Software Models

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

***Abstract*—Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

***Index Terms*—Computer design, concurrent computing, correctness, multiprocessing, parallel processing.**

A high-speed processor may execute operations in a different order than is specified by the program. The correct execution is guaranteed if the processor satisfies the condition: the result of an execution is the same as if the operations had been executed in the order specified by the program. A processor satisfying this condition will be called sequential. To consider a computer composed of several such processors sharing a common memory. The customary approach to proving the correctness of multiprocess algorithms for such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of 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. A multiprocessor satisfying this condition will be called sequential. The sequentiality

Shared Memory Consistency Models: A Tutorial *

Sarita V. Adve† and Kourosh Gharachorloo‡

†Department of Electrical and Computer Engineering
Rice University
Houston, Texas 77251-1892

‡Western Research Laboratory
Digital Equipment Corporation
Palo Alto, California 94301-1616

Rice University ECE Technical Report 9512
Western Research Laboratory Research Report 95/7

September 1995

Abstract

Parallel systems that support the shared memory abstraction are becoming widely accepted in many areas of computing. Writing correct and efficient programs for such systems requires a formal specification of memory semantics, called a *memory consistency model*. The most intuitive model—*sequential consistency*—greatly restricts the use of many performance optimizations commonly used by uniprocessor hardware and compiler designers, thereby reducing the benefit of using a multiprocessor. To alleviate this problem, many current multiprocessors support more *relaxed consistency models*. Unfortunately, the models supported by various systems differ from each other in subtle yet important ways. Furthermore, precisely defining the semantics of each model often leads to complex specifications that are difficult to understand for typical users and builders of computer systems.

The purpose of this tutorial paper is to describe issues related to memory consistency models in a way that would be understandable to most computer professionals. We focus on consistency models proposed for hardware-based shared-memory systems. Many of these models are originally specified with an emphasis on the

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a processor does

guarantee the
conditions are
executes mult

Index Terms—
correctness,

A high-sp
order than
execution i
condition:
tions had
processor
sider a co
common
proving
such a co
the resul
the proc

Shared Memory Consistency Models: A Tutorial *

Sarita V. Adve† and Kourosh Gharachorloo‡

†Department of Electrical and Computer Engineering
Rice
892

ory
ion
616

t 9512
port 95/7

accepted in many areas of
specification of memory
ial consistency—greatly
hardware and compiler
roblem, many current
supported by various
ing the semantics of
sers and builders of

models in a way
dels proposed for
emphasis on the

The Java Memory Model

Jeremy Manson and William Pugh
Department of Computer Science
University of Maryland, College Park
College Park, MD
{jmanson, pugh}@cs.umd.edu

Sarita V. Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana-Champaign, IL
sadve@cs.uiuc.edu

ABSTRACT

This paper describes the new Java memory model, which has been revised as part of Java 5.0. The model specifies the legal behaviors for a multithreaded program; it defines the semantics of multithreaded Java programs and partially determines legal implementations of Java virtual machines and compilers.

The new Java model provides a simple interface for correctly synchronized programs – it guarantees sequential consistency to data-race-free programs. Its novel contribution is requiring that the behavior of incorrectly synchronized programs be bounded by a well defined notion of causality. The causality requirement is strong enough to respect the safety and security properties of Java and weak enough to allow standard compiler and hardware optimizations. To our knowledge, other models are either too weak because they do not provide for sufficient safety/security, or are too strong because they rely on a strong notion of data and control dependences that precludes some standard compiler transformations.

Meanings of Programs]: Operational Semantics
General Terms: Design, Languages
Keywords: Concurrency, Java, Memory Model, Multithreading

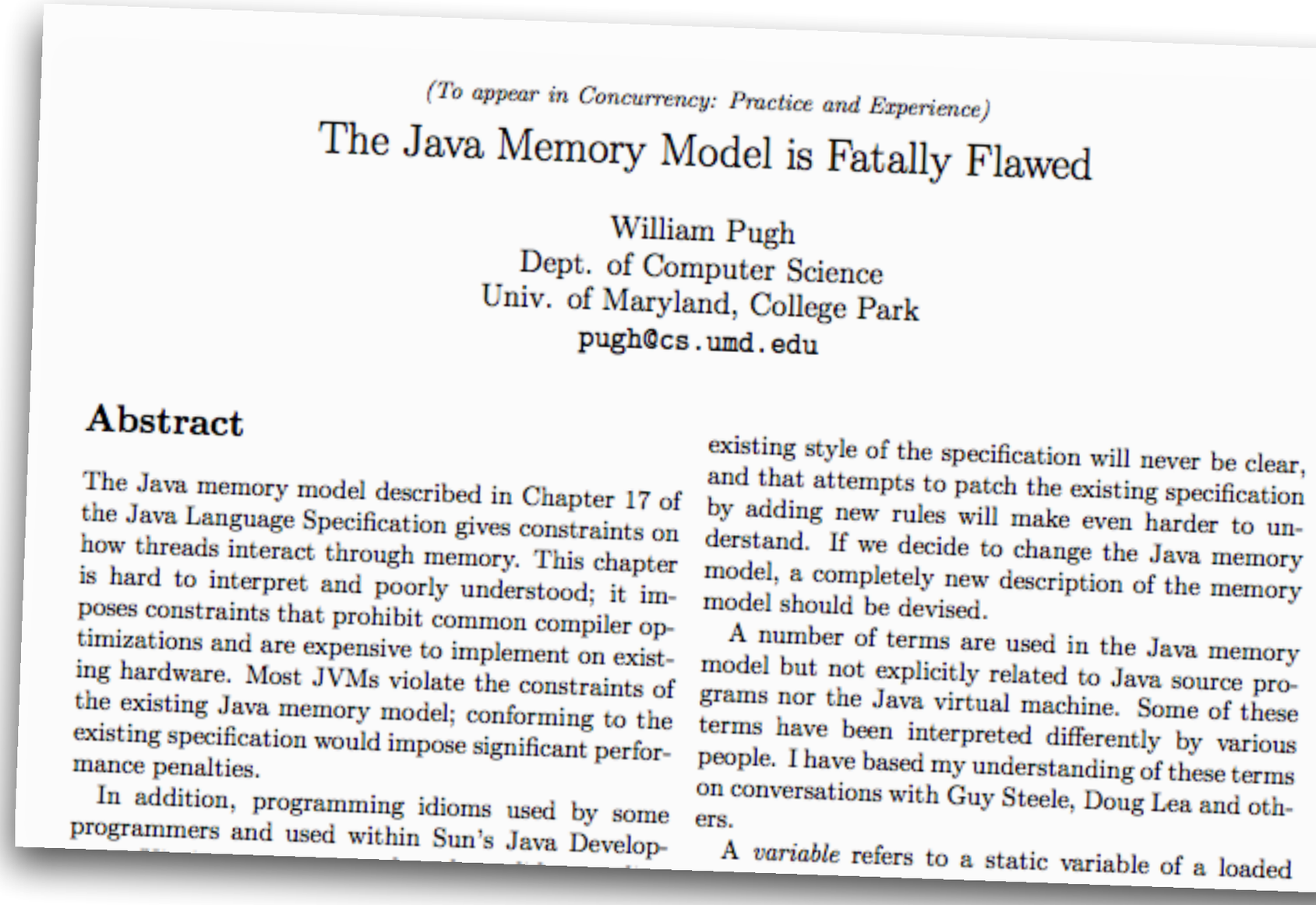
1. INTRODUCTION

The memory model for a multithreaded system specifies how memory actions (e.g., reads and writes) in a program will appear to execute to the programmer, and specifically, which value each read of a memory location may return. Every hardware and software interface of a system that admits multithreaded access to shared memory requires a memory model. The model determines the transformations that the system (compiler, virtual machine, or hardware) can apply to a program written at that interface. For example, given a program in machine language, the memory model for the machine language / hardware interface will determine the optimizations the hardware can perform.

For a high-level programming language such as Java, the memory model determines the transformations the compiler

The case of Java

The case of Java



(To appear in Concurrency: Practice and Experience)

The Java Memory Model is Fatally Flawed

William Pugh
Dept. of Computer Science
Univ. of Maryland, College Park
pugh@cs.umd.edu

Abstract

The Java memory model described in Chapter 17 of the Java Language Specification gives constraints on how threads interact through memory. This chapter is hard to interpret and poorly understood; it imposes constraints that prohibit common compiler optimizations and are expensive to implement on existing hardware. Most JVMs violate the constraints of the existing Java memory model; conforming to the existing specification would impose significant performance penalties.

In addition, programming idioms used by some programmers and used within Sun's Java Develop-

existing style of the specification will never be clear, and that attempts to patch the existing specification by adding new rules will make even harder to understand. If we decide to change the Java memory model, a completely new description of the memory model should be devised.

A number of terms are used in the Java memory model but not explicitly related to Java source programs nor the Java virtual machine. Some of these terms have been interpreted differently by various people. I have based my understanding of these terms on conversations with Guy Steele, Doug Lea and others.

A *variable* refers to a static variable of a loaded

The case of Java

(To appear in Concurrency: Practice and Experience)

The Java Memory Model is Fatally Flawed

Pugh
uter Science
, College Park
nd.edu

isting style of the specification will never be clear,
d that attempts to patch the existing specification
adding new rules will make even harder to un-
stand. If we decide to change the Java memory
del, a completely new description of the memory
del should be devised.
number of terms are used in the Java memory
del but not explicitly related to Java source pro-
ns nor the Java virtual machine. Some of these
s have been interpreted differently by various
le. I have based my understanding of these terms
nversations with Guy Steele, Doug Lea and oth-

variable refers to a static variable of a loaded

The Java Memory Model*

Jeremy Manson and William Pugh
Department of Computer Science
University of Maryland, College Park
College Park, MD
{jmanson, pugh}@cs.umd.edu

Sarita V. Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana-Champaign, IL
sadve@cs.uiuc.edu

ABSTRACT

This paper describes the new Java memory model, which has been revised as part of Java 5.0. The model specifies the legal behaviors for a multithreaded program; it defines the semantics of multithreaded Java programs and partially determines legal implementations of Java virtual machines and compilers.

The new Java model provides a simple interface for correctly synchronized programs – it guarantees sequential consistency to data-race-free programs. Its novel contribution is requiring that the behavior of incorrectly synchronized programs be bounded by a well defined notion of causality. The causality requirement is strong enough to respect the safety and security properties of Java and weak enough to allow standard compiler and hardware optimizations. To our knowledge, other models are either too weak because they do not provide for sufficient safety/security, or are too

Meanings of Programs]: Operational Semantics

General Terms: Design, Languages

Keywords: Concurrency, Java, Memory Model, Multithreading

1. INTRODUCTION

The memory model for a multithreaded system specifies how memory actions (e.g., reads and writes) in a program will appear to execute to the programmer, and specifically, which value each read of a memory location may return. Every hardware and software interface of a system that admits multithreaded access to shared memory requires a memory model. The model determines the transformations that the system (compiler, virtual machine, or hardware) can apply to a program written at that interface. For example, given a program in machine language, the memory model for the machine language / hardware interface will determine the

The case of Java

(To appear in Concurrency: Practice and Experience)

The Java Memory Model is Fatally Flawed

Pugh
uter Science
, College Park
nd.edu

The Java Memory Model

Jeremy Manson and William Pugh
Department of Computer Science
University of Maryland, College Park
College Park, MD
{jmanson, pugh}@cs.umd.edu

Dep
Universi

ABSTRACT

This paper describes the new Java memory model, which has been revised as part of Java 5.0. The model specifies the legal behaviors for a multithreaded program; it defines the semantics of multithreaded Java programs and partially determines legal implementations of Java virtual machines and compilers.

The new Java model provides a simple interface for correctly synchronized programs – it guarantees sequential consistency to data-race-free programs. Its novel contribution is requiring that the behavior of incorrectly synchronized programs be bounded by a well defined notion of causality. The causality requirement is strong enough to respect the safety and security properties of Java and weak enough to allow standard compiler and hardware optimizations. To our knowledge, other models are either too weak because they do not provide for sufficient safety/security, or are too

Meanin
Genera
Keywo
ing

1. I

The
how m
will ap
which
ery ha
multi
mode
syste
to a
a pr
mac

Java Memory Model Examples:
Good, Bad and Ugly

David Aspinall and Jaroslav Ševčík

August 8, 2007

Abstract

We review a number of illustrative example programs for the Java Memory Model (JMM) [6, 3], relating them to the original design goals and giving intuitive explanations (which can be made precise). We consider good, bad and ugly examples. The good examples are allowed behaviours in the JMM, showing possibilities for non sequentially consistent executions and reordering optimisations. The bad examples are prohibited behaviours, which are clearly ruled out by the JMM. The ugly examples are most interesting: these are tricky cases which illustrate some problem areas for the current formulation of the memory model, where the anticipated design goals are not met. For some of these we mention possible fixes, drawing on knowledge we gained while formalising the memory model in the theorem prover Isabelle [1]

The case of Java

(To appear in Concurrency: Practice and Experience)

The Java Memory Model is Fatally Flawed

Pugh
uter Science
, College Park
nd.edu

The Java Memory Model

Jeremy Manson and William Pugh
Department of Computer Science
University of Maryland, College Park
College Park, MD
{jmanson, pugh}@cs.umd.edu

Dep
Universi

ABS
This
has b
the l
the s
deter
and c
The
rectly
sisten
is req
progr
The c
safety
allow
our k
they c

On Validity of Program Transformations in the Java Memory Model

Jaroslav Ševčík and David Aspinall

LFCS, School of Informatics, University of Edinburgh

Abstract. We analyse the validity of several common program transformations in multi-threaded Java, as defined by the Java Memory Model (JMM) section of Chapter 17 of the Java Language Specification. The main design goal of the JMM was to allow as many optimisations as possible. However, we find that commonly used optimisations, such as common subexpression elimination, can introduce new behaviours and

Java Memory Model Examples:
d, Bad and Ugly

pinall and Jaroslav Ševčík

August 8, 2007

Abstract

illustrative example programs for the Java
, relating them to the original design goals
ions (which can be made precise). We con-
mples. The good examples are allowed be-
possibilities for non sequentially consistent
nisations. The bad examples are prohibited
uled out by the JMM. The ugly examples
tricky cases which illustrate some prob-
ulation of the memory model, where the
met. For some of these we mention possi-
we gained while formalising the memory

