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 Appendix D 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, see Appendix F). 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.

▸ Definition 8 (Valid Program Transformation). Let P_{src} be a Java program which has a set of candidate executions, $Histories(P_{src})$. Let $T : \mathbb{H} \rightarrow \mathbb{H}$ 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.

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

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 2 (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:

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

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 3 (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.

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. ◀

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

Figure 6 Allowed Deordering Pairs in JAM_{21}

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}.

5.3.0.1 Deordering

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

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

▸ Definition 9 (Adjacent Events). Two events a and b are **adjacent** in a partial order 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 po edges that does not impose any synchronization in the program. Therefore, deordering (removing the po edge) does not introduce new program behavior.

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 4 (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

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	$\mathtt{W}^{m};\mathtt{R}^{\mathsf{acq}} \rightsquigarrow \mathtt{W}^{m}$ $\mathtt{W}^{\mathsf{sc}};\mathtt{R}^{\mathsf{sc}} \rightsquigarrow \mathtt{W}^{\mathsf{sc}}$ $\mathtt{RMW}^{m};\mathtt{R}^{m_r\sqsubseteq m} \rightsquigarrow \mathtt{RMW}^{m}$	$\mathtt{W}^{m};\mathtt{R}^{m\sqsubseteq\mathsf{Opq}} \rightsquigarrow \mathtt{W}^{m}$ ✗ $\mathtt{RMW}^{m};\mathtt{R}^{m\sqsubseteq\mathsf{Opq}} \rightsquigarrow \mathtt{RMW}^{m}$
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

- $H_{src}.\mathtt{po} = H_{tgt}.\mathtt{po} \cup \{\langle a, b\rangle\}$ where a and b are $\mathtt{po}$-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 Appendix G 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 2 (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 5 (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:

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

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

Please see Appendix G 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 Appendix G.

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 6 (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:

- $H_{src}.\texttt{po} = H_{tgt}.\texttt{po} \cup \{\langle a, b\rangle\} \cup \{\langle i, a\rangle \mid i \xrightarrow{\texttt{po}} b\} \cup \{\langle a, j\rangle \mid b \xrightarrow{\texttt{po}} j\}$
- $H_{src}.\texttt{rf} = H_{tgt}.\texttt{rf}$

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

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

Please see Appendix G for the proof.

Note that write-write merging is not valid for Volatile mode writes. We provide an example of this in Appendix G.

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 counterexample in Appendix G to show that write/RMW-read merging is invalid when the read is (or is stronger than) Acquire mode.

▸ Theorem 7 (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:

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

Please see Appendix G 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 8 (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:

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

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

Please see Appendix G 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 9 (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 $H_{tgt}.AccessMode(a) \in \{\text{Release, Acquire}\}$. Let $b \notin H_{tgt}.\mathtt{E}$, there exists a H_{src} such that:

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

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

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

Please see Appendix G 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 10 (Weakening for non-shared variable). Let H_{tgt} be a JAM_{21}-consistent execution such that, for all accesses i and j in $H_{tgt}.\texttt{E}$, $loc(i) = loc(j) = x \Rightarrow Tid(i) = Tid(j)$ for some memory location x. In addition, $\forall i \in H_{tgt}.\texttt{E}, loc(i) = x \Rightarrow AccessMode(i) = \textsf{Plain}$. There exists an execution H_{src} such that:

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

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

Please see Appendix G for the proof.

▸ **Theorem 11** (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 Appendix G for the proof.

5.5.0.1 Counter Example

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

```
Thread0 {                                   Thread2 {
  int r1 = X.getOpaque(); // 1                X.setOpaque(2);
  int r2 = X.getOpaque(); // 2                Z.setVolatile(1);
}                                             Y.setVolatile(1);
                                            }
Thread1 {
  int r3 = Y.getOpaque(); // 1              Thread3 {
  int r4 = Y.getOpaque(); // 2                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 $\mathtt{ra}$ edge in Thread 2. The total order among $\mathtt{push}$ edges gives use two cases:

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

In both cases there is a contradiction (a $\mathtt{co}$ 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:

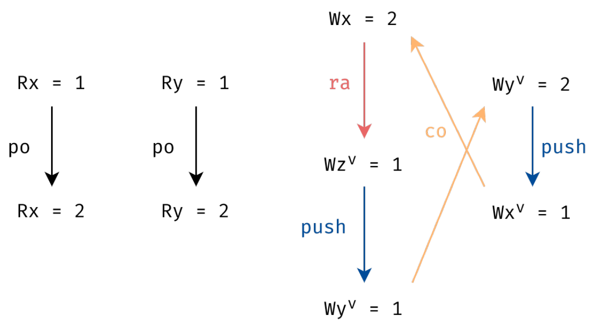


Figure 8 Before Register Promotion on Volatile access (Forbidden)

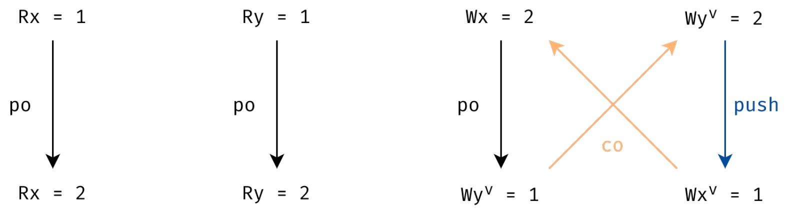


Figure 9 After Register Promotion on Volatile access (Allowed)

```
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);
}
```

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 [18] 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 Appendix F 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 [18].

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* [17] 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, 10 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. *arXiv preprint arXiv:2012.01067*, 2020.

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, page 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, page 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 Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988. URL: `http://doi.acm.org/10.1145/42190.42277`, `doi:10.1145/42190.42277`.

17 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.

18 Aleksey Shipilev. [JDK-8262877] PPC sequential consistency problem: volatile stores are too weak. Technical report, OpenJDK Bug System, 03 2021. URL: `https://bugs.openjdk.java.net/browse/JDK-8262877`.

19 Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. Repairing and mechanising the javascript relaxed memory model. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 346–361, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3385412.3385973`.

A the Full JAM_{21} Model

```
let opq = O | RA | V
let rel = W & (RA | V)
let acq = R & (RA | V)
let f_rel = REL | V
let f_acq = ACQ | V
let vol = V
let fence = F

(* volatile accesses extend push order *)
let svo = po;[fence & f_rel];po;[W] | [R];po;[fence & f_acq];po
let spush = po;[fence & vol];po

(* release acquire ordering *)
let ra = po;[rel] | [acq];po

(* intra thread volatile ordering *)
let volint =  [vol];po;[vol] (* OLD: po;[vol & R] | [vol & W];po *)

(* intrathread ordering contraints *)
let into = svo | spush | ra | volint

(* cross thread push ordering extended with volatile memory accesses *)
let push = spush | volint
with pushto from linearisations(domain(push), ((W * FW) & loc & ~id) | rf | po)

(* extend ra visibility *)
let vvo = rf | svo | ra | push | pushto;push
let vo = vvo+ | po-loc

include "filters.cat" (* includes WW filter *)
let WWco(rel) = WW(rel) & loc & ~id
(* final writes are co-after everything *)
let cofw = WWco((W * FW))

(* jam coherence *)
let coww = WWco(vo)
let cowr = WWco(vo;invrf)
let corw = WWco(vo;po)
let corr = [opq] ; WWco(rf;po;invrf) ; [opq]
let coinit = loc & IW*(W\IW)

include "cross.cat"
let co0 = loc & (IW * (W \ IW)|(W \ FW) * FW)
with cormwtotal from generate_orders(RMW, co0)

let rec co-jom = coww | cowr | corw | corr | cormwtotal
  | WWco((rf;[RMW])^-1;co-jom) | coinit | cofw

acyclic (po | rf) ; [opq] as no-thin-air
acyclic co-jom as coherence
```

B the Full JAM'_{21} Model

```
include "filters.cat"
include "cross.cat"
let WWco(rel) = WW(rel) & loc & ~id
let co0 = loc & (IW * (W \ IW)|(W \ FW) * FW)
with cormwtotal from generate_orders(RMW, co0)

let opq = O | RA | V
let rel = W & (RA | V)
let acq = R & (RA | V)
let f_rel = REL | V
let f_acq = ACQ | V
let vol = V
let fence = F

(* volatile accesses extend push order *)
let svo = po;[fence & f_rel];po;[W] | [R];po;[fence & f_acq];po
let spush = po;[fence & vol];po

(* release acquire ordering *)
let ra = po;[rel] | [acq];po

(* intra thread volatile ordering *)
let volint =  [vol];po;[vol] (* OLD: po;[vol & R] | [vol & W];po *)

(* intrathread ordering contraints *)
let into = svo | spush | ra | volint
let push = spush | volint

let rec co-jom = coww | cowr | corw | corr | cormwtotal
  | WWco((rf;[RMW])^-1;co-jom) | coinit | cofw

and fr-jom = rf^-1 ; co-jom
and fr-jom-e = fr-jom & ext
and co-jom-e = co-jom & ext
and chapo = rfe | fr-jom-e | co-jom-e | (fr-jom-e ; rfe) | (co-jom-e ; rfe)

(* extend ra visibility *)
and vvo = rf | svo | ra | push | push ; chapo ; push
and vo = vvo+ | po-loc

and cofw = WWco((W * FW))
and coww = WWco(vo)
and cowr = WWco(vo;invrf)
and corw = WWco(vo;po)
and corr = [opq] ; WWco(rf;po;invrf) ; [opq]
and coinit = loc & IW*(W\IW)

acyclic (po | rf) ; [opq] as no-thin-air
acyclic co-jom as coherence
```

C The Power Memory Model in Herd7

```
PPC
(* Model for Power *)
include "cos.cat" (* Used to compute the coherence order*)

(* Uniproc *)
acyclic po-loc | rf | fr | co as scperlocation

(* Atomic *)
empty rmw & (fre;coe) as atomic

(* Utilities *)
let dd = addr | data
let rdw = po-loc & (fre;rfe)
let detour = po-loc & (coe ; rfe)
let addrpo = addr;po

(*******)
(* ppo *)
(*******)

let sync = try fencerel(SYNC) with 0
let lwsync = try fencerel(LWSYNC) with 0
let eieio = try fencerel(EIEIO) with 0
let isync = try fencerel(ISYNC) with 0
show sync,lwsync,eieio

(* Dependencies *)
show data,addr
let ctrlisync = try ctrlcfence(ISYNC) with 0
show ctrlisync
show isync  ctrlisync as isync
show ctrl  ctrlisync as ctrl
show isync,ctrlisync

(* Initial value *)
let ci0 = ctrlisync | detour
let ii0 = dd | rfi | rdw
let cc0 = dd | po-loc | ctrl | addrpo
let ic0 = 0

(* Fixpoint from i -> c in instructions and transitivity *)
let rec ci = ci0 | (ci;ii) | (cc;ci)
and ii = ii0 | ci | (ic;ci) | (ii;ii)
and cc = cc0 | ci | (ci;ic) | (cc;cc)
and ic = ic0 | ii | cc | (ic;cc) | (ii ; ic) (* | ci inclus dans ii et cc *)

let ppo =
  let ppoR = ii & (R * R)
  and ppoW = ic & (R * W) in
    ppoR | ppoW

(* fences *)

let lwsync = lwsync  (W * R)
let eieio = eieio & (W * W)

(* All arm barriers are strong *)
let strong = sync
let light = lwsync|eieio

let fence = strong|light

(* happens before *)
let hb = ppo | fence | rfe
```

```
acyclic hb as thinair

(* prop *)
let hbstar = hb*
let propbase = (fence|(rfe;fence));hbstar

let chapo = rfe|fre|coe|(fre;rfe)|(coe;rfe)

let prop = propbase & (W * W) | (chapo? ; propbase*; strong; hbstar)

