There's Nothing Wrong with Out-of-Thin-Air: Compiler Optimization and Memory Models

Clark Verbrugge*
Allan Kielstra†
Yi Zhang*

*McGill University
†IBM Toronto Lab

MSPC, June 5, 2011
Introduction

• Memory (consistency) models
  – Important part of concurrent systems
    • Concurrent hardware
    • Concurrent languages
  – Define ordering, visibility of R/W
Introduction

- Java Memory Model
  - Revised in 2005
    - Well-defined semantics
    - Allow most/reasonable compiler optimizations
  - Multiple flaws
    - Proposed fixes
Introduction

- Java Memory Model
  - Revised in 2005
    - Well-defined semantics
    - Allow most/reasonable compiler optimizations
  - Multiple flaws
    - Proposed fixes

- Fundamental concerns for optimization
Contents

- Two problems the JMM creates for optimization
  1. Racey programs
  2. Non-racey programs
- A language proposal
  - Example
- Conclusions & Future Work
The Problem (1)

- “Out-of-Thin-Air”
  - A consequence of simplistic MM semantics

```
x = y = 0;

Thread 1
r1 = x;
y = r1;

Thread 2
r2 = y
x = r2;

r1 == r2 == ...?
```

[Manson et al., 2005]
The Problem (1)

- “Out-of-Thin-Air”
  - A consequence of simplistic MM semantics

\[
x = y = 0;
\]

- Thread 1
  - \( r1 = x; \)
  - \( y = r1; \)

- Thread 2
  - \( r2 = y \)
  - \( x = r2; \)

\( r1 == r2 == \ldots ? \) [Manson et al., 2005]
The Problem (1)

- “Out-of-Thin-Air”
  - A consequence of simplistic MM semantics

```plaintext
x = y = 0;

Thread 1
r1 = x;
y = r1;

Thread 2
r2 = y
x = r2;

r1 == r2 == ...?
```

[Manson et al., 2005]
The Problem (1)

- “Out-of-Thin-Air”
  - A consequence of simplistic MM semantics

\[
\begin{align*}
\text{Thread 1} & : r1 = x; \\
& \quad y = r1; \\
\text{Thread 2} & : r2 = y \\
& \quad x = r2; \\
\end{align*}
\]

\[x = y = 0;\]

\[r1 == r2 == \ldots?\]

[Manson et al., 2005]
The Problem (1)

- “Out-of-Thin-Air”
  - A consequence of simplistic MM semantics

\[
x = y = 0;
\]

\[
\begin{align*}
\text{Thread 1} & \quad \text{Thread 2} \\
\text{r1 = x;} & \quad \text{r2 = y} \\
y = \text{r1;} & \quad x = \text{r2;} \\
\end{align*}
\]

\[
r1 == r2 == 42
\]

[Manson et al., 2005]
The Problem (1)

- “Out-of-Thin-Air”
  - A consequence of simplistic MM semantics
    
    \[
    x = y = 0;
    \]
    
    \[
    \begin{align*}
    \text{Thread 1} & : & r1 &= x; & y &= r1; \\
    \text{Thread 2} & : & r2 &= y & x &= r2;
    \end{align*}
    \]
    
    \[
    r1 \equiv r2 \equiv 42
    \]
  
  - Avoid out-of-thin-air values
    - Ensure causality for all visible values

[Manson et al., 2005]
The Problem with the Solution (1)

- What about compiler optimization?
  - Remember, we want to allow many opts!
  - But compiler opts reuse space...
    - Speculative optimizations
    - Advanced, algorithmic improvements
      - e.g. ...
The Problem with the Solution (1)

x.f = 0;
for (int i=0;i<UPPER;i++) {
    if (x.a[i]) ++x.f;
}

• If lots of true values, a more efficient version:

x.f = UPPER;
for (int i=0;i<UPPER;i++) {
    if (!x.a[i]) --x.f;
}

  - Fewer writes!

  - But now there are out-of-thin-air values...

