

The Java Memory Model Simulator

Jeremy Manson and William Pugh, {jmanson,pugh}@cs.umd.edu

University of Maryland, College Park

Abstract. With the advent of modern, multithreaded programming languages, it has become vitally important to describe in a clear and understandable way how threads interact through memory. The existing *Java Memory Model* is fundamentally broken; the process of creating an appropriate replacement has demonstrated exactly how deeply complex these issues can be.

A formal specification [MP01a] of a replacement memory model is being developed as part of the Java Community Process. As part of that effort, we have developed a simulator that reflects the current version of this model. This simulator has proved invaluable in this effort and will be an important tool for those trying to understand the final model.

1 Introduction

Two of the main advantages of the Java programming language are its built in multithreading support and its portability. Unfortunately, these features clash: threading and synchronization do not work the same way on every system, hampering portability. To account for this, the designers of Java had to create a document that would tell Java Virtual Machine (JVM) implementors and Java programmers what behaviors could be expected in all JVMs. This document is usually referred to as the *Java Memory Model* [GJS96, §17].

The concept of a formalized memory model for a programming language is a relatively recent one. Previous work in memory models has focused almost exclusively on hardware architectures. For a good discussion of issues related to memory models in modern processors, see [AG96]. However, memory models for architectures have different goals from those of programming languages. In particular, programming language memory models need to account for the analysis and transformations that might be applied by a compiler.

The current Java Memory Model is fatally flawed [Pug99,Pug00]. A formal specification [MP01a] of a proposed replacement memory model is being developed as part of the Java Community Process. However, understanding the full implications of this model is not easy. In this paper, we present a simulator that can test some of the properties of the model. This simulator can be fed small concurrent programs in a restricted Java-like language; it then reports *all* possible results of those programs according to the memory model.

The simulator can be used in a number of ways. For example, you could feed a program P into the simulator to obtain a set of results R . Then, you

can apply by hand a compiler transformation to convert P into P' , and feed P' into the simulator, giving a set of results R' . The transformation from P to P' is legal if and only if $R' \subseteq R$. The insight here is that while a compiler can do transformations that eliminate possible behaviors (e.g., performing forward substitution or moving memory references inside a synchronized block), transformations must not introduce new behaviors.

We have two different implementations of the simulator; one in Haskell and one in Java. We have a growing collection of more than 50 litmus tests, and we check that the results of the simulators are in line with our expectations and each other.

The simulator provides three important benefits:

- It gives us confidence that the formal model means what we believe, and that we believe what it means.
- As we fine tune and modify the formal model, we gain confidence that we are changing just the things we intend to change.
- The formal model is not easy to understand; it is likely that only a subset of the people who need to understand the Java memory model will understand the formal description of model. Many people, like JVM implementors and authors of books and articles on thread programming, will find this simulator a useful tool for understanding the memory model.

2 The Replacement Java Memory Model

The formal specification is presented in [MP01a]. Space limitations prevent a substantial description of the semantics in this paper, and the reader is referred to the paper that presents the formal semantics. We will just present a small portion of the semantics here to give a flavor of their behavior.

The model is a global system that atomically executes one operation from one thread in each step. This creates a total order over the execution of all operations. Each of these total orders is a possible *program order*. Some operations, such as reads, can non-deterministically perform one of several possible actions. The result of a program run on the model is the combined results over every possible program order and non-deterministic choice.

Within each thread, operations are usually done in their original order. The exception is that writes may be done presciently, i.e., executed early.

2.1 Operations

In this semantics, each operation corresponds to one JVM opcode. A getfield, getstatic or array load corresponds to a Read. A putfield, putstatic or array store corresponds to a Write. A monitorenter opcode corresponds to a lock, and a monitorexit corresponds to an unlock.

The semantics uses sets to model memory behavior. Each thread has a previous and overwritten set associated with it. Each write takes place in

```

performWrite(Write  $\langle v, w, g \rangle$ )
  Assert  $\langle v, w, g \rangle \notin \text{previousReads}_t$ 
  overwritten $_t \cup = \text{previous}_t(v)$ 
  previous $_t + = \langle v, w, g \rangle$ 
  uncommitted $_t - = \langle v, w, g \rangle$ 

```

Fig. 1. Semantics of a Write

two stages: `initWrite` and `performWrite`. The `initWrite` places the value being written into a set previous_t (where t is the thread that performed the write). This set contains all of the writes known by the thread to have occurred previously. When a thread overwrites a variable v , all the writes to v that were known to have occurred previously get added to a set overwritten_t of writes that can no longer be seen by reads within that thread. The full rule, including some changes for prescient writes, but excluding treatment of final fields, can be seen in Figure 1. Due to space constraints, it is impossible to give complete details of the formal rules for the memory model. For further details, the reader is referred to the full paper on the memory model [MP01a].

