On Validity of Program Transformations in the Java Memory Model

Jaroslav Ševčík and David Aspinall

LFCS, School of Informatics, University of Edinburgh

Abstract. We analyse the validity of several common program transformations in multi-threaded Java, as defined by the Java Memory Model (JMM) section of Chapter 17 of the Java Language Specification. The main design goal of the JMM was to allow as many optimisations as possible. However, we find that commonly used optimisations, such as common subexpression elimination, can introduce new behaviours and so are invalid for Java. In this paper, we describe several kinds of transformations and explain the problems with a number of counterexamples. More positively, we also examine some valid transformations, and prove their validity. Our study contributes to the understanding of the JMM, and has the practical impact of revealing some cases where the Sun Hotspot JVM does not comply with the Java Memory Model.

1 Introduction

Although programmers generally assume an interleaved semantics, the Java Language Specification [11] defines more relaxed semantics, which is called the Java Memory Model [12, 19]. The reasons for having a weaker semantics become apparent from the following example:

Initially, `x = y = 0`	

`x = 1`
`if (y==1)`
`  print x`
	`if (x==1)`
	`{ x = 0`
	`  y = 1}`

The question is: can this program ever print 1? In the interleaved semantics, the answer is *no*, because if the program prints at all then all the instructions of the second thread must be executed between the statements `x=1` and `if (y==1)` of the first thread. Hence, if the program prints, the write `x=1` must be overwritten by the assignment `x=0`, and the program prints 0.

In reality, a modern optimising compiler, such as Sun HotSpot JVM or GCJ, will replace `print x` by `print 1`, because the read of `x` can reuse the value previously written to `x`. After this optimisation, the program can print 1, which was not a possible behaviour of the original program. One could argue that compilers should only perform safe optimisations; however, most modern processors would perform similar optimisations. Preventing the hardware from optimising memory accesses comes at much higher cost than a missed optimisation in a

compiler—typical memory barrier instructions consume hundreds of cycles and should be avoided if they are not necessary.

Instead of guaranteeing sequential consistency for all programs, the Java Language Specification defines a semantics that guarantees sequential consistency (interleaved semantics) for data race free programs, while giving some basic security guarantees for programs with data races. The authors of the Java Memory Model claim that the JMM is flexible enough to validate commonly used hardware and compiler optimisations. They give a theorem in [19], which states that reordering of certain combinations of statements is a valid transformation. However, Cenciarelli et al. [9] discovered a counterexample, which shows that reordering of independent memory accesses is invalid in the JMM.

This raises several questions: What common transformations are valid in the JMM? Can we fix the memory model so that more or all these transformations become valid? We have made initial steps to address this question—in earlier work we suggested a subtle variation of the JMM definition and conjectured that their version allows reordering of independent statements.

Contribution. In this paper, we analyse several commonly used local optimisations and classify them by their validity in the Java Memory Model. We prove that removal of redundant reads after writes and writes after writes are valid transformations in the JMM. With the alternative definition suggested in [5] we also establish validity of reordering of independent statements. On the other hand, we demonstrate that some other cases of reordering, which [19] claims to be valid, are not generally valid transformations. For example, swapping a normal memory access with a consequent lock can introduce new behaviours, and thus is not a valid transformation. Another example of an invalid transformation is reusing a value of a read for a subsequent read, or an introduction of an irrelevant read. With this analysis, we establish that the JMM is still flawed, because these transformations are performed by hardware and compilers. Even Sun's Hotspot JVM [20] performs transformations that are not compliant with the JMM.

1.1 Introduction to the JMM

We illustrate the key properties of the JMM on three canonical examples (from [19]), given in Fig. 1. The programs show statements in parallel threads, operating on thread-local registers (`r1`, `r2`, ...) and shared memory locations (`x`, `y`, ...). We assume no aliasing, different location names denote different locations.

In an interleaved semantics, program A could not result in `r1` = `r2` = 1, because one of the statements `r1=x`, `r2=y` must be executed first, thus either `r1` or `r2` must be 0. However, current hardware can, and often does, execute instructions out of order. Imagine a scenario where the read `r1=x` is too slow because of cache management. The processor can realise that the next statement `y=1` is independent of the read, and instead of waiting for the read it performs the write. The second thread then might execute both of its instructions, seeing the write `y=1` (so `r2` = 1). Finally, the postponed read of `x` can see the value

A. (allowed)

```
initially x = y = 0
r1 = x | r2 = y
y = 1  | x = 1
```

B. (prohibited)

```
initially x = y = 0
lock m1    | lock m2
 r1=x      |  r2=y
unlock m1  | unlock m2
lock m2    | lock m1
 y=1       |  x=1
unlock m2  | unlock m2
```

C. (prohibited)

```
initially x = y = 0
r1 = x  | r2 = y
y = r1  | x = r2
```

Is it possible to get $\texttt{r1} = \texttt{r2} = 1$ at the end of an execution?

Fig. 1. Examples of legal and illegal executions.

1 written by the second thread, resulting in $\texttt{r1} = \texttt{r2} = 1$. Similar non-intuitive behaviours could result from simple compiler optimisations, as illustrated in the introduction.

However, there are limits on the optimisations allowed—if the programmer synchronises properly, e.g., by guarding each access to a field by a synchronised section on a designated monitor, then the program should only have sequentially consistent behaviours. This is why the behaviour $\texttt{r1} = \texttt{r2} = 1$ must be prohibited in program B of Fig. 1. This guarantee for data race free programs is called DRF guarantee.

Even if a program contains data races, there must be some security guarantees. Program C in Fig. 1 illustrates an unwanted "out-of-thin-air" behaviour—if a value does not occur anywhere in the program, it should not be read in any execution of the program. The out-of-thin-air behaviours could cause security leaks, because references to objects from possibly confidential parts of program could suddenly appear as a result of a self-justifying data race. This might let an applet on a web page to see possibly sensitive information, or, even worse, get a reference to an object that allows unprotected access to the host computer.

2 Transformations and Traces

In this section we give an overview of the classes of program transformations that we have considered. Most common compiler transformations, such as common subexpression elimination, dead code elimination, and various types of loop optimisations can be expressed as a composition of our basic transformations. Similarly to [19], we will consider a transformation valid if it does not introduce any new behaviours. A valid transformation may reduce the possible behaviours. In Table 1 we classify the transformations by their validity under sequential consistency (column 'SC'), in the current Java Memory Model (column 'JMM'), and in the memory model modification suggested in [5] (column 'JMM-Alt'). Note that the JMM is in fact stricter than sequential consistency in terms of closure under some transformations, even though the JMM is more relaxed in the sense that any sequentially consistent execution is a JMM execution.

In the following subsections we describe the transformations and explain them on examples. The proofs and counterexamples for (in)validity in the JMM will follow in Sect. 4, after we explain the mechanics of the JMM in Sect. 3.

Table 1. Validity of transformations in the JMM.

Transformation	SC	JMM	JMM-Alt
Trace-preserving transformations	✓	✓	✓
Reordering normal memory accesses	×	×	✓
Redundant read after read elimination	✓	×	×
Redundant read after write elimination	✓	✓	✓
Irrelevant read elimination	✓	✓	✓
Irrelevant read introduction	✓	×	×
Redundant write before write elimination	✓	✓	✓
Redundant write after read elimination	✓	×	×
Roach-motel reordering	×(✓for locks)	×	×
External action reordering	×	×	×

2.1 Traces

To describe some of the thread-local transformations we introduce the notion of memory traces, which also constitute the connection between the JMM and the sequential part of the Java language[1]. In fact, all our transformations can be generalised as transformations on memory traces, and we will show this later in this paper when proving validity of some transformations (Subsect. 4.2 and App. B). The memory traces are finite sequences of memory operations, which can be of the following kinds:

- *volatile read* $\mathrm{Rd}_v(v, i)$,
- *volatile write* $\mathrm{Wr}_v(v, i)$,
- *normal read* $\mathrm{Rd}(x, i)$,
- *normal write* $\mathrm{Wr}(x, i)$,
- *external action* $\mathrm{Ex}(i)$,
- *lock* $\mathrm{L}(m)$,
- *unlock* $\mathrm{U}(m)$,
- *thread start* St,
- *thread finish* Fin,

where x is a non-volatile memory location, v is a volatile memory location, i is a value, and m is a synchronisation monitor. In the spirit of the JMM, we consider an external action to be an output of a value. The meaning of a sequential program is then a prefix-closed set of the memory traces that can be performed by the program.

