Java Consistency: Non-Operational Characterizations for Java Memory Behavior *

Alex Gontmakher Assaf Schuster

Computer Science Department, Technion
{gsasha,assaf}@cs.technion.ac.il

Abstract

The Java Language Specification (JLS) [9] provides an operational definition for the consistency of shared variables. This definition remains unchanged in the JLS 2nd edition, currently under peer review [12]. The definition, which relies on a specific abstract machine as its underlying model, is very complicated. Several subsequent works have tried to simplify and formalize it. However, these revised definitions are also operational, and thus have failed to highlight the intuition behind the original specification.

In this work we provide a complete non-operational specification for Java and for the JVM, excluding synchronized operations. We provide a simpler definition, in which we clearly distinguish the consistency model that is promised to the programmer from that which should be implemented in the JVM. This distinction, which was implicit in the original definition, is crucial for building the JVM. We find that the programmer model is strictly weaker than that of the JVM, and precisely define their discrepancy. Moreover, our definition is independent of any specific (or even abstract) machine, and can thus be used to verify JVM implementations and compiler optimizations on any platform. Finally, we show the precise range of consistency relaxations obtainable for the Java memory model when a certain compiler optimization – called *prescient stores* in JLS – is applicable.

*Conditionally accepted to ACM Transactions on Computer Systems. Some of the results in this research were presented at IPPS, Orlando (April 1998) and at the Java workshop, Rhodes (June 1999)

Contents

1 Introduction

1.1 Background

One of the interesting and useful features of Java is its built-in concurrency support through multithreading. Multithreading can be exploited for many purposes [14]. Programs can use it to improve the responsiveness of applications, or to achieve high performance in computation-intensive applications. A single program running multiple threads is able to utilize many processors of the same computer, or even the processing power of several machines running on a network. The threads communicate by accessing shared variables whose consistency is preserved by propagating updates from one thread to another.

A Java system consists of a compiler, which translates the source code to bytecodes that are platform-independent and thus highly portable, and the *Java Virtual Machine* (JVM), the engine that executes the bytecodes on the target platform. To preserve portability, the JVM must be able to execute the bytecodes produced by a common compiler, without any modifications. The compiler must thus comply with the standard definition of Java, as given in the Java Language Specification (JLS) [9]. Chapter 17 of the JLS provides specifications for Java memory behavior.

Although the JLS provides a standard definition for the Java memory model, the description is given in terms of an implementation on some specific abstract memory system (AMS). The definition in JLS consists of a set of constraints binding the program code with the actual execution on the AMS (see Appendix A). All the constraints given are operational, i.e., they define how possible AMS executions for a given thread can be produced from its program code. Since Java allows for some weakening of shared memory consistency, JLS actually uses the AMS to describe how multiple copies of the same data maintain consistency. It thus implicitly defines the strength of the derived consistency on any other memory system as well.

The specification of Java Consistency on the AMS consists of two main layers: the upper layer, which defines the relation between the program code and the Java bytecodes, and the lower layer, which defines the relation between the bytecodes and the actual execution sequences. This layered structure is illustrated in Figure 1. The resulting consistency model, described by the relationship between the program code and its execution in the thread, as well as by the relations between executions on different threads, is rather complicated. Obviously, the definition given in JLS is inconvenient for both the Java programmer and the JVM implementor.

Several independent attempts have been made to restate the original definition in more formal yet still operational terms, using either *Abstract State Machine* methodology [10, 3], or *Structural Operational Semantics* [2, 4, 5]. However, these revised definitions, which are based, like the original, on a set of constraints on some abstract machine, fail to provide the intuition behind the original specification.

In a preliminary conference publication we showed that Java is coherent [7]. Pugh later used our proof to show that several shipping compilers violate the original Java specification [15]. The fact that he was able to do so exemplifies the usefulness of non-operational definitions.

1.2 This Work

In this work we provide a useful, non-operational model of Java memory. We call this model Java Consistency, or simply Java. Non-operational models specify how operations in one thread interact with operations in another. They attempt to keep the specification independent of a specific – or even an abstract – machine. As such, they are "cleaner," easier to implement, and simpler to understand than the operational models.

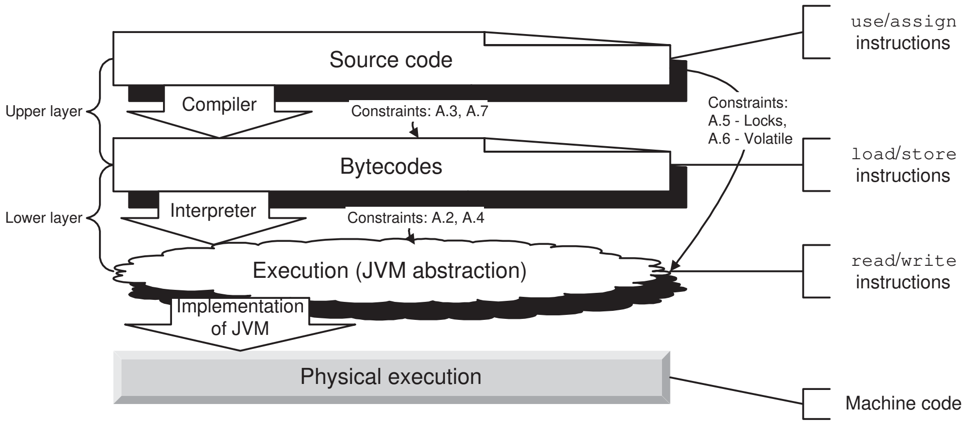

Figure 1: The layered structure of the Java programming paradigm.

The two-layered structure of Java results in different memory models for the programmer and the implementor. The programmer uses the specifications of the upper layer, whereas the implementor uses those of the lower layer. Figure 2 schematically depicts the programmer and implementor views. At different levels, different types of AMS operations are added to the execution (the process is explained in Section 1.3 below). Note that the original operational specification defines the relations between different levels of operations in the same thread, whereas the non-operational definition gives the relations between the views of different threads at the same level.

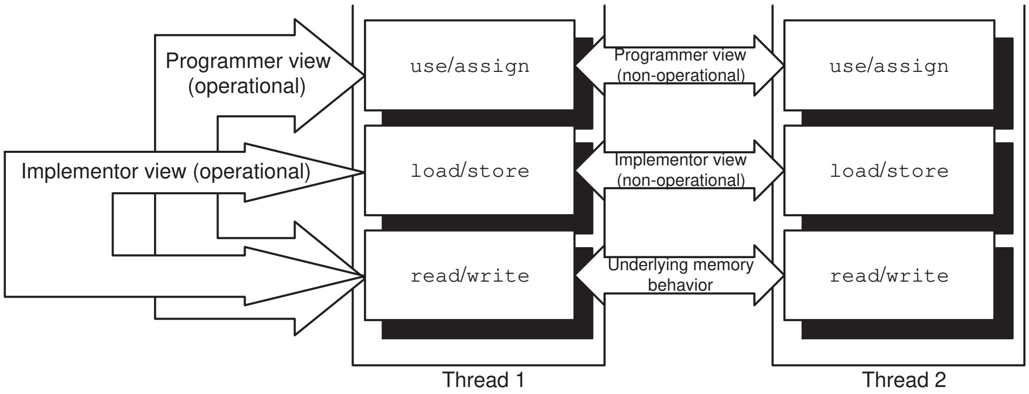

Figure 2: Programmer and implementor views in Java: operational vs. non-operational.

We compare Java to existing memory models for which implementation protocols and programming methodologies have already been devised. All the conventional consistency models that we refer to are presented in the literature in terms of non-operational definitions. The comparison may thus assist in the selection of those programs and algorithms which can be adapted to Java, even though they have been designed for other memory models.

We present non-operational definitions for both the programmer and the implementor views, and show that they match the original definitions exactly. In order to simplify the presentation, we provide two different definitions for the programmer view, one of which includes a compiler optimization called *prescient stores*. The new definitions are relatively simple and can be used for several purposes. They can be used, for example, to guide programmers in how to use the shared memory efficiently, as well as to verify compiler correctness.

Our non-operational definitions led to a very interesting result: We found that the programmer model is strictly weaker than the implementor model. This result is almost impossible to derive using the original, operational specification.

We remark here that our approach applies whether the JVM is implemented as an interpreter, which executes the instructions one-by-one, or as an optimizing *Just In Time compiler* (JIT), which compiles the bytecodes to native machine code. We are interested, in either case, in the exhibited memory behavior. Thus, the internals of the implementation, whether as an interpreter or a JIT, have no effect on the results presented in this paper.

The paper is organized as follows. In the rest of this section we give notations and definitions, as well as explain the Java memory model. In Section 2 we compare Java with other well-known models. In Section 3 we give a non-operational definition for the JVM memory model. In Section 4 we provide a non-operational definition for the Java memory model, excluding prescient stores. Section 5 completes the previous two sections, giving a non-operational model for the programmer view that includes prescient stores. Section 6 discusses issues related to strong operations such as volatiles and locks. We conclude in Section 7 with some remarks on future work. For completeness, we also provide in Appendix A the original list of constraints from [9] for the implementation of Java on top of the AMS.

1.3 The Java Virtual Machine (JVM)

As mentioned above, the Java Language Specification, (see Appendix A and Chapter 17 in [9]) is defined by means of specifying an abstract memory system, AMS, a set of operations, and the constraints that are imposed upon them. The machine consists of a *main memory agent* (for brevity, *main memory*) and *threads*, each of which has its own *local memory*. *Variables* are stored in the main memory, and their values are available for computation by a thread only after they are explicitly brought to its local memory. In some situations, they are also written back from the thread's local memory to the main memory.

Each one of the threads is executed by a *thread engine*, which can be implemented as a separate CPU, a thread provided by the operating system, or some other mechanism.

The threads and the main memory issue a series of operations which can be categorized into four classes:

Operations local to the thread engine. The operations are `use` and `assign`. `use` puts the local copy of a variable into the engine for some calculation, and `assign` puts the result of the calculation into the local copy of the result variable. There are no explicit `use` and `assign` instructions in the bytecode, but the operations specified by the bytecodes implicitly use several such instructions during the execution. For example, `x=y+z` in the Java source code implies bytecode instruction `add`, which implicitly involves `use y`, `use z`, and finally, `assign x`.

Operations between the thread and the main memory. There are two such operations: `load` and `store`. `load` transfers the value of a given variable from the main memory to the local copy, and `store` transfers it back. In our interpretation (see below), these operations are represented by explicit instructions in the bytecode. A `store` may be *prescient*, in which case it writes a result of some future `assign`.

Operations performed by the main memory. They are: `read` and `write`. These operations are initiated by the main memory as a result of `load`s and `store`s. `read` actually reads the value which is delivered later by the `load` instruction, and `write` stores the value supplied by the `store` instruction.

Locking operations. They are: `lock` and `unlock`. These operations are responsible for synchronization of memory and program control flow. There are explicit `lock` and `unlock` operations in the bytecode, and they are performed by the thread in tight interaction with the main memory.

The JLS includes a set of constraints which define relationships between all the instructions above. All the constraints are quoted in Appendix A and will be referred to by their section number and index within the section. For example, *Constraint A.2.1* refers to Constraint 1 in Section A.2 of the appendix.

1.3.1 Mapping the AMS Operations to the Bytecode

The JLS defines `use`/`assign` instructions as instructions that correspond exactly to the program source code. This follows from Constraint A.3.1. The `load`/`store` instructions are then treated as instructions that can be manipulated by compiler optimization. (The following remark in the JLS definition of prescient stores makes it clear that this is indeed the original intention: "The purpose of [the prescient stores] relaxation is to allow optimizing Java compilers to perform certain kinds of code rearrangement that preserve the semantics of properly synchronized programs ...".) The `read`/`write` operations obviously belong to the runtime environment and should not be reflected in the bytecode.

Since the bytecode has explicit memory access operations (`getfield`, `putfield`, `getstatic`, `putstatic`, `getarray`, `putarray`), our interpretation is that the `load`/`store` instructions of the AMS directly correspond to these operations: `load` — to `getfield`, `getstatic` or `getarray` and `store` — to `putfield`, `putstatic` or `putarray`.

Two important remarks about the above interpretation are due. First, this interpretation enables us to define a consistency model which should be provided by the JVM, named the implementor view. Without it, the consistency model which should be implemented by the JVM is left unspecified. Considering that the bytecodes are intended to be portable across JVM implementations, the lack of a memory model is a cause of great confusion.

Second, our interpretation applies only to the results in section 3. The definitions in other sections do not use notions of concrete Java instructions in any way, and correspond directly to the AMS.

1.3.2 Programmer and Implementor View

According to the interpretation above, the AMS operations can be seen as executing in different layers. During the compilation, the `use` and `assign` operations follow immediately from the source code of the Java program. The `load` and `store` instructions are inserted, according to the constraints, between `use`/`assign`s and `load`/`store`s. These are the *upper layer* constraints.

When the program is executed, the `load` and `store` instructions initiate the execution of `read` and `write` instructions, which are performed in the main memory. The set of constraints that govern the relations between the `load`s/`store`s and the `read`/`write`s constitutes the *lower layer*.

Since the lower layer defines how the AMS should behave at runtime, it determines the consistency model that should be provided by the implementor of the JVM. Therefore, we call the lower layer model the *implementor view* of the Java programming language.

The programmer explicitly influences only the `use` and `assign` instructions, and is therefore interested in how these instructions interact. Since these interactions are determined by `load/store` and `read/write` instructions, the *programmer view* of the Java programming language is defined by the combination of the lower and the upper layers of the constraints.

1.4 Memory Consistency Models

The JLS definition of the consistency model ignores the underlying computation, including the flow of control and the calculation of the values. Likewise, our definition is defined in terms of memory access operations only.

A *program* is a collection of *local programs* for each one of the participating threads. A local program is a sequence of READ and WRITE operations, in the order in which they are to be issued by the thread.

Program activation results in an *execution*. We determine for each operation in the execution the value it supplies or yields. For our purposes, the execution is concerned only with the way the values are transferred from WRITEs to READs. We assume that a READ can always identify its WRITE. In other words, no two WRITE operations supply the same value. Therefore, an execution determines the mapping from the WRITEs to the READs, where the value transfer is implicitly defined by the values of the operations.

A *local history* of a thread p comprises all the operations performed by p. The term "local history" is applicable to both executions and to *schedules*, which are defined in Subsection 1.4.1 below. For each thread p, its *program order*, *po*, is the order of operations in its local history.

For a given execution H and a thread p, H_{p+w} denotes the partial execution consisting of all the operations of p and all the WRITE operations of the other threads. $H|x$ denotes the partial execution of H consisting only of operations on the variable x.

A *consistency model* defines for each program a set of executions that are *valid* for this program.

For two consistency models, C_1 and C_2, C_1 is *not weaker* than C_2 if for any program P, for any execution H of P, if H is valid under C_1 then H is valid under C_2.

Two consistency models are *equivalent* if neither one is weaker than the other.

If C_1 is neither weaker than C_2 nor equivalent to it, then it is *stronger* than C_2.

C_1 and C_2 are *incomparable* if there exists an execution H_1 that is valid under C_1 and invalid under C_2, and another execution H_2 (not necessarily of the same program) that is valid under C_2 and invalid under C_1.

1.4.1 JLS Operational Models

For the Java programming language, we identify three operational consistency models. These are called Java_{VM}, Java_{PS} and Java.

A *schedule* in each of these models consists of `use/assign`, `load/store`, `read/write` operations as defined in the JLS, and a timing for their execution. The schedule should comply with constraints from JLS, which we also list in Appendix A for the convenience of the reader. Just as the execution defines a transfer of values from WRITEs to READs, a schedule defines the transfer of values from `assign`s to `use`s.

When comparing a schedule with an execution, we require that there exist a one-to-one correspondence between the READ X,VS in the execution and the `use x,vs` in the schedule, and between WRITE X,VS and `assign x,vs`. Since a schedule contains additional `load/store`, `read/write` operations, there may be many schedules which correspond one-to-one with the same execution. In the opposite direction, however, there is only a single execution which corresponds one-to-one with a given schedule.

Note that the transfer of values defined by the schedule is the same as that defined by the corresponding execution. Thus, when a schedule corresponds one-to-one with a given execution, we say that it is *a schedule of the given execution.*

For the implementor view and Java$_{\mathrm{VM}}$, the schedule will consist of only `load/store/read/write` operations, and therefore will define the transfer of values from `store`s to `load`s.

