Prescient Memory: Exposing Weak Memory Model Behavior by Looking into the Future

MAN CAO
JAKE ROEMER
ARITRA SENGUPTA
MICHAEL D. BOND
Parallel Programming is Hard
Parallel Programming is Hard

• Shared-memory
Parallel Programming is Hard

• Shared-memory

• Difficult to be both correct and scalable
Parallel Programming is Hard

- Shared-memory

- Difficult to be both correct and scalable
  - Data race
Parallel Programming is Hard

- Shared-memory

- Difficult to be both correct and scalable
  - Data race
    - Fundamentally, lacks strong semantic guarantees
Example #1: Weak Semantics

```java
Foo data = null;
boolean flag = false;

T1

data = new Foo();
flag = true;

if (flag)
data.bar();
```

T2
Example #1: Weak Semantics

Foo data = null;
boolean flag = false;

data = new Foo();
flag = true;

if (flag)
data.bar();

Null pointer exception!
Example #1: Weak Semantics

Foo data = null;
boolean flag = false;

data = new Foo();
flag = true;

if (flag)
data.bar();

No data dependence
Exposing Behaviors of Data Races

• Existing Approaches
  • Dynamic analyses

• Model checkers
Exposing Behaviors of Data Races

• Existing Approaches
  • Dynamic analyses
    - Limitation: coverage
  • Model checkers
    - Limitation: scalability
Exposing Behaviors of Data Races

• Existing Approaches
  • Dynamic analyses
    - Limitation: coverage
  • Model checkers
    - Limitation: scalability

• Prescient Memory (PM)

*Dynamic analysis* with better coverage
Outline

• Memory Models and Behaviors of Data Races

• Design
  • Prescient Memory (PM)
  • PM-profiler
  • PM Workflow

• Evaluation
Memory Model

• Defines possible values that a load can return
Memory Model

- Defines possible values that a load can return

Strong
- Sequential Consistency (SC)
- Impractical to enforce
Memory Model

- Defines possible values that a load can return

**Strong**
- Sequential Consistency (SC)
- Impractical to enforce

**Weak**
- Enables compiler & hardware optimizations
- DRF0, C++11, Java
Behaviors Allowed by Memory Models

DRF0 Memory Model

Java Memory Model (JMM)
Behaviors Allowed by Memory Models

- **DRF0 Memory Model**
  - Data-race-free execution
  - Racy execution

- **Java Memory Model (JMM)**
  - Strong semantics (SC)
  - No semantics
Behaviors Allowed by Memory Models

**DRF0 Memory Model**
- Data-race-free execution
- Racy execution

**Java Memory Model (JMM)**
- Data-race-free execution
- Racy execution

**Semantics**
- Strong semantics (SC)
- No semantics
- Weak semantics
Behaviors Allowed by Memory Models

- **DRF0 Memory Model**
  - Data-race-free execution → Strong semantics (SC)
  - Racy execution → No semantics

- **Java Memory Model (JMM)**
  - Data-race-free execution → Strong semantics (SC)
  - Racy execution → Weak semantics

Racy execution can still lead to surprising behaviors!
Behaviors Allowed in JMM #1: Revisit

Foo data = null;
boolean flag = false;

data = new Foo();
flag = true;

if (flag)
data.bar();
Behaviors Allowed in JMM #1: Revisit

T1

data = new Foo();
flag = true;

Foo data = null;
boolean flag= false;

T2

if (flag)
data.bar();

stale value

latest value
Behaviors Allowed in JMM #1: Revisit

Foo data = null;
boolean flag = false;

T1

data = new Foo();
flag = true;

T2

if (flag)
data.bar();

stale value

latest value

Null pointer exception!
Behaviors Allowed in JMM #1: Revisit

Foo data = null;
boolean flag = false;

T1

data = new Foo();
flag = true;

T2

if (flag)
    data.bar();

Returning stale value can trigger the exception
Behaviors Allowed in JMM #2

int data = flag = 0;