acyclic co|prop as propagation
irreflexive fre;prop;hbstar as observation

let xx = po & (X * X)
acyclic co | xx as scXX
```

D A Proof of Compilation Correctness to Power

D.1 The JAM'_{21} Model

The motivation of JAM'_{21} is to enable simpler compilation proofs. JAM_{21} enforces a total order among `fullFence()`s, which introduces complexity when proving compilation correctness. Therefore, we introduce an intermediate memory model JAM'_{21} that is observationally equivalent to the JAM_{21}. Thus, we start by defining and proving the observational equivalence of the two models. Then we use JAM'_{21} to prove the correctness of the compilation schemes.

The JAM'_{21} model is the same as JAM_{21} except for the semantics of full fences. Instead of having a total order on full fences, JAM'_{21} only enforces order when there is a communication edge. The full semantics of JAM'_{21} can be found in Appendix B. Here we only include the updated portion. The definition for `chapo` is newly added[6]. The cross-thread synchronization effect of `fullFence()`s is then defined as `push`;`chapo`;`push` (instead of `pushto`;`push` as before):

$$\texttt{chapo} \triangleq \texttt{rfe}\,|\,\texttt{fre}\,|\,\texttt{coe}\,|\,(\texttt{fre}\ ;\ \texttt{rfe})\,|\,(\texttt{coe}\ ;\ \texttt{rfe})$$
$$\texttt{vvo} \triangleq \ldots\,|\,\texttt{push}\ ;\ \texttt{chapo}\ ;\ \texttt{push}$$

The rest of JAM'_{21} are the same as JAM_{21}.

▸ Definition 10 (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**.

▸ Lemma 2. Observational equivalence is a transitive relation. That is, if H and H' are observationally equivalent, and H' and H'' are observationally equivalent, then H and H'' are observationally equivalent.

Proof. The transitivity follows directly from the transitivity of set equivalence in the definition. ◂

▸ Lemma 3. (⇒) Given a program P, for any JAM_{21}-consistent execution $H \in Histories_{JAM}(P)$, there exists a $H' \in Histories_{JAM'}$ such that H' is observationally equivalent to H.

Proof. Given a program P, it's obvious that there exists an H' that is observationally equivalent to H. We prove that $H' \in Histories_{JAM'}(P)$. That is, let H' be a candidate execution of P and H' is observationally equivalent to H. We show that H' is JAM'_{21}-consistent. Since the only difference between JAM_{21} and JAM'_{21} is at the effect of full fences, we focus on this part in our proof. In particular, we first show that, if $i_1 \xrightarrow{\texttt{push}} i_3$, $i_2 \xrightarrow{\texttt{push}} i_4$, and $i_3 \xrightarrow{\texttt{chapo}} i_2$, then it must be that $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$. We can prove this by analyzing five cases:

[6] The name `chapo` comes directly from the Power Memory Model in the Herd [1] repository. We use the same name here so that the readers can easily relate them

1. $i_3 \xrightarrow{\texttt{rf}} i_2$: then we have $i_1 \xrightarrow{\texttt{push}} i_3 \xrightarrow{\texttt{rf}} i_2 \xrightarrow{\texttt{push}} i_4$, which is equivalent to $i_1 \xrightarrow{\texttt{vvo}} i_3 \xrightarrow{\texttt{vvo}} i_2 \xrightarrow{\texttt{vvo}} i_4$, which means $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$ (as it'd create a vo cycle in the latter case).
2. $i_3 \xrightarrow{\texttt{co}} i_2$: then it cannot be $i_2 \xrightarrow{\texttt{vo}} i_3$ on the right side as it immediately gives us a coherence cycle by coww. So it must be $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$.
3. $i_3 \xrightarrow{\texttt{fr}} i_2$: then there exists a write event W such that $W \xrightarrow{\texttt{rf}} i_3$ and $W \xrightarrow{\texttt{co}} i_2$. If we have $i_2 \xrightarrow{\texttt{vo}} i_3$ then we would have $i_2 \xrightarrow{\texttt{co}} W$ by cowr, which gives us a coherence cycle. Therefore it must be that $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$.
4. $i_3 \xrightarrow{\texttt{co}} W_1 \xrightarrow{\texttt{rf}} i_2$: if we have $i_2 \xrightarrow{\texttt{vo}} i_3$ then, because rf is also a visibility order, we have $W_1 \xrightarrow{\texttt{vo}} i_3$. By coww we get $W_1 \xrightarrow{\texttt{co}} i_3$, contradicting with the earlier assumption that $i_3 \xrightarrow{\texttt{co}} W_1$. So it must be that $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$.
5. $i_3 \xrightarrow{\texttt{fr}} W_1 \xrightarrow{\texttt{rf}} i_2$: then it means there is W_2 such that $W_2 \xrightarrow{\texttt{rf}} i_3$ and $W_2 \xrightarrow{\texttt{co}} W_1$. If we have $i_2 \xrightarrow{\texttt{vo}} i_3$, since rf is also a visibility order, we have $W_1 \xrightarrow{\texttt{vo}} i_3$. By cowr, we have $W_1 \xrightarrow{\texttt{co}} W_2$, contradicting with the assumption of $W_2 \xrightarrow{\texttt{co}} W_1$ earlier. So it must be $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$.

Thus we have shown that if we have $i_1 \xrightarrow{\texttt{push}} i_3$, $i_2 \xrightarrow{\texttt{push}} i_4$, and $i_3 \xrightarrow{\texttt{chapo}} i_2$, then it must be that $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$.

We now show that H' is JAM'_{21}-consistent. The No-Thin-Air requirement is automatically fulfilled since $H.\texttt{E} = H'.\texttt{E}$, $H.\texttt{po} = H'.\texttt{po}$ and $H.\texttt{rf} = H'.\texttt{rf}$.

Previously, we have shown that $i_1 \xrightarrow{\texttt{push}} i_3$, $i_2 \xrightarrow{\texttt{push}} i_4$, and $i_3 \xrightarrow{\texttt{chapo}} i_2$ implies $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$ in H. Since H is JAM_{21}-consistent, then co-jom is acyclic with either $i_1 \xrightarrow{\texttt{vo}} i_4$ or $i_2 \xrightarrow{\texttt{vo}} i_3$. Now we have two cases:

1. No communication: then we do not have any extra vo edge we can use to infer in H' either. Since co-jom is acyclic in H, and $H.\texttt{vo} = H'.\texttt{vo}$, co-jom is acyclic in H', too.
2. With communication chapo: suppose $i_3 \xrightarrow{\texttt{chapo}} i_2$ (the other direction is symmetrically the same), then it must be that $i_1 \xrightarrow{\texttt{vo}} i_4$ and not $i_2 \xrightarrow{\texttt{vo}} i_3$ in H. Since H is JAM_{21}-consistent, we know that $i_1 \xrightarrow{\texttt{vo}} i_4$ cannot lead to any co-jom cycle. In H', we use (vo-5') to infer that $i_1 \xrightarrow{\texttt{vo}} i_4$. Since other portions of H' satisfies the above conditions, we can infer that $i_1 \xrightarrow{\texttt{vo}} i_4$ cannot lead to any co-jom cycle in H' either.

Since neither case leads to a co-jom cycle in H', we can conclude that H' is JAM'_{21}-consistent and hence $H' \in Histories_{JAM'}(P)$

◀

▶ Lemma 4. Let fullfence-vo order be the vo order derived using the rule (vo-5) of JAM_{21}. In JAM_{21}, for any co-jom cycle derived from the fullfence-vo orders, there is a vo*; fullfence-vo; vo*; chapo cycle.

Proof. Let i_1, i_2, i_3 and i_4 be four events in an execution history H such that $i_1 \xrightarrow{\texttt{push}} i_3$ and $i_2 \xrightarrow{\texttt{push}} i_4$. By (vo-5), we derive the condition $(i_1 \xrightarrow{\texttt{fullfence-vo}} i_4) \vee (i_2 \xrightarrow{\texttt{fullfence-vo}} i_3)$. Suppose that H is not consistent due to this condition under JAM_{21}, i.e., following either side of the disjunction we can derive a co-jom cycle. We analyze one side of the disjunction since the other side of the disjunction symmetrically follow the same reasoning. We analyze each possible rule to derive a coherence cycle. Since fullfence-vo is included in vo order, we only need to analyze the cases where vo appears:

- coww. If we derived the coherence cycle from coww rule, then it means there exists W_1 and W_2 such that $W_1 \xrightarrow{\texttt{vo}} W_2$ and $W_2 \xrightarrow{\texttt{co}} W_1$. The fact that $i_1 \xrightarrow{\texttt{vo}} i_4$ "enables" us (we can only use rule (vo-7) here as other visibility rules imply program structures that does not

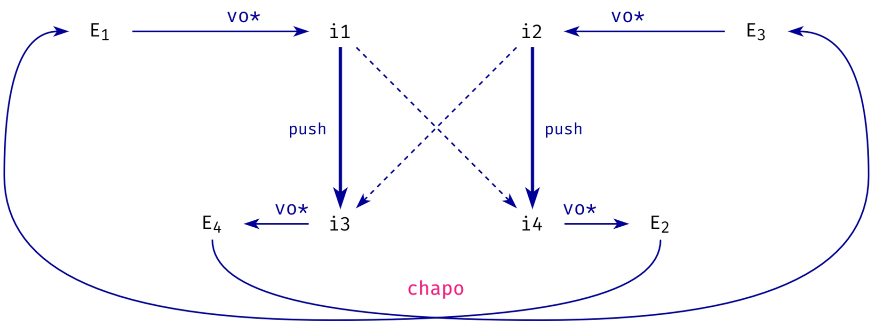


Figure 10 $i_1 \xrightarrow{\texttt{push}} i_3 \xrightarrow{\texttt{vo}}{}^* E_4 \xrightarrow{\texttt{chapo}} E_3 \xrightarrow{\texttt{vo}}{}^* i_2 \xrightarrow{\texttt{push}} i_4 \xrightarrow{\texttt{vo}}{}^* E_2 \xrightarrow{\texttt{chapo}} E_1$ cycle

require $i_1 \xrightarrow{\texttt{vo}} i_4$) to derive this contradiction implies the following structure: $W_1 \xrightarrow{\texttt{vo}}{}^* i_1 \xrightarrow{\texttt{vo}} i_4 \xrightarrow{\texttt{vo}}{}^* W_2 \xrightarrow{\texttt{co}} W_1$, revealing a cycle of $\texttt{vo}^*$; $\texttt{fullfence-vo}$; $\texttt{vo}^*$; $\texttt{chapo}$.

- `cowr`. If we derived the coherence cycle from `cowr` rule, then it means there exists R_1, W_1, and W_2 such that $W_1 \xrightarrow{\texttt{rf}} R_1$, $W_2 \xrightarrow{\texttt{vo}} R_1$, and $W_1 \xrightarrow{\texttt{co}} W_2$. The fact that $i_1 \xrightarrow{\texttt{vo}} i_4$ "enables" us to derive this contradiction implies the following structure: $W_2 \xrightarrow{\texttt{vo}}{}^* i_1 \xrightarrow{\texttt{vo}} i_4 \xrightarrow{\texttt{vo}}{}^* R_1$. Because $W_1 \xrightarrow{\texttt{co}} W_2$ and $W_1 \xrightarrow{\texttt{rf}} R_1$, we have $R_1 \xrightarrow{\texttt{fr}} W_2$. We now have a cycle $W_2 \xrightarrow{\texttt{vo}}{}^* i_1 \xrightarrow{\texttt{vo}} i_4 \xrightarrow{\texttt{vo}}{}^* R_1 \xrightarrow{\texttt{fr}} W_2$, which is a cycle of $\texttt{vo}^*$; $\texttt{fullfence-vo}$; $\texttt{vo}^*$; $\texttt{chapo}$.
- `corw`. If we derived the coherence cycle from `corw` rule, then it means there exists R_1, W_1, and W_2, such that $W_1 \xrightarrow{\texttt{rf}} R_1$, $R_1 \xrightarrow{\texttt{vo}} W_2$, and $W_2 \xrightarrow{\texttt{co}} W_1$. The fact that $i_1 \xrightarrow{\texttt{vo}} i_4$ "enables" us to derive this contradiction implies the following structure: $R_1 \xrightarrow{\texttt{vo}}{}^* i_1 \xrightarrow{\texttt{vo}} i_4 \xrightarrow{\texttt{vo}}{}^* W_2 \xrightarrow{\texttt{co}} W_1 \xrightarrow{\texttt{rf}} R_1$, which is a cycle of $\texttt{vo}^*$; $\texttt{fullfence-vo}$; $\texttt{vo}^*$; $\texttt{chapo}$.

◀

▶ LEMMA 5. (⇐) Given a program P, for any JAM'_{21}-consistent execution $H' \in \mathit{Histories}_{JAM'}(P)$, there exists an execution history $H \in \mathit{Histories}_{JAM}(P)$ such that H is observationally equivalent to H'.

Proof. Given a program P, it's obvious that there exists an H that is observationally equivalent to H'. We prove that $H \in \mathit{Histories}_{JAM}(P)$. That is, let H be a candidate execution of P and H is observationally equivalent to H'. We show that H is JAM_{21}-consistent. To help the reader better understand this, consider Fig. 10. Suppose H is an execution history that is forbidden by the rules of JAM_{21}, we show that its corresponding H' is also forbidden by JAM'_{21}. Since the only difference between JAM_{21} and JAM'_{21} is at the effects of full fences, we only analyze that part. In other words, H is forbidden by JAM_{21} precisely due to the total order of full fences. Let i_1, i_2, i_3, and i_4 be events in H such that $i_1 \xrightarrow{\texttt{push}} i_3$ and $i_2 \xrightarrow{\texttt{push}} i_4$. By Lemma 4, we can generalize the structure and infer that there are E_1, E_2, E_3, and E_4 such that $E_1 \xrightarrow{\texttt{vo}}{}^* i_1$, $E_3 \xrightarrow{\texttt{vo}}{}^* i_2$, $i_4 \xrightarrow{\texttt{vo}}{}^* E_2$, and $i_3 \xrightarrow{\texttt{vo}}{}^* E_4$. In addition, we also have $E_2 \xrightarrow{\texttt{chapo}} E_1$ and $E_4 \xrightarrow{\texttt{chapo}} E_3$. Because H is forbidden under JAM_{21}, it means we have two cycles, $i_1 \xrightarrow{\texttt{vo}} i_4 \xrightarrow{\texttt{vo}}{}^* E_2 \xrightarrow{\texttt{chapo}} E_1 \xrightarrow{\texttt{vo}}{}^* i_1$ and $i_2 \xrightarrow{\texttt{vo}} i_3 \xrightarrow{\texttt{vo}}{}^* E_4 \xrightarrow{\texttt{chapo}} E_3 \xrightarrow{\texttt{vo}}{}^* i_2$, so that no matter which side of the disjunction we choose, we always end up with a contradiction. In H', on the other hand, we do not have $i_1 \xrightarrow{\texttt{vo}} i_4$ or $i_2 \xrightarrow{\texttt{vo}} i_3$. However, despite the absence of the two edges, we now have a larger cycle: $i_1 \xrightarrow{\texttt{push}} i_3 \xrightarrow{\texttt{vo}}{}^* E_4 \xrightarrow{\texttt{chapo}} E_3 \xrightarrow{\texttt{vo}}{}^* i_2 \xrightarrow{\texttt{push}} i_4 \xrightarrow{\texttt{vo}}{}^* E_2 \xrightarrow{\texttt{chapo}} E_1$, which forms a $\texttt{vo}$ cycle by ($\texttt{vo}$-5') in JAM'_{21}. Therefore, execution history H' is forbidden under JAM'_{21}

as well, which contradicts to our previous assumption. Thus, since $H' \in Histories_{JAM'}(P)$ implies that H' is JAM'_{21}-consistent, $H \in Histories_{JAM}(P)$. ◀

Essentially, the replacement of the (vo-5) rule give no actual effect in forbidding executions. In JAM_{21}, we look for two vo cycles to forbid an execution, whereas in JAM'_{21} we combine the two cycles into one to forbid the execution.

▸ Theorem 12 (Observational Equivalence of JAM_{21} and JAM'_{21}). JAM'_{21} is observationally equivalent to JAM_{21}.

Proof. Given a program P, let $Histories_{JAM}(P)$ be the set of JAM_{21}-consistent candidate executions of P and $Histories_{JAM'}(P)$ be the set of JAM'_{21}-consistent candidate executions of P. By Lemma 3, we know that for all $H \in Histories_{JAM}(P)$, there exists an $H' \in Histories_{JAM'}(P)$ such that H' and H are observationally equivalent. Similarly, by Lemma 5, we know that for all $H' \in Histories_{JAM'}(P)$, there exists an $H \in Histories_{JAM}(P)$ such that H and H' are observationally equivalent. Combining together, we can conclude that JAM_{21} and JAM'_{21} are observationally equivalent. ◀

▸ Corollary 3. JAM'_{21} satisfies the same important properties (Theorem 14, Theorem 15, Theorem 16, Theorem 17 and Corollary 4) in Section. H.

Proof. Since the definitions in JAM'_{21} are the same as JAM_{21} except for the semantics of `fullFences`, JAM'_{21} automatically satisfies Theorem. 14, Theorem. 15, Theorem 16, and Theorem. 17. By Theorem. 12, JAM_{21} and JAM'_{21} allow the same set of execution histories up to observational equivalence. Therefore, JAM'_{21} also satisfies Corollary. 4. ◀

D.2 Compilation to Power

▸ Lemma 1 (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 $H_{src} \rightsquigarrow H_{tgt}$.

Proof. It is obvious that there exists a candidate execution history H_{src} of P_{src} such that $H_{src} \rightsquigarrow H_{tgt}$. We show that $H_{src} \in Histories_{JAM'}(P)$. That is, H_{src} is JAM'_{21}-consistent. In order to be consistent under JAM'_{21}, we need H_{src} to satisfy two requirements:

1. No-Thin-Air Requirement: (po | rf) is acyclic. The intra-thread rfi order cannot contradicting the po given that H_{tgt} is Power-consistent. Therefore, we need to show that (po | rfe) is acyclic. Note that since the only inter-thread order is rfe. If there is a cycle in (po | rfe), then the head of each thread is a R^{Opq} and the last event in each thread participating in this cycle is a W^{Opq}, where $R^{\mathsf{Opq}} \xrightarrow{\mathtt{po}} W^{\mathsf{Opq}}$ in H_{src}. Using a compiler that follows the compilation scheme, this translates to $R \xrightarrow{\mathtt{ctrl}} W$ in H_{tgt}. Further we can infer that $R \xrightarrow{\mathtt{hb}} W$ in H_{tgt}. Power ensures that (hb | rf) is acyclic. Therefore, if there is a cycle of (po | rf) in H_{src}, H_{tgt} would not be consistent under Power's memory model, contradicting to our previous assumption. For readers who do not care about this guarantee, `getOpaque()` can be directly compiled to a `lwz` instruction.
2. Coherence Requirement: co-jom is acyclic. We now prove that co-jom is acyclic in H_{src}. In order to show this, we show that co-jom is a partial order of co in H_{tgt}. In other words, co is a linear extension of co-jom. We prove this by assuming the opposite and deriving a contradiction.

Suppose that there exists $i_1 \xrightarrow{\texttt{co-jom}} i_2$ in H_{src} but $i_2 \xrightarrow{\texttt{co}} i_1$ in H_{tgt}. We analyze each of the possible cases where we can derive a `co-jom` order.

- coinit. This automatically gives us a contradiction since $H_{src}.\texttt{IW} = H_{tgt}.\texttt{IW}$.
- cofw. This automatically gives us a contradiction since $H_{src}.\texttt{FW} = H_{tgt}.\texttt{FW}$.
- corr. This implies that there exist R_1 and R_2 such that, $R_1 \xrightarrow{\texttt{po}} R_2$, $i_1 \xrightarrow{\texttt{rf}} R_1$, and $i_2 \xrightarrow{\texttt{rf}} R_2$. We also have $i_2 \xrightarrow{\texttt{co}} i_1$. Now the SC-per-location requirement of Power is violated, contradicting with our previous assumption that H_{tgt} is Power-consistent.
- coww. This implies that $i_1 \xrightarrow{\texttt{vo}} i_2$ but $i_2 \xrightarrow{\texttt{co}} i_1$.
  - $i_1 \xrightarrow{\texttt{po-loc}} i_2$. $i_2 \xrightarrow{\texttt{co}} i_1$ would violates the SC-Per-Location Requirement of Power, making H_{tgt} inconsistent, contradicting to our previous assumption.
  - $i_1 \xrightarrow{\texttt{svo}} i_2$ or $i_1 \xrightarrow{\texttt{ra}} i_2$ or $i_1 \xrightarrow{\texttt{push}} i_2$. In H_{tgt}, this means $i_1 \xrightarrow{\texttt{lwsync}} i_2$ or $i_1 \xrightarrow{\texttt{sync}} i_2$. Note that all three cases of them are included in program order with access to the same location. Therefore, $i_2 \xrightarrow{\texttt{co}} i_1$ would violates the SC-Per-Location Requirement of Power, making H_{tgt} inconsistent, contradicting to our previous assumption.
  - $i_1 \xrightarrow{\texttt{push}} E_1 \xrightarrow{\texttt{chapo}} E_2 \xrightarrow{\texttt{push}} i_2$ for some E_1 and E_2. Note that we have $E_1 \xrightarrow{\texttt{chapo}} E_2 \xrightarrow{\texttt{sync}} i_2$ and $i_2 \xrightarrow{\texttt{co}} i_1 \xrightarrow{\texttt{sync}} E_1$. They form a propagation (`prop`) cycle between i_2 and E_1, which makes H_{tgt} not Power-consistent, giving us a contradiction.
  - $i_1 \xrightarrow{\texttt{vvo}} i_3 \xrightarrow{\texttt{vvo}} i_2$. This corresponds to the inductive case where the visibility order is formed by two visibility orders through another event, i_3. Note that all the cases of `vo` orders produce propagation (`prop`) orders. Therefore, we get a violation of the Propagation requirement if $i_2 \xrightarrow{\texttt{co}} i_1$ in Power, contradicting to our previous assumption that H_{tgt} is Power-consistent.
- cowr. This implies that there exists R such that $i_2 \xrightarrow{\texttt{rf}} R$ and $i_1 \xrightarrow{\texttt{vo}} R$.
  - $i_1 \xrightarrow{\texttt{po-loc}} i_2$. $i_2 \xrightarrow{\texttt{co}} i_1$ would violates the SC-Per-Location Requirement of Power, making H_{tgt} inconsistent, contradicting to our previous assumption.
  - $i_1 \xrightarrow{\texttt{svo}} R$ or $i_1 \xrightarrow{\texttt{ra}} R$ or $i_1 \xrightarrow{\texttt{push}} R$. In H_{tgt}, this means $i_1 \xrightarrow{\texttt{lwsync}} i_2$ or $i_1 \xrightarrow{\texttt{sync}} i_2$. Note that all three cases of them are included in program order with access to the same location. Therefore, $i_2 \xrightarrow{\texttt{co}} i_1$ would violates the SC-Per-Location Requirement of Power, making H_{tgt} inconsistent, contradicting to our previous assumption.
  - $i_1 \xrightarrow{\texttt{push}} E_1 \xrightarrow{\texttt{chapo}} E_2 \xrightarrow{\texttt{push}} R$ for some E_1 and E_2. Because $i_2 \xrightarrow{\texttt{rf}} R$ and $i_2 \xrightarrow{\texttt{co}} i_1$, we have $R \xrightarrow{\texttt{fr}} i_1$. We now have $E_1 \xrightarrow{\texttt{chapo}} E_2 \xrightarrow{\texttt{sync}} R$ and $R \xrightarrow{\texttt{fr}} i_1 \xrightarrow{\texttt{sync}} E_1$. Both of them form a propagation order between R and E_1 and they contradict with each other.
  - $i_1 \xrightarrow{\texttt{vvo}} i_3 \xrightarrow{\texttt{vvo}} R$ This corresponds to the inductive case where the visibility order is formed by two visibility orders through another event, i_3. Note that all the cases of `vvo` orders produce propagation orders. Therefore, we get $R \xrightarrow{\texttt{fr}} i_1 \xrightarrow{\texttt{prop}} R$, which violates the observation requirement in Power.
- corw. This implies that there exists R such that $R \xrightarrow{\texttt{po}} i_2$ and $i_1 \xrightarrow{\texttt{rf}} R$. $i_2 \xrightarrow{\texttt{co}} i_1$ would cause a violation of SC-Per-Location requirement in Power.
- cormwexcl. This implies that i_1 is a RMW operation and there exists i_3 such that $i_3 \xrightarrow{\texttt{rf}} i_1$ and $i_3 \xrightarrow{\texttt{co}} i_2$. Having $i_2 \xrightarrow{\texttt{co}} i_1$ immediately violates the atomicity requirement of Power.
- cormwtotal. $i_1 \xrightarrow{\texttt{co}} i_2$ because $H_{tgt}.\texttt{co} \subseteq H_{src}.\texttt{to}$. Therefore, having $i_2 \xrightarrow{\texttt{co}} i_1$ in this case would yield a cycle in `co`, violating the propagation requirement in Power.

Thus, we have shown that $\texttt{co-jom}$ is a partial order of $\texttt{co}$ in H_{tgt} and the coherence requirement is automatically fulfilled because $\texttt{co}$ is acyclic. Hence, $H_{src} \in Histories_{JAM'}(P)$. ◀

▶ Theorem 1 (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 $H_{src} \rightsquigarrow H_{tgt}$.

Proof. By Lemma 1, we know that there exists an $H'_{src} \in Histories_{JAM'}(P_{src})$ such that $H'_{src} \rightsquigarrow H_{tgt}$. Therefore, by definition of the $\rightsquigarrow$ relation,

- H_{tgt} is observationally equivalent to H'_{src}
- $H_{tgt}.\texttt{co} \subseteq H'_{src}.\texttt{to}$
- If $RMW, i_1 \in H'_{src}.\texttt{E}$ and $RMW \xrightarrow{\texttt{po}} i_1$, then $RMW \xrightarrow{\texttt{ctrl}} i_1$ in H_{tgt}
- If $R^{\mathsf{Opq}}, i_1 \in H'_{src}.\texttt{E}$ and $R^{\mathsf{Opq}} \xrightarrow{\texttt{po}} i_1$, then $R \xrightarrow{\texttt{ctrl}} i_1$ in H_{tgt}
- If $i_1, i_2 \in H'_{src}.\texttt{E}$ and $i_1 \xrightarrow{\texttt{push}} i_2$, then $i_1 \xrightarrow{\texttt{sync}} i_2$ for $i_1, i_2 \in H_{tgt}.\texttt{E}$
- If $i_1, i_2 \in H'_{src}.\texttt{E}$ and $i_1 \xrightarrow{\texttt{ra}} i_2$, then $i_1 \xrightarrow{\texttt{lwsync}} i_2$ for $i_1, i_2 \in H_{tgt}.\texttt{E}$

By Theorem 12, we know that for all H'_{src}, there exists an $H_{src} \in Histories_{JAM}(P)$ such that H_{src} is observationally equivalent to H'_{src}. By Lemma 2, H_{src} is observationally equivalent to H_{tgt}. Furthermore,

- $H_{tgt}.\texttt{co} \subseteq H_{src}.\texttt{to}$ because $H_{src}.\texttt{to} = H'_{src}.\texttt{to}$.
- If $RMW, i_1 \in H_{src}.\texttt{E}$ and $RMW \xrightarrow{\texttt{po}} i_1$, then $RMW \xrightarrow{\texttt{ctrl}} i_1$ in H_{tgt} because $H_{src}.\texttt{E} = H'_{src}.\texttt{E}$ and $H_{src}.\texttt{po} = H'_{src}.\texttt{po}$.
- If $R^{\mathsf{Opq}}, i_1 \in H_{src}.\texttt{E}$ and $R^{\mathsf{Opq}} \xrightarrow{\texttt{po}} i_1$, then $R \xrightarrow{\texttt{ctrl}} i_1$ in H_{tgt} because $H_{src}.\texttt{E} = H'_{src}.\texttt{E}$ and $H_{src}.\texttt{po} = H'_{src}.\texttt{po}$.
- If $i_1, i_2 \in H_{src}.\texttt{E}$ and $i_1 \xrightarrow{\texttt{push}} i_2$, then $i_1 \xrightarrow{\texttt{sync}} i_2$ for $i_1, i_2 \in H_{tgt}.\texttt{E}$ because $\forall i \in H_{src}.\texttt{E}, H_{src}.AccessMode(i) = H'_{src}.AccessMode(i)$ and $H_{src}.\texttt{po} = H'_{src}.\texttt{po}$, which means $H_{src}.\texttt{push} = H'_{src}.\texttt{push}$.
- If $i_1, i_2 \in H_{src}.\texttt{E}$ and $i_1 \xrightarrow{\texttt{ra}} i_2$, then $i_1 \xrightarrow{\texttt{lwsync}} i_2$ for $i_1, i_2 \in H_{tgt}.\texttt{E}$ because $\forall i \in H_{src}.\texttt{E}, H_{src}.AccessMode(i) = H'_{src}.AccessMode(i)$ and $H_{src}.\texttt{po} = H'_{src}.\texttt{po}$, which means $H_{src}.\texttt{ra} = H'_{src}.\texttt{ra}$.

Therefore, we have shown that for all $H_{tgt} \in Histories_{Power}(P)$, there exists an $H_{src} \in Histories_{JAM}(P)$ such that $H_{src} \rightsquigarrow H_{tgt}$. That is, the compilation scheme shown in Fig. 4 is correct. ◀

▶ Corollary 1 (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 $H_{src} \rightsquigarrow H_{tgt}$.

Proof. It is obvious that all the properties described in Definition 4 still hold with the trailing fence convention. Most importantly, the property that transforms `push` in the source level execution to `sync` in the target level execution is preserved as long as there is a `hwsync` instruction inserted between every Volatile accesses. The trailing fence convention, if used consistently, clearly satisfy this property. Then the rest of Definition 4 is unchanged since leading/trailing fence convention only concerns the compilation schemes for Volatile accesses. Therefore, the rest of the proof for the correctness of the trailing fence convention can be naturally derived similarly. ◀

E The x86 TSO Model in Herd7

