AutoMO: Automatic Inference of Memory Order Parameters for C/C++11

Peizhao Ou and Brian Demsky

University of California, Irvine

Oct 28, 2015
Programming with Multi-cores

Building Blocks

- Shared Memory
- Concurrent Data Structures with Atomics
C/C++11 Memory Model

Pseudo code

```c
int x, y;
// Thread 1
x = 1;
y = 1;
// Thread 2
r1 = y;
r2 = x;
```

C/C++11

```c
atomic_int x, y;
// Thread 1
x.store(1, relaxed);
y.store(1, release);
// Thread 2
r1 = y.load(acquire);
r2 = x.load(relaxed);
```

memory order parameters
C/C++11 Memory Model

- Language-level atomics → portability
- Memory order parameters → complicated

```
int x, y;
// Thread 1
x = 1;
// Thread 2
y = 1;
```

```
atomic_int x, y;
// Thread 1
x.store(1, relaxed);
// Thread 2
y.load(acquire);
```

```
int x, y;
// Thread 1
r1 = y;
// Thread 2
r2 = x.load(relaxed);
```
Memory Order Parameters

- seq_cst
- acquire
- consume
- relaxed

- Parameter weakness
  - stronger
  - weaker

- Speed
  - slower
  - faster
Choosing Memory Order Parameters

```
atomic_int x, y;
// Thread 1                                                // Thread 2
x.store(1, ___?___);                     r1 = y.load(____?____);
y.store(1, ___?___);                     r2 = x.load(____?____);
```
Choosing Memory Order Parameters

atomic_int x, y;

// Thread 1
x.store(1, seq_cst);
y.store(1, seq_cst);

// Thread 2
r1 = y.load(seq_cst);
r2 = x.load(seq_cst);

Overly strong parameters hurt performance
Choosing Memory Order Parameters

```c
atomic_int x, y;
// Thread 1
x.store(1, relaxed);
y.store(1, relaxed);

// Thread 2
r1 = y.load(relaxed);
r2 = x.load(relaxed);
```

Too weak parameters → bugs

Wrong
Choosing Memory Order Parameters

atomic_int x, y;

// Thread 1
x.store(1, relaxed);
y.store(1, release);

// Thread 2
r1 = y.load(acquire);
r2 = x.load(relaxed);

While ensuring correctness seek for weaker parameters

Difficult
What We Propose

A tool that AUTOMATES the process of configuring memory order parameters.
A Motivational Scenario

Existing Implementations

Port

C/C++11 Implementations

A fairly exhaustive test suite

Some Memory Model

C/C++11 Memory Model

Same test suite has correct behaviors

What parameters??
Our Solution

- All possible behaviors
  - Originally allowed behaviors
    - Allowed behaviors of AutoMO implementation
  - Criterion: Only allow SC behaviors
  - Strong enough parameters
It Boils down to

Infer *memory order parameters*

 Guarantee **SC** for provided test cases
atomc_int x, y;

// Thread 1
x.store(1, __?__);
y.store(1, __?__);

// Thread 2
r1 = y.load(__?__);
r2 = x.load(__?__);

Fill out these blanks with parameters
A Search Problem

atomic_int x, y;
// Thread 1
x.store(1, relaxed);
y.store(1, relaxed);
// Thread 2
r1 = y.load(relaxed);
r2 = x.load(relaxed);

Start with the weakest parameters (relaxed)
A Search Problem

atomic_int x, y;
// Thread 1
x.store(1, relaxed);
y.store(1, relaxed);
// Thread 2
r1 = y.load(relaxed);
r2 = x.load(relaxed);

Such parameter assignment allows non-SC behaviors
A Search Problem

```
atomic_int x, y;

// Thread 1
x.store(1, relaxed);
y.store(1, release);

// Thread 2
r1 = y.load(acquire);
r2 = x.load(relaxed);
```

Try some stronger parameter assignment
A Search Problem

atomic_int x, y;

// Thread 1
x.store(1, relaxed);
y.store(1, release);

// Thread 2
r1 = y.load(acquire);
r2 = x.load(relaxed);

This parameter assignment only allows SC behaviors: terminate
A Search Problem

```c
atomic_int x, y;
// Thread 1
x.store(1, relaxed);
y.store(1, release);
// Thread 2
r1 = y.load(acquire);
r2 = x.load(relaxed);
```