T1
r = data;
flag = 1;

assert r == 0;

T2
while (flag == 0)
{
    data = 1;
}
Behaviors Allowed in JMM #2

```java
int data = flag = 0;

T1
r = data;
flag = 1;

T2
while (flag == 0) {} 
data = 1;

assert r == 0;
```
Behaviors Allowed in JMM #2

\[
\text{int data = flag = 0;}
\]

\[
\text{latest value}
\]

\[
\text{T1}
\]
\[
\text{r = data;}
\]
\[
\text{flag = 1;}
\]

\[
\text{future value}
\]

\[
\text{T2}
\]
\[
\text{while (flag == 0) {}}
\]
\[
\text{data = 1;}
\]

\[
\text{assert r == 0;}
\]
Behaviors Allowed in JMM #2

int data = flag = 0;

T1
r = data;
flag = 1;

T2

while (flag == 0) {}
data = 1;

assert r == 0;

Valid due to lack of happens-before ordering
Behaviors Allowed in JMM #2

```c
int data = flag = 0;

T1
r = data;
flag = 1;

T2
while (flag == 0) {}
data = 1;

assert r == 0;

Assertion failure!
```
Behaviors Allowed in JMM #2

```c
int data = flag = 0;

T1
r = data;
flag = 1;

T2
while (flag == 0) {} 
data = 1;

assert r == 0;

Assertion failure!
```
Behaviors Allowed in JMM #2

int data = flag = 0;

T1
r = data;
flag = 1;
assert r == 0;

T2
while (flag == 0) {}
data = 1;

Requires returning future value or reordering to trigger the assertion failure
Example #3

```c
int x = y = 0;

T1
r1 = x;
y = r1;

T2
r2 = y;
if (r2 == 1) {
    r3 = y;
    x = r3;
} else x = 1;

assert r2 == 0;
```
Example #3

```
int x = y = 0;

T1
r1 = x;
y = r1;

T2
r2 = y;
if (r2 == 1) {
    r3 = y;
    x = r3;
} else x = 1;
```

JMM disallows r2 == 1 because of causality requirements

– Ševčík and Aspinall, ECOOP, 2008
Example #3

```java
int x = y = 0;

T1
r1 = x;
y = r1;

T2
r2 = y;
if (r2 == 1) {
    r3 = r2;
    x = r3;
} else x = 1;

assert r2 == 0;
```

However, in a JVM, after redundant read elimination
Example #3

```c
int x = y = 0;

T1
r1 = x;
y = r1;

T2
r2 = y;
if (r2 == 1) {
  r3 = r2;
  x = r3;
} else x = 1;

assert r2 == 0;
```

However, in a JVM, after redundant read elimination:

```c
int x = y = 0;

T1
r1 = x;
y = r1;

T2
r2 = y;
if (r2 == 1) {
  x = r2;
} else x = 1;

assert r2 == 0;
```
Example #3

```java
int x = y = 0;

T1
r1 = x;
y = r1;

T2
r2 = y;
if (r2 == 1) {
    r3 = r2;
x = r3;
} else x = 1;

assert r2 == 0;
```

However, in a JVM, after redundant read elimination

```java
r2 = y;
if (r2 == 1) {
    x = r2;
x = r3;
} else x = 1;
```

```java
r2 = y;
x = 1;
```
Example #3

```java
int x = y = 0;

T1
r1 = x;
y = r1;

T2
r2 = y;
if (r2 == 1) {
r3 = r2;
x = r3;
} else x = 1;
assert r2 == 0;
```

However, in a JVM, after redundant read elimination

```java
r2 = y;
if (r2 == 1) {
x = r2;
} else x = 1;
```

Assertion failure possible!
Behaviors Allowed by Memory Models and JVMs

DRF0 Memory Model

Java Memory Model

Typical JVMs
Behaviors Allowed by Memory Models and JVMs

DRF0 Memory Model

Java Memory Model

Typical JVMs

Unsatisfactory, impractical to enforce
Exposing Behaviors of Example #3

```c
int x = y = 0;

T1
r1 = x;
y = r1;

T2
r2 = y;
if (r2 == 1) {
    r3 = y;
x = r3;
} else x = 1;

assert r2 == 0;
```
Exposing Behaviors of Example #3

