# The Java Memory Model

Jaroslav Ševčík

University of Edinburgh

Supported by ITI Techmedia

# Overview

**Part I**:

- Motivation for weak memory models:
  - Compromise between standard optimisations and intuitive semantics for concurrency.
- Overview of memory models:
  - Language-level memory models.
  - Hardware memory models.

**Part II**:

- Introduction to the Java Memory Model (JMM) and optimisations.
- Notes on implementation of the JMM.
- The JMM operationally, examples.

# Intuitive Semantics for Concurrency

Intuitively, programmers assume interleaved semantics (also called sequential consistency):

The result of an execution must be the same as if operations of all threads were executed in some sequence that is consistent with the order specified by the program.

# Intuitive Semantics for Concurrency

Intuitively, programmers assume interleaved semantics (also called sequential consistency):

The result of an execution must be the same as if operations of all threads were executed in some sequence that is consistent with the order specified by the program.

. . . where the operations are atomic operations of the program:

- shared memory reads, writes, locks, unlocks, I/O. . .
- thread-local operations are often omitted.

and the result is the observable outcome of the execution, i.e., the sequence of I/O operations.

# Interleaving – Example

For example, consider the program

| x = y = 0 | |
|---|---|
| `y := 1` | `x := 1` |
| `r1 := x` | `r2 := y` |
| `print r1` | `print r2` |

An interleaving of the program might be the sequence:

$$\texttt{y:=1}, \texttt{r1:=x}, \texttt{x:=1}, \texttt{r2:=y}, \texttt{print r1}, \texttt{print r2}$$

In memory model literature, only memory-related actions are included:

$$\mathsf{W}(\mathrm{x}, 0), \mathsf{W}(\mathrm{y}, 0), \mathsf{W}(\mathrm{y}, 1), \mathsf{R}(\mathrm{x}, 0), \mathsf{W}(\mathrm{x}, 1), \mathsf{R}(\mathrm{y}, 1), \mathsf{Ext}(0), \mathsf{Ext}(1)$$

# Interleaving – Example

| x = y = 0 | |
|---|---|
| `y := 1` | `x := 1` |
| `r1 := x` | `r2 := y` |
| `print r1` | `print r2` |

What are the possible results?

# Interleaving – Example

| x = y = 0 | |
|---|---|
| `y := 1` | `x := 1` |
| `r1 := x` | `r2 := y` |
| `print r1` | `print r2` |

What are the possible results?

$\mathsf{Ext}(1), \mathsf{Ext}(0)$

$\mathsf{Ext}(0), \mathsf{Ext}(1)$

$\mathsf{Ext}(1), \mathsf{Ext}(1)$

Observe that the result $\mathsf{Ext}(0), \mathsf{Ext}(0)$ is not possible!

# Optimisations and Concurrency

The problem:

Many standard optimisations that are valid (not observable) for sequential programs, become observable in interleaved semantics. Examples:

- Constant propagation,
- Common subexpression elimination,
- Code motion,
- Write buffering (in hardware).

All of these are performed by most compilers or processors.

# Compilers and Concurrency

Example: what are the possible outputs of the following program?

initially `requestRdy = responseRdy = data = 0`

| Thread 1 | Thread 2 |
|---|---|
| `data := 1`<br>`requestRdy := 1`<br>`// Interleave here`<br>`if(responseRdy == 1)`<br>`  print data` | `if(requestRdy == 1) {`<br>`  data := 2`<br>`  responseRdy := 1`<br>`}` |

# Compilers and Concurrency

In any interleaving, the program

initially `requestRdy = responseRdy = data = 0`

| Thread 1 | Thread 2 |
|---|---|
| `data := 1`<br>`requestRdy := 1`<br>`// Interleave here`<br>`if(responseRdy == 1)`<br>`  print data` | `if(requestRdy == 1) {`<br>`  data := 2`<br>`  responseRdy := 1`<br>`}` |

can only output 2 (if anything).

What if the compiler reuses the value of `data`?

# Compilers and Concurrency

After the constant propagation:

initially `requestRdy = responseRdy = data = 0`

| Thread 1 | Thread 2 |
|---|---|
| `data := 1` | `if(requestRdy == 1) {` |
| `requestRdy := 1` | `  data := 2` |
| `// Interleave here` | `  responseRdy := 1` |
| `if(responseRdy == 1)` | `}` |
| `  print 1` | |

Suddenly the program can output `1`. Tested with `gcj`.