For example, assuming that `v` is a volatile memory location, `x` and `y` non-volatile locations, `m` a monitor, and `r` a thread-local register, the meaning of the program

[1] The JMM calls this connection intra-thread consistency.

```
v:=1; lock m; r:=x; y:=r; unlock m; print(r)
```

is the prefix closure of the set

$$\{[\mathrm{St}, \mathrm{Wr}_v(v,1), \mathrm{L}(m), \mathrm{Rd}(x,i), \mathrm{Wr}(y,i), \mathrm{U}(m), \mathrm{Ex}(i), \mathrm{Fin}] \mid i \text{ is a value}\}.$$

2.2 Transformations

In the following paragraphs we describe the transformations that we have considered in our analysis. Our transformations are local, i.e., they should be valid in any context.

Trace-preserving Transformations. Because the meaning of a program in the JMM is just the set of its traces, any transformation that does not change the set of traces must trivially be valid. E.g., if both branches of a conditional—whose guard does not examine memory—contain the same code, it is valid to eliminate the conditional, as illustrated by the transformation

```
if (r1==1)
     {x=1;y=1}        ⇄       x=1
else {x=1;y=1}                y=1
```

Reordering. Reordering of independent statements is an important transformation that swaps two consecutive non-synchronisation memory accesses. It is often performed in hardware [14, 13, 25], or in a compiler's loop optimiser [16, 10]. Although Manson et al. claim this transformation to be valid in the JMM [19, Theorem 1], Cenciarelli et al. [9] found a counterexample to this. The following program transformation illustrates this particular example: it shows the reordering of two independent statements in one branch of an `if`-statement.

```
r1=z;                      r1=z;
if (r1==1)                 if (r1==1)
     {x=1;y=1}      ⟶          {x=1;y=1}
else {y=1;x=1}             else {x=1;y=1}
```

In earlier work [5], we suggested a simple fix and conjectured that it makes reordering of independent memory accesses valid. We state and prove this claim precisely in Subsect. 4.2 and App. B. Demonstrating a successful repair for this crucial property is one of the main contributions of this paper.

Redundant Read Elimination. Elimination of a redundant read is a transformation that replaces a read immediately preceded by a read or a write to the same variable by the value of that read/write. This transformation is often performed as a part of common subexpression elimination optimisations in compilers. For example, the two examples of transformations below reuse the value of `x` stored in register `r1` instead of re-reading `x`:

```
r1 = x                  r1 = x              x = r1                 x = r1
r2 = x          ⟶       r2 = r1             r2 = x         ⟶       r2 = r1
if (r1==r2)             if (r1==r2)         if (r1==r2)            if (r1==r2)
  y = 1                   y = 1               y = 1                  y = 1
```

(read after read) (read after write)

Later we will show that redundant read elimination is valid in the JMM for a read after a write, but *invalid* for a read after a read.

Irrelevant Read Elimination. A read statement can also be removed if the value of the read is not used. For example, `r1=x;r1=1` can be replaced by `r1=1`, because the register `r1` is overwritten by the value 1 immediately after reading shared variable x, and thus the value read is irrelevant for the continuation for the program. An example of this transformation is dead code elimination because of dead variables. It is valid in the JMM.

Irrelevant Read Introduction. Irrelevant read introduction is the inverse transformation to the irrelevant read elimination. It might seem that this transformation is not an optimisation, but modern processor hardware often introduces irrelevant reads speculatively. For example, the first transformation in

```
                    if (r1==1)
if (r1==1)           {r2=x;            r2=x
 {r2=x;       →       y=r2}      ⇄     if (r1==1)
  y=r2}             else r2=x            y=r2
```

introduces irrelevant read of `x` in the `else` branch of the conditional (assuming that `r2` is not used in the rest of the program). In terms of traces, this is equivalent to reading `x` speculatively, as demonstrated by the program on the right. In Subsect. 4.1, we show that this is an *invalid* transformation in the JMM.

Redundant Write Elimination. This transformation eliminates a write in two cases: (i) if it follows a read of the same value, or (ii) if it precedes another write to the same variable. For example, in the first transformation in

```
r = x
if (r == 1)   ⟶   r = x                x = 1
  x = 1                                x = 3   ⟶   x = 3
```

(write after read) (write before write)

the write `x=1` can be eliminated, because in all traces where the write occurs, it always follows a read of `x` with value 1. The other transformation shows the elimination of a previous overwritten write. This transformation is often included in peephole optimisations [4]. Similarly to the read elimination, it is valid in the JMM before a write, but *invalid* after a read.

Roach-motel Semantics. Intuitively, increasing synchronisation should limit a program's behaviours. In the limit, if a program is fully synchronised, i.e., data race free, the DRF guarantee promises only sequentially consistent behaviours. One way of increasing synchronisation is moving normal memory accesses into synchronised blocks, as in

```
x=1                 lock m
lock m                x=1
  y=1       ⟶         y=1
unlock m            unlock m
```

Although compilers do not perform this transformation explicitly, it may be performed by underlying hardware if a synchronisation action uses only one memory fence to prevent the code inside a synchronised section from being reordered to outside of the section. Manson et al. [19, Theorem 1] claim that this transformation is valid. We show a counterexample to this in Subsect. 4.1, so unfortunately it is *invalid* in general.

Reordering with external actions. As well as reordering memory operations with one another, one may consider richer reorderings, for example, reordering memory operations with external operations. This seems more likely to alter the behaviour of a program, but it is valid for data race free programs under sequential consistency. For example, the exchange of printing a constant with a memory write: `x=1;print 1` $\longrightarrow$ `print 1;x=1`. Theorem 1 of [19] incorrectly states that this transformation is valid in the JMM.

3 JMM Operationally

To reason about the Java Memory Model, we introduce an intuitive operational interpretation, based on some observations about the construction of the formal definition[2]. This re-interpretation will allow us to explain key counterexamples in a direct way in the next section. The formal definition of the memory model is used to argue about validity; our adjusted definition is given in detail App. A (the original is given in [12]).

Unlike interleaved semantics, the Java Memory Model has no explicit global ordering of all actions by time consistent with each thread's perception of time, and has no global store. Instead, executions are described in terms of memory related actions, partial orders on these actions, and a visibility function that assigns a write action to each read action. We first explain the building blocks of Java executions, then we show how Java builds *legal executions* out of simple "well-behaved" executions.

[2] Jeremy Manson made essentially the same observations in his description of his model checker for the JMM [18].

JMM actions and orders. An action is a tuple consisting of a *thread identifier*, an *action kind*, and a *unique identifier*. Action kinds were described in Sect. 2.1.

The volatile read/write and lock/unlock actions are called *synchronisation actions*. An *execution* consists of a *set of actions*, a *program order*, a *synchronisation order*, a *write-seen* function, and a *value-written* function. The program order ($\leq_{po}$) is a total order on the actions of each thread, but it does not relate actions of different threads. All synchronisation actions are totally ordered by the synchronisation order ($\leq_{so}$). From these two orders we construct a happens-before order of the execution: action a happens-before action b ($a \leq_{hb} b$) if (1) a synchronises-with b, i.e., $a \leq_{so} b$, a is an unlock of m and b is a lock of m, or a is a volatile write to v and b is a volatile read from v, or (2) $a \leq_{po} b$, or (3) there is an action c such that $a \leq_{hb} c \leq_{hb} b$. The happens-before order determines an upper bound on the visibility of writes—a read happening before a write should never see that write, and a read r should not see a write w if there is another write happening "in-between", i.e., if $w \leq_{hb} w' \leq_{hb} r$ and $w \neq w'$, then r cannot see w.[3]

We say that an execution is *sequentially consistent* if there is a total order consistent with the program order, such that each read sees the most recent write to the same variable in that order. A pair of memory accesses to the same variable is called a *data race* if at least one of the accesses is a write and they are not ordered by the happens-before order. A program is *correctly synchronised* (or *data-race-free*) if no sequentially consistent execution contains a data race.