T1
   r1 = x; // r1 = 1
   y = r1; // y = 1

T2
   r2 = y; // r2 = 1
   if (r2 == 1) {
      r3 = y; // r3 = 1
      x = r3; // x = 1
   } else x = 1;

assert r2 == 0;

int x = y = 0;
Exposing Behaviors of Example #3

T1
r1 = x;  // r1 = 1
y = r1;  // y = 1

T2
r2 = y;  // r2 = 1
if (r2 == 1) {
    r3 = y;  // r3 = 1
    x = r3;  // x = 1
} else x = 1;
assert r2 == 0;

int x = y = 0;

Consider future value

r1 = 1 justified!

Assertion failure!
Exposing Behaviors of Example #3

int x = y = 0;

T1
r1 = x;
y = r1;

T2
r2 = y;
if (r2 == 1) {
    r3 = y;
x = r3;
} else x = 1;
assert r2 == 0;

Requires returning future value or compiler optimization and reordering to trigger the assertion failure
Exposing Behaviors with Dynamic Analyses

• Typical approaches
  • Simulate weak memory models behaviors \([1,2,3]\)
  • Explore multiple thread interleavings \([4, 5]\)

1. Adversarial Memory, Flanagan & Freund, PLDI’09
2. Relaxer, Burnim et al, ISSTA’11
3. Portend+, Kasikci et al, TOPLAS’15
4. Replay Analysis, Narayanasamy et al, PLDI’07
5. RaceFuzzer, Sen, PLDI’08
Exposing Behaviors with Dynamic Analyses

• Typical approaches
  • Simulate weak memory models behaviors \cite{1,2,3}
  • Explore multiple thread interleavings \cite{4, 5}

• Coverage Limitation
  • Return **stale values only**, not future values
  • **Cannot** expose assertion failures in Examples #2, #3

1. Adversarial Memory, Flanagan & Freund, PLDI’09
2. Relaxer, Burnim et al, ISSTA’11
3. Portend+, Kasikci et al, TOPLAS’15
4. Replay Analysis, Narayanasamy et al, PLDI’07
5. RaceFuzzer, Sen, PLDI’08
Relationship among memory models and exposed behaviors

DRF0 Memory Model

Java Memory Model

Existing Dynamic Analyses

Typical JVMs
Relationship among memory models and exposed behaviors

DRF0 Memory Model

Our Goal

Java Memory Model

Existing Dynamic Analyses

Typical JVMs
Relationship among memory models and exposed behaviors

DRF0 Memory Model

Our Goal

Java Memory Model

Existing Dynamic Analyses

Typical JVMs

Example #1
data = new Foo(); if (flag)
flag = true; data.bar();

Example #2
r = data; while (flag == 0) {}
flag = 1; data = 1;

Example #3
r1 = x; r2 = y;
y = r1; if (r2 == 1) {
r3 = y;
x = r3;
} else x = 1;
Relationship among memory models and exposed behaviors

DRF0 Memory Model

Our Goal

Java Memory Model

Existing Dynamic Analyses

Typical JVMs

Real-world evidence is valuable here!

Example #1
data = new Foo(); if (flag)
flag = true; data.bar();

Example #2
r = data; while (flag == 0) {}
flag = 1; data = 1;

Example #3
1 = x; r2 = y;
y = r1; if (r2 == 1) {
   r3 = y;
x = r3;
} else x = 1;
Outline

