The Sun Hotspot JVM does not conform with the Java Memory Model

Jaroslav Ševčík

April 28, 2008

Abstract

In this paper, we report on our experiment, which shows that Sun's Java Virtual Machine, the Hotspot JVM, does not conform with the current Java Language Specification, namely with the Java Memory Model (JMM). Although we have not been able to observe a behaviour that is forbidden by the JMM directly, we can show that the Hotspot JVM produces code that can lead to such behaviour.

1 Introduction

The Java Memory Model (JMM) describes the semantics of multi-threaded Java programs. The JMM was designed to allow as many program and hardware optimisations as possible while giving reasonable guarantees to programmers, see Gosling *et al.* (2005); Manson *et al.* (2005) for details. In our theoretical work, we have shown that several commonly used optimisations, such as the common subexpression elimination, are illegal in the JMM (Ševčík and Aspinall, 2008).

This paper shows that the violation of the JMM can occur even in practice. First, we show a program that cannot produce certain behaviour in the JMM. When we run this program on the Hotspot JVM, the JVM compiles our program into an assembly code that can exhibit the forbidden behaviour even when executed by a sequentially consistent processor.

2 Experiment

The full listing of the program from our experiment is in Figure 1. By the JMM specification, printing value 1 by the spawned thread (in method `run`) is a forbidden behaviour. We have run our program on the fastdebug server version of the Hotspot JVM on a Pentium D architecture with the Linux operating system[1] with the command-line switch `-XX:+PrintOptoAssembly`

[1]For further details on the runtime system, see the header of the Hotspot JVM's log file in Figure 5.

to output the assembly code into the JVM's log file. The Hotspot JVM compiled methods `thread1` and `thread2` into the assembly code listed in Figures 2 and 3. While the compiled code for method `thread1` retains the meaning from its Java source code, the assembly code of method `thread2` has a slightly different memory behaviour, and its meaning is essentially equivalent to the Java code in Figure 4. After this compilation, the program can print the value 1 in the spawned thread even when executed sequentially consistently[2]. This violates the Java Memory Model.

We should mention that the JVM compiled the method `test2` consistently into the program from Figure 4 only in about half of our attempts. This is because the Hotspot JVM's optimisations are guided by runtime measurements and several other heuristics that differ from run to run (Click, 1995).

3 Conclusion

We have demonstrated that the Sun's own Java Virtual Machine does not conform with the current Java Memory Model. This shows that either the Java Specification or the Hotspot JVM should be fixed. We believe that in our example, the Hotspot JVM only performs standard transformations that are safe for multi-threaded programs, i.e., they provide the guarantee of sequential consistency for correctly synchronised programs. Therefore, we suggest that it is the Java Memory Model that should be revised.

References

Click, C. (1995). Global code motion/global value numbering. *SIGPLAN Not.*, **30**(6), 246–257.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *Java(TM) Language Specification, The (3rd Edition) (Java Series)*, chapter Threads and Locks, pages 557–573. Addison-Wesley Professional.

Manson, J., Pugh, W., and Adve, S. V. (2005). The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 378–391, New York, NY, USA. ACM Press.

Ševčík, J. and Aspinall, D. (2008). On validity of program transformations in the Java Memory Model. Accepted to ECOOP 2008.

[2]For further discussion of the example, please see Ševčík and Aspinall (2008).

```
public class Tester extends Thread
{
  public static int x = 0;
public static int y = 0;

    public static volatile int z = 0;

    public static void main(String [] args)
    {
        Tester t = new Tester();

        int acc = 0;

        for(int i=0; i<10000000;i++) acc += thread1(t);
        for(int i=0; i<10000000;i++) acc += thread2(t);

        x = 0;
        y = 0;
        t.start();

        int r = thread1(t);
        System.out.println("Main thread = " + r);
    }

    public void run()
    {
        int r = thread2(this);
        System.out.println("Thread 2 = " + r);
    }


    public static int thread1(Object snc)
    {
        int r1 = x;
        y = r1;
        return r1;
    }

    public static int thread2(Object snc)
    {
        int r1 = z;
        int r2 = y;
        x = (r2==1)?y:1;
        return r2;
    }
}
```

