

## References

- [1] The Millipede Project Home Page. <http://www.cs.technion.ac.il/Labs/Millipede>.
- [2] 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.
- [3] H. Attiya and J.L. Welch. Sequential Consistency versus Linearizability. In *ACM Transactions on Computer Systems*, May 1994.
- [4] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *COMPCON*, pages 528–537. IEEE, 1993.
- [5] 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.
- [6] J. R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.
- [7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- [8] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.
- [9] 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.
- [10] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: a Bridge between Release Consistency and Entry Consistency. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures*. ACM, June 1996.
- [11] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.
- [12] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1996.
- [13] S.V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, pages 66–76, December 1996.
- [14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
- [15] A.S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

### 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 is 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. Each **lock** and **unlock** is performed jointly by some thread and the main memory.
2. 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  $V$ .
3. For every **store** performed by  $T$  on its working copy of  $V$ , there must be a corresponding following **write** by the main memory.
4. 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 **unlocks** equals the number of preceding **locks**.
2. An **unlock** of  $L$  by  $T$  may occur only if the number of preceding **unlocks** of  $L$  by  $T$  is strictly less than the number of preceding **locks**.
3. **Locks** and **unlocks** of  $L$  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  $Q$ .

We mention that this work was carried out in the scope of the Millipede project [1] with the goal of having Java implemented distributively in a correct and efficient way.

## A The Java Memory Model Specification

For completeness we present the operational Java memory model specification as defined in [7] (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 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 program orders always uses the most recent value that was given to  $x$  by an **assign** or a **load** operation in that order.<sup>3</sup>
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.<sup>4</sup> If there was no preceding **write** in the order, the value yielded by the **read** is some initialization value.
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.

---

<sup>3</sup>The “register property” here follows implicitly from JLS [7].

<sup>4</sup>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 [7].

### 5.2.1 Locks

Once again, in order to preserve the locks' semantics, the JVM depends on the compiler. For **write** operations to terminate before the **unlock**, the corresponding **store** instructions must be placed by the compiler before the **unlock** instruction. The same holds for the **reads** and their corresponding **stores**.

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**, and 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 **locks** and **unlocks** performed by the processor is compatible with its program order. As before, the tight coupling of the **lock/unlock** operations performed by the processors and the corresponding **lock/unlock** performed by main memory imply Linearizability. Note this does not necessarily require having a single memory agent: the required consistency can be achieved by applying appropriate protocols between multiple memory agents.

### 5.2.2 Volatile variables

As with regular variables, the JVM executes the bytecodes with no indication regarding the original order of operations in the source code. Thus the JVM must assume that the compiler preserves the order of **loads** and **stores**, and **uses** and **assigns**, according to the JLS constraints. Now, because for volatile variables the order of **uses** and **assigns** is equal to the order of **loads** and **stores** performed by the processor, the same reasoning as in Section 5.1.2 holds, and Volatile Consistency in Java' is equal to SC.

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

## 6 Conclusions

In this work we gave non-operational characterizations of Java memory behavior. In order for her program to work correctly on all JVM implementations that comply with the standard Java definition, the Java programmer relies on memory behavior which is not stronger than the standard definition as given in the JLS [7]. We showed that the programmer can rely on Coherence and some weak variation of Causality for regular variables, on Sequential Consistency for Volatile variables, and on Release Consistency when using the *synchronized* construct.

On the other hand, in order for the implementation to support all bytecode programs that comply with standard Java specifications, the implementor of the Java Virtual Machine must guarantee that it is at least as strong as the original definition. We showed that the original definition of JVM memory behavior in the JLS is some combination variant of Coherency and Causality.

We provided both the implementor and the programmer with two non-operational definitions that are a lot simpler to understand than the original, operational one, and can be implemented more freely. Our results imply that the programmer view is slightly weaker than the implementor view in the causal relations that are imposed.

We examine the behavior of code employing volatile variables only, and dub the corresponding memory model *Volatile Consistency*.

**Theorem 5.2** *The consistency model among volatile variables, namely, Volatile Consistency, is equal to Sequential Consistency.*

The two different directions are proven in the claims below.

**Claim 5.1** *Volatile Consistency is not weaker than SC.*

**Proof** Given an execution  $H$  and a timing  $T$ , we show that there is a serialization of accesses to volatile variables, which is consistent with the program order of all the processors. Let  $H|v$  denote a history consisting only on the operations in  $H$  on volatile variables.

Consider the main memory part  $T|v$  in  $T$  consisting 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 **uses** and **assigns** by a given processor, and for the **reads** and **writes** 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  $H|v$ , which implies SC for accesses to volatile variables.

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

1. There exists a total order of main memory accesses to any given variable, and in this order a **write** must precede **reads** 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 processor, and this order is compatible with the program order of the processor (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 over  $T|v$ . 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 processors.

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

We remark that the claim still holds when  $H$  contains accesses to both volatile and regular variables, as the accesses to regular variables can only add constraints for the serialization of volatile variables, thus making their consistency model equal to or stronger than Volatile Consistency.

**Claim 5.2** *SC is not weaker than Volatile Consistency.*

**Proof** Consider a sequentially consistent history  $H$ . It is consistent with the program orders of all the processors; therefore, all the main memory accesses are in the same order as the **use** and **assign** operations. This implies that  $H$  does not violate the volatile variables constraints, and is thus valid under Java, even when all variables are volatile. ■

## 5.2 Implementor View

We now consider the lower level of Java memory behavior with respect to volatile variables and locks. This constitutes the JVM interface to the applications, and thus is what must be guaranteed by the implementor.

throughout the paper. From this point of view *Java is stronger than RC*, as it enforces more constraints on the execution.

The following theorem shows how code that is written for RC can perform correctly on the Java memory model. In other words, it shows that the programmer can rely on at least Release Consistency when writing in Java.

**Theorem 5.1** *Java can simulate RC by mapping the special operations one to one: ACQUIRE to lock and RELEASE to unlock.*

In the original notation of RC, the operation corresponding to `lock` in Java is called ACQUIRE, and the operation corresponding to `unlock` is called RELEASE, but here we will rename them `lock` and `unlock`, as in Java.

**Release Consistency:** The requirements for RC are [5]:

1. Before an ordinary access to a shared variable is performed, all previous LOCKs done by the processor must have already completed.
2. Before an UNLOCK is allowed to be performed, all previous READs and WRITEs done by the processor must have already completed.
3. The LOCK and UNLOCK accesses must be processor consistent, with the corresponding memory model denoted  $RC_{PC}$ , or sequentially consistent, in which case the corresponding memory model is denoted  $RC_{SC}$  [13].

**Proof** To prove the theorem we must show that any execution possible under Java is also possible under RC, or alternatively, that the Java constraints imply those of RC.

The first condition states that a LOCK succeeds in RC only if all the previous allocations of the lock (initiated by the LOCK operations) are freed by matching UNLOCK operations. This is obviously preserved in Java as required by Constraint A.5.1.

The second condition is also satisfied: all the `uses` (regarded as READs in the RC 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/assigns` that do not have corresponding `load/stores` as if they are completed instantly). The `write` operations (corresponding to the `assigns`) which appear before the `unlock` instruction are performed before the `unlock` in the main memory, according to Constraint A.5.4.

The third condition is also satisfied. The Constraint A.4.1 forces the main memory agent to participate in all the operations on locks. Because Java defines a single main memory agent, this implies that only one such operation can be performed at a time. Hence, the consistency between locks is in fact *Linearizability* [8]. According to [2] and [3], Linearizability is strictly stronger than Processor Consistency or Sequential Consistency, whichever is required for the RC model. Note that this result remains valid even when the actual implementation employs several main memory agents, as the memory behavior will remain unchanged. ■

### 5.1.2 Volatile Variables

Sometimes there are several variables in the program that are heavily used for transferring data between processors. In this case it is not convenient and not efficient to use locks for each access to these variables, since (as explained above) locks preserve Linearizability between them, and this requirement may not always be essential. This is where *volatile* variables are useful, as defined by the constraints in Appendix A.6.

Clearly, in  $T$  there is a Causal Relation from the `load x,v` to the `store`, which is preserved in  $S'$  and  $\hat{S}$ . Hence, the `load x,v` precedes the `store` in  $\hat{S}$ . Now, in the construction of  $\hat{S}$  the `assign x,w` is inserted immediately before the `store` (or in a sequence of `assign`s which are inserted consecutively immediately before the `store`). Thus, it succeeds the `load x,v` in  $\hat{S}$ , which is a contradiction.

- Similarly, if the corresponding operations are `assign` and a `load`, with the `use` corresponding to the `assign` preceding that corresponding to the `load`, this implies on the one hand that the `assign` precedes the `load` in  $\hat{S}$ . On the other hand, we find a matching `store` for the `assign` in  $\hat{S}$ , which, when traced back to  $T$ , and by Constraint A.3.3, the `store` would appear before the `load` in  $T$ . Due to Causal Relation the `store` would also precede the `load` in  $\hat{S}$ ; hence, by the construction, the `assign` will also precede the `load`, a contradiction. ■

## 5 Volatile Variables and Locks

Here we consider the consistency guarantees when strong definitions are employed, including the use of Locks (Constraints in A.5), and volatile variables (Constraints in A.6).

The section is organized as follows. In Section 5.1 we consider the effect of using locks and volatile variables by the programmer. In Section 5.2 we consider the requirements on the implementation in order for the operations to work correctly.

### 5.1 Programmer View

#### 5.1.1 Locks

There are two purposes for the use of locks in the Java language: first, to synchronize the flow of control between processors, and second, to synchronize the views of memory between different processors. 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 appear 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. So, `lock` and `unlock` operations synchronize all the variables, and not only the ones stored in the object whose method was called.

We compare Java to Release Consistency (RC) [5], which is defined by distinguishing between two classes of operations: regular and special. Special operations are: `ACQUIRE` — the same as `lock` in Java, and `RELEASE` — the same as `unlock` in Java. Java is different from the 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 processor to another when no processor enters or leaves a synchronized code section. In particular, 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

**same variable: assign before use.** *The violating pair is an assign  $x$ , denoted  $a$ , which precedes a use  $x$ , denoted  $u$ , in  $L$ .*

If  $u$  was inserted because of  $a$  (they refer to the same value), then by the construction it would appear after  $a$  in  $\hat{S}$ , a contradiction.

If  $u$  was inserted because of some other **assign**, denoted  $a'$ , then for  $u$  to yield the value assigned to  $x$  by  $a'$ ,  $a'$  must appear in  $L$  between  $a$  and  $u$  (Constraint A.3.1). We already saw that the **assigns** keep their program order in  $S$ , and since  $u$  is inserted after  $a'$ , we conclude that  $u$  succeeds  $a$  in  $S$ , a contradiction.

If  $u$  was inserted because of some **load**, denoted  $l$ , then  $l$  must appear between  $a$  and  $u$  in  $T$  (Constraint A.3.1). But then, by Constraint A.3.2, a **store** must intervene between  $a$  and  $l$ . Because the **store** has a Causal Relation to  $l$ , it would precede  $l$  in  $\hat{S}$  as well. Therefore, by the construction,  $a$  (inserted somewhere before the **store**) would precede  $u$  (inserted immediately after  $l$ ) in  $\hat{S}$ .

**same variable: use before assign.** *The violating pair is a use  $x$ , denoted  $u$ , which precedes an assign  $x$ , denoted  $a$ , in  $L$ .*

By Constraint A.3.1,  $u$  and  $a$  cannot refer to the value (unless  $a$  precedes  $u$  in  $L$ ). Thus, there are two cases to consider: either  $u$  was inserted to  $\hat{S}$  because of some other **assign**, or it was inserted because of some **load**.