It’s not just the processor …

It’s not just the processor …

Atomic Operations

```
int x = 1;
x++;        double x = 1.0;

monitorenter
monitorexit
```

It’s not just the processor …

Atomic Operations

```
int x = 1;
x++;        double x = 1.0

monitorenter
monitorexit
```

Synchronizing Operations

```
volatile int x = 0;

thread.start();
thread.join();

monitorenter
monitorexit
```

It’s not just the processor …

Atomic Operations

```
int x = 1;
x++;        double x = 1.0

monitorenter
monitorexit
```

Synchronizing Operations

```
volatile int x = 0;

thread.start();
thread.join();

monitorenter
monitorexit
```

Compiler Optimizations

Reordering

Eliminations (CSE)

Introductions

It’s not just the processor …

Atomic Operations

```
int x = 1;
x++;        double x = 1.0

monitorenter
monitorexit
```

Synchronizing Operations

```
volatile int x = 0;
                thread.start();
                thread.join();
monitorenter
monitorexit
```

Compiler Optimizations

Reordering

Eliminations (CSE)

Introductions

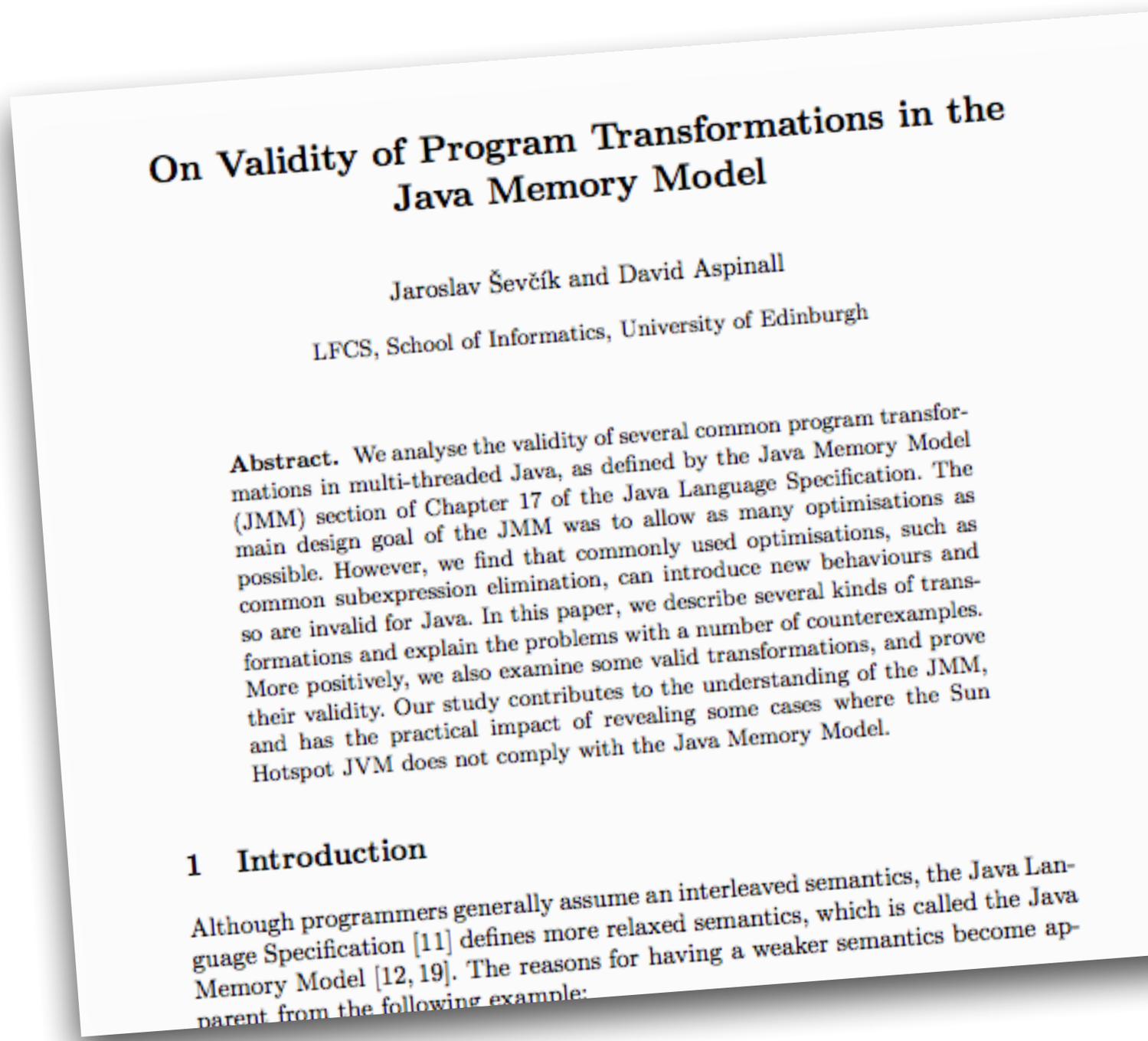
VM Optimizations

Just in time compilation

Lock elision

Biased Locking

Compiler Optimizations



On Validity of Program Transformations in the Java Memory Model

Jaroslav Ševčík and David Aspinall

LFCS, School of Informatics, University of Edinburgh

Abstract. We analyse the validity of several common program transformations in multi-threaded Java, as defined by the Java Memory Model (JMM) section of Chapter 17 of the Java Language Specification. The main design goal of the JMM was to allow as many optimisations as possible. However, we find that commonly used optimisations, such as common subexpression elimination, can introduce new behaviours and so are invalid for Java. In this paper, we describe several kinds of transformations and explain the problems with a number of counterexamples. More positively, we also examine some valid transformations, and prove their validity. Our study contributes to the understanding of the JMM, and has the practical impact of revealing some cases where the Sun Hotspot JVM does not comply with the Java Memory Model.

1 Introduction

Although programmers generally assume an interleaved semantics, the Java Language Specification [11] defines more relaxed semantics, which is called the Java Memory Model [12, 19]. The reasons for having a weaker semantics become ap-

parent from the following example:

Compiler Optimizations

Compiler Optimizations

```
                    x = y = z = 0
```

```
r1 = z;                          ||   r2 = x;
if (r1 == 1) {                   ||   r3 = y;
   x = 1;                        ||   if (r2==1 && r3==1) {
   y = 1;                        ||      z = 1;
} else {                         ||   }
   y = 1;                        ||
   x = 1;                        ||
}                                ||
```

```
                    r1 = r2 = r3 = 1
```

Compiler Optimizations

```
                    x = y = z = 0
```

```
r1 = z;                          ||   r2 = x;
if (r1 == 1) {                   ||   r3 = y;
   x = 1;                        ||   if (r2==1 && r3==1) {
   y = 1;                        ||      z = 1;
} else {                         ||   }
   x = 1;                        ||
   y = 1;                        ||
}                                ||
```

```
                    r1 = r2 = r3 = 1
```

Compiler Optimizations

```
                    x = y = z = 0
```

```
r1 = z;          ||   r2 = x;
x = 1;           ||   r3 = y;
y = 1;           ||   if (r2==1 && r3==1) {
                 ||      z = 1;
                 ||   }
```

```
                    r1 = r2 = r3 = 1
```

Compiler Optimizations

```
                    x = y = z = 0
```

```
x = 1;          ||    r2 = x;
y = 1;          ||    r3 = y;
r1 = z;         ||    if (r2==1 && r3==1) {
                ||       z = 1;
                ||    }
```

```
                    r1 = r2 = r3 = 1
```

Compiler Optimizations

x = y = z = 0

```
x = 1;          ||   r2 = x;
y = 1;          ||   r3 = y;
r1 = z;         ||   if (r2==1 && r3==1) {
                ||      z = 1;
                ||   }
```

r1 = r2 = r3 = 1 ✔

Compiler Optimizations

x = y = z = 0	
`x = 1;`	`r2 = x;`
`y = 1;`	`r3 = y;`
`r1 = z;`	`if (r2==1 && r3==1) {`
	`  z = 1;`
	`}`
`r1 = r2 = r3 = 1`	

Do we allow the behavior in the JMM?

Compiler Optimizations

`x = y = z = 0`

```
x = 1;          ||   r2 = x;
y = 1;          ||   r3 = y;
r1 = z;         ||   if (r2==1 && r3==1) {
                ||     z = 1;
                ||   }
```

`r1 = r2 = r3 = 1` ✔

Do we allow the behavior in the JMM?

Do we forbid these simple optimizations?

Thin-Air-Reads

$$\mathtt{x} = 0 \quad \& \quad \mathtt{y} = 0$$

$\mathtt{r}_1 = \mathtt{y};$ $\mathtt{if}(\mathtt{r}_1 != 0)$ $\mathtt{x} = \mathtt{r}_1;$	$\parallel$	$\mathtt{r}_2 = \mathtt{x};$ $\mathtt{if}(\mathtt{r}_2 != 0)$ $\mathtt{x} = \mathtt{r}_2;$

$$\mathtt{r}_1 = \mathtt{r}_3 = 42?$$

Thin-Air-Reads

$$\mathtt{x} = 0 \quad \& \quad \mathtt{y} = 0$$

$\mathtt{r}_1 = \mathtt{y};$ $\mathtt{if}(\mathtt{r}_1 != 0)$ $\mathtt{x} = \mathtt{r}_1;$	$\parallel$	$\mathtt{r}_2 = \mathtt{x};$ $\mathtt{if}(\mathtt{r}_2 != 0)$ $\mathtt{x} = \mathtt{r}_2;$

$$\mathtt{r}_1 = \mathtt{r}_3 = 42?$$

If we consider SC executions this program has **no data races**

Thin-Air-Reads

$$\mathtt{x} = 0 \quad \& \quad \mathtt{y} = 0$$

$\mathtt{r}_1 = \mathtt{y};$ $\mathtt{if}(\mathtt{r}_1 != 0)$ $\mathtt{x} = \mathtt{r}_1;$	$\parallel$	$\mathtt{r}_2 = \mathtt{x};$ $\mathtt{if}(\mathtt{r}_2 != 0)$ $\mathtt{x} = \mathtt{r}_2;$

$$\mathtt{r}_1 = \mathtt{r}_3 = 42?$$

If we consider SC executions this program has **no data races**

The value 42 appears nowhere in the program

Thin-Air-Reads

$$\mathtt{x} = 0 \quad \& \quad \mathtt{y} = 0$$

$\mathtt{r}_1 = \mathtt{y};$	$\Vert$	$\mathtt{r}_2 = \mathtt{x};$
$\mathtt{if}(\mathtt{r}_1 != 0)$		$\mathtt{if}(\mathtt{r}_2 != 0)$
$\mathtt{x} = \mathtt{r}_1;$		$\mathtt{x} = \mathtt{r}_2;$

$$\mathtt{r}_1 = \mathtt{r}_3 = 42?$$

If we consider SC executions this program has **no data races**

The value 42 appears nowhere in the program

Explicitly **forbidden** by the JMM and C++11

Thin-Air-Reads

$$\mathtt{x} = 0 \quad \& \quad \mathtt{y} = 0$$

$\mathtt{r}_1 = \mathtt{y};$ $\mathtt{x} = (\mathtt{r}_1 == 1)\ ?\ \mathtt{r}_1 : 1;$	$\parallel$	$\mathtt{r}_2 = \mathtt{x};$ $\mathtt{y} = (\mathtt{r}_2 == 1)?\mathtt{r}_2 : 1;$

$$\mathtt{r}_1 = \mathtt{r}_2 = 1?$$

Thin-Air-Reads

$$\mathtt{x} = 0 \quad \& \quad \mathtt{y} = 0$$

$\mathtt{r}_1 = \mathtt{y};$	$\Vert$	$\mathtt{r}_2 = \mathtt{x};$
$\mathtt{x} = (\mathtt{r}_1 == 1)\ ?\ \mathtt{r}_1 : 1;$	$\Vert$	$\mathtt{y} = (\mathtt{r}_2 == 1)?\mathtt{r}_2 : 1;$

$$\mathtt{r}_1 = \mathtt{r}_2 = 1?$$

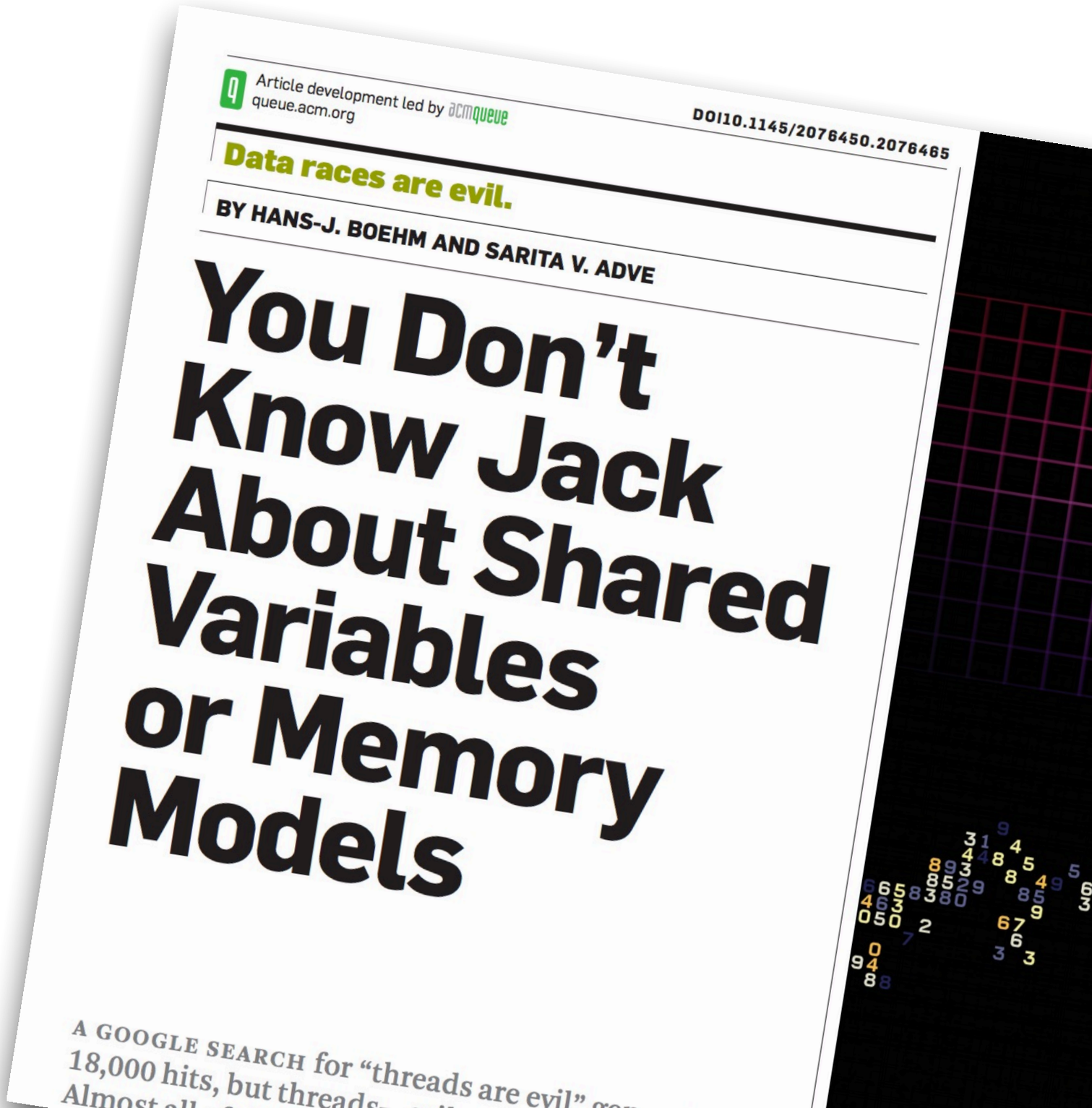
$$\approx$$

$$\mathtt{x} = 0 \quad \& \quad \mathtt{y} = 0$$

$\mathtt{r}_1 = \mathtt{y};$	$\Vert$	$\mathtt{r}_2 = \mathtt{x};$
$\mathtt{x} = 1;$	$\Vert$	$\mathtt{y} = 1;$

$$\mathtt{r}_1 = \mathtt{r}_2 = 1?$$

The fundamental Property of Memory Models



Article development led by acmqueue
queue.acm.org

DOI10.1145/2076450.2076465

Data races are evil.