# Hardware and Concurrency

All mainstream multi-core processors (Intel, AMD x86, PowerPC, Intel Itanium) perform write buffering:

- Writes are not written to main memory immediately.
- Instead, writes are stored in a (per processor) write buffer.
- Reads can be satisfied from the buffer.
- Buffer flushed by a special fence instruction or by a write to another memory location.

This optimisation is not observable by sequential programs.

However, this can be observable by multi-threaded programs.

# Hardware and Concurrency Example

For example, recall our ingterleaving example

| x = y = 0 | |
|---|---|
| `y := 1` | `x := 1` |
| `r1 := x` | `r2 := y` |
| `print r1` | `print r2` |

where the result $\text{Ext}(0), \text{Ext}(0)$ was not possible in the interleaved semantics.

However, if the threads only write to write buffers, the result becomes possible! (because the reads will see the initial values in the main memory.)

# Weak Memory Models

Possibilities:

1. Give up some optimisations (and promise the interleaved model).
   - This gives a simple programming model, but...
   - Do you really want to give up constant propagation, CSE, loop optimisations and limit processor pipelines?

# Weak Memory Models

Possibilities:

1. Give up some optimisations (and promise the interleaved model).
   - This gives a simple programming model, but...
   - Do you really want to give up constant propagation, CSE, loop optimisations and limit processor pipelines?
2. Relax the interleaved model a bit.
   - Possibly less intuitive model.
   - Can use (most) standard optimisations.
   - Almost all languages/processors take this path...
   - ...including Java $\Rightarrow$ the Java Memory Model.

# DRF Guarantee

Do we really have to give up the interleaved semantics completely?

# DRF Guarantee

Do we really have to give up the interleaved semantics completely?

No!

. . . because many optimisations are unobservable under interleaved semantics if the program is data race free.

# DRF Guarantee

Do we really have to give up the interleaved semantics completely?

No!

. . . because many optimisations are unobservable under interleaved semantics if the program is data race free.

Indeed, the current trend is to promise interleaved semantics for data race free programs (**DRF guarantee**).

Java promises this!

# Data Race Freedom

What is data race freedom?

Program is data race free if there is no interleaving with a write immediately followed by a memory access to the same (non-volatile) memory location from a different thread.

Note: This is slightly different from the definition in the JMM, but it is equivalent to the JMM definition.

# DRF Guarantee Example I

First, consider the program

| x = y = 0 | |
|---|---|
| `y := 1` | `x := 1` |
| `r1 := x` | `r2 := y` |
| `print r1` | `print r2` |

and observe that it has an interleaving with a data race:

$$\mathsf{W}(y,1), \underline{\mathsf{W}(x,1), \mathsf{R}(x,1)}, \mathsf{R}(y,1), \mathsf{Ext}(1), \mathsf{Ext}(1)$$

# DRF Guarantee Example I

Keep considering the program

| x = y = 0 | |
|---|---|
| `y := 1` | `x := 1` |
| `r1 := x` | `r2 := y` |
| `print r1` | `print r2` |

To make the program DRF, protect shared memory `x`, `y` with locks . . .

# DRF Guarantee Example I

Shared memory `x` protected with `m1`, `y` with `m2`:

| x = y = 0 | |
|---|---|
| `lock m2` | `lock m1` |
| `y := 1` | `x := 1` |
| `unlock m2` | `unlock m1` |
| `lock m1` | `lock m2` |
| `r1 := x` | `r2 := y` |
| `unlock m1` | `unlock m2` |
| `print r1` | `print r2` |

This is DRF because between any two accesses to the same memory there must be an unlock and a lock of the protecting monitor . . .

# DRF Guarantee Example I

Shared memory `x` protected with `m1`, `y` with `m2`:

| x = y = 0 | |
|---|---|
| `lock m2` | `lock m1` |
| `y := 1` | `x := 1` |
| `unlock m2` | `unlock m1` |
| `lock m1` | `lock m2` |
| `r1 := x` | `r2 := y` |
| `unlock m1` | `unlock m2` |
| `print r1` | `print r2` |

. . . so reasonable languages guarantee sequentially consistent behaviours, i.e., it is guaranteed that the program prints $11$ or $01$ or $10$ (but never $00$).

# DRF Guarantee Example II

Program can be data race free even in the absence of synchronisation.

For example, the program

| x = y = 0 | |
|---|---|
| `r1 := x` | `r2 := y` |
| `if (r1 == 1)` | `if (r2 == 1)` |
| `y := 1` | `x := 1` |
| `print r1` | `print r2` |

is DRF because the writes are not executed in any interleaving.