```
X86 TSO
include "x86fences.cat"
include "filters.cat"
include "cos.cat"

(* Uniproc check *)
let com = rf | fr | co
acyclic po-loc | com

(* Atomic *)
empty rmw & (fre;coe)

(* GHB *)
#ppo
let po_ghb = WW(po) | RM(po)

#mfence
let mfence = try fencerel(MFENCE) with 0
let lfence = try fencerel(LFENCE) with 0
let sfence = try fencerel(SFENCE) with 0

show data,addr,ctrl

#implied barriers
let poWR = WR(po)
let i1 = MA(poWR)
let i2 = AM(poWR)
let implied = i1 | i2

let ghb = mfence | implied | po_ghb | rfe | fr | co
show implied
acyclic ghb as tso
```

F Compilation to x86-TSO

In this section we show that the current compilation scheme to x86-TSO is correct with respect to the TSO memory model.

F.0.1 The x86-TSO Model

We use the x86-TSO model defined in Herd7 [1], and the full model can be found in Appendix E):

▸ Definition 11. An execution history H is TSO-**consistent** if it is trace coherent and satiesfies the following three requirements:

1. SC-per-location: `po-loc | com` is acyclic
2. Atomicity: `rmw & (fre ; coe)` is empty
3. Global Happens-Before: `ghb` is acyclic

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

F.0.2 Compilation Scheme

We use the following compilation scheme[7]:

```
        getOpaque() ⇝ mov
        setOpaque() ⇝ mov
       getAcquire() ⇝ mov
       setRelease() ⇝ mov
      getVolatile() ⇝ mov
      setVolatile() ⇝ mov ; mfence
     AcquireFence() ⇝ NoOp
     ReleaseFence() ⇝ NoOp
        fullFence() ⇝ mfence
        getAndAdd() ⇝ lock xaddl
 getAndAddAcquire() ⇝ lock xaddl
 getAndAddRelease() ⇝ lock xaddl
```

F.0.3 Proof of Compilation Correctness

▸ Definition 12 (Compilation of an Execution). We define the "CompilesTo" relation $\rightsquigarrow \subseteq \mathbb{H} \times \mathbb{H}$ for the compilation from Java to x86 as the followings: Given a Java program P_{src} and a memory model J that supports Java, let P_{tgt} be the target-level program compiled from P_{src} using the compilation scheme to x86 as shown above. 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:

1. H_{tgt} is observationally equivalent to H_{src}
2. $H_{tgt}.\texttt{co} \subseteq H_{src}.\texttt{to}$
3. If $i_1, i_2 \in H_{src}.\texttt{E}$ and $i_1 \xrightarrow{\texttt{push}} i_2$ and i_1 is a write, then $i_1 \xrightarrow{\texttt{po}} i_3 \xrightarrow{\texttt{po}} i_2$ for $i_1, i_2 \in H_{tgt}.\texttt{E}$ and $i_3 \in H_{tgt}.\texttt{F}$ where i_3 is an event stem from an `mfence` instruction.

[7] Note that instead of using `mfence` instruction for full fences, HotSpot uses a read-modify-write instruction to emulate the synchronization effect of it. According to the definition of TSO, the synchronization effect of a RMW event is exactly the same as an `mfence` event. Both of them produce a `ghb` order before and after the event. Therefore, we keep the simplicity of the proof here by using the `mfence` instruction.

Note that this definition does not say anything about whether an execution graph is consistent under a memory model.

▸ Lemma 6 (JAM'_{21} to x86-TSO). Let P_{src} be a Java program, P_{tgt} be the x86 program compiled from P_{src} using the compilation scheme to x86 as shown above. For all $H_{tgt} \in Histories_{TSO}(P_{tgt})$ there exists a $H_{src} \in Histories_{JAM'}(P_{src})$ such that $H_{src} \rightsquigarrow H_{tgt}$.

Proof. It is obvious that there exists an H_{src} such that $H_{src} \rightsquigarrow H_{tgt}$. We show that $H_{src} \in Histories_{JAM'}(P)$. That is, we show that H_{src} is consistent under JAM'_{21} by showing that it fulfills the two requirements of JAM'_{21}.

1. No-Thin-Air Requirement. $(\texttt{po}|\texttt{rf})$ is acyclic. The $\texttt{rfi}$ order is included in the $\texttt{po}$. Therefore, we need to show that $(\texttt{po}|\texttt{rfe})$ is acyclic. Note that since the only inter-thread order is $\texttt{rfe}$. If there is a cycle in $(\texttt{po}|\texttt{rfe})$, then the head of each thread is a R^{Opq} and the last event in each thread participating in this cycle is a W^{Opq}, where $R^{\mathsf{Opq}} \xrightarrow{\texttt{po}} W^{\mathsf{Opq}}$ in H_{src}. Using a compiler that follows the compilation scheme, this translates to $R \xrightarrow{\texttt{po}} W$ in H_{tgt}. Further we can infer that $R \xrightarrow{\texttt{ghb}} W$ in H_{tgt}. x86-TSO ensures that $\texttt{ghb}$ is acyclic. Therefore, if there is a cycle of $(\texttt{po}|\texttt{rf})$ in H_{src}, H_{tgt} would not be consistent under x86-TSO's memory model, contradicting to our previous assumption.
2. Coherence Requirement. In order to show that H_{src} fulfills the coherence requirement, we need to show that $\texttt{co}$ in H_{tgt} is a linear extension of $\texttt{co-jom}$ in H_{src}. We prove this by analyzing each case for $\texttt{co-jom}$. That is, if $i_1 \xrightarrow{\texttt{co-jom}} i_2$ but $i_2 \xrightarrow{\texttt{co}} i_1$, then H_{tgt} is inconsistent under x86-TSO.

   - coinit. This follows naturally as $H_{src}.\texttt{IW} = H_{tgt}.\texttt{IW}$.
   - cofw. This follows naturally as $H_{src}.\texttt{FW} = H_{tgt}.\texttt{FW}$.
   - corr. This implies that there exists R_1 and R_2 such that $i_1 \xrightarrow{\texttt{rf}} R_1$, $i_2 \xrightarrow{\texttt{rf}} R_2$, and $R_1 \xrightarrow{\texttt{po}} R_2$. From $i_2 \xrightarrow{\texttt{co}} i_1$ we can infer that $R_2 \xrightarrow{\texttt{fr}} i_1$. Note that $R_1 \xrightarrow{\texttt{ghb}} R_2$ in this case since a $\texttt{po}$ order from a read event is preserved in TSO. Now we have a $\texttt{ghb}$ cycle $R_2 \xrightarrow{\texttt{fr}} i_1 \xrightarrow{\texttt{rf}} R_1 \xrightarrow{\texttt{ghb}} R_2$, contradicting to out previous assumption that H_{tgt} is consistent under x86-TSO.
   - coww. This implies that $i_1 \xrightarrow{\texttt{vo}} i_2$. We analyze each case of $\texttt{vo}$ order.
     - $i_1 \xrightarrow{\texttt{po-loc}} i_2$. $i_2 \xrightarrow{\texttt{co}} i_1$ would violates the SC-Per-Location Requirement of Power, making H_{tgt} inconsistent, contradicting to our previous assumption.
     - $i_1 \xrightarrow{\texttt{ra}} i_2$ or $i_1 \xrightarrow{\texttt{push}} i_2$ or $i_1 \xrightarrow{\texttt{svo}} i_2$. Note that all three cases are included in $i_1 \xrightarrow{\texttt{po}} i_2$ to the same location. Thus, if $i_2 \xrightarrow{\texttt{co}} i_1$, then there would be a cycle of $(\texttt{po-loc}|\texttt{com})$ in H_{tgt}, contradicting to our previous assumption that H_{tgt} is consistent under TSO.
     - $i_1 \xrightarrow{\texttt{push}} E_1 \xrightarrow{\texttt{chapo}} E_2 \xrightarrow{\texttt{push}} i_2$ for some E_1 and E_2 in H_{src}. Since i_1 is a write, we can infer that $i_1 \xrightarrow{\texttt{po}} i_{LOCK} \xrightarrow{\texttt{po}} E_1$ in H_{tgt}, where i_{LOCK} is a RMW event. According to x86-TSO, we can further infer that $i_1 \xrightarrow{\texttt{ghb}} i_{LOCK} \xrightarrow{\texttt{ghb}} E_1$ in H_{tgt}, which can be simplified to $i_1 \xrightarrow{\texttt{ghb}} E_1$. Similarly, we can infer that $E_2 \xrightarrow{\texttt{ghb}} i_2$ (if E_2 is a read then the $\texttt{po}$ order is included in $\texttt{ghb}$; otherwise, there must be a RMW event between E_2 and i_2, which yields a $\texttt{ghb}$ order too). Now, since the communication edges induced by $\texttt{chapo}$ are also included in $\texttt{ghb}$ in H_{tgt}, $i_2 \xrightarrow{\texttt{co}} i_1$ would directly produce a $\texttt{ghb}$ cycle, contradicting with our previous assumptions.
     - $i_1 \xrightarrow{\texttt{vvo}} E \xrightarrow{\texttt{vvo}} i_2$ for some E in H_{src}. Note that $\texttt{vo}$ orders in H_{src} only produce $\texttt{ghb}$ orders in H_{tgt}. Therefore, $i_2 \xrightarrow{\texttt{co}} i_1$ would always result in a $\texttt{ghb}$ cycle in H_{tgt}, contradicting to the previous assumption.

- cowr. This implies that $i_1 \xrightarrow{\texttt{vo}} R$ and $i_2 \xrightarrow{\texttt{rf}} R$ for some R in H_{src}. With $i_2 \xrightarrow{\texttt{co}} i_1$ we can infer that $R \xrightarrow{\texttt{fr}} i_1$. As in previous cases, we observe that vo only produce ghb in H_{tgt}. Therefore, having $i_2 \xrightarrow{\texttt{co}} i_1$ would result in a ghb cycle in H_{tgt}.
- corw. This implies that there is R in H_{src} such that $R \xrightarrow{\texttt{po}} i_2$ and $i_1 \xrightarrow{\texttt{vo}} R$. Note that po order from a Read access is included in ghb in H_{tgt}. As in previous cases, we observe that vo only produce ghb in H_{tgt}. Therefore, having $i_2 \xrightarrow{\texttt{co}} i_1$ would result in a ghb cycle, contradicting to our previous assumption.
- cormwexcl. This implies that $W \xrightarrow{\texttt{rf}} R$, $R \xrightarrow{\texttt{rmw}} i_1$, and $W \xrightarrow{\texttt{co}} i_2$ for some W and R in H_{src}. We also have $R \xrightarrow{\texttt{fr}} i_2$. Now having $i_2 \xrightarrow{\texttt{co}} i_1$ would violate the Atomicity Requirement of TSO, contradicting to our previous assumptions.
- cormwtotal. Since there is no other restrictions on the total order among RMW operations except that it has to be compatible with the rf and intra-thread orders, it automatics becomes a subset of co in H_{tgt}. Therefore, having $i_2 \xrightarrow{\texttt{co}} i_1$ in this case would yield a cycle in co, producing a ghb cycle in H_{tgt}.

Thus we have shown that co is a linear extension of co-jom. As a consequence, co-jom is guaranteed to be acyclic is H_{tgt} is consistent under TSO.

Thus H_{src} is JAM'_{21}-consistent. That is, $H_{src} \in Histories_{JAM'}(P)$. ◀

▸ **Theorem 13** (Compilation Correctness to x86-TSO). *Let P_{src} be a Java program, P_{tgt} be the x86 program compiled from P_{src} using the compilation scheme to x86 as shown above. For all $H_{tgt} \in Histories_{TSO}(P_{tgt})$ there exists a $H_{src} \in Histories_{JAM}(P_{src})$ such that $H_{src} \rightsquigarrow H_{tgt}$.*

Proof. By Lemma 6, we know that there exists an $H'_{src} \in Histories_{JAM'}(P_{src})$ such that $H'_{src} \rightsquigarrow H_{tgt}$. Therefore, by definition of the $\rightsquigarrow$ relation,

1. H_{tgt} is observationally equivalent to H'_{src}
2. $H_{tgt}.\texttt{co} \subseteq H'_{src}.\texttt{to}$
3. If $i_1, i_2 \in H'_{src}.\texttt{E}$ and $i_1 \xrightarrow{\texttt{push}} i_2$ and i_1 is a write, then $i_1 \xrightarrow{\texttt{po}} i_3 \xrightarrow{\texttt{po}} i_2$ for $i_1, i_2 \in H_{tgt}.\texttt{E}$ and $i_3 \in H_{tgt}.\texttt{F}$ where i_3 is an event stem from an mfence instruction.

By Theorem 12, we know that for all H'_{src}, there exists an $H_{src} \in Histories_{JAM}(P)$ such that H_{src} is observationally equivalent to H'_{src}. By Lemma 2, H_{src} is observationally equivalent to H_{tgt}. Furthermore,

1. $H_{tgt}.\texttt{co} \subseteq H_{src}.\texttt{to}$ because $H_{src}.\texttt{to} = H'_{src}.\texttt{to}$.
2. If $i_1, i_2 \in H_{src}.\texttt{E}$ and $i_1 \xrightarrow{\texttt{push}} i_2$ and i_1 is a write, then $i_1 \xrightarrow{\texttt{po}} i_3 \xrightarrow{\texttt{po}} i_2$ for $i_1, i_2 \in H_{tgt}.\texttt{E}$ and $i_3 \in H_{tgt}.\texttt{F}$ where i_3 is an event stem from an mfence instruction, because $\forall i \in H_{src}.\texttt{E}, H_{src}.AccessMode(i) = H'_{src}.AccessMode(i)$ and $H_{src}.\texttt{po} = H'_{src}.\texttt{po}$, which means $H_{src}.\texttt{push} = H'_{src}.\texttt{push}$.

Therefore, we have shown that for all $H_{tgt} \in Histories_{TSO}(P)$, there exists an $H_{src} \in Histories_{JAM}(P)$ such that $H_{src} \rightsquigarrow H_{tgt}$. That is, the compilation scheme to x86 is correct. ◀

G Program Transformations

G.1 Deordering and Reordering

▸ Theorem 4 (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.

Proof. Note that for the Coherence requirement, only three kinds of edges contributes to co-jam: vo, rf, and po to the same location. Since here we are considering deorderable pairs, which are pairs of accesses to different locations related by po in H_{src}, we only need to consider whether deordering them would affect the set of vo in the execution. We can analyze this case by case.

- $(R_x * R_y)$. The No-Thin-Air requirement is fulfilled automatically in H_{src} since we are deordering a pair of reads. Since $\mathtt{po} \cap (R_x^{\sqsubseteq \mathsf{Opq}} * R_y) \not\subseteq \mathtt{vo}$, it follows that H_{src} is also JAM_{21}-consistent.
- $(R_x * W_y)$. H_{src} fulfills Coherence since the po edge between two accesses whose access mode is weaker or equal to Opaque mode does not contribute to any new vo edge. In addition, H_{src} fulfills the No-Thin-Air requirement because one of the accesses is in Plain mode where as No-Thin-Air only requires the acyclicy of $\mathtt{po} \cup \mathtt{rf}$ among Opaque mode accesses.
- $(R_x * RMW_y)$. First note that the Volatile mode for RMWs include the effect of Release Mode. H_{src} fulfills Coherence since the po edge between R_x and RMW_y does not contribute to any new vo edge. H_{src} fulfills No-Thin-Air because R_x is Plain mode.
- $(R_x * F)$. It's easy to see that the po edge added in H_{src} does not contribute to any new vo edge therefore the Coherence is fulfilled. Since we are deordering a read and a fence, H_{src} fulfills No-Thin-Air automatically.
- $(W_x * R_y)$. It's easy to see that the po edge added in H_{src} does not contribute to any new vo edge therefore the Coherence is fulfilled. Since we are deordering a write and a read, H_{src} fulfills No-Thin-Air automatically. The only situation the two accesses cannot be deordered is when they are both in Volatile mode because the po between two Volatile accesses can be derived into a vo order.
- $(W_x * W_y)$. It's easy to see that the po edge added in H_{src} does not contribute to any new vo edge therefore the Coherence is fulfilled. Since we are deordering a write and a write, H_{src} fulfills No-Thin-Air automatically. Here we need the second write W_y to be weaker than release mode to ensure that the po between the two accesses does not contribute to the vo order.
- $(W_x * RMW_y)$. It's easy to see that the po edge added in H_{src} does not contribute to any new vo edge therefore the Coherence is fulfilled. Since we are deordering a write and a read-modify-write, H_{src} fulfills No-Thin-Air automatically. Since RMW_y is both in the

set of reads and in the set of writes of the execution graph H_{src}, we take the intersection of previous cases. In addition, `rel` and `acq` do not subsume each other, so it is safe for o_2 to be `acq` mode.

- $(W_x * F)$. It's easy to see that the `po` edge added in H_{src} does not contribute to any new `vo` edge therefore the COHERENCE is fulfilled. Since we are deordering a write and a fence, H_{src} fulfills NO-THIN-AIR automatically. Here we are basically avoiding the situation when W_x and F can form any `svo`. In addition, there are also situations where W_x and F have a `svo` if the writes that follows the fence are already in `rel` modes. Since `svo` and `ra` are considered equivalently in terms of their memory order effect in the JAM_{21} model, the `svo` they form is redundant in the presence of all the `ra`.
- $(RMW_x * R_y)$. It's easy to see that the `po` edge added in H_{src} does not contribute to any new `vo` edge therefore the COHERENCE is fulfilled. Since we are deordering a read-modify-write and a read, H_{src} fulfills NO-THIN-AIR automatically. Here we want to avoid the RMW_x to have an access mode stronger or equal to `acq` mode because it'd create an `ra` edge which is considered as a `vo` edge.
- $(RMW_x * W_y)$. It's easy to see that the `po` edge added in H_{src} does not contribute to any new `vo` edge therefore the COHERENCE is fulfilled. For the NO-THIN-AIR requirement, since W_y is in Plain mode, it does not contribute to any `po` ∪ `rf` cycle among Opaque accesses.
- $(RMW_x * RMW_y)$. They cannot be deordered due to the NO-THIN-AIR requirement. (Note that RMW operations are atomic by definition, so there is no Plain mode or Opaque mode for RMW).
- $(RMW_x * F)$. Similar to the previous cases.
- Deordering with fence. Similar to the previous cases.

◀

▸ COROLLARY 2 (REORDERING). JAM_{21} supports the reordering transformation for pairs of adjacent accesses shown in Fig. 6.

Proof. Let a and b be a pair of such memory events and $a \xrightarrow{\texttt{po}} b$ in H_{src}. By Theorem 4, we know that removing the `po` edge between a and b does not introduce new program behavior. Let H' be the execution graph after the deordering transformation. By Theorem 3, we know that adding a `po` edge from b to a in H' does not introduce new program behavior either. Therefore, reordering of access pairs in Fig. 6 is supported by JAM_{21}. ◀

G.2 Merging

G.2.1 Read-read Merging

▸ THEOREM 5 (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:

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

- $H_{src}.AccessMode(b) = H_{src}.AccessMode(a) \sqsubseteq \mathsf{Acquire}$

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

Proof. We show that H_{src} fulfills the two requirements needed to be JAM_{21}-consistent.

- Suppose H_{src} violates the No-Thin-Air requirement, then there is a $(\mathsf{po} \cup \texttt{rf})+$ cycle involving b. If we have $a' \xrightarrow{\texttt{rf}} b \xrightarrow{(\mathsf{po}|\texttt{rf})+} a'$, then $a' \xrightarrow{\texttt{rf}} a \xrightarrow{(\mathsf{po}|\texttt{rf})+} a'$. If we have $a \xrightarrow{\mathsf{po}} b \xrightarrow{(\mathsf{po}|\texttt{rf})+} a$, then $a \xrightarrow{(\mathsf{po}|\texttt{rf})+} a$. In both of the cases, H_{tgt} is inconsistent, which contradict with our previous assumption. Therefore, the No-Thin-Air requirement is fulfilled by H_{src}.
- Suppose H_{src} violates the Coherence requirement, then there is a $\texttt{co}$ cycle. Note that $AccessMode(b) = AccessMode(a) \sqsubseteq \mathsf{Acq}$. In addition, for all events i, if $b \xrightarrow{\texttt{vo}} i$, then $a \xrightarrow{\texttt{vo}} i$ and for all events j, if $j \xrightarrow{\texttt{vo}} b$, then $j \xrightarrow{\texttt{vo}} a$. Therefore, for any coherence cycle derived from the edges from and to b, there is also a coherence cycle derived from the edges from and to a. If H_{src} has a $\texttt{co}$ cycle, H_{tgt} also has a $\texttt{co}$ cycle, which contradicts with our previous assumption.

◀

G.2.1.1 Counter Example

Here, we give an example showing that read-read merging is not allowed by JAM_{21} if the read accesses are both Volatile mode. 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);
}
```

```
Thread3 {
  int r5 = X.getVolatile(); // 2
  int r6 = X.getVolatile(); // 2
  Y.setRelease(1);
}

Thread4 {
  Y.setVolatile(2);
  X.setVolatile(1);
}
```