- **Java$_{\mathrm{PS}}$ Consistency:** An execution H of a program P belongs to Java$_{\mathrm{PS}}$ consistency if there exists a schedule T of H which complies with all the JLS constraints regarding regular variables, excluding prescient stores (i.e., the schedule complies with constraints from A.2, A.3 and A.4).

  This model describes the programmer view, but excludes the prescient stores.

- **Java Consistency:** An execution H of a program P belongs to Java consistency if there exists a schedule T of H which complies with all the Java constraints, including those for prescient stores (i.e., constraints from A.2, A.3, A.4 and A.7).

  This model describes the programmer view with prescient stores.

- **Java$_{\mathrm{VM}}$ Consistency:** An execution H of a program P belongs to Java$_{\mathrm{VM}}$ consistency if there exists a schedule T of H which complies with the constraints from A.2 and A.4. The constraints from A.3 and A.7 are not relevant to Java$_{\mathrm{VM}}$.

  This model describes the implementor view.

Since the programmer should always assume that the compiler can apply all optimizations, only Java consistency should be used. However, the simpler Java$_{\mathrm{PS}}$ definition aids in building the intuition on which the Java consistency model relies. Furthermore, the two definitions may be viewed as an "upper-bound" (Java$_{\mathrm{PS}}$, strongest) and a "lower-bound" (Java, weakest) on the range of consistency models that become available when using prescient stores for none, some, or all of the `store`s.

1.4.2 Non-Operational Consistency Models

Our non-operational consistency models are defined using the notion of a *legal serialization.*

A *serialization* S of a program P is a sequence containing all the operations in P. S is *legal* if each READ X operation yields the result of the most recent WRITE X operation preceding the READ in S.

Let C be a set of order relations. If o_1 should precede o_2 according to C, we denote this by $o_1 \xrightarrow{C} o_2$.

The *legal serialization consistency for C*, denoted by $LS(C)$, is the following:

$LS(C)$: Execution H is $LS(C)$ if there is a *legal serialization* S of H such that $o_1 \xrightarrow{C} o_2 \Rightarrow o_1 \xrightarrow{S} o_2$.

In this paper we define several consistency models derived from $LS(C)$. The non-operational model equivalent to Java$_{\mathrm{VM}}$ is called Java$^N_{\mathrm{VM}}$. The model equivalent to Java is called Java^N, and the model equivalent to Java$_{\mathrm{PS}}$ is called Java$^N_{\mathrm{PS}}$.

1.5 Notation

To denote an operation/instruction that accesses variable x, we write it as READ X or WRITE X. If we also want to denote that it supplies (or returns) value v, we write it as READ X,V or WRITE X,V.

READ$^{\swarrow}$ denotes an operation that yields a value written at another thread. Similarly, WRITE$_{\searrow}$ denotes an operation providing a value read by another thread. READ$^{\downarrow}$ denotes an operation that yields a value written by the same thread, and WRITE$_{\downarrow}$ denotes an operation that is seen in the same thread only. A plain READ operation may or may not see a value written at another thread. Similarly, WRITE denotes an operation that may or may not be seen by another thread.

The motivation for the above distinction is that READ$^{\swarrow}$ and WRITE$_{\searrow}$ denote operations that must have the corresponding `load/store` instruction in the Java schedule, whereas READ$^{\downarrow}$ and WRITE$_{\downarrow}$ denote operations that may or may not have a `load/store`.

According to JLS, a WRITE operation is represented in the schedule by a chain of `assign`, `store` and `write` instructions. We say that the instructions that represent an operation o *correspond* to o. We denote the operations corresponding to o by `assign(`o`)`, `store(`o`)`, `write(`o`)`. Similarly, the instructions corresponding to a READ operation o are denoted by `read(`o`)`, `load(`o`)` and `use(`o`)`. In some cases we use one of the operations of the schedule to represent other operations in its chain. For example, for an `assign x,v` operation o, there may exist `store(`o`)` and `write(`o`)`.

The instructions in the execution examples are indexed by the thread name and the instruction index in the thread's local program. Thus, instruction i in the program of thread j is denoted by $j.i$. When presenting execution examples we always assume that the variables are initialized to 0.

2 Comparison with Conventional Models

In this section we compare the programmer view of Java to other models that appear in the literature. The section is organized as follows. Section 2.1 shows that Java$_{\mathrm{PS}}$ and Java are coherent. Section 2.2 shows that Java$_{\mathrm{PS}}$ and Java are incomparable with PRAM consistency. Section 2.3 shows that Java$_{\mathrm{PS}}$ and Java are incomparable with both versions of processor consistency.

All claims in Sections 2.2 and 2.3 are made for Java only, but they are intended for Java$_{\mathrm{PS}}$ as well; all examples apply to both models.

2.1 Coherence

The definition of Coherence, as in [1], is:

> **Coherence:** An execution H is said to be *coherent* if for each variable x there is a legal serialization S_x of $H|x$ such that if o_1 and o_2 are two operations in $H|x$, and $o_1 \xrightarrow{po} o_2$, then $o_1 \xrightarrow{S_x} o_2$. A consistency model is said to be *coherent* if every execution under it is coherent. The corresponding memory model is called *Coherence*.

Theorem 2.1 *Java$_{\mathrm{PS}}$ consistency and Java consistency are stronger than Coherence.*

In order to prove the theorem we must demonstrate, first, that Java$_{\mathrm{PS}}$ and Java are not weaker than Coherence. This means that any execution that is valid for Java$_{\mathrm{PS}}$ or Java is also valid for Coherence, and second, that Java$_{\mathrm{PS}}$ and Java are not equivalent to Coherence. That is to say, there exists an execution that is valid for Coherence but is not valid for Java$_{\mathrm{PS}}$ or Java.

We begin by presenting the proof only for Java$_{\mathrm{PS}}$. Coherence for both Java$_{\mathrm{PS}}$ and Java can be derived from our exact non-operational definitions (see Sections 4 and 5).

Claim 2.1 *Java*$_{\mathrm{PS}}$ *is coherent.*

Proof: We need to show that for every program P, for every execution H of P which is valid under Java$_{\mathrm{PS}}$, and for each variable x, there is a global serialization S of $H|x$ which is consistent with the views of all the threads. If H is valid under Java$_{\mathrm{PS}}$, there exists a schedule T of H consistent with all of the Java$_{\mathrm{PS}}$ constraints. Consider the operations on some variable x in T.

Dividing the operations into blocks

For each thread p, we divide the sequence of `use`/`assign`/`load`/`store` operations performed for p in T into blocks, and then arrange the blocks in a global order consistent with the view of all the threads.

The division is specified by the following regular expression. Note that the last block from the sequence may terminate prematurely.

Order = (*load-block* | *store-block*)*
load-block = `load` (`use`)*
store-block = `assign` (`use` | `assign`)* `store` (`use`)*

By Constraint A.3.4, the sequence of operations on x always begins with an `assign` or a `load`. This means that a *load-block* or a *store-block* can start.

Next, we need to show that a *load-block* or *store-block* can always be matched in the sequence of operations, and that another *load-block* or *store-block* can begin after it. Consider some block B. If B is a *load-block*, it begins with a `load`. Any number of `use`s following the `load` will match B. If a `load` instruction appears in the instruction sequence, it will begin a new *load-block*. An `assign` instruction will begin a new *store-block*. A `store` instruction would neither match the block nor start a new one; however, it cannot appear because of Constraint A.3.3. Therefore, B can always be matched and a new block can be started after it.

If B is a *store-block*, it begins with an `assign`. Any number of `use`/`assign` instructions following the `assign` will match the block. A `store` will continue the matching, and no `load` can appear between the first `assign` and the `store`, by Constraint A.3.2. After the `store`, any number of `use`s fits in the block, and an `assign` or a `load` will begin a new one. No `store` can appear, by Constraint A.3.3. Therefore, B can always be matched and a new block can be started after it.

Constructing the serialization S

Each block contains only one `load` or `store` instruction. By constraints A.4.1 and A.4.2, this implies that in H there is exactly one `read`/`write` operation corresponding to each block. By Constraint A.2.2, these operations are totally ordered in T. We arrange the blocks in a serialization S' in the order of their corresponding main memory accesses.

By Constraint A.3.1, the order of `use`/`assign` operations performed on variable x by a thread in T is the same as the order of READ/WRITE operations in the local history of the same thread in H. By Constraint A.4.3, for each thread p, the order of main memory operations accessing x performed in T on behalf of p is the same as the order of `load`/`store` operations on x in the local history of p. Therefore, for each thread p, the blocks are arranged in the same order as in the local history of p in T, and the order of `use`/`assign` operations executed by p in S' is the same as the order of READ/WRITE operations executed by p in H.

Let S be the order of `use`/`assign` operations from S', converted to READ/WRITEs. Since S has the same orders of operations as S', the order of operations of each thread p is the same as the order of those operations in H. It remains to show that S is a legal serialization.

S is a legal serialization

Since the order of READ/WRITE operations in S is the same as the order of `use`s/`assign`s in S', it is enough to show that each `use` operation in S' yields the result of the most recent `assign` operation preceding it.

Given a `use` operation u in the local history of some thread p in T, consider the most recent `assign` or `load` operation o that appears before u. It is clear from the definition of the regular expression that o and u belong to the same block. Therefore, no other operation can intervene between o and u in S'. Let us examine the possibilities:

- o is an `assign`. Since o is the most recent operation preceding u in T, u yields in T the result supplied by o (Constraint A.2.1). Therefore, u does not violate legal serialization in S'.
- o is a `load`. In T, the value brought by `load` was read by a corresponding `read` operation r (Constraint A.4.1), which, in turn, sees the result written by a `write` operation w. w corresponds to a `store` instruction s by some thread (Constraint A.4.2). By Constraint A.2.2, w is the most recent operation accessing x that precedes t in the order of memory accesses of T. Therefore, in S', the *store-block* of w appears immediately before the *load-block* of r.

  By Constraint A.2.1, s sees the result of the most recent preceding `assign` operation a. From the definition of the blocks, a belongs to the same block as s. Since a *store-block* contains no `assign` instructions after the `store`, a is the latest operation preceding u in S'.

  We conclude that u does not violate legal serialization in S'.

■

Claim 2.2 *Java$_{\text{PS}}$ and Java are strictly stronger than Coherence.*

Proof: We first show an execution which is invalid for Java$_{\text{PS}}$ and valid for Coherence. Then we modify the execution in such a way that it will also be invalid for Java. The idea for this modification is taken from Pugh [15].

	Thread 1	Thread 2
1	READ X,1	READ Y,1
2	WRITE Y,1	WRITE X,1

Figure 3: Execution valid for Coherence and invalid for Java$_{\text{PS}}$.

Consider the execution in Figure 3. In order to show that it is coherent, set the order of accesses in the serialization for variable x as 2.2, 1.1, and the order of accesses to y as 1.2, 2.1. Clearly, these are legal serializations which preserve program order.

For Java$_{\text{PS}}$, however, this execution is impossible. In order for 1.1 to see the result written by 2.2, the schedule must contain the instructions `load(1.1)` and `store(2.2)`. Similarly, it must contain `load(2.1)` and `store(1.2)`. By Constraint A.2.1, the `load(1.1)` operation by thread 1 should precede the `use(1.1)` in order for the `use` to see the result of the `load`. Similarly, the operation `store(1.2)` should follow `assign(1.2)`. Because of the program order (Constraint

A.2.1), `use(1.1)` must precede `assign(1.2)`, and thus `load(1.1)` must precede `store(1.2)`. Now, because of constraints A.4.1 and A.4.2, `read(1.1)` should precede `load(1.1)`, and `write(1.2)` should follow `store(1.2)`. By transitivity, `read(1.1)` must precede `write(1.2)` in any schedule. For the same reasons, `read(2.1)` must precede `write(2.2)`.

By Constraint A.2.2, the required results will be produced if memory accesses are scheduled as follows: `write x,1` before `read x,1`, and `write y,1` before `read y,1`. However, together with the constraints discussed in the previous paragraph, this schedule induces a dependency cycle in the schedule (`write(1.1)` → `write(1.2)` → `write(2.1)` → `write(2.2)` → `write(1.1)`), in which `write(1.1)` follows itself. This is prohibited by Constraint A.2.4. Thus, this execution is invalid under Java_{PS}.

This execution is, however, valid under Java. If prescient stores are allowed, `store(1.2)` can be scheduled before `load(1.1)`. Similarly, `store(2.2)` can be scheduled before `load(2.1)`. The following order of memory operations will then produce a timing which does not contradict any Java rules: `write(1.2)` → `write(2.2)` → `read(1.1)` → `read(2.1)`.

Pugh [15] showed that the example can be extended by adding READ operations to the execution. The resulting execution is still coherent, but it is invalid under Java. This extended execution is presented in Figure 4. It contains the original R X,1, W Y,1, R Y,1 and W X,1 operations, as well as additional operations. The additional operations ensure that even if the `store` performed for W Y,1 (or W X,1) is prescient, the constraints will forbid it from switching places with the `load` of R X,1 (or R Y,1, respectively).

Let us examine the example in greater detail. Since READ X,1 reads the value of WRITE X,1 executed by another thread, a `load x` operation must precede the corresponding `use x,1`. The operations READ Y,0 and READ Y,2 see different values. Therefore, there should exist a `load y,2` operation performed after `use y,0` and before `use y,2`. Program order requires that `use x,1` precede `use y,0`. Therefore, `load x,1` must precede `load y,0`. Because of Constraint A.7.3, even if `store y,1` is prescient, it cannot be placed before `load y,2`. Applying all the ordering constraints, we get that the `load x,1` must be executed prior to the `store y,1`, exactly as in the original example. Similarly, the `load y,1` must precede the `store x,1`. Thus, this execution is forbidden for Java for the same reason that the execution in Figure 3 is forbidden under Java_{PS}. ■

	Thread 1	Thread 2	Thread 3
1	READ X, 1	READ Y, 1	WRITE X,2
2	READ Y, 0	READ X, 0	WRITE Y,2
3	READ Y, 2	READ X, 2	
4	WRITE Y, 1	WRITE X, 1	

Figure 4: Execution valid for Coherence and invalid for Java.

2.2 Comparison with PRAM Consistency

The definition of PRAM consistency is as follows [1]:

PRAM: An execution H is PRAM if for each thread p there is a legal serialization S_p of H_{p+w}, such that if o_1 and o_2 are two operations in H_{p+w}, and $o_1 \xrightarrow{po} o_2$, then $o_1 \xrightarrow{S_p} o_2$.

Theorem 2.2 *Java is incomparable (neither stronger nor weaker) with PRAM.*

To prove the theorem we show two executions: one that is valid under Java but is invalid under PRAM, and another that is valid for PRAM but not for Java.

Proof: Direction 1 Figure 5 shows an execution which is valid for Java but invalid for PRAM. Intuitively, the difference between Java and PRAM in this example can be explained by the fact that the Java constraints impose very little connection between operations on different variables. In particular, they do not require that the program order be preserved, whereas PRAM requires that program order be preserved for all operations in the thread.

Thread 1	*Thread 2*
WRITE X,1	READ Y,1
WRITE X,2	READ X,1
WRITE Y,1	READ X,2

Figure 5: An execution valid for Java but invalid for PRAM.

According to the program order of thread 1, the operation WRITE X,2 precedes WRITE Y,1. However, thread 2 sees the change of x to 2 after it sees the result of the WRITE to y; hence, it sees WRITE X,2 after WRITE Y,1. Therefore, the execution is invalid under PRAM.

Now let us show how this execution might take place in Java. As stated before, the operation READ X,1 denotes a `use` in the Java execution. The `store`s performed by thread 1 to x and to y are independent, enabling the thread to perform `store y` before it performs `store x`. Thus, the instructions actually executed by thread 1 (left to right) could be as follows:

a x,1	s x,1	a x,2	a y,1	s y,1	s x,2

A possible local history of thread 2 could be:

l y,1	u y,1	l x,1	u x,1	l x,2	u x,2

Now, coupled with the main memory agent, this can produce the schedule shown in Figure 6. This schedule conforms with all the Java$_{PS}$ rules. Therefore, the execution in Figure 5 is valid under Java$_{PS}$.

Note that this schedule is valid for both Java$_{PS}$ and Java, since none of the `store` operations are prescient in the constructed schedule.

Direction 2 Figure 7 shows an example for an execution which is valid for PRAM but invalid for Java. The intuitive reason for the difference is that Java requires Coherence while PRAM does not.