BY HANS-J. BOEHM AND SARITA V. ADVE

You Don't Know Jack About Shared Variables or Memory Models

A GOOGLE SEARCH for "threads are evil" gives
18,000 hits, but threads

The fundamental Property of Memory Models

The fundamental Property of Memory Models

- Data Race
  - concurrent memory accesses on a single variable
  - at least one is a write

$$x = 1 \parallel x = 2 \qquad\qquad x = 1 \parallel r = x$$

The fundamental Property of Memory Models

- Data Race
  - concurrent memory accesses on a single variable
  - at least one is a write

$$x = 1 \parallel x = 2 \qquad\qquad x = 1 \parallel r = x$$

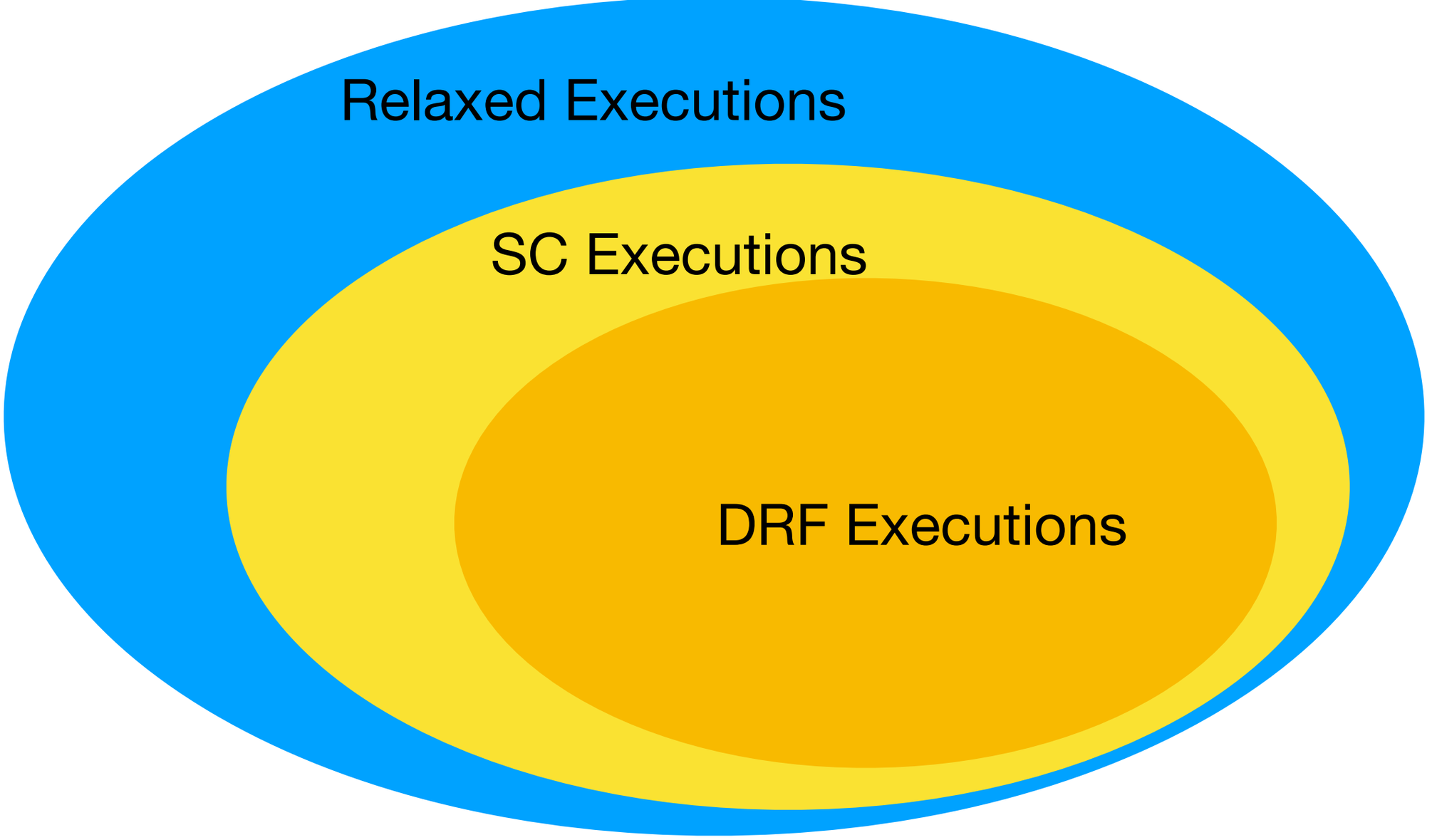
- Data Race Freedom
  - Every SC execution is free of data races

The fundamental Property of Memory Models

- Data Race
  - concurrent memory accesses on a single variable
  - at least one is a write

$$x = 1 \parallel x = 2 \qquad\qquad x = 1 \parallel r = x$$

- Data Race Freedom
  - Every SC execution is free of data races
- DRF Guarantee
  - DRF programs do not exhibit relaxed behaviors


Relaxed Executions
SC Executions
DRF Executions

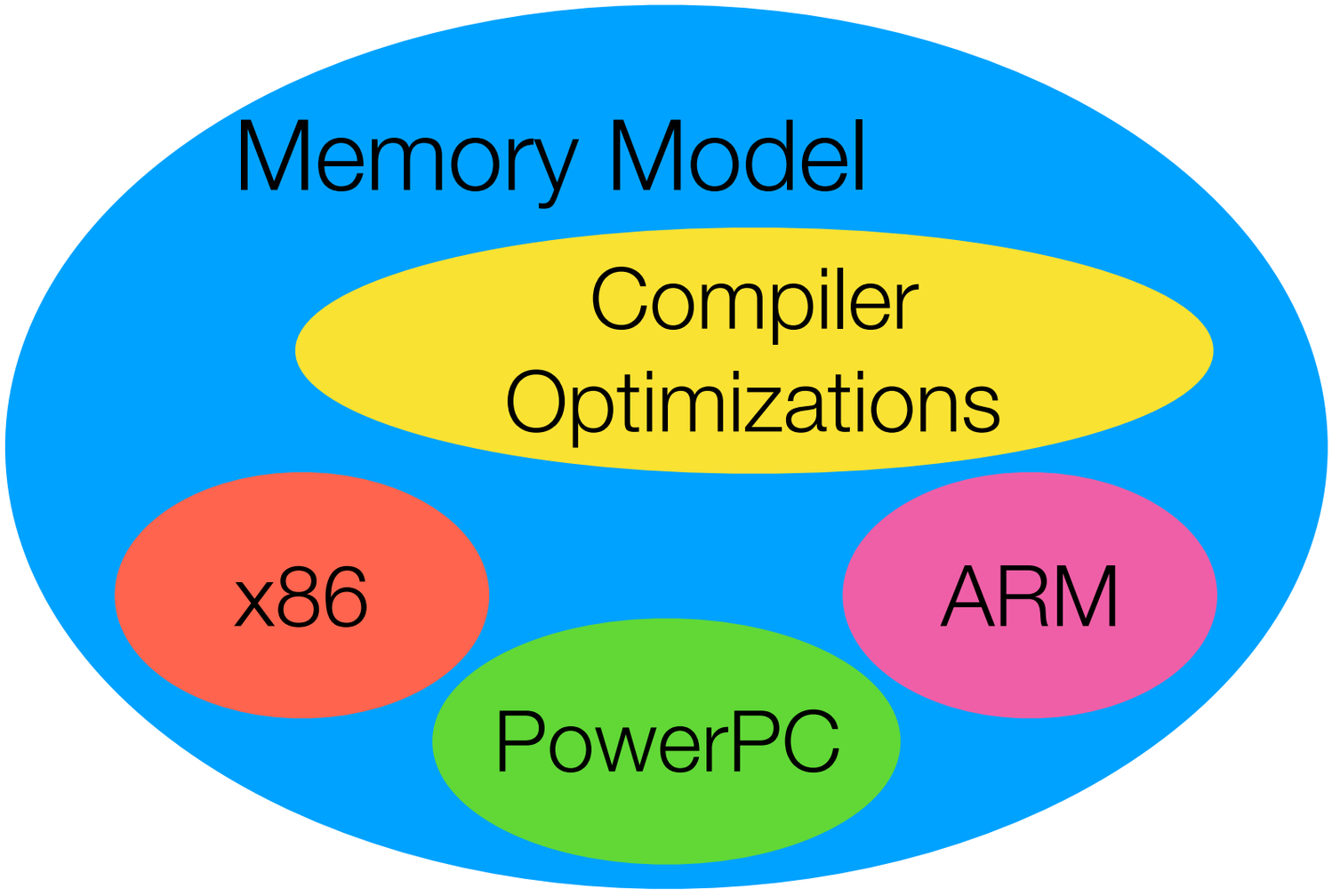

Relaxed Executions
SC Executions
DRF Executions

- Remark: To check if a program is DRF we can use SC tools

- Remark: To check if a program is DRF we can use SC tools
- Corollary: If a program is DRF we can use any SC too

- Remark: To check if a program is DRF we can use SC tools
- Corollary: If a program is DRF we can use any SC too
- Verification Strategy:
    1. Check that the program is DRF
    2. Verify the program using any SC tool

Memory Models in Practice



What if we define DRF to be the memory model?

- Someone has to implement the JVM
- Someone has to implement java.util.concurrent
- There are no complete DRF detection algorithms
- Performance critical lock-free data structures
- What happens if despite the programmers effort there are data races?

C++ Memory Model

C++ Memory Model

- Catch-fire semantics for data races

C++ Memory Model

- Catch-fire semantics for data races
- Variables declared atomic cannot race
  `std::atomic<int> x;`
  - Memory Ordering Decorations:
    `x.load(mo), x.store(v, mo)`
    `x.store(5, memory_order_release)`
    `x.load(memory_order_seq_cst)`
  - Release/Acquire: Message Passing
  - Release/Consume: Message Passing modulo dependencies
  - Relaxed: simply not a data race

C++ Memory Model

```
bool t1;
bool t2;
int counter = 1;

t1 = true;          ||  t2 = true;
if (!t2)            ||  if (!t1)
  counter--;        ||    counter--;

      assert(counter >= 0);
```

C++ Memory Model

```
bool t1;
bool t2;
int counter = 1;
```

`t1 = true;`	`t2 = true;`
`if (!t2)`	`if (!t1)`
`  counter--;`	`  counter--;`

```
assert(counter >= 0);
```

C++ Memory Model

```
atomic<bool> t1;
atomic<bool> t2;
int counter = 1;
```

```
t1.store(true, relaxed);
if (!t2.load(relaxed))
  counter--;
```

```
t2.store(true, relaxed);
if (!t1.load(relaxed))
  counter--;
```

```
assert(counter >= 0);
```

C++ Memory Model

```
atomic<bool> t1;
atomic<bool> t2;
int counter = 1;
```

```
t1.store(true, relaxed);
if (!t2.load(relaxed))
  counter--;
```

```
t2.store(true, relaxed);
if (!t1.load(relaxed))
  counter--;
```

```
assert(counter >= 0);
```

C++ Memory Model

```
atomic<bool> t1;
atomic<bool> t2;
int counter = 1;
```

```
t1.store(true, release);  ||  t2.store(true, release);
if (!t2.load(acquire))    ||  if (!t1.load(acquire))
  counter--;              ||    counter--;
```

```
assert(counter >= 0);
```

C++ Memory Model

```
atomic<bool> t1;
atomic<bool> t2;
int counter = 1;
```

```
t1.store(true, release);
if (!t2.load(acquire))
  counter--;
```

```
t2.store(true, release);
if (!t1.load(acquire))
  counter--;
```

```
assert(counter >= 0);
```

C++ Memory Model

```
atomic<bool> t1;
atomic<bool> t2;
int counter = 1;
```

`t1.store(true, seq_cst);`	`t2.store(true, seq_cst);`
`if (!t2.load(seq_cst))`	`if (!t1.load(seq_cst))`
`  counter--;`	`  counter--;`

```
assert(counter >= 0);
```

C++ Memory Model

```
atomic<bool> t1;
atomic<bool> t2;
int counter = 1;
```

```
t1.store(true, seq_cst);
if (!t2.load(seq_cst))
  counter--;
```

```
t2.store(true, seq_cst);
if (!t1.load(seq_cst))
  counter--;
```

```
assert(counter >= 0);
```

C++11 demo

CPPMem

The Java Memory Model (JMM)

The JMM by Intimidation

$$x = 0$$

$$\texttt{lock}(\ell); \; x{+}{+}; \; \texttt{unlock}(\ell); \quad \| \quad \texttt{lock}(\ell); \; x{+}{+}; \; \texttt{unlock}(\ell); \quad \| \quad \texttt{lock}(\ell); \; x{+}{+}; \; \texttt{unlock}(\ell); \quad \| \quad \texttt{while}(1)\ \mathrm{print}(x);$$

0 0 0 1 1 2 2 2 2 3 3 ...