• x.f contains UPPER...n vs 0..n
The Problem with the Solution (1)

- A surprisingly deep problem!
  - Traditional compiler opts only promise *functional equivalence*
    - Same input, same output
The Problem with the Solution (1)

- Out-of-thin-air guarantees opens this up
  - A variable which cannot be *proved* thread-private, may be arbitrarily *observed*
    - And so must not contain out-of-thin-air values
The Problem (2)

- What about “correct” programs?
  - Program has no data races (DRF)
The Problem (2)

- What about “correct” programs?
  - Program has no data races (DRF)
  - Good programmer!
    - Give a reward
The Solution (2)

- Sequential Consistency for DRF
  - A wonderful property!
    - Program is correctly synchronized
    - Correctly synchronized implies DRF
    - DRF implies SC
    - Programmer understands behaviour!
The Solution (2)

- Sequential Consistency for DRF
  - A wonderful property!
    - Program is correctly synchronized
    - Correctly synchronized implies DRF
    - DRF implies SC
    - Programmer understands behaviour!
  - Considered The Fundamental Property
    - C++, Java, ...
The Problem with the Solution (2)

- DRF is a *runtime* property
  - Not a static one

  ```
  x = y = 0;
  Thread 1
  do {
    r1 = x;
    } while (!r1);
  y = 42;
  Thread 2
  do {
    r2 = y;
    } while (!r2);
  x = 42;
  ```

- Above program is DRF through *divergence*
  - Notice write to y (resp. x) is not dependent on the loop...

[Manson et al., 2005]
The Problem with the Solution (2)

- DRF is a runtime property
  - Not a static one

  ```
x = y = 0;

Thread 1
y = 42;
do {
   r1 = x;
} while (!r1);

Thread 2
x = 42;
do {
   r2 = y;
} while (!r2);

[Manson et al., 2005]
```

- No longer DRF...
  - Disallow these opts?
The Problem with the Solution (2)

- Lots of optimizations move code through control-flow
  - Partial Redundancy Elimination
  - Global code scheduling
  - ....
- New step in optimization strategy
  - Determine runtime control flow
    - Step 1: Solve the halting problem...
The Problem with the Solutions

• Of course we can handle both problems:
  – Conservative race detection
  – DRF-preserving optimizations

• Expensive
  – Accurate conflict detection is hard!

• Optimization quality depends on conflict detection
A Solution to the Problem with the Solutions

- Why not make visibility guarantees explicit?
  - Statically declare shared data
  - Compiler knows what it can do

- Race-free by design

- Borrow ideas from OpenMP, UPC, etc.
  - Not backward compatible in general

MSPC, June 5, 2011
A Race-Free Java

• Syntactic change:
  - Use “volatile” declaration for all shared data

• Semantic change:
  - All non-volatile data is thread-specific
    • Every thread has its own copy
A Race-Free Java

class Q {
    volatile Object x;
}

volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;
A Race-Free Java

class Q {
    volatile Object x;
}

volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

T1          Shared          T2
A Race-Free Java

class Q {
    volatile Object x;
}

volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

MSPC, June 5, 2011
A Race-Free Java

class Q {
    volatile Object x;
}

volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;
A Race-Free Java

class Q {
    volatile Object x;
}

volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

MSPC, June 5, 2011
A Race-Free Java

class Q {
    volatile Object x;
}

volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

MSPC, June 5, 2011
class Q {
    volatile Object x;
}

class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;
A Race-Free Java

class Q {
    volatile Object x;
}

volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;
A Race-Free Java

class Q {
    volatile Object x;
}

volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

T1

T2

Shared
A Race-Free Java

class Q {
    volatile Object x;
}

volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

T1

Shared

T2
A Race-Free Java

class Q {
    volatile Object x;
}
volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

MSPC, June 5, 2011
A Race-Free Java

class Q {
    volatile Object x;
}

volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;
A Race-Free Java

class Q {
    volatile Object x;
}

volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

MSPC, June 5, 2011
A Race-Free Java

class Q {
    volatile Object x;
}

volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

MSPC, June 5, 2011
A Race-Free Java

class Q {
    volatile Object x;
}

volatile class P {
    Object x;
}

volatile P v;
P w;
Q a;

Thread 1
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

Thread 2
v = new P();
w = new P();
a = new Q();
v.x = w;
w.x = v;
a.x = w;

MSPC, June 5, 2011
A Race-Free Java

- SC and DRF as a language given
- Makes correctness a baseline
  - Still can optimize
    - Reduce/eliminate volatile requirements
    - But starting from a trivially known safe state
A Race-Free Java

- Lots of issues to think about
  - Shared to/from local
  - Different copy in/out semantics?
  - Type system changes
  - GC impact
  - Synchronization (locks)
    - Separate atomicity from visibility requirements
Conclusions

- Need to do *something*
  - JMM too restrictive
    - Observability requirements are subtle
    - Conservative safety restricts optimization

- Basic dichotomy in optimization approach
  a) Start from unknown, prove safe, optimize
  b) Start from trivially safe, optimize
Future Work

- Fully develop the language
  - Explore larger examples
    - Need to show programmability too!
  - Prototype compiler
    - Work underway using JikesRVM
- Optimize thread-local/specific data
  - Including copy-in/out models
Thank You

Questions?