Threads 3 and 4 see an order consistent with the program orders of both writing threads 1 and 2. Because PRAM does not require correlation between the views of threads 3 and 4, there is no contradiction, and thus the execution is valid under PRAM. However, the conflict in the views of threads 3 and 4 implies that there is no legal serialization which preserves program order for both. Therefore, the execution is not coherent and (as was shown in Section 2.1) is thus invalid in Java.

In this example, threads 1 and 2 have only one instruction in the program order. The prescient stores optimization is irrelevant, and the execution is valid under both Java and Java$_{PS}$. ∎

Thread 1	*Thread 2*	*Main Memory*
1. `a x,1`		
2. `s x,1`		
3. `a x,2`		`w x,1` ; 1.2
4. `a y,1`		
5. `s y,1`		
6. `s x,2`		`w y,1` ; 1.5
		`r y,1` ; 2.1
	1. `l y,1`	`r x,1` ; 2.3
	2. `u y,1`	`w x,2` ; 1.6
	3. `l x,1`	`r x,2` ; 2.5
	4. `u x,1`	
	5. `l x,2`	
	6. `u x,2`	

Figure 6: A possible Java schedule of the execution from Figure 5. The comment after each operation in the main memory column tells which `load` or `store` instruction initiated this operation; for instance, "; 1.2" after the operation `w x,1` indicates that this operation was invoked by instruction 2 in thread 1 (`s x,1`).

Thread 1	*Thread 2*	*Thread 3*	*Thread 4*
W X,1	W X,2	R X,1	R X,2
		R X,2	R X,1

Figure 7: An execution valid for PRAM and invalid for Java. This execution is taken from [17].

2.3 Comparison with Processor Consistency (PC)

[1] defines two variants of PC: PCD and PCG, of which the corresponding non-operational definitions are taken from [6] and [8] respectively. Both variants are known to be stronger than PRAM. Since we have seen an execution belonging to Java and not to PRAM, this execution is also invalid under either PCD or PCG. The following theorems answer the question of whether Java is strictly weaker than either of them.

The definition of PCG, according to [1], is the following:

PCG: For each thread p there is a legal serialization S_p of H_{p+w} such that:

1. If o_1 and o_2 are two operations in H_{p+w}, and $o_1 \xrightarrow{po} o_2$, then $o_1 \xrightarrow{S_p} o_2$.
2. For each variable x, if there are two WRITE X operations, then they appear in the same order in the serializations of all the threads.

Theorem 2.3 *Java is incomparable with PCG.*

Proof: To prove the theorem we must show an execution which is valid for PCG but invalid for Java. We will use the same example given for Coherence in Figure 4 above. We have already shown that this execution is invalid for Java. Now we will show that it is valid for PCG.

In order to satisfy the first condition, we use the serialization W X,2, W X,1, R X,1, R Y,0, W Y,2, R Y,2, W Y,1 for thread 1 and the serialization W Y,2, W Y,1, R Y,1, R X,0, W X,2, R X,2, W X,1 for thread 2.

It is easy to see that each serialization is legal and complies with the program order of its thread. In both serializations, W X,2 precedes W X,1 and W Y,2 precedes W Y,1, so the second condition is satisfied as well. ∎

The definition of PCD consistency in [1] is as follows:

PCD: We first define several notions:

- *Weak order relation* between two operations of the same thread: We say that o_1 *weakly precedes* o_2 (and denote this as $o_1 \xrightarrow{wpo} o_2$), if $o_1 \xrightarrow{po} o_2$, and either
  1. o_1 and o_2 are operations on the same variable, or
  2. o_1 and o_2 are both reads or both writes, or
  3. o_1 is a read and o_2 is a write, or
  4. (transitivity) there is another operation o', such that $o_1 \xrightarrow{wpo} o' \xrightarrow{wpo} o_2$.
- *Weak writes-before.* $o_1 \xrightarrow{wwb} o_2$ iff $o_1 =$ W X,V, $o_2 =$ R Y,U, and there is another operation $o' =$ W Y,U such that $o_1 \xrightarrow{wpo} o'$.
- *Weak reads-before.* $o_1 \xrightarrow{wrb} o_2$ iff $o_1 =$ R X,V, $o_2 =$ W Y,U, and there is another operation $o' =$ W X,V', such that $o_1 \xrightarrow{S_x} o'$ and $o' \xrightarrow{wpo} o_2$, where $\xrightarrow{S_x}$ is some legal serialization for all operations on the variable x.
- *Semi-causality.* Semi-causality (denoted $\xrightarrow{s}$) is the transitive closure of the weak order, the weak writes-before, and the weak reads-before relations.

Now, an execution H is PCD if:

1. H is coherent, i.e., for every variable x there exists a legal serialization S_x of $H|x$ that is consistent with all the threads' views.
2. For each thread p, there is a legal serialization S_p of H_{p+w} such that
   (a) if o_1 and o_2 are two operations in H_{p+w}, and $o_1 \xrightarrow{s} o_2$, then $o_1 \xrightarrow{S_p} o_2$;
   (b) for each variable x, if there are two write operations o_1 and o_2 to x, then these operations appear in the same order in S_x and S_p.

Theorem 2.4 *Java is incomparable with PCD.*

Proof: To prove this theorem we use again the execution from Figure 4. We have seen that it is not valid under Java. In order to show that it is valid under PCD we check the execution against the stages of the PCD definition, verifying that none of them is violated.
For Condition 1 we check that the execution is coherent. Since Coherence was shown for Java by Claim 2.2, the condition is satisfied.
For Condition 2 we calculate the semi-causality relation for this execution, according to the following steps:

1. *Program order relation.* For thread 1, we get that 1.1 $\xrightarrow{po}$ 1.2, 1.2 $\xrightarrow{po}$ 1.3, 1.3 $\xrightarrow{po}$ 1.4, and similarly for thread 2.
2. *Weak order relation.* 1.1, 1.2 and 1.3 are all reads, 1.1 $\xrightarrow{wpo}$ 1.2 and 1.2 $\xrightarrow{wpo}$ 1.3. 1.3 and 1.4 are read before write. Thus 1.3 $\xrightarrow{wpo}$ 1.4. We get that the weak order relation is equivalent to the program order for thread 1, and the same applies to thread 2.

3. *Weak writes-before and weak reads-before relation.* We can see from the definitions that no two operations have these relations.
4. Finally, we obtain the *semi-causality relation.* It is the transitive closure of the relations above, and thus is equal to the program order.

Now we can reuse the serializations from the proof of Theorem 2.3. The serializations maintain the program order of their corresponding threads. Thus they do not violate the semi-causality relation. The Coherence order for variable x is R X,0; W X,2; R X,2; W X,1; R X,1. It is easy to see that this order is consistent with the serializations for thread 1 and thread 2. The situation is symmetric for y. ∎

3 Specification for the JVM Memory Behavior

In this section we explore the Java_{VM} consistency model defined in 1.4.1. This section is organized as follows. In Sections 3.1 and 3.2 we compare $\text{Java}_{\text{VM}}^{N}$ with other standard consistency models. In Section 3.3 we prove that $\text{Java}_{\text{VM}}^{N}$ is equivalent to Java_{VM}.

For the non-operational definition, we first introduce the *Causality relation*, denoted CR:

> **Causality Relation:** Let o_1 and o_2 be two instructions performed by the same thread, such that $o_1 \xrightarrow{po} o_2$. Then $o_1 \xrightarrow{CR} o_2$ if one of the following holds:
>
> **same variable,** where o_1 and o_2 access the same variable, or
>
> **read-before-write,** where o_1 is a READ and o_2 is a WRITE.

We define $\text{Java}_{\text{VM}}^{N}$ to be $LS(CR)$ (see Section 1.4.2).

3.1 $\text{Java}_{\text{VM}}^{N}$ vs. Processor Consistency

As mentioned in Section 2.3 (refer to this section for precise definitions), Ahamad et al. [1] defined two variants of PC, namely PCG and PCD. Here we give examples that can be shown to be $\text{Java}_{\text{VM}}^{N}$ and are known not to be PC (one for each variant).

	Thread 1	*Thread 2*	*Thread 3*	*Thread 4*
1	W X,0		W Z,0	
2	W X,1		W Z,1	
3	W Y,1	R Y,1	W V,1	R V,1
4		R Z,0		R X,0

Figure 8: An execution (taken from [1]) which is invalid for PCD and is valid for $\text{Java}_{\text{VM}}^{N}$.

Figure 8 presents an execution which is shown in [1] to be invalid for PCD. However, it is valid in $\text{Java}_{\text{VM}}^{N}$, as we proceed to show by constructing a legal serialization consistent with CR.

By the rules of $\text{Java}_{\text{VM}}^{N}$, the relevant constraints are 1.1 $\xrightarrow{CR}$ 1.2, 3.1 $\xrightarrow{CR}$ 3.2. Now the following serialization is legal and obeys the constraints:

W Y,1	R Y,1	W X,0	R X,0	W X,1	W V,1	R V,1	W Z,0	R Z,0	W Z,1

	Thread 1	*Thread 2*
1	W X,0	W Y,0
2	W X,1	W Y,1
3	W Z,0	W Z,1
4	R Y,0	R X,0

Figure 9: An execution (taken from [1]) which is invalid for PCG and is valid for $\text{Java}^{N}_{\text{VM}}$.

Figure 9 presents an execution which is shown in [1] to be invalid for PCG. However, it is valid for $\text{Java}^{N}_{\text{VM}}$, as follows from the legal serialization below.

W X,0	R X,0	W Y,0	R Y,0	W X,1	W Z,0	W Y,1	W Z,1

We claim that both PCG and PCD are incomparable with $\text{Java}^{N}_{\text{VM}}$. To this end, recall that we have already shown in Section 2 that PCD and PCG contain executions that are not valid in Java_{PS}. We now refer the reader to the results in the next section, which show that $\text{Java}^{N}_{\text{VM}}$ is stronger than Java_{PS}. From these two directions, the claim is derived.

3.2 $\text{Java}^{N}_{\text{VM}}$ vs. Causal Consistency

Although $\text{Java}^{N}_{\text{VM}}$ is reminiscent of Causal consistency [11, 17], they are incomparable. Basically, Causal consistency requires that if two WRITEs are causally related, they are necessarily seen by all other threads in the same order. In $\text{Java}^{N}_{\text{VM}}$, on the other hand, two READs that are not dependent can see the WRITEs in reverse order.

Thread 1	*Thread 2*	*Thread 3*
W X,1	R Y,1	R X,1
R X,1	R X,0	R Y,0
W Y,1		

Figure 10: An execution which is invalid for Causal consistency as in [11, 17] and is valid for $\text{Java}^{N}_{\text{VM}}$.

Figure 10 gives an execution valid for Coherence but invalid in Causal consistency: although the W Y,1 is causally dependent on the W X,1, threads 2 and 3 see the writes in the reverse order. The following legal serialization for this same example preserves the Causality relation, thus showing that the example is $\text{Java}^{N}_{\text{VM}}$.

R X, 0	W X, 1	R X, 1	R X, 1	R Y, 0	W Y, 1	R Y, 1

Figure 7 showed an execution which is valid for Causal consistency. This example is invalid for $\text{Java}^{N}_{\text{VM}}$, as we later show that $\text{Java}^{N}_{\text{VM}}$ is equivalent to Java_{VM}, and Java_{VM} can be shown to be coherent along the lines of the proof given in Section 2.1. The fact that the example is invalid for $\text{Java}^{N}_{\text{VM}}$ also follows from the definition of $\text{Java}^{N}_{\text{VM}}$.

3.3 The Proof

Theorem 3.1 *Java$_{\mathrm{VM}}$ is equivalent to Java$^N_{\mathrm{VM}}$.*

Proof: Direction 1: *Java$_{\mathrm{VM}}$ is not weaker than Java$^N_{\mathrm{VM}}$.* For any program P, for any execution H of P valid under Java$_{\mathrm{VM}}$, H is valid under Java$^N_{\mathrm{VM}}$.

If H is valid under Java$_{\mathrm{VM}}$, it has a schedule T of `load/store` and `read/write` operations obeying the Java$_{\mathrm{VM}}$ constraints. To show that it is valid under Java$^N_{\mathrm{VM}}$ we must construct a serialization S of the operations in H that is consistent with CR.

By Constraint A.2.4, the application of the Java$_{\mathrm{VM}}$ constraints to T may not contain a cycle. This implies that the constraints induce a DAG on the set of operations in the execution. Thus, by applying a topological sort to T, we can obtain a serialization S' of T which preserves the Java$_{\mathrm{VM}}$ constraints. Let S be the sub-sequence of the `read/write` operations in S', where each `read` operation is converted to a READ and each `write` operation is converted to a WRITE.

We must show that S is a legal serialization of H and that it is consistent with CR.

S is a legal serialization of H

Let o_1 be a WRITE X,V and o_2 be a READ X,V. We must show that o_1 precedes o_2 in S and that no other READ X/WRITE X operation intervenes between them.
By Constraint A.2.2, the order of `read/write` operations on each variable must be a legal serialization. Therefore, `write(`o_1`)` must precede `read(`o_2`)` in T. By the construction, `write(`o_1`)` will precede `read(`o_2`)` in S'. Therefore, o_1 will precede o_2 in S.

Since the order of `read/write` operations in T is a legal serialization, no `read x,w/write x,w` appears in T between `write(`o_1`)` and `read(`o_2`)`. By the construction, this implies that no READ X,W/WRITE X,W would appear between o_1 and o_2 in S.

S is consistent with CR

Let o_1 and o_2 be two operations such that $o_1 \xrightarrow{CR} o_2$. Let us examine the possible cases.

1. *o_1 and o_2 are two accesses to the same variable, performed by the same thread.* If $o_1 \xrightarrow{CR} o_2$, then `load(`o_1`)/store(`o_1`)` precedes `load(`o_1`)/store(`o_1`)` in T. By Constraint A.4.3, `read(`o_1`)/write(`o_1`)` will precede `read(`o_2`)/write(`o_2`)`. By the construction, this implies that o_1 will precede o_2 in S.

2. *o_1 is a* READ X,V *and o_2 is a* WRITE Y,W *performed by the same thread.* Since $o_1 \xrightarrow{CR} o_2$, `load(`o_1`)` precedes `store(`o_2`)` in T. By Constraint A.4.1, `read(`o_1`)` must precede `load(`o_1`)` in T, and by Constraint A.4.2, `write(`o_2`)` must follow `store(`o_2`)`. By transitivity, `read(`o_1`)` must precede `write(`o_2`)` in T, and therefore o_1 will precede o_2 in S.

Direction 2: *Java$^N_{\mathrm{VM}}$ is not weaker than Java$_{\mathrm{VM}}$.* For any program P, for any execution H of P, if H is valid under Java$^N_{\mathrm{VM}}$, then H is valid under Java$_{\mathrm{VM}}$.
If H is valid under Java$^N_{\mathrm{VM}}$, there exists a legal serialization S of H that preserves CR. To show that H is valid under Java$^N_{\mathrm{VM}}$, we must construct a schedule T of H comprised of `load/store/read/write` instructions, and this schedule must be consistent with the Java$_{\mathrm{VM}}$ constraints.

We construct the schedule by the following steps:

1. Construct S' out of S by replacing each READ X,V with a `read x,v` and each WRITE X,V with a `write x,v`.

2. Extend S' to a sequence of operations S^T according to the following iterative process.
   Let S^T be S'. For every thread p in H, for every operation o in the local history of p:
   - If o is a READ X,V, insert `load`^{p} `x,v` immediately after whichever instruction came last: its corresponding `read x,v` or the last `load/store` instruction that was inserted.
   - If o is a WRITE X,V, insert `store`^{p} `x,v` immediately after the last `load/store` instruction that was inserted.

   S^T may be converted to a required schedule T in the following way: Each instruction performed at location i in S is performed at time i in T. Each `load`^{p}/`store`^{p} instruction is performed in the local history of thread p in T. The `read/write` operations are performed in the main memory part of T.

   The tag p on `load/store` operations is used to construct the schedule only; it is discarded afterwards.

We next show that T complies with the Java$_{\mathrm{VM}}$ constraints. In order for T to comply with these constraints, the following requirements must hold:

1. A `load x,v` must follow its corresponding `read x,v` (Constraint A.4.1).
2. A `store x,v` must precede its corresponding `write x,v` (Constraint A.4.2).
3. Main memory instructions which access the same variable and for which the corresponding `load/store`s appear in the same local history must agree with the program order of that local history (Constraint A.4.3).
4. In the order of operations performed by the main memory for each variable x, every `read x` must yield the value of the latest `write x` (Constraint A.2.2).