A thorny issue is initialisation of variables. The JMM says

> The write of the default value (zero, false, or null) to each variable synchronises-with to the first action in every thread [12]

However, normal writes are not synchronisation actions and synchronises-with only relates synchronisation actions, so normal writes cannot synchronise-with any action. For this paper, we will assume that all default writes are executed in a special initialisation thread and the thread is finished before all other threads start.

Committing semantics. The basic building blocks are *well-behaved* executions, in which reads are only allowed to see writes that happen before them. In other words, in these executions reads cannot see writes through data races, and threads can only communicate through synchronisation. For example, programs A and C in Fig. 1 have just one such execution—the one, where `r1` = `r2` = 0. On the other hand, the behaviours of program B are exactly the behaviours that could be observed by the interleaved semantics, i.e. `r1` = `r2` = 0, or `r1` = 1 and `r2` = 0, or `r1` = 0 and `r2` = 1. In fact, if a program is correctly synchronised then its execution is well-behaved if and only if it is sequentially consistent [19, Lemma 2]. This does not hold for incorrectly synchronised programs (e.g., see the first counterexample in Subsect. 4.1).

[3] For details, see Defs. 2, 4 and 7 in App. A.

The Java Memory Model starts from a well-behaved execution and *commits* one or more read-write data races from the well-behaved execution. After committing the actions involved in the data races it "restarts" the execution, but this time it must execute the committed actions. This means that each read in the execution must be either committed and see the value through the race, or it must see the write that happens-before it. Similarly, all committed writes must be executed in the restarted execution and must write the same value. The JMM can repeat the process, i.e., it may choose some non-committed reads involved in a data race, commit the writes involved in these data races if they are not committed already, commit the chosen reads, and restart the execution. The executions constructed using this procedure are called *legal executions*.

The JMM imposes several requirements on the committing sequence:

1. All subsequent (restarted) executions must preserve happens-before ordering of all the committed actions. Cenciarelli et al. [9] observed that this requirement makes reordering of independent statements invalid. In our earlier work [5], we suggested that the happens-before ordering should be preserved only between a read and the write it sees. We showed there that this revision still satisfies the DRF guarantee; in this paper we further establish that validity of reordering is indeed rescued in this version.
2. If some synchronisation happens-before the committed data race(s), the synchronisation must be preserved in all subsequent executions[4].
3. All external actions that happen-before any committed action must be committed, as well.

This committing semantics imposes a causality order on races—the outcome of a race must be explained in terms of previously committed races. This prevents causality loops, where the outcome of a race depends on the outcome of the very same race, e.g., the outcome $\texttt{r1} = 1$ in program C in Fig. 1. The DRF guarantee is a simple consequence of this procedure. If there are no data races in the program, there is nothing to commit, and we can only generate well-behaved executions, which are sequentially consistent for data race free programs.

In fact, the JMM, as defined in [12], actually commits all actions in an execution, but committing a read that sees a write that happens-before it does not create any opportunities for committing new races, because reads can see writes that happen-before them in a well-behaved execution. Similarly, committing synchronisation actions does not create any committing opportunities and can be always performed in the last step. Therefore, the central issue is committing data races, and we explain our examples using this observation.

Example. An example should help make the operational interpretation clearer. First, we demonstrate the committing semantics on program A in Fig. 1. In the well-behaved execution of this program, illustrated by the first diagram in Fig. 2, the reads of `x` and `y` can only see the default writes of 0, because there is no synchronisation. This results in $\texttt{r1} = \texttt{r2} = 0$.

[4] For a formal definition, see rule 8 in the list that follows Def. 8.

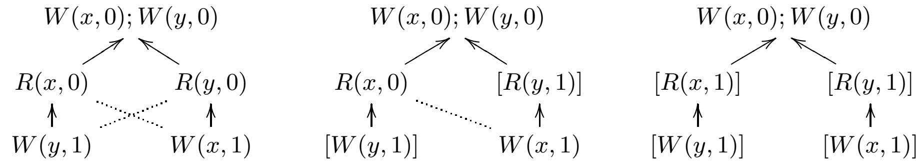

Fig. 2. Justifying executions of program A from Fig. 1.

There are two data races in this execution (depicted by the dotted lines, the solid lines represent the happens-before order)—one on `x` and one on `y`. We can commit either one of the races or both of them. Suppose we commit the race on `y`. In the second diagram we show the only restarted execution that uses this data race; the committed actions are in brackets and the committed read sees the value of (the write in) the data race. The non-committed read sees the write that happens-before it, i.e., the default write. This execution gives the result `r1` = 0 and `r2` = 1. The JMM can again decide to commit a data race from the execution. There is only one such data race. Committing the data race on `x` gives the last diagram, and results in `r1` = `r2` = 1.

4 Validity of Transformations

This sections contains the technical explanations of validity and invalidity of the transformations. All invalidity arguments will be carried in the finite version[5] of the Java Memory Model as described in [12], but the same arguments apply to the alternative weaker memory model JMM-Alt. On the other hand, the validity argument will refer to the more permissive JMM-Alt. It is straightforward to simplify the argument to prove the valid transformations of the original JMM.

4.1 Invalid Transformations

In this subsection we show and explain our counterexamples for the invalid transformations. The examples follow the same pattern—at first we list a program where a certain behaviour is not possible in the JMM, and then we show that after the transformation the behaviour becomes possible (in the JMM). This shows that the transformation in question is invalid, because any run of the transformed program should be indistinguishable from some run of the original program. In the Java Memory Model, the behaviour of a program is essentially the set of external actions, such as printing, performed by the program[6]. In our examples, we will consider final contents of registers being part of the program's behaviour, because we could observe them by printing them at the end of each thread.

[5] We use the finite version, because the infinite JMM is inconsistent [5].

[6] The definition in [19] is slightly more complex because of non-terminating executions and ordering, see Def. 10 for details. Our examples are always terminating.

Redundant Write after Read Elimination.

initially `x = 0`

`lock m1`	`lock m2`	`lock m1`
`x=2`	`x=1`	`lock m2`
`unlock m1`	`unlock m2`	`r1=x`
		`x=r1`
		`r2=x`
		`unlock m2`
		`unlock m1`

First note that no well-behaved execution of this program contains a read-write data race, so all legal executions of this program are well-behaved. Moreover, in all executions the read `r2=x` must see the write `x=r1`, because it overwrites any other write. As the write `x=r1` always writes the value that is read by `r1=x`, we have that $\texttt{r1} = \texttt{r2}$.

On the other hand, if a compiler removes the redundant write `x=r1`, the reads `r1=x` and `r2=x` can see different values in a well-behaved execution, e.g., we might get the outcome $\texttt{r1} = 1$ and $\texttt{r2} = 2$.

Redundant Read after Read Elimination. The counterexample for the elimination of a read after a read uses a trick with switching the branches of an `if` statement—in the first well-behaved execution we take one branch, and then we commit a data race so that we can take the other branch after we restart. Let us examine the program below.

`x = y = 0`

`r1=x`	`r2=y`
`y=r1`	`if (r2==1)`
	`{`**`r3=y`**`; x=r3}`
	`else x=1`

The question is whether we can observe the result $\texttt{r2} = 1$. This result is not possible in this program, but it becomes possible after rewriting `r3=y` to `r3=r2`.

First we show that this is not possible with the original program: With the initially empty commit set we can get just one well-behaved execution—the one, where $\texttt{r1} = \texttt{r2} = 0$. In this well-behaved execution, we have two data races: (i) between the actions preformed by `y=r1` and `r2=y` with value 0, (ii) between the actions performed by `r1=x` and `x=1` with value 1. If we commit (i), we are stuck with $\texttt{r2} = 0$, because all subsequent restarted executions must perform the committed read of `y` with the value 0. If we commit (ii) and restart, we get an execution, where $\texttt{r1} = 1$, so we can now commit the data race between `y=r1` and `r2=y` with value 1. After we restart the execution, we could read $\texttt{r1} = \texttt{r2} = 1$, but we cannot keep our commitment to perform the write of 1 to `x`, because the read `r3=y` must read a value that happens-before it and the only such value is the default value 0.