• Memory Models and Behaviors of Data Races

• Design
  • Prescient Memory (PM)
  • PM-profiler
  • PM Workflow

• Evaluation
Prescient Memory: Key Idea

- *Speculatively* “guess” a future value at a load

- *Validate* the speculative value at a later store
Prescient Memory: Key Idea

• *Speculatively* “guess” a future value at a load

• *Validate* the speculative value at a later store
Returning Future Values is Tricky

```c
int x = y = 0;

T1
r1 = x;
y = r1;

T2
r2 = y;
if (r2 == 0)
x = 1;

assert r1 == 0 || r2 == 0;
```
Returning Future Values is Tricky

int x = y = 0;

T1
r1 = x;
y = r1;

T2
r2 = y;
if (r2 == 0)
x = 1;

assert r1 == 0 || r2 == 0;
Returning Future Values is Tricky

```c
int x = y = 0;

T1
r1 = x;  // r1 = 1
y = r1;  // y = 1

T2
r2 = y;  // r2 = 1
if (r2 == 0)
    x = 1;

assert r1 == 0 || r2 == 0;
```
Returning Future Values is Tricky

```c
int x = y = 0;

T1
r1 = x; // r1 = 1
y = r1; // y = 1

T2
r2 = y; // r2 = 1
if (r2 == 0)
x = 1;

assert r1 == 0 || r2 == 0;
```

r1 = 1 not justified
Returning Future Values is Tricky

```c
int x = y = 0;

T1
r1 = x; // r1 = 1
y = r1; // y = 1

T2
r2 = y; // r2 = 1
if (r2 == 0)
  x = 1;

assert r1 == 0 || r2 == 0;
```

Invalid execution!
Returning Future Values is Tricky

```c
int x = y = 0;

T1
r1 = x;
y = r1;

T2
r2 = y;
if (r2 == 0)
x = 1;

assert r1 == 0 || r2 == 0;
```

Should never fail!
Returning Future Values is Tricky

```
int x = y = 0;

T1
r1 = x;
y = r1;

T2
r2 = y;
if (r2 == 0)
x = 1;
```

assert r1 == 0 || r2 == 0;

Validating speculative values is necessary to prevent nonsensical results
Prescient Memory: Key Idea

• *Speculatively* “guess” a future value at a load

• *Validate* the speculative value at a later store
Prescient Memory: Key Idea

- *Speculatively* “guess” a future value at a *load*

- *Validate* the speculative value at a *later store*

  - Valid future value
  - Store writes the *same* value
  - Store *races* with load
Prescient Memory: Key Idea

• **Speculatively** “guess” a future value at a load
  • Maintain a per-variable speculative read history
  • Records <logical timestamp, speculative value>

• **Validate** the speculative value at a later store

Valid future value

- Store writes the **same** value
- Store **races** with load
int x = y = 0;
S[x] = ∅

T1  Timestamp: $K_1$

1: $r = x$;
2: $y = 1$;

T2  Timestamp: $K_2$

3: while ($y == 0$) {} 
4: $x = 1$;

assert $r == 0$;
PM Example

\[
\begin{align*}
\text{int } x &= y = 0; \\
S[x] &= \emptyset \\
\text{T1 } &\quad \text{Timestamp: } K_1 \\
1: \ r &= x; & 1 \leftarrow \text{predict(...) } // \text{guess value 1} \\
2: \ y &= 1; & S[x] = \{<K_1, 1>\} \\
\text{T2 } &\quad \text{Timestamp: } K_2 \\
3: \ &\text{while } (y == 0) \{} \\
4: \ x &= 1; \\
\text{assert } r == 0;
\end{align*}
\]
PM Example

```c
int x = y = 0;
S[x] = ∅
```

T1  Timestamp: $K_1$
T2  Timestamp: $K_2$

1: $r = x$;  $1 \leftarrow$ predict(...)  // guess value 1
2: $y = 1$;  $S[x] = \{<K_1, 1>\}$

3: while ($y == 0$) {}

validate $S[x]$:
$K_1 \not\subset K_2 \&\& 1 == 1$

assert $r == 0$;

1 is a valid future value!
Challenges

• How to guess a future value? predict(...)?
Challenges

• How to guess a future value?
  • Which *load* should return a future value?
  • What *value* should be returned?
Challenges

• How to guess a future value?
  • Which *load* should return a future value?
  • What *value* should be returned?