Figure 1: This program cannot print '`Thread 2 = 1`' in Java.

```
#
#  int ( java/lang/Object * )
#
#r000 ecx   : parm 0: java/lang/Object *
# -- Old esp -- Framesize: 16 --
#r045 esp+12: return address
#r044 esp+ 8: pad2, in_preserve
#r043 esp+ 4: pad2, in_preserve
#r042 esp+ 0: Fixed slot 0
#
abababab   N1: # B1 &lt;- B1  Freq: 10001
abababab
000   B1: # N1 &lt;- BLOCK HEAD IS JUNK   Freq: 10001
000    PUSHL  EBP
SUB    ESP,8 # Create frame
007    MOV    EBX,#440
00c    MOV    EAX,[EBX + precise klass Tester: 0x097a06f8:Constant:exact *] ! Field Tester.x
012    MOV    EBX,#444
017    MOV    [EBX + precise klass Tester: 0x097a06f8:Constant:exact *],EAX ! Field Tester.y
01d    ADD    ESP,8 # Destroy frame
POPL   EBP
TEST   PollPage,EAX ! Poll Safepoint

027    RET
027
```

Figure 2: The assembly code for method `thread1`.

```
#
#  int ( java/lang/Object * )
#
#r000 ecx   : parm 0: java/lang/Object *
# -- Old esp -- Framesize: 16 --
#r045 esp+12: return address
#r044 esp+ 8: pad2, in_preserve
#r043 esp+ 4: pad2, in_preserve
#r042 esp+ 0: Fixed slot 0
#
abababab   N1: # B1 &lt;- B1  Freq: 10001
abababab
000   B1: # N1 &lt;- BLOCK HEAD IS JUNK   Freq: 10001
000    PUSHL  EBP
SUB    ESP,8 # Create frame
007    MOV    EBX,#444
00c    MOV    ECX,#440
011    MEMBAR-acquire
011    MOV    [ECX + precise klass Tester: 0x09865020:Constant:exact *],#1 ! Field Tester.x
01b    MOV    EAX,[EBX + precise klass Tester: 0x09865020:Constant:exact *] ! Field Tester.y
021    ADD    ESP,8 # Destroy frame
POPL   EBP
TEST   PollPage,EAX ! Poll Safepoint

02b    RET
02b
```

Figure 3: The assembly code for method `thread2`.

```
    public static int thread2(Object snc)
    {
        x = 1;
        int r2 = y;
        return r2;
    }
```

Figure 4: Optimised method `thread2`.

```
<?xml version='1.0' encoding='UTF-8'?>
<hotspot_log version='160 1' process='8604' time_ms='1197284856453'>
<vm_version>
<name>
Java HotSpot(TM) Tiered VM
</name>
<release>
1.7.0-ea-fastdebug-b16-fastdebug
</release>
<info>
Java HotSpot(TM) Tiered VM (1.7.0-ea-fastdebug-b16-fastdebug) for linux-x86,
built on Jul 20 2007 02:07:04 by &quot;java_re&quot; with gcc 3.2.1-7a (J2SE release)
</info>
</vm_version>
<vm_arguments>
<args>
-XX:+PrintOptoAssembly
</args>
<command>
Tester
</command>
<launcher>
SUN_STANDARD
</launcher>
<properties>
java.vm.specification.version=1.0
java.vm.specification.name=Java Virtual Machine Specification
java.vm.specification.vendor=Sun Microsystems Inc.
java.vm.version=1.7.0-ea-fastdebug-b16-fastdebug
java.vm.name=Java HotSpot(TM) Tiered VM
java.vm.vendor=Sun Microsystems Inc.
java.vm.info=mixed mode, sharing
<!-- paths removed -->
sun.java.launcher=SUN_STANDARD
</properties>
```

Figure 5: The header of the `hotspot.log` file.