Aalto University
School of Science

Java Memory Model and Data Races

CS-E4110 Concurrent Programming

Keijo Heljanko

Department of Computer Science
Aalto University School of Science

November 2nd, 2016

Slides by Keijo Heljanko

Java Memory Model

The Java Memory Model (JMM) is needed to:

- Give programmer guarantees on the behaviour of a concurrent program
- Give a sound way to compile programs to the assembly language (e.g., x86, ARM, Power)
- Give a way decide which compiler optimizations are sound
- Give security guarantees for code: The Java security (sandboxing) depends on sound static analysis of arbitrary Java Bytecode, which might contain data races

Java Memory Model - Data Races

The Java Memory Model (JMM) guarantees that:

- Data race free programs have exactly the same behaviour as if the Java program would have been run under the sequentially consistent memory model
  - How are data races defined?
  - How difficult is it to decide whether a program has a data race?
  - What kind of tools are available to detect some of the data races?

Java Memory Model - Short Version

Java Memory Model happens-before - Short version following Java Concurrency in Practice by Brian Goetz:

- Program order rule: Each action in a thread happens-before every action in that thread that comes later in the program order
- Monitor lock rule: An unlock on a monitor lock happens-before every subsequent lock on the same monitor lock (both implicit and explicit locks are treated the same)
- Volatile variable rule: A write to a volatile field happens-before every subsequent read of that same field (atomic variables are also treated like volatiles)

Java Memory Model - Short Version (cnt.)

Java Memory Model happens-before - Short version following Java Concurrency in Practice by Brian Goetz:

- Thread start rule: A call to `Thread.start` happens-before every action in the started thread
- Thread termination rule: Any action in a thread happens-before any other thread detects the thread has terminated
- Interruption rule: A thread calling `interrupt` on another thread happens-before the interrupted thread detects the interrupt

Java Memory Model - Short Version (cnt.)

Java Memory Model happens-before - Short version following Java Concurrency in Practice by Brian Goetz:

- Finalizer rule: The end of a constructor for an object happens-before the start of the finalizer for that object
- Transitivity: If a happens-before b, and b happens-before c, then a happens-before c

Java Memory Model - Data Races

Java Memory Model - Data Races

- Data race freedom: A Java program has a data race if and only if it has a sequentially consistent execution that contains two accesses in different threads to a variable that are not ordered by the happens-before relation in either order, and at least one of the accesses is a write
- A program with no data races is called correctly synchronized, also called data race free (DRF)

Java Memory Model - Underapproximation of races

- Concurrent access race definition (underapproximation, exact for almost all Java programs but not exact for Java due to subtle issues in the JMM definition[1]): A Java program has a data race if and only if it has a sequentially consistent execution that contains two concurrent accesses in different threads to a variable, and at least one of the accesses is a write
- Only listed here because of the discrepancy in the JMM data races for programs with so called “covert communication channels” between threads - communication between threads not inducing a happens-before relation in JMM

[1] Andreas Lochbihler: Java and the Java Memory Model - A Unified, Machine-Checked Formalisation. ESOP 2012: 497-517
Olli Saarikivi, Keijo Heljanko: Reporting Races in Dynamic Partial Order Reduction. NFM 2015: 450-456

Java Memory Model - Underapproximation of races

Java Memory Model - Underapproximation of races

- The program below has a data race between the assignment of `y = 1` and reading of `y`. This race has no concurrently enabled accesses of `y`!!!
- Because these two accesses can be executed in a sequentially consistent execution but they are not ordered by happens-before (even when there is communication from the thread on the left to the thread on the right!)
- Therefore `r = y` can be assigned value `0` even in the case statement `y = 1` has been executed!

```
initially: x = new Thread(); y = 0
```

Thread 1:	Thread 2:
`y = 1;`	`try { x.start(); }`
`x.start();`	`catch (Exception e) { r = y; }`

Java Memory Model - Underapproximation of races (cnt.)

Java Memory Model - Underapproximation of races (cnt.)