Each monitor m also has a `previous` and `overwritten` set associated with it. When an unlock is performed, the values in the sets overwritten_t and previous_t are added to the sets overwritten_m and previous_m . When a lock is performed, the values in the sets overwritten_m and previous_m are added to the sets overwritten_t and previous_t . This allows for one thread to communicate the values of its `overwritten` and `previous` sets to another. When a read occurs, the value read must be in the `allWrites` set, but not in the overwritten_t set.

3 Formulation of the Simulator

The primary goal in creating a simulator for the memory model was to ensure that the model fulfilled some of the requirements of a new model without having unintended consequences. In addition, we wished to find an effective way of simulating programming language memory models. To this end, we wrote two simulators, using different approaches. One simulator was written in Java, and the other in the programming language Haskell [PH99]. The implementation decisions reported in this paper apply to both simulators.

The transfer from the model to the simulator is fairly straightforward. The simulator should generate every possible program order. To get the possible results of a program, we simply feed that program into the simulator.

To simulate all of these orderings, the verifier starts with an initial program state. This state consists of initial values for all the sets in the memory model, plus values for the threads' program counters, local variables and monitor states.

For each instruction that can be executed, the simulator creates a copy of the current state, executes the instruction on it, and removes that instruction from the list of actions that can be performed on that state. This process repeats until no states have any instructions left that can be executed.

This raises another question: how are the contents of memory modeled? In contrast to simulators for hardware-based memory models, we represent the contents of memory as a higher level abstraction. Instead of a flat memory space, we represent memory locations as objects and fields. This is a much more intuitive way of representing the memory state of an object oriented environment.

3.1 Control Flow

In order for a programming language to be modeled effectively, it is necessary to provide a mechanism for control flow. Our simulator provides two: a syntax for decision statements, and a syntax for iteration. The decision statements are easy; for these, we simply use an `if ... else ... endif` construction.

Looping adds much more complexity. Instructions must be handled differently if they might be executed more than once. In addition, if a program loops forever, a simulation of it will also loop forever. Since this cannot be reliably detected, our simulator might not terminate. This is an undesirable behavior. For these reasons, universal looping is not currently included in the simulator.

We must therefore limit looping to simple behavior that is known to terminate. Since we are modeling thread interaction issues, we can focus on the use of looping in interthread communication. This generally takes the form of one thread “waiting” for another to complete a task. To effect this, we included a *spin wait statement*. A spin wait statement simply causes a thread to wait until a condition is fulfilled. We model spin waits as never terminating unless they succeed on their first attempt. Since Java provides no fairness guarantees, spin waits that never terminate are legal results. We report executions with threads in non-terminating spin waits as possible results of an execution.

3.2 Prescient Writes and Guaranteed Reads

The verifier does not completely simulate the model. The model allows for automatic placement of prescient writes and replacement of reads with guaranteed reads. For the full details of and justification behind this automatic placement, see the paper on the model [MP01a].

Unfortunately, the legal placement of guaranteed reads and prescient writes depends on global properties of the simulation itself. For example, a prescient write is handled by breaking the write into an initial write (or *initWrite*) and a placeholder for the original write (or *performWrite*). For

the `initWrite` to be placed before program execution, we must know statically that the corresponding `performWrite` will always occur if the `initWrite` does. This is not always possible.

A *guaranteed read* is similar. A guaranteed read is a read that is guaranteed to return the same value as another read, or a value stored by a particular write. For a guaranteed read to be placed correctly, it must be true that the read or write with which it was associated has occurred. This is not always possible to determine statically.

The simulator does not place any guaranteed reads, and only places some of the legal prescient writes. To produce the appropriate behavior, these instructions can be placed by hand.

4 Implementing the Simulator

In order to provide better understanding of the model, we wrote two simulators. By ensuring that the results of each were in line with the other, we provided an important check on the correctness of the model.

```
performWrite globalState
  (threadNum, (value, destVarName, destPtr, guid))
| elem guid (retrievePreviousRead threadNum globalState) = []
| otherwise
  = [(replaceUncommitted threadNum uncommitted_t'
    (replaceOverwrittenT threadNum overwritten_t'
      (replacePreviousT threadNum previous_t' globalState)))]
  where uncommitted_t' = (filter (notSameGUID guid) uncommitted_t)
        overwritten_t' = overwritten_t 'union' previous_t
        previous_t'
          = previous_t 'plus'
            (destPtr, destVarName, value, False, False, guid)
```

Fig. 2. Semantics of a Write in Haskell

4.1 Haskell and Java

The Haskell simulator was written first. This was because of the ease of expressing the actions in Haskell. A slightly simplified example of this can be seen in Figure 2 (the lines of code where `previous_t`, `uncommitted_t` and `overwritten_t` are defined are omitted for brevity). The correspondence between the original rule and the Haskell rule is fairly straightforward.