Proof of 1. The proof follows from the construction: each `load x,v` is explicitly placed in S^T after its corresponding `read x,v`.

Proof of 2. Assume the contrary, i.e., that there exists a WRITE operation w in the local history of thread p such that `store(`w`)` is placed in S^T after `write(`w`)`. Suppose that w is the first such operation in the program order of p. By the construction of T, this can happen only if there exists another operation o whose corresponding `load/store` instruction precedes `store(`w`)` in T and was inserted after `write(`w`)`. Only the insertion of a `load` may cause a `load/store` instruction to succeed a `write` instruction; therefore o must be a READ. By the construction, this could happen only if `read(`o`)` appears in T after `write(`w`)`.

Thus, we have the following situation in T: `load(`o`)` precedes `store(`w`)`, and `write(`w`)` precedes `read(`o`)`. However, we know that in H, $o \xrightarrow{CR} w$, and thus o would precede w in S. Therefore, in T, `read(`o`)` would precede `write(`w`)`, a contradiction.

Proof of 3. Since all operations on the same variable in a local history are causally related, the program order of all the accesses to the same variable is preserved in S. By the construction of T, this order is preserved in T, and so this requirement of Java$_{\mathrm{VM}}$ is satisfied.

Proof of 4. Since the main memory operations in S^T appear according to the order of the corresponding instructions in S, which is a legal serialization, the constraint follows. ∎

4 Specification for Java Programmer View (Excluding Prescient Stores)

In this section we provide definitions for Java$_{\mathrm{PS}}$, the memory model consisting of all the constraints from Appendix A for regular variables. The definitions do not include the optimizations that are made possible by the prescient stores from Appendix A.7.

We first explain the difference between the implementor view (Java_{VM}) and the programmer view (Java_{PS}), and then provide a non-operational definition and a formal proof for its equivalence to the original, operational one.

4.1 Java_{PS} vs. Java_{VM}

We emphasize here that Java_{PS} is not equivalent to Java_{VM}. They differ in the freedom given to the compiler by the Java_{PS} definition to decide whether to insert a `load` instruction for a given `use`. The absence of a `load` instruction in the execution sometimes permits a schedule that would be impossible in its presence. See, for example, execution depicted in Figure 11. This execution is possible under Java_{PS}, as the compiler can generate the following sequence of `use`/`assign`/`load`/`store`s for thread 1:

$$\boxed{\texttt{a x,0}}\ \boxed{\texttt{s x,0}}\ \boxed{\texttt{a x,1}}\ \boxed{\texttt{u x}}\ \boxed{\texttt{a y,1}}\ \boxed{\texttt{s y,1}}\ \boxed{\texttt{s x,1}},$$

and the following sequence for thread 2:

$$\boxed{\texttt{l y}}\ \boxed{\texttt{u y}}\ \boxed{\texttt{a x,2}}\ \boxed{\texttt{s x,2}}\ \boxed{\texttt{l x}}\ \boxed{\texttt{u x}}.$$

Then, it is easy to find a schedule in which the `load x` in thread 2 will see the `store x,0` in thread 1, and the `load y` in thread 2 will see the `store y,1` in thread 1. The resulting execution will obtain the required result.

However, this execution is impossible under $\text{Java}^N_{\text{VM}}$. There is a Causality relation between each pair of successive operations in the local history of thread 1: $1.0 \xrightarrow{CR} 1.1$ and $1.1 \xrightarrow{CR} 1.2$ because of the same variable rule, and $1.2 \xrightarrow{CR} 1.3$ because of the read-before-write rule. There is also a Causality relation between each pair of successive operations in thread 2: $2.1 \xrightarrow{CR} 2.2$ because of read-before-write, $2.2 \xrightarrow{CR} 2.3$ because of same variable. By transitivity, $1.0 \xrightarrow{CR} 1.3$ and $2.1 \xrightarrow{CR} 2.3$.

In order for a serialization to be legal, 1.0 should precede 2.3, and 1.1 should not intervene between them. Since $1.0 \xrightarrow{CR} 1.1$, we get that 2.3 must precede 1.1. Also, 1.3 should precede 2.1. From all the constraints above, we get the following cycle: $2.3 \longrightarrow 1.1 \xrightarrow{CR} 1.3 \longrightarrow 2.1 \xrightarrow{CR} 2.3$. Therefore, no legal serialization can satisfy these constraints. We conclude that the execution is not valid under $\text{Java}^N_{\text{VM}}$ and therefore it is not valid for Java_{VM}.

	Thread 1	Thread 2
0	WRITE X,0	
1	WRITE X,1	READ Y,1
2	READ X,1	WRITE X,2
3	WRITE Y,1	READ X,0

Figure 11: Execution which is valid for Java_{PS} and invalid for Java_{VM}.

To adjust $\text{Java}^N_{\text{VM}}$ for this scenario, we weaken it in the following way: We say that there is CR^T dependency from a `use` to a subsequent `assign` in the local history when the `use` yields the result of an `assign` from some other local history. In other words, since the `use` sees the `assign` from another thread it must have a corresponding `load`, and thus to behave according to the Causality relation.

4.2 $\text{Java}_{\text{PS}}^{N}$

We now formally introduce the resulting model, called $\text{Java}_{\text{PS}}^{N}$. We start with the *Causality*^{T} *relation*, denoted CR^T:

> **Causality^T relation:** Let o_1 and o_2 be two instructions performed by the same thread, where $o_1 \xrightarrow{po} o_2$. Then, $o_1 \xrightarrow{CR^T} o_2$ if one of the following holds:
>
> **same variable,** where o_1 and o_2 access the same variable, or
>
> **transistor rule,** [1] where o_1 is a $\text{READ}^{\swarrow}$ and o_2 is a WRITE.

We define $\text{Java}_{\text{PS}}^{N}$ to be $LS(CR^T)$.

Note that CR^T is strictly weaker than CR. They are distinguished by the switch from the read-before-write rule to the transistor rule. The difference implies that some pairs of `use`-before-`assign` operations which were related by CR cease to be related by CR^T. In the reverse direction, when there are no `use`-before-`assign` operations which are related by CR but not related by CR^T, then the set of relations induced on the execution is the same.

Theorem 4.1 *$Java_{\text{PS}}$ is equivalent to $Java_{\text{PS}}^{N}$.*

Proof: Direction 1. *$Java_{\text{PS}}^{N}$ is not weaker than $Java_{\text{PS}}$.* For each program P, for each execution H of P, if H is valid under $\text{Java}_{\text{PS}}^{N}$ then it is also valid under Java_{PS}.

Since H is valid under $\text{Java}_{\text{PS}}^{N}$, there exists a legal serialization S of all operations in H, consistent with the Causality^T relation. To show that H is valid under Java_{PS}, we provide the orders of `use/assign/load/store` instructions performed by each thread and the `read/write` instructions performed by the main memory. We construct the scheduling for all these operations that is consistent with the Java_{PS} rules.

The construction proceeds as follows. First, we extract from S the sub-serialization S' consisting only of $\text{READ}^{\swarrow}$ and WRITE instructions, converting all the $\text{READ}^{\swarrow}$ operations into equivalent `reads` and WRITE operations — into `writes`. Then we augment S' with `use`, `assign`, `load` and `store` instructions, as follows.

For each thread p in H, for each instruction o in the local history of p:

- If o is a $\text{READ}^{\downarrow}$, insert $\texttt{use}^p(o)$ after the last operation inserted for p.
- If o is a $\text{READ}^{\swarrow}$, insert $\texttt{load}^p(o)$ and $\texttt{use}^p(o)$ after whichever operation was inserted last: the last operation inserted for p or $\texttt{read}(o)$.
- If o is a $\text{WRITE}_{\downarrow}$ or a $\text{WRITE}_{\searrow}$, insert $\texttt{assign}^p(o)$ and $\texttt{store}^p(o)$ after the last operation inserted for p.

The tag p on each inserted instruction denotes the thread from which it originated; it is used to construct the schedule and then discarded. Note that the operations are explicitly inserted in a way that preserves the original program order.

S' is converted to the required schedule T in the following way. The i^{th} instruction in S' is performed at the i^{th} time step, where `use/assign/load/store` instructions are performed by the thread matching their tags, and the `read/write` instructions are performed by the main memory.

In T, all the `use` operations yield the same values as their corresponding READ origins in H, and `assign`s provide the same values as their corresponding WRITEs. This implies that T defines

[1] The transistor rule is named after the transistor mode of operation, where there is a connection between the output and the input if there is a signal coming from another direction. This is also the origin of the superscript T in CR^T.

the transfer of values from WRITEs to READs in exactly the same way as H. It thus remains to show that T complies with all the Java$_{PS}$ constraints (i.e., those listed in A.2, A.3, and A.4).

Constraint A.2.1. This constraint requires that every `use`^{p} `x` and `store`^{p} `x` see the value written by the most recent `assign`^{p} `x` or `load`^{p} `x`.

By the construction, a `use` can be inserted as a consequence of a READ$^{\downarrow}$ or a READ$^{\swarrow}$ instruction. A `store` can be inserted as a consequence of a WRITE$_{\downarrow}$ or a WRITE$_{\searrow}$ instruction. Let us examine the possible cases.

- *A* READ$^{\downarrow}$X,V *operation o* (issued by thread p). By the construction, `use`^{p} `x,v` has been inserted to S' for o. Let o' be the WRITE X,V operation. Since o is not a READ$^{\swarrow}$, o' must also be performed by p. Since S is a legal serialization, $o' \xrightarrow{S} o$, and there is no other WRITE X,W or READ X,W operation between o' and o. Since o and o' access the same variable, there is a CR^T relation between them. This implies that o' precedes o in the program order of p.

  Since the `use`/`assign` instructions from the same thread are inserted according to its program order, `assign`(o') is inserted to S' before `use`(o) and no other `use`^{p} `x,w`/`assign`^{p} `x,w` instruction can be inserted between them. `load` and `store` instructions are inserted adjacent to the corresponding `use`s and `assign`s. Therefore, there are also no `load`^{p} `x,w` and `store`^{p} `x,w` instructions between `assign`$^p(o')$ and `use`$^p(o)$.
- *A* READ$^{\swarrow}$X,V *operation o*. By the construction, for o, `load`$^p(o)$ and `use`$^p(o)$ are inserted. Therefore, `load`$^p(o)$ immediately precedes `use`$^p(o)$, and `use`$^p(o)$ sees the result of the most recent `load`^{p} `x`/`assign`^{p} `x` operation.
- A WRITE$_{\downarrow}$X,V or a WRITE$_{\searrow}$ operation o. For o, `store`$^p(o)$ is inserted immediately after `assign`$^p(o)$, and therefore yields the result of the most recent `load`^{p} `x`/`assign`^{p} `x` operation.

Constraint A.2.2 is satisfied since S is a legal serialization and the order of `read`/`write` operations in S' is identical to the order of READ/WRITEs in S.

Constraint A.2.3 is irrelevant since we examine the model without synchronization operations.

Constraint A.2.4 clearly follows from the fact that S' is a serialization.

Constraint A.3.1 holds since the construction preserves for any one thread its original program order from H.

Constraints A.3.2 and A.3.3 hold since by the construction a `store` is inserted immediately after each `assign` in S'.

Constraint A.3.4 holds. Since S is a legal serialization, a READ X operation o sees a value written by some WRITE X operation o'. By the construction, if o' is performed in the same thread as o, then `assign`(o') precedes `use`(o) in the program order. If o' is performed in a different thread, then a `load x` is inserted before `use`(o). In any case, every `use`(o) yields a value written by some `assign` or `load` instruction.

Constraint A.4.1 holds since a `load x,v` is explicitly inserted to S' after the `read x,v`.

Constraint A.4.2 holds. Assume the contrary, i.e., that there exists an operation o in H in the local history of thread p such that `store`(o) is inserted into S' after `write`(o).

By the construction, if a `store`^{p} `x,v` is inserted after a `write x,v`, another `use`^{p}/`assign`^{p}/`load`^{p}/`store`^{p} instruction must already have been inserted after `write x,v`.

Consider the first instruction i from the program order of p whose `use/assign/load/store` is inserted after `write`(o). If i is a WRITE, then by the construction, `store`$^p(i)$ and `assign`$^p(i)$ would be inserted immediately after the last instruction already inserted, i.e., before `write`(o). Similarly, if i is a READ, `use`$^p(i)$ would be inserted before `write`(o). Therefore, i must be a READ$^{\swarrow}$. Moreover, in order for `load`$^p(i)$ and `use`$^p(i)$ to be inserted after `write`(o), `read`(i) must follow `write`(o) in S'.

In summary, `use`$^p(i)$ precedes `assign`$^p(o)$ in S', and `write(`o`)` precedes `read(`i`)`. However, this means that i precedes o in the program, while o precedes i in S. Since i is a READ$^{\checkmark}$, this implies that S violates the transistor rule between i and o.

Constraint A.4.3 holds. Consider two operations, o_1 and o_2, accessing x in the same thread p. Assume WLOG that o_1 precedes o_2 in the program order. The two operations access the same variable. Thus $o_1 \xrightarrow{CR^T} o_2$, and, as a result, $o_1 \xrightarrow{S} o_2$.

By the construction, `read(`o_1`)/write(`o_1`)` will be inserted to S' before the `read(`o_2`)/write(`o_2`)`. Also, the `use(`o_1`)/assign(`o_1`)` will be inserted to S' before the `use(`o_2`)/assign(`o_2`)`. Therefore, o_1 and o_2 comply with the constraint.

Direction 2. *Java*$_{\mathrm{PS}}$ *is not weaker than Java*$^N_{\mathrm{PS}}$. For each program P, for each execution H of P, if H is valid under Java$_{\mathrm{PS}}$ then H is valid under Java$^N_{\mathrm{PS}}$.

Given an execution H which has a Java$_{\mathrm{PS}}$ schedule T of `u/a/l/s/r/w` instructions, we construct a legal serialization S of H that is consistent with CR^T, as follows. First, we construct a serialization S' of the `load/store` instructions in T which is consistent with the Causality relation between `load/store`s. Then, we add the `use/assign`s from T to S', and use the sub-sequence of added instructions as the required serialization S of H.

Constructing the serialization S

Let T' be the sub-schedule of T consisting of only the `load/store/read/write` instructions. If we view the sequences of `load/store` instructions in T' as an execution H', then T' is a valid Java$_{\mathrm{VM}}$ schedule of H'. Since Java$_{\mathrm{VM}}$=Java$^N_{\mathrm{VM}}$, there exists a legal serialization S' of H' consistent with CR. We extend S' by embedding the `use/assign` operations from T, as follows.

Stage 1 – Mapping `assign`s: Iterate through the `store` operations in S'. For a `store x` operation o, let p be the thread that issued it in H. Insert `assign(`o`)`, along with all the `assign x` instructions which precede it in the program order of p and which have not yet been inserted, immediately before `store(`o`)`.

Stage 2 – Mapping `use`s: Iterate through the `load` and `assign` instructions in S'. For a `load x/assign x` operation o, let p be the thread that issued it in H. Insert, immediately after o, all the `use x` instructions in the program order of p which see o's value. (Recall our assumption that the value yielded by a `use` can always identify a `load` or an `assign`.)

Stage 3 – Mapping leftovers: For each thread p, attach the sequence of all the remaining `assign`s and `use`s (for all variables) to the end of S', preserving their program order.

Let S be the sub-sequence of `use/assign` operations in S', where the `use` operations are converted into READs, and the `assign`s — into WRITEs. We use S as the required serialization of H.

The construction of S immediately implies that the values yielded by READ operations and the values supplied by WRITE operations in S are the same as in H. Therefore, S defines the same transfer of values as H. If S is a legal serialization and the order of READs and WRITEs in S is consistent with CR^T, this will imply that H is valid for Java$^N_{\mathrm{PS}}$.

Properties of S'

Let us first explore some properties of S'.

Claim 1 *For each* `store x,v` *operation s in S', the corresponding* `assign x,v` *operation a is inserted immediately before s.*

Proof: Consider the process of inserting the `assign` operations (stage 1). If a has not yet been inserted to S' when s is processed, the algorithm inserts it immediately before s and the claim holds. If a has already been inserted, then there must exist an `assign x,w` operation a' following a in T such that the `store x,w` operation s' precedes s in S'. However, by Constraint A.2.1, if a precedes a' then s must also precede s' — a contradiction. ∎

Claim 2 *For each variable x, for each thread p, the order of* `assign x` *and* `load x` *operations in the local history of p in T is preserved in S'.*

Proof: Let o_1 and o_2 be two `assign x` or `load x` operations such that o_1 precedes o_2 in T. Consider the possible cases.