Applying the read-read merging transformation to this program yields:

```
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);
}
```

```
Thread3 {
  int r5 = X.getVolatile(); // 2
  int r6 = r5
  Y.setRelease(1);
}

Thread4 {
  Y.setVolatile(2);
  X.setVolatile(1);
}
```

The execution graphs with the annotated read values is shown in Fig. 11 and Fig. 12.

For the two read accesses of x on Thread 3, one may think it's OK to merge them into one. However, since they are Volatile accesses, they also impose a $\texttt{push}$ edge which is totally

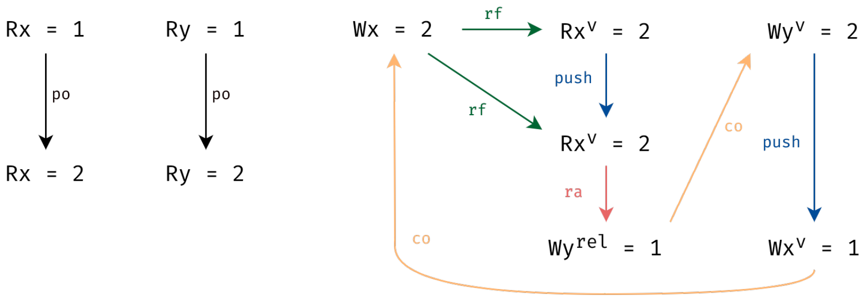


Figure 11 Execution Graph before read-read merge on Volatile (Forbidden)

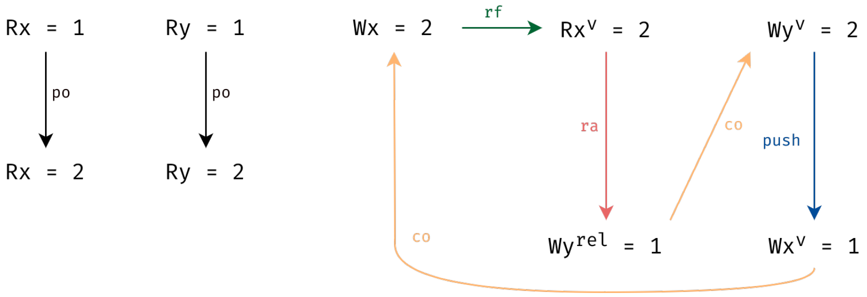


Figure 12 Execution Graph after read-read merge on Volatile (Allowed)

ordered with other push edges. Merging the two reads removes the synchronization provided by the push edge, introducing the program behavior shown in Fig. 12.

G.2.2 Write-write Merging

▸ Theorem 6 (Write-Write Merging). Let H_{tgt} be an JAM_{21}-consistent execution. Let $b \in H_{tgt}.W \backslash RMW$ and let $a \notin H_{tgt}.\mathtt{E}$ 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:

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

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

Proof. We show that H_{src} fulfills the two requirements to be JAM_{21}-consistent.

- No-Thin-Air. Note that $a \xrightarrow{\mathtt{po}} b \xrightarrow{(\mathtt{po}|\mathtt{rf})+} a$ implies that $b \xrightarrow{(\mathtt{po}|\mathtt{rf})+} b$. Therefore, if H_{src} violates No-Thin-Air, H_{tgt} also violates No-Thin-Air, contradicting to our previous assumption.

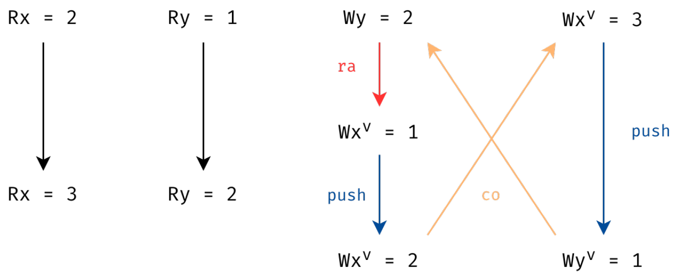


Figure 13 Execution graph before write-write merge on Volatile (Forbidden)

- Coherence. First note that there is no extra rf edge from a and $\forall i, (i \xrightarrow{\text{vo}} a \Rightarrow i \xrightarrow{\text{vo}} b) \land (a \xrightarrow{\text{vo}} i \Rightarrow b \xrightarrow{\text{vo}} i)$ (because a and b have the same access mode and they are not in Volatile mode). Therefore, any co cycle derived from a, we can derive the same co cycle with b. While $a \xrightarrow{\text{po}} b$ implies that $a \xrightarrow{\text{vo}} b$, since there is no rf edge from a, it cannot contribute to any extra co cycle. Therefore, if there is a co cycle in H_{src}, then it implies that there is a co cycle in H_{tgt}, contradicting to our previous assumption.

◀

G.2.2.1 Counter Example

We now provide a counter-example showing write-write merge is not valid for Volatile mode writes. Consider the following example program:

```
Thread0 {
  int r1 = X.getOpaque(); // 2
  int r2 = X.getOpaque(); // 3
}

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

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

Thread3 {
  X.setVolatile(3);
  Y.setVolatile(1);
}
```

The execution graph of the program before the transformation is shown in Fig. 13. Applying write-write merging transformation to Thread 2, we have:

```
Thread0 {
  int r1 = X.getOpaque(); // 2
  int r2 = X.getOpaque(); // 3
}

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

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

Thread3 {
  X.setVolatile(3);
  Y.setVolatile(1);
}
```

The execution graph after the transformation is shown in Fig. 14. After removing the write access in Volatile mode, the cross-thread synchronization effect between Thread 2 and Thread 3 is also removed, introducing the new behavior in the figure.

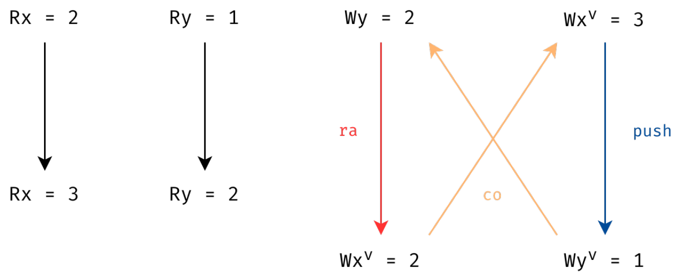


Figure 14 Execution graph after write-write merge on Volatile (Allowed)

G.2.3 Write/RMW-read Merging

▶ Theorem 7 (Write/RMW-Read Merging). Let H_{tgt} be a JAM_{21}-consistent execution. Let $a \in H_{tgt}.\mathtt{W}$ and $b \notin H_{tgt}.\mathtt{E}$. There exists a H_{src} such that:

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

Proof. We show that H_{src} fulfills the two requirements to be JAM_{21}-consistent.

- No-Thin-Air. First note that, by the well-formedness of $\mathtt{rf}$ order, a is the only access in the execution graph that has a $\mathtt{rf}$ edge to b. Therefore, $a \xrightarrow{\mathtt{rf}} b \xrightarrow{(\mathtt{po}|\mathtt{rf})+} a$ implies that $a \xrightarrow{(\mathtt{po}|\mathtt{rf})+} a$, which means there is also a $(\mathtt{po}|\mathtt{rf})+$ cycle in H_{tgt}, contradicting to our previous assumption.
- Coherence. Since $AccessMode(b) = \mathsf{Opaque}$, there is no out-going cross-thread edge from b and for all event i such that $b \xrightarrow{\mathtt{vo}} i$, we have $a \xrightarrow{\mathtt{vo}} i$ (similarly, for all event j such that $j \xrightarrow{\mathtt{vo}} b$, we have $j \xrightarrow{\mathtt{vo}} a$). Since $a \xrightarrow{\mathtt{rf}} b$ is intra-thread, for any $\mathtt{co}$ edge derived from $a \xrightarrow{\mathtt{rf}} b$ using the $\mathtt{corr}$ rule, it implies that there is a read access R and write access W such that $a \xrightarrow{\mathtt{rf}} b \xrightarrow{\mathtt{po}} R$ and $W \xrightarrow{\mathtt{rf}} R$ we can derive the same $\mathtt{co}$ edge using the $\mathtt{cowr}$ rule with $a \xrightarrow{\mathtt{vo}} R$ and $W \xrightarrow{\mathtt{rf}} R$. Similarly, for any $\mathtt{co}$ edge derived from $a \xrightarrow{\mathtt{rf}} b$ using the $\mathtt{cowr}$ rule, it implies that there is a write access W such that $W \xrightarrow{\mathtt{vo}} b$. Then $W \xrightarrow{\mathtt{vo}} a$ as well. Using the $\mathtt{coww}$ rule we can derive the same $\mathtt{co}$ edge. Thus, if there is any $\mathtt{co}$ cycle in H_{src}, the same $\mathtt{co}$ cycle also appear in H_{tgt}, contradicting to our previous assumption.

◀

G.2.3.1 Counter Example

Here we show that write/RMW-read merging is not valid if the read is or is stronger than Acquire mode. Consider the following example:

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

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

Thread2 {
  Y.setOpaque(1);
}
```

```
Thread3 {
  X.setRelease(2);
  int r7 = X.getAcquire(); // 2
  int r5 = Z.getVolatile(); // 0
  int r6 = Y.getVolatile(); // 1
}

Thread4 {
  Y.setVolatile(2);
  X.setVolatile(1);
}
```

The execution graph can be found in Fig. 15. The execution is forbidden. Indeed, there are two possible cases:

1. $R_z^V = 0 \xrightarrow{\texttt{vvo}} W_x^V = 1$. Since $\texttt{rf} \subseteq \texttt{vvo}$ and $\texttt{ra} \subseteq \texttt{vvo}$, we can infer that $W_x^{\texttt{rel}} = 2 \xrightarrow{\texttt{vvo}} R_z^V = 0 \xrightarrow{\texttt{vvo}} W_x^V = 1$. Using the `coww` rule, we can infer that $W_x^{\texttt{rel}} = 2 \xrightarrow{\texttt{co}} W_x^V = 1$, which contradicts with the `co` edge we inferred using the `corr` rule and Thread 0.
2. $W_y^V = 2 \xrightarrow{\texttt{vvo}} R_y^V = 1$. This immediately contradicts with the `co` edge we derived using the `corr` rule with Thread 1.

Applying the transformation, we have:

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

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

Thread2 {
  Y.setOpaque(1);
}
```

```
Thread3 {
  X.setRelease(2);
  int r7 = 2;
  int r5 = Z.getVolatile(); // 0
  int r6 = Y.getVolatile(); // 1
}

Thread4 {
  Y.setVolatile(2);
  X.setVolatile(1);
}
```

The execution graph is shown in Fig. 16. Due to the removal of the `rf` and `ra` edge, the previously forbidden behavior is introduced after the transformation.