# DRF and Optimisations

Many optimisations are unobservable for data race free programs:

- Redundant read/write elimination:
  - Constant propagation,
  - Common subexpression elimination,
  - Overwritten write removal.
- Reordering of independent statements:
  - Loop optimisations,
  - Code motion.
- Hardware optimisations:
  - Store buffering,
  - Out-of-order execution.

# DRF guarantee-violating optimisations

Not all program transformations that are valid for sequential programs are valid for data race free programs.

For example, loop transformation

```
for(int i=0;                    int tmp = x;
    i<n; i++)        ⇒          for(int i=0; i<n; i++)
  x=x+f(i);                       tmp=tmp+f(i);
                                x = tmp;
```

is valid for sequential programs in any context, but is invalid for parallel programs.

# DRF guarantee-violating optimisations

Not all program transformations that are valid for sequential programs are valid for data race free programs.

To see this, note that the program

Initially, `x = n = 0`

| | |
|---|---|
| `for(int i=0; i<n; i++)` | `x = 1;` |
| `  x = x + f(i);` | `print x;` |

can only print `1` (because the loop body is never executed).

What about the transformed program?

# DRF guarantee-violating optimisations

. . . but the transformed program

Initially, `x = n = 0`

| | |
|---|---|
| `int tmp = x;` | `x = 1;` |
| `// Interleave here...` | `print x;` |
| `for(int i=0; i<n; i++)` | |
| `  tmp=tmp+f(i);` | |
| `x = tmp;` | |
| `// ...and here` | |

can print $1$ and $0$!

Some C compilers do perform such an optimisation!

# DRF as a memory model

DRF guarantee can be viewed as a memory model (Ada, C++0x).

Disadvantages:

- too weak: there are no guarantees for programs with races
  - unacceptable for Java (security, sand-boxing).
  - programmers need more: out-of-thin-air guarantees, isolation, immutability...
- too inflexible: sometimes we do not want to pay the price of enforcing data race freedom
  - For example, performance counters.

# Out-of-thin-air

Programs should never read values that cannot be written by the program(!?).

For example, in

initially `x = y = 0`

| | |
|---|---|
| `r1 := x` | `r2 := y` |
| `y := r1` | `x := r2` |
| `print r1` | `print r2` |

the only possible result should be printing two zeros because no other value appears in or can be created by the program.

# Out-of-thin-air on references

The previous example might seem benign (program can always leak numeric values through non-determinism and arithmetic, in any case).

However, this is not so benign for references:

initially `x = y = null`

| | |
|---|---|
| `r1 := x` | `r2 := y` |
| `y := r1` | `x := r2` |
| `r1.run()` | |

What should `r1.run()` call? If we allow out-of-thin-air, then it could do anything.

# Out-of-thin-air and Optimisations

Out-of-thin-air excludes some program transformations that are valid under the DRF guarantee.

For example, under the DRF guarantee it is legal to speculate on values of writes:

```
                    tmp := y
r1 := x      ⇒      y := 42
y := r1             r1 := x
                    if (r1 != 42) y := tmp;
```

Using this„ our out-of-thin-air example could output $42$!

# Out-of-thin-air and Optimisations

Consider our out-of-thin-air example:

| initially `x = y = 0` | |
|---|---|
| `r1 := x` | `r2 := y` |
| `y := r1` | `x := r2` |

which should never print $42$.

However, if we use the value speculation and rewrite the first thread. . .

# Out-of-thin-air and Optimisations

The transformed program

initially `x = y = 0`

| | |
|---|---|
| `tmp := y` | `r2 := y` |
| `y := 42` | `x := r2` |
| `// Interleave here` | |
| `r1 := x` | |
| `if (r1 != 42) y := tmp` | |

can suddenly print $42$!

This will be theoretically possible in the upcoming revision of C++ (C++0x).

# Isolation

Safe languages should allow compositional semantics for isolated groups of threads — it should be possible for a program to isolate itself from the effects of data races (e.g., in libraries or in unsafe code).

This is still an open research area, but the simplest case can be reasonably well described:

If we have a program consisting of two groups of threads that do not communicate with each other, then the behaviour of the program should be composition of the behaviours of both groups.

# Final Fields

One related issue in Java are final fields and immutable objects.

For instance, programmers assume that instances of `String` never change.

This might be tricky in presence of optimisations.

Consider the program

Initially, `s = s1 = null`

| | |
|---|---|
| `s = "ab"` | `print s1` |
| `s1 = s.substring(1, 1)` | |
| `print s1` | |

# Final Fields

In reality, strings are often implemented as objects containing character buffer (`b`), start index (`s`) and length (`l`). So our program becomes

Initially, `s = s1 = null`

| | |
|---|---|
| `r=alloc(...); r.b="ab"` | `printn s1.b+s1.s,` |
| `r.l=2; r.s=0; s=r` | `s1.l` |
| `r1=alloc(...); r1.b=s.b` | |
| `r1.l=1;r1.s=s.s+1;s1=r1` | |
| `printn s1.b+s1.s, s1.l` | |

(`printn p,n` prints `n` characters, starting from pointer `p`.)