- *Both o_1 and o_2 are* `load`*s.* Since o_1 and o_2 access the same variable, $o_1 \xrightarrow{CR} o_2$, and therefore o_1 must precede o_2 in S'.
- *Both o_1 and o_2 are* `assign`*s.* If at the time o_2 is processed, o_1 has not yet been inserted, then it would be inserted before o_2 by the algorithm.
- *o_1 is a* `assign` *and o_2 is a* `load`*.* By Constraint A.3.2, in T there must be a `store x` operation o_3 between o_1 and o_2. Since $o_3 \xrightarrow{CR} o_2$, o_3 will precede o_2 in S'. By Claim 1, o_1 would precede o_3 in S', and by transitivity, o_1 must precede o_2 in S'.
- *o_1 is a* `load` *and o_2 is a* `assign`*.* If o_2 is inserted at stage 3, then it necessarily follows o_1. If o_2 is inserted at stage 1, then there is a `store` operation o_3 which follows o_2 (and thus o_1) in T. Therefore, $o_1 \xrightarrow{CR} o_3$ and o_1 precedes o_3 in S'. By Claim 1, o_2 is inserted to S' immediately before o_3, and therefore o_1 precedes o_2 in S'.

∎

S is a legal serialization of H

We show that the order of `use` and `assign` operations in S' is a legal serialization, i.e., each `use x` sees the result of the most recent `assign x`. Since the order of READ/WRITE operations in S is the same as the order of `use`/`assign` operations in S', this will imply that S is a legal serialization.

There are three ways in which `use`s are inserted to S':

- *A* `use x,v` *operation u that was inserted at stage 2 as a consequence of an* `assign x,v` *operation a.* By the construction, u is inserted to S' immediately after a, and thus no other `assign x` can intervene between them. Therefore, u sees the value of the most recent `assign x` in S'.
- *A* `use x,v` *operation u that was inserted at stage 2 as a consequence of a* `load x,v` *operation l.* Once again, u is inserted immediately after l, and no other `assign x` can intervene between them in S'.

  By the construction, S' is a legal serialization of `load`/`store`s. Therefore, l yields the result of the most recent `store x` operation s in S'. By Constraint A.3.3, s necessarily had a corresponding `assign x,v` operation. By Claim 1, a was inserted immediately before s. The same claim implies that `assign`s are inserted immediately before their corresponding `store`s Therefore no `assign x,w` could be inserted between s and l in S'. Hence, S' contains a sequence of `assign x,v`–`store x,v`–`load x,v`–`use x,v` operations with no other `assign x` instruction in between. Thus, in S', u yields the result of the closest preceding `assign x`.
- *A* `use x` *operation u that was inserted at stage 3.* By Constraint A.2.1, `use` yields the value produced by some `load` or `assign`. By the construction, all `use`s are inserted into S' during

stage 2, except those that see the value provided by an `assign` which was not inserted at stage 1. Thus, the `use`s that are inserted at stage 3 have their corresponding `assign`s preceding them in the program order of the same thread, and these `assign`s are also added to S' at stage 3.

Each `use x` which is inserted at stage 3 sees a result of an `assign x` which was also handled at that stage. By Constraint A.2.1, `use`s and `assign`s of each thread constitute a legal serialization. Therefore, each sequence of `assign`s and `use`s inserted to S' at stage 3 is itself a legal serialization. Since those sequences are not interleaved by the insertion algorithm, they remain legal in S'.

We conclude that u yields the value provided by the most recent `assign` operation in S'.

Since stage 3 does not interleave the inserted sequences, the sequence remains legal in S', and thus u sees the value written by the most recent `assign x`.

S is consistent with $\mathbf{CR}^T$

We show that the order of `use`/`assign` operations in S' is consistent with CR^T. Since the order of READ/WRITE operations in S is the same as the order of `use`/`assign` operations in S', this will imply that S is consistent with CR^T.

We check all the pairs of instructions related by CR^T to see that the order is preserved in S'.

Let o_1 and o_2 be two instructions such that $o_1 \xrightarrow{CR^T} o_2$. This implies that o_1 and o_2 belong to the same thread and o_1 precedes o_2 in H. By Constraint A.3.1, the `use`/`assign` of o_1 precedes the `use`/`assign` of o_2 in T. We need to show that the `use`/`assign` of o_1 precedes the `use`/`assign` of o_2 in S'.

If o_1 and o_2 were inserted at stage 3, then by the construction, the program order would be preserved between them. If o_1 was inserted at stage 1 or 2, and o_2 at stage 3, then again o_1 would precede o_2 in S'.

If o_1 was inserted at stage 3 and o_2 at stage 1 or 2, we get a contradiction:

- *Same Variable Rule.* If both o_1 and o_2 are `assign`s, then by the construction, insertion of o_2 at stage 1 would also insert o_1.

  If o_1 is an `assign` and o_2 is a `use`, o_2 could be handled before stage 3 because it sees the result of a `load` operation o_3 or an `assign` operation o_4 which was inserted at stage 1. In the first case, Constraint A.3.2 requires a `store x,v` between o_1 and o_2. Therefore, o_1 would be inserted at stage 1. In the second case, we have again two `assign`s to the same variable, such that the first is inserted at stage 3 and the second — at stage 1, which is impossible.

  If o_1 is a `use`, then it necessarily sees a value provided by an `assign` operation o_3, where the `assign` is inserted at stage 3. Whether o_2 is an `assign` or a `load`, by one of the previous cases, if o_2 is inserted to S' before stage 3, then o_3 could not be inserted at stage 3 — a contradiction.
- *Transistor Rule.* In this case, o_1 is a `use` and it sees the result of a `load` instruction. Therefore, it could be inserted only at stage 2.

If both o_1 and o_2 were inserted at stage 1 or 2:

- *Same Variable Rule.* By the construction, a `use` is inserted immediately after the `load` or `assign` instruction that provides its value. Let o_1' be o_1 if o_1 is an `assign`, or the `load`/`assign` operation which provided the value of o_1 if o_1 is a `use`. Let o_2' be defined in a similar fashion for o_2. By Constraint A.2.1, o_1' must precede o_2' in T. By Claim 2, o_1' must precede o_2' in S'. Since a `use` is inserted immediately after the `load`/`assign` operation which provided its value, this implies that o_1 precedes o_2 in S'.

- *Transistor Rule.* Since o_1 yields the result written in another thread, there must be a `load` instruction l which provided the value for o_1.

  Since o_2 is an `assign`, and it was handled before stage 3, it could be inserted only during stage 1. This implies that there exists a `store` operation s corresponding to o_2 or to some other `assign` operation o_2' which writes the same variable as o_2, such that o_2 is inserted adjacent to s.

  l precedes o_1 in T, and o_2 precedes s; therefore, l precedes s in T. Thus, $l \xrightarrow{CR} s$, and thus, l precedes s in S'. By the construction, o_1 is inserted immediately after l, and o_2 is inserted just before s. Therefore, o_1 must precede o_2 in S'.

■

5 Specification for Java Programmer View

The JLS includes a compiler optimization called "prescient stores," described in A.7. The optimization lets the compiler make additional transformations to the bytecodes, weakening the consistency model. Therefore, to accommodate our model for prescient stores, we must weaken it accordingly.

The JLS defines that a `store` instruction may be moved to a location earlier in the schedule, if the compiler can determine the value it will have to write. However, it does not define exactly in what situations the compiler can determine the value of a `store` in advance.

Compilers may differ in their ability to predict values of `store`s. Moreover, our model does not consider any semantic information from the program. Therefore, we will characterize the weakest possible interpretation of the prescient stores rules. We thus assume that any `store` instruction can be made prescient, constrained only by the rules from A.7.

For the Java programmer, this means that any program based on our definition would remain correct under any implementation of prescient stores. This also gives freedom to the compiler writer, who can choose the extent to which prescient stores will be implemented.

5.1 The Prescient Transistor Rules

The definition of $\mathrm{Java}^N_{\mathrm{PS}}$ is based on the Same Variable Rule, which governs operations that access the same variable and the Transistor Rule, which governs operations that access different variables. Applying the prescient stores optimization can enable an execution that does not have a serialization that complies with these rules. We address this issue by subdividing the rules into cases. It turns out that we can characterize exactly all the situations in which the rules from $\mathrm{Java}^N_{\mathrm{PS}}$ continue to hold. Therefore, the definition of Java^N will contain the original rules of $\mathrm{Java}^N_{\mathrm{PS}}$ along with additional context that causes these rules to hold.

The Transistor Rule states that if an execution contains two operations, $o_1{=}\text{READ}^{\checkmark}\ \text{X}$ and $o_2{=}\text{WRITE Y}$, such that o_1 precedes o_2 in the program order of some thread, then o_1 will precede o_2 in a conforming serialization. This rule works because o_1 sees a value written in another thread, and thus the $\mathrm{Java}_{\mathrm{PS}}$ schedule must contain `load(`o_1`)`. `load(`o_1`)` must precede `use(`o_1`)`, and therefore it must precede `assign(`o_2`)`. Since `assign(`o_2`)` must precede `store(`o_2`)`, by transitivity `load(`o_1`)` must precede `store(`o_2`)`. Then, similarly, `read(`o_1`)` must precede `write(`o_2`)`. Therefore it is possible to serialize o_1 before o_2.

The prescient stores optimization might violate this order — if `store(`o_2`)` became prescient, it must then be moved to a location preceding `load(`o_1`)`. However, there are situations in which the order is kept. A `store x`, even if prescient, cannot switch places with another `store x` or `load x`.

So if the execution requires a schedule to have a `load x` or `store x` operation between `load(`o_1`)` and `store(`o_2`)`, then the Transistor Rule from o_1 to o_2 would hold.

We have identified three such cases. In Subsection 5.2 we will show that these cases cover all of the possible subcases of the Transistor Rule that hold in the presence of prescient stores.

Case 1 Suppose that the program order of some thread contains the following operations:

$$o_1\text{=READ}^{\swarrow}\text{ X},\ t_1\text{=READ Y,V}_1,\ t_2\text{=READ}^{\swarrow}\text{ Y,V}_2,\ o_2\text{=WRITE Y}$$

Since o_1 sees the result of an operation performed by another thread, a Java schedule must contain `load(`o_1`)`. By Constraint A.2.1, `load(`o_1`)` must precede `use(`o_1`)`. Similarly, t_2 must have a corresponding `load y` operation. By Constraint A.2.1, `load(`t_2`)` must appear between `use(`t_1`)` and `use(`t_2`)`, and thus it must appear after `use(`o_1`)`. By transitivity, `load(`o_1`)` must precede `load(`t_2`)`. By Constraint A.7.3, `store(`o_2`)` cannot be placed before `load(`t_2`)`. Therefore, `store(`o_2`)` must be executed after `load(`o_1`)`. We conclude that in this case the Transistor Rule holds, i.e., o_1 can be serialized before o_2.

Figure 12 presents an example of this case. Since READ Y,2 sees a value written by another thread, the schedule must contain a `load y` operation. It must appear after `use y,0`, since 1.2 sees value 0 and not value 2. The `load x` corresponding to 1.1 must precede `use x,1`. Since 1.1 precedes 1.2 in the program order, the `load x` operation is performed before the `load y` operation. Now even if the `store y` corresponding to 1.4 is prescient, it cannot switch places with `load y` (Constraint A.7.3). Note that this case is the same as the prescient stores fix of the Coherence-not-Java example (Figure 4).

Case 2 Suppose that the program order of some thread contains the following operations:

$$o_1\text{=READ}^{\swarrow}\text{ X},\ t_1\text{=WRITE Y,V}_1,\ t_2\text{=READ}^{\swarrow}\text{ Y,V}_2,\ o_2\text{=WRITE Y.}$$

As in the previous case, the schedule must include `load(`t_2`)`. Since t_1 and t_2 see different values, `load(`t_2`)` must be executed after the `assign(`t_1`)`. Therefore, even if `store(`o_2`)` becomes prescient, it must be executed after `assign(`t_1`)`, and by transitivity, after `use(`o_1`)`. So in this case the Transistor Rule holds too, i.e. $o_1 \longrightarrow o_2$.

Figure 13 presents an example of this case. This case is similar to the previous one, except that operation 1.2 is a WRITE Y. Because of Constraint A.2.1, `load y,2` must be performed after `assign y,0`. Then, as in transistor rule 1, `load(1.3)` must be performed after the `read(1.1)`, and `store(1.4)` cannot be scheduled after `load(1.3)`.

Case 3 Suppose that the program order of some thread contains the following operations:

$o_1\text{=READ}^{\swarrow}\text{ X},\ t_1\text{=WRITE}_{\searrow}\text{Y},\ o_2\text{=WRITE Y.}$

For o_2 to appear before o_1, `store(`o_2`)` must become prescient. However, it cannot switch places with `store(`t_1`)` (A.7.4), so `store(`t_1`)` must also be prescient and execute before `use(`o_1`)`. To summarize, the order of these operations must be `store y` (of t_1), `store y` (of o_2), `use x` (of o_1). But this means that `store y` of o_2 intervenes between `store y` of t_1 and `assign y` of t_1. By Constraint A.7.4, `store y` of o_1 cannot be prescient. We conclude that in this situation $o_1 \longrightarrow o_2$.

Figure 14 presents an example of the above case. In this case, 1.2 is seen by another thread. Thus, a corresponding `store` instruction exists. Now the `store y` of 1.3 cannot switch places with the `load x` of 1.1 because `store y,1` can be placed before the `load x` only if it is prescient. By Constraint A.7.4, however, `store y,1` cannot be moved to a location that precedes `store y,2`. But even if `store y,2` is made prescient, the same constraint forbids putting `store y,1` between `store y,2` and `assign y,2`. Therefore, we conclude that `store y,1` must follow `load x,1`.

	Thread 1	Thread 2	Thread 3
1	READ X,1	WRITE Y,2	WRITE X,1
2	READ Y,0		
3	READ Y,2		
4	WRITE Y,1		

Figure 12: Prescient Transistor Rule 1.

	Thread 1	Thread 2	Thread 3
1	READ X,1	WRITE Y,2	WRITE X,1
2	WRITE Y,0		
3	READ Y,2		
4	WRITE Y,1		

Figure 13: Prescient Transistor Rule 2.

	Thread 1	Thread 2	Thread 3
1	READ X,1	READ Y,2	WRITE X,1
2	WRITE Y,2		
3	WRITE Y,1		

Figure 14: Prescient Transistor Rule 3.

5.2 Java^N

We now give a non-operational definition of the programmer view. We call this definition Java^N. We begin with the definition of the *Causality$^T_{PS}$ relation*, denoted CR^T_{PS}:

Causality$^T_{PS}$ Relation: Let o_1 and o_2 be two instructions performed by the same thread p, such that $o_1 \xrightarrow{po} o_2$. Then, $o_1 \xrightarrow{CR^T_{PS}} o_2$ if one of the following holds:

Same Variable: o_1 and o_2 access the same variable.

Prescient Transistor Rule 1: the program order of p includes the sub-sequence

o_1 =READ$^{\swarrow}$ X, READ Y,V, READ$^{\swarrow}$ Y,W, o_2 =WRITE Y.

Prescient Transistor Rule 2: the program order of p includes the sub-sequence

o_1 =READ$^{\swarrow}$ X, WRITE Y,V, READ$^{\swarrow}$ Y,W, o_2 =WRITE Y.

Prescient Transistor Rule 3: the program order of p includes the sub-sequence

o_1 =READ$^{\swarrow}$ X, WRITE$_{\searrow}$Y, o_2=WRITE Y.

We define Java^N as $LS(CR^T_{PS})$.

Theorem 5.1 *Java is equivalent to Java^N.*

Proof: Direction 1 *Java is not weaker than Java^N.* For each program P, for each execution H of P, if H is valid under Java, it is also valid under Java^N.

If H is valid under Java, then there exists a schedule T of H consistent with all the Java rules. To show that H is valid under Java^N, we must show a serialization of the READ/WRITE operations in H consistent with CR^T_{PS}.

In brief, we construct the serialization in the following way. We convert T to a Java$_{\mathrm{PS}}$ execution T' by moving the `use`/`assign` operations so that every `assign` precedes its corresponding `store`. Then, by Theorem 4.1, T' has an execution H', where H' has a legal Java$^N_{\mathrm{PS}}$ serialization S'. We show that S' is also a valid Java^N serialization of H.