The JMM by Intimidation

$$x = 0$$

$$\begin{array}{l}\texttt{lock}(\ell);\\ \quad x{+}{+};\\ \texttt{unlock}(\ell);\end{array} \quad \| \quad \begin{array}{l}\texttt{lock}(\ell);\\ \quad x{+}{+};\\ \texttt{unlock}(\ell);\end{array} \quad \| \quad \begin{array}{l}\texttt{lock}(\ell);\\ \quad x{+}{+};\\ \texttt{unlock}(\ell);\end{array} \quad \| \quad \texttt{while}(1)\ \mathrm{print}(x);$$

$$0\ 0\ 0\ 1\ 1\ 2\ 2\ 2\ 2\ 3\ 3\ \ldots$$

$$\approx$$

$$\begin{array}{l}\texttt{lock}(\ell);\\ \quad x{+}{+};\\ \texttt{unlock}(\ell);\end{array} \quad \| \quad \begin{array}{l}\texttt{lock}(\ell);\\ \quad x{+}{+};\\ \texttt{unlock}(\ell);\end{array} \quad \| \quad \begin{array}{l}\texttt{lock}(\ell);\\ \quad x{+}{+};\\ \texttt{unlock}(\ell);\end{array} \quad \| \quad \begin{array}{l}\texttt{while}(1)\\ \quad \mathrm{print}(\mathrm{random}(0,3));\end{array}$$

$$0\ 3\ 1\ 3\ 2\ 0\ 0\ 1\ 2\ 0\ 0\ \ldots$$

The JMM by Intimidation

$$x = 0$$

```
lock(ℓ);          lock(ℓ);          lock(ℓ);
  x++;      ‖      x++;      ‖      x++;      ‖  while(1) print(x);
unlock(ℓ);        unlock(ℓ);        unlock(ℓ);
```

$$0\ 0\ 0\ 1\ 1\ 2\ 2\ 2\ 2\ 3\ 3\ \ldots$$

$$\approx$$

```
lock(ℓ);          lock(ℓ);          lock(ℓ);
  x++;      ‖      x++;      ‖      x++;      ‖  while(1)
unlock(ℓ);        unlock(ℓ);        unlock(ℓ);      print(random(0,3));
```

$$0\ 3\ 1\ 3\ 2\ 0\ 0\ 1\ 2\ 0\ 0\ \ldots$$

Of course, no reasonable implementation would do that!

JMM in a Nutshell

- happens-before is enforced through locks or volatile variables
- reads can see any non-hb related write, or
- any immediate predecessor in hb

x = y = 0	
1: lock m	5: lock m
2: r1 = x	6: r2 = y
3: y = r1	7: x = r2
4: unlock m	8: unlock m

JMM in a Nutshell

- happens-before is enforced through locks or volatile variables
- reads can see any non-hb related write, or
- any immediate predecessor in hb

x = y = 0	
1: lock m	5: lock m
2: r1 = x	6: r2 = y
3: y = r1	7: x = r2
4: unlock m	8: unlock m
po: $1 \overset{po}{\rightarrow} 2 \overset{po}{\rightarrow} 3 \overset{po}{\rightarrow} 4; 5 \overset{po}{\rightarrow} 6 \overset{po}{\rightarrow} 7 \overset{po}{\rightarrow} 8$	

JMM in a Nutshell

- happens-before is enforced through locks or volatile variables
- reads can see any non-hb related write, or
- any immediate predecessor in hb

x = y = 0	
1: lock m 2: r1 = x 3: y = r1 4: unlock m	5: lock m 6: r2 = y 7: x = r2 8: unlock m
po: $1 \xrightarrow{po} 2 \xrightarrow{po} 3 \xrightarrow{po} 4; 5 \xrightarrow{po} 6 \xrightarrow{po} 7 \xrightarrow{po} 8$	
so: $1 \xrightarrow{so} 4 \xrightarrow{so} 5 \xrightarrow{so} 8 \vee 5 \xrightarrow{so} 8 \xrightarrow{so} 1 \xrightarrow{so} 4$	

JMM in a Nutshell

- happens-before is enforced through locks or volatile variables
- reads can see any non-hb related write, or
- any immediate predecessor in hb

x = y = 0	
1: lock m 2: r1 = x 3: y = r1 4: unlock m	5: lock m 6: r2 = y 7: x = r2 8: unlock m
po: $1 \xrightarrow{po} 2 \xrightarrow{po} 3 \xrightarrow{po} 4; 5 \xrightarrow{po} 6 \xrightarrow{po} 7 \xrightarrow{po} 8$	
so: $1 \xrightarrow{so} 4 \xrightarrow{so} 5 \xrightarrow{so} 8 \vee 5 \xrightarrow{so} 8 \xrightarrow{so} 1 \xrightarrow{so} 4$	
sw: $4 \xrightarrow{sw} 5 \vee 8 \xrightarrow{sw} 1$	

JMM in a Nutshell

- happens-before is enforced through locks or volatile variables
- reads can see any non-hb related write, or
- any immediate predecessor in hb

x = y = 0	
1: lock m 2: r1 = x 3: y = r1 4: unlock m	5: lock m 6: r2 = y 7: x = r2 8: unlock m
po: $1 \xrightarrow{po} 2 \xrightarrow{po} 3 \xrightarrow{po} 4; 5 \xrightarrow{po} 6 \xrightarrow{po} 7 \xrightarrow{po} 8$	
so: $1 \xrightarrow{so} 4 \xrightarrow{so} 5 \xrightarrow{so} 8 \vee 5 \xrightarrow{so} 8 \xrightarrow{so} 1 \xrightarrow{so} 4$	
sw: $4 \xrightarrow{sw} 5 \vee 8 \xrightarrow{sw} 1$	
hb: $1 \xrightarrow{hb} 2 \xrightarrow{hb} 3 \xrightarrow{hb} 4 \xrightarrow{hb} 5 \xrightarrow{hb} 6 \xrightarrow{hb} 7 \xrightarrow{hb} 8 \vee$ hb: $5 \xrightarrow{hb} 6 \xrightarrow{hb} 7 \xrightarrow{hb} 8 \xrightarrow{hb} 1 \xrightarrow{hb} 2 \xrightarrow{hb} 3 \xrightarrow{hb} 4$	

JMM in a Nutshell

- happens-before is enforced through locks or volatile variables
- reads can see any non-hb related write, or
- any immediate predecessor in hb

x = y = 0	
1: lock m 2: r1 = x 3: y = r1 4: unlock m	5: lock m 6: r2 = y 7: x = r2 8: unlock m
po: $1 \xrightarrow{po} 2 \xrightarrow{po} 3 \xrightarrow{po} 4; 5 \xrightarrow{po} 6 \xrightarrow{po} 7 \xrightarrow{po} 8$	
so: $1 \xrightarrow{so} 4 \xrightarrow{so} 5 \xrightarrow{so} 8 \vee 5 \xrightarrow{so} 8 \xrightarrow{so} 1 \xrightarrow{so} 4$	
sw: $4 \xrightarrow{sw} 5 \vee 8 \xrightarrow{sw} 1$	
hb: $1 \xrightarrow{hb} 2 \xrightarrow{hb} 3 \xrightarrow{hb} 4 \xrightarrow{hb} 5 \xrightarrow{hb} 6 \xrightarrow{hb} 7 \xrightarrow{hb} 8 \vee$ hb: $5 \xrightarrow{hb} 6 \xrightarrow{hb} 7 \xrightarrow{hb} 8 \xrightarrow{hb} 1 \xrightarrow{hb} 2 \xrightarrow{hb} 3 \xrightarrow{hb} 4$	

Theorem:

The JMM respects the DRF guarantee

JMM in a Nutshell

- happens-before is enforced through locks or volatile variables
- reads can see any non-hb related write, or
- any immediate predecessor in hb

x = y = 0	
1: lock m 2: r1 = x 3: y = r1 4: unlock m	5: lock m 6: r2 = y 7: x = r2 8: unlock m
po: $1 \xrightarrow{po} 2 \xrightarrow{po} 3 \xrightarrow{po} 4; 5 \xrightarrow{po} 6 \xrightarrow{po} 7 \xrightarrow{po} 8$	
so: $1 \xrightarrow{so} 4 \xrightarrow{so} 5 \xrightarrow{so} 8 \vee 5 \xrightarrow{so} 8 \xrightarrow{so} 1 \xrightarrow{so} 4$	
sw: $4 \xrightarrow{sw} 5 \vee 8 \xrightarrow{sw} 1$	
hb: $1 \xrightarrow{hb} 2 \xrightarrow{hb} 3 \xrightarrow{hb} 4 \xrightarrow{hb} 5 \xrightarrow{hb} 6 \xrightarrow{hb} 7 \xrightarrow{hb} 8 \vee$ hb: $5 \xrightarrow{hb} 6 \xrightarrow{hb} 7 \xrightarrow{hb} 8 \xrightarrow{hb} 1 \xrightarrow{hb} 2 \xrightarrow{hb} 3 \xrightarrow{hb} 4$	

Theorem:

The JMM respects the DRF guarantee

Theorem:

Independent non-synchronizing statements can always be reordered

JMM in a Nutshell

- happens-before is enforced through locks or volatile variables
- reads can see any non-hb related write, or
- any immediate predecessor in hb

x = y = 0	
1: lock m 2: r1 = x 3: y = r1 4: unlock m	5: lock m 6: r2 = y 7: x = r2 8: unlock m
po: $1 \xrightarrow{po} 2 \xrightarrow{po} 3 \xrightarrow{po} 4; 5 \xrightarrow{po} 6 \xrightarrow{po} 7 \xrightarrow{po} 8$	
so: $1 \xrightarrow{so} 4 \xrightarrow{so} 5 \xrightarrow{so} 8 \vee 5 \xrightarrow{so} 8 \xrightarrow{so} 1 \xrightarrow{so} 4$	
sw: $4 \xrightarrow{sw} 5 \vee 8 \xrightarrow{sw} 1$	
hb: $1 \xrightarrow{hb} 2 \xrightarrow{hb} 3 \xrightarrow{hb} 4 \xrightarrow{hb} 5 \xrightarrow{hb} 6 \xrightarrow{hb} 7 \xrightarrow{hb} 8 \vee$ hb: $5 \xrightarrow{hb} 6 \xrightarrow{hb} 7 \xrightarrow{hb} 8 \xrightarrow{hb} 1 \xrightarrow{hb} 2 \xrightarrow{hb} 3 \xrightarrow{hb} 4$	

Theorem:

The JMM respects the DRF guarantee

Theorem:

Independent non-synchronizing statements can always be reordered

✗ Sevcik,Aspinall 2008

JMM Roach Motel Semantics

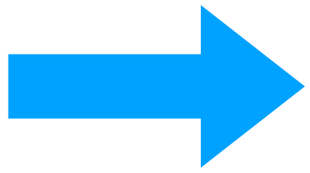
Increasing happens-before is always safe

```
x = 1;
synchronized(this) {
  y = 1;
  r1 = x;
}
r2 = y;
```

JMM Roach Motel Semantics

Increasing happens-before is always safe

```
x = 1;
synchronized(this) {
  y = 1;
  r1 = x;
}
r2 = y;
```

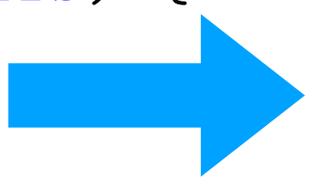


```
synchronized(this) {
  x = 1;
  y = 1;
  r1 = x;
  r2 = y;
}
```

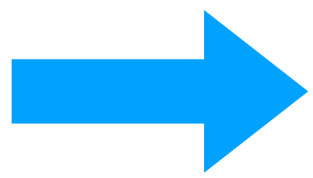
JMM Roach Motel Semantics

Increasing happens-before is always safe

```
x = 1;
synchronized(this) {
  y = 1;
  r1 = x;
}
r2 = y;
```



```
synchronized(this) {
  x = 1;
  y = 1;
  r1 = x;
  r2 = y;
}
```

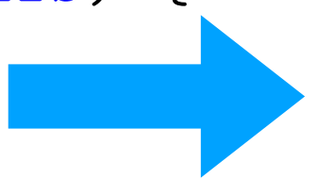


```
synchronized(this) {
  x = 1;
  y = 1;
  r1 = x;
  r2 = y;
}
```

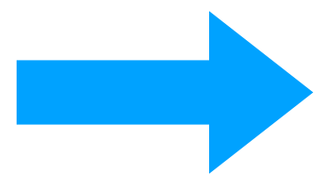
JMM Roach Motel Semantics

Increasing happens-before is always safe

```
x = 1;
synchronized(this) {
  y = 1;
  r1 = x;
}
r2 = y;
```



```
synchronized(this) {
  x = 1;
  y = 1;
  r1 = x;
  r2 = y;
}
```



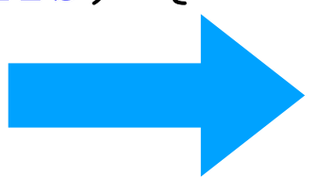
```
synchronized(this) {
  x = 1;
  y = 1;
  r1 = x;
  r2 = y;
}
```

```
v = 1;
x = 1;
```

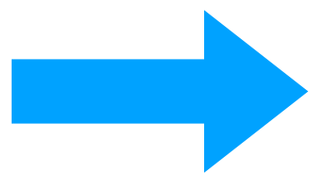
JMM Roach Motel Semantics

Increasing happens-before is always safe

```
x = 1;
synchronized(this) {
  y = 1;
  r1 = x;
}
r2 = y;
```

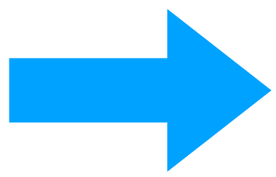


```
synchronized(this) {
  x = 1;
  y = 1;
  r1 = x;
  r2 = y;
}
```



```
synchronized(this) {
  x = 1;
  y = 1;
  r1 = x;
  r2 = y;
}
```

```
v = 1;
x = 1;
```