On the other hand, if JVM transforms the read `r3=y` into `r3=r2`, we can obtain the result $\texttt{r2} = 1$ by committing the data race between `r1=x` and `x=1`,

restarting, committing the data race between `y=r1` and `r2=y`, and restarting again. As opposed to the original program, now we can keep the commitment to write 1 to `x`, because `r3` = `r2` = 1 in the transformed program.

Roach Motel Semantics. We demonstrate the invalidity of roach motel semantics on the program:

initially `x = y = z = 0`

`lock m`
`x=2`
`unlock m`
	`lock m`
	`x=1`
	`unlock m`
		`r1=x`
		`lock m`
		`r2=z`
		`if (r1==2)`
		`  y=1`
		`else`
		`  y=r2`
		`unlock m`
			`r3= y`
			`z=r3`

This program cannot result in `r1` = `r2` = `r3` = 1 in the JMM: In all well-behaved executions of this program, we have `r1` = `r2` = `r3` = 0, and four data races—two on `x` with values 1 and 2, then on `y` and `z` with value 0. If we commit the data race on `y` (resp. `z`, resp. `x` with value 2) we would be stuck with `r3` = 0 (resp. `r2` = 0, resp. `r1` = 2), so we can only commit a race on `x`. However, if we commit the race with `x=1` and restart, we are only left with races on `z` and `y` with value 0. Committing any of these races would result in `r2` and `r3` being 0.

However, after swapping `r1=x` and `lock m` the program offers more freedom to well-behaved executions, e.g., the read `r1=x` can see value 2 (without committing any action on `x`!), and we can commit the data race on `y` with value 1 (see execution A from Fig. 3). After restarting, we can commit data race on `z` with value 1. After another restart, we change the synchronisation order so that the write `x=1` overwrites the write `x=2`, and the read `r1=x` sees value 1 (see execution B from Fig. 3). In this execution, we have `r1` = `r2` = `r3` = 1.

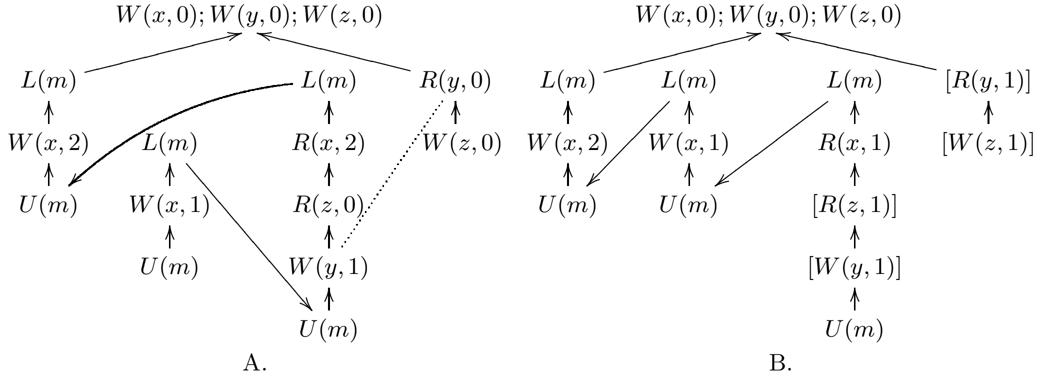

Fig. 3. Justifying and final executions for the roach motel semantics counterexample.

Note that this committing sequence respects the rule that all the subsequent restarted executions must preserve synchronisation that was used to justify the previous data races, because our committing sequence only introduces new synchronisation that in effect overwrites the write `x=2` with the write `x=1`. This problem seems to be hard to solve in a committing semantics based on well-behaved executions, because more synchronisation gives more freedom to well-behaved executions and allows more actions to be committed.

Irrelevant Read Introduction. The counterexample for irrelevant read introduction uses the trick with switching branches again. The program

```
x = y = z = 0
r1 = z                      | x = 1
if (r1==0)                  | r2 = y
{ r3 = x                    | z = r2
  if (r3==1) y = 1 }        |
else {  //r4 = x            |
        r4 = 1              |
        y = r1  }           |
```

cannot result in `r1` = `r2` = 1: its only well-formed execution has data races on `x` with value 1 and `z` with value 0. We cannot commit the data race on `z`, because then `r1` would remain 0. If we commit the data race on `x` and restart, we have a new data race between `y=1` and `r2=y`. After committing it and restarting, we can try to commit the data race on `z` with value 1. However, after this commit and restart, we cannot fulfil the commitment to perform the data race on `x`.

On the other hand, if we introduce the irrelevant read `r4=x` by uncommenting the commented-out line, we can keep the commitment to perform the committed read on `x`, and the program can result in `r1` = `r2` = 1. This seems to be another deep problem with committing semantics—even introducing a benign irrelevant read may validate some committing sequence that was previously invalid.

Reordering with external actions.

```
x = y = 0
r1=y                        | r2=x
if (r1==1)                  | y=r2
   x=1                      |
else {print "!"; x=1}       |
```

cannot result in `r1` = `r2` = 1 in the JMM, because to have `r2` = 1 we must commit the data race on `x` and, by the rule for committing external actions, also the external printing action. To get `r1` = 1 we must also commit the race on `y`, but then we are not able to keep the commitment to perform the committed printing action.

However, if we swap `print "!"` with `x=1` in the else-branch, the rule for external actions does not apply, and we can commit the race on `x`, and then the race on `y`, resulting in `r1` = `r2` = 1.

4.2 Valid Transformations

In this subsection we outline the proof of the validity of irrelevant read elimination, read after write elimination, write after write elimination, and reordering of independent non-volatile memory accesses in the weaker memory model (JMM-Alt). Using the same method one could also prove that the first three of these transformations are valid in the standard JMM [12].

The validity of a transformation says that any behaviour of the transformed program is a behaviour of the original program. We prove the validity very directly—we take an execution of the transformed program that exhibits the behaviour in question, then we apply an 'inverse' transformation to the execution, and finally we show that the untransformed execution has the same behaviour as the one of the transformed program. Since the details of the proof are somewhat technical, we show a careful proof in App. B. In this section we only explain informally the main ideas, i.e., the construction of the inverse transformation, and the relationship between the transformations on programs and the memory traces.

The main idea of the proof is that we describe transformations using their 'inverse' transformations. We will say that P' is a transformation of P if for any trace $t' \in P'$ there is an *untransformation* in P. By the untransformation we mean a trace t of P together with an injective function f that describes a valid reordering of the actions of t'. Moreover, each action of t that is not in $\mathrm{rng}(f)$ must be either (i) a redundant read after write, i.e., it must be a read of the same value as the last write to the same variable in the trace, and there cannot be any synchronisation or read from the same variable in between, or (ii) a redundant write before write, i.e., the write must precede another write to the same variable such that there is no read from the same location or synchronisation in between, or (iii) an irrelevant read, i.e., the value of the read cannot affect validity of the trace t in P. For formal details, see Def. 11. By induction on the operational execution of sequential programs, we can show that the program transformations on the syntax level implies the existence of an untransformed trace and an untransformation function for each trace of the transformed program.

For example, the program on the left in Fig. 4 can be transformed to the program on the right of the arrow, because for each trace of the transformed program there is its untransformation. For example, for the trace t' (on the right of Fig. 4) of the transformed program there is a trace t of the original program, and a function f that determines the reordering of the actions. Moreover, $\mathrm{Wr}(x, 2)$ is a redundant write before write, $\mathrm{Rd}(x, 2)$ is a redundant read after write, and $\mathrm{Rd}(y, *)$ is an irrelevant read, i.e., t is a valid trace of P if we replace $*$ by any value.