G.2.4 Write-RMW Merging

▶ Theorem 8 (Write-RMW Merging). Let H_{tgt} be a JAM_{21}-consistent execution. Let $b \in H_{tgt}.\texttt{W} \backslash H_{tgt}.\texttt{RMW}$, $a \notin H_{tgt}.\texttt{E}$ and $v \in \texttt{Val}$. There exists a H_{src} such that:

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

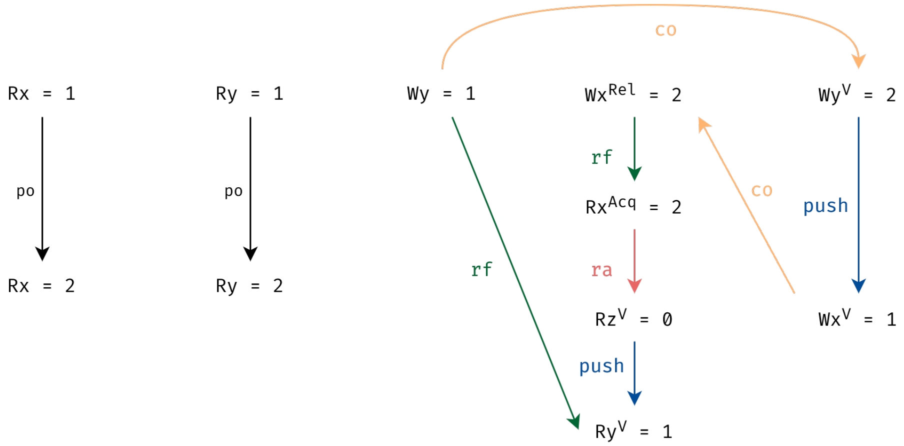


Figure 15 Execution Graph before Write-read Merge Transformation (Forbidden)

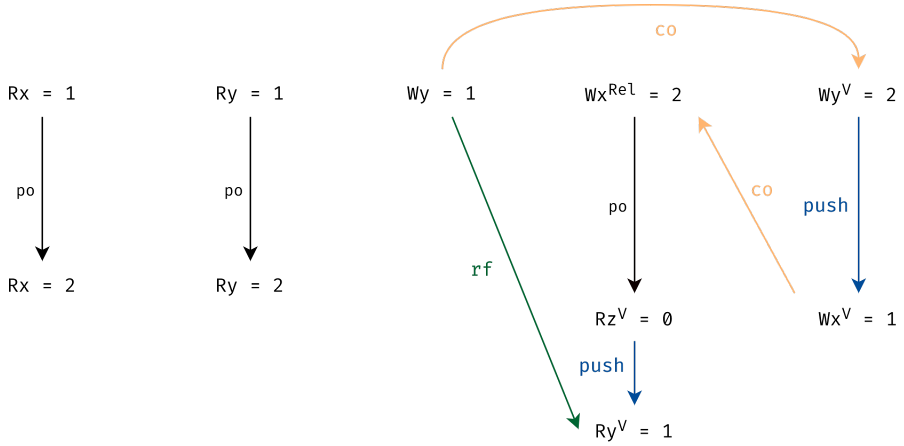


Figure 16 Execution Graph after Write-read Merge Transformation (Allowed)

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

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

Proof. Most parts of the proof is similar to the proof for write-write merging except for the case where there is a co cycle in H_{src} due to the total coherence order among RMW operations. Suppose H_{src} violates COHERENCE by having a co cycle built from the `cormwtotal` rule. That is, we have a RMW operation i such that:

- If $b \xrightarrow{\texttt{cormwtotal}} i$, then $i \xrightarrow{\texttt{co}} b$
- If $i \xrightarrow{\texttt{cormwtotal}} b$, then $b \xrightarrow{\texttt{co}} i$

Note that we cannot use existing co orders to derive other orders than $\xrightarrow{\texttt{cormwexcl}}$ orders (which is also a co order). If the co between i and b are not $\xrightarrow{\texttt{cormwexcl}}$ edges, then they co-exists in one execution. Now we have $i \xrightarrow{\texttt{co}} b \xrightarrow{\texttt{co}} i$. If the co between i and b are $\xrightarrow{\texttt{cormwexcl}}$ edges, then there exist two RMW operations j and k, such that, $b \xrightarrow{\texttt{rf}} j$, $i \xrightarrow{\texttt{rf}} k$, $i \xrightarrow{\texttt{co}} j$ and $b \xrightarrow{\texttt{co}} k$. Note that there is still a total order among i, j, k in H_{tgt}. Now we have either $j \xrightarrow{\texttt{cormwtotal}} k$ or $k \xrightarrow{\texttt{cormwtotal}} j$. Each case yields a contradiction by the `coermwexcl` rule. Therefore, if there is a co cycle in H_{src}, H_{tgt} is also forbidden, which contradicts to our previous assumption. ◀

G.2.5 RMW-RMW Merging

▸ THEOREM 9 (RMW-RMW MERGING). Let H_{tgt} be a JAM_{21}-consistent execution. Let x be a memory location and $a \in H_{tgt}.\texttt{E}$ with $H_{tgt}.val(a) = (v_r, v_w)$, $H_{tgt}.loc(a) = x$, and $H_{tgt}.AccessMode(a) \in \{\textsf{Release}, \textsf{Acquire}\}$. Let $b \notin H_{tgt}.\texttt{E}$, there exists a H_{src} such that:

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

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

Proof. We show that H_{src} fulfills the two requirements of JAM_{21}-consistency.

- NO-THIN-AIR. Suppose H_{src} violates this requirement and has a $(\texttt{po}|\texttt{rf})+$ cycle. Since $H_{src}.val(a) = (v_r, v)$ and $H_{src}.val(b) = (v, v_w)$, if $a \xrightarrow{\texttt{po}} b \xrightarrow{(\texttt{po}|\texttt{rf})+} a$ in H_{src}, it implies that $a \xrightarrow{(\texttt{po}|\texttt{rf})+} a$ in H_{tgt}, contradicting to our previous assumption.
- COHERENCE. First note that there is only one rf edge from a in H_{src} and that is $a \xrightarrow{\texttt{rf}} b$. In addition, for all event i such that $i \xrightarrow{\texttt{vo}} b$ in H_{src}, $i \xrightarrow{\texttt{vo}} a$ in H_{tgt}. For all j such that $b \xrightarrow{\texttt{vo}} j$ in H_{src}, $a \xrightarrow{\texttt{vo}} j$ in H_{tgt}. Therefore, if there is a co cycle in H_{src}, there is also a co cycle in H_{tgt}, contradicting to our previous assumption.

◀

G.3 Register Promotion for non-shared Variable

▸ Theorem 10 (Weakening for non-shared variable). Let H_{tgt} be a JAM_{21}-consistent execution such that, for all accesses i and j in $H_{tgt}.\mathtt{E}$, $loc(i) = loc(j) = x \Rightarrow Tid(i) = Tid(j)$ for some memory location x. In addition, $\forall i \in H_{tgt}.\mathtt{E}, loc(i) = x \Rightarrow AccessMode(i) = \mathsf{Plain}$. There exists an execution H_{src} such that:

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

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

Proof. We show that H_{src} fulfills the two requirements of JAM_{21}-consistency.

1. No-Thin-Air. Note that there is no cross-thread $\mathtt{rf}$ edge from or to accesses of location x. Therefore, since $H_{src}.\mathtt{po} = H_{tgt}.\mathtt{po}$ and $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf}$, if there is a $(\mathtt{po}|\mathtt{rf})+$ cycle in H_{src}, there is a $(\mathtt{po}|\mathtt{rf})+$ cycle in H_{tgt}, contradicting to our previous assumption.
2. Coherence. Note that the transformation is equivalent to removing all the $\mathtt{ra}$ edges that involve accesses to x. Therefore, $H_{src}.\mathtt{vo} = H_{tgt}.\mathtt{vo} \backslash \{\langle a, b\rangle \mid (loc(a) = x \wedge a \xrightarrow{\mathtt{ra}} b \wedge AccessMode(b) \neq \mathtt{Release}) \vee (loc(b) = x \wedge a \xrightarrow{\mathtt{ra}} b \wedge AccessMode(a) \neq \mathtt{Acquire})\}$. For accesses i and j such that $loc(i) \neq x$ and $loc(j) \neq x$, if $i \xrightarrow{\mathtt{vo}} j$ in H_{src}, $i \xrightarrow{\mathtt{vo}} j$ in H_{tgt}. In addition, since x is not shared across different threads, all accesses to location x are related by $\mathtt{po}$. Since all accesses to x have an access mode of either $\mathsf{Release}$ or $\mathsf{Acquire}$, there is no cross-thread $\mathtt{vo}$ edges or $\mathtt{rf}$ edges from or to these accesses. Therefore, for all memory location $y \neq x$, $H_{src}.\mathtt{vo} \upharpoonright_y = H_{tgt}.\mathtt{vo} \upharpoonright_y$. Suppose H_{src} violates this requirement by having a $\mathtt{co}$ cycle:
   - If there is a $\mathtt{co}$ cycle with accesses to location x. Since $H_{src}.\mathtt{po\text{-}loc} = H_{tgt}.\mathtt{po\text{-}loc}$ and $\mathtt{po\text{-}loc} \subseteq \mathtt{vo}$, then there is also a $\mathtt{co}$ cycle with accesses to location x in H_{tgt}, contradicting to our previous assumption.
   - If there is a $\mathtt{co}$ cycle with accesses to other locations. Since for all memory location $y \neq x$, $H_{src}.\mathtt{vo} \upharpoonright_y = H_{tgt}.\mathtt{vo} \upharpoonright_y$ and $H_{src}.\mathtt{rf} = H_{tgt}.\mathtt{rf}$, it implies there is also a $\mathtt{co}$ cycle in H_{tgt}, contradicting to our previous assumption.

◂

▸ Theorem 11 (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 $(i \in H_{src}.W) \wedge (loc(i) = x) \wedge (i \xrightarrow{\mathtt{po}} a) \wedge (\forall j \in H_{src}.\mathtt{E}, (loc(j) = x) \wedge (j \xrightarrow{\mathtt{po}} a) \Rightarrow (j \xrightarrow{\mathtt{po}} i))$.
- $H_{src}.\mathtt{to} = H_{tgt}.\mathtt{to}$

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

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

Proof. It is clear that H_{src} does not violate No-Thin-Air and there is no co cycle for accesses to location x. For Coherence, note that for all memory location $y \neq x$, $H_{src}.\texttt{vo} \restriction_y = H_{tgt}.\texttt{vo} \restriction_y$ and $H_{src}.\texttt{rf} = H_{tgt}.\texttt{rf}$, it implies that if there is a co cycle in H_{src} there is also a co cycle in H_{tgt}, contradicting to our previous assumption. ◀

H Key Properties of the JAM_{21} Model

In this section, we show some key properties of JAM_{21}. First we show that the prior theorems of JAM_{19} still hold for JAM_{21} in Section H.1. Then in Section H.2, we prove that when all accesses in program order are push ordered then the semantics of executions is sequentially consistent. As a corollary when all accesses are Volatile, which implies a push order, then the executions are sequentially consistent. We have defined and proved these theorems in Coq. The Coq source code is included in our supplementary materials.

H.1 Prior Theorems

The JAM_{21} model preserves the two main theoretical results of [3], namely the monotonicity of access modes and the causal-acquire reads. We recount each theorem here briefly beginning with the monotonicity of access modes. In addition, we prove the DRF-SC theorem for JAM_{21}.

We use the reflexive ordering of the access modes as Plain $\sqsubseteq$ Opaque $\sqsubseteq$ ReleaseAcquire $\sqsubseteq$ Volatile and extend it to accesses $l_{m_1} \sqsubseteq l_{m_2}$, $l_{m_1} := n_1 \sqsubseteq l_{m_2} := n_2$, $\mathsf{RMW}(l, n_1) \sqsubseteq \mathsf{RMW}(l, n_2)$ whenever $m_1 \sqsubseteq m_2$. We treat read-modify-write (RMW) events as always having the same order. We extend the order to histories by matching identifiers and ordering the accesses.

$$H_1 \sqsubseteq H_2 \triangleq \forall\, i\, a_1\, a_2, H_1(\mathsf{is}(i, a_1)) \wedge H_2(\mathsf{is}(i, a_2)) \Rightarrow a_1 \sqsubseteq a_2$$

We adopt the same notion of "well-formedness" from [3] for a given history H, i.e., *trace coherence*[8].

▸ Definition 13 (Trace Coherence). An execution history H is trace coherent if:

- Each memory location is initialized by an initial write. For each event $i \in H.\mathtt{E} \setminus \{H.\mathtt{IW} \cup H.\mathtt{F}\}$, there exists an initial write event $iw \in H.\mathtt{IW}$ such that $H.loc(i) = H.loc(iw)$ and $iw \xrightarrow{\mathtt{to}} i$.
- Reads-from edges are well-formed. For all $r \in H.\mathtt{R}$, there exists a unique write w such that $H.loc(w) = H.loc(r)$, $H.val(w) = H.val(r)$, and $w \xrightarrow{\mathtt{rf}} r$.
- There exists a total trace order to for all $e \in H.\mathtt{E}$ such that to is compatible with po, rf, ra, svo, and push.

When the po, rf, and to relations of two histories H_1 and H_2 have the following relationships: $H_2.\mathtt{po} \subseteq H_1.\mathtt{po}$, $H_2.\mathtt{to} \subseteq H_1.\mathtt{to}$, $H_2.\mathtt{rf} \subseteq H_1.\mathtt{rf}$, then we say they *match*.

▸ Theorem 14 (Monotonicity). [coq/Monotonicity.v, monotonicity]
For two histories H_1 and H_2, suppose that both match, both are trace coherent, and $H_2 \sqsubseteq H_1$. Further suppose that $\mathsf{acyclic}(\xrightarrow{\mathtt{co}}_{H_1})$ and that there are no specified visibility orders or push orders in H_2, then $\mathsf{acyclic}(\xrightarrow{\mathtt{co}}_{H_2})$

A version of DRF-SC theorem was proved in [3]. However, the theorem was different from the standard DRF-SC theorem.

- It did not use the conventional definition of data race with the "happens-before" order. Instead. [3] defined a sync order that captures the synchronizations between events and defined the notion of "data-race-free" using sync.

[8] We have omitted some of the details of trace coherence that are related to the internals of the modeling language as they are irrelevant here

- It used a stronger assumption than the standard DRF-SC theorem. In particular, given a program P, the standard DRF-SC theorem assumes only the SC-consistent executions of P are data race free. On the other hand, the DRF-SC theorem proved in [3] assumes all executions of P are data race free. A similar theorem was also proved in [19], called a "model-agnostic" definition of DRF-SC.

Here, we first prove the standard DRF-SC theorem (DRF-SC) with a weaker assumption than [3], and then prove the "model-agnostic" DRF-SC theorem (Execution-DRF), both using "happens-before" (hb).

We require the following standard definitions including the traditional notion of sequential consistency (SC-consistency) [16]:

$$i_1 \xrightarrow{\mathtt{fr}} i_2 \triangleq \exists i_3, i_3 \xrightarrow{\mathsf{rf}} i_1 \wedge i_3 \xrightarrow{\mathtt{co}} i_2$$
$$i_1 \xrightarrow{\mathsf{com}} i_2 \triangleq i_1 \xrightarrow{\mathtt{co}} i_2 \vee i_1 \xrightarrow{\mathsf{rf}} i_2 \vee i_1 \xrightarrow{\mathsf{fr}} i_2$$
$$i_1 \xrightarrow{\mathsf{sc}} i_2 \triangleq i_1 \xrightarrow{\mathsf{po}} i_2 \vee i_1 \xrightarrow{\mathsf{com}} i_2$$

An execution H is SC-consistent if $\mathsf{acyclic}(\xrightarrow{\mathsf{sc}}_H)$.

We also require the notion of *happens-before* (hb) defined using the *synchronizes-with* (sw) order:

$$i_1 \xrightarrow{\mathtt{sw}} i_2 \triangleq (i_1 \xrightarrow{\mathtt{rf}} i_2 \wedge AccessMode(i_1) = \mathtt{Release} \wedge AccessMode(i_2) = \mathtt{Acquire})$$
$$\vee (\exists i_3 i_4 \in \mathtt{F}, AccessMode(i_3) = \mathtt{Release} \wedge AccessMode(i_4) = \mathtt{Acquire}$$
$$\wedge\, i_3 \xrightarrow{\mathtt{po}} i_1 \xrightarrow{\mathtt{rf}} i_2 \xrightarrow{\mathtt{po}} i_4)$$
$$i_1 \xrightarrow{\mathtt{hb}} i_2 \triangleq i_1 \xrightarrow{\mathtt{po}} i_2 \vee i_1 \xrightarrow{\mathtt{sw}} i_2 \vee \exists i_3, i_1 \xrightarrow{\mathtt{hb}} i_3 \xrightarrow{\mathtt{hb}} i_2$$

▸ Definition 14. Two memory accesses i_1 and i_2 are **conflicting** in an execution H if:

- $i_1, i_2 \in H.\mathtt{E}$
- $H.loc(i_1) = H.loc(i_2)$
- At least one of i_1 and i_2 is a write

▸ Definition 15. Two memory accesses i_1 and i_2 form a **data race** if:

- i_1 and i_2 are conflicting
- $\neg(i_1 \xrightarrow{\mathtt{hb}} i_2 \vee i_2 \xrightarrow{\mathtt{hb}} i_1)$

We say they form a **volatile-race** if both i_1 and i_2 are Volatile mode accesses.

Finally, our DRF-SC theorem is stated as the following:

▸ Theorem 15 (DRF-SC). Given a program P, if all its SC-consistent executions are data-race-free or only have volatile-races, then the set of all JAM-consistent executions of P coincide with the set of SC-consistent executions.

Please see Appendix I for the proof.

We also provide the "model-agnostic" [19] version of the DRF-SC theorem:

▸ Theorem 16 (Execution-drf). Any JAM-consistent execution that is data race free or only has volatile-races is SC-consistent.

Please see Appendix J for the proof.

Finally, we demonstrate the revised semantics preserves causality with acquire reads.

▸ Theorem 17 (Causal Acquire-Reads). `[coq/ReleaseAcquire.v, acq_causality]` If H is trace coherent and all reads in H are acquire-reads, then $\mathsf{acyclic}(\xrightarrow{\mathsf{po} \mid \mathtt{rf}})$.

H.2 Volatile implies SC

Here we demonstrate that when all accesses are volatile, the program will have SC semantics.

To begin, we note that both full fences and Volatile pairs result in push orders in the formalism of [3]. That is, either a full fences or a volint edge implies a push edge. Our approach is to prove that when all program order accesses are push ordered then the semantics is SC. Thus, SC semantics follows as a corollary when all accesses are volint.

Recall from [3] that visibility order is acyclic. Intuitively, ordering induced by synchronization should not admit cycles.

▸ Lemma 7 (Acyclic Visibility). [coq/Truncate.v, vop_irreflex]
If H is trace coherent then, $\mathsf{acyclic}(\xrightarrow{\mathsf{vvo}})$.

Next we show that the communication relation is not contradicted by visibility. Since the com relationship is composed from reads and coherence relationships, both of which encode the ordering of effects, we expect that visibility should not contradict such an ordering.

▸ Lemma 8 (Communication Write Not-Visible). [coq/SC/Volatile.v, coms_vo_contra]
If H is trace coherent, all accesses are executed, $i_1 \xrightarrow{\mathsf{com}}{}^* i_2$ and i_2 is a write then $\neg(i_2 \xrightarrow{\mathsf{vvo}}{}^+ i_1)$

Next we will establish that, when two pairs of push ordered accesses are connected by a possibly empty sequence of com edges, the first access of the first pair has been executed before the first access of the second pair. Intuitively, whenever there is a full fence between these two pairs of accesses then the order in which those fences executed must be consistent with the direction of the com relation.

▸ Lemma 9 (Push Trace-ordered). [coq/SC/Volatile.v, svo_comp_svo_to]
If H is trace coherent, $\mathsf{acyclic}(\xrightarrow{\mathsf{co}})$, all accesses are executed, all accesses are push ordered, $i_1 \xrightarrow{\mathsf{push}} i_2 \xrightarrow{\mathsf{com}}{}^* i_3 \xrightarrow{\mathsf{push}} i_4$, then $i_1 \xrightarrow{\mathsf{to}} i_3$.

Proof. First, note that it is decidable whether i_3 is a write. We will begin by considering the case where it is a write. Since $\xrightarrow{\mathsf{to}}$ is total we consider each case for i_1 and i_3. First, if $i_1 \xrightarrow{\mathsf{to}} i_3$ we are done. Second, for $i_1 = i_3$ we will demonstrate a contradiction. By assumption we have $i_1 \xrightarrow{\mathsf{push}} i_2$, then by substitution we have $i_3 \xrightarrow{\mathsf{push}} i_2$. By the definition of push we have $i_3 \xrightarrow{\mathsf{vo}} i_2$. By assumption we have $i_2 \xrightarrow{\mathsf{com}}{}^* i_3$. By Lemma 8 we have $\neg(i_3 \xrightarrow{\mathsf{vo}} i_2)$ and a contradiction. Finally, for $i_3 \xrightarrow{\mathsf{to}} i_1$ we will also demonstrate a contradiction. By assumption we have both $i_1 \xrightarrow{\mathsf{push}} i_2$ and $i_3 \xrightarrow{\mathsf{push}} i_4$. Then by the definition of $\xrightarrow{\mathsf{pushto}}$ and $i_3 \xrightarrow{\mathsf{to}} i_1$ we have $i_3 \xrightarrow{\mathsf{vo}} i_2$. As before we have $i_2 \xrightarrow{\mathsf{com}}{}^* i_3$. By Lemma 8 we have $\neg(i_3 \xrightarrow{\mathsf{vo}} i_2)$ and a contradiction.