If  $u$  was inserted because of some other **assign**, denoted  $a'$ , then for  $u$  to yield the value assigned to  $x$  by  $a'$ ,  $a'$  must precede  $u$  in  $L$  (Constraint A.3.1), and thus it also precedes  $a$  in  $L$ . We already saw that the **assigns** maintain their program order in  $S$ , and since  $u$  is inserted immediately after  $a'$ , we conclude that  $u$  precedes  $a$  in  $S$ , a contradiction.

If  $u$  was inserted because of some **load**, denoted  $l$ , then the situation is the same as the one handled in the case of the transistor rule above, and the same reasoning holds.

**same variable: two uses.** *The violating pair involves two use  $x$  instructions.*

If both **uses** yield the same value, then there is no problem. Suppose the **uses** return different values. We consider four cases:

- If the corresponding operations are both **assigns**, then by the arguments in the previous case we know that they maintain their program order in  $S$ . However, since  $S$  is a legal serialization, a **use** returns the value of the most recent **assign/load**. Thus, since the order of **uses** is switched, so is the order of the corresponding **assign/loads**, a contradiction.
- If the corresponding operations are both **loads**, then by Constraint A.3.1 they must appear in  $T$  in the order of the **use** operations in  $L$ . These **loads** are also linked by a Causality Relation, and therefore they keep their order in  $S'$  and in  $\hat{S}$ . However, by the construction, the **use** operations are inserted to  $\hat{S}$  in the same order as the **loads**, and since we assumed the order of the **uses** is switched we get a contradiction.
- Suppose the corresponding operations are **load  $x,v$**  and **assign  $x,w$** , so that **use  $x,v$**  precedes **use  $x,w$**  in  $L$ , and **use  $x,v$**  succeeds **use  $x,w$**  in  $S$ . Since the **uses** are inserted in  $\hat{S}$  immediately after **load/assigns**, we conclude that in  $\hat{S}$  **assign  $x,w$**  precedes **load  $x,v$** .

On the other hand, since **use  $x,v$**  precedes **use  $x,w$**  in  $L$ , by Constraint A.3.1, **load  $x,v$**  precedes **assign  $x,w$**  in  $T$ . Since **assign  $x,w$**  is inserted to  $\hat{S}$  at Stage 1 of the construction, we know that in  $\hat{S}$  there is either a **store  $x,w$**  or a **store  $x,w'$**  for which the corresponding **assign  $x,w'$**  succeeds the **assign  $x,w$** . In any case, tracing the **store** back to  $T$  we conclude that it appears there after the **assign  $x,w$** , and hence after the **load  $x,v$** .

**assign**  $x$  instruction in between. Thus, in  $S$ , the **use** sees the result of the closest preceding **assign** in the serialization.

- *A use  $x$  that was inserted at Stage 3.* By Constraint A.2.1, a **use** instruction yields a value produced by some **load** or some **assign**. By the construction, all **uses** are inserted into  $\hat{S}$  during Stage 2, unless there is no corresponding **load** and the corresponding **assign** was not inserted at Stage 1. Thus, the **uses** that are inserted at Stage 3 have their corresponding **assigns** preceding them in the program order of  $L$ , and these **assigns** are added to  $\hat{S}$  at Stage 3.

Furthermore, the sequence of **assigns** and **uses** added to  $\hat{S}$  at Stage 3 constitutes a legal serialization by itself; otherwise there would be a **use**  $x, v$  with no preceding **assign**  $x, v$ , in which case there would necessarily be a corresponding **load**  $x, v$ , that would result in adding the **use**  $x, v$  to  $\hat{S}$  at Stage 2 and not at Stage 3. We conclude that the same sequence remains legal in  $S$ , and so the **use** sees the value given to  $x$  by the most recent **assign**.

### $S$ is consistent with the Causality<sup>T</sup> Relation

We now show that the order of **use** and **assign** instructions is consistent with the Causality<sup>T</sup> Relation. We consider the relations defined in some local history  $L$ .

Assume the contrary, i.e., that the order of two **use/assign** instructions that are related by  $CR^T$  in  $L$  is switched in  $S$ . We call these instructions a *violating pair*.

Let us look at one of the violating pairs. We can always choose it so that the corresponding application of the Causality<sup>T</sup> Relation is *basic*, that is, transitivity is not involved. The reason is that any application of transitivity and a corresponding violating pair involves a sequence of basic applications, of which at least one is violating.

Note that the instructions inserted to  $\hat{S}$  at Stage 3 of the construction preserve their original program order from  $L$ ; hence, they cannot violate the Causality<sup>T</sup> Relation. Thus, none of the instructions in the violating pair could be inserted at that stage.

We now check the rules of the Causality<sup>T</sup> Relation for the other possibilities.

**transistor rule.** *The violating pair involves a use  $x, v$ , denoted  $u$ , which precedes an assign, denoted  $a$  (not necessarily referring to  $x$ ) in  $L$ . There exists an assign  $x, v$ , denoted  $a'$ , which is executed in a processor other than  $L$ .*

Suppose  $u$  is inserted to  $\hat{S}$  after  $a$ .  $u$  cannot correspond to  $a$  (same variable, same value) because  $a'$  does, and we assume that writes are unique.

Since  $u$  sees the result of  $a'$ , this implies that there is a sequence of **s-w-r-l** operations in  $T$  that transfers the value from  $a'$  to  $u$ . Thus,  $u$  gets its value from a **load**  $x, v$  operation denoted  $l$ .

$l$  must precede  $u$  in the timing  $T$ , and the **store** corresponding to  $a$ , denoted  $s$ , should succeed  $a$  in  $T$ . Since  $u$  precedes  $a$  in  $L$ , by transitivity we get that  $l$  precedes  $s$  in  $T$ . Thus,  $l \xrightarrow{CR} s$  (read-before-write), and therefore  $l$  precedes  $s$  in  $S'$ , and this order is also preserved in  $\hat{S}$ . Therefore, since by the construction,  $u$  immediately succeeds  $l$  and  $a$  immediately precedes  $s$ ,  $u$  must precede  $a$  in  $S$ , a contradiction.

**same variable: two assigns.** *The violating pair involves two assign  $x$  instructions.*

At Stage 1 of the construction, when an **assign** is inserted to  $\hat{S}$ , all the previous **assigns** that have not yet been inserted join it, where their order in  $L$  is preserved. Thus, at the end of Stage 1 all **assigns** (except for those left for Stage 3) have been inserted to  $\hat{S}$  in their program order. This shows that all pairs of **assigns** in  $S$  are non violating.

### The construction for local history $L$ :

**Stage 1 – Mapping assigns:** We go through the operations in  $\hat{S}$ ; for each **store**  $x, v$  we add the corresponding **assign**  $x, v$  from  $L$  to  $\hat{S}$ , so that it immediately precedes the **store**. We also add all the **assigns** preceding **assign**  $x, v$  in  $L$ 's program order that were not yet added to  $\hat{S}$ , and write them just before **assign**  $x, v$  in the order in which they appear in  $L$ .

**Stage 2 – Mapping uses:** We now go through the operations in  $L$ : for each **use**  $x, v$ , if there is a corresponding **load**  $x, v$  or **assign**  $x, v$  in  $\hat{S}$ , then we add the **use**  $x, v$  to  $\hat{S}$  immediately after the corresponding operation. Note that there may be several **use**  $x, v$ 's. (Recall also that we assume that the value yielded by a **USE** can always identify a **LOAD** or an **ASSIGN**.)

**Stage 3 – Mapping leftovers:** Finally, we add all the remaining **assigns** and **uses** to  $\hat{S}$  (for all variables), so that their program order in  $L$  is preserved, and so that they reside at the end of the constructed sequence  $\hat{S}$  after all the previously inserted instructions.

At Stage 1 every **assign** in  $L$  gets inserted into  $\hat{S}$ , except for the ones at the end of the program order. Every **use** returns the value that was brought by some **load** or some **assign**; thus, they are all inserted into the serialization at Stage 2 (except possibly those at the end of  $L$  that do not have a corresponding **load** and whose corresponding **assign** was not inserted at Stage 1). Thus, after stages 1 and 2, all instructions in  $L$  have been inserted into  $\hat{S}$ , except possibly for a suffix at the end of  $L$ . This suffix is inserted at Stage 3.

Denote by  $S$  the sub-sequence of **use/assign** operations in  $\hat{S}$ . It remains to be shown that  $S$  is a legal serialization of  $H$ , and that the order of **uses** and **assigns** is consistent with the Causality<sup>T</sup> Relation.

### $S$ is a legal serialization of $H$

For the serialization  $S$  to be legal, we need to show that every **use** sees the result of the closest preceding **assign** in  $S$ .

We first note that instructions that are inserted to  $\hat{S}$  at Stage 3 do not interfere with sequences of instructions that are inserted at the other stages. Thus, they never block a **use** from seeing the value put in the variable by some **assign**.

There are three ways in which **uses** are inserted to  $\hat{S}$  by the construction:

- *A use  $x, v$  is inserted after an assign  $x, v$  (Stage 2).* In this case, the **use** is inserted to  $\hat{S}$  immediately after the **assign**, as they both yield the same value, and thus no other **assign** can intervene between them. The same holds for the sub-sequence  $S$ .
- *A use  $x, v$  is inserted after a load  $x, v$  (Stage 2).* Once again, the **use** is inserted immediately after the **load**, and no other **assign** or **load** can intervene between them in  $\hat{S}$ . Since the serialization  $S'$  of **load/stores** was legal, the **load**  $x, v$  sees the result of the closest preceding **store**  $x, v$  in  $S'$ .

Consider the process of inserting **uses** and **assigns** into  $\hat{S}$ . Since **assigns** are inserted immediately before the corresponding **stores**, we conclude that no **assign**  $x, v'$  could be inserted between the **store**  $x, v$  and the **load**  $x, v$ . The **store**  $x, v$  certainly had a corresponding **assign**  $x, v$  (Constraint A.3.3). By the construction, the **assign**  $x, v$  was inserted immediately before the **store**  $x, v$ . We get a sequence of **assign-store-load-use** with no other

*Constraint A.4.2* holds, as by the construction, a `load x,v` is inserted to  $S'$  (and thus is performed in  $T$ ) after the `read x,v`.

*Constraint A.4.3.* Suppose by contradiction that some `store x,v` is performed in  $T$  and appears in  $S'$  after its corresponding `write x,v`.

This could happen only if some `load y,w` was inserted after some `read y,w`, where the `read y,w` succeeds the `write x,v`, and the `store x,v` was inserted later in the process.

In addition, the insertion of the `load y,w` must have been accompanied by an insertion of an immediately subsequent `use y,w`, and thus this `use` does not violate  $CR$ .

Finally, the insertion of the `store x,v` must have been accompanied by an immediately preceding `assign x,v`.

So, to summarize, the assumption leads to the existence of the following sub-sequence of instructions in  $S'$ :

write x,v, 
 read y,w, 
 load y,w, 
 use y,w, 
 assign x,v, 
 store x,v

According to the above the `store x,v` was inserted to  $S'$  after the `use y,w`. Since the insertion process follows the program order, we conclude that the `use y,w` precedes the `assign x,v` in  $H$ , and so they are causally related.

We thus know that in  $S$  the `use y,w` precedes the `assign x,v`, which contradicts the assumption that in  $S'$  the `read y,w` (corresponding to the `use y,w`) succeeds the `write x,v` (which corresponds to the `assign x,v`).