Having this definition, the proof is technical, but straightforward—given an execution of the transformed program we construct an execution of the original program by untransforming the traces of all its threads, while preserving the synchronisation order (see the details in App. B). This is possible because the definition of program transformation preserves ordering of synchronisation

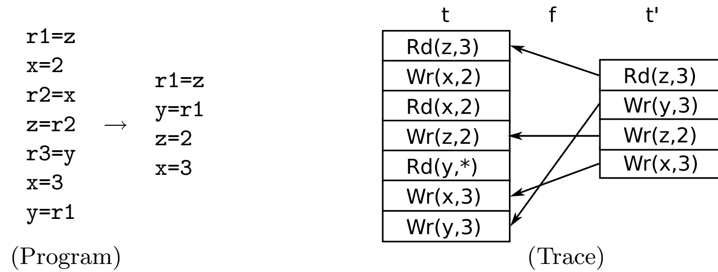

Fig. 4. Transformation of a program as a transformation on traces.

actions, thus guaranteeing consistency of the program order with the synchronisation order.

We also observe that the untransformed execution is legal—if we take the committing sequence of data races and justifying executions, and untransform the justifying executions, we get a legal committing sequence for the untransformed program (Lemma 3). We conclude that any behaviour of the transformed program is a behaviour of the original program (Theorem 1).

5 Practical Impact

The flaw in the memory model is important in theory, but it is conceivable that it might not be manifested in practical implementations, because JVMs compile to stricter memory models than the JMM. It is natural to ask whether some widely used JVM actually implements optimisations that lead to forbidden behaviours. In fact, this is indeed the case! We have experimented with the Sun Hotspot JVM [20] to discover this. For example, the first program in Fig. 5 cannot print 1 under the JMM (for details, see the counterexample for redundant read after read elimination in Subsect. 4.1). An optimising compiler can reuse the value of `y` in `r2` and transform `x=(r2==1)?y:1` → `x=(r2==1)?r2:1`, which is equivalent to the second program from Fig. 5. Then it can reorder the write to `x` with read of `y`, yielding the last program in Fig. 5. Observe that this transformed program can print 1 using the interleaving `x=1`, `r1=x`, `y=r1`, `r2:=y`, `print r2`. After minor modifications to the program, Sun Hotspot JVMs will perform these transformations, so it does not comply with the JMM[7].

The program in Fig. 5 is not data-race-free. Should we worry about behaviours of correctly synchronised programs after optimisations? We conjecture that any composition of the transformations from this paper applied to a correctly synchronised program can only yield a program that does not have any new behaviours. This means that Java implementations might be in fact correct, i.e., satisfy the DRF guarantee, and it is only the JMM specification that needs fixing.

[7] Tested on Java HotSpot(TM) Tiered VM (build 1.7.0-ea-fastdebug-b16-fastdebug, mixed mode).

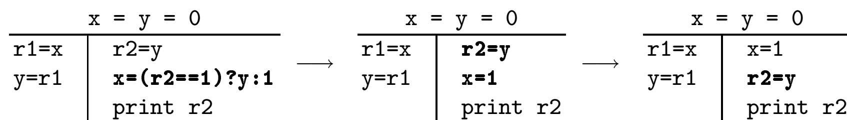

Fig. 5. Hotspot JVM's transformations violating the JMM.

6 Conclusion

We have examined the most common software and hardware local program transformations and classified them by their validity in the Java Memory Model, and its variation suggested by [5]. For each class of transformations we give either a proof of its validity or a counterexample. Despite the JMM's main design goal to enable common optimisations, we show that the JMM does not allow several commonly used optimisations although some of these transformations are valid under the natural strict memory model–sequential consistency. This is a serious flaw in the Java Memory Model, which does not seem to have an easy fix, as discussed in the explanations of the counterexamples (Subsect. 4.1).

Related Work. The computer architecture community has studied the problem of weak memory models (MM) for a long time, for a detailed survey see [3, 2]. However, the problems of MMs in programming languages seems to be more complex [21]. Most of the work has focused on alternative definitions of memory models and proving the guarantee of sequential consistency for data race free programs [17, 1, 9, 24]. E.g., Cenciarelli et al. [9] describe a subset of the JMM using the theory of configuration structures. However, they do not attempt to prove validity of compiler transformations or compliance with any hardware memory model. Saraswat et al. [24] use denotations of commands as functions on partial stores and transformations on them to describe a memory model for their X10 language. Although their work is based on transformations of denotations, it is hard to map their transformations to program transformations in a Java-like language because they use a language with restricted control-flow constructs. Moreover, both [9] and [24] use languages that do not have any general loops, and we do not see any easy way of adding them. To our knowledge, the only work dealing with a program transformation in a weak memory model is the POPL paper about the JMM [19]. Our paper shows a corrected version of their proof, together with counterexamples for cases that seem to be hard to fix. Brookes [8] studied program transformations in interleaved semantics using a trace semantics, but his technique uses traces of global states, which makes it hard to use with weak memory models.

Our work is not the first one that points out some defects in the JMM. While looking for an alternative description of the JMM based on event structures, Cenciarelli et al. [9] observed that reordering of independent statements is an invalid transformation in the JMM. In our previous work [5] we have also found

several minor flaws in the memory model. We believe that our work is the first one that shows serious flaws in the JMM and identifies some deep problems in the JMM.

Future Work. Our main objective is to analyse the effects of the above transformations on programs. We conjecture that for data race free programs these transformations cannot introduce new behaviours, and for programs with data races they satisfy some form of out-of-thin-air guarantees. To analyse the transformations we intend to continue using the trace semantics, and employ ideas from the trace semantics literature on shared memory and concurrency [7, 23, 15].

7 Acknowledgements

The authors enjoyed discussions on some of the examples in this paper with P. Cenciarelli, M. Huisman, and G. Petri. We would especially like to G. Petri, who supplied a simpler counterexample for the read after read elimination transformation, and checked our counterexamples. The first author is supported by a PhD studentship awarded by the UK EPSRC, grant EP/C537068. Both authors also acknowledge the support of the EU project MOBIUS (IST-15905). Some of the content of this paper was first presented at the VAMP'2007 workshop [6].

References

1. S. Adve. The SC- memory model for Java, 2004. http://www.cs.uiuc.edu/ sadve/jmm.
2. S. V. Adve and J. K. Aggarwal. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624, 1993.
3. S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.
4. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
5. D. Aspinall and J. Ševčík. Formalising Java's data race free guarantee. In K. Schneider and J. Brandt, editors, *TPHOLs*, volume 4732 of *LNCS*, pages 22–37. Springer, 2007.
6. D. Aspinall and J. Ševčík. Java memory model examples: Good, bad and ugly. Technical Report EDI-INF-RR-1121, School of Informatics, University of Edinburgh, 2007.
7. S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.
8. S. D. Brookes. Full abstraction for a shared variable parallel language. In *LICS*, pages 98–109. IEEE Computer Society, 1993.
9. P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *16th ESOP*, 2007.
10. C. Click. Global code motion/global value numbering. *SIGPLAN Not.*, 30(6):246–257, 1995.

11. J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.
12. J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*, chapter Memory Model, pages 557–573. Addison-Wesley Professional, July 2005.
13. Intel. A formal specification of Intel Itanium processor family memory ordering, 2002. Available from http://www.intel.com/design/itanium/downloads/251429.htm.
14. Intel. Intel 64 architecture memory ordering white paper, 2007. Available from http://www.intel.com/products/processor/manuals/318147.pdf.
15. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3):17–63, 2005.
16. K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
17. J.-W. Maessen and X. Shen. Improving the Java memory model using CRF. In *OOPSLA*, pages 1–12, New York, NY, USA, 2000. ACM Press.
18. J. Manson. *The Java memory model*. PhD thesis, University of Maryland, College Park, 2004.
19. J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.
20. M. Paleczny, C. Vick, and C. Click. The Java Hotspot(TM) server compiler. In *USENIX Java(TM) Virtual Machine Research and Technology Symposium*, April 2001.
21. W. Pugh. The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455, 2000.
22. W. Pugh and J. Manson. Java memory model causality test cases, 2004. http://www.cs.umd.edu/ pugh/java/memoryModel/CausalityTestCases.html.
23. J. C. Reynolds. Toward a grainless semantics for shared-variable concurrency. In K. Lodaya and M. Mahajan, editors, *FSTTCS*, volume 3328 of *Lecture Notes in Computer Science*, pages 35–48. Springer, 2004.
24. V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *ACM 2007 SIGPLAN Conference on Principles and Practice of Parallel Computing*. ACM, Mar 2007.
25. Sparc International. Sparc architecture manual, version 9, 2000. Available from http://developers.sun.com/solaris/articles/sparcv9.html.

