Characterizations for Java Memory Behavior

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 [6].

We study the relation of Java memory behavior to that of some well known models, proving that Java is incomparable with PRAM Consistency and with both variants of Processor Consistency; it is neither stronger nor weaker. We show that a programmer can rely on Coherence and a certain variant of Causality for regular variables, Sequential Consistency for volatile variables, and Release Consistency when locks are employed.

Proofs are omitted in this extended abstract, see the full version [4].

1. Introduction

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 [7]. Programs can use multithreading so that different threads may execute in parallel.

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 [6] (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. 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 consistency (thus implicitly defining the strength of the derived consistency on any other memory system as well).

In this work we are interested in providing useful non-operational characterizations of the Java memory model (Java Consistency, or simply Java).

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.

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 discusses issues related to strong operations such as volatiles and locks. Section 4 gives some concluding remarks.

1.1. 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. 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 follow the definitions of notions such as *history*, *execution*, etc. as in [2], and omit them here for lack of space.

1.2. The Java Virtual Machine (JVM)

As mentioned above, the memory behavior in the Java Language Specification (which we call JLS, see Chapter 17 in [6]) 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.

Operations performed by the main memory. They are: `read` and `write`. These operations are initiated by the main memory as a result of `load`s and `store`s. `Read` actually reads the value 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.

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

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 `use`s/`assign`s and `load`s/`store`s. 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 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 `use`s (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 full version [4] and are omitted here for lack of space.

1.3. 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 `use`s and `assign`s.

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 `use`s and `assign`s belongs to Java Consistency (or, Java), when there is an assignment of `load`s and `store`s to the processor parts, and an assignment of `read`s and `write`s to the main memory part, and a timing for the execution and the additional `load/store/read/write`s which complies with the Java constraints.

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.

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.

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.*

Claim 2.2 *Java is strictly stronger than Coherence.*

To prove the claim we show in Figure 1 an execution which is valid for Coherence and is invalid for Java. The details are omitted for lack of space, see [4].

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.

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.

Claim 2.3 *Java is not stronger than PRAM.*

Proof Figure 2 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).

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. The `store`s 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:

$$\boxed{\texttt{a x,1}}, \boxed{\texttt{s x,1}}, \boxed{\texttt{a x,2}}, \boxed{\texttt{a y,1}}, \boxed{\texttt{s y,1}}, \boxed{\texttt{s x,2}}$$

A possible local history of Processor 2 could be:

$$\boxed{\texttt{l y,1}}, \boxed{\texttt{u y,1}}, \boxed{\texttt{l x,1}}, \boxed{\texttt{u x,1}}, \boxed{\texttt{l x,2}}, \boxed{\texttt{u x,2}}$$

Now, coupled with the main memory agent, this can produce the timing shown in Figure 3.

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

Claim 2.4 *PRAM is not stronger than Java.*

Proof Figure 4 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. ∎

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 [3] and [5] 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:

<u>PCG:</u> 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.*

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 1. 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 which is given in [2], and the proof of the following claim are omitted here for lack of space. The proof of the claim shows that the execution in Figure 1 which was shown invalid for Java, is valid under PCD.

Claim 2.6 *Java is incomparable with PCD.*

3. Volatile Variables and Locks

Here we consider the consistency guarantees when strong operations are employed, including the use of locks and volatile variables.

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) [3], 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 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 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 3.1 *Java can simulate RC by mapping the special operations one to one:* ACQUIRE *to* `lock` *and* RELEASE *to* `unlock`*.*

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

Theorem 3.2 *The consistency model among volatile variables, namely, Volatile Consistency, is equal to Sequential Consistency.*

4. Concluding Remarks

In this work we provided comparisons of Java Consistency to several well studied models. In a follow up work [4] we 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 also give there two non-operational definitions and show their equivalence to Java Consistency.

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.

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] 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.

[4] A. Gontmakher and A. Schsuter. Java Consistency: Non Operational Characterizations for Java Memory Behavior. Technical Report TR #0922, Computer Sciences Department, Technion, Nov. 1997.

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

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

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

	Processor 1	Processor 2
1	READ X, 1	READ Y, 1
2	WRITE Y, 1	WRITE X, 1

Figure 1. Execution valid for Coherence and invalid for Java. As explained above, for Java READ denotes `use` and WRITE denotes `assign`.

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

Figure 2. An execution valid for Java but invalid for PRAM.

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

Figure 3. A possible timing for a Java execution implementing the program from Figure 2. 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`).

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

Figure 4. A valid execution for PRAM which is invalid for Java.