*Constraint A.4.4.* There is Causality Relation between all the operations on the same variable in the same local history; thus, all the `use/assigns` on the same variable preserve the program order from  $H$  in  $S$  and in  $S_{CR}$ , and the `read/writes` on the same variable in  $S'$  keep the program order. By the construction, there are corresponding `load/stores` for all instructions in  $S_{CR}$ , and their insertion process preserves the program order as well. Therefore, the constraint follows.

**Direction 2.** Java is not weaker than  $\text{JavaL}^T$ . We need to show that each execution that is valid under Java is also valid under  $\text{JavaL}^T$ . In other words, given a Java execution  $H$  we show that there exists a legal serialization  $S$  of `use` and `assign` instructions which is consistent with the Causality<sup>T</sup> Relation.

We construct the required serialization in the following way. First, we show a serialization of `load/store` instructions which is consistent with the Causality Relation between `load/stores`. Then, we add to the `load/stores` the `use/assigns` (as in the given execution  $H$ ), and show that the added instructions form the required serialization.

### Constructing the serialization $S$

Let  $T$  be an assignment of `l/s/r/w` operations and a timing for all instructions (those in  $H$  and the additional ones from the assignment) which complies with the Java constraints. Let  $T'$  be the `l/s/r/w` instructions in  $T$  and the timing induced on them. Let  $H'$  be an execution consisting of the `load/store` instructions in  $T$ . Since  $T$  complies with Java,  $T'$  complies with  $\text{Java}'$ , hence  $H'$  is  $\text{Java}'$ . Since  $\text{Java}' = \text{JavaL}$ ,  $H'$  is  $\text{JavaL}$ , i.e., there exists a legal serialization  $S'$  of the `load/store` instructions in  $H'$ , which is consistent with the Causality Relation.

Now, we extend  $S'$  to include also the `use/assign` operations from  $H$ . The resulting sequence is denoted  $\hat{S}$ .  $\hat{S}$  is initialized to  $S'$ , and then is extended by iterating on all local histories in  $H$ . For each variable  $x$  in the current local history  $L$ , its `use/assign` operations in  $L$  are inserted to  $\hat{S}$  in the following way.

- For a **use**  $x,v$  that was not violating  $CR$ , insert to  $S_{CR}$  a **load**  $x,v$  and the **use**  $x,v$  after the corresponding **read**  $x,v$ , and after the last operation inserted from this local history.
- A **use**  $x,v$  that violated  $CR$  is inserted after its corresponding **assign**  $x,v$  and after the last operation inserted from this local history.
- For an **assign**  $x,v$ , insert the **assign**  $x,v$  and a **store**  $x,v$  after the last operation inserted for this local history.

The resulting sequence of operations is called  $S'$ . We tag each instruction in  $S'$  by the index of the original local history in  $H$  of the corresponding **use** or **assign**.

3.  $S'$  may be viewed as the required timing  $T$  in the following way. The  $i$ th instruction in  $S'$  is performed in the  $i$ th time step, where **u/a/l/s** instructions are performed by the processor with the index matching their tags, and the **r/w** instructions are performed by the main memory.

It remains to be shown that  $T$  complies with all the Java constraints from A.2, A.3, and A.4.

Of the four constraints in A.2, the latter two are irrelevant. Constraint A.2.2 is clearly satisfied by the fact that  $S$  is a legal serialization. For Constraint A.2.1 we need to show that every **use**  $x$  and **store**  $x$  see the value written to  $x$  by the most recent **assign** or **load**. We consider the following cases.

- *use that did not violate the  $CR$ .* Such **uses** are inserted immediately after their **loads** and thus they see the right value.
- *use  $u$  that violated the  $CR$ .* The **use** that violated  $CR$  but did not violate  $CR^T$  gets its value from an **assign** operation  $a$  in its local history. No other **assign** operation to the same variable could intervene between  $a$  and  $u$  in  $S$ , as  $S$  is a legal serialization. Since in  $H$  there is a Causality<sup>T</sup> Relation between all the operations to the same variable, the order of these operations is preserved in  $S$ . We conclude that no **assign** operation to the same variable could intervene between  $a$  and  $u$  in  $H$ . Similarly, no **use** operation between  $a$  and  $u$  could see a different value.

In the construction of  $S'$  the  $u$  is inserted after  $a$ , or after a sequence of other instructions from the same local history. As shown above, none of the intervening instructions can change the yielded value for  $u$ . Thus, in  $T$  the **use** sees the value of  $a$ , its corresponding **assign**.

- *stores.* All the **stores** are inserted immediately after their **assigns**, and thus see the right value.

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

*Constraint A.3.2* holds because in  $S'$  (and thus in  $T$ ) there is a **store**  $x,v$  immediately after each **assign**  $x,v$ .

*Constraint A.3.3* holds as in  $S'$  (and thus in  $T$ ) for each **store**  $x,v$  there exists an immediately preceding **assign**  $x,v$ .

*Constraint A.3.4* holds as a **use**  $x,v$  is inserted to  $S'$  in one of two ways: either with a **load**  $x,v$  in front, or otherwise when the transistor rule was not applicable, and thus there was necessarily a local **assign**  $x,v$  preceding the **use**  $x,v$ .

*Constraint A.4.1* is obviously irrelevant.

**Causality Relation:**  $1.0 \xrightarrow{CR} 1.1$ ,  $1.1 \xrightarrow{CR} 1.2$  (same variable),  $1.2 \xrightarrow{CR} 1.3$  (read-before-write).  $2.1 \xrightarrow{CR} 2.2$  (read-before-write),  $2.2 \xrightarrow{CR} 2.3$  (same variable).

**write-before-read:** This gives  $1.0 \xrightarrow{DD} 2.3$  and  $1.3 \xrightarrow{DD} 2.1$ .

**triangle rule:** There is one application:  $1.0 \xrightarrow{DD} 2.3$  and  $1.0 \xrightarrow{DD} 1.1$  imply  $2.3 \xrightarrow{DD} 1.1$ .

The following is a cycle of Data Dependency:  $1.1 \xrightarrow{DD} 1.2 \xrightarrow{DD} 1.3 \xrightarrow{DD} 2.1 \xrightarrow{DD} 2.2 \xrightarrow{DD} 2.3 \xrightarrow{DD} 1.1$ . Hence, this execution is not valid under JavaD.

	Processor 1	Processor 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 and invalid for JavaD. The instruction 1.0 serves for initialization of  $x$  and is shown only for the sake of clarity in constructing the Data Dependency relation; generally, its presence is assumed implicitly.

To adjust JavaD for this scenario, we weaken it in the following way: We say that there is Data<sup>T</sup> Dependency from a `use` to a following `assign` in the program order when the `use` sees the result of an `assign` from some other program order. Intuitively, the fact that the `use` sees the `assign` from another processor, compels it to have the corresponding `load`, and thus to behave according to the Causality Relation. The resulting model, called JavaD<sup>T</sup>, as well as its equivalent JavaL<sup>T</sup>, were formally introduced in Section 3.3.

We are now ready to show that JavaD<sup>T</sup> is a tight specification of Java Consistency.

**Theorem 4.2** *Java Consistency is equivalent to JavaD<sup>T</sup>.*

As was shown in Section 3.3, JavaD<sup>T</sup> is actually equivalent to JavaL<sup>T</sup>. Thus, we will show instead that Java is equivalent to JavaL<sup>T</sup>.

In the proof we use the fact, proven in Section 4.2, that the Java' is equivalent to JavaL.

**Proof Direction 1.** JavaL<sup>T</sup> is not weaker than Java. In order to prove this direction, we need to show that each execution which is valid under JavaL<sup>T</sup> is also valid under Java. In other words, given an execution  $H$  that has a legal serialization  $S$  of the `use/assigns` which is consistent with the Causality<sup>T</sup> Relation, we construct an assignment of the `l/s/r/w` operations with a timing on all the instructions,  $T$ , which complies with all the Java constraints. The construction of  $T$  goes as follows.

1. Calculate the Causality Relation  $CR$  on  $H$ . Let  $S_{CR}$  be  $S$  with all the `uses` which take part in a violation of  $CR$  in  $S$  eliminated. Since  $S_{CR}$  has all the `assign` operations from  $S$ , and only those `uses` that do not violate  $CR$ ,  $S_{CR}$  is consistent with  $CR$  (see remark after the definition of  $CR^T$ ).

Rename the `use` operations in  $S_{CR}$  `reads`, and the `assigns`, `writes`.

2. We next add the `u/a/l/s` operations to the  $S_{CR}$ . We treat the instructions of each local history according to the program order, as follows:

3. Main memory instructions which access a single variable and for which the corresponding **load/stores** appear in the same local history agree with the program order of that local history (Constraint A.4.4).
4. In the order of operations for each variable  $x$  every **read**  $x$  yields the value of the latest **write**  $x$  (Constraint A.2.2).

*Proof of 1.* This follows by the construction: the **load**  $x,v$  is placed in  $T$  after its corresponding **read**  $x,v$ .

*Proof of 2.* Let us assume the contrary, i.e., that in  $T$  there is some **store**  $x,v$  that is placed after its corresponding **write**  $x,v$ . Suppose our **store**  $x,v$  is the first such situation in  $T$ . By the construction of  $T$ , this can happen only if there is some other **load/store** instruction  $o$  which precedes our **store**  $x,v$  in the program order (of the corresponding local history), and that must be inserted after the **write**  $x,v$ . The only case when a **load/store** instruction is forced to appear after a main memory **write** instruction during the construction is in the insertion of a **load**, so we know that  $o$  is a **load**  $y,w$ .

Thus, we have the following situation: In  $T$ , the **load**  $y,w$  appears before the **store**  $x,v$ , but the corresponding **write**  $x,v$  appears before the **read**  $y,w$  that corresponds to the **load**  $y,w$ . But we know that in  $H$   $\text{STORE } X,V \xrightarrow{CR} \text{LOAD } Y,W$ , and thus the **LOAD**  $Y,W$  would precede the **STORE**  $X,V$  in  $S$ . Therefore, in the timing  $T$ , the **read**  $y,w$  would appear before the **write**  $x,v$ , a contradiction.

*Proof of 3.* Since all operations on the same variable in a local history are Causally Related, JavaL requires that all accesses to the same variable appear in  $S$  in the program order. By the construction of the Java' timing  $T$ , this order remains as is, and so this Java' requirement is satisfied.

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

### 4.3 Specification for the Java Consistency

We emphasize here that Java is not equivalent to JavaD, because, according to the Java specification the compiler is not required to insert a **load** instruction for each **use**. If the **use** has no **load** (which Java constraints would require to precede the **use**), an **assign** that precedes the **use** in the program order can have its corresponding **store** succeed the **store** corresponding to some other **assign** which follows the first **assign** in the program order. Thus, the Causality Relation is violated in this timing.

This situation is illustrated by the example depicted in Figure 11. This execution is possible under Java, as follows. The compiler can generate the **load/stores** for Processor 1 in the following way:

a x,0, 
 a x,1, 
 u x, 
 a y,1, 
 s y,1, 
 s x,1

And for Processor 2:

l y, 
 u y, 
 a x,2, 
 s x,2, 
 l x, 
 u x

Then, it is easy to find a timing in which the **load**  $x$  in Processor 2 will not see the **store**  $x,1$  in Processor 1, but the **load**  $y$  from Processor 2 will see the **store**  $y,1$  in Processor 1, thus obtaining the required result.