A JMM Definitions

The following definitions are mostly from [12, 19]; however, we have weakened the definition of execution legality as suggested in [5]. We use letters θ for thread names, m for synchronisation monitor names, and v for variables (i.e., memory locations, in examples, `x`, `y`, `v` etc.). The abstract type $\mathcal{V}$ will denote values.

The starting point is the notion of *action*.

Definition 1. *An* action *is a memory-related operation; it is modelled by an abstract type $\mathcal{A}$ with the following properties: (1) Each action belongs to one thread, we will denote it by $T(a)$. (2) An action is one of the following* action

kinds*:*

- volatile read *of* v*,*
- volatile write *to* v*,*
- normal read *from* v*,*
- normal write *to* v*,*
- lock *on* m*,*
- unlock *on* m*,*
- thread start*,*
- thread finish*,*
- external action*.*

We denote the action kind of a *by* $K(a)$*, the action kinds will be abbreviated to* $\mathrm{Rd}_v(v)$*,* $\mathrm{Wr}_v(v)$*,* $\mathrm{Rd}(v)$*,* $\mathrm{Wr}(v)$*,* $\mathrm{L}(m)$*,* $\mathrm{U}(m)$*,* St*,* Fin*,* Ex*. An action kind also includes the associated variable or monitor. The volatile read, volatile write, lock, unlock, start, finish actions are called* synchronisation actions*.*

Definition 2. *An* execution E *is a tuple* $E = \langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$*, where* $A \subseteq \mathcal{A}$ *is a set of actions;* P *is a program, represented as a thread-indexed set of memory traces; the partial order* $\leq_{po} \subseteq A \times A$ *is the program order, which is a union of total orders on actions of each thread; the partial order* $\leq_{so} \subseteq A \times A$ *is the synchronisation order, which is a total order on all synchronisation actions in* A*;* $V :: \mathcal{A} \Rightarrow \mathcal{V}$ *is a* value-written *function that assigns a value to each write from* A*;* $W :: \mathcal{A} \Rightarrow \mathcal{A}$ *is a* write-seen *function that assigns a write to each read action from* A*, the* $W(r)$ *denotes the write seen by* r*, i.e. the value read by* r *is* $V(W(r))$*.*

Definition 3. *In an execution with synchronisation order* $\leq_{so}$*, an action* a synchronises-with *an action* b *(written* $a <_{sw} b$*) if* $a \leq_{so} b$ *and* a *and* b *satisfy one of the following conditions:*

- *a is an unlock on monitor m and b is a lock on monitor m,*
- *a is a volatile write to v and b is a volatile read from v.*

Definition 4. *The* happens-before *order of an execution is the transitive closure of the composition of its synchronises-with order and its program order, i.e.* $\leq_{hb} = (<_{sw} \cup \leq_{po})^+$*.*

To relate a (sequential) program to a sequence of actions performed by one thread we must define a notion of *sequential validity.* We consider single-thread programs as sets of sequences of pairs of an action kind and a value, which we call *traces.* A multi-thread program is a set of single-thread programs indexed by thread identifiers.

Definition 5. *The* action trace *of thread* θ *in an execution* $E = \langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$*, denoted* $\overline{\mathrm{Tr}}_E(\theta)$*, is the list of actions of thread* θ *in the order* $\leq_{po}$*. The* trace *of thread* θ *in* E*, written* $\mathrm{Tr}_E(\theta)$ *is the list of action kinds and corresponding values obtained from the action trace (i.e.,* $V(W(a))$ *if* a *is a read,* $V(a)$ *otherwise).*

By writing $t \leq t'$ we mean that t is a prefix of t', $set(t)$ is the set of elements of the list t, $\iota(t, a)$ is an index i such that $t_i = a$, or 0 if $a \notin set(t)$. For an action kind-value pair $p = \langle k, v\rangle$ we will use the notation $\pi_K(p)$ for the action kind k and $\pi_V(p)$ for the value v. We say that a sequence s of action kind-value pairs is *sequentially valid* with respect to a program P if $t \in P$. A sequentially valid

trace t is *finished* for P if there is no sequentially valid trace $t' > t$. A finished trace t is *properly finished* if the last action in t is of the thread finish kind. We say that a sequentially valid trace t *can perform step* k in P if there is v, such that $[t_0, \dots, t_{|t|-1}, \langle k, v\rangle]$ is sequentially valid in P.

To establish reasonable properties of concurrent programs we must assume reasonable properties of the underlying sequential language:

Definition 6. *We say that program P is* well-formed *if sequential validity of trace t in P implies:*

1. *any trace $t' \leq t$ is sequentially valid (prefix closedness),*
2. *if the last action of t is a read with value v, then the trace obtained from t by replacing the value in the last action by v' is also sequentially valid in P (final read value independence),*
3. $|t| > 0$ *implies* $\pi_1(t_0) = \mathrm{St}$ *(start action first),*
4. $\pi_1(t_i) = \mathrm{Fin}$ *implies* $i = |t| - 1$ *(finish action last).*
5. $\theta = \theta_{init}$ *implies* $\forall i.\ 1 \leq i < |t| - 1 \rightarrow \exists v.\ \pi_1(t_i) = \mathrm{Wr}(v) \vee \pi_1(t_i) = \mathrm{Wr}_v(v)$ *and* $\pi_1(t_{|t|-1}) = \mathrm{Fin}$ *(initialisation thread only contains writes).*

The well-formedness of programs should not be hard to establish for any reasonable sequential language.

The next definition places some sensible restriction on executions.

Definition 7. *We say that an* execution $\langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$ *is* well-formed *if*

1. *A is finite.*
2. *$\leq_{po}$ restricted on actions of one thread is a total order, $\leq_{po}$ does not relate actions of different threads.*
3. *$\leq_{so}$ is total on synchronisation actions of A.*
4. *$\leq_{so}$ is consistent with $\leq_{po}$.*
5. *W is properly typed: for every non-volatile read $r \in A$, $W(r)$ is a non-volatile write; for every volatile read $r \in A$, $W(r)$ is a volatile write.*
6. *Locking is proper: for all lock actions $l \in A$ on monitors m and all threads θ different from the thread of l, the number of locks in θ before l in $\leq_{so}$ is the same as the number of unlocks in θ before l in $\leq_{so}$.*
7. *Program order is intra-thread consistent: for each thread θ, the trace of θ in E is sequentially valid for P_θ.*
8. *$\leq_{so}$ is consistent with W: for every volatile read r of a variable v we have $W(r) \leq_{so} r$ and for any volatile write w to v, either $w \leq_{so} W(r)$ or $r \leq_{so} w$.*
9. *$\leq_{hb}$ is consistent with W: for all reads r of v it holds that $r \not\leq_{hb} W(r)$ and there is no intervening write w to v, i.e. if $W(r) \leq_{hb} w \leq_{hb} r$ and w writes to v then $W(r) = w$.*
10. *The initialisation thread θ_{init} finishes before any other thread starts, i.e.,* $\forall a, b \in A.\ K(a) = \mathrm{Fin} \wedge T(a) = \theta_{init} \wedge K(b) = \mathrm{St} \wedge T(b) \neq \theta_{init} \rightarrow a \leq_{so} b$.

The following definition of *legal execution* constitutes the core of the Java Memory Model. In our work, we use a weakened version of the memory model that we suggested in [5] and which permits more transformations than the original version. In Tbl. 1, we label this version by ‘JMM-Alt’.