This can still only print `b` (possibly twice), but if the compiler/hardware reorders the statement `s1=r1` earlier . . .

# Final Fields

. . . then we get the program

Initially, `s = s1 = null`

| | |
|---|---|
| `r=alloc(...); r.b="ab"` | `printn s1.b+s1.s,` |
| `r.l=2; r.s=0; s=r` | `s1.l` |
| `r1=alloc(...); s1=r1` | |
| `r1.b=s.b; r1.l=1;` | |
| `// Interleave here` | |
| `r1.s=s.s+1;` | |
| `printn s1.b+s1.s, s1.l` | |

. . . which can print `a` and `b`. So printing the same string could yield two different values. Compilers must prevent such optimisations!

# Summary of Motivations

Memory model is a semantics for concurrent languages with shared memory.

Memory models should:

- allow common compiler and processor optimisations.
  - Validity of compiler transformations should be compositional.
- guarantee sequential consistency for data race free programs.
- prevent out-of-thin-air values.
- allow compositional reasoning about threads (isolation).
- provide support for language constructs (e.g., final fields).

# Memory Model Overview

| Requirement | Java | C++0x | OCaml | C | x86 |
|---|---|---|---|---|---|
| Compiler opt. | Some | More | Few | Many | Few |
| Processor opt. | Many | Many | Few | Many | Few |
| DRF | Yes | Yes | Yes | No | (Yes) |
| Isolation | Yes | No | Yes | No | Yes |
| Out-of-thin-air | Yes | ? | Yes | No | Yes |

This is illustrative rather than exhaustive.

Memory model names: Java = JMM, C++0x = DRFG, OCaml = SC, C = none, x86 = write buffering (Sun TSO).

# Language Memory Models

- No memory model at all.
  - C, Python, Pascal, . . .
  - . . . so programmers must understand the language implementation (processor and compiler) to write correct programs.
  - This might mean including processor-specific memory fences manually.
- No memory model needed (because there is no shared memory).
  - Haskell, Erlang.
- Interleaved semantics.
  - E.g., Caml.

# Language Memory Models II

- The DRF guarantee (undefined behaviour for programs with races).
  - Ada.
  - C++0x specifies DRF guarantee + semantics of low-level atomics.
- Hardware-like memory model (specifies allowed reorderings):
  - .NET language family (unclear)
- Speculative semantics with justifications.
  - Java.

Note: Transactions are not much better because the ideal semantics, strong atomicity, is expensive to implement.

# Hardware Memory Models

- Sun Total Store Order corresponds to write buffering. This model is used in modern UltraSPARC chips.
  - the model is quite restrictive: for example, it does not allow changing order of writes or performing writes earlier.
- Intel IA-32 (x86) model is unclear at the moment, for all practical purposes it seems to be Sun TSO.
- Intel IA-64 (Itanium) memory model allows reordering of syntactically independent statements and different orders and different visibilities for different processors.
- Power and ARM models have semi-formal definitions describing per-processor visibility orders. Precise details not completely clear.

# Hardware Memory Models

Why we do not use a processor memory model for programming languages? Because hardware memory models are too weak:

- Write buffering does not allow eliminations of reads/writes based on previous reads, or any reordering other than write with a subsequent read.
- Itanium and Power memory models disallow reordering of syntactically dependent statements, so the programs

```
r1 = x
if (r1 == r1)
  y = 1
```

and

```
r1 = x
y = 1
```

have different meanings. This is unacceptable for optimising compilers.

# Sources

Intel x86 memory model saga: Sarkar et al. 2009 (POPL).

Power memory model saga: Sewell et al. 2009 (DAMP).

Java Memory Model: Manson, Pugh and Adve 2005 (POPL).

C++0x memory model: Boehm and Adve 2008 (PLDI).

Java Memory Model and Optimisations: Sevcik and Aspinall 2008 (ECOOP).

# To be continued

- Introduction to the Java Memory Model (JMM)
- Overview of transformation legality in the JMM.
- Notes on implementing the JMM.
- Definition of the JMM:
  - overview of the formal definition,
  - operational view of the JMM,
  - examples.
- Flaws in the JMM:
  - several standard optimisations not legal. . .
  - . . . including some that are implemented in HotSpot.