However, from the JavaD point of view this execution is impossible. Let us build the Data Dependency relation to see that it contains a cycle:

We now show that the serialization  $S$  is legal. We assume otherwise and get a contradiction. This can happen when a `LOAD x,v` that returns the result of some `STORE x,v` cannot see the result of the `STORE` according to  $S$ . There are two cases: the `LOAD` precedes the `STORE` in  $S$ , or the `LOAD` succeeds the `STORE` but there is another `STORE x,w` between them. Let us examine the cases:

1.  $o_1$  is a `STORE x,v` and  $o_2$  is a `LOAD x,v` but  $o_2$  precedes  $o_1$  in  $S$ . Java' requires the order of operations on every single variable in the main memory (Constraint A.2.2), and this order is preserved by the construction of  $S$ . Thus, it is also the case that in  $T$  the corresponding main memory operation `read x,v` precedes the corresponding main memory operation `write x,v`. However, by Constraint A.2.2, this implies that the `read` cannot see the result of the `write`, a contradiction.
2.  $o_1$  is a `STORE x,v` and  $o_2$  is a `LOAD x,v`, and there is a `STORE x,w` operation  $o$  in  $S$  between  $o_1$  and  $o_2$ . As above,  $T$  implies a total order of main memory accesses to any single variable, and this order is preserved in the construction of  $S$ . Thus, the `write x,w` instruction corresponding to  $o$  appears between  $o_1$  and  $o_2$  in  $T$  as well, which by Constraint A.2.2 implies a contradiction.

**Direction 2:** *JavaL is not weaker than Java'.*

Given an execution  $H$  which is valid under JavaL we show that  $H$  has both an assignment of reads and writes to the main memory part and a timing which comply with the Java' constraints.

Since  $H$  is JavaL, there exists a legal serialization  $S$  of  $H$  which preserves the Causal Relations in  $H$ . We construct the timing through the following steps:

1. We construct  $S'$  out of  $S$  by replacing each `LOAD x,v` by a `read x,v` and each `STORE x,v` by a `write x,v`.
2. We now extend  $S'$  to a sequence of operations  $T$  according to the following iterative process. We first let  $T$  consist of  $S'$ . Then, for every local history in  $H$ , for each of its `LOAD/STORE` instructions we insert a `load/store` instruction into  $T$ , respectively. The local histories are handled in an arbitrary order, and the instructions of each local history are treated one by one according to the program order in  $H$ . The insertion of an instruction is as follows:
  - A `load x,v` is inserted immediately after whichever instruction came last: its corresponding `read x,v` or the last `load/store` instruction that was inserted.
  - A `store` is inserted immediately after the last `load/store` instruction that was inserted.

By the construction, the instructions inserted into  $T$  for a certain local history of  $H$  preserve the program order of its corresponding instructions. Also, for all `load/stores`,  $T$  contains corresponding `read/write` instructions. Thus,  $T$  contains an assignment of such operations to the main memory part of  $H$ .

Now, since  $T$  is a serialization of all operations, including the main memory part, we view it as a possible timing, so that the  $i$ th instruction in  $T$  is executed at the  $i$ th time-step. If it is a `read/write` instruction, it is executed by the main memory. If it is a `load/store` instruction of which the original `LOAD/STORE` instruction in  $H$  belongs to the local history of processor  $j$ , then this instruction is performed by processor  $j$ .

We next show that the timing  $T$  complies with the Java' constraints. To this end, the following is required:

1. A `load x,v` is performed in  $T$  after its corresponding `read x,v` (Constraint A.4.2).
2. A `store x,v` is performed in  $T$  before its corresponding `write x,v` (Constraint A.4.3).

**Java'**: An execution consisting of **loads** and **stores** belongs to Java' Consistency (or, Java'), when there is an assignment of **reads** and **writes** to the main memory part, so that there is a timing for the execution and the additional **read/writes** which complies with constraints from A.2 and A.4.

Note that Java' cannot be seen as a subset of the Java constraints because the Java memory behavior defines the interaction of the **use/assigns** with the **read/writes**, whereas Java' deals with the interaction of **load/stores** with the **read/writes**.

**Theorem 4.1** *Java' is equivalent to JavaL.*

**Proof Direction 1:** *Java' is not weaker than JavaL.*

Given a Java' execution  $H$  we show that it has a legal serialization which preserves the Causality Relation, and thus  $H$  is also in JavaL.

Because  $H$  is in Java' it has an assignment of **reads** and **writes** to the main memory part and a timing which complies with the Java' constraints. We use such an assignment, and denote it by  $T$ .

By Constraint A.2.4 the application of the Java' 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 using a topological sort, we obtain a serialization of  $T$  (including the main memory part) which preserves the Java' constraints. Consider the sub-sequence of this serialization  $S$  which consists of the main memory operations only, and in which every **read**  $x,v$  (resp. **write**  $x,v$ ) is replaced by the corresponding processor instruction **LOAD**  $x,v$  (resp. **STORE**  $x,v$ ). Here we use the one-to-one correspondence between **read/write** and **load/store** operations, which follows from the constraints in A.4.

To prove that  $H$  is JavaL we now claim that  $S$  is a legal serialization of  $H$  which preserves the Causality Relation.

Assume the opposite. First assume that in  $S$  there are two operations,  $o_1$  and  $o_2$ , so that  $o_1 \xrightarrow{CR} o_2$ , but  $o_2 \xrightarrow{S} o_1$ . Let us call the same-variable and the read-before-write rules of the Causality Relation (i.e., those rules that do not follow from transitivity) *basic*. Wlog we may assume that the Causal Relation between  $o_1$  and  $o_2$  is basic. (Otherwise, there is a sequence of operations  $o'_1 = o_1, o'_2, \dots, o'_n = o_2$ , such that each pair of  $o'_i, o'_{i+1}$  is Causally Related by one of the basic rules of Causality. Since  $o_2$  precedes  $o_1$  in the serialization, there must be at least one pair of operations  $o'_i, o'_{i+1}$  such that  $o'_i \xrightarrow{CR} o'_{i+1}$  by a basic rule, but  $o'_{i+1} \xrightarrow{S} o'_i$ .)

Consider the basic rules of the Causality Relation:

1.  $o_1$  and  $o_2$  are two accesses to the same variable that are performed by the same processor. In  $T$  the **read/write** operations must appear in the same order as the corresponding **load/store** operations (Constraint A.4.4). Thus, the corresponding operations in  $S$  must appear in the order that complies with Causality Relation requirements (which in  $H$  are the same as in  $T$ ), a contradiction.
2.  $o_1$  is a **LOAD**  $x,v$  and  $o_2$  is a **STORE**  $y,w$  performed by the same processor. The **LOAD** corresponds to a **load** instruction in  $T$ , and the **STORE** corresponds to a **store** in  $T$ , where the **load** precedes the **store** in the program order. But, by constraints A.4.2 and A.4.3, in  $T$  the corresponding **read** should precede the **load**, and the corresponding **write** should succeed the **store**. By transitivity, the **read** must precede the **write** in  $T$ . By the construction this implies that the **LOAD** would precede the **STORE** in  $S$ , a contradiction.

## 4 Tight Non-Operational Specifications for Java

In this section we show that JavaL and JavaD are equivalent to the implementor view of Java, and that JavaL<sup>T</sup> and JavaD<sup>T</sup> are equivalent to the programmer view of Java Consistency.

The section is organized as follows. First, in Section 4.1 we discuss the assumptions that can be made on the bytecode production by the compiler, and its execution by the JVM. Then, in Section 4.2, the implementor view of Java memory behavior is presented; we dub this view Java', and show that Java' is equal to JavaL (Theorem 4.1). Section 4.3 provides the proof that JavaD<sup>T</sup> is equivalent to Java Consistency, i.e., it matches the Java programmer view (Theorem 4.2).

### 4.1 Compliance to the Java Consistency Model

In the compilation stage of the Java source code, the order of **use** and **assign** instructions is determined by the compiler from the program code, and then **load** and **store** instructions are placed among them, as necessary. Note that the compiler may generate all the instructions at the same time, but the two stage structure is a more convenient way to examine the compilation process from a methodological point of view. The reason is that the placement of the **use** and **assign** instructions follows directly from the source code (Constraint A.3.1), whereas the placement of the **loads** and the **stores** is governed by the Java constraints according to the appearance of the **uses** and the **assigns**.

Note that during the execution stage the JVM can rely on the bytecodes only and has almost no information concerning the source code (some additional information is contained in the `.class` file, which specifies, for instance, which variables are volatile). If the compiler produces code that does not comply with the program, the JVM will not execute correctly. Therefore, at the implementation level, it can be assumed that the bytecodes produced by the compiler comply with the JLS constraints.

The same reasoning holds for the placement of **store** and **load** instructions by the compiler. There are cases in which it is possible to check the compliance of these instructions with the **assigns** and **uses**, but it seems infeasible to perform these checks during execution. For example, let us assume a Java method that begins with a **use** instruction of some variable (which is generally prohibited by Constraint A.3.4). This could happen, for example, because the compiler performed a global data flow analysis of the program and found out that every call to the method is preceded by an **assign** to the variable. Validating the code in this situation would require recalculating the global data flow of the program, which is obviously too expensive for run-time. Thus it can be assumed by the JVM implementor that the compiler produces correct bytecodes for **loads** and **stores**.

Since we conclude that the placement of **load** and **store** instructions should be left to the compiler, the JVM is left with the responsibility to execute the **u/a/l/s** instructions and map **load/store** instructions into **read** and **write** operations in the main memory.

### 4.2 Specification for the JVM Memory Behavior

Out of all the constraints in Appendix A, those in A.2 are general, and those in A.4 define the interaction of operations performed by the main memory with the instructions executed by the processors. The rest of the constraints in Appendix A are irrelevant for the implementation of the JVM memory manager (with respect to regular variables only). We thus refer to Constraints A.2 and A.4 as the definition of JVM memory behavior, and call them: Java'.

<i>Processor 1</i>	<i>Processor 2</i>	<i>Processor 3</i>
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 [9, 15] and is valid for JavaL.

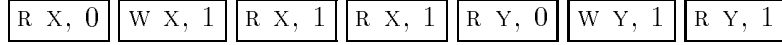


Figure 6 gives an example execution (taken from [15]) which is valid for Causal Consistency. This example is invalid for JavaL, as we later show that JavaL is equivalent to Java', and Java' can be shown to be coherent by arguments similar to those of the proof given in Section 2.1. The fact that the example is invalid for JavaL is also easy to show directly from the definition of JavaL.

### 3.3 JavaL<sup>T</sup> and JavaD<sup>T</sup>

In this section we give two equivalent non-operational definitions, JavaL<sup>T</sup> and JavaD<sup>T</sup>. In Section 4.3 it will be proved that they are equivalent to Java Consistency (the programmer view).

We start with the definition of the *Causality<sup>T</sup> Relation* denoted  $CR^T$ :

**Causality<sup>T</sup> Relation:** Let  $o_1$  and  $o_2$  be two instructions performed by the same processor, 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**, where  $o_1$  and  $o_2$  access different variables, but  $o_1$  is a READ and  $o_2$  is a WRITE, and there exists an additional WRITE operation  $o'$ , such that  $o_1$  sees the result of  $o'$  and the processor in which  $o'$  is executed is different from the one in which  $o_1$  is executed.<sup>2</sup>

**transitivity**, i.e., there exists instruction  $o'$  such that  $o_1 \xrightarrow{C^T} o'$  and  $o' \xrightarrow{C^T} o_2$ .

We remark that  $CR^T$  is strictly weaker than  $CR$ . They are distinguished by the switch from the read-before-write rule to the transistor rule. This switch implies that some pairs of **use-before-assign** which were related in  $CR$  cease to be related in  $CR^T$ . In the reverse direction, when there is no **use-before-assign** which is related in  $CR$  but is not related in  $CR^T$  then the induced set of relations on the execution is the same. We will use this fact in the proof of Theorem 4.2.