In contrast, Figure 3 shows a simplified version of the write action in Java. The Java simulator is optimized for speed, not comprehensibility, and

```

void execute(State state, NormalVariable v) {

    Known k = newState.known.get(thread);
    if (k.previousReads.contains(w))
        throw new IllegalProgram(newState, "prescient write was seen");

    State newState = state.advancePC();
    Write vwg = constructWrite(newState, v);
    Known newKnown = newState.known.get(thread);
    newKnown.overwrittenWrites.addAll
        ( State.intersection( state.allWrites.get(v), k.previousWrites));
    newKnown.previousWrites.add(vwg);
    newState.uncommittedWrites.get(thread).remove(g);
    State.addState(newState);
}

```

Fig. 3. Semantics of a Write in Java

is therefore slightly more difficult to read. The additional simplicity of the Haskell simulator can be seen with another metric: the Haskell simulator is only 1887 lines of code, and the Java one is 3807 lines of code.

As a trade-off for its simplicity, the Haskell simulator is much slower than the Java one. As a result, the Haskell and Java simulators really do serve different purposes. The Haskell verifier provides an easily modified base from which changes to the model can be quickly implemented and tested. The Java verifier provides a platform for rapid turnaround when testing multiple or lengthy programs. Many of the implementation decisions reported in this paper apply to both simulators.

4.2 Non-deterministic Choice and Reducing the Search Space

The simulator approximates the non-deterministic behavior of the model; at each step of the execution, there are choices to be made. One choice is which thread should be selected for execution. Within a thread, barring spin wait failure, the next statement is always executed.

In the presence of data races, one of several possible values for a read is chosen non-deterministically. We do not model reordering by general statement reordering rules; instead, much of the effect of reordering within a thread is handled by these non-deterministic reads. Some of the effects of statement reordering cannot be handled by non-deterministic reads, and are instead handled by prescient writes (i.e., `initWrite` instructions), as described in Section 3.2.

Even without data races, the number of possible execution orderings grows very quickly. For t threads of n statements each, the number of possible

execution orders is $\frac{(tn)!}{n!^t}$. Thus, a number of techniques are needed to reduce the size of the search space.

Within the Haskell simulator, we simply enumerate over all the possible choices at each step and explore each. The fact that Haskell is a functional language and none of the data structures are modified makes this particularly easy.

Implementing non-deterministic choice in Java is not as easy. When a new state is generated, care must be taken to ensure that neither the current state nor any other state is modified.

Thread 1	Initially, $i = j = k = 0, p = q.$	Thread 2
1: $p.x = 1;$		3: $q.x = 1;$
2: $i = q.x;$		4: $j = p.x;$

Fig. 4. An Example of a Small Program

The Java implementation performs a number of optimizations designed to allow it to scale to larger simulators. The Java simulator uses a work list of states to be explored, as well as a set of all states seen so far. Each step in the simulator consists of removing a state from the work list, and enumerating all possible next states. Each possible next state is added to the work list unless it has already been seen (for example, in Figure 4, executing statement 1 followed by statement 3 will result in the same state as executing statement 3 followed by statement 1). Maintaining a list of all previously seen states does require us to maintain all previously seen states in memory; however, it does allow for a substantial reduction in the number of states explored.

5 Running the Simulator

We ran 54 programs through the simulator; these ranged in length from 2 to 5 threads, with each thread having anything from 2 to 17 instructions. The results we obtained did not deviate from our expectations from the model.

The model has been in flux for the time we have been developing the simulator. The simulator has helped this process by giving us a systematic way to compare between different versions of the model.

5.1 Performance Results

A precise analysis of the exact number of states these optimizations save the simulator from analyzing is lengthy and of little use in determining how much real savings these optimizations afford us. This is because of the wide variety

Test Name	Optimized			Unoptimized		
	States	CPU	Total Time	States	CPU	Total Time
coherence	23	0:02	0:02	67	0:02	0:02
alpha-3	77	0:03	0:03	364	0:04	0:02
final-2	77	0:04	0:04	379	0:05	0:04
non-atomic-volatiles	209	0:11	0:10	2720	1:01	0:51
PC-5	2277	2:25	1:42	dnf	dnf	dnf
Total	6148	6:19	5:23	dnf	dnf	dnf

Fig. 5. Comparison of Simulator Results

of variables that impact such an analysis; the number of threads, length of threads, number of writes, number of reads that correspond to each of those writes, and the order in which those reads and writes occur in their respective threads all vary widely from program to program.