Definition 8. *A well-formed execution* $\langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$ *with happens before order* $\leq_{hb}$ *is* legal *if there is a finite sequence of sets of actions* C_i *and well-formed executions* $E_i = \langle A_i, P, \leq_{po_i}, \leq_{so_i}, W_i, V_i\rangle$ *with happens-before* $\leq_{hb_i}$ *and synchronises-with* $<_{sw_i}$ *such that* $C_0 = \emptyset$, $C_{i-1} \subseteq C_i$ *for all* $i > 0$, $\bigcup C_i = A$, *and for each* $i > 0$ *the following rules are satisfied:*

1. $C_i \subseteq A_i$.
2. *For all reads* $r \in C_i$ *we have* $W(r) \leq_{hb} r \iff W(r) \leq_{hb_i} r$*, and* $r \not\leq_{hb_i} W(r)$*,*
3. $V_i|_{C_i} = V|_{C_i}$.
4. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$.
5. *For all reads* $r \in A_i - C_{i-1}$ *we have* $W_i(r) \leq_{hb_i} r$.
6. *For all reads* $r \in C_i - C_{i-1}$ *we have* $W(r) \in C_{i-1}$.
7. *If* $y \in C_i$ *is an external action and* $x \leq_{hb} y$ *then* $x \in C_i$.

The original definition of legality from [12, 19] differs in rules 2 and 6, and adds rule 8:

2. $\leq_{hb_i}|_{C_i} = \leq_{hb}|_{C_i}$.
6. For all reads $r \in C_i - C_{i-1}$ we have $W(r) \in C_{i-1}$ and $W_i(r) \in C_{i-1}$.
8. If $x <_{ssw_i} y \leq_{hb_i} z$ and $z \in C_i - C_{i-1}$, then $x <_{sw_j} y$ for all $j \geq i$, where $<_{ssw_i}$ is the transitive reduction of $\leq_{hb_i}$ without any $\leq_{po_i}$ edges, and the transitive reduction of $\leq_{hb_i}$ is a minimum relation such that its transitive closure is $\leq_{hb_i}$.

The reasons for weakening the rules are invalidity of reordering of independent statements, broken JMM causality tests 17–20 [22], and redundancy. For details, see [5, 6].

For reasoning about validity of reordering, we define observable behaviours of executions and programs. Intuitively, a program P has an observable behaviour B if B is a subset of external actions of some execution of P, and B is downward closed on happens-before order (restricted to external actions). As the JMM also captures non-termination as a behaviour in the definition of allowable behaviours.

Definition 9. *An execution* $\langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$ *with happens-before order* $\leq_{hb}$ *has a set of* observable behaviours O *if for all* $x \in O$ *we have* $y \leq_{hb} x$ *or* $y \leq_{so} x$ *implies* $y \in O$ *or* $T(y) = \theta_{init}$*. Moreover, there is no* $x \in O$ *such that* $T(x) = \theta_{init}$*.*

The allowable behaviours may contain a special external *hang* action if the execution does not terminate. We will use the notation $\mathrm{Ext}(A)$) for all external actions of set A, i.e., $\mathrm{Ext}(A) = \{a \mid K(a) = \mathrm{Ex}\}$.

Definition 10. *A finite set of actions* B *is an* allowable behaviour *of a program* P *if either*

– *There is a legal execution* E *of* P *with a set of observable behaviours* O *such that* $B = \mathrm{Ext}(O)$*, or* $B = \mathrm{Ext}(O) \cup \{hang\}$ *and* E *is hung.*

– *There is a set O such that $B = \text{Ext}(O) \cup \{hang\}$, and for all $n \geq |O|$ there must be a legal execution E of P with set of actions A, and a set of actions O' such that (i) O and O' are observable behaviours of E, (ii) $O \subseteq O' \subseteq A$, (iii) $n \leq |O'|$, and (iv) $\text{Ext}(O') = \text{Ext}(O)$.*

B Proof

We prove validity of irrelevant read elimination, elimination of redundant write before write, elimination of redundant read after write, and reordering of non-volatile memory accesses to different variables.

The plan of the proof is straightforward—for any behaviour B of a transformed program P' we need to show that the original program P had the same behaviour. Given a legal execution E' of P' with behaviour B we build a legal execution of P with (almost) the same behaviour. Using this construction, we will prove that transformations do not introduce new allowable behaviours (Def. 10), except hanging. The issues with hanging are tricky—its definition does not correspond with the committing semeantics.

Effects of Transformations on Traces. First, we define the notion of transformed program loosely enough so that redundant read/write elimination, irrelevant read elimination and reordering fit our definition. The idea is that for any trace of the transformed program there should be a trace of the original program that is just reordered with the redundant and irrelevant operations added.

To describe the effects of irrelevant read elimination formally we define *wildcard traces* that may contain star $*$ symbols instead of some values. For example, sequence $[\langle \text{Wr}(x), 2\rangle, \langle \text{Rd}(y), *\rangle, \langle \text{Rd}(x), 3\rangle]$ is a wildcard trace. If $\hat{t}$ is a wildcard trace, then $[\![\hat{t}]\!]$ stands for a family of all (normal) traces with the $*$ symbols replaced by some values.

Given a wildcard trace $\hat{t}$, we say its ith component $\hat{t}_i = \langle a, v\rangle$ is

– *irrelevant read* if a is a read and v is the wildcard symbol $*$,
– *redundant read* if a is a read of some x and the most recent access of x is a write of the same value, and there is no synchronisation or external action in between; formally, there must be $j < i$ such that $\hat{t}_j = \langle \text{Wr}(x), v\rangle$ and for each k such that $j < k < i$ it must be that $\hat{t}_k = \langle \text{Wr}(y), v'\rangle$ or $\hat{t}_k = \langle \text{Rd}(y), v'$ for some $y \neq x$ and some v',
– *redundant write* if a is a write to some x and one of these two cases holds: (i) the write is overwritten by a subsequent write to the same variable and there are no synchronisation or external actions, and no read of x in between, or (ii) $\hat{t}_i$ is the last access of the variable in the trace and there are no synchronisation or external actions in the rest of the trace.

Definition 11. *We will say that P' is a* transformed program *from P if for any trace t' in P' there is a wildcard trace $\hat{t}$ and a function $f :: \{0, \ldots, |t'| - 1\} \rightarrow \{0, \ldots, |\hat{t}| - 1\}$ such that:*

1. *all traces in* $[\![\hat{t}]\!]$ *are sequentially valid in* P.
2. *if* t' *is finished in* P' *then all traces in* $[\![\hat{t}]\!]$ *are finished in* P,
3. *function* f *is injective,*
4. *the action kind-value pair* t'_i *is equal to* $\hat{t}_{f(i)}$,
5. *for* $0 \leq i \leq j < |t'|$ *we have that* $f(i) \leq f(j)$ *if any of the following reordering restrictions holds:*
   (a) t'_i *or* t'_j *is a synchronisation or external actions, or*
   (b) t'_i *and* t'_j *are conflicting memory accesses, i.e., accesses to the same variables such that at least one is a write,*
6. *if there is an index* $j < |\hat{t}|$ *such that* $f(i) \neq j$ *for any* i, *then* $\hat{t}_j$ *must be a redundant read, a redundant write, or an irrelevant read.*

A multi-thread program P' is a transformed program of P if all single-thread programs of P' are transformed programs of single-thread programs of P with the same index. For space reasons we omit the link between the concrete syntax and the meaning in terms of traces. It is straightforward to establish by induction on derivation in operational semantics that if we obtain a program P' from a program P by a memory trace preserving transformation, or by an elimination of a redundant read after write, or by an elimination of a redundant write before write, or by an elimination of an irrelevant read, or by reordering of independent non-volatile memory accesses, then the set of traces of P' is a transformed program from the set of traces of P. The only non-trivial part is proving that reordering of independent non-volatile memory accesses on source level corresponds to a transformed program if the trace ends in between the reordered statements. In this case we can consider the missing part of the statement as being eliminated (either as a redundant write or an irrelevant read), and finish the proof.

Transforming Executions. Let P' be a program transformed from P, and $E' = \langle A', P', \leq'_{po}, \leq'_{so}, W', V' \rangle$ be a legal execution of P'. Our goal is to construct a legal execution E of P with the following properties with the same observable behaviours.

The main idea of the construction is to take the memory trace of each thread in E' and use Def. 11 to obtain a trace of P, and mapping of actions and program order of E' to actions and program order of our newly constructed execution. We will also need to restore actions that were eliminated by the transformation and construct the visibility functions W and V for the reconstructed actions.

