The Java™ Memory Model (JMM)

Foivos Zakkak

27th of March 2017

The University of Manchester

University of Crete

Introduction

- Multi-core processors create multiple copies of data on their cores' caches to improve performance
- Maintaining these copies consistent is not trivial
- Trade-off between performance and consistency
  - Usually we are only interested about data we can observe

Introduction

Different architectures feature different memory models

Type	ARMv7 and Power	X86	AMD64
Loads reordered after loads	✔	✘	✘
Loads reordered after stores	✔	✘	✘
Stores reordered after stores	✔	✘	✘
Stores reordered after loads	✔	✔	✔
Atomic reordered with loads	✔	✘	✘
Atomic reordered with stores	✔	✘	✘
Dependent loads reordered	✘	✘	✘
Incoherent instruction cache pipeline	✔	✔	✘

Source: https://en.wikipedia.org/wiki/Memory_ordering

Introduction

- Some programming languages define their memory models
  - Java™
  - C++11
  - Unified Parallel C (UPC)
- Why?
  - To provide consistent behavior across different platforms

What is a Memory Model ?

- Contract between language or processor designers and developers
- Helps language implementers build the runtime and/or compiler
- Helps developers understand a program’s behavior

The Java™ Memory Model

- Guarantees **sequential consistency** in data-race-free (DRF) programs
  - *"[...] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."* – L. Lamport
- No *out of thin air* values under any circumstances
  - Reads may only observe values written by some write acting on the corresponding variable
- Monitors provide mutual exclusion

The Java™ Memory Model Approach

- Model all legal executions (in algebraic way)
- Map instructions to actions
  e.g. `a=4` maps to a *write action*
- Synchronization actions
  - Release-Acquire pairs
  - Implicit vs Explicit

Release-Acquire pairs

- Thread finish / Thread join – Explicit
- Monitor.wait() / Monitor.notify() – Explicit
- Monitor exit / Monitor enter – Implicit
- Thread.start() / Thread.run() – Implicit
- …

Reads following an acquire action must be able to observe writes performed *before* the corresponding release action

Examples

- Implicit

```
synchronized (this) {
  b = 3;
}

…

synchronized (this) {
  a = b;
}
```

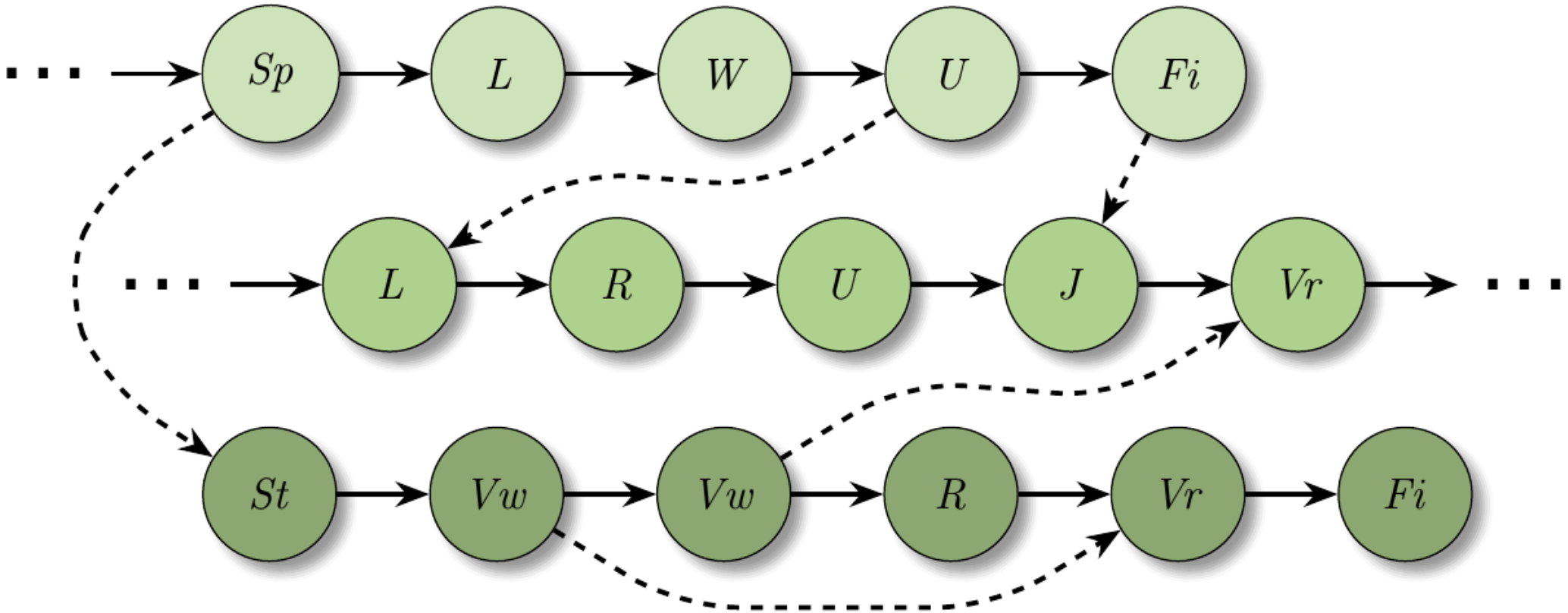
- Explicit

```
b.notifyall();

…

b.wait();

…

t.join();
```

Formal Definition

- Action kinds
  - read, write, wait, join, etc.
  - Grouped in regular and synchronization actions
- Total orders
  - Synchronization order
- Partial orders
  - Program order
  - Synchronizes-with order
  - Happens-before order
- Well-formedness conditions

Actions ordering visualization



Well-formedness conditions

- Each read of a variable sees a write to this variable
- All reads and writes of volatile variable are volatile actions
- The number of synchronization actions preceeding another synchronization action is finite
- Synchronization order is consistent with the program order
- Lock operations (monitors) are consistent with mutual exclusion
- The execution obeys synchronization order consistency
- The execution obeys happens-before order consistency
- Every thread's start action happens before its other actions except for initialization actions

The Java™ Memory Model Limitations

- Defined by observing existing JVMs
- Tailored after shared memory architectures
- Not machine checkable
- Non-intuitive definition

Better Understanding

For in depth understanding of JMM refer to the following resources:

- The JSR-133 Cookbook for Compiler Writers
- JSR-133 Java Memory Model and Thread Specification
- SPECIAL POPL ISSUE: The Java Memory Model (not published)
- Jeremy Manson's Ph.D. Thesis