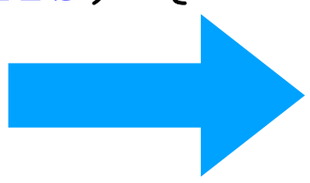


```
x = 1;
v = 1;
```

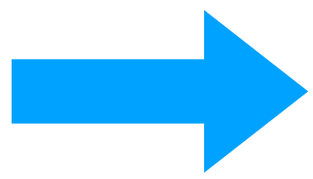
JMM Roach Motel Semantics

Increasing happens-before is always safe

```
x = 1;
synchronized(this) {
  y = 1;
  r1 = x;
}
r2 = y;
```

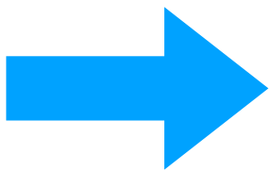


```
synchronized(this) {
  x = 1;
  y = 1;
  r1 = x;
  r2 = y;
}
```



```
synchronized(this) {
  x = 1;
  y = 1;
  r1 = x;
  r2 = y;
}
```

```
v = 1;
x = 1;
```



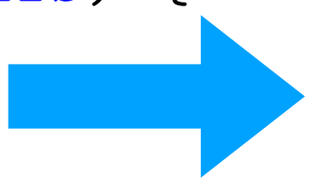
```
x = 1;
v = 1;
```

```
r1 = x;
r2 = v;
```

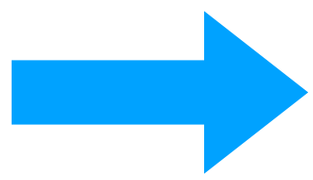
JMM Roach Motel Semantics

Increasing happens-before is always safe

```
x = 1;
synchronized(this) {
  y = 1;
  r1 = x;
}
r2 = y;
```

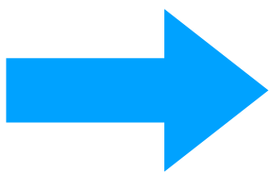


```
synchronized(this) {
  x = 1;
  y = 1;
  r1 = x;
  r2 = y;
}
```



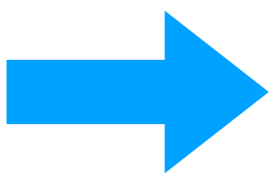
```
synchronized(this) {
  x = 1;
  y = 1;
  r1 = x;
  r2 = y;
}
```

```
v = 1;
x = 1;
```



```
x = 1;
v = 1;
```

```
r1 = x;
r2 = v;
```



```
r2 = v;
r1 = x;
```

Compiler Optimizations

Transformation	SC	JMM	DRF
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	✓	×	✓
External action reordering	×	×	✓

✓– correct, ×– incorrect, ✓* – correct only for adjacent memory accesses.

Validity of Program Transformations in the Java Memory Model
Sevcik & Aspinall'2008

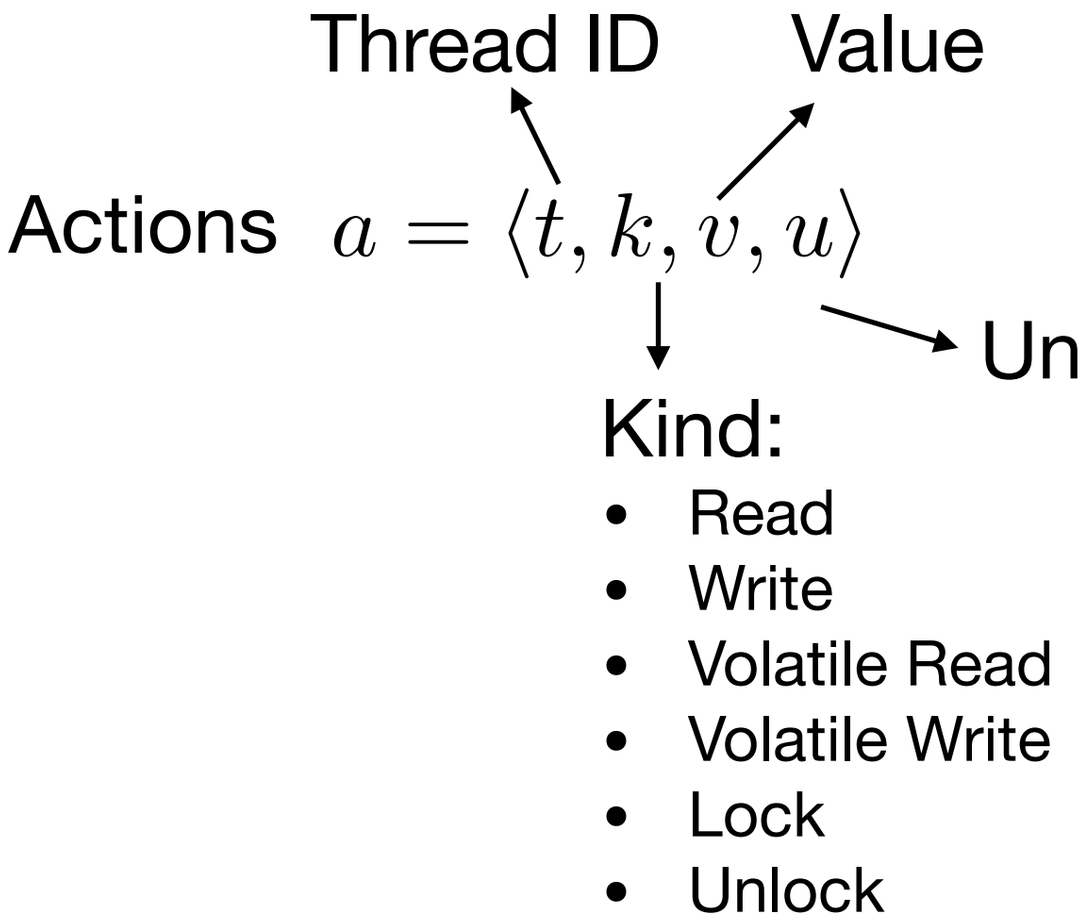
The Formal JMM

Adve, Manson, Pugh'05

Actions $a = \langle t, k, v, u \rangle$

The Formal JMM

Adve, Manson, Pugh'05



The Formal JMM

Adve, Manson, Pugh'05

Actions $a = \langle t, k, v, u \rangle$

Executions $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$

The Formal JMM

Adve, Manson, Pugh'05

Actions $a = \langle t, k, v, u \rangle$

Program

Executions $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$

The Formal JMM

Adve, Manson, Pugh'05

Actions $a = \langle t, k, v, u \rangle$

Program Actions

Executions $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$

The Formal JMM

Adve, Manson, Pugh'05

Actions $a = \langle t, k, v, u \rangle$

Program

Actions

Executions $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$

Program Order

The Formal JMM

Adve, Manson, Pugh'05

Actions $a = \langle t, k, v, u \rangle$

Program Actions

Executions $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$

Program Order

Synchronization Order

Total order over synchronization actions

The Formal JMM

Adve, Manson, Pugh'05

Actions $a = \langle t, k, v, u \rangle$

Program Actions

Executions $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$

Program Order

Synchronization Order

Total order over synchronization actions

Write-Seen function

For each read, returns the write that stored the value read

The Formal JMM

Adve, Manson, Pugh'05

Actions $a = \langle t, k, v, u \rangle$

Program

Actions

Executions $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$

Program Order

Synchronization Order

Total order over synchronization actions

Write-Seen function

For each read, returns the write that stored the value read

Value-written function

The Formal JMM

Adve, Manson, Pugh'05

Actions $a = \langle t, k, v, u \rangle$

Program

Actions

Executions $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$

Program Order

Synchronization Order

Total order over synchronization actions

Synchronized-With

Write-Seen function

For each read, returns the write that stored the value read

Value-written function

The Formal JMM

Adve, Manson, Pugh'05

Actions $a = \langle t, k, v, u \rangle$

Program

Actions

Happens-Before $hb = (po \cup sw)^+$

Executions $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$

Program Order

Synchronization Order

Total order over synchronization actions

Synchronized-With

Write-Seen function

For each read, returns the write that stored the value read

Value-written function

The Formal JMM

Adve, Manson, Pugh'05

Well-Formedness

- for every read r, $V(W(r)) = r.v$
- $\xrightarrow{po}$ is consistent with $\xrightarrow{so}$ ($\xrightarrow{hb}$ is a partial order)
- lock are consistent with mutual exclusion
- the semantics of single threads is respected
- synchronization is consistent
- happens-before is consistent

The Formal JMM

Adve, Manson, Pugh'05

Causality: we need to justify each of the actions of the execution

Commit actions of E step by step

$$(C_0, E_0) \cdot (C_1, E_1) \cdot \cdots \cdot (C_n, E_n)$$

Justifying execution

Set of committed actions

A E

The Formal JMM

Adve, Manson, Pugh'05

1. $C_i \subseteq A_i$
2. $\xrightarrow{hb_i}|_{C_i} = \xrightarrow{hb}|_{C_i}$
3. $\xrightarrow{so_i}|_{C_i} = \xrightarrow{so}|_{C_i}$
4. $V_i|_{C_i} = V|_{C_i}$
5. $\forall r \in Reads(A_i - C_{i-1}) : W_i(r) \xrightarrow{hb_i} r$
6. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$
7. $\forall r \in Reads(C_i - C_{i-1}) : \{W_i(r), W(r)\} \subseteq C_{i-1}$

The Formal JMM

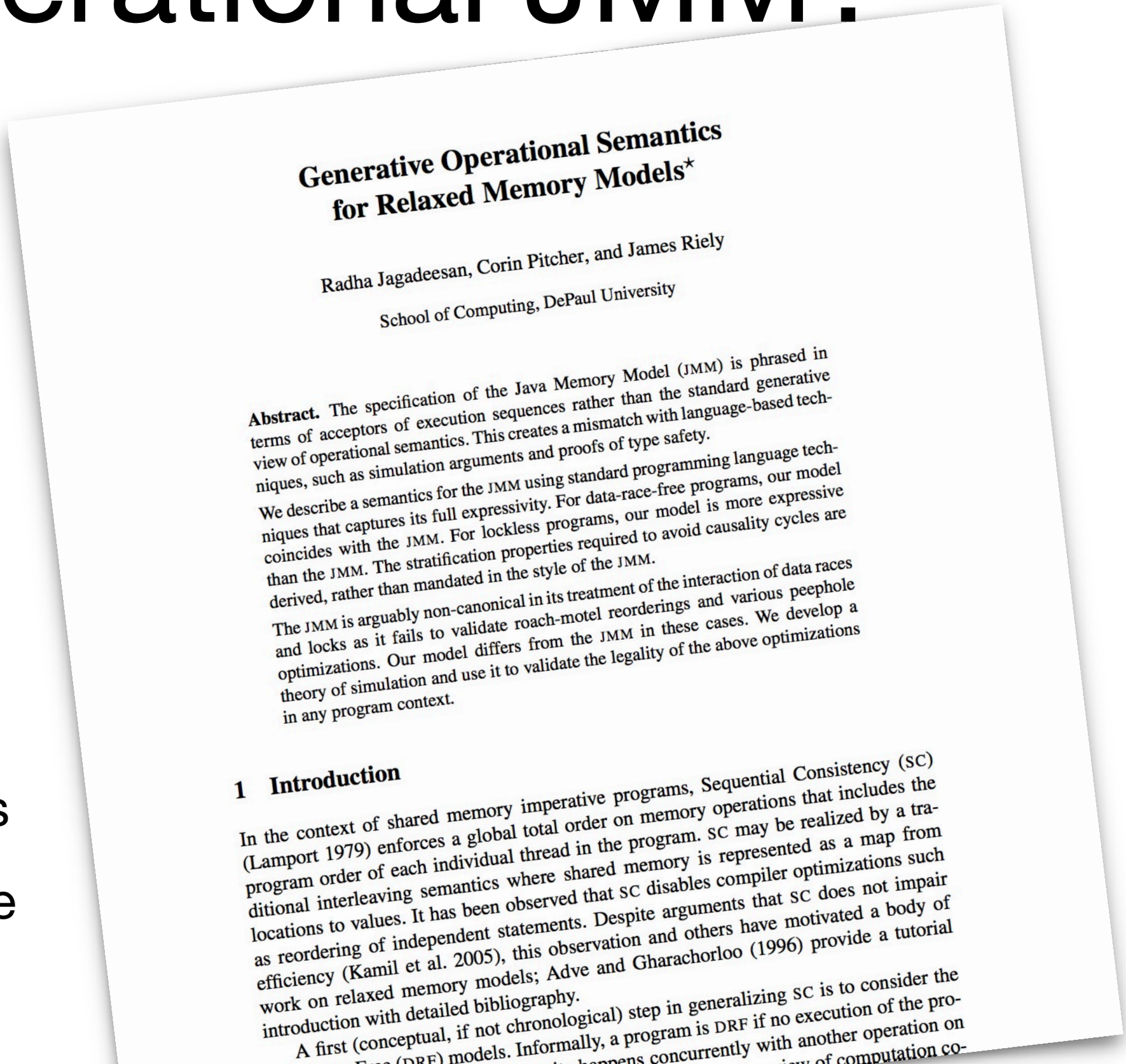
Adve, Manson, Pugh'05

1. $C_i \subseteq A_i$
2. $\xrightarrow{hb_i}|_{C_i} = \xrightarrow{hb}|_{C_i}$
3. $\xrightarrow{so_i}|_{C_i} = \xrightarrow{so}|_{C_i}$
4. $V_i|_{C_i} = V|_{C_i}$
5. $\forall r \in Reads(A_i - C_{i-1}) : W_i(r) \xrightarrow{hb_i} r$
6. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$
7. $\forall r \in Reads(C_i - C_{i-1}) : \{W_i(r), W(r)\} \subseteq C_{i-1}$

Causality Test Cases

Operational JMM?

- Partial solutions
- Remains elusive



Generative Operational Semantics for Relaxed Memory Models*

Radha Jagadeesan, Corin Pitcher, and James Riely

School of Computing, DePaul University

Abstract. The specification of the Java Memory Model (JMM) is phrased in terms of acceptors of execution sequences rather than the standard generative view of operational semantics. This creates a mismatch with language-based techniques, such as simulation arguments and proofs of type safety.