Now consider the case where i_3 is not a write, then it must be a read and there exists some write i_w such that $i_w \xrightarrow{\mathsf{rf}} i_3$. It can be shown that $i_2 \xrightarrow{\mathsf{com}}{}^* i_w$. Thus we have $i_1 \xrightarrow{\mathsf{push}} i_2 \xrightarrow{\mathsf{com}}{}^* i_w \xrightarrow{\mathsf{rf}} i_3 \xrightarrow{\mathsf{push}} i_4$. Note that, because Lemma 8 applies to $i_2 \xrightarrow{\mathsf{com}}{}^* i_w$ we have that $\neg(i_2 \xrightarrow{\mathsf{vvo}}{}^+ i_w)$ and we must derive a contradiction by showing $i_w \xrightarrow{\mathsf{vvo}}{}^+ i_2$. For $i_1 = i_3$ we have $i_3 \xrightarrow{\mathsf{vvo}} i_2$ as before. Since $i_w \xrightarrow{\mathsf{rf}} i_3$ by the definition of $\xrightarrow{\mathsf{vvo}}$ we have $i_w \xrightarrow{\mathsf{vvo}} i_3$ and $i_w \xrightarrow{\mathsf{vvo}} i_2 \xrightarrow{\mathsf{vvo}} i_3$ as required. For $i_3 \xrightarrow{\mathsf{to}} i_1$ again we have that $i_3 \xrightarrow{\mathsf{vvo}} i_2$ and in turn we have $i_w \xrightarrow{\mathsf{vvo}} i_2 \xrightarrow{\mathsf{vvo}} i_3$ as required. ◀

Now we can demonstrate that when all accesses are push ordered the program semantics is SC. The key idea is that any cycle in the sc relation (i.e. a non-SC execution) will have at least one program order edge and at least one com edge. Thus we can show that the program order edge will appear twice in the cycle and then use Push Trace-ordered inductively to show that such a push order would have to execute before itself, thereby deriving a contradiction.

▸ Theorem 18 (All Push SC). [coq/SC/Volatile.v, push_sc]
If H is trace coherent, acyclic($\xrightarrow{\mathsf{co}}$), all accesses are executed, and all accesses are `push` ordered then acyclic($\xrightarrow{\mathsf{sc}}$).

Proof. We assume $i_1 \xrightarrow{\mathsf{sc}} i_1$ and derive a contradiction. Observe that $i_1 \xrightarrow{\mathsf{sc}} i_1$ must include at least one $\xrightarrow{\mathsf{com}}$ edge because $\xrightarrow{\mathsf{po}}$ is acyclic. Further observe that it must also include at least one program order edge because $\xrightarrow{\mathsf{com}}$ is also acyclic. Thus there exists some access i_2 such that we can rearrange to obtain a sequence of program order and communication edges, $i_2(\xrightarrow{\mathsf{po}}\xrightarrow{\mathsf{com}}{}^{+})^{+}i_2$. We proceed by induction on the length of this sequence. In the base case there exists some i_3 such that $i_2 \xrightarrow{\mathsf{po}} i_3 \xrightarrow{\mathsf{com}}{}^{+} i_2$ which wraps around to give $i_2 \xrightarrow{\mathsf{po}} i_3 \xrightarrow{\mathsf{com}}{}^{+} i_2 \xrightarrow{\mathsf{po}} i_3$. Then Push Trace-ordered applies to give $i_2 \xrightarrow{\mathsf{to}} i_2$, but this is a contradiction since the trace order is total. In the inductive case we use the same argument and connect the trace order from the inductive hypothesis to give a contradiction. ◂

From Theorem 18 we can derive two corollaries. The first shows that when all accesses are Volatile the semantics is SC. The second shows that when all accesses have full fences between them, represented by `spush` in the model, the semantics is SC. The structure of the model and our definition for Volatile accesses shines through here as both results follow directly from a single result about the behavior of full fences.

▸ Corollary 4 (All Volatile SC). [coq/SC/Volatile.v, volatile_sc]
If H is trace coherent, acyclic($\xrightarrow{\mathsf{co}}$), all accesses are executed, and all accesses are Volatile mode accesses then acyclic($\xrightarrow{\mathsf{sc}}$).

Proof. If all accesses are volatile then any two program order accesses are `push` ordered and we can appeal to Theorem 18. ◂

▸ Corollary 5 (All Specified Push SC). [coq/SC/Volatile.v, spush_sc]
If H is trace coherent, acyclic($\xrightarrow{\mathsf{co}}$), all accesses are executed, and all program ordered (`po`) accesses are ordered by specified push order (`spush`) then acyclic($\xrightarrow{\mathsf{sc}}$).

Proof. If any two program order accesses have a specified push order then they are similarly `push` ordered and we can again appeal to Theorem 18. ◂

I The Standard DRF-SC Theorem

▸ Theorem 15 (DRF-SC). Given a program P, if all its SC-consistent executions are data-race-free or only have volatile-races, then the set of all JAM-consistent executions of P coincide with the set of SC-consistent executions.

Proof. Let P be a program, and suppose all its SC-consistent executions only has Volatile-races. We want to show that P has no weak behavior. Toward contradiction, let's assume there exists an execution H of P such that H is JAM-consistent but not SC-consistent.

▸ Definition 16. An execution H' is called a *prefix* of an execution H if H' is obtained by restricting H to a set of events E such that:

1. the set of initialization events $E_0 \in E$
2. for any event $b \in E$, if there is $a \xrightarrow{\mathsf{po}} b$ or $a \xrightarrow{\mathtt{rf}} b$ in H, then $a \in E$. (Closed with respect to $(H.\mathsf{po} \cup H.\mathtt{rf})$)

I.0.0.1 Claim 1

Any prefix of a JAM-consistent execution is JAM-consistent.

I.0.0.2 Claim 2

Any prefix of an SC-consistent execution is SC-consistent.

Proof. The above two claims are true because a prefix consists of a subset of edges and events of the original execution. If there is a cycle that violates the requirements of the memory models, then the same cycle is present in the original execution graph. Therefore, since we assumed H is JAM-consistent, any prefix of H is also JAM-consistent. ◀

I.0.0.3 Notations

For a set of events E, let $\Pi(E)$ denote the set of all pairs $\langle a, b\rangle \in E \times E$ of conflicting events, such that $\{H.AccessMode(a), H.AccessMode(b)\} \neq \{\mathsf{Volatile}\}$ and $\langle a, b\rangle$, $\langle b, a\rangle \notin (H.\mathsf{po} \cup H.\mathtt{rf} \restriction_{\mathsf{Volatile}})^+$.

$$\Pi(E) = \{\langle a, b\rangle \in E \times E \mid \{H.AccessMode(a), H.AccessMode(b)\} \neq \{\mathsf{Volatile}\},$$
$$\langle a, b\rangle, \langle b, a\rangle \notin (H.\mathsf{po} \cup H.\mathtt{rf} \restriction_{\mathsf{Volatile}})^+\}$$

Let $a_1, ..., a_n$ be an enumeration of events ordered by trace orders (recall that trace order is a total order among the events in an execution that is compatible with $(H.\mathsf{po} \cup H.\mathtt{rf})^+$).

Let E_i denotes the subset of events $E_0 \cup \{a_1, ..., a_i\}$ and H_i be the execution restrict to E_i. This is easy to see that each H_i is a prefix to H because the trace order is compatible with $(H.\mathsf{po} \cup H.\mathtt{rf})^+$. Therefore, H_i is also JAM-consistent by **Claim 1**.

I.0.0.4 Claim 3

For every $1 \leq i \leq n$, if $\Pi(E_i) = \emptyset$, then H_i is SC-consistent.

Proof. Suppose that $\Pi(E_i) = \emptyset$. Then, for every conflicting pair $\langle a, b\rangle$, either $H.AccessMode(a) = H.AccessMode(b) = \mathsf{Volatile}$ or $\langle a, b\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \restriction_{\mathsf{Volatile}})^+$ or $\langle b, a\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \restriction_{\mathsf{Volatile}})^+$.

1. For any $a \xrightarrow{\mathtt{rf}} b$ in H_i.
   - If $H.AccessMode(a) = H.AccessMode(b) = \mathsf{Volatile}$, then $\langle a, b\rangle \in H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}}$.
   - If $\langle a, b\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+$, then $\langle a, b\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+$.
   - If $\langle b, a\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+$, then there is a $(\mathsf{po} \cup \mathtt{rf})+$ cycle between a and b. By the No-Thin-Air requirement, H_i is not JAM-consistent. Contradicting to our previous assumption. Therefore it is impossible to have $\langle b, a\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+$.

   Thus, $H_i.\mathtt{rf} \subseteq (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+$
2. For any $a \xrightarrow{\mathtt{co}} b$ in H_i,
   - If $H.AccessMode(a) = H.AccessMode(b) = \mathsf{Volatile}$, then $\langle a, b\rangle \in H.\mathtt{co} \upharpoonright_{\mathsf{Volatile}}$.
   - If $\langle a, b\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+$, then $\langle a, b\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+$.
   - If $\langle b, a\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+$, then the the domains of the $H.\mathtt{rf}$ on the path from b to a has access mode equal to Volatile which includes Release semantics. Similarly, the ranges of the $H.\mathtt{rf}$ have access mode equal to Volatile which includes Acquire semantics. Therefore, $\mathsf{po} \subseteq \mathtt{ra}$ on this path. That is, we have $\langle b, a\rangle \in (H.\mathtt{ra} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+ \subseteq H.\mathtt{vvo}^+$. By $\mathtt{coww}$, we have $\langle b, a\rangle \in H.\mathtt{co}$. With $a \xrightarrow{\mathtt{co}} b$, we now have a $\mathtt{co}$ cycle, contradicting to the earlier assumption that H_i is JAM-consistent. Therefore, it is impossible that $\langle b, a\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+$.

   Thus, we have $H_i.\mathtt{co} \subseteq (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+ \cup H.\mathtt{co} \upharpoonright_{\mathsf{Volatile}}$.
3. For any $a \xrightarrow{\mathtt{fr}} b$ in H_i,
   - If $H.AccessMode(a) = H.AccessMode(b) = \mathsf{Volatile}$, then $\langle a, b\rangle \in H.\mathtt{fr} \upharpoonright_{\mathsf{Volatile}}$.
   - If $\langle a, b\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+$, then $\langle a, b\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+$.
   - If $\langle b, a\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+$, then the the domains of the $H.\mathtt{rf}$ on the path from b to a has access mode equal to Volatile which includes Release semantics. Similarly, the ranges of the $H.\mathtt{rf}$ have access mode equal to Volatile which includes Acquire semantics. Therefore, $\mathsf{po} \subseteq \mathtt{ra}$ on this path. That is, we have $\langle b, a\rangle \in (H.\mathtt{ra} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+ \subseteq H.\mathtt{vvo}^+$. Expanding the definition of $\mathtt{fr}$, there exists a write event i such that $\langle i, a\rangle \in H_i.\mathtt{rf}$ and $\langle i, b\rangle \in H_i.\mathtt{co}$. By $\mathtt{cowr}$, we have $\langle b, i\rangle \in H_i.\mathtt{co}$. Now we have a $\mathtt{co}$ cycle, contradicting with the earlier assumption that H_i is JAM-consistent. Therefore, it is impossible that $\langle b, a\rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+$.

   Thus, we have $H_i.\mathtt{fr} \subseteq (H.\mathsf{po} \cup H.\mathtt{rf} \upharpoonright_{\mathsf{Volatile}})^+ \cup H.\mathtt{fr} \upharpoonright_{\mathsf{Volatile}}$.

Therefore, we have $H_i.\mathsf{po} \cup \mathtt{rf} \cup \mathtt{fr} \cup \mathtt{co} \subseteq H.\mathsf{po} \cup \mathtt{rf} \upharpoonright_{\mathsf{Volatile}} \cup \mathtt{fr} \upharpoonright_{\mathsf{Volatile}} \cup \mathtt{co} \upharpoonright_{\mathsf{Volatile}}$. Since H_i is JAM-consistent, any prefix of H_i is JAM-consistent as well (by **Claim 1**). By our previous lemma that $Volatile \Rightarrow SC$, any prefix of H_i with all events marked as Volatile are SC-consistent. If H_i is not SC-consistent and there is a cycle of $H.\mathsf{po} \cup \mathtt{rf} \upharpoonright_{\mathsf{Volatile}} \cup \mathtt{fr} \upharpoonright_{\mathsf{Volatile}} \cup \mathtt{co} \upharpoonright_{\mathsf{Volatile}}$, then there exists a prefix of H_i containing all the events in this cycle and hence not SC-consistent. This contradicts with our previous assumption. Thus, H_i is SC-consistent. ◀

Now, continuing our proof, since H is not SC-consistent, we know that $\Pi(E) \neq \emptyset$.

I.0.0.5 Convention

In the rest of the proof, we treat an Read-modify-write (RMW) event as two separate events ordered by the $\mathtt{rmw}$ order. That is, each RMW event in H consists of two events i_1 and i_2 such that $\langle i_1, i_2\rangle \in H.\mathtt{rmw}$, where $i_1 \in H.\mathtt{R}$ and $i_2 \in H.\mathtt{W}$.

Let $k = min\{i \,|\, \Pi(E_i) \neq \emptyset\}$. Then it is clear that H_{k-1} is the maximum SC-consistent prefix of H. That is, $\Pi(E_{k-1}) = \emptyset$ and H_{k-1} is SC-consistent. We also have $\Pi(E_k) \neq \emptyset$ and H_k is not SC-consistent. That is, there exists $j < k$ such that $\{H.AccessMode(a_j), H.AccessMode(a_k)\} \neq \{\mathsf{Volatile}\}$, and $\langle a_j, a_k \rangle$, $\langle a_k, a_j \rangle \notin H.\mathsf{po} \cup H.\mathtt{rf} \restriction_{\mathsf{Volatile}}$.

I.0.0.6 Claim 4

Let $B = \{b \,|\, \langle b, a_k \rangle \in H_k.\mathsf{po}\}$, then $\langle a_j, b \rangle \notin (H.\mathsf{po} \cup H.\mathtt{rf})^+$.

Proof. Since $\Pi(E_{k-1}) = \emptyset$, we have $H_{k-1}.\mathtt{rf} \subseteq (H.\mathsf{po} \cup H.\mathtt{rf} \restriction_{\mathsf{Volatile}})^+$. If $\langle a_j, b \rangle \in (H.\mathsf{po} \cup H.\mathtt{rf})^+$, then we have $\langle a_j, b \rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \restriction_{\mathsf{Volatile}})^+$. From that we have $\langle a_j, a_k \rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \restriction_{\mathsf{Volatile}})+$ since $\langle b, a_k \rangle \in H.\mathsf{po}$, which contradicts to our previous assumption. Therefore, $\langle a_j, b \rangle \notin (H.\mathsf{po} \cup H.\mathtt{rf})^+$. ◀

1. $a_k \in H.\mathtt{W}$.

I.0.0.7 Claim 5

$\langle a_j, a_k \rangle$ forms a data race.

Proof. Since H_k is closed under $(H.\mathsf{po} \cup H.\mathtt{rf})$ and a_k is the last event in the total trace order in H_k, there is no outgoing edge from a_k. Therefore, $\langle a_k, a_j \rangle \notin (H.\mathsf{po} \cup H.\mathtt{rf})^+$. In addition, Since a_k is a write, we cannot have $\langle a_j, a_k \rangle \in H.\mathtt{rf}$. Therefore, if $\langle a_j, a_k \rangle \in H.\mathtt{hb}$, then it would imply that $\langle a_j, b \rangle \in (H.\mathsf{po} \cup H.\mathtt{rf})^+$ for some b such that $\langle b, a_k \rangle \in H.\mathsf{po}$, which contradicts with **Claim 4**. Therefore, $\langle a_j, a_k \rangle \notin (H.\mathsf{po} \cup H.\mathtt{rf})^+$ and $\langle a_j, a_k \rangle$ forms a race in H_k. ◀

I.0.0.8 Claim 6

H_k is not SC-consistent.

Proof. Given that $\langle a_j, a_k \rangle$ forms a data race in H_k and $\{H.AccessMode(a_j), H.AccessMode(a_k)\} \neq \{\mathsf{Volatile}\}$, this follows from the assumption. ◀

I.0.0.9 Claim 7

There does not exists a read event b such that $\langle b, a_k \rangle \in H.\mathtt{rmw}$.

Proof. Suppose toward conradiction that there is b such $\langle b, a_k \rangle \in H.\mathtt{rmw}$. Since H_k is not SC-consistent, there is cycle of $(H_k.\mathsf{po} \cup H_k.\mathtt{rf} \cup H_k.\mathtt{fr} \cup H_k.\mathtt{co})^+$. Since H_{k-1} is SC-consistent, it must be that a_k is part of the cycle in H_k. That is, there is a $(H_k.\mathsf{po} \cup H_k.\mathtt{rf} \cup H_k.\mathtt{fr} \cup H_k.\mathtt{co})$ edge from a_k. Additionally, it cannot be a po or $\mathtt{rf}$ because H_k is a closed prefix of H. Since a_k is a write, it cannot be $\mathtt{fr}$ either. Therefore, there is some c in H_{k-1} such that $\langle a_k, c \rangle \in H.\mathtt{co}$ and $\langle c, a_k \rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \cup H.\mathtt{fr} \cup H.\mathtt{co})^+$. Extending the edges, we have $\langle c, d \rangle \in (H.\mathsf{po} \cup H.\mathtt{rf} \cup H.\mathtt{fr} \cup H.\mathtt{co})^*$, and $\langle d, a_k \rangle \in (H.\mathtt{co} \cup H.\mathtt{fr} \cup H.\mathsf{po})$. We analyze each case below.

- $\langle d, a_k \rangle \in H.\mathtt{co}$. Then we know that $c, d \in H_{k-1}$ are writes to the same location. Since H_{k-1} is SC-consistent, and $\langle c, d \rangle \in (H_{k-1}.\mathsf{po} \cup H_{k-1}.\mathtt{rf} \cup H_{k-1}.\mathtt{fr} \cup H_{k-1}.\mathtt{co})^*$, we have $\langle c, d \rangle \in H_{k-1}.\mathtt{co}^*$. If $c = d$, then we have a $H_k.\mathtt{co}$ cycle between c and a_k, making H_k not JAM-consistent. Therefore, we have $\langle c, d \rangle \in H_{k-1}.\mathtt{co}$. But we also have $\langle d, a_k \rangle, \langle a_k, c \rangle \in H_k.\mathtt{co}$. Together, they yield a $\mathtt{co}$ cycle, contradicting with our eariler assumption that H_k is JAM-consistent.