Transforming T into a Java$_{\mathrm{PS}}$ schedule T'

> **Algorithm $\mathbf{A}^{5.1}_1$** For each thread p in P, for each `assign x` operation o, if `store(`o`)` precedes o in T, extract all the `use x`/`assign x` instructions performed by p between `store(`o`)` and o, including o, and insert them, preserving their order, immediately before `store(`o`)`.

The algorithm defines a new program P', a new execution H' and a new schedule T'. By moving all the `assign`s before their corresponding `store`s, we ensure that no `store` instruction in T' is prescient. However, H' still consists of the same operations and defines the same transfer of values between those operations as H.

Note that the algorithm relocates sequences of `use`/`assign` operations which access the same variable. Therefore, for each thread, the orders of `use`/`assign` operations for each variable are not changed. The orders of `load`/`store` operations are not changed either, since those were not moved by the algorithm. However, certain `use`/`assign` operations may have moved relative to `load`/`store`s. In other words, the orders of `use`/`assign`s and of `load`/`store`s do not change, but the interleaving of these orders may change.

To proceed with the proof, and since in T' there are no prescient `store`s, we use Theorem 4.1. In order to use the Theorem, we verify that none of the Java$_{\mathrm{PS}}$ constraints is violated in T'.
<u>*Constraint A.2.1*</u>. The constraint requires that `load x,v`/`assign x,v` must precede the corresponding `use x,v`/`store x,v`.

Constraint A.2.1 holds in T, except for some `store`s which have already been moved presciently to a location that precedes their corresponding `assign`s. In T', each `assign x,v` precedes the `store x,v` if one exists.

Let us examine possible violations of the constraint due to the changed interleaving. We do this by looking at all possible situations that might have been changed by the algorithm.

- *An* `assign x,v` *operation* o_1 *before a* `use x,v` *operation* o_2. Because of Constraint A.2.1, no `use x,w`/`assign x,w` intervenes between o_1 and o_2 in T. The algorithm does not change the relative order of `use x`/`assign x` operations Therefore it would not move any other `use x,w`/`assign x,w` operation between o_1 and o_2. However, `assign x,v` may be moved to a location earlier in the schedule, so that another `load x,w`/`store x,w` operation o' will intervene between o_1 and o_2.

  This can happen only if `store x,v` is scheduled in T before `load x,w`/`store x,w` (see Figure 15a). In this case, `store x,v` violates either Constraint A.7.3 or A.7.4, depending on whether o' is a `load` or a `store`.

- *A* `load x,v` *before a* `store x,v`. By Constraint A.3.3, `assign x,v` must be scheduled in T between the `load` and the `store`. The `load` yields the value provided by another `assign x,v` operation that precedes the `load` (not necessarily in the same thread). Since we assume that each `load` can identify the `assign` operation that provided its value, this case is impossible.
- *An* `assign x,v` *before a* `store x,v`. As noted above, in T' the `assign` will precede the `store`. Another operation o' accessing x may intervene between the `assign` and the `store` in one of the following ways:
  1. o' preceded `store x,v` in T, and `assign x,v` is moved by $A_1^{5.1}$ to a location that precedes o'. Since the relative order of `use/assign` operations is not changed by the algorithm, o' must be a `load x/store x`.
     `assign x,v` could be moved by the algorithm to a location before o' only if this `assign x,v` is followed by an `assign x,w`, such that the `store x,w` precedes o' (see Figure 15b and c). However, in the case of Figure 15b, `store x,w` must violate Constraint A.7.4, and in the case of Figure 15c, `store x,v` violates ConstraintA.2.1.
  2. o' succeeded `store x,v` in T, and it was moved between `assign x,v` and `store x,v`. Since the relative order of `load/store` operations is not changed by the algorithm, o' must be a `use x/assign x`.
     o' could be moved to a location that precedes `store x,v` only if there is an `assign x,w` (which may be o' itself) following `store x,v` such that the `store x,w` is scheduled before `store x,v` in T (See Figure 15d). However, in this case `store x,w` violates Constraint A.7.4 in T.
- *A* `load x,v` *before a* `use x,v`. Since `load x,v` precedes `use x,v` in T, and `use/assign` instructions are only moved by the transformation to a location earlier in the schedule, and since the order of `use x/assign x` instructions is not changed by the algorithm, no other `use/assign` instruction can intervene in T' between `load x,v` and `use x,v`. Therefore, Constraint A.2.1 could only be violated in T' if `use x,v` were to be moved to a location preceding `load x,v`. However, this could happen only if in T there is an `assign x,w` following `use x,w` such that `store x,w` precedes `load x,v` (See Figure 15e). This means that `store x,w` violates Constraint A.7.3 in T.

Constraint A.2.2. This constraint is unaffected by the algorithm since the `load/store` and `read/write` instructions in the schedule are left unchanged.
Constraint A.2.3. This constraint is irrelevant since Java^N does not consider synchronization operations.
Constraint A.2.4. This constraint holds since T' is a schedule which determines a specific timing for each operation.
Constraint A.3.1. This constraint holds since we explicitly define the order of operations in P' exactly as we define it in T'.
Constraint A.3.2. Let o_1 and o_2 be an `assign x,v` and a `load x,v` in the same thread, such that o_1 precedes o_2. Except when they are moved presciently to an earlier location in the schedule, the `store`s in T must comply with the Java constraints. Thus, there exists a `store` operation o' in T, where o' corresponds to o_1 or to some other `assign` operation between o_1 and o_2 in T, and o' would be placed between o_1 and o_2 in T if it was not prescient. By the algorithm, o_1 will precede o' in T'. Therefore, the constraint holds.
Constraint A.3.3. Let o_1 be a `load x,v` or a `store x,v`, and o_2 be a `store x,w`, such that o_1 precedes o_2 in the program order of some thread in T. By Constraint A.3.3, T must contain an

`assign x,w` operation o between o_1 and o_2. However, o_2 might be scheduled in T to precede o as a prescient `store`.

The algorithm would move o to a location that precedes o_2, restoring the constraint. o could not be moved to a location that precedes o_1 because it would then violate Constraint A.2.1. We have already shown that Constraint A.2.1 holds in T'. Therefore, o is scheduled in T' between o_1 and o_2, and the constraint holds.

Constraint A.3.4. If the sequence of operations on variable x of some thread in T begins with an `assign x` or a `load x` operation o, then the algorithm would not insert any other operation that accesses x in a location preceding o. The sequence could begin with a `store x,v` operation o', if o' was a prescient `store`. Then, the sub-sequence of operations from o' to `assign x,v` would contain only `use`/`assign`s, and because of Constraint A.3.4, it would begin with an `assign`. By the algorithm, this sub-sequence will be moved to a location preceding o', and therefore the order of operations on x will begin with an `assign x` operation.

Constraints in A.4. These constraints are not affected by the transformation, since it does not change the relative order of `load`/`store` and `read`/`write` operations.

We have shown that T' is a Java_{PS} reschedule of T. By Theorem 4.1, there exists a $\text{Java}^N_{\text{PS}}$ serialization S' of the operations of an execution H' of T'. We use S' as the required Java^N serialization S of H.

S' is a Java^N serialization for H

We show that for every two instructions o_1 and o_2 in H such that $o_1 \overset{CR^T_{PS}}{\longrightarrow} o_2$, o_1 precedes o_2 in S'. Since S' consists of the same operations as S, this will imply that S' is a valid serialization for H.

All the instances of the Prescient Transistor Rule consist of a READ$^{\swarrow}$ operation o_1 followed by a WRITE operation o_2. For each Prescient Transistor Rule instance in H, we show that the algorithm would keep o_1 before o_2 in H'. In Java_{PS}, the Transistor Rule always holds between a READ$^{\swarrow}$ and a following WRITE. Therefore, o_1 will precede o_2 in S'. This will imply that S' is a valid Java serialization for H.

Since the orders of READ/WRITE operations in H' are the same as the orders of `use`/`assign` operations in T', we show that for every pair of operations o_1 and o_2 which create a Prescient Transistor Rule in H, `use(`o_1`)` precedes the `assign(`o_2`)` in T'.

Let o_1 and o_2 be two operations in the local history of some thread p in H, such that $o_1 \overset{CR^T_{PS}}{\longrightarrow} o_2$.

Same Variable Rule Since the transformation does not change the relative order of the `use`/`assign` operations that access the same variable, `use(`o_1`)/assign(`o_1`)` precedes `use(`o_2`)/assign(`o_2`)` in T'.

Prescient Transistor Rule 1 H contains the following sub-sequence in the program order of thread p:

$$o_1\text{=READ}^{\swarrow}\text{ X},\ t_1\text{=READ Y,V}_1,\ t_2\text{=READ}^{\swarrow}\text{ Y,V}_2,\ o_2\text{=WRITE Y}$$

Since t_2 is a READ$^{\swarrow}$, T must contain a `load(`t_2`)`. By Constraint A.2.1, `use(`t_1`)` must precede `load(`t_2`)`. By Constraint A.3.1, `use(`o_1`)` must precede `use(`t_1`)`. By transitivity, `use(`o_1`)` must precede `load(`t_2`)`. There are two cases involving o_2:

- H contains `store(`o_2`)`. By Constraint A.7.3, the `store`, even if it is prescient, must follow `load(`t_2`)` in T. By transitivity, `use(`o_1`)` must precede `store(`o_2`)`. Therefore, the algorithm $A^{5.1}_1$ would not move `assign(`o_2`)` to a location preceding `use(`o_1`)`. This implies that `use(`o_1`)` precedes `assign(`o_2`)` in T'.

- H has no `store(`o_2`)`. The algorithm could move `assign(`o_2`)` to a location preceding `use(`o_1`)` in T' only if H contains another WRITE operation o', such that `store(`o'`)` precedes `load(`t_2`)` in T, while `assign(`o'`)` is scheduled after `assign(`o`)`. However, this implies that `store(`o'`)` violated Constraint A.7.3.

We conclude that `assign(`o_2`)` must be scheduled after `use(`o_1`)` in T'.

Prescient Transistor Rule 2 H contains the following sub-sequence in the program order of thread p:

$$o_1 = \text{READ}^{\swarrow}\ \text{X},\ t_1 = \text{WRITE Y},\ t_2 = \text{READ}^{\swarrow}\ \text{Y},\ o_2 = \text{WRITE Y}.$$

Since t_2 sees the value written in another thread, H must contain `load(`t_2`)`. By Constraint A.2.1, `assign(`t_1`)` must precede `load(`t_2`)` in T. By transitivity, `use(`o_1`)` must precede `load(`t_2`)` in T. Then, as in the case of Prescient Transistor Rule 1, `use(`o_1`)` will precede `assign(`o_2`)` in T'.

Prescient Transistor Rule 3 H contains the following sub-sequence in the program order of thread p:

$$o_1 = \text{READ}^{\swarrow}\ \text{X},\ t_1 = \text{WRITE}_{\searrow}\text{Y},\ o_2 = \text{WRITE Y}.$$

Since the value written by t_1 is seen in another thread, H must contain `store(`t_1`)`. There are two cases involving the `store`:

- `store(`t_1`)` follows `use(`o_1`)` in T. By Constraint A.7.4, `store(`o_2`)`, if it exists, or any `store x` corresponding to an operation following o_2, must be scheduled after `store(`t_1`)`. Therefore, the algorithm would not move `assign(`o_2`)` to a location preceding `use(`o_1`)`. Thus, `use(`o_1`)` must precede `assign(`o_2`)` in T'.
- `store(`t_1`)` is moved presciently to a location preceding `use(`o_1`)` in T. If `store(`o_2`)` or a `store y` of any operation succeeding o_2 is moved to a location preceding `assign(`t_1`)`, then `store(`t_1`)` violates Constraint A.7.4. Therefore, `store(`o_2`)`, if it exists, or any `store y` corresponding to an operation following o_2 in p, must be scheduled in T after `assign(`t_1`)`. Therefore, the transformation would not move `assign(`o_2`)` to a location preceding `use(`o_1`)` in T'. We thus conclude that `use(`o_1`)` must precede `assign(`o_2`)` in T'.

Direction 2 *Java^N is not weaker than Java. For every program P, for every execution H of P, if H is valid under Java^N, then H is also valid under Java.*

If H is valid under Java^N, there exists a legal serialization S of H consistent with CR_{PS}^{T}. To show that H is valid under Java, we must construct a schedule T of H consistent with the Java constraints. We construct T by the following algorithm.

Algorithm $A_2^{5.1}$.

1. Extract from S a sub-serialization S', containing only the WRITE and READ$^{\swarrow}$ operations.
2. Define the orders of operations in T in the following way:

   Convert the READ/WRITE operations in S' to the corresponding `read/write` instructions, and use them as the order of operations in the main memory, denoted as O_{rw}. For each thread p, denote as O_{rw}^{p} the sub-sequence of `read/write` operations in O_{rw} performed on its behalf. For each thread p, convert the READ/WRITE operations in the local history of p in H to a sequence of `use/assign` operations, denoted as O_{ua}^{p}.
3. For every thread, merge the sub-sequence of `read/write` operations into the sequence of `uses/assigns`, as follows.

For each thread p, let O^p be O^p_{ua}. Iteratively, for each operation o in O^p_{rw}, and according to their order in O^p_{rw}:

- [1] *If o is a* `read x`*:* Let o' be the latest `use/assign` operation accessing x in O^p that precedes `use(`o`)`.
  - **[1.1]** If o' is a `use`, insert a `load(`o`)` to O^p after whichever came last: o' or the last inserted `read/write` operation.
  - **[1.2]** If o' is an `assign`, insert `load(`o`)` to O^p after whichever came last: o' or the last inserted `read/write` operation.
  - **[1.3]** If there is no such operation, insert `load(`o`)` to O^p after the last inserted `read/write` operation.
  - **[1.4]** Insert o to O^p just after the last inserted `read/write` operation.
- [2] *If o is a* `write x`*:* Let o' be the latest `use/assign` operation that accesses x and precedes `assign(`o`)` in O^p, and that has a corresponding operation in O_{rw}.
  - **[2.1]** If o' is a `use`, insert a `store(`o`)` to O^p after `load(`o'`)`.
  - **[2.2]** If o' is an `assign`, insert a `store(`o`)` to O^p after o'.
  - **[2.3]** If there is no such operation, insert a `store(`o`)` to O^p at the beginning of O^p.
  - **[2.4]** Insert o to O^p after whichever came last: `store(`o`)` or the last inserted `read/write` operation.

4. Since the order of `read/write` operations in each O^p is the same as in O_{rw} (as is shown below), we can now merge all O^p into the global schedule T according to their interleaving in O_{rw}.

Remark 1: Note that the insertion process in [1.1] and [1.2] is the same. We distinguish between the two cases for deductive purposes, as they are used differently in the proof below.

Remark 2: The `load(`o'`)` that is used in applying [2.1] always exists, for the following reason. For o' we select an operation that has a corresponding operation in O_{rw}. Therefore, during the processing of o', case [1.1] was applied to it, and `load(`o'`)` must have been generated. By the selection of o', the `use(`o'`)` precedes `assign(`o`)`. Since o and o' access the same variable, the Same Variable Rule applies to them, and therefore `read(`o'`)` precedes `write(`o`)`. Therefore, at the time o was processed, o' has been processed already, and the `load` has been inserted.

Remark 3: In our interpretation, the compiler first generates a schedule according to the constraints, excluding the prescient stores optimization, and then optimizes the schedule by making some stores prescient. As will be shown later, all the `store`s are prescient in the constructed schedule T. Thus, in order to verify compliance of `store`s to the non-prescient constraints, we must follow them back to their location just before they became prescient. We call this the *non-prescient location* of the `store`, as opposed to its later *prescient location* in T. There may be several possibilities for the non-prescient location of a `store` which is currently scheduled presciently. In order to keep things simple, in the proof below we define the non-prescient location of a prescient `store x,v` as the slot immediately following its `assign x,v`.

Before we start checking constraints in T, we show the following properties of the constructed order O^p.