We now define JavaL<sup>T</sup> to be  $LS(CR^T)$ , and JavaD<sup>T</sup> to be  $DD(CR^T)$ . We also define Data<sup>T</sup> Dependency to be  $C_D(CR^T)$ .

Now, as in the case of JavaL and JavaD, Lemma 3.1 implies the following theorem:

**Theorem 3.2** *JavaL<sup>T</sup> is equivalent to JavaD<sup>T</sup>.*

<sup>2</sup>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  $T$ , as in “JavaL<sup>T</sup>” and “JavaD<sup>T</sup>”.

	<i>Processor 1</i>	<i>Processor 2</i>	<i>Processor 3</i>	<i>Processor 4</i>
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 which is invalid for PCD (taken from [2]) and is valid for JavaD and JavaL.

Figure 8 gives an execution which is shown in [2] to be invalid for PCD. However, it is valid in JavaD, as we show by computing the Data Dependency relations. There are two applications of the same-variable rule:  $1.1 \xrightarrow{DD} 1.2$  and  $3.1 \xrightarrow{DD} 3.2$ . There are four applications of the write-before-read rule:  $1.1 \xrightarrow{DD} 4.4$ ,  $3.1 \xrightarrow{DD} 2.4$ ,  $1.3 \xrightarrow{DD} 2.3$ , and  $3.3 \xrightarrow{DD} 4.3$ . There are two applications of the triangle rule:  $4.4 \xrightarrow{DD} 1.2$  and  $2.4 \xrightarrow{DD} 3.2$ . There are no relations that may be inferred by transitivity. Hence there are no Data Dependency relation cycles and the execution is JavaD.

	<i>Processor 1</i>	<i>Processor 2</i>
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 which is invalid for PCG (taken from [2]) and is valid for JavaD and JavaL.

Figure 9 presents an execution which is shown in [2] to be invalid for PCG. However, it is valid for JavaD, as follows. There are two applications of the same-variable rule:  $1.1 \xrightarrow{DD} 1.2$  and  $2.1 \xrightarrow{DD} 2.2$ . There are two applications of the write-before-read rule:  $1.1 \xrightarrow{DD} 2.4$  and  $2.1 \xrightarrow{DD} 1.4$ . There are two applications of the triangle rule:  $1.4 \xrightarrow{DD} 2.2$  and  $2.4 \xrightarrow{DD} 1.2$ . There is no application of transitivity. Thus, there are no Data Dependency cycles, and the execution is JavaD.

We claim that both PCG and PCD are incomparable with JavaD. To this end, recall that we have already shown in Section 2 that PCD and PCG contain executions that are not Java. We now refer the reader to the results in the next section, which show that JavaD is stronger than Java, from which the claim is derived.

### 3.2.2 JavaL vs. Causal Consistency

Although JavaL is reminiscent of Causal Consistency [9, 15] they are incomparable. Basically, Causal Consistency requires that if two WRITES are causally related, they are necessarily seen by all other processors in the same order. In JavaL, on the other hand, two READS that are not data dependent can see the WRITES in reverse order.

Figure 10 gives an example which is invalid in Causal Consistency: although the  $w\ Y,1$  is causally dependent on the  $w\ X,1$ , processors 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 JavaL.

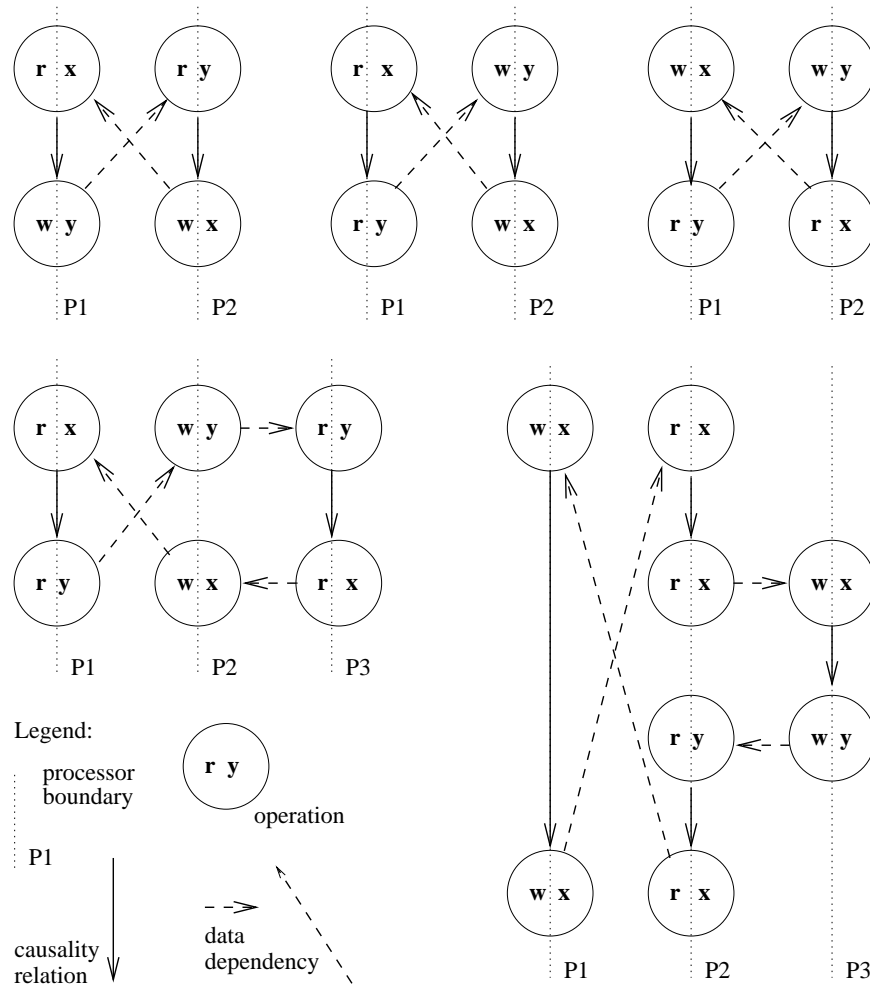


Figure 7: Some examples of Data Dependency cycles that are forbidden in JavaD. The relations drawn are not necessarily basic; for instance,  $r\ x \xrightarrow{DD} r\ y$  when there exists a  $w\ y$  in between which is not shown in the figure.

**Theorem 3.1** *JavaL is equivalent to JavaD.*

We remark that in contrast to JavaL, deciding whether a given execution is in JavaD is a simple, constructive process. Thus, the equivalence of Java', JavaL and JavaD, that is shown below, implies an “automatic” efficient verifying mechanism for the compliance of a given execution with the Java' constraints.

Figure 7 shows some examples of Data Dependency cycles that are forbidden in JavaD.

### 3.2.1 JavaD vs. Processor Consistency

As mentioned in Section 2.3 (refer to this section for precise definitions), Ahamad et.al. [2] defined two variants of PC, namely PCG and PCD. We give here examples that can be shown to be JavaD and are known not to be PC (one for each variant).

serialization, it must see the value  $v'$ , and not the value  $v$ . Thus,  $r$  must appear before  $w'$  in  $LS_H^C(r \xrightarrow{LS_H^C} w')$ .

We have shown that  $LS_H^C$  is a serialization of  $H$  that is consistent with all the constraints in  $C_D(C)$ ; thus, there can be no cycles in the induced relations in  $H$ , and  $H$  is  $DD(C)$ .

**Direction 2.**  $DD(C)$  is not weaker than  $LS(C)$ .

Let  $H$  be a valid  $DD(C)$  execution. We show that it is also valid under  $LS(C)$ .

Since  $H$  is valid under  $DD(C)$ , there is no cycle of  $C_D(C)$  in it. Thus a serialization which preserves the dependencies can be obtained from  $H$ , e.g., by applying a topological sort.

From all possible serializations we choose the following: given a WRITE  $x, v$  denoted  $w_1$  and a READ  $x, v$  denoted  $r_1$ , which sees the result of  $w_1$ , and another write to  $x$ : WRITE  $x, v'$  denoted  $w_2$ , then  $w_2$  is placed after  $r_1$  if  $w_1 \xrightarrow{C_D(C)} w_2$ , and before  $w_1$  otherwise.

We need to show that such a serialization exists in the partial topological order, and that it is legal.

Assume that the required serialization does not exist among the possible topological orders. We use the same notation as above. Let us examine some  $w_2$  that cannot be placed according to the requirements. There are two cases:

1.  $w_2 \xrightarrow{C_D(C)} w_1$ .  $w_1$  is seen by  $r_1$ , but  $w_2$  cannot succeed  $r_1$ . This implies that  $w_2 \xrightarrow{C_D(C)} r_1$ . However,  $w_1 \xrightarrow{C_D(C)} w_2$  and  $w_1 \xrightarrow{C_D(C)} r_1$  (the latter, because  $r_1$  sees the result of  $w_1$ ), so by the Triangle rule  $r_1 \xrightarrow{C_D(C)} w_2$ . This implies a cycle of  $C_D(C)$  in  $H$ :  $w_2 \xrightarrow{C_D(C)} r_1 \xrightarrow{C_D(C)} w_2$ , a contradiction.
2. There is no  $C_D(C)$  constraint from  $w_1$  to  $w_2$ , but  $w_2$  cannot precede  $w_1$ . This implies that  $w_1 \xrightarrow{C_D(C)} w_2$ , a contradiction.

It still must be shown that the elected serialization is legal. A READ  $x, v$  that sees the result of a WRITE  $x, v$  is dependent (according to  $C_D(C)$ ) on it, and thus appears in the serialization after the WRITE. Now, by the choice of serialization, any other WRITE  $x, v'$  to the same variable would be placed either before our WRITE  $x, v$  or after the READ  $x, v$ . We conclude that the serialization is legal. ■

### 3.2 JavaL and JavaD

In this section we give two equivalent non-operational definitions, JavaL and JavaD. In Section 4.2 we show that they are equivalent to Java'.

We start with the definition of the *Causality Relation*, denoted  $CR$ :

**Causality Relation:** Let  $o_1$  and  $o_2$  be two instructions performed by the same processor,

where  $o_1 \xrightarrow{po} o_2$ . 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  $o_1$  and  $o_2$  access different variables, but  $o_1$  is a READ and  $o_2$  is a WRITE, or

**transitivity**, i.e., there exists an instruction  $o'$  such that  $o_1 \xrightarrow{CR} o'$  and  $o' \xrightarrow{CR} o_2$ .

Now we define JavaL to be  $LS(CR)$ , and JavaD to be  $DD(CR)$ . We also call the relation  $C_D(CR)$  *Data Dependency (DD)*.

By Lemma 3.1 the following theorem follows.

**write-before-read rule:** If a READ X,V operation  $r$  sees the value written by a WRITE X,V operation  $w$ , then  $w \xrightarrow{C_D(C)} r$ .

**triangle rule:** If a READ X,V operation  $r$  sees a WRITE X,V operation  $w$ , and there is another WRITE X operation  $w'$  such that  $w \xrightarrow{C_D(C)} w'$ , then  $r \xrightarrow{C_D(C)} w'$ .

**transitivity:** If there exists an operation  $o'$  such that  $o_1 \xrightarrow{C_D(C)} o'$  and  $o' \xrightarrow{C_D(C)} o_2$ , then  $o_1 \xrightarrow{C_D(C)} o_2$ .

Let the *Data Dependency consistency for  $C$* , denoted by  $DD(C)$ , be the following:

$DD(C)$ : Execution  $H$  is said to be valid under  $DD(C)$  if there are no (directed) cycles of the  $C_D(C)$  relation between operations in  $H$ .