- Java memory model can be quite counterintuitive:
  - `r = y` is only ever executed if `y = 1` assignment has been executed before in the same sequentially consistent execution
  - However, because these accesses are not ordered by the happens-before relation (starting a thread gives covert information to the neighoring thread that the thread `x` has already been started), `r` can be assigned to either the initial value or the value written by `y = 1`, resulting in two different outcomes, i.e., a data race

```
initially: x = new Thread(); y = 0

Thread 1:                Thread 2:

y = 1;                   try { x.start(); }
x.start();               catch (Exception e) { r = y; }
```

Java Memory Model - Correctly synchronized programs

- Correctly synchronized programs exhibit only sequentially consistent executions
- Programs that are not correctly synchronized exhibit weak memory model semantics defined by the happens-before relation and an additional constraint disallowing “out-of-thin-air” behaviours, see:
  `http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5`

Java Memory Model - Out of Thin Air

Out of thin air example

- The happens-before relation will not order any of the reads or writes below:

Initially x = y = 0

Thread 0	Thread 1

r1 = x;
if (r1 == 42) {
  y = 42 };
	r2 = y;
	if (r2 == 42) {
	  x = 42 };

- Because reads and writes are completely unordered by happens-before, just using happens-before would let the write of 42 in thread 0 be read by the first read in thread 1 and vice versa. (“A cyclic data dependency!”) This is disallowed by JMM: In addition to happens-before also no out of thin air executions are allowed!

Java Memory Model - Out of thin air (cnt.)

Out of thin air example (cnt.)

- Such “out of thin air” memory access cycles are disallowed by the Java memory model in addition to happens-before (Actually C/C++ memory model currently allow these executions, a fix to the standard is being proposed to!)
- The out of thin air requirement is needed for Java sandbox security to work!

Java Memory Model - Programs with Data Races

Why should we care about programs with data races?

- Security: If there would be no execution semantics for programs with data races, they could break the Java security model by generating a data race and the behaving in an uncontrolled (undefined) fashion to break out of the Java sandbox
- Ultra high performance: For some algorithms data races can be mundane, and avoiding locking improve performance. However, programs with races are really hard to reason about in the JMM!!!
- Ultra high performance code can often be made in a fully race free fashion, see for example lock-free and wait-free algorithms in `java.util.concurrent`, e.g., `ConcurrentLinkedQueue`
- Such lock-free algorithms are often implemented using `java.util.concurrent.atomic`, especially the `compareAndSet` method

Java Memory Model - Programs with Data Races

Semantics of programs with data races

- A read `r` of a variable `v` is allowed to observe a write `w` to `v`, if:
  - `r` is not ordered before `w` in the happens-before order, and
  - there is no intervening write `w'` to `v` (i.e, no write `w'` to `v` exists such that `w` happens-before `w'` and `w'` happens-before `r`

Java Memory Model - Data Races and Compilers

Common compiler optimizations can be unsound for programs with data races

- Problem: Many common compiler optimizations are unsound for programs with data races
- Jaroslav Sevcík, David Aspinall: On Validity of Program Transformations in the Java Memory Model. ECOOP 2008: 27-51
- Some compilers like Sun HotSpot compiler has had bugs due to enabling optimizations that are only valid for sequentially consistent programs (i.e., data race free Java)

Java Memory Model - Data Races and Compilers

Compiler problems for programs with data races

- See following table from Jaroslav Sevcík, David Aspinall: On Validity of Program Transformations in the Java Memory Model.

Table 1. Validity of transformations in the JMM.

Transformation	SC	JMM	JMM-Alt
Trace-preserving transformations	✓	✓	✓
Reordering normal memory accesses	×	×	✓
Redundant read after read elimination	✓	×	×
Redundant read after write elimination	✓	✓	✓
Irrelevant read elimination	✓	✓	✓
Irrelevant read introduction	✓	×	×
Redundant write before write elimination	✓	✓	✓
Redundant write after read elimination	✓	×	×
Roach-motel reordering	×(✓for locks)	×	×
External action reordering	×	×	×

Race Detection Complexity

Race Detection Complexity

- If we could cheaply check if a program is correctly synchronized (i.e., it has no data races), we could enable much more aggressive optimizations in our Java compiler
- We could also warn reliably a programmer about the data races in his/her code
- Unfortunately checking whether a program has a data race is as difficult as checking whether a Java subroutine terminates
- Unfortunately that means the problem is as hard as the Turing machine halting problem as we can allocate an arbitrary amount of heap space to simulate an arbitrary log Turing machine tape
- Thus the problem of deciding whether a program has a data race is an undecidable problem

Race Detection Tools - Happens-before

- Static race detection would be useful: Given a program, decide whether it has a data race or not. Some tools do exist that analyze Java programs and try to approximately decide this problem: RCC/Java and ESC/Java. Both require code annotations and possibly generate false warnings
- We can resort to race detection: Run a test run of a program, and try to detect whether the program has a data race based on the information gathered during the test run
- To precisely decide which test runs exhibit data races we can run the program, and construct the happens-before relation for all variable

Race Detection Tools - Happens-before (cnt.)

