Hybrid Static-Dynamic Analysis for
Statically Bounded Region Serializability

Aritra Sengupta,
Swarndu Biswas,
Minjia Zhang,
Michael D. Bond

and

Milind Kulkarni

ASPLOS 2015, ISTANBUL, TURKEY
Programming Language Semantics?

- Data Races
  - C++ no guarantee of semantics – “catch-fire” semantics
  - Java provides weak semantics
Weak Semantics

\[
\begin{align*}
T_1 & \quad A \ a = \text{null}; \\
& \quad \text{boolean } \text{init} = \text{false}; \\
T_2 & \quad \text{if (init)} \\
& \quad \text{a.field}++; \\
\end{align*}
\]

\[
\begin{align*}
a & = \text{new } A(); \\
\text{init} & = \text{true};
\end{align*}
\]
Weak Semantics

\[ a = \text{new A}(); \]
\[ \text{init} = \text{true}; \]
\[ \text{if (init)} \]
\[ a.\text{field}++; \]
Weak Semantics

A a = null;
boolean init = false;
a = new A();
init = true;
if (init)
a.field++;

\( T_1 \)

\( T_2 \)
Weak Semantics

\[ T1 \quad T2 \]

\[
\text{init} = \text{true};
\]

\[
\text{if (init)}
\]
\[
\text{a.field}++; 
\]

\[
\text{a = new A();} 
\]
Weak Semantics

T1

init = true;

a = new A();

T2

if (init)
    a.field++;
DRF0

- Atomicity of synchronization-free regions for data-race-free programs
- Data races - no semantics
- C++, Java follow variants of DRF0

– Adve and Hill, ISCA, 1990
Need for Stronger Memory Models

“The inability to define reasonable semantics for programs with data races is not just a theoretical shortcoming, but a fundamental hole in the foundation of our languages and systems…”

- Give better semantics to programs with data races
- Stronger memory models

– Adve and Boehm, CACM, 2010
Sequential Consistency (SC)

Shared memory accesses interleave arbitrarily while each thread maintains program order
Sequential Consistency
An Example Program Under SC

```
int pos = 0
int [ ] buffer = {0, 0}

T1

buffer[pos++] = 5

T2

buffer[pos++] = 6
```
An Example Program Under SC

```plaintext
int pos = 0
int [ ] buffer = [0, 0]

t1 = pos
buffer [t1] = 5
t1 = t1 + 1
pos = t1

T1

T2
t2 = pos
buffer [t2] = 6
t2 = t2 + 1
pos = t2
```
An Example Program Under SC

```
int pos = 0
int [ ] buffer = 0 0

T1

t1 = pos
buffer [t1] = 5
t1 = t1 + 1
pos = t1

T2

t2 = pos
buffer [t2] = 6
t2 = t2 + 1
pos = t2
```
An Example Program Under SC

```cpp
int pos = 0
int [ ] buffer = {0, 0}

T1

pos = = 1
buffer = {6, 0}

T2

t1 = pos
buffer

t1 = t1 + 1

pos = t1

[t2] = 6

t2 = t2 + 1

pos = t2
```
An Example Program Under SC

```c
int pos = 0
int [ ] buffer = {0, 0}

T1
buffer[pos++] = 5
pos == 1
buffer[6] == 0

T2
buffer[pos++] = 6
pos == 2
buffer[6] == 5

buffer[6] == 0
```
An Example Program Under SC

```plaintext
int pos = 0
int[] buffer = {0, 0}

T1
buffer[pos++] = 5
pos = 1
buffer = {5, 0}

T2
buffer[pos++] = 6

SC execution
```
Programmer Assumption

Atomicity of high-level operations
Can SC Eliminate Common Concurrency Bugs?

“...programmers do not reason about correctness of parallel code in terms of interleavings of individual memory accesses…”

• SC does not prevent common concurrency bugs

• Data races dangerous even under SC

– Adve and Boehm, CACM 2010
Run-time cost vs Strength

1. Ouyang et al. ... and region serializability for all. In HotPar, 2013.
Run-time cost vs Strength

1. Ouyang et al. ... and region serializability for all. In HotPar, 2013.
Run-time cost vs Strength

- SC
- DRFO
- Statically Bounded Region Serializability
- Synchronization-free region serializability

Strength

Run-time cost
Contribution

EnfoRSer: An analysis to enforce SBRS practically

Evaluation: Low run-time cost, eliminates real bugs

Synchronization-free region serializability

Statically Bounded Region Serializability

Run-time cost

Strength
New Memory Model: Statically Bounded Region Serializability (SBRS)
Program Execution Behaviors

- Statically Bounded Region
- Serializability

- DRF0
- SC
Statically Bounded Region Serializability (SBRS)

methodCall()

acq(lock)

rel(lock)

Synchronization operations
Method calls
Loop backedges
Statically Bounded Region Serializability (SBRS)