Let the *Legal Serialization consistency for  $C$* , denoted by  $LS(C)$ , be the following:

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

We are now ready for the basic lemma relating legal serializations and Data Dependencies.

**Lemma 3.1** *Given a set of constraints  $C$ ,  $LS(C)$  is equivalent to  $DD(C)$ .*

**Proof Direction 1.**  *$LS(C)$  is not weaker than  $DD(C)$ .*

Let  $H$  be a valid  $LS(C)$  execution. We show that it is valid also under  $DD(C)$ .

Since  $H$  is a valid  $LS(C)$ , there exists a serialization of operations in  $H$  which is consistent with the rules of  $C$ . We need to show that this serialization is also consistent with the rest of the  $C_D(C)$  rules. Then we can conclude that there are no cycles of  $C_D(C)$ , and thus the execution is  $DD(C)$ .

Note that the write-before-read and the restrictions from  $C$  are applied directly to the instructions in a given execution, whereas the triangle rule and the transitivity require the write-before-read and the restrictions from  $C$  to build upon. We may thus view the process of applying restrictions to produce the constraints in  $C_D(C)$  on the instructions in  $H$  as being divided into stages, where all the constraints that result by applying restrictions from  $C$  and the write-before-read rule belong to Stage 1, and each application of any of the additional constraints begins in a new stage.

We prove that  $LS_H^C$  is consistent with  $C_D(C)$  by induction on the stage index.

**Basis.** At Stage 1, the set of applicable constraints is  $C$  and the write-before-read rule. By definition of  $LS_H^C$ , it is consistent with all the constraints in  $C$ . When there is a READ X,V operation  $r$  and a WRITE X,V operation  $w$ , then, since  $LS_H^C$  is a legal serialization,  $w \xrightarrow{LS_H^C} r$ . Thus  $LS_H^C$  is consistent with all the constraints introduced in Stage 1.

**Step.** Let us assume that all the constraints that were introduced at stages  $1, 2, \dots, i$  are preserved in  $LS_H^C$ , and prove that the constraint introduced at Stage  $i + 1$  is also preserved in  $LS_H^C$ .

The constraint introduced at Stage  $i + 1$  can be one of the following:

**Transitivity.** There are three operations  $o_1, o_2$  and  $o'$ , such that  $o_1 \xrightarrow{C_D(C)} o'$  and  $o' \xrightarrow{C_D(C)} o_2$ . By the induction assumption,  $o_1 \xrightarrow{LS_H^C} o'$  and  $o' \xrightarrow{LS_H^C} o_2$ . Since  $LS_H^C$  is a serialization, it holds that  $o_1 \xrightarrow{LS_H^C} o_2$ .

**Triangle rule.** There is a READ X,V operation  $r$ , a WRITE X,V operation  $w$  and a WRITE X,V operation  $w'$ , such that  $w \xrightarrow{C_D(C)} w'$ . By the induction assumption,  $w \xrightarrow{LS_H^C} w'$ . Since  $LS_H^C$  is a legal serialization,  $w \xrightarrow{LS_H^C} r$ . If  $r$  appears after  $w'$  in  $LS_H^C$ , then, since  $LS_H^C$  is a legal

2. *Weak order relation.* Since a READ appears before WRITE in both processors, the dependencies remain the same:  $1.1 \xrightarrow{wpo} 1.2$  and  $2.1 \xrightarrow{wpo} 2.2$ .
3. *Weak writes-before and Weak reads-before relation.* We can see from the definitions that no two operations have these relations.
4. Finally, *semi-causality relation.* It is the transitive closure of the relations above, and thus is equal to the program order. So, the semi-causality relations are:  $1.1 \xrightarrow{s} 1.2$  and  $2.1 \xrightarrow{s} 2.2$ .

We then find serializations for each processor. For Processor 1, the required serialization is 2.2, 1.1, 1.2. It does not violate the semi-causality relation (because 1.1 precedes 1.2, as is required by the  $\xrightarrow{s}$ ), and there is only one WRITE operation for each variable, so the second part of the condition is satisfied trivially. The same can be shown for Processor 2.

We have shown that Java is neither weaker than PCD, nor stronger. We thus conclude that Java is incomparable with PCD. ■

### 3 Data Dependency and Legal Serialization

In this section we introduce non-operational declarations which will be used to specify the consistency models of both the programmer and the implementor views of Java. Each of the views is given two definitions, one in which each execution is required to have a legal serialization with certain properties, and another in which a certain Data Dependency relation does not include a cycle. Lemma 3.1 gives the basic result from which the connection between the two types of definitions is derived.

The two models which are shown to be equivalent to the Java implementor view are called JavaL and JavaD, and the two models which are equivalent to the Java programmer view are called JavaL<sup>T</sup> and JavaD<sup>T</sup>. JavaL and JavaL<sup>T</sup> are defined by means of restrictions on their legal serializations, whereas JavaD and JavaD<sup>T</sup> are defined in terms of Data Dependency relations (see 3.1).

The models introduced are abstract, and thus are defined using the standard READ/WRITE notation from the literature. In this respect, a READ denotes a generic read operation that may correspond to any concrete read operation, and likewise for the WRITE. We discuss Java in terms of concrete operations; therefore the READ/WRITE notation denotes *use/assign* when referring to the programmer level, and denotes the *load/store* when referring to the implementor level.

The section is organized as follows. Section 3.1 introduces and examines the relationship between the Legal Serialization and the Data Dependency consistencies. In Section 3.2 we present two non-operational definitions which we call JavaL and JavaD, and show their equivalence. In Section 3.3 we present two non-operational definitions, which we call JavaL<sup>T</sup> and JavaD<sup>T</sup>, and show their equivalence.

#### 3.1 Data Dependency and Legal Serialization

In this section we show that two rules, namely the *write-before-read rule* and the *triangle rule* are sufficient to imply legal serialization on the execution.

Let  $C$  denote a set of relations, and denote  $o_1 \xrightarrow{C} o_2$  when instruction  $o_2$  is related to instruction  $o_1$  according to  $C$ .

We define a relation  $C_D(C)$  as follows:

**rules from  $C$ :** If  $o_1 \xrightarrow{C} o_2$  then  $o_1 \xrightarrow{C_D(C)} o_2$ .

**Proof** To prove the claim we must show that Java is not weaker than PCG, i.e., present an execution which is valid for PCG and is invalid for Java. We will use the same example given above for Coherence in Figure 3. We have already shown that this execution is invalid for Java. Now, we will show that it is valid for PCG.

The second condition of the PCG definition is trivially satisfied as there is only one WRITE operation to each variable. In order to satisfy the first condition, we use the serialization 2.2, 1.1, 1.2 for Processor 1, and the serialization 1.2, 2.1, 2.2 for Processor 2. It is easy to see that these orders are legal in PCG and that the READS yield the required results. ■

### 2.3.2 Java vs. PCD

The definition of the PCD consistency in [2] is as follows:

**PCD:** We first define several notions:

- *Weak order relation* between two operations of the same processor: We say that  $o_1$  *weakly precedes*  $o_2$ , denoted  $o_1 \xrightarrow{wpo} o_2$ , 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.*  $o_1 \xrightarrow{wrb} o_2$  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, a history  $H$  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 processors' views.
2. For each processor  $p$ , 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  $S_p$ .

**Claim 2.6** *Java is incomparable with PCD.*

**Proof** We present the claim using, once again, the example from Figure 3. 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 are violated.

For Condition 1 we check that the execution is coherent. Since Coherency was shown by Claim 2.2, the condition is satisfied.

For Condition 2 we calculate the semi-causality relation for this execution.

1. *Program order relation.* There are two related dependencies:  $1.1 \xrightarrow{po} 1.2$ , and  $2.1 \xrightarrow{po} 2.2$ .

<i>Processor 1</i>	<i>Processor 2</i>	<i>Main Memory</i>
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 5: A possible timing for a Java execution implementing the program from Figure 4. 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 provoked by instruction 2 in Processor 1 (**s x,1**).

<i>Processor 1</i>	<i>Processor 2</i>	<i>Processor 3</i>	<i>Processor 4</i>
w x, 1	w x, 2	r x, 1	r x, 2
		r x, 2	r x, 1

Figure 6: A valid execution for PRAM which is invalid for Java.

### 2.3 Java vs. Processor Consistency (PC)

[2] defines two variants of PC: PCD and PCG, of which the corresponding non-operational definitions are taken from [5] and [6] respectively. Both variants are shown to be stronger than PRAM, and since we have seen an example indicating that Java is not stronger than PRAM, we also conclude that Java is not stronger than either PCD or PCG. The following theorem answers the question of whether Java is strictly weaker than any of them.

**Theorem 2.3** *Java is incomparable with both PCG and PCD.*

#### 2.3.1 Java vs. PCG

The definition of PCG, according to [2] is the following:

**PCG:** For each processor  $p$  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 operations to  $x$  then they appear in the same order in the serializations of all the processors.

**Claim 2.5** *Java is incomparable with PCG.*

**Claim 2.3** *Java is not stronger than PRAM.*

**Proof** Figure 4 shows an execution which is valid for Java and is invalid for PRAM, thus suggesting that Java is not stronger than PRAM. The intuitive reason for the difference between Java and PRAM in this example is that the Java constraints impose very little connection between operations on different variables, whereas PRAM requires that program order be preserved for all operations in the processor. (On the other hand, we already know that Java imposes some connections between operations on different variables, as it is stronger than Coherence).

<i>Processor 1</i>	<i>Processor 2</i>
WRITE X, 1	READ Y, 1
WRITE X, 2	READ X, 1
WRITE Y, 1	READ X, 2

Figure 4: An execution valid for Java but invalid for PRAM.

From the program order of Processor 1, the operation WRITE X,2 precedes WRITE Y,1. However, Processor 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 could happen in Java. As stated before, the operation READ X,1 denotes a **use** in the Java execution. The **stores** done by Processor 1 to  $x$  and to  $y$  are independent, and so it is possible for the processor to perform a **store** into  $y$  before it performs a **store** into  $x$ . Thus, the instructions actually executed by Processor 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 Processor 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 timing shown in Figure 5.

Since the values yielded by the **use** operations in Processor 2 are the same as these obtained by the READ operations in Figure 4, we conclude that this scenario is valid under the Java constraints. ■

**Claim 2.4** *PRAM is not stronger than Java.*

**Proof** Figure 6 shows an example for an execution which is valid for PRAM and is invalid for Java. The intuitive reason for the difference is that the Java constraints require Coherence for every single variable (as was shown in the previous section), while PRAM does not.

Both processors 3 and 4 see an order consistent with program orders of both writing processors (1 and 2). Because PRAM does not require correlation between the views of processors 3 and 4, there is no contradiction, and thus the execution is valid under PRAM. However, the conflict in the views of processors 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. ■

**Claim 2.2** *Java is strictly stronger than Coherence.*

**Proof** To prove the claim we show an execution which is valid for Coherence and is invalid for Java.

	Processor 1	Processor 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. As explained in Section 1.5, for Java READ denotes `use` and WRITE denotes `assign`.

Consider the execution in Figure 3. In order to show 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, however, this execution is impossible. We must show that no assignment of `load/store` and `read/write` operations that is consistent with the Java constraints can yield the same results as in the example. The `load x` operation by Processor 1 should appear before the `use x` for the `use` to see the result of `load`. Similarly, the operation `store y` by Processor 1 should appear after `assign y`. Because of the program order (Constraint A.2.1), `use x` must complete before `assign y`, and thus `load x` appears before `store y`. Now, because of Constraints A.4.2 and A.4.3, the `read x` by the main memory on behalf of Processor 1 should appear before `load x`, and `write y` should appear after `store y`. By transitivity, `read x` performed by the main memory for Processor 1 precedes `write y`. For the same reasons, `read y` by the main memory on behalf of Processor 2 precedes `write x`.