In this case, the proof of the pudding is in the eating. How much wall clock time were we saved by not examining every state? To answer this question, we ran the Java version of the simulator both with and without the major optimization. The experiments were performed under Sun JDK 1.4.0 on dual 350 MHz Pentium II processors with 1 GB of RAM.

The total results and a small sampling of individual results are summarized in Figure 5. Times are given in (MM:SS) format. CPU time is different from total time because the states were examined concurrently. Any test that ran for more than 24 hours was deemed not to have completed. The programs are generally named after properties for which they test; to get further information, the reader is referred to the test programs themselves, which are available on-line [JMMb].

The results indicate that although the second optimization does not help much for very small programs, the help that it gives for larger examples makes the difference between a feasible and infeasible test. To give a sense of the scale of these programs, the outlier, PC-5, has 4 threads: 2 with 3 instructions and 2 with 6 instructions. final-2 has two threads with 9 and 11 instructions.

6 Related Work

There has been a great deal of work on memory models for computer architectures. The most famous model is probably *Sequential Consistency* [Lam79]. Our model more closely resembles *Location Consistency* [GS98]. A discussion of the differences between various memory models can be found in [AG96].

An alternate proposal for a replacement Java Memory Model was presented in [AMS00]. It is described in the Commit/Reconcile/Fence framework, which was designed for addressing memory model issues on hardware.

A simulator for the CRF model was described in [YGL01]. This work applies the Mur φ verification system [Dil96] to the CRF JMM proposal. As with our work, a suite of litmus tests was used to verify properties of the model. However, the CRF model is different enough from ours that comparisons between the challenges in simulating them are somewhat moot.

[MKLP01] and [MP01b] use ACL2 to verify bytecode. Their aim is to demonstrate the validity of using an operational semantics to specify Java. In doing so, they demonstrate their ability to detect some data races in small programs. However, they make no attempt to simulate the memory model; their verifier assumes sequential consistency, and is therefore not complete.

7 Conclusion

A verifier can provide much needed assurances as to the properties of a memory model. Many useful simulators, based on model checking techniques, exist for architecture level memory models. These techniques do not, however, take advantage of programming level abstractions. The simulator described in this paper treats programming level constructs as such: objects are treated like objects, final and volatile fields are treated with their own semantics.

Our simulator has been valuable both in helping to further the understanding of the Java memory model and in developing changes to it. As our suite of sample programs grows, so does our understanding of the necessary final form of the model.

8 Acknowledgments

We are deeply indebted to David Hovemeyer for his work on a parser for the input language. Many of the issues involved in both the model and the simulator were hashed out on the Java Memory Model mailing list [JMMa]: we are grateful to all participants.

References

- [AG96] Sarita Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
- [AMS00] Arvind, Jan-Willem Maessen, and Xiaowei Shen. Improving the Java memory model using CRF. In *OOPSLA*, pages 1–12, October 2000.
- [Dil96] David Dill. The Mur φ Verification System. In *8th International Conference on Computer Aided Verification*, pages 390–393, 1996.
- [GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.
- [GS98] Guang Gao and Vivek Sarkar. Location consistency – a new memory model and cache consistency protocol. Technical Report 16, CAPSL, Univ. of Delaware, February 1998.

- [JMMa] The Java memory model. Mailing list and web page. <http://www.cs.umd.edu/users/pugh/java/memoryModel>.
- [JMMb] The Java Simulator. Web Page. <http://www.cs.umd.edu/users/jmanson/java.html>.
- [Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 9(29):690–691, 1979.
- [MKLP01] J Strother Moore, Robert Krug, Hanbing Liu, and George Porter. Formal Models of Java at the JVM Level: A Survey from the ACL2 Perspective. In *Workshop on Formal Techniques for Java Programs, in association with ECOOP 2001*, June 2001.
- [MP01a] Jeremy Manson and William Pugh. Semantics of multithreaded java. Technical Report CS-TR-4215, Dept. of Computer Science, University of Maryland, College Park, March 2001.
- [MP01b] J Strother Moore and George Porter. An executable formal java virtual machine thread model. In *Java Virtual Machine Research and Technology Symposium*, April 2001.
- [PH99] Simon Peyton-Jones and John Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language*. <http://www.haskell.org/onlinereport/>, February 1999.
- [Pug99] William Pugh. Fixing the Java memory model. In *ACM Java Grande Conference*, June 1999.
- [Pug00] William Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(1):1–11, 2000.
- [YGL01] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Analyzing the CRF Java Memory Model. In *The 8th Asia-Pacific Software Engineering Conference*, pages 21–28, 2001.