Race Detection Tools - Happens-before (cnt.)

- The problem is that tracking the exact happens-before relation is very expensive but does not generate any false alarms
- The Fasttrack algorithm[2] is a fast new happens-before based race detection method, exploiting the fact that one can for many programs keep the happens-before relation represented with simple data structures and algorithms, and only for complex programs pay the cost of fully tracking the happens-before relation, with reported average of 8.5 times slowdown
- Fasttrack is implemented in the Roadrunner tool (`http://cs.williams.edu/~freund/rr/`)

[2] Cormac Flanagan, Stephen N. Freund: FastTrack: efficient and precise dynamic race detection. Commun. ACM 53(11): 93-101 (2010)

Race Detection Tools - Locksets Algorithm

Race Detection Tools - Lockset based algorithms

- The Eraser algorithm[3] is a classical algorithm for finding data races using "Locksets"
- The basic idea: For each shared variable keep track of which locks are held at each point in time when the variable is accessed
- Each time the variable is accessed, its lockset is intersected with the locks being held at the point in time of the variable access
- If the lockset for any variable becomes empty, the variable does not follow a strict locking scheme that requires that each variable is protected by at least one lock

[3] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas E. Anderson: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Trans. Comput. Syst. 15(4): 391-411 (1997)

Race Detection Tools - Eraser State Machine

Race Detection Tools - Eraser state machine

- To minimize false data race alarms, Eraser algorithm keeps track of which variables are accessed by several threads such that at least one of the accesses is a write:

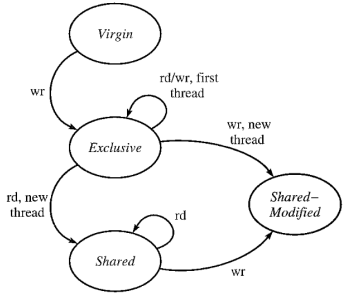

Fig. 4. Eraser keeps the state of all locations in memory. Newly allocated locations begin in the *Virgin* state. As various threads read and write a location, its state changes according to the transition in the figure. Race conditions are reported only for locations in the *Shared-Modified* state.

Race Detection Tools - Eraser state machine (cnt.)

- Eraser reports a data race if a variable in “Shared-Modified” state has an empty Lockset

Race Detection Tools - Locksets Algorithm (cnt.)

Race Detection Tools - Lockset based algorithms (cnt.)

- Variable might be protected by one lock in one phase of the program execution and by another lock in another phase of the execution
- Thus an empty lockset will cause a race detection warning but some of the warnings are false warnings that do not realize as real races - However, empty locksets mean a strict locking regime has been violated
- Programs using lock-free datastructures will have lots of false warnings from locksets
- Locksets can also sometimes detect races in other traces than the one being tested: Sometimes the empty lockset will be symptom of a problem that manifests in another test run

Race Detection Tools - Locksets for Java

- The Racer algorithm[4] is an adaptation of the basic Lockset Eraser algorithm with Java language specific extensions
- The RacerAJ (`http://www.bodden.de/tools/raceraj/`) is an implemention of the Racer algorithm
- Several variants of the original Eraser lockset algorithm are also implemented in the Roadrunner tool (`http://cs.williams.edu/~freund/rr/`)

[4] Eric Bodden, Klaus Havelund: Racer: effective race detection using AspectJ. ISSTA 2008: 155-166

Race Detection Tools - Automated Testing Tools

Race Detection Tools - Automated Testing Tools

- One can run automated testing tools such as Java Pathfinder (more on this later in the course) to find data races in approximate manner
- The approach by Saarikivi and Heljanko[5] is an automated testing approach for Java programs that systematically tests all different ways to resolve data races in a Java program (without testing all interleavings!)
- For programs that terminate and that do not have covert communication channels the approach is guaranteed to find a data race iff one exists
- The reason covert communication is disallowed is the fact the tool uses the concurrent access data definition as a way to find races

[5] Olli Saarikivi, Keijo Heljanko: Reporting Races in Dynamic Partial Order Reduction. NFM 2015: 450-456