• Solution
  • *Profile* possible future values in a prior run
Profiling Future Values

Helper Dynamic Analysis: PM-profiler

• Maintains a per-variable concrete read history

• At a load, records:
  • <logical timestamp, instruction ID, set of visible values>
Profilering Future Values

Helper Dynamic Analysis: PM-profiler

• At a store, detects:

  Potential future value for a previous load

  Store races with the previous load

  Store writes a value distinct from visible values of the previous load
Prescient Memory Workflow

Data race detector → Racy accesses → PM-Profiler → Potential future values and loads → PM

First Execution → Second Execution
Prescient Memory Workflow

Run-to-run nondeterminism affects validatable future values
Prescient Memory Workflow

Data race detector → Racy accesses → PM-Profiler → Potential future values and loads → PM

First Execution

Second Execution

Run-to-run nondeterminism affects validatable future values

• Solution: record and replay
Prescient Memory Workflow

Data race detector → Racy accesses → PM-Profiler

PM-Profiler → Potential future values and loads → PM

PM → Fuzzy Replay
Prescient Memory Workflow

Data race detector → PM-Profiler → PM

Racy accesses → Potential future values and loads → Fuzzy Replay

Returning a future value could diverge from the record execution
- Best-effort, fuzzy replay
Outline

• Memory Models and Behaviors of Data Races

• Design
  • Prescient Memory (PM)
  • PM-profiler
  • PM Workflow

• Evaluation
Methodology and Implementation

- Compare with

  Adversarial Memory (AM) [Flanagan & Freund, PLDI’09]: a dynamic analysis that only uses stale values
Methodology and Implementation

- **Compare with**
  Adversarial Memory (AM) [Flanagan & Freund, PLDI’09]: a dynamic analysis that only uses *stale* values

- **Platform**
  Jikes RVM 3.1.3
  4-Core Intel Core i5-2500
  Record and Replay [Replay, Bond et al. PPPJ’15]
Methodology and Implementation

◦ Compare with

Adversarial Memory (AM) [Flanagan & Freund, PLDI’09]: a dynamic analysis that only uses stale values

◦ Platform

Jikes RVM 3.1.3
4-Core Intel Core i5-2500
Record and Replay [Replay, Bond et al. PPPJ’15]

◦ Implementation limitation

Does not support reference-type fields
# Exposed Erroneous Behaviors

<table>
<thead>
<tr>
<th>Program</th>
<th>AM</th>
<th>PM</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsqldb</td>
<td>Non-termination</td>
<td>Data corruption</td>
</tr>
<tr>
<td>hsqldb</td>
<td>None</td>
<td>Performance bug</td>
</tr>
<tr>
<td>avrora</td>
<td>Data corruption</td>
<td>Data corruption</td>
</tr>
<tr>
<td>lusearch (GNU Classpath)</td>
<td>Performance bug</td>
<td>None</td>
</tr>
<tr>
<td>sunflow</td>
<td>Null ptr exception</td>
<td>Null ptr exception</td>
</tr>
<tr>
<td>jbb2000</td>
<td>Non-termination</td>
<td>Data corruption</td>
</tr>
<tr>
<td>jbb2000</td>
<td>Data corruption</td>
<td>Data corruption</td>
</tr>
<tr>
<td>jbb2005 (GNU Classpath)</td>
<td>Data corruption</td>
<td>Data corruption</td>
</tr>
<tr>
<td>jbb2005 (GNU Classpath)</td>
<td>Data corruption</td>
<td>None</td>
</tr>
</tbody>
</table>
Exposed Erroneous Behaviors