Let us denote the memory accesses by the indices of their corresponding instructions, e.g., the operation `read x` performed by the main memory will be denoted 1.1. In order to produce the required results, the interleaving of memory accesses must have 2.2 before 1.1, and 1.2 before 2.1. However, together with the constraints discussed in the previous paragraph, this induces a dependency loop in the timing ( $1.1 \rightarrow 1.2 \rightarrow 2.1 \rightarrow 2.2 \rightarrow 1.1$ ), in which the instruction 1.1 follows itself. This is prohibited by Constraint A.2.4. Thus, this scenario is impossible as a Java execution. ■

We remark here that the additional restriction on the memory behavior which is imposed by Java but not by Coherence is “Causality”, which basically (and informally) enforces writes that follow local reads to keep this order in the view of all processors. This and the following sections will give examples, explanations, and formal definitions of the way Java enforces this restriction.

## 2.2 Java vs. PRAM Consistency

The definition of the PRAM consistency is as follows [2]:

**PRAM:** A history  $H$  is PRAM if for each processor  $p$  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 example executions, one that is valid under Java but is invalid under PRAM, and another which is valid for PRAM but not for Java.

Constraint A.3.4 requires that the first instruction in the sequence for any variable be **load** or **assign**, thus the first block in the beginning of the sequence can start correctly.

If the current block is a **load-block**, then any number of **uses** goes to the same block. By Constraint A.3.3, no **store** instruction can appear. The appearance of **load** or **assign** will start a new block, which is well-defined (**load-block** or **store-block**, respectively). Therefore, a **load-block** terminates correctly and is followed by another block.

If the current block is **store-block**, then Constraint A.3.2 dictates that no **load** can appear after the **assign** before at least one **store** instruction. Once the **store** appears, A.3.3 dictates that no additional **stores** appear until the **store-block** terminates, which, by definition of a **store-block**, happens when the next instruction in the input is not **use** or **store**. Thus, when the block terminates, the next instruction will be either **assign** or **load**, implying the beginning of either a new **load-block** or a new **store-block**, accordingly.

Next, we notice that each block has only one associated **load** or **store** instruction. By constraints A.4.2 and A.4.3, there is exactly one main memory operation corresponding to each block. By Constraint A.2.2, those memory accesses are totally ordered in  $T$ . Now, we construct a global order of the blocks by placing them one after another in the order of their corresponding main memory accesses. This induces a serialization of the instructions, which, as we show below, can serve as the required serialization of  $H|x$ .

By Constraint A.3.1, the order of **use/assign** operations performed by a processor in  $T$ , and in particular the order of operations on  $x$ , is the same as in the corresponding local history in  $H$ . Note that by Constraint A.4.4, the order of operations performed by any individual processor during the construction does not change.

Although there are other (**load** and **store**) operations in the constructed order, only the order of the **assign** and **use** instructions is of interest. In order to show that this is indeed a legal serialization, we must show that the results seen by **use** operations in the original execution are the same as in the constructed order.

Consider the last **assign** or **load** operation,  $op$ , that appears before a **use** in the program order of some processor. No operation which appears between  $op$  and the **use** can change the value which was set by  $op$ , so this value is yielded by the **use**. It is clear from the definition of the expression that  $op$  appears in the same block as the **use**. Thus, in the constructed serialization no operation by another processor may intervene between  $op$  and the **use**. Let us look at the possibilities:

- $op$  is **assign**. In the serialization constructed above, the result of this **assign** is seen by the **use**. Since in the original Java execution, the value provided by this **assign** is seen by the **use**, we conclude that the **use** sees the same value in the original execution and in our serialization.
- $op$  is **load**. In  $T$ , the value brought by **load** was read by a corresponding preceding **read** operation (Constraint A.4.2), which, in turn, sees the result written by the closest preceding **write**. The **write** corresponds to a **store** instruction by some processor (Constraint A.4.3). In the order of memory accesses in the original execution, this **write** is the closest to our **read**, and thus, in the serialization, it resides in the closest **store-block** to the **load-block** that contains our **load** and **use** instructions. Since there are no **assigns** in the blocks that can be placed between these two blocks, the last **assign** performed by the **store-block** is seen by our **use** in the serialization. Now, in the original execution, the last **assign** of the **store-block** provides the value to be written by the **store** of that block, and its value is seen by the **use**. We conclude that both in the original execution, and in our serialization, the **use** sees the value provided by the same **assign**.

■

As mentioned above, execution examples for both Java and other models use a `READ` instruction to refer to either `use` or `READ`, depending on which model is being considered. Similarly, `WRITE` corresponds to either `assign` or `WRITE`.

For brevity, we sometimes denote `use` as `u`, `assign` as `a`, `load` as `l`, `store` as `s`, `read` as `r` and `write` as `w`.

## 2 Java Consistency and Conventional Models

In this section we compare Java Consistency to some other models that appear in the literature. The section is organized as follows. Section 2.1 shows that Java is coherent. Section 2.2 shows that Java is incomparable with PRAM Consistency. Section 2.3 shows the Java is incomparable with both versions of Processor Consistency.

In addition, we will show in Section 3.2.2 that Java is incomparable with Causality (This discussion is deferred to Section 3 for the sake of readability).

### 2.1 Java vs. Coherence

The definition of Coherence, as in [2], is:

**Coherence:** A history  $H$  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 history under it is coherent. The corresponding memory model is called *Coherence*.

**Theorem 2.1** *Java Consistency is stronger than Coherence.*

In order to prove the theorem we will show two things: first, that Java is not weaker than Coherence, i.e., that any execution that is valid for Java is also valid for Coherence, and second, that Java is not equivalent to Coherence, i.e., that there exists an execution that is valid for Coherence but is not valid for Java.

**Claim 2.1** *Java is coherent, i.e., for each Java execution  $H$  and a variable  $x$  there is a global serialization of  $H|x$  which is consistent with the views of all the processors.*

**Proof** If  $H$  is a Java execution, there exists a timing of all `u/a/l/s/r/w` instructions, consistent with all Java constraints. Consider the set of operations on some variable  $x$  in  $T$ .

In order to show the required serialization, for each local history we divide the sequence of operations into blocks, and then arrange the blocks in a global order between all the processors. The division is specified by the following regular expression.

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

First, we show that the expression covers all possible executions. We need to show that when a block terminates, the next operation in the sequence will be `assign` or `load`. A new block begins at the beginning of a sequence or when the previous block terminates. If the execution is finite, a block may terminate prematurely.

**Operations performed by the main memory.** They are: `read` and `write`. These operations are initiated by the main memory as a result of `loads` and `stores`. `Read` actually reads the value that is yielded by the `load` instruction, and `write` stores the value brought by the `store` instruction.

**Locking operations.** They are: `lock` and `unlock`. They serve for synchronization of memory and program control flow. There are explicit `lock` and `unlock` operations in the bytecode, and they are performed in tight coupling with the main memory agent.

These operations can be seen as executing in different layers. The `use` and `assign` operations follow immediately from the source code of the Java program. We say that the *program order* of a thread is the order of the `use` and `assign` instructions it issues. These instructions are generated by the compiler from the source code according to the semantics of the Java language. They are local to the processor, and their generation is thus independent of memory behavior constraints, and does not interfere with the memory consistency specification. The `load` and `store` instructions are inserted into the bytecode by the compiler according to the constraints between `uses/assigns` and `loads/stores`. These constraints constitute the *upper layer*.

When the program is executed on a JVM, the `load` and `store` instructions in the bytecode initiate the execution of `read` and `write` instructions, which are performed in the main memory. The set of constraints that govern the relation between the `loads/stores` and the `reads/writes` constitutes the *lower layer*, or the *implementor view* (as this is the memory behavior which must be implemented for the JVM). This set of constraints will be denoted as *Java'* in Section 4.

The full set of constraints from both the lower and the upper layers determines the *programmer view* of Java, and is called below *Java Consistency*, or simply *Java*. Since the programmer explicitly influences only the `use` and `assign` instructions, we are interested in how these instructions can be seen by other processors. The `use` and `assign` instructions are local, so a local `use` sees the result of a local `assign`. The result of the `assign` is only seen by remote `uses` (which access the same variable) at other processors when it is propagated to them by a sequence of `store-write-read-load` operations. A local `use` instruction is typically not seen by remote processors since it produces nothing that can be propagated.

All the constraints are quoted in the Appendix A, and will be referred to below by their section number and index within the section. For example, *Constraint A.2.1* means Constraint 1 in Section A.2 of the Appendix.

## 1.5 Java Execution, Java Consistency and Notation

The standard notation in the literature denotes the operations that are executed locally by a processor as `READ` and `WRITE`. Since these operations are originally denoted in JLS as `use` and `assign`, respectively, and since `read` and `write` are used in JLS for operations performed by the main memory, we chose to present Java executions by means of sequences of `uses` and `assigns`.

A *timing* for a Java execution determines, for each operation, the time-step in which it is performed, where each operation is assumed to take a single time-step.

**Java Consistency:** An execution consisting of `uses` and `assigns` belongs to Java Consistency (or, Java), when there is an assignment of `loads` and `stores` to the processor parts, and an assignment of `reads` and `writes` to the main memory part, and a timing for the execution and the additional `load/store/read/writes` which complies with the Java constraints.

for the consistency would specify the set of executions that are possible according to the constraints. Throughout the paper we introduce consistency models in both ways. Loosely speaking, they refer to operational and non-operational definitions, accordingly.

We say that consistency  $A$  is *stronger* than consistency  $B$  if the set of histories possible under  $A$ ,  $H_A$ , is contained in the set of histories possible under  $B$ ,  $H_B$  ( $H_A \subseteq H_B$ ). Consistency  $A$  is *strictly stronger* than  $B$  if  $H_A \subset H_B$ .  $A$  is said to be *incomparable* with  $B$  when it is neither stronger nor weaker (this term is taken from [2]).  $A$  is *equivalent*, or *equal*, to  $B$  when  $H_A = H_B$ . Alternatively, definition  $A$  is stronger than definition  $B$  if it has more restrictions on the set of possible variable updates. Similar alternate definitions exist for the other relations as well.

All READS and WRITES operate on individual variables. The variables are assumed to be of some built-in atomic type in the machine architecture. For instance, *objects*, which are regarded as variables in some object-oriented languages, are not considered variables here, since an object may consist of many atomic variables.

One important model is Sequential Consistency (SC) [11].

**Sequential Consistency:** A history  $H$  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  $H$  and  $o_1 \xrightarrow{po} o_2$  then  $o_1 \xrightarrow{S_x} o_2$ . In other words, there is a serialization of  $H$  which is consistent with the views of all the threads. A memory model is called Sequential Consistency when every execution under this model is sequentially consistent.

## 1.4 The Java Virtual Machine (JVM)

As mentioned above, the memory behavior in the Java Language Specification (which we call JLS, see Appendix A.1 and Chapter 17 in [7]) defines Java by 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* (here referred to as *processors*, as explained below), 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. For the sake of consistency with previous works, we use the term *processor*, which is the term used in the definition of most conventional consistency models. Thus, when talking about Java, the term *processor* refers to the notion of a thread.