- $\langle d, a_k\rangle \in H.\mathtt{fr}$. Then we know d is a read. Hiven that $\langle a_k, c\rangle \in H.\mathtt{co}$, we can infer that $\langle d, c\rangle \in H_{k-1}.\mathtt{fr}$. But we also have $\langle c, d\rangle \in (H_{k-1}.\mathtt{po} \cup H_{k-1}.\mathtt{rf} \cup H_{k-1}.\mathtt{fr} \cup H_{k-1}.\mathtt{co})^*$, which forms a cycle between c and d, contradicting to the assumption that H_{k-1} is SC-consistent.
- $\langle d, a_k\rangle \in H.\mathtt{po}$. Then we have $\langle d, b\rangle \in H_{k-1}.\mathtt{po}^?$. Since $\langle b, a_k\rangle \in H_k.\mathtt{rmw}$, by `cormwexcl`, we know that there exists some e such that $\langle e, b\rangle \in H_{k-1}.\mathtt{rf}$ and $\langle e, c\rangle \in H_{k-1}.\mathtt{co}$. Therefore, $\langle b, c\rangle \in H_{k-1}.\mathtt{fr}$. However, we also have $\langle c, d\rangle \in (H_{k-1}.\mathtt{po} \cup H_{k-1}.\mathtt{rf} \cup H_{k-1}.\mathtt{fr} \cup H_{k-1}.\mathtt{co})^*$ and $\langle d, b\rangle \in H_{k-1}.\mathtt{po}^?$. Now we have a cycle contradicting to the assumption that H_{k-1} is SC-consistent.

◀

Now let H'_k be a transformation of H_k such that for all $\langle a_k, c\rangle \in H_k.\mathtt{co}$, we have $\langle c, a_k\rangle \in H'_k.\mathtt{co}$. Since there is no b such that $\langle b, a_k\rangle \in H.\mathtt{rmw}$, transforming the `co` edges in H_k in this way won't affect any other `fr` or `rf` edges in H_k. As a result, H'_k is the same as H_k except for the `co` edges. Moreover, H'_k is SC-consistent because we have established that the only way to form a cycle in H_k is to have $\langle a_k, c\rangle \in H_k.\mathtt{co}$ for some c and H'_k essentially breaks the cycle by flipping the `co` arrows. However, we still have $\langle a_j, a_k\rangle$ forming a race in H'_k and $\{H'_k.AccessMode(a_j), H'_k.AccessMode(a_k)\} \neq \{\mathsf{Volatile}\}$.

I.0.0.10 Claim 8

All prefixes of SC-consistent executions of P are data race free.

Proof. This follows from the fact that prefixes are closed under $\mathtt{po} \cup \mathtt{rf}$. ◀

I.0.0.11 Claim 9

H'_k is a prefix of some SC-consistent execution of P.

Proof. Since H'_k is SC-consistent and closed under $\mathtt{po} \cup \mathtt{rf}$, we can construct an SC-consistent execution H' such that for all event $i \in H'.E \backslash H'_k.E$ and $e \in H'_k.E$, we have $\langle e, i\rangle \in H'.\mathtt{to}$ (trace order). ◀

It is clear that the fact H'_k has a data race $\langle a_j, a_k\rangle$ contradicts with **Claim 8** and **Claim 9**.

2. $a_k \in H.\mathtt{R}$.
Then we know that a_j is a write.
Let $E = \{a \in H.E \,|\, \langle a, a_k\rangle \in (H.\mathtt{po} \cup H_{k-1}.\mathtt{rf})^*\} \cup \{a \in H.E \,|\, \langle a, a_j\rangle \in (H.\mathtt{po} \cup H_{k-1}.\mathtt{rf})^*\}$. Note that there does not exist any $a \in E$ such that $\langle a_j, a\rangle \in (H.\mathtt{po} \cup H_{k-1}.\mathtt{rf})^*$ and $\langle a, a_k\rangle \in (H.\mathtt{po} \cup H_{k-1}.\mathtt{rf})^*$ since H_{k-1} is closed under $(H.\mathtt{po} \cup H.\mathtt{rf})$.
Let H' be the restriction of H_k to the events in E.

I.0.0.12 Claim 10

$\langle a_j, a_k\rangle$ forms a data race in H'

Proof. Since a_k is a read event, there is no out-going `rf` edge from a_k. Because H' itself is closed under $H.\mathtt{po} \cup H.\mathtt{rf}$, we have $\langle a_k, a_j\rangle \notin (H'.\mathtt{po} \cup H'.\mathtt{rf})^+$. In addition, there is no out-going edge from a_j in H', so $\langle a_j, a_k\rangle \notin (H'.\mathtt{po} \cup H'.\mathtt{rf})^+$. ◀

Now, we want to construct an SC-consistent execution that contains the race $\langle a_j, a_k\rangle$ to show a contradiction.
Let x be the location a_k accesses and c be the last write to x according to H'.co.

- $c \neq a_j$. Let d be the write to x such that $\langle d, a_k\rangle \in H'$.rf. We transform H' so that a_k reads from c instead of d. That is, we remove $\langle d, a_k\rangle$ from H'.rf, change the value of a_k to the value of c, and add $\langle c, a_k\rangle$ to H'.rf. The immediate consequence of this transformation is, for all e such that $\langle d, e\rangle \in H'$.co, the fr edge from a_k to e are also removed. Since a_k is a read event, there cannot be rf or co edge going out from a_k. As a result, we cannot form a $(\mathtt{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr})^+$ cycle with a_k and the resulting execution is SC-consistent. Then the fact that $\langle a_j, a_k\rangle$ is a data race gives us a contradiction.
- $c = a_j$. That is, a_k forms a race with the last write to the same location. Let d be the write to x such that $\langle a, a_k\rangle \in H'$.rf. We transform H' so that a_k reads from the write that immediately co ordered before c. Let e be that write. We remove $\langle d, a_k\rangle$ from H'.rf, change the value of a_k to the value of e, and add $\langle e, a_k\rangle$ to H'.rf. Since $\langle e, c\rangle \in H'$.co, we have $\langle a_k, c\rangle \in H'$.fr. However, since $c = a_j$ and $\langle a_j, a_k\rangle$ forms a race, $\langle c, a_k\rangle \notin (H'.\mathtt{po} \cup H'.\mathtt{rf})^+$. That is, there is no path from c to a_k in H'. As a result, we cannot form any cycle with the added fr edge from a_k and the execution after the transformation is SC-consistent. Since we still have $\langle c, a_k\rangle$ forming a race, we now have a contradiction.

◀

J A Proof of "Model-agnostic" DRF-SC

▸ THEOREM 16 (EXECUTION-DRF). Any *JAM*-consistent execution that is data race free or only has volatile-races is SC-consistent.

Proof. Let P be a program. We first consider the case without any Volatile-race. That is, all conflicting pairs of accesses are ordered by the happens-before (hb) order in some SC-consistent execution of P. We prove that there does not exist any *JAM*-consistent execution of P that is not SC-consistent. Suppose toward a contradiction that there exists an execution H of P such that H is *JAM*-consistent, race-free, but not SC-consistent. That is, H has a $(\mathtt{po} \cup \mathtt{rf} \cup \mathtt{fr} \cup \mathtt{co})+$ cycle. In addition, we assume that $\mathtt{co\text{-}jam} \subseteq \mathtt{co}$.

First note that each of the communication edges in the cycle of H are also pairs of conflicting accesses. Indeed, they are defined between accesses to the same location and at least one of the accesses is a write event. Then, by our assumption that H data-race-free, they are also ordered by the hb order. In addition, for all conflicting accesses i_1 and i_2 in H:

- $i_1 \xrightarrow{\mathtt{rf}} i_2 \Rightarrow i_1 \xrightarrow{\mathtt{hb}} i_2$
- $i_1 \xrightarrow{\mathtt{fr}} i_2 \Rightarrow i_1 \xrightarrow{\mathtt{hb}} i_2$
- $i_1 \xrightarrow{\mathtt{co}} i_2 \Rightarrow i_1 \xrightarrow{\mathtt{hb}} i_2$

because the other direction can immediately lead to contradictions.

Then the cycle in H a cycle of $(\mathtt{po} \cup \mathtt{hb})+$. We expend the definition of hb, we have a cycle of $(\mathtt{po} \cup (\mathtt{po} \cup \mathtt{sw})+)+$, which simplifies to a $(\mathtt{po} \cup \mathtt{sw})+$ cycle. Hraphically the cycle has a shape shown in Figure 17.

By the definition of sw order, the domain of each sync edge is a Release mode write (or a release fence followed by a write) and the range of sw is an Acquire mode read (or a read followed by an acquire fence). Therefore, we know that the "head" of each thread in this cycle is an Acquire read and the "end" of each thread in this cycle is a Release write. By the semantics of Release-Acquire mode, all of the po order to a Release write and all of the po order from an Acquire read is preserved and captured in the ra order, which is a subset of vo. In addition, sw $\subseteq$ vo. As a result, the cycle in H is actually a vo cycle. However, we assumed that H is JAM-consistent and by the previous lemma by Bender et al. [3], the vo order is acyclic in all JAM-consistent executions. Thus a contradiction.

We now consider the case where there are Volatile-races in the execution. We prove this by incrementally inserting a pair of Volatile-race into an execution that is data-race-free and JAM-consistent and prove that it does not introduce any weak behavior to the execution. Let H' be such an execution. As we just have shown, H' is SC-consistent. We would like to insert a pair of Volatile-race $\langle a, b\rangle$ into H'. Let T_1 be the thread where a is inserted and T_2 be the thread where b is inserted. By definition of data race, $T_1 \neq T_2$. We have three possible cases:

- T_1 and T_2 are not connected by any $(\texttt{po} \cup \texttt{sw})+$ edge before inserting $\langle a, b\rangle$ in H'. That is, there does not exist any $(\texttt{po} \cup \texttt{sw})+$ path from T_1 to T_2. Then inserting $\langle a, b\rangle$ into the execution cannot form any cycle in $(\texttt{po} \cup \texttt{rf} \cup \texttt{fr} \cup \texttt{co})+$ since it can only add at most one edge to the graph.
- There is a $(\texttt{po} \cup \texttt{sw})+$ path from T_1 to T_2. First note that H' before inserting $\langle a, b\rangle$ has a similar shape to H in Figure 17 except that it does not have a cycle. That is, if two threads are connected, then they must be connected by at least an sw edge. This implies there is a release write W^{Rel} on T_1, an acquire read R^{Acq} on T_2, and $W^{Rel} \xrightarrow{\texttt{sw}}\xrightarrow{(\texttt{po}\,|\,\texttt{sw})*} R^{Acq}$. Due to this structure, the only way to insert $\langle a, b\rangle$ and build a cycle of $(\texttt{po} \cup \texttt{rf} \cup \texttt{fr} \cup \texttt{co})+$ is to insert a before W^{Rel} and insert b after R^{Acq}. Note that this implies $a \xrightarrow{\texttt{ra}} W^{Rel} \xrightarrow{\texttt{sw}}\xrightarrow{(\texttt{po}\,|\,\texttt{sw})*} R^{Acq} \xrightarrow{\texttt{ra}} b$. So depending on what type of access a and b are, we have three cases:
  - a is a Volatile write and b is a Volatile read and $b \xrightarrow{\texttt{fr}} a$. By definition of fr, there exists a write i such that $i \xrightarrow{\texttt{rf}} b$ and $i \xrightarrow{\texttt{co}} a$. But $a \xrightarrow{\texttt{ra}} W^{Rel} \xrightarrow{\texttt{sw}}\xrightarrow{(\texttt{po}\,|\,\texttt{sw})*} R^{Acq} \xrightarrow{\texttt{ra}} b$ implies that $a \xrightarrow{\texttt{vo}} b$, and by the cowr coherence rule, we have $a \xrightarrow{\texttt{co}} i$. Now we have a co cycle, contradicting to our assumption that H'' is JAM-consistency.
  - a is a Volatile read and b is a Volatile write and $b \xrightarrow{\texttt{rf}} a$. Again, we have $a \xrightarrow{\texttt{vo}} b$ and $b \xrightarrow{\texttt{rf}} a$. Since rf $\subseteq$ vo, we get a vo cycle, contradicting to our previous assumption of JAM-consistency.
  - a is a Volatile write and b is a Volatile write and $b \xrightarrow{\texttt{co}} a$. Again, we have $a \xrightarrow{\texttt{vo}} b$ and $b \xrightarrow{\texttt{co}} a$. By coww, we have $a \xrightarrow{\texttt{co}} b$ and $b \xrightarrow{\texttt{co}} a$, a coherence cycle contradicting to our previous assumption of JAM-consistency.
- There is a $(\texttt{po} \cup \texttt{sw})+$ path from T_2 to T_1. Symmetrical to the previous case.

◀

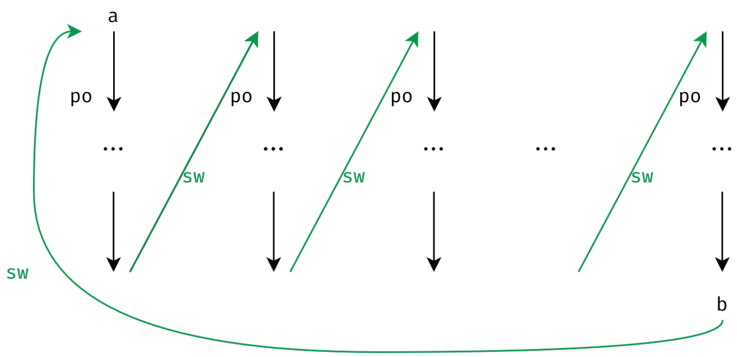


Figure 17 The Shape of the Cycle in H

K Validation

In this section, we explain our implementation of Java architecture for Herd7 [1] and experimental results with the JAM_{21} model. The source code of our Java architecture implementation will become available for artifact evaluation.

K.1 Methods supported by Java Architecture for Herd7

The list of supported methods in our implementation of Java in Herd7 [1] can be found in the following table. We provide a description for each one of the method and its corresponding action in Herd7.

Method	Memory Action	Description
`getM()`	R(M)	Read operation with access mode specified by `M`, where `M` can be omitted (Plain mode), `Opaque`, `Acquire`, or `Volatile`.
`setM(val)`	W(M)	Write operation with access mode specified by `M`, where `M` can be omitted (Plain mode), `Opaque`, `Acquire`, or `Volatile`, the value written is specified by `val`, which can be either an integer or a local variable.
`compareAndExchangeM(expect, dest)`	RMW(M)	An atomic compare and update operation with access mode specified by M, where `M` can be omitted (Volatile mode), `Acquire`, or `Release`.
`getAndOpM(val)`	RMW(M)	A numeric or bitwise atomic update operation with modifying operation specified by `Op` and access mode specified by `M`, where `Op` can be `Add`, `And`, `Or`, or `Xor`; and `M` can be omitted, `Acquire`, or `Release`.
`fullFence()`	F(Volatile)	A full fence.
`releaseFence()`	F(Release)	A release fence.
`acquireFence()`	F(Acquire)	An acquire fence.

Table 1 Methods supported by the Java Architecture

Name	JAM_{19}	JAM_{21}
volatile-non-sc.4	Sometimes	Never
volatile-non-sc.5	Sometimes	Never

Figure 18 volatile-non-sc Experimental Results

K.2 Experimental Results

In this section, we show the experimental results of running the same set of litmus tests as in JAM_{19} and compare their outcomes. Three types of results can be yielded by Herd7 at the end, `Always`, `Sometimes`, and `Never`. `Always` and `Sometimes` means the behavior specified in the litmus test is allowed, whereas `Never` means it is forbidden.

Fig. 18 shows the experimental results of running volatile-non-sc.4 example and its 5-thread version with the JAM_{19} and the JAM_{21} model. As we expected, the update to the JAM_{21} model fixes the issue we addressed earlier in the paper and the executions changed from `Sometimes` to `Forbidden`.

Fig. 19 shows the rest of the experimental results in details. Note that not all litmus tests used for JAM_{19} [3] are translatable to Java. We marked those non-translatable tests as "N/A" in the tables (since Java does not have the notion of address dependency). The result agrees with our expectation that most of the litmus tests yield the same results as JAM_{19}, except those that are related to the inconsistency issue (highlighted using bold font). We discuss each of the exceptions.

The execution graphs of IRIW-acq-sc are shown in Fig. 20. Our experimental results show that this execution is forbidden under JAM_{19} but is allowed under JAM_{21}. Under the JAM_{19} model, because the definition of `volint` includes orders from any instruction to a Volatile read program ordered after the instruction, we have `(c)`$\xrightarrow{\texttt{volint}}$`(d)` and `(e)`$\xrightarrow{\texttt{volint}}$`(f)`. Between the two threads (`P3` and `P4`), there is the visibility order `(c)`$\xrightarrow{\texttt{vo}}$`(f)`, or `(e)`$\xrightarrow{\texttt{vo}}$`(d)`. Both cases can produce the contradictory result that one of the threads observes the non-initialization write before the initialization write, i.e., a coherence cycle. Therefore, this execution is forbidden under the JAM_{19} model. In JAM_{21}, the two `volint` orders are no longer present in the execution graph because the new definition of `volint` requires both of the memory accesses to be Volatile. As a result, the execution becomes allowed under the JAM_{21} model. To see why allowing this execution is an improvement, note that the JAM_{19} model only captures the "leading fence" convention that `fullFence()` are inserted before Volatile accesses. On the other hand, if the compiler follows the "trailing fence" convention, there would not be `fullFence()`s in `P3` and `P4`. In that case, the execution is allowed. In order to accommodate both conventions, the JAM_{21} model relaxes to allow this execution.