<table>
<thead>
<tr>
<th>Program</th>
<th>AM</th>
<th>PM</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsqldb</td>
<td>Non-termination</td>
<td>Data corruption</td>
</tr>
<tr>
<td>hsqldb</td>
<td>None</td>
<td>Performance bug</td>
</tr>
<tr>
<td>avrora</td>
<td>Data corruption</td>
<td>Data corruption</td>
</tr>
<tr>
<td>lusearch (GNU Classpath)</td>
<td>Performance bug</td>
<td>None</td>
</tr>
<tr>
<td>sunflow</td>
<td>Null ptr exception</td>
<td>Null ptr exception</td>
</tr>
<tr>
<td>jbb2000</td>
<td>Non-termination</td>
<td>Data corruption</td>
</tr>
<tr>
<td>jbb2000</td>
<td>Data corruption</td>
<td>Data corruption</td>
</tr>
<tr>
<td>jbb2005 (GNU Classpath)</td>
<td>Data corruption</td>
<td>Data corruption</td>
</tr>
<tr>
<td>jbb2005 (GNU Classpath)</td>
<td>Data corruption</td>
<td>None</td>
</tr>
</tbody>
</table>

PM found 3 new erroneous behaviors!
Exposed Erroneous Behaviors

<table>
<thead>
<tr>
<th>Program</th>
<th>AM</th>
<th>PM</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>hsqldb</td>
<td>Non-termination</td>
<td>Data corruption</td>
<td>✓</td>
</tr>
<tr>
<td>hsqldb</td>
<td>None</td>
<td>Performance bug</td>
<td>✓</td>
</tr>
<tr>
<td>avrora</td>
<td>Data corruption</td>
<td>Data corruption</td>
<td>✓</td>
</tr>
<tr>
<td>lusearch (GNU Classpath)</td>
<td>Performance bug</td>
<td>None</td>
<td>✗</td>
</tr>
<tr>
<td>sunflow</td>
<td>Null ptr exception</td>
<td>Null ptr exception</td>
<td>✓</td>
</tr>
<tr>
<td>jbb2000</td>
<td>Non-termination</td>
<td>Data corruption</td>
<td>✓</td>
</tr>
<tr>
<td>jbb2000</td>
<td>Data corruption</td>
<td>Data corruption</td>
<td>✓</td>
</tr>
<tr>
<td>jbb2005 (GNU Classpath)</td>
<td>Data corruption</td>
<td>Data corruption</td>
<td>✓</td>
</tr>
<tr>
<td>jbb2005 (GNU Classpath)</td>
<td>Data corruption</td>
<td>None</td>
<td>✗</td>
</tr>
</tbody>
</table>

PM exposes most bugs that AM found.
Exposed Erroneous Behaviors

<table>
<thead>
<tr>
<th>Program</th>
<th>AM</th>
<th>PM</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsqldb</td>
<td>Non-termination</td>
<td>Data corruption</td>
</tr>
<tr>
<td>hsqldb</td>
<td>None</td>
<td>Performance bug</td>
</tr>
<tr>
<td>avrora</td>
<td>Data corruption</td>
<td>Data corruption</td>
</tr>
<tr>
<td>lusearch (GNU Classpath)</td>
<td>Performance bug</td>
<td>None</td>
</tr>
<tr>
<td>sunflow</td>
<td>Null ptr exception</td>
<td>Null ptr exception</td>
</tr>
<tr>
<td>jbb2000</td>
<td>Non-termination</td>
<td>Data corruption</td>
</tr>
<tr>
<td>jbb2000</td>
<td>Data corruption</td>
<td>Data corruption</td>
</tr>
<tr>
<td>jbb2005 (GNU Classpath)</td>
<td>Data corruption</td>
<td>Data corruption</td>
</tr>
<tr>
<td>jbb2005 (GNU Classpath)</td>
<td>Data corruption</td>
<td>None</td>
</tr>
</tbody>
</table>

Paper contains detailed analysis of each bug.
Conclusion

- First dynamic analysis to expose legal behaviors due to future values in large, real programs
- Successfully found new harmful behaviors due to future values in real programs
- Reaffirms that “benign” races are harmful
- Helps future revisions to language specifications by finding evidence of controversial behaviors in real programs