Statically and dynamically bounded

Loop backedges
Under SBRS

pos = 0

buffer = [0, 0]

buffer[pos++] = 5

buffer[pos++] = 6
Under SBRS

pos = 0
buffer [t1] = 5
t1 = t1 + 1
pos = t1

pos = 0
buffer [t2] = 6
t2 = t2 + 1
pos = t2
Under SBRS

\[
\begin{align*}
\text{pos} &= 0 \\
\text{buffer} &= \begin{array}{c}
0 \\
0 \\
\end{array} \\
\text{buffer}[\text{t1}] &= 5 \\
\text{t1} &= \text{t1} + 1 \\
\text{pos} &= \text{t1} \\
\end{align*}
\]

\[
\begin{align*}
\text{t1} &= \text{pos} \\
\text{buffer}[\text{t1}] &= 6 \\
\text{t1} &= \text{t1} + 1 \\
\text{pos} &= \text{t1} \\
\end{align*}
\]

\[
\begin{array}{c}
5 \\
6 \\
\end{array}
\]

\[
\text{pos} &== 2
\]
Under SBRS

T1

pos = 0
buffer

T2

t1 = pos
buffer [t1] = 6
t1 = t1 + 1
pos = t1

pos == 2
buffer

\[
\begin{align*}
t1 &= \text{pos} \\
\text{buffer}[t1] &= 5 \\
t1 &= t1 + 1 \\
pos &= t1
\end{align*}
\]
Bug Elimination

```c
x += 42;
if (o != null)
{
  ...
  o.f;
}
buffer[pos++] = val;
```

- Read-modify-write
- Check before use
- Multi-variable operation
EnfoRSer: A Hybrid Static-Dynamic Analysis to Enforce SBRS
EnfoRSer, an efficient enforcement of SBRS

- Compiler Transformations

+ Runtime Enforcement

= Two-phase Locking
Basic Mechanism

\[ Y = X \]

\[ Y = \]
Basic Mechanism

\[ Y = X \]

Write lock

Read lock
Basic Mechanism

\[ Y = \square \]

\[ = X \]

\[ Y = \]
Basic Mechanism
Basic Mechanism

\[ Y = \quad \text{Ownership transferred} \]
Basic Mechanism

\[
X = X
\]

\[
Y = X
\]
Basic Mechanism

T1

T2

Conflict
Basic Mechanism

\[ Y = X \]

\[ X = Z \]

\[ Z = Y \]

T1

T2

Deadlock
Basic Mechanism

- Lightweight reader-writer locks\(^2\)
- Biased synchronization
- Lose ownership while acquiring locks

Deadlock

Basic Mechanism

Ownership transferred

Two-phase locking violated
Basic Mechanism

\[ Y = X \]

\[ Y = X \]

\[ X = Y \]

\[ Z = \]

T1

T2
Basic Mechanism

Y = 
= X
Y =

Locks acquired
Challenges in Basic Mechanism

\[ Y = X \]

Stores executed
EnfoRSer Atomicity Transformations

- Idempotent: Defer stores until all locks are acquired

- Speculation: Execute stores speculatively and roll back in case of a conflict
Idempotent Transformation

\[ X = Y = Y = X = \]
Idempotence Transformation

\[ X = Y \]
Idempotence Transformation

Stores deferred
Idempotence Mechanism

\[ X = \ldots = Y \]

[Diagram showing idempotence mechanism with locked elements and conflict]
Idempotence Mechanism

\[ X = Y \]

Side-effect free
Idempotence Mechanism

... = Y

Execute stores

X =
Idempotence Challenges
Idempotence Challenges

Loads data dependent on stores

\[ X = Y \]

\[ X = X \]
Idempotence Challenges

- Aliasing between loads and stores
- Data dependence?

\[
\begin{align*}
X &= \ldots \\
 &= Y \\
&= Z
\end{align*}
\]
Speculation Transformation

$X = Y$
Speculation Transformation

Backup store values

Generate roll-back code

old_X = X
X = Y
old_X = Z
Z = old_Z
X = old_X

Z = old_Z
Speculation Mechanism

old_X = X
X =
   = Y
old_X = Z
Z =
Z = old_Z
X = old_X
Conflict
Execute roll-back code
Speculation Mechanism

old_X = X
X = X = Y
old_X = Z
Z = Z = old_Z
X = old_X

Conflict
Speculation Mechanism

\[
\begin{align*}
\text{old}_X &= X \\
X &= Y \\
\text{old}_X &= Z \\
Z &= \text{old}_Z \\
X &= \text{old}_X
\end{align*}
\]

All locks acquired
Speculation Challenges

\[ Y = \]

\[ X = \]

\[ = Z \]
Speculation Challenges

\[ Y = \]

\[ = Z \]

\[ X = \]

Executed store
Speculation Challenges

Y =

X =

Conflict

Y = old_Y

X = old_X

Executed store
Similar to Software Transactional Memory (STM)?