We describe a semantics for the JMM using standard programming language techniques that captures its full expressivity. For data-race-free programs, our model coincides with the JMM. For lockless programs, our model is more expressive than the JMM. The stratification properties required to avoid causality cycles are derived, rather than mandated in the style of the JMM.

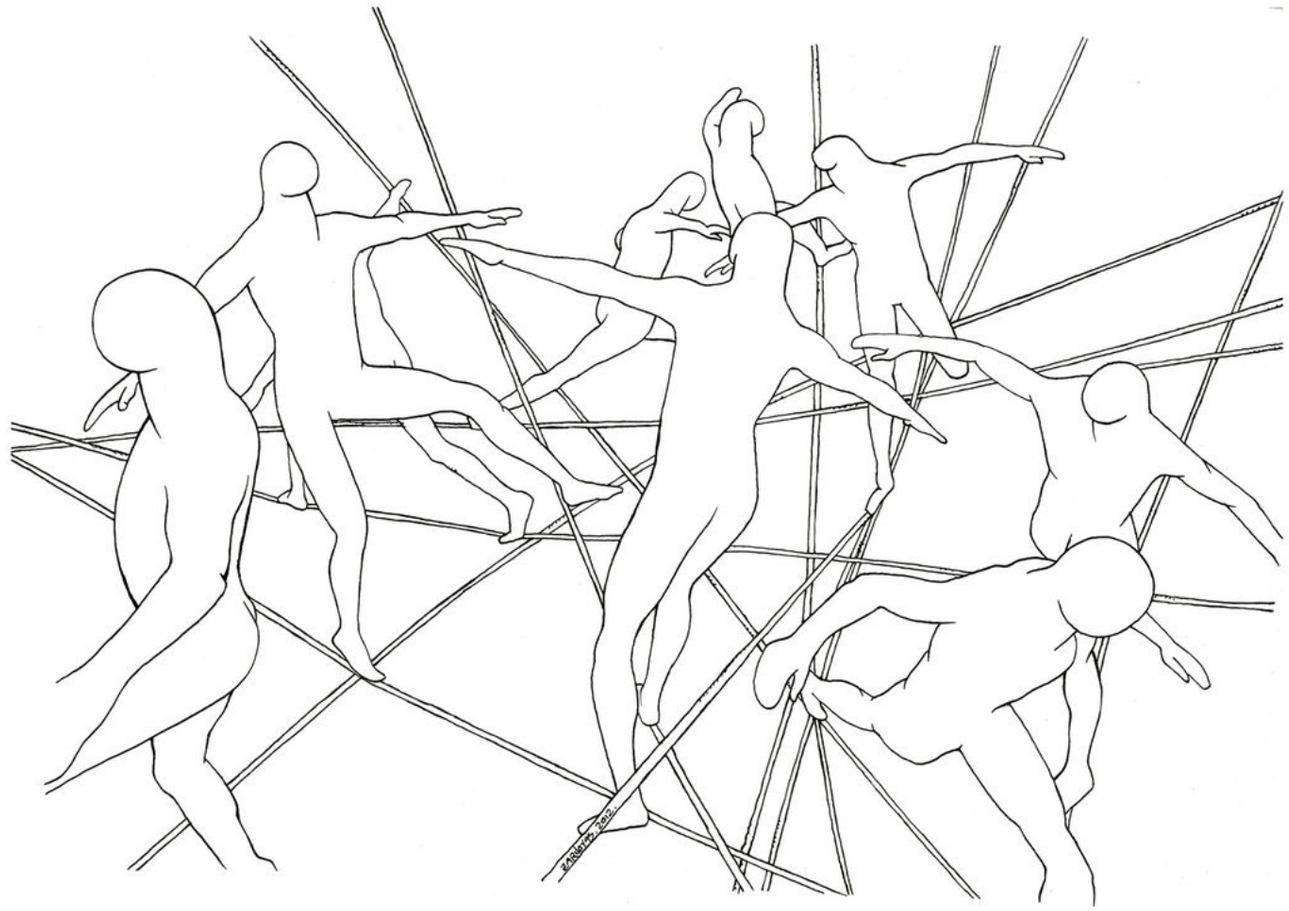
The JMM is arguably non-canonical in its treatment of the interaction of data races and locks as it fails to validate roach-motel reorderings and various peephole optimizations. Our model differs from the JMM in these cases. We develop a theory of simulation and use it to validate the legality of the above optimizations in any program context.

1 Introduction

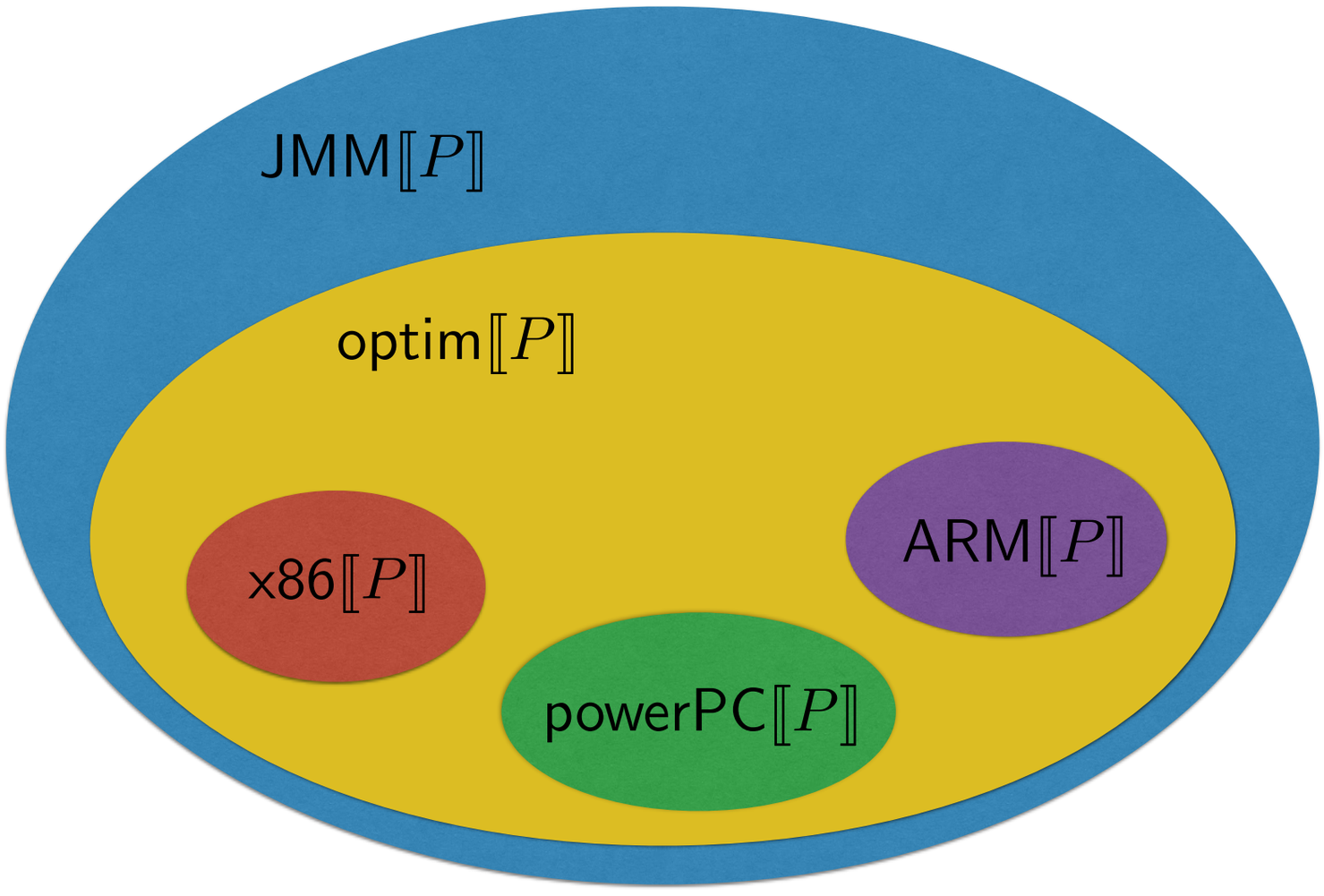
In the context of shared memory imperative programs, Sequential Consistency (SC) (Lamport 1979) enforces a global total order on memory operations that includes the program order of each individual thread in the program. SC may be realized by a traditional interleaving semantics where shared memory is represented as a map from locations to values. It has been observed that SC disables compiler optimizations such as reordering of independent statements. Despite arguments that SC does not impair efficiency (Kamil et al. 2005), this observation and others have motivated a body of work on relaxed memory models; Adve and Gharachorloo (1996) provide a tutorial introduction with detailed bibliography.

A first (conceptual, if not chronological) step in generalizing SC is to consider the Data Race Free (DRF) models. Informally, a program is DRF if no execution of the program in which a write happens concurrently with another operation on

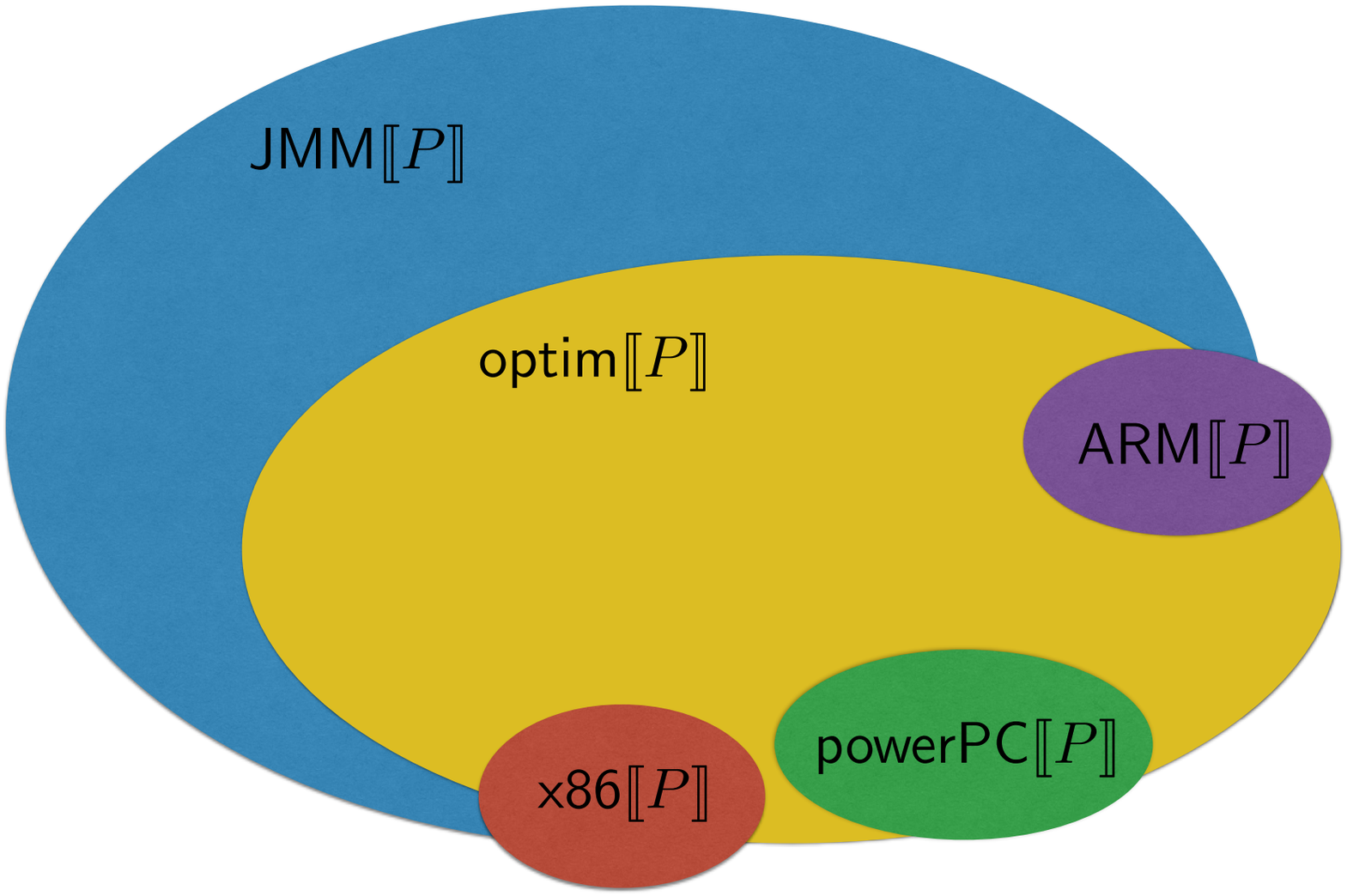
Implementing the JMM



Given a Source Java Program *P* we expect



Given a Source Java Program *P* we expect



The JSR-133 Cookbook for Compiler Writers

http://g.oswego.edu/dl/jmm/cookbook.html

The JSR-133 Cookbook for Compiler Writers

by Doug Lea, with help from members of the JMM mailing list.

dl@cs.oswego.edu.

Preface: Over the 10+ years since this was initially written, many processor and language memory model specifications and issues have become clearer and better understood. And many have not. While this guide is maintained to remain accurate, it is incomplete about some of these evolving details. For more extensive coverage, see especially the work of Peter Sewell and the Cambridge Relaxed Memory Concurrency Group

This is an unofficial guide to implementing the new Java Memory Model (JMM) specified by JSR-133 . It provides at most brief backgrounds about why various rules exist, instead concentrating on their consequences for compilers and JVMs with respect to instruction reorderings, multiprocessor barrier instructions, and atomic operations. It includes a set of recommended recipes for complying to JSR-133. This guide is "unofficial" because it includes interpretations of particular processor properties and specifications. We cannot guarantee that the intepretations are correct. Also, processor specifications and implementations may change over time.

Reorderings

For a compiler writer, the JMM mainly consists of rules disallowing reorderings of certain instructions that access fields (where "fields" include array elements) as well as monitors (locks).

Volatiles and Monitors

The main JMM rules for volatiles and monitors can be viewed as a matrix with cells indicating that you cannot reorder instructions associated with particular sequences of bytecodes. This table is not itself the JMM specification; it is just a useful way of viewing its main consequences for compilers and runtime systems.