1. *The* `read/write` *operations appear in O^p in the same order as in O_{rw}.*

   Proof: Follows immediately from steps 1.4 and 2.4, since each `read/write` operation is inserted after the latest `read/write` operation that was inserted previously. ∎

2. *The* `use/assign` *operations appear in O^p in the same order as in O_{ua}.*

Proof: Follows immediately from the algorithm, since the order of operations of each thread O^p begins with O_{ua}, and other operations are inserted into it without changing the orders of the existing ones. ∎

3. *Each* `read` *operation appears in* O^p *before its corresponding* `load` *.*

   Proof: By the algorithm, `load x,v` is always inserted after the last `read/write` instruction in O^p, and then `read x,v` is inserted immediately after the last `read/write`. Therefore, `read x,v` always precedes `load x,v`. ∎

4. *Each* `write` *operation appears after its corresponding* `store` *in* O^p*.*

   Proof: Follows immediately from the algorithm, since `write x,v` is explicitly inserted after `store x,v`. ∎

5. *Each* `load` *operation appears before its corresponding* `use` *in* O^p*.*

   Proof: Assume that there exists a `use` operation whose `load` is inserted to O^p in a location following the `use`. Consider the first such `use x,v` operation `use(`o`)`.

   `load(`o`)` was inserted in a location that follows the latest `use x/assign x` operation preceding `use(`o`)` in O^p, and follows the latest `read/write` operation in O^p. Because of the first condition, `load(`o`)` cannot appear after `use(`o`)`. Therefore, when `load(`o`)` is inserted, there must exist a `read/write` operation that appears in O^p after `use(`o`)`. Consider the first operation o' in O_{rw} such that `read(`o'`)/write(`o'`)` was already inserted to O^p after `use(`o`)`. Since `read(`o'`)/write(`o'`)` was inserted when o was processed, `read(`o'`)/write(`o'`)` must precede `read(`o`)` in O_{rw}.

   If o' accesses x: Since S' complies with the Same Variable Rule, and the order of `read/write`s in O_{rw} is the same as the order of READ/WRITEs in S', o' must precede o in H. By the construction, this implies that `use(`o'`)/assign(`o'`)` must precede `use(`o`)` in O^p.

   If o' is a READ: since `use(`o'`)` precedes `use(`o`)`, and o is the first operation such that `use` o precedes `load` o, `load(`o'`)` precedes `use(`o'`)` in O^p. By Property 3, `read(`o'`)` appears in O^p before `load(`o'`)`. By transitivity, `read(`o'`)` precedes `use(`o`)` in O^p.

   If o' is a WRITE: since `write(`o'`)` is the first `read/write` that appears after `use(`o`)`, [2.4] implies that `store(`o'`)` must appear after `use(`o`)`. [2.2] and [2.3] would insert `store(`o'`)` before `assign(`o'`)`, and therefore, before `use(`o`)`. The only remaining possibility is that `store(`o'`)` has been inserted by [2.1], and there was another operation o'' accessing x such that at the time READ o' was processed, `load(`o''`)` had already been inserted to O^p at a location following `use(`o`)`.

   Since `load(`o''`)` was already inserted when `write(`o'`)` was processed, `read(`o''`)` must precede `write(`o'`)` in O_{rw}. Since O_{rw} obeys the Same Variable Rule, `use(`o''`)` must precede `assign(`o'`)`. By transitivity, `use(`o''`)` precedes `use(`o`)` in O_{ua}. Since `load(`o''`)` appears in O^p after `use(`o`)`, o'' is also an operation for which the `use` precedes the `load`, in contradiction to the assumption that o is the first such operation.

   If o' accesses another variable y:

   If o' is a READ, then `read(`o'`)` would be placed immediately after the last `read/write` operation o'' in O^p. Since o' is the first operation such that `read(`o'`)/write(`o'`)` follows `use(`o`)`, `read(`o''`)/write(`o''`)` precedes `use(`o`)`. By the algorithm, `read(`o'`)` would also be placed before `use(`o`)` — a contradiction.

   If o' is a WRITE, `write(`o'`)` can be placed after `use(`o`)` only if `store(`o'`)` has also been placed after `use(`o`)`. `store(`o'`)` could be inserted by one of [2.1], [2.2], [2.3].

- *If* `store(`o'`)` *was inserted by [2.1]*, there must be a READ$^{\swarrow}$ Y operation o'' such that `load(`o''`)` has already been inserted after `use(`o`)`. Since o' is the first instruction for which the `read/write` is placed after `use(`o`)`, at the time `load(`o''`)` was inserted, there was no `read/write` instruction after `use(`o`)`. Since `load(`o''`)` was inserted after `use(`o`)`, there should be either a `use y` or a `assign y` instruction o''' between `use(`o`)` and `use(`o''`)`. In the first case, `load(`o''`)` must have been inserted by [1.1], and in the second — by [1.2].
  - `load(`o''`)` *was inserted by [1.1]*: o is a READ$^{\swarrow}$ X since, by the algorithm, only READ$^{\swarrow}$ instructions have a `load` in O^p.
    o''' is a READ Y that follows o in H.
    o'' is a READ$^{\swarrow}$ Y since it has a `load` in O^p. Since `load(`o''`)` was inserted after `use(`o'''`)`, `use(`o''`)` must follow `use(`o'''`)`. Therefore, o''' precedes o'' in H.
    o' is a WRITE Y. Since `store(`o'`)` was inserted after `load(`o''`)`, by the algorithm, this implies that `use(`o''`)` precedes `assign(`o`)`. Therefore, o'' precedes o' in H.
    We conclude that H has the following sub-sequence: o=READ$^{\swarrow}$ X, READ Y, READ$^{\swarrow}$ Y, o'=WRITE Y. By Prescient Transistor Rule 1, o must precede o' in S, and therefore `read(`o`)` must precede `write(`o'`)` in O_{rw} — a contradiction.
  - `load(`o''`)` *was inserted by [1.2]*: As in the previous case, H has the following sub-sequence: o=READ$^{\swarrow}$ X, WRITE Y, READ$^{\swarrow}$ Y, o'=WRITE Y. By Prescient Transistor Rule 2, o must precede o' in S, and therefore `read(`o`)` must precede `write(`o'`)` in O_{rw} — a contradiction.
- *If* `store(`o'`)` *was inserted by [2.2]*, there must be a WRITE Y operation o'' such that `assign(`o''`)` follows `use(`o`)` in O^p. By the algorithm, if `store(`o'`)` was inserted after `assign(`o''`)`, then there exists `write(`o''`)` in O_{rw}. By the algorithm, `write(`o''`)` will be created if o'' is a WRITE$_{\searrow}$, or if it is immediately followed by a READ$^{\swarrow}$ Y.
  In the first case, H includes the following sub-sequence: o=READ$^{\swarrow}$ X, WRITE$_{\searrow}$Y, o'=WRITE Y. By Prescient Transistor Rule 3, o must precede o' in H, and therefore, by the algorithm, `read(`o`)` must precede `write(`o'`)` in O_{rw} — a contradiction.

∎

6. *Each* `store x,v` *operation s appears in O^p before its corresponding* `assign x,v` *operation a.*
   Proof: If s is inserted by part [2.1] of the algorithm, it is placed after a `load` of a preceding `use x` operation u. By Property 5, the `load` appears in O^p before u. Therefore, s is inserted before a.
   If s is inserted by [2.2], it is placed immediately after some `assign x` preceding a. Therefore, s also appears before a.
   If a `store x` is inserted by [2.3], it is placed at the very beginning of O^p, which is clearly before a. ∎
7. *No* `load x,w`*,* `store x,w` *or* `assign x` *in O^p can intervene between* `load x,v` *and* `use x,v`*.*
   Proof: If an `assign x,w` operation precedes `use x,v` in O^p, the `load x,v` will not be placed before it, because of [1.2]. The `load`s of preceding `use x` operations are placed in O^p before their `use`s, by Property 5. The `load`s of subsequent `use x`s are not inserted before `use x,v` because of [1.1].
   A `store x,w` can be inserted between the `load x,v` and `use x,v` only if there is already a `load x,w'` ([2.1]) or an `assign x,w'` ([2.2]) between them. Since there is no intervening `load x`/`assign x`, no `store x` can intervene either. ∎
8. *No* `load x,w`*,* `store x,w` *or* `assign x,w` *in O^p intervenes between a* `assign x,v` *and its corresponding* `use x,v`*.*

Proof: Similar to Property 7 above. ■

9. *A* `store x` *intervenes in* O^p *between an* `assign x,v`$_1$ *and a subsequent* `load x,v`$_2$.

   Proof: By the algorithm, a `store x,v` is inserted to O^p for each `assign x,v`. The `store` is inserted before the `assign`, as a prescient store. However, its non-prescient location is after the `assign`, and therefore it belongs between `assign x,v`$_1$ and `load x,v`$_2$. ■

10. *An* `assign x` *must intervene between a* `load x` *or a* `store x` *and a subsequent* `store x,v`.

    Proof: If there is a `store x,v` in O^p, then by Property 6 it is inserted before its corresponding `assign x,v`. For any preceding `load x`/`store x`, the `assign x,v` appears between it and the non-prescient location of `store x,v`. ■

11. *Each* `store x,v` *operation* s *is in a location in* O^p *which is permitted by prescient constraints.*

    Proof: By Property 6, `store x,v` is inserted before its corresponding `assign x,v` operation a.

    The prescient constraints require that no other `load x`/`store x` appear between s and a. We show that when s is inserted to O^p, there is no intervening `load x`/`store x` operation between it and a, and no `load x`/`store x` is inserted between them later.

    From [2.1] and [2.2], s is inserted either after the latest preceding `load x` or after the latest preceding `assign x` which has a corresponding `store`. In both cases, it is explicitly inserted after the preceding `load`/`store`.

    For every `assign x` operation following a, the corresponding `store` would be inserted after a by [2.2]. For every `use x` operation following a, the corresponding `load` would be inserted after a because of [1.2]. ■

12. *For any variable* x*, the* `load`/`store` *operations in* O^p *appear in the same order as the corresponding* `read`/`write` *operations in* O_{rw}.

    Proof: Since S obeys the Same Variable Rule, the order of READ X/WRITE X operations in O^p_{rw} is the same as the order of `use x`/`assign x` operations in O^p_{ua}.

    A `load x` is inserted after the most recent `use x`/`assign x`. By Properties 5 and 6, it follows any `load x`/`store x` of a preceding `use x`/`assign x`.

    A `store x` is inserted after the `load`/`store` of the most recent `use x`/`assign x` operation that has a `load`/`store`. Therefore, a `store x` is always inserted after any preceding `load x`/`store x`. ■

Now we can verify that all the Java constraints hold for the constructed execution. The relevant constraints are: A.2, A.3, A.4 and A.7.

A.2.1 For the register property, we need to show that each `use x,v`/`store x,v` operation o_2 appears in O^p after the `assign x,v`/`load x,v` operation o_1, and that no other `use x,w`/`assign x,w`/`load x,w`/`store x,w` appears between them.

- *If* o_1 *is an* `assign` *and* o_2 *is a* `use`, then, since S is a legal serialization, o_1 precedes o_2 in S, and no other READ X,W/WRITE X,W instruction appears between them.
  By the Same Variable Rule, o_1 precedes o_2 in H. By the algorithm, o_1 must precede o_2 in O_{ua}. From Property 8, no other instruction can intervene in O^p between o_1 and o_2.
- *If* o_1 *is a* `load` *and* o_2 *is a* `use`, then o_1 precedes o_2 by Item 5. By Property 7, no other instruction accessing x can intervene between them.
- *If* o_1 *is a* `assign` *and* o_2 *is a* `store`, the non-prescient location for the `store` is immediately after the `assign`, as discussed before, and therefore, no other instruction accessing x can intervene between them.

- *If* o_1 *is a* `load` *and* o_2 *is a* `store`, then o_1 must yield a result of some preceding `assign x`. The `assign x` that supplies the value for o_2 follows o_2 in O^p. Therefore, there must be two `assign`s that produce the same value, which contradicts our assumption that a `load` can always identify the value of the `store`.

A.2.2 Follows from Property 1.

A.2.3 The synchronization operations are not considered in this work.

A.2.4 The constructed schedule T is a result of a merge of several serializations. Therefore it does not contain cycles.

A.3.1 Follows from Property 2.

A.3.2 Follows from Property 9.

A.3.3 Follows from Property 10.

A.3.4 If the first operation in O_{ua} for some variable x begins with an `assign`, then that `assign` will be the first operation on x in O^p. This is because `load x` operations will not be inserted before it, and `store` operations should be thought of as scheduled in their non-prescient locations, right after their corresponding `assign`s.

If the first operation in O_{ua} for x begins with a `use`, then, since S is a legal serialization, the `use` necessarily sees a result written in another thread. By the algorithm, a `load` operation would be generated for the `use`, and inserted before it.

A.4.1 Follows from Property 3.

A.4.2 Follows from Property 4.

A.4.3 Follows from Property 12.

A.7.3 and A.7.4 These constraints are shown in Property 11.

■

6 Volatile Variables and Locks

Here we consider the consistency guarantees when strong operations are employed, including the use of locks (Constraints in A.5) and volatile variables (Constraints in A.6). The resulting memory model is called *Java* in this section (Note this is the same name as that used in previous sections for the unsynchronized model). Although we do not give precise characterizations for Java, we prove some useful relations with traditional models.

The section is organized as follows. In Section 6.1 we consider how the use of locks and volatile variables by the programmer affects the memory model. In Section 6.2 we consider the requirements on the implementation in order for the operations to work correctly.

6.1 Programmer View

6.1.1 Locks

Locks in the Java language have two purposes. First, they synchronize the flow of control between threads. Second, they synchronize memory views between different threads. The corresponding constraints are given in Appendix A.5.

In the bytecodes, the locks are implemented by `lock` and `unlock` instructions. However, in the Java programming language there are no explicit `lock` and `unlock` operations. Instead, fragments of the source code may be marked as *synchronized*, implying the `lock` instruction at the beginning of the sequence and `unlock` when it terminates. Thus, `lock` and `unlock` operations in Java always come in pairs.

Although the Java language defines correspondence between objects and locks at the level of the source code, there is no such correspondence at the bytecode level. Therefore, `lock` and `unlock` operations synchronize all the variables, and not only those stored in the object whose method was called.

We compare Java to release consistency (RC) [6], which is defined by distinguishing between two classes of operations: regular and special. Special operations are ACQUIRE, which is similar to `lock` in Java, and RELEASE, which is similar to `unlock` in Java. Java is different from release consistency in that it associates a lock with each object, while in the release consistency there is only one global lock.

Another difference between Java and RC is that the latter does not specify how the updates of variables propagate from one thread to another when no thread enters or leaves a synchronized code section. For instance, it is valid to implement RC with no updates whatsoever, except at synchronization points. In contrast, in Java the updates still follow the constraints as discussed throughout the paper. Thus, on the one hand, Java can be stronger than certain implementations of RC, as it enforces more constraints on the execution. On the other hand, other implementations of RC may be stronger than Java. Such implementations may restrict the propagation of updates to certain points during run-time. Java constraints, although not generally fulfilled, may turn out to hold at these specific points.

Release Consistency: The requirements for RC are [6]:

1. Before an ordinary access to a shared variable is performed, all previous ACQUIREs performed by the thread must have already completed.
2. Before a RELEASE can be performed, all previous `read`s and `write`s performed by the thread must have already completed.
3. The ACQUIRE and RELEASE accesses must be processor consistent (see Section 2.3), with the corresponding memory model denoted RC_{PC}, or sequentially consistent, in which case the corresponding memory model is denoted RC_{SC} [16]. Sequential Consistency is defined in Section 6.1.2 below.

The JLS states that "With respect to a lock, the `lock`/`unlock` operations are performed in some sequential order consistent with the order on the actions of each thread." This implies that the consistency model for `lock` operations in Java is Coherence. It can thus be shown that programs with a single lock follow the RC_{SC} model. In order to avoid the single-lock restriction, and since the model presented by the locks is weaker than SC, we redefine RC for multiple locks by rewriting the third requirement as follows:

3. The ACQUIRE and RELEASE accesses must be coherent with respect to any one lock. The corresponding memory model is denoted RC_{Co}.

The following theorem states that some legal implementations of RC_{Co} are not stronger than Java. This implies that Java can "simulate" RC_{Co}, and that a program written for RC_{Co} may execute correctly using Java programming and run-time environment.

Theorem 6.1 *Java operations can be used to implement RC_{Co}.*

Proof: The implementation maps the special operations one to one: ACQUIRE to `lock` and RELEASE to `unlock`. The implementation uses Java constraints to govern the shared variable updates.

To prove the theorem we must show that the Java constraints imply those of RC_{Co}.

The first condition states that an ACQUIRE succeeds in RC_{Co} only if all the previous allocations of the lock (initiated by the ACQUIRE operations) are freed by matching RELEASE operations. This is obviously preserved in Java (for `lock` and `unlock` operations), as required by Constraint A.5.1.