• Idempotent approach similar to STMs that defer stores until a transaction commits

• Speculation approach similar to STMs that execute stores but *undo* them if a transaction needs to abort
Similar to Software Transactional Memory (STM)?

- Idempotent approach similar to STMs that defer stores until a transaction commit.
- Speculation approach similar to STMs that undo stores before a transaction abort.

EnfoRSer provides atomicity of statically bounded regions more efficiently!
Similar to Software Transactional Memory (STM)?

- Idempotent approach similar STMs that defer stores until a transaction commit.
- Speculation approach similar STMs that undo stores before a transaction abort.

EnfoRSer provides atomicity of statically bounded regions more efficiently!

Bounded regions: efficient code generation

Short regions: conservative conflict detection
Implementation and Evaluation
Implementation

• Developed in Jikes RVM 3.1.3

• Code publicly available on Jikes RVM Research Archive
Experimental Methodology

• Benchmarks
  • DaCapo 2006, 9.12-bach
  • Fixed-workload versions of SPECjbb2000 and SPECjbb2005

• Platform
  • AMD Opteron system: 32 cores
Whole-Program Static Analysis

Remove instrumentation from data-race-free accesses
[Naik et al.’s 2006 race detection algorithm, Chord]
EnfoRSer: Run-time Performance

32% overhead on average
EnfoRSer: Run-time Performance

27% overhead on average

% overhead over unmodified JVM
EnfoRSer: Run-time Performance

pjbb2005, over 100% overhead
Cao et al., WoDet 2014
EnfoRSer: Run-time Performance

53% overhead on average (speculation via log)
EnfoRSer: Run-time Performance with and without static race detection

% overhead over unmodified JVM

- Geomean w/ race det
- Geomean w/o race det

- Idempotent
- Speculation
- Speculation via log
Evaluation: Concurrency Errors Avoidance

SBRS’s potential to eliminate concurrency bugs exposed on relaxed memory models
Avoiding Concurrency Errors

<table>
<thead>
<tr>
<th></th>
<th>DRF0 (AM)</th>
<th>SC</th>
<th>SBRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsqldb6</td>
<td>Infinite loop</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>sunflow9</td>
<td>Null pointer</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td></td>
<td>exception</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jbb2000</td>
<td>Corrupt output</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>jbb2000</td>
<td>Infinite loop</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>sor</td>
<td>Infinite loop</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>lufact</td>
<td>Infinite loop</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>moldyn</td>
<td>Infinite loop</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>raytracer</td>
<td>Fails validation</td>
<td>Fails</td>
<td>validation</td>
</tr>
</tbody>
</table>

AM = Adversarial Memory, Flanagan and Freund, PLDI 2010
## Avoiding Concurrency Errors

<table>
<thead>
<tr>
<th></th>
<th>DRF0(AM)</th>
<th>SC</th>
<th>SBRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsqldb6</td>
<td>Infinite loop</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>sunflow9</td>
<td>Null pointer exception</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>jbb2000</td>
<td>Corrupt output</td>
<td>Corrupt output</td>
<td>Correct</td>
</tr>
<tr>
<td>jbb2000</td>
<td>Infinite loop</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>sor</td>
<td>Infinite loop</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>lufact</td>
<td>Infinite loop</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>moldyn</td>
<td>Infinite loop</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>raytracer</td>
<td>Fails validation</td>
<td>Fails validation</td>
<td>Correct</td>
</tr>
</tbody>
</table>

AM = Adversarial Memory, Flanagan and Freund, PLDI 2010
Avoiding Concurrency Errors

<table>
<thead>
<tr>
<th></th>
<th>SC</th>
<th>SBRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsqldb</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>sunflow</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>jbb2000</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>jbb2000</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>sor</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>lufact</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>moldyn</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>raytrace</td>
<td>Fails validation</td>
<td>Correct</td>
</tr>
</tbody>
</table>

Avoids all the errors exposed by AM.

AM = Adversarial Memory, Flanagan and Freund, PLDI 2010
Related Work

- Checks conflicts in bounded region
  DRFx, Marino et al., PLDI 2010
- Checks conflicts in synchronization-free regions
  Conflict Exceptions, Lucia et al., ISCA 2010
- Enforces atomicity of bounded regions
  Bulk Compiler, Ahn et al., MICRO 2009
- Enforces atomicity of synchronization free regions
  … and region serializability for all, Ouyang et al., HotPar 2013

Requires customized hardware

Requires additional cores
Conclusion

EnfoRSer: An analysis to enforce SBRS practically

Evaluation: Low run-time cost, eliminates real bugs

Synchronization-free region serializability

Statically Bounded Region Serializability

Run-time cost

Strength

SC

DRFO