The operations are divided into four classes:

**Operations local to the thread engine.** The operations are `use` and `assign`. `use` loads the local copy of a variable into the engine for some calculation, and `assign` writes the result of the calculation into the local copy of the result variable. There are no individual `use` and `assign` instructions in the bytecode, but the operations specified by the bytecodes use several such instructions during the execution. For example, `x=y+z` in the Java source code implies bytecode instruction `add`, which 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. These operations are represented by explicit instructions in the bytecode.<sup>1</sup>

---

<sup>1</sup>Their corresponding JVM mnemonics are `get field` and `set field` [14].

of legal serialization, and another, more “constructive” in nature, which is based on relations which can be calculated directly for a given execution.

The JVM can be 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 and then executes it. We are interested, in either case, in the exhibited memory behavior; the possible optimizations do not concern us here. The internals of the implementation, and in particular whether it is 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 definitions and explain the Java memory model. In Section 2 we compare Java to some conventional consistency models that appear in the literature. Section 3 introduces two equivalent non-operational consistency models for the implementor level, called JavaL and JavaD, and two such models for the programmer level, JavaL<sup>T</sup> and JavaD<sup>T</sup>. Section 4 provides the proofs that JavaL and JavaD represent the memory model for the implementor of the JVM, and that JavaL<sup>T</sup> and JavaD<sup>T</sup> represent such a model for the Java programmer. Section 5 discusses issues related to strong operations such as volatiles and locks. Section 6 gives our conclusions. For completeness, we also provide in Appendix A the original list of constraints from [7] for the implementation of Java on top of the AMS.

### 1.3 Definitions and Conventions

There are three types of instructions in this paper. We distinguish them by font, as follows.

**Java code** – a typewriter font will be used: use, assign, load, store, read, write.

**Code in other memory models** – a small caps font will be used: READ, WRITE.

**Code for both Java and other models** – once again, a small caps font will be used.

In our examples we always assume that the variables are initialized to 0. We also assume that a READ can identify a WRITE, e.g., by assuming that each of them writes a different value. The instructions in the examples are indexed by the processor name and the instruction index in the processor’s local program, so instruction  $i$  in the program of Processor  $j$  is denoted as  $j.i$ .

We denote  $o_1 \xrightarrow{XXX} o_2$  when  $XXX$  is a set of relations and  $o_2$  is dependent on  $o_1$ , or if  $XXX$  is a sequence of instructions and  $o_1$  appears before  $o_2$  in the sequence.

We follow the definitions presented in [2].

A *local history* of a processor  $p$ , denoted  $H_p$ , is a sequence of READ and WRITE operations, denoted  $o_1, o_2, \dots$ , that are performed by  $p$ . READ  $x,v$  denotes a read operation, where the source variable is  $x$  and the operation yields  $v$  as its result. WRITE  $x,v$  denotes a write operation, where the destination variable is  $x$ , and the value to be written is  $v$ . The order of operations in  $H_p$  defines the *program order*,  $po$ . A *global history*, or just *history*  $H$ , is a collection of all local histories for the execution. In what follows we sometimes refer to history as *execution*, and use these terms interchangeably.

For a given history  $H$  and a processor  $p$ , denote as  $H_{p+w}$  the partial history consisting of all the operations of  $p$  and all the WRITE operations of other processors. Denote also  $H|x$  the partial history of  $H$  consisting only of operations on the variable  $x$ .

A *serialization*  $S$  of the history  $H$  is a linear sequence containing all the operations of  $H$ . A serialization is *legal* if each READ operation returns the result of the latest WRITE to the corresponding variable.

A *memory consistency model* (or simply, *consistency*) for a system of processors which use a collection of variables is a set of constraints (or, a governing protocol) on the way the variables are modified, and/or the way these modifications are seen by the processors. An alternative definition

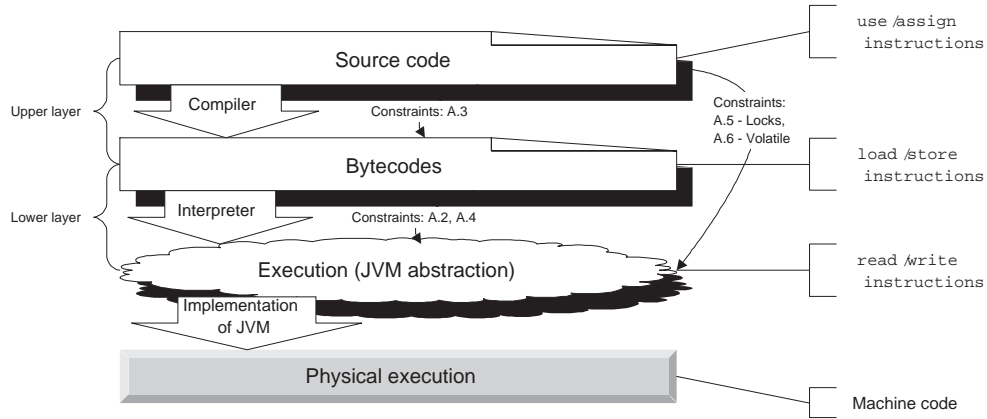


Figure 1: The layered structure of the Java programming paradigm.

definition gives the relations between the views of different processors at the same level.

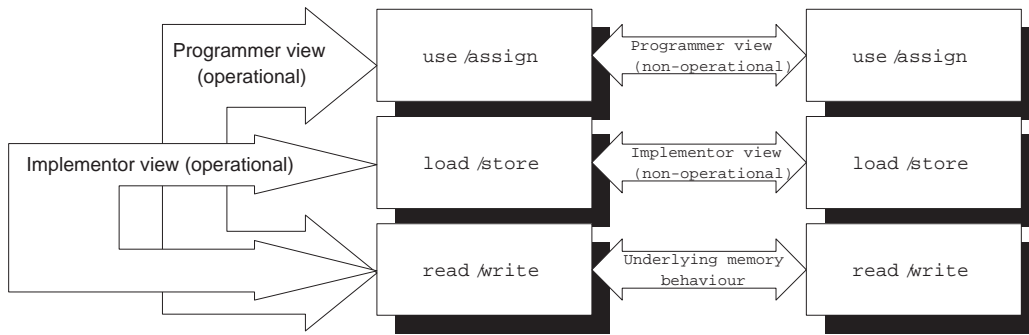


Figure 2: Programmer and implementor views in Java: Operational vs. Non-Operational.

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. Comparing them to Java may assist in the selection of those programs and algorithms which can be adapted to Java even though they may be using 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. The new definitions are relatively simple, and are useful for guiding programmers in how to use the shared memory efficiently, and in the design of efficient virtual machines for distributed environments. In particular, our results imply that both views resemble a combination of variants of Coherency and Causality, and that the programmer model is weaker than the implementor model.

Prior to presenting the new definitions, we introduce a relation called *Causal Relation* and two consistency models called *Data Dependency (DD)* and *Legal Serialization (LS)*. We then prove a lemma which draws a tight connection between DDs and LSs. Using this basic lemma, we provide two definitions for each of the programmer and implementor views: one which is based on the notion

# 1 Introduction

## 1.1 Background

One of the interesting and useful features of Java is its built-in concurrency support through multithreading, a feature that can be exploited for several purposes [12]. Programs can use multithreading so that different threads may execute in parallel, performed by different processors or different machines. Since the threads use shared memory, executing on a distributed environment requires that the system provide distributed shared memory (DSM). In order to execute efficiently in parallel on such an environment the conditions on the coherency of the shared memory are typically weakened, thus preventing the “implicit” communication that shared memory tends to provoke.

A Java system consists of a compiler which, given the source code, produces bytecodes that are platform-independent and are thus absolutely portable, and an interpreter, called the *Java Virtual Machine* (JVM), that executes the bytecodes on the chosen 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 book [7] (JLS). Chapter 17 of this book provides specifications for the memory behavior of Java.

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., defining 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 the way multiple copies of the same data maintain coherency (thus implicitly defining the strength of the derived coherency 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 1.1. The resulting consistency model, described by the relationship between the program code and its execution in the processor, as well as the relations between executions on different processors, are rather complicated. Obviously, the definition given in JLS is inconvenient for both the Java programmer and the JVM implementor.

## 1.2 This Work

In this work we are interested in providing useful non-operational characterizations of the Java memory model (Java Consistency, or simply Java). Non-operational models specify just how strongly the shared memory should be kept consistent across processors in a distributed environment. They attempt to keep the specification independent of a specific – even an abstract – machine, and thus are “cleaner”, easier to implement, and simpler to understand than the operational models.

Java two-layered structure results in different memory specifications for the programmer and the implementor. The programmer uses the specifications for the upper layer, whereas the implementor uses those for 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 1.4 below). Note that the original operational specification defines the relation between different levels of operations in the same processor, whereas the non-operational

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	This Work . . . . .	1
1.3	Definitions and Conventions . . . . .	3
1.4	The Java Virtual Machine (JVM) . . . . .	4
1.5	Java Execution, Java Consistency and Notation . . . . .	5
<b>2</b>	<b>Java Consistency and Conventional Models</b>	<b>6</b>
2.1	Java vs. Coherence . . . . .	6
2.2	Java vs. PRAM Consistency . . . . .	8
2.3	Java vs. Processor Consistency (PC) . . . . .	10
2.3.1	Java vs. PCG . . . . .	10
2.3.2	Java vs. PCD . . . . .	11
<b>3</b>	<b>Data Dependency and Legal Serialization</b>	<b>12</b>
3.1	Data Dependency and Legal Serialization . . . . .	12
3.2	JavaL and JavaD . . . . .	14
3.2.1	JavaD vs. Processor Consistency . . . . .	15
3.2.2	JavaL vs. Causal Consistency . . . . .	16
3.3	JavaL <sup>T</sup> and JavaD <sup>T</sup> . . . . .	17
<b>4</b>	<b>Tight Non-Operational Specifications for Java</b>	<b>18</b>
4.1	Compliance to the Java Consistency Model . . . . .	18
4.2	Specification for the JVM Memory Behavior . . . . .	18
4.3	Specification for the Java Consistency . . . . .	21
<b>5</b>	<b>Volatile Variables and Locks</b>	<b>28</b>
5.1	Programmer View . . . . .	28
5.1.1	Locks . . . . .	28
5.1.2	Volatile Variables . . . . .	29
5.2	Implementor View . . . . .	30
5.2.1	Locks . . . . .	31
5.2.2	Volatile variables . . . . .	31
<b>6</b>	<b>Conclusions</b>	<b>31</b>
<b>A</b>	<b>The Java Memory Model Specification</b>	<b>32</b>
A.1	Operations . . . . .	32
A.2	General constraints . . . . .	32
A.3	Constraints inside a thread . . . . .	33
A.4	Constraints between a thread and the main memory . . . . .	33
A.5	Locks . . . . .	33
A.6	Volatile variables . . . . .	33

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

Technion/CS Technical Report #CS0922, November 1997

Alex Gontmakher

Assaf Schuster

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

## Abstract

We provide non-operational characterizations of Java memory consistency model (Java Consistency, or simply Java). The work is based on the operational definition of the Java memory consistency as given in the Java Language Specification [7].

We first compare Java memory behavior to that of some previously studied models, proving that Java is incomparable to PRAM Consistency and to both variants of Processor Consistency; it is neither stronger nor weaker. We show that a programmer can rely on Coherence for regular variables, Sequential Consistency for volatile variables, and Release Consistency when locks are employed.

We then present two non-operational definitions, and show their equivalence to the memory behavior of the Java Virtual Machine (which determines the implementor view of Java). We proceed to give two non-operational definitions and show their equivalence to Java Consistency (which determines the Java programmer view).

**Acknowledgments:** We thank Jan Groth, Ayal Itskovitz, and Avi Mendelson, who took part in discussions that motivated this work.