The second condition is also satisfied: all the `use`s (regarded as `read`s in the RC_{Co} definition) which appear before the `unlock` instruction have their corresponding `read` operations performed before the `unlock` instruction is issued, and thus before the `unlock` operation is performed by the main memory. (We may regard `use`/`assign`s that do not have corresponding `load`/`store`s as if they are completed instantly.) The `write` operations (corresponding to the `assign`s) which appear before the `unlock` instruction are performed before the `unlock` in the main memory, according to Constraint A.5.4.

∎

In the other direction, some implementations of RC_{Co} may be shown to be stronger than Java. This means that they can be used as programming and run-time environments for Java programs.

Theorem 6.2 *There exists a legal implementation of RC_{Co} which is stronger than Java.*

Proof sketch: Consider an implementation of RC_{Co} in which all local updates since the last RELEASE are sent to main memory upon execution of a new RELEASE operation. Nothing, however, is sent in between. The RELEASE is not allowed to proceed until all data has been modified in the main memory. Similarly, all updates to the main memory since the last ACQUIRE are fetched upon execution of a new ACQUIRE operation. The ACQUIRE is not allowed to proceed until all modifications have been fetched from the main memory and locally updated. It is easy to see that the above implements RC_{Co} correctly.

The crux of the proof (not provided here) is the following observation: since no updates are sent between synchronization points, the transistor rules for shared variable updates are not applicable to this implementation. Local ordering constraints are naturally obeyed as all updates of a synchronization-free segment of a local history arrive to the main memory at the same time.

6.1.2 Volatile Variables

There may be several variables in the program that are heavily utilized for transferring data between threads. In this case it is neither convenient nor efficient to use locks for each access to these variables, since locks impose unnecessary overheads. This is where *volatile* variables are useful, as defined by the constraints in Appendix A.6.

We examine the behavior of code that employs volatile variables only, and dub the corresponding memory model *volatile consistency.*

Sequential consistency (SC) is defined as follows [13].

> **Sequential Consistency:** An execution H is said to be *sequentially consistent* if there is a legal serialization S of H such that if o_1 and o_2 are two operations in the same local history of H, and $o_1 \xrightarrow{po} o_2$, then $o_1 \xrightarrow{S} o_2$. In other words, there is a serialization of H which is consistent with the views of all the threads. The memory model which consists of all sequentially consistent executions is called Sequential Consistency.

Theorem 6.3 *The consistency model among volatile variables, namely, Volatile Consistency, is equal to Sequential Consistency.*

Proof: Direction 1: *Volatile Consistency is not weaker than SC.* Given an execution H and a schedule T, we show that there is a serialization of accesses to volatile variables which is consistent with the program order of all the threads. Let $H|v$ denote an execution consisting only of the operations in H on volatile variables.

Consider the main memory part $T|v$ in T consisting only of accesses to volatile variables. From the constraints A.6.1 and A.6.2, there is a one-to-one correspondence between the operations in $T|v$ and those in $H|v$. Furthermore, from Constraint A.6.3, the order of the accesses is the same for the `use`s and `assign`s executed by a given thread, and for the `read`s and `write`s performed by the main memory on its behalf, respectively. Thus, if we show that there exists a legal serialization of $T|v$, it will induce a similar serialization on $H|v$. This in turn implies SC for accesses to volatile variables.

It can be verified that for volatile variables the following set of constraints subsumes all other relevant constraints:

1. There exists a total order of main memory accesses to any given variable. In this order, a `write` must precede `read`s that yield the written value (Constraint A.2.2).
2. There exists a total order of main memory accesses performed on behalf of any given thread. This order is compatible with the program order of the thread (Constraint A.6.3).
3. There cannot be a set of dependencies between operations in which an operation is (transitively) dependent on itself (Constraint A.2.4).

The actual memory accesses in $T|v$ comply with all the constraints which imply acyclic ordering relations between the instructions. We thus construct the required serialization by a topological sort of $T|v$. By the constraints, this serialization is valid; hence, the corresponding serialization on $H|v$ is also valid, and is consistent with the program orders of all the threads.

We conclude that the interaction between volatile variables is at least as strong as Sequential Consistency.

We remark that this direction still holds when H contains accesses to both volatile and regular variables. This is because the accesses to regular variables can only add constraints for the serialization of volatile variables. Thus, their consistency model is equal to or stronger than volatile consistency.

Direction 2: *SC is not weaker than volatile consistency.* Consider a sequentially consistent execution H. It is consistent with the program orders of all the threads; therefore, all the main memory accesses are in the same order as the READ and WRITE operations. This implies that H does not violate the volatile variables' constraints, and is thus valid under Java. This is true even when all variables are volatile. ∎

6.2 Implementor View

We now examine the behavior of volatile variables and locks in Java_{VM}.

6.2.1 Locks

Once again, the JVM depends on the compiler to preserve the locks' semantics. In order for a `write` operation to terminate before an `unlock`, the compiler must place the corresponding `store` before the unlock. The same holds for the `read`s and their corresponding `store`s.

Thus, the implementation requires that if there is a `lock` instruction followed by a `load` instruction, the main memory should preserve the same order of their corresponding `lock` and `read`.

In addition, if there is a `store` instruction followed by an `unlock` instruction, their corresponding operations in the main memory should be performed in the same order.

Because no other instruction can bypass a `lock` or `unlock`, the order of `lock`s and `unlock`s performed by the thread is compatible with its program order. We conclude that the consistency model for the `lock/unlock` operations is Coherence as in the programmer view.

6.2.2 Volatile Variables

As with regular variables, the JVM executes the bytecodes with no indication as to the original order of operations in the source code. Thus the JVM must assume that the compiler preserves the order of `load`s and `store`s, and the order of `use`s and `assign`s, according to the JLS constraints. Now, because for volatile variables the order of `use`s and `assign`s is thus equal to the order of `load`s and `store`s performed by the thread, the same reasoning as in Section 6.1.2 holds, and Volatile Consistency in Java_{VM} is equal to SC.

Note that the bytecodes do not distinguish between operations on the volatile and regular variables. However, this information is contained in the class headers, and can be accessed during class loading time. Therefore, it is reasonable that the volatile variables and the regular variables will be mapped to different memory segments which implement different consistency protocols. This may ease the implementation in some cases, but should be done with care when the same object contains both volatile and regular variables.

7 Conclusions

In this work we gave exact non-operational characterizations of Java memory behavior, excluding synchronization constructs. We provided both the implementor and the programmer with non-operational characterizations that are simpler to understand than the original, operational definitions, and can be implemented more freely. Our distillation of the original operational memory model is independent of a specific machine. It is highly useful for formal verification. Indeed, our initial results led to the detection of incorrect compiler implementations (Javasoft Bug #4242244).

To the best of our knowledge, this is the first work that provides the JVM with its own memory model (the implementor view), implicitly embedded in the JLS specification of the programmer view. Although the implementor view is our own interpretation of the JLS, it is supported by several remarks in that document (and does not affect our results concerning the programmer view). We believe it to be unreasonable that such an important aspect of the JVM, which is supposed to provide a standard virtual machine across all architectures, will be left unspecified (or at least not explicitly specified). In support of this claim, consider that in our model the implementor view is shown to be strictly stronger than the programmer view. This was effectively impossible to see using the original definition in JLS.

We gave separate characterizations of the programmer memory model in the case when the compiler optimization known as prescient stores is not applicable, as well as in the case when it can be applied for all writes in the program. In this way we bound the power of this optimization in relaxing the consistency of shared variables.

One observation that evolves out of our non-operational approach concern the role of Constraint A.3.2. We conjecture that when prescient stores can be applied to all writes Constraint A.3.2 is redundant in the specification; it doesn't affect the consistency model.

Two interesting questions remain open, and we intend to address them in our future research. First, we conjecture that verifying compliance to Java consistency is NP-Complete, in either the programmer view or the implementor view. Second, we think that it is possible to add the synchronization operations to our non-operational setup. In this way, we hope to come up with a model which is unified and complete, yet not too complicated. The idea is to add transistor-like rules from/to `read`s/`write`s to/from `lock`s/`unlock`s. The proofs, however, do not seem so easy to obtain.

A The Java Memory Model Specification

For the reader's convenience, we present the operational Java memory model specification as defined in [9] (JLS). The specification consists of two parts: the set of operations in the model, and the set of constraints on those operations.

For simplicity, in the specifications below, V and W always denote variables, T always denotes a thread, and L always denotes a lock.

A.1 Operations

A single Java thread issues a stream of `use`, `assign`, `lock` and `unlock` instructions, according to the program source code. The underlying Java implementation is then required to perform appropriate `load`, `store`, `read` and `write` instructions, according to the Java constraints. The semantics of these operations are as follows:

- A `use` operation transfers the value of the variable from the thread's local copy into the execution engine.
- An `assign` operation transfers the value of the variable from the execution engine into the local copy.
- A `load` operation transfers the value of the variable from the main memory (which was read by the preceding `read` operation) to the local copy.
- A `store` operation transfers the value of the variable stored in the local copy of the thread to the main memory, to be written to the master copy of the variable by the `write` operation.
- A `read` operation transmits the value of the master copy of the variable to the thread for the use of a `load` operation.
- A `write` operation writes the value of the variable transferred by the `store` operation to the master copy in the main memory.

A.2 General Constraints

1. The operations performed by any one thread are totally ordered. A `use x` or a `store x` in one of the local histories always uses the most recent value that was given to x by an `assign` or a `load` operation in that order.[2]
2. The operations performed by the main memory for any one variable are totally ordered. A `read` in the order of one of the variables always yields the value that was written by the last `write` in that order.[3] If there was no preceding `write` in the order, the value yielded by the `read` is some initialization value.

[2] The "register property" here follows implicitly from JLS [9].

[3] The semantics of the main memory, implying that a `read` from a master copy of a variable always returns the value stored there by the most recent `write` to the same variable, follows from several remarks in the JLS [9].

3. The operations performed by the main memory for any one lock are totally ordered.
4. It is not permitted for an instruction to follow itself.

A.3 Constraints inside a Thread

1. A `use` or `assign` of V is permitted only when dictated by the execution of T.
2. A `store` of V by T must intervene between an `assign` of V by T and a subsequent `load` of V by T.
3. An `assign` of V by T must intervene between a `load` or a `store` of V by T and a subsequent `store` of V by T.
4. A variable is said to be *new* when it it used by a thread for the first time, or when it is created by a thread. For each new variable, `assign` or `load` must be performed on it previous to any `use` or `store`.

A.4 Constraints between a Thread and the Main Memory

1. For every `load` performed by T on its working copy of V, there should be a corresponding preceding `read` by the main memory on the master copy of V.
2. For every `store` performed by T on its working copy of V, there must be a corresponding subsequent `write` by the main memory.
3. Let A be a `load` or a `store` of V by T, and let P be the corresponding `read` or `write` by the main memory. Let B and Q be two other such operations by T and the main memory (on V), correspondingly. Now, if A precedes B, then P precedes Q.

A.5 Locks

1. A `lock` of L by T may occur only if for every other thread the number of preceding `unlock`s equals the number of preceding `lock`s.
2. An `unlock` of L by T may occur only if the number of preceding `unlock`s of L by T is strictly less than the number of preceding `lock`s.
3. `lock`s and `unlock`s of a lock L are performed in some sequential order which is consistent with the program order of all the threads.
4. A `store` must intervene between an `assign` of V by T and a subsequent `unlock` of L by T, and the `write` which corresponds to the `store` must occur before the `unlock` by the main memory.
5. Between a `lock` of L by T and a subsequent `use` or `store` of V by T, an `assign` or `load` of V must appear. If what appears is a `load`, then its corresponding `write` should appear before the `lock` by the main memory.

A.6 Volatile Variables

1. A `use` of V by T is permitted only if the previous access to V by T was a `load`, and a `load` is permitted only if the next access to V by T is a `use`.
2. A `store` of V by T is permitted only if the previous access to V by T was an `assign`, and an `assign` is permitted only if the next access to V by T is a `store`.

3. Let A denote a `use` or an `assign` of V by T, and let F denote the corresponding `load` or `store`, and P denote the `read` or `write` corresponding to F. Similarly, let B denote a `use` or an `assign` of W by T, and let G denote the corresponding `load` or `store`, and Q denote the `read` or `write` corresponding to G. Then if A precedes B, P must precede Q.

A.7 Prescient Stores

If a `store` of a non-volatile variable V by T follows an `assign` of V by T without any intervening `load` or `assign` to V by T, then the `store` can occur before the `assign` provided the following conditions hold:

1. If the `store` occurs, then the `assign` is bound to occur.
2. No `lock` action intervenes between the relocated `store` and the `assign`.
3. No `load` of V intervenes between the relocated `store` and the `assign`.
4. No other `store` of V intervenes between the relocated `store` and the `assign`.
5. The `store` writes the value yielded by the `assign`.

References

[1] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The Power of Processor Consistency. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures (SPAA'93)*, 1993.

[2] I. Attali, D. Caromel, and M. Russo. A Formal Executable Semantics for Java. In *OOPSLA Workshop on the Formal Underpinnings of Java*, 1998.

[3] E. Börger and W. Schulte. *A programmer friendly modular definition of the semantics of Java.* Springer, 1998. In J. Alves-Foss Ed., Formal Syntax and Semantics of Java.

[4] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. *From Sequential to Multi-threaded Java: An Event-based Operational Semantics.* Springer, 1997. In M. Johnson Ed., Algebraic Methodology and Software Technology.

[5] E. Coscia and G. Reggio. A proposal for a semantics of a subset of multi-threaded "good" Java Programs. In *OOPSLA Workshop on the Formal Underpinnings of Java*, 1998.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. 17th Int'l ACM Symp. on Computer Architecture*, pages 15–26, 1990.

[7] A. Gontmakher and A. Schuster. Characterizations for Java Memory Behavior. In *IPPS/SPDP*, pages 682–686, March 1998.

[8] J. R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.

[9] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[10] Y. Gurevich, W. Schulte, and C. Wallace. Investigating Java Concurrency using Abstract State Machines. 1999.

[11] P. W. Hutto and M. Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. In *Proc. of the 10th Int'l Conf. on Distributed Computing Systems (ICDCS-10)*, pages 302–311, May 1990.

[12] Peer Review for Java Language Specification, 2nd Edition. `http://java.sun.com/aboutJava/communityprocess/maintenance/JLS/index.html`.

[13] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[14] D. Lea. *Concurrent Programming in Java.* Addison-Wesley, 1996.

[15] W. Pugh. Fixing the Java Memory Model. In *Proc. ACM Java Grande Conference*, June 1999.

[16] S.V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, pages 66–76, December 1996.

[17] A. S. Tanenbaum. *Distributed Operating Systems.* Prentice-Hall, Inc., 1995.

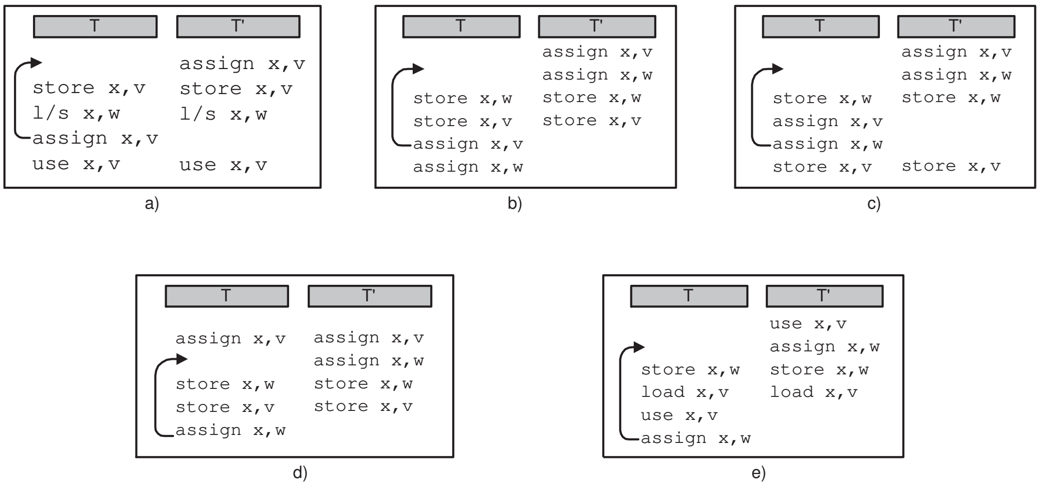

Figure 15: Execution valid for Coherence and invalid for $\mathrm{Java}_{\mathrm{PS}}$.