Can Reorder	*2nd operation*		
1st operation	Normal Load Normal Store	Volatile Load MonitorEnter	Volatile Store MonitorExit
Normal Load Normal Store			No
Volatile Load			

The JSR-133 Cookbook for Compiler Writers

- A collection of informal arguments about what is possible under the JMM (Java) — *Roach-Motel Semantics* for Volatiles and Locks

Can Reorder	*2nd operation*		
1st operation	Normal Load Normal Store	Volatile Load MonitorEnter	Volatile Store MonitorExit
Normal Load Normal Store			No
Volatile Load MonitorEnter	No	No	No
Volatile store MonitorExit		No	No

The JSR-133 Cookbook for Compiler Writers

- A collection of informal “recipes” indicating how disallow unwanted *reorderings* by adding generic barrier instructions

Required barriers	*2nd operation*			
1st operation	Normal Load	Normal Store	Volatile Load MonitorEnter	Volatile Store MonitorExit
Normal Load				LoadStore
Normal Store				StoreStore
Volatile Load MonitorEnter	LoadLoad	LoadStore	LoadLoad	LoadStore
Volatile Store MonitorExit			StoreLoad	StoreStore

The JSR-133 Cookbook for Compiler Writers

- A final table indicating how to implement each of the generic barriers in the different architectures

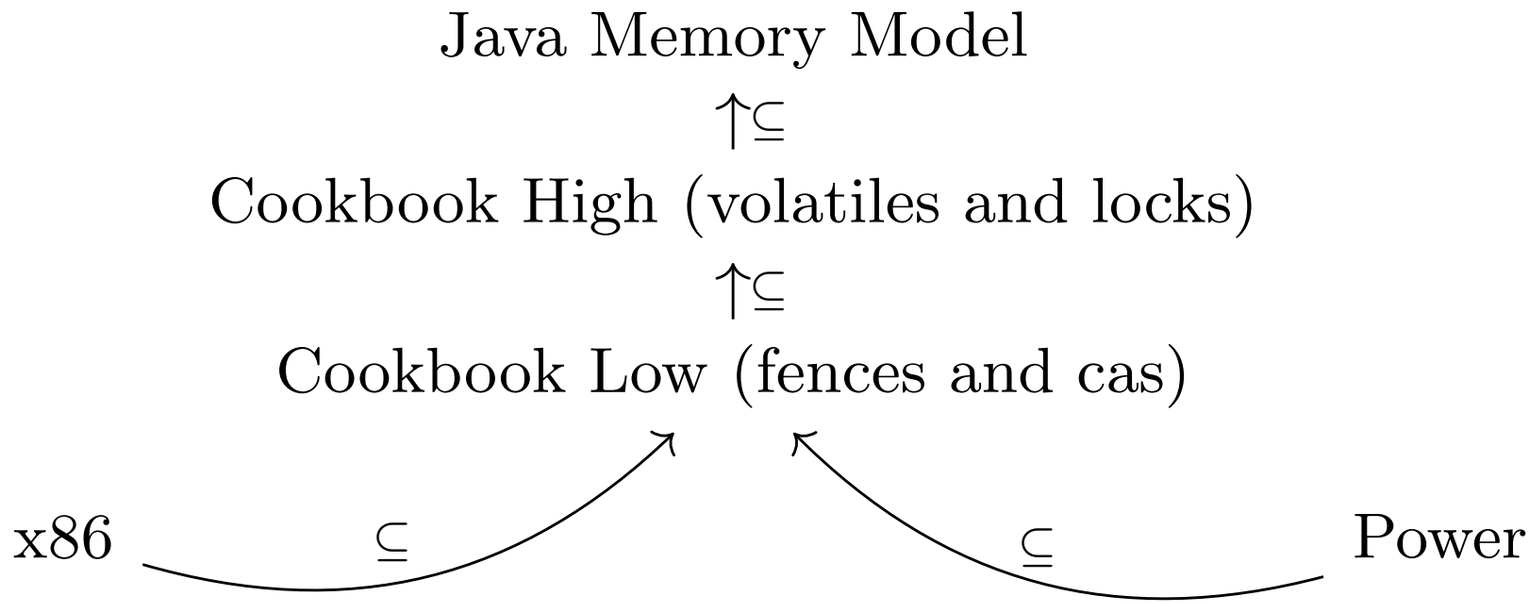
Processor	LoadStore	LoadLoad	StoreStore	StoreLoad	Data dependency orders loads?	Atomic Conditional	Other Atomics	Atomics provide barrier?
sparc-TSO	no-op	no-op	no-op	membar (StoreLoad)	yes	CAS: casa	swap, ldstub	full
x86	no-op	no-op	no-op	mfence or cpuid or locked insn	yes	CAS: cmpxchg	xchg, locked insn	full
ia64	*combine with* st.rel or ld.acq	ld.acq	st.rel	mf	yes	CAS: cmpxchg	xchg, fetchadd	target + acq/rel
arm	dmb (see below)	dmb (see below)	dmb-st	dmb	indirection only	LL/SC: ldrex/strex		target only
ppc	lwsync (see below)	lwsync (see below)	lwsync	hwsync	indirection only	LL/SC: ldarx/stwcx		target only
alpha	mb	mb	wmb	mb	no	LL/SC: ldx_l/stx_c		target only
pa-risc	no-op	no-op	no-op	no-op	yes	*build from* ldcw	ldcw	*(NA)*

The JSR-133 Cookbook for Compiler Writers

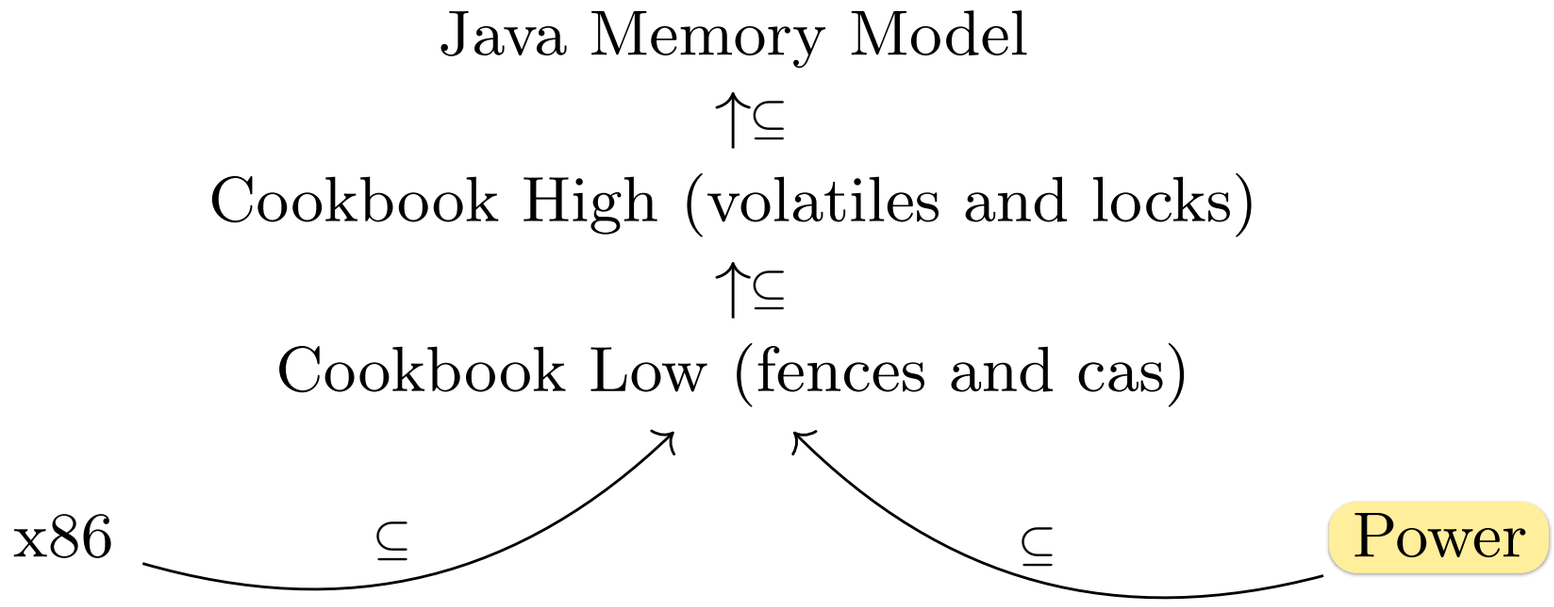
- A final table indicating how to implement each of the generic barriers in the different architectures

Processor	LoadStore	LoadLoad	StoreStore	StoreLoad	Data dependency orders loads?	Atomic Conditional	Other Atomics	Atomics provide barrier?
sparc-TSO	no-op	no-op	no-op	membar (StoreLoad)	yes	CAS: casa	swap, ldstub	full
x86	no-op	no-op	no-op	mfence or cpuid or locked insn	yes	CAS: cmpxchg	xchg, locked insn	full
ia64	*combine with* st.rel or ld.acq	ld.acq	st.rel	mf	yes	CAS: cmpxchg	xchg, fetchadd	target + acq/rel
arm	dmb (see below)	dmb (see below)	dmb-st	dmb	indirection only	LL/SC: ldrex/strex		target only
ppc	lwsync (see below)	lwsync (see below)	lwsync	hwsync	indirection only	LL/SC: ldarx/stwcx		target only
alpha	mb	mb	wmb	mb	no	LL/SC: ldx_l/stx_c		target only
pa-risc	no-op	no-op	no-op	no-op	yes	*build from* ldcw	ldcw	*(NA)*

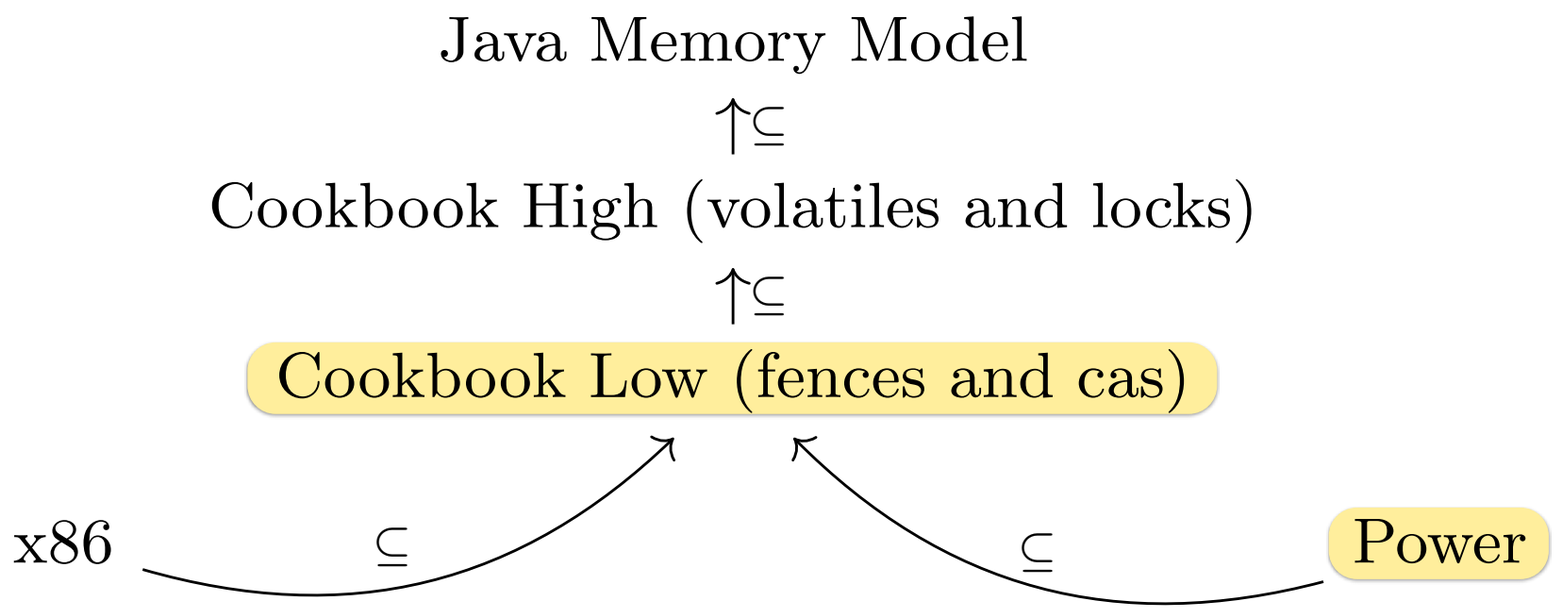
A Semantical Framework to Verify the Cookbook



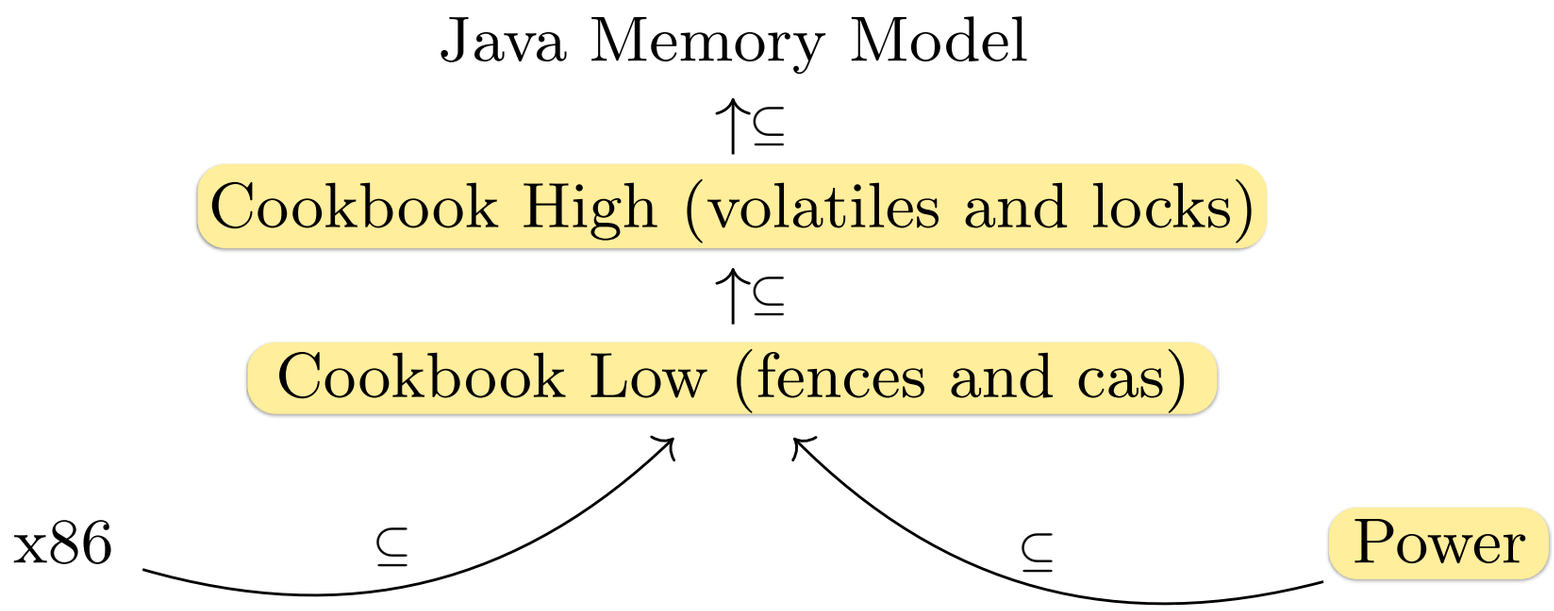
A Semantical Framework to Verify the Cookbook