**Question 1:**
How to detect SC violations

**Question 2:**
How to repair SC violations
Overview of Approach

Input: test case

Parameter assignment

run

CDSChecker

each trace

SC Analysis

non-SC trace

Trace Reordering

reordered trace

Parameter Synthesis

stronger parameters

AutoMO

initialize
Overview of Approach

Input: test case

Parameter assignment

run

CDSChecker

each trace

SC Analysis

non-SC trace

Trace Reordering

reordered trace

Parameter Synthesis

AutoMO

initialize

stronger parameters
Overview of Approach

Input: test case

run

CDSChecker

each trace

SC Analysis

non-SC trace

Trace Reordering

reordered trace

Parameter Synthesis

stronger parameters

Parameter assignment

initialize

AutoMO
SC Analysis

A C/C++ trace

SC

Not SC
Trace Reordering

A non-SC trace

- Rearrange non-SC trace to mostly SC
  - expose real SC violations
- Ensure the involved SC violations are repairable
  - preserve \( hb \) & \( SC \) relation

A reordered trace
Parameter Synthesis – Naïve Approach

A reordered non-SC trace

Parameter Synthesis

Try all stronger parameter assignments

Impractical complexity

Stronger parameters
Parameter Synthesis – Our Approach

A reordered non-SC trace

Parameter Synthesis

Two universal non-SC patterns

Heuristic rules to repair SC violations

Stronger parameters
Two Universal Non-SC Patterns

**Stale Read**

1. A: v.store(0)
2. B: v.store(1)
3. C: v.load()

**Future Read**

1. A: v.load()
2. B: v.store(1)
Inference Rule Example

Future Read

Inference Rule
Inference Rule Example

- **happens-before**: a chain of reads-from & sequenced-before: 
  establish synchronization

Inference Rule
Inference Rule Example

- Impose $SC$:
  - both $\rightarrow$ $seq\_cst$

Inference Rule
Termination of Inference Process

- Able to apply some rule for each violation
- Each rule application strengthens at least some parameter
- Finite number of parameter assignments
Benchmarks

- **11 real-world data structures**
  - Barrier
  - Dekker's algorithm
  - Three concurrent queues: SPSC, M&S queue & MPMC
  - Three locks: Linux RW lock, Seqlock & MCS lock
  - Treiber stack
  - Chase-Lev deque
  - Concurrent hashtable
## Inference Algorithm Performance

- On Intel Core i7 3770

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Inference Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chase-Lev deque</td>
<td>536.322</td>
</tr>
<tr>
<td>Dekker</td>
<td>396.756</td>
</tr>
<tr>
<td>Linux RW lock</td>
<td>24.982</td>
</tr>
<tr>
<td>M&amp;S queue</td>
<td>4.808</td>
</tr>
<tr>
<td>MCS lock</td>
<td>4.056</td>
</tr>
<tr>
<td>MPMC</td>
<td>0.143</td>
</tr>
<tr>
<td>Seqlock</td>
<td>0.095</td>
</tr>
<tr>
<td>Barrier</td>
<td>0.019</td>
</tr>
<tr>
<td>Treiber's stack</td>
<td>0.018</td>
</tr>
<tr>
<td>Concurrent hashtable</td>
<td>0.016</td>
</tr>
<tr>
<td>SPSC</td>
<td>0.015</td>
</tr>
</tbody>
</table>

- within 9 min
- 8/11 within 5 sec
As Good As Manual Version

- Dekker
- Linux RW lock
- Treiber's stack
- Seqlock
Better Than Manual Version

• MCS lock
• Barrier
Exposé Bugs of Manual Version

- M&S queue implementation
  - AutoMO infers two stronger parameters
  - Both are necessarily stronger (fixed two bugs)
Close to Manual Version

- Chase-Lev deque
  - Only take 9 min to finish
  - Found an incorrect claim
Overly Strong Parameters

- MPMC & Concurrent hashtable
  - Take advantage of SC-violations
Related Work

- Test behaviors for relaxed language models
  - C/C++: CPPMEM, Nitpick, CDSChecker, Relacy...
  - Axiomatic model: MemSAT

- Detect data race (lock-based)
  - Glodilocks, RacerX, FastTrack, Eraser...

- Automatic parallelization (sequential → parallel)

- Check SC
  - Complexity: At least NP-Complete
  - Hardware M.M.: TRF, Herding cat, CheckFence, DFence...

- Automatically infer SC
  - Dfence (infer fences for hardware memory model)
Conclusion

- **AutoMO**
  - Automatically infers memory order parameters for C/C++11 programs
  - Available on our site (http://plrg.eecs.uci.edu/automo/)
Questions