Given an execution $E' = \langle A', P', \leq'_{po}, \leq'_{so}, W, V \rangle$ we construct *untransformed execution* E of P: for each thread $\theta \neq \theta_{init}$ let $\mathrm{Tr}_{E'}(\theta)$ be the trace of θ in E'. By the definition of transformed program (Def. 11), there must be a wildcard trace of P, let's denote it by $\hat{t}^\theta$ and corresponding transformation function f_θ.

For the initialisation thread θ_{init} we define

$$\hat{t}^{\theta_{init}} = [\langle \mathrm{St}, 0 \rangle] +\!\!+ \mathrm{Tr}_{E'}(\theta_{init})|_W +\!\!+ \mathrm{Init}_E +\!\!+ [\langle \mathrm{Fin}, 0 \rangle],$$

where $\mathrm{Tr}_{E'}(\theta_{init})|_W$ is the trace of the initialisation thread of E' restricted (filtered) to possibly volatile write actions, and Init_E is any sequence of initialisation writes for all variables that appear in any component of $\hat{t}_\theta$ ($\theta \neq \theta_{init}$), but

are not initialised in E'. We set $f_{\theta_{init}}(i) = i$ if $0 \leq i < |\mathrm{Tr}_{E'}(\theta_{init})| - 1$, and $f_{\theta_{init}}(|\mathrm{Tr}_{E'}(\theta_{init})| - 1) = |\hat{t}^{\theta_{init}}| - 1$.

From the traces $\hat{t}^\theta$ we build action traces t^θ of the same length. For $0 \leq i < |\hat{t}^\theta|$, we set the i-th component of t^θ to be

- $f_\theta^{-1}(i)$-th element of $\mathrm{Tr}_{E'}(\theta)$ if $f_\theta^{-1}(i)$ exists, or
- fresh action a such that $K(a) = \hat{t}^\theta_i$ and $T(a) = \theta$, if there is no j such that $i = f_\theta(j)$.

Using the action traces t^θ we define our *untransformed* execution $E = \langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$:

1. $A = \{t^\theta_i \mid 0 \leq i < |\hat{t}^\theta|\}$,
2. order $\leq_{po}$ is the order induced by the traces t^θ, i.e.
$$\leq_{po} = \{(a, b) \mid T(a) = T(b) \wedge \iota(t^{T(a)}, a) \leq \iota(t^{T(a)}, b)\}$$
3. order $\leq_{so}$ is equal to $\leq'_{so}$,
4. the write-seen function $W(a)$ is
   - $W'(a)$ if $a \in A'$,
   - most recent write[8] to x in $\leq_{hb}$ if $a \notin A'$ and a is a read from x,
   - a otherwise[9],
5. $V(a)$ is the corresponding value in the wildcard trace $\hat{t}^\theta$, i.e., $V(a) = \pi_V(\hat{t}^{T(a)}_{\iota(t^{T(a)},a)})$.

Lemma 1. *Let P' be a transformation of P, E' be a well-formed execution of P' with happens-before order $\leq'_{hb}$ and E be the untransformed execution of P with happens-before order $\leq_{hb}$. Let x and y be two actions from E' such that any of them is synchronisation action, or they are conflicting memory accesses[10], or $T(x) \neq T(y)$. Then $x \leq_{hb} y$ if and only if $x \leq'_{hb} y$ for all actions x and y from E'.*

Proof. Observe that by point 3 of Def. 11 we have $x \leq_{po} y$ iff $x \leq'_{po} y$ for all x and y from E' such that x or y is a synchronisation or external action, or x and y are conflicting memory accesses. By induction on the transitive closure definition of $\leq_{hb}$ we get that for any $z \leq_{hb} y$ either $z \leq_{po} y$ or there is a synchronisation action s such that $z \leq_{po} s \leq'_{hb} y$. With the observation above we conclude that $x \leq_{hb} y$ implies $x \leq_{hb} y$ if x is in E' and x or y is a synchronisation action, or x and y are conflicting memory accesses, or $T(a) \neq T(b)$. On the other hand, we prove that $z \leq'_{hb} x$ implies that either $z \leq'_{po} x$ or there is a synchronisation action s such that $z \leq'_{po} s \leq_{hb} x$ by induction on the definition of $\leq'_{hb}$. This implies the other direction of the equivalence.

Lemma 2. *Let P' be a transformation of P, E' be a well-formed execution of P' and E be the untransformed execution of P. Then E is a well-formed execution of P.*

[8] Note that the initialisation writes in thread θ_{init} happen-before any read action, so a most recent write always exists.

[9] This is just a technicality for actions a that are not reads.

[10] I.e. a read and a write to the same variable, or two writes to the same variable.

Proof. Properties 1–8 and 10 of well-formedness (Def. 7) are satisfied directly by our construction. We prove property 9, the hb-consistency, i.e., that for all reads in E, $r \not\leq_{hb} W(r)$ and there is no write w to the same variable as $W(r)$ such that $W(r) <_{hb} w \leq_{hb} r$. There are two cases: (i) for r being an irrelevant read or a redundant read the hb-consistency is satisfied trivially by construction, (ii) for $r \in E'$, we get the result from using hb-consistency of E' and Lemma 1.

Lemma 3. *Let P' be a transformation of P, E' be a legal execution of P' and E be the untransformed execution of P. Then E is a legal execution of P.*

Proof. Let $\{C_i\}_{i=0}^n$ be a sequence of committing sets and $\{E'_i\}_{i=0}^n$ the corresponding justifying executions. Let E_i be untransformed executions of E'_i. Let's define C_{n+1} as the set of actions of E and $E_{n+1} = E$. Then it is straightforward to show that the committing sequence $\{C_i\}_{i=0}^{n+1}$ with justifying executions $\{E_i\}_{i=0}^{n+1}$ satisfies the conditions (1), (3), (4) and (6) of Def. 8. To establish rules (2), (5) and (7) we use Lemma 1 and legality of E'.

In the following we will write $C_{\leq_{so},\leq_{po}}(X)$ to denote $\leq_{so}$ and $\leq_{po}$ downward closure of X without the initialisation actions, i.e.

$$C_{\leq_{so},\leq_{po}}(X) = \{y \mid \exists x \in X.\ (y \leq_{po\cup so} x \wedge T(y) \neq \theta_{init}\},$$

where $\leq_{po\cup so} = (\leq_{po} \cup \leq_{so})^+$. We will often use $C_E(X)$ for $C_{\leq_{so},\leq_{hb}}(X)$, where E has synchronisation order $\leq_{so}$ and happens-before order $\leq_{hb}$. The set $C_E(X)$ is an observable behaviour of execution E with actions A for any $X \subseteq A$.

Lemma 4. *Let P' be a transformation of P, E' be a legal execution of P' with observable behaviour O', and E be the untransformed execution of P. Then* $\mathrm{Ext}(C_E(O')) = \mathrm{Ext}(O')$.

Proof. The direction $\supseteq$ is trivial, because $\mathrm{Ext}(-)$ is monotone and $C_E(O') \supseteq O'$.

On the other hand, if an external action $x \leq_{po\cup so} y \in O'$, then for any $z \leq_{po\cup so} y$ there is s such that $z \leq_{po} s \leq'_{po\cup so} y$ by induction on the transitive definition of $\leq_{po\cup so}$. By Lemma 1 we get $x \leq'_{po\cup so} y$, thus $x \in O'$.

The main theorem says that transforming a program using Def. 11 cannot introduce any new behaviour, except hanging.

Theorem 1. *Let P' be a program transformed from P. If B is an allowable behaviour of P' then $B \setminus \{hang\}$ is an allowable behaviour of P.*

Proof. By Def. 10, there must an execution E' of P' with observable behaviour O' such that $B = \mathrm{Ext}(O')$ or $B = \mathrm{Ext}(O') \cup \{hang\}$.

Let's take an untransformation E of E' and let $O = C_E(O')$. Using Lemma 4, we have $\mathrm{Ext}(O) = \mathrm{Ext}(O')$. Since O is an observable behaviour of E and E is legal (Lemma 3), the set $B \setminus \{hang\} = \mathrm{Ext}(O)$ is an allowable behaviour of P.