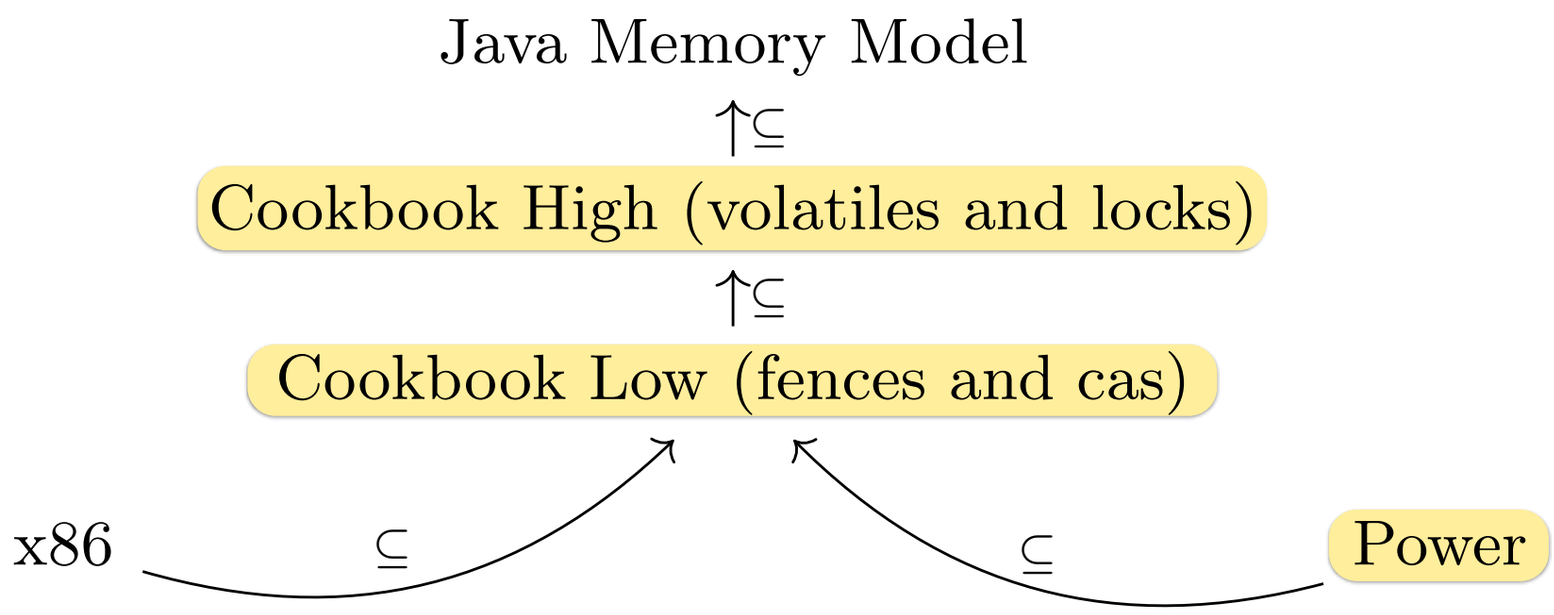
A Semantical Framework to Verify the Cookbook



A Semantical Framework to Verify the Cookbook

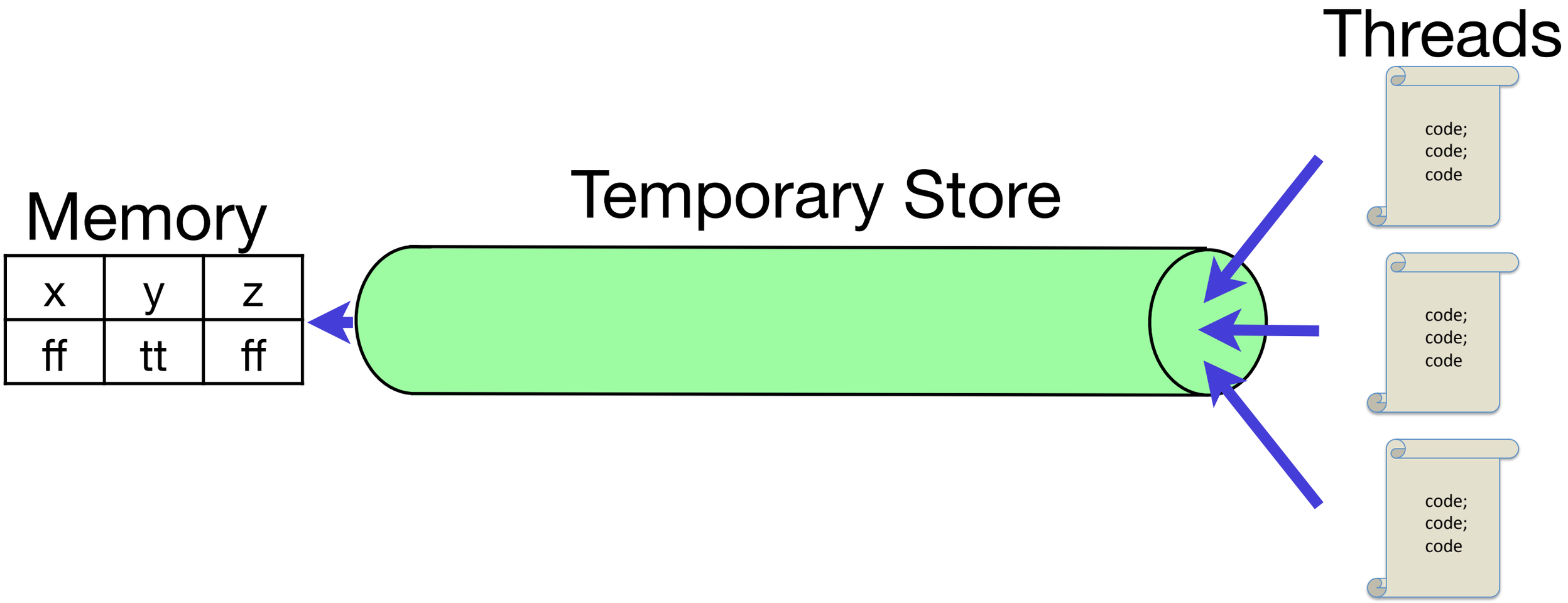


A Semantical Framework to Verify the Cookbook



- We propagate the semantics of Power for *Normal* Variables
- We Give SC semantics to *Volatiles* and *Locks*

Use the Pipeline Framework



- Encode the different architectures: TSO, PPC
- Encode Cookbook Low and Cookbook High
- Prove simulation at every step
- Prove that Cookbook High is a correct implementation of JMM

Power Barriers

- *sync*: Enforces ordering between all operations
  - In our pipeline no action can bypass a sync action of the same thread
  - It requires global consensus
- *lwsync*: It enforces ordering between all operations except: ***wr*** *lwsync rd*
  - It does not enforce consensus

$$\left[\ x := 1;\ \right] \quad \| \quad \left[\ y := 1;\ \right] \quad \| \quad \left[\begin{array}{l} r_1 := x; \\ \text{lwsync}; \\ r_2 := y; \end{array}\right] \quad \| \quad \left[\begin{array}{l} r_3 := y; \\ \text{lwsync}; \\ r_4 := x; \end{array}\right]$$

$$r_1 = r3 = 0 \ \&\ r_2 = r_4 = 1 \text{ is allowed}$$

The JSR-133 Cookbook for Compiler Writers

- A final table indicating how to implement each of the generic barriers in the different architectures

Processor	LoadStore	LoadLoad	StoreStore	StoreLoad	Data dependency orders loads?	Atomic Conditional	Other Atomics	Atomics provide barrier?
sparc-TSO	no-op	no-op	no-op	membar (StoreLoad)	yes	CAS: casa	swap, ldstub	full
x86	no-op	no-op	no-op	mfence or cpuid or locked insn	yes	CAS: cmpxchg	xchg, locked insn	full
ia64	*combine with* st.rel or ld.acq	ld.acq	st.rel	mf	yes	CAS: cmpxchg	xchg, fetchadd	target + acq/rel
arm	dmb (see below)	dmb (see below)	dmb-st	dmb	indirection only	LL/SC: ldrex/strex		target only
ppc	lwsync (see below)	lwsync (see below)	lwsync	hwsync	indirection only	LL/SC: ldarx/stwcx		target only
alpha	mb	mb	wmb	mb	no	LL/SC: ldx_l/stx_c		target only
pa-risc	no-op	no-op	no-op	no-op	yes	*build from* ldcw	ldcw	*(NA)*

IRIW with Volatiles

$$v_0 = v_1 = 0 \text{ and } v_0, v_1 \text{ are volatile}$$

$$\left[\ v_0 := 1;\ \right] \quad \| \quad \left[\ v_1 := 1;\ \right] \quad \| \quad \left[\begin{array}{l} r_1 := v_0; \\ r_2 := v_1; \end{array}\right] \quad \| \quad \left[\begin{array}{l} r_3 := v_1; \\ r_4 := v_0; \end{array}\right]$$

$$r_1 = r3 = 0 \ \&\ r_2 = r_4 = 1 \text{ is allowed}$$

IRIW with Volatiles

$$v_0 = v_1 = 0 \text{ and } v_0, v_1 \text{ are volatile}$$

$$\left[\ v_0 := 1;\ \right] \quad \| \quad \left[\ v_1 := 1;\ \right] \quad \| \quad \left[\begin{array}{l} r_1 := v_0; \\ r_2 := v_1; \end{array}\right] \quad \| \quad \left[\begin{array}{l} r_3 := v_1; \\ r_4 := v_0; \end{array}\right]$$

$$r_1 = r3 = 0 \ \&\ r_2 = r_4 = 1 \text{ is allowed}$$

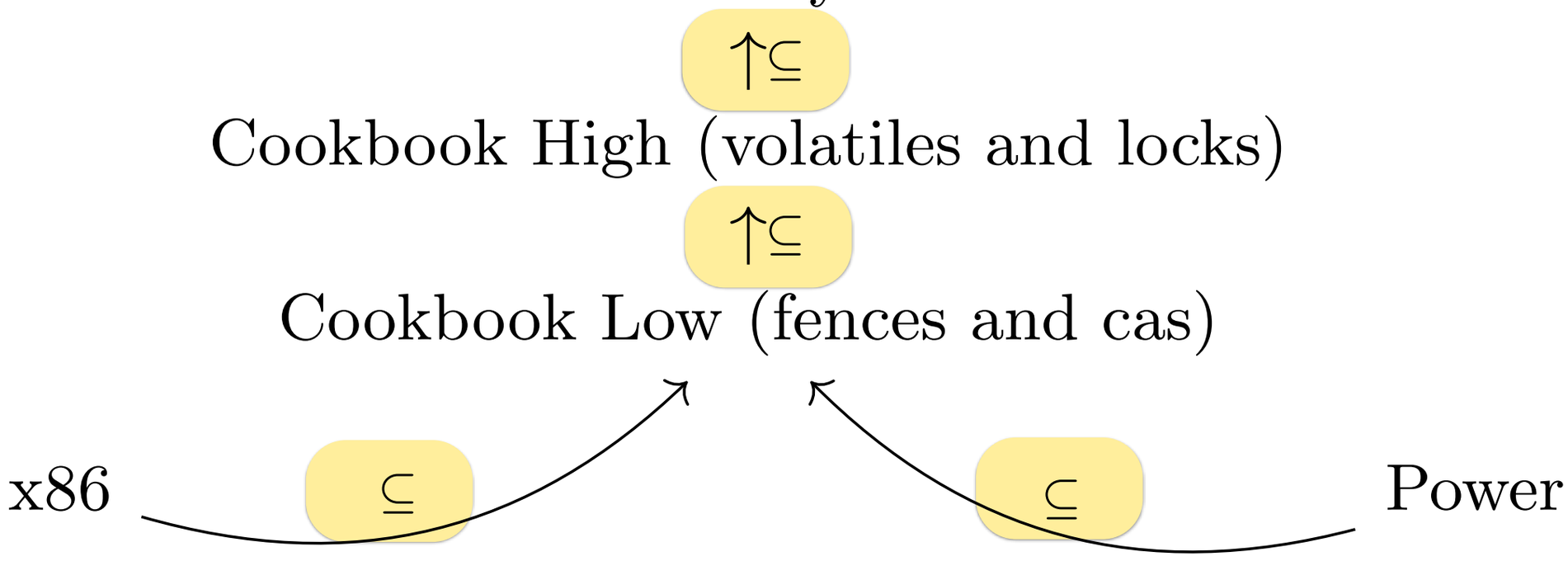
Fix

$$\begin{array}{lcl} \mathrm{PPC}(\texttt{LoadLoad}) & = & \texttt{sync} \\ \mathrm{PPC}(\texttt{LoadStore}) & = & \texttt{lwsync} \\ \mathrm{PPC}(\texttt{StoreStore}) & = & \texttt{lwsync} \\ \mathrm{PPC}(\texttt{StoreLoad}) & = & \texttt{sync} \end{array}$$

Surprises

- IRIW with volatiles is reproducible on a 32-core Power7 machine
  - With two different VM's
- We found bugs in the implementation of volatiles of the Fiji ahead of time compiler
- Many implementers of JVM's follow the Cookbook

Guarantees


Java Memory Model
↑⊆
Cookbook High (volatiles and locks)
↑⊆
Cookbook Low (fences and cas)
x86
⊆
⊆
Power

End of day 2