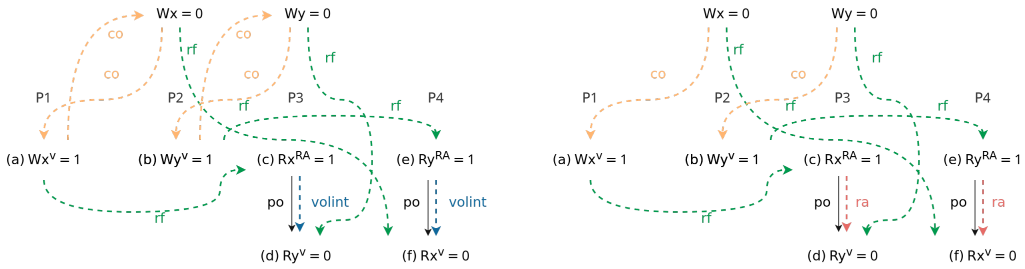
The execution graph of Z6.U are shown in Fig. 21. Due to the original problematic encoding of Volatile writes, `volint` includes orders from Volatile writes to any program ordered later memory accesses. Therefore, `(a)`$\xrightarrow{\texttt{volint}}$`(b)`, `(c)`$\xrightarrow{\texttt{volint}}$`(d)`, and `(e)`$\xrightarrow{\texttt{volint}}$`(f)`. Similar to the previous example, there are two possible visibility orders, `(a)`$\xrightarrow{\texttt{vo}}$`(f)` or `(e)`$\xrightarrow{\texttt{vo}}$`(b)`. The former case leads to the derivation of `(a)`$\xrightarrow{\texttt{co}}$`(Wx=0)`, which contradicts the assumption that all initialization writes to variable `x` are ordered before all non-initialization writes to `x`. The latter case leads to a contradiction as well. Because `(d)` reads the value written by `(e)` and `(c)`$\xrightarrow{\texttt{volint}}$`(d)`, we can infer that `(c)`$\xrightarrow{\texttt{co}}$`(e)`. If `(e)`$\xrightarrow{\texttt{vo}}$`(b)`, then `(e)`$\xrightarrow{\texttt{vo}}$`(c)`. By the coww rule, `(e)`$\xrightarrow{\texttt{co}}$`(c)`. This leads to a coherence `co` cycle between `(c)` and `(e)`. The JAM_{21} model relaxes the `volint` edges in `P1` and `P2` in order to accommodate both the leading fence convention and the trailing fence convention. If the compiler follows the convention of inserting `fullFence()` before the Volatile accesses, there is only `(a)`$\xrightarrow{\texttt{ra}}$`(b)`

Name	JAM_{19}	JAM_{21}
WRC+addrs	Never	N/A
LB+data+data-wsi	Never	N/A
W+RR	Never	Never
totalco	Never	Never
PPOCA	Sometimes	N/A
IRIW	Sometimes	Sometimes
IRIW+addrs	Sometimes	N/A
IRIW+poaas+LL	Sometimes	Sometimes
IRIW+poaps+LL	Sometimes	Sometimes
MP+dmb.sy+addr-ws-rf-addr	Sometimes	N/A
WW+RR+WW+RR+wsilp+poaa+wsilp+poaa	Sometimes	Sometimes
LB	Never	Never

Name	JAM_{19}	JAM_{21}
a1	Sometimes	Sometimes
a1_reorder	Sometimes	Sometimes
a3	Sometimes	Sometimes
a3_reorder	Sometimes	Sometimes
a3v2	Sometimes	Sometimes
a4	Never	Never
a4_reorder	Sometimes	Sometimes
arfna	Never	Never
arfna_transformed	Never	Never
b	Never	Never
b_reorder	Sometimes	Sometimes
c	Never	Never
c_p	Never	Never
c_p_reorder	Never	Never
c_pq	Never	Never
c_pq_reorder	Never	Never
c_q	Never	Never
c_q_reorder	Never	Never
c_reorder	Never	Never
cyc	Never	Never
cyc_na	Sometimes	Sometimes
fig1	Always	Always
fig6	timed out	time out
fig6_translated	timed out	time out
lb	Never	Never
linearisation	Never	Never
linearisation2	Never	Never
roachmotel	Never	Never
roachmotel2	Never	Never
rseq_weak	Sometimes	Sometimes
rseq_weak2	Always	Always
seq	Never	Never
seq2	Never	Never
strengthen	Never	Never
strengthen2	Never	Never

Name	JAM_{19}	JAM_{21}
2+2W	Never	Never
IRIW-acq-sc	Never	Sometimes
RWC+syncs	Never	Never
W+RWC	Never	Never
Z6.U	Never	Sometimes
IRIW-sc-rlx-acq	Never	Sometimes
cppmem_iriw_relacq	Sometimes	Sometimes
cppmem_sc_atomics	Never	Never
iriw_sc	Never	Never
mp_fences	Never	Never
mp_relacq	Never	Never
mp_relacq_rs	Sometimes	Sometimes
mp_relaxed	Sometimes	Sometimes
mp_sc	Never	Never
4.SB	Sometimes	Sometimes
6.SB	timeout	timeout
6.SB+prefetch	timeout	timeout
CoRWR	Never	Never
SB+SC	Sometimes	Sometimes
SB+mfences	Never	Never
SB+rfi-pos	Sometimes	Sometimes
SB	Sometimes	Sometimes
X000	Sometimes	Sometimes
X001	Sometimes	Sometimes
X002	Sometimes	Sometimes
X003	Sometimes	Sometimes
X004	Sometimes	Sometimes
X005	Sometimes	Sometimes
X006	Sometimes	Sometimes
iriw-internal	Sometimes	Sometimes
iriw	Sometimes	Sometimes
podrw000	Sometimes	Sometimes
podrw001	Sometimes	Sometimes
x86-2+2W	Sometimes	Sometimes

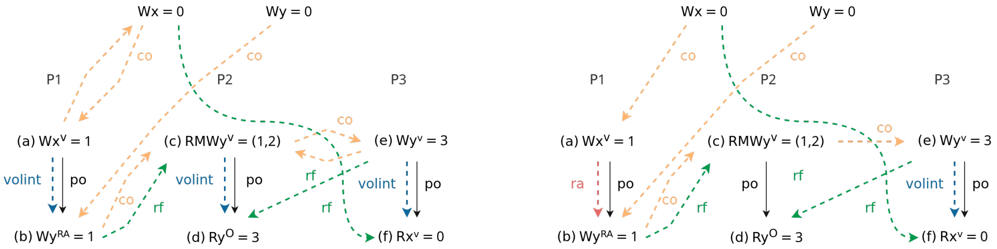
Figure 19 Litmus Test Comparisons



(a) Before: Forbidden

(b) After: Allowed

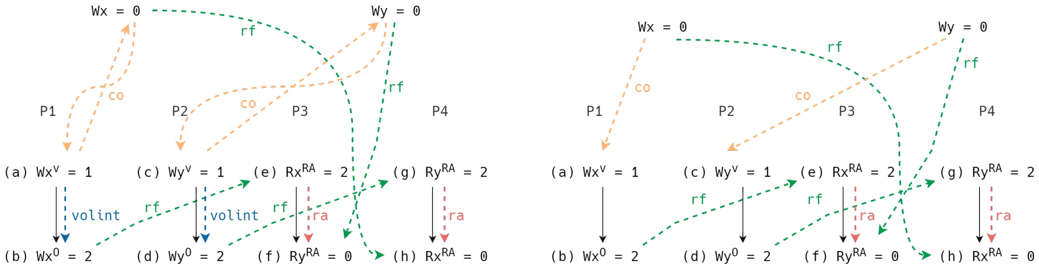
Figure 20 IRIW-acq-sc



(a) Before: Forbidden

(b) After: Allowed

Figure 21 Z6.U



(a) Before: Forbidden

(b) After: Allowed

Figure 22 IRIW-seq-rlx

Name	Power [14]
volatile-non-sc.4.ppc	`Sometimes`
volatile-non-sc.5.ppc	`Sometimes`

Figure 23 volatile-non-sc on Power with the incorrect compilation scheme

in `P1` and no synchronization between the two instructions in `P2`. Thus this execution is allowed under the JAM_{21} model.

Lastly, the execution graphs of IRIW-seq-rlx are shown in Fig. 22. Originally, due to the old encoding of Volatile writes, $\texttt{(a)}\xrightarrow{\texttt{volint}}\texttt{(b)}$ and $\texttt{(c)}\xrightarrow{\texttt{volint}}\texttt{(d)}$. Two possible visibility orders can be inferred, either $\texttt{(a)}\xrightarrow{\texttt{vo}}\texttt{(d)}$ or $\texttt{(c)}\xrightarrow{\texttt{vo}}\texttt{(b)}$. The former case leads to the conclusion that $\texttt{(a)}\xrightarrow{\texttt{co}}\texttt{Wx=0}$ because $\texttt{(a)}\xrightarrow{\texttt{vo}}\texttt{(d)}\xrightarrow{\texttt{rf}}\texttt{(g)}\xrightarrow{\texttt{ra}}\texttt{(h)}$ and $\texttt{(Wx=0)}\xrightarrow{\texttt{rf}}\texttt{(h)}$. Similarly, the latter case leads to the conclusion that $\texttt{(c)}\xrightarrow{\texttt{co}}\texttt{(Wy=0)}$ because $\texttt{(c)}\xrightarrow{\texttt{vo}}\texttt{(b)}\xrightarrow{\texttt{rf}}\texttt{(e)}\xrightarrow{\texttt{ra}}\texttt{(f)}$ and $\texttt{(Wy=0)}\xrightarrow{\texttt{rf}}\texttt{(f)}$. Each of the two conclusions contradicts the assumption that initialization writes are coherence `co` ordered before non-initialization writes. Therefore this execution is forbidden by the JAM_{19} model. In the JAM_{21} model, we relax the `volint` order in `P1` and `P2` to include the situation of which the compiler inserts the **`fullFence()`** before Volatile accesses. Thus, under the new JAM_{21} model, this execution is allowed.

In summary, the JAM_{21} model has two main differences comparing to the JAM_{19} model. First, under the JAM_{21} model, when all memory accesses use Volatile mode, the execution is guaranteed to be sequentially consistent, whereas the old JAM_{19} model has the inconsistency issue we pointed out earlier. Second, when mixing Volatile and other access modes in a program, the new JAM_{21} model accommodates both the "leading fence" convention and the "trailing fence" convention so that the compiler is free to choose either one to implement.

K.3 Compilation to Power

We translated the volatile-non-sc.4 and the volatile-non-sc.5 example to Power instructions according to the original compilation scheme:

The source code of the litmus tests in Power instructions can be found in Appendix L. Fig. 23 shows the results of running the litmus tests with Power instructions on Herd7 using Power's memory model. Both of the executions are allowed under Power's memory model, which confirms the problem we addressed in this paper. The executions becomes forbidden if we change the `lwsync` instruction in the program to `hwsync`.

L Source Code of litmus tests

In this section we provide the source code of the two examples that demonstrate the inconsistency issue we addressed in the paper. In addition, we include the same tests translated to Power instructions.

L.1 volatile-non-sc.4.litmus

```
Java volatile-non-sc.4
{
    x = 0; y = 0;
    0:X=x; 0:Y=y;
    1:X=x; 1:Y=y;
    2:X=x; 2:Y=y;
    3:X=x; 3:Y=y;
}

Thread0 {
  Y.setVolatile(2);
  int r0 = X.getVolatile();
}

Thread1 {
  X.setVolatile(1);
}

Thread2 {
  int r0 = X.getVolatile();
  Y.setVolatile(1);
}

Thread3 {
  int r0 = Y.getVolatile();
  int r1 = Y.getVolatile();
}

exists
(0:r0=0 /\ 2:r0=1 /\ 3:r0=1 /\ 3:r1=2)
```

L.2 volatile-non-sc.5.litmus

```
Java volatile-non-sc.5
{
    x = 0;
    y = 0;
    z = 0;
    0:X=x;0:Y=y;0:Z=z;
    1:X=x;1:Y=y;1:Z=z;
    2:X=x;2:Y=y;2:Z=z;
    3:X=x;3:Y=y;3:Z=z;
    4:X=x;4:Y=y;4:Z=z;
}

Thread0 {
  X.setVolatile(1);
  int r1 = Y.getVolatile();
}

Thread1 {
  Y.setVolatile(1);
}

Thread2 {
  int r1 = Y.getVolatile();
  Z.setVolatile(1);
}

Thread3 {
  Z.setVolatile(2);
  int r1 = X.getVolatile();
}

Thread4 {
  int r1 = Z.getVolatile();
  int r2 = Z.getVolatile();
}

exists
(0:r1 = 0 /\ 2:r1 = 1 /\ 3:r1 = 0
    /\ 4:r1 = 1 /\ 4:r2 = 2)
```

L.3 volatile-non-sc.4.ppc.litmus

```
PPC volatile-non-sc.4.ppc
{
    0:r1=x; 0:r2=y;
    1:r2=y;
    2:r1=x; 2:r2=y;
    3:r1=x;
}

P0           | P1           | P2           | P3           ;
li r3,2      | li r3,1      | li r3,1      | sync         ;
lwsync       | lwsync       | sync         | lwz r3,0(r1) ;
stw r3,0(r1) | stw r3,0(r2) | lwz r4,0(r2) | lwsync       ;
sync         | sync         | sync         | sync         ;
lwz r4,0(r2) |              | stw r3,0(r1) | lwz r4,0(r1) ;
lwsync       |              | sync         | lwsync       ;

exists
(0:r4=0 /\ 2:r4=1 /\ 3:r3=1 /\ 3:r4=2)
```

L.4 volatile-non-sc.5.ppc.litmus

```
PPC volatile-non-sc.5.ppc
{
    0:r1=x; 0:r2=y;
    1:r2=y;
    2:r2=y; 2:r3=z;
    3:r1=x; 3:r3=z;
    4:r3=z;
}
 P0           | P1           | P2             | P3             | P4           ;
 li r4,1      | li r4,1      | li r4,1        | li r4,2        | sync         ;
 lwsync       | lwsync       | sync           | lwsync         | lwz r4, 0(r3);
 stw r4,0(r1) | stw r4,0(r2) | lwz r5, 0(r2)  | stw r4, 0(r3)  | lwsync       ;
 sync         | sync         | lwsync         | sync           | sync         ;
 lwz r5,0(r2) |              | stw r4,0(r3)   | lwz r5, 0(r1)  | lwz r5, 0(r3);
 lwsync       |              | sync           | lwsync         | lwsync       ;

exists
(0:r5 = 0 /\ 2:r5 = 1 /\ 3:r5 = 0 /\ 4:r4 = 1 /\ 4:r5 = 2)
```

M Full Trace and Litmus Test the example in Section. 2

The litmus test of the example of Fig. 2 is shown below. We labeled each memory instruction (in blue) in the litmus test for better readability of the trace. We obtained the trace by running the `ppcmem` tool by [14] in the online interactive mode.

```
PPC volatile-non-sc.4.ppc
{
0:r1=x; 0:r2=y;
1:r2=y;
2:r1=x; 2:r2=y;
3:r1=x;
}

P0              | P1              | P2              | P3                ;
li r3,2         | li r3,1         | li r3,1         | r:   sync         ;
a: lwsync       | f: lwsync       | m: sync         | s:   lwz r3,0(r1) ;
b: stw r3,0(r1) | g: stw r3,0(r2) | n: lwz r4,0(r2) | t:   sync         ;
c: sync         | h: sync         | o: lwsync       | t16: lwz r4,0(r1) ;
d: lwz r4,0(r2) |                 | p: stw r3,0(r1) | t17: sync         ;
e: sync         |                 | q: sync         |                   ;

exists
(0:r4=0 /\ 2:r4=1 /\ 3:r3=1 /\ 3:r4=2)
```

One of the traces to show that this execution is allowed

```
(0:0) Commit reg or branch: li r3,2
(1:6) Commit reg or branch: li r3,1
(2:10) Commit reg or branch: li r3,1
(0:1) Commit barrier: lwsync: a:lwsync
(1:) Barrier propagate to thread: a:lwsync  to Thread 1
(2:) Barrier propagate to thread: a:lwsync  to Thread 2
(3:) Barrier propagate to thread: a:lwsync  to Thread 3
(1:7) Commit barrier: sync: f:Sync
(0:) Barrier propagate to thread: f:Sync  to Thread 0
(2:) Barrier propagate to thread: f:Sync  to Thread 2
(3:) Barrier propagate to thread: f:Sync  to Thread 3
Acknowledge sync: Sync f:Sync
(2:11) Commit barrier: sync: m:Sync
(0:) Barrier propagate to thread: m:Sync  to Thread 0
(1:) Barrier propagate to thread: m:Sync  to Thread 1
(3:) Barrier propagate to thread: m:Sync  to Thread 3
Acknowledge sync: Sync m:Sync
(3:16) Commit barrier: sync: r:Sync
(0:) Barrier propagate to thread: r:Sync  to Thread 0
(1:) Barrier propagate to thread: r:Sync  to Thread 1
(2:) Barrier propagate to thread: r:Sync  to Thread 2
Acknowledge sync: Sync r:Sync

(1:8) Commit write: stw r3,0(r2): g:W y=1 i:W x=0,j:W y=0
Write reaching coherence point: g:W y=1
(2:) Write propagate to thread: g:W y=1 to Thread 2
(2:12) Read from storage subsystem: lwz r4,0(r2) (from g:W y=1)
(2:12) Commit read: lwz r4,0(r2): n:R y=1
(2:13) Commit barrier: lwsync: o:Lwsync
(2:14) Commit write: stw r3,0(r1): p:W x=1 g:W y=1,i:W x=0
Write reaching coherence point: p:W x=1
(3:) Write propagate to thread: g:W y=1 to Thread 3
(3:) Barrier propagate to thread: o:Lwsync  to Thread 3
(3:) Write propagate to thread: p:W x=1 to Thread 3
(3:17) Read from storage subsystem: lwz r3,0(r1) (from p:W x=1)
(3:17) Commit read: lwz r3,0(r1): s:R x=1
(0:2) Commit write: stw r3,0(r1): b:W x=2 i:W x=0,j:W y=0
Write reaching coherence point: b:W x=2
```

```
(3:) Write propagate to thread: b:W x=2 to Thread 3
(0:3) Commit barrier: sync: c:Sync
(3:) Barrier propagate to thread: c:Sync  to Thread 3
(1:) Barrier propagate to thread: o:Lwsync  to Thread 1
(1:) Write propagate to thread: p:W x=1 to Thread 1
(1:) Write propagate to thread: b:W x=2 to Thread 1
(1:) Barrier propagate to thread: c:Sync  to Thread 1
(2:) Write propagate to thread: b:W x=2 to Thread 2
(2:) Barrier propagate to thread: c:Sync  to Thread 2
Acknowledge sync: Sync c:Sync
(0:4) Read from storage subsystem: lwz r4,0(r2) (from j:W y=0)
(0:4) Commit read: lwz r4,0(r2): d:R y=0
(0:) Write propagate to thread: g:W y=1 to Thread 0
(0:) Barrier propagate to thread: o:Lwsync  to Thread 0
(3:18) Commit barrier: sync: t:Sync
(0:) Barrier propagate to thread: t:Sync  to Thread 0
(1:) Barrier propagate to thread: t:Sync  to Thread 1
(2:) Barrier propagate to thread: t:Sync  to Thread 2
Acknowledge sync: Sync t:Sync
(3:19) Read from storage subsystem: lwz r4,0(r1) (from b:W x=2)
(3:19) Commit read: lwz r4,0(r1): t16:R x=2

(0:5) Commit barrier: sync: e:Sync
(1:) Barrier propagate to thread: e:Sync  to Thread 1
(2:) Barrier propagate to thread: e:Sync  to Thread 2
(3:) Barrier propagate to thread: e:Sync  to Thread 3
Acknowledge sync: Sync e:Sync
(1:9) Commit barrier: sync: h:Sync
(0:) Barrier propagate to thread: h:Sync  to Thread 0
(2:) Barrier propagate to thread: h:Sync  to Thread 2
(3:) Barrier propagate to thread: h:Sync  to Thread 3
Acknowledge sync: Sync h:Sync
(2:15) Commit barrier: sync: q:Sync
(0:) Barrier propagate to thread: q:Sync  to Thread 0
(1:) Barrier propagate to thread: q:Sync  to Thread 1
(3:) Barrier propagate to thread: q:Sync  to Thread 3
Acknowledge sync: Sync q:Sync
(3:20) Commit barrier: sync: t17:Sync
(0:) Barrier propagate to thread: t17:Sync  to Thread 0
(1:) Barrier propagate to thread: t17:Sync  to Thread 1
(2:) Barrier propagate to thread: t17:Sync  to Thread 2
Acknowledge sync: Sync t17:Sync

Result:
0:r4=0; 2:r4=1; 3:r3=1; 3:r4=2;
```