Towards Understanding the Costs of Avoiding Out-of-Thin-Air Results

Peizhao Ou and Brian Demsky
University of California, Irvine
The Out-of-Thin-Air Problem

- Everything initialized to 0
- Loads & stores on x & y are C++ relaxed atomics

```
// T1
r1 = x;
y = r1;

// T2
r2 = y;
x = 42;
```

Load Buffering

```plaintext
r1 = r2 = 42 ✔
```
The Out-of-Thin-Air Problem

- Everything initialized to 0
- Loads & stores on x & y are C++ relaxed atomics

```
// T1
r1 = x;
y = r1;

// T2
r2 = y;
x = 42;
```

Load Buffering

```
r1 = r2 = 42 ✓
```

```
// T1
r1 = x;
y = r1;

// T2
r2 = y;
x = r2;
```

OOTAA Example

```
r1 = r2 = 42??
```
The Out-of-Thin-Air Problem

- Everything initialized to 0
- Loads & stores on x & y are C++ relaxed atomics

```c
// T1
r1 = x;                                      r2 = y;
y = r1;                                      x = 42;
// T2
r2 = y;
x = 42;
```
Causality Cycles

- Causality cycle
  - a store causes itself to happen!
  - makes reasoning difficult
- Hardware forbids causality cycles
  - respects a notion of syntactic dependency
- Compiler optimizations + relaxed hardware implementation
  → challenging to precisely disallow OOTA executions
In Our Paper

- Two approaches
  - enforce slightly stronger memory models to forbid OOTA results
- LLVM-based implementations
- Initial evaluations on their runtime overheads
Dependency-Preserving Approach

- Targeted towards Java-like languages
  - supposed to run untrusted code
    - may have data races
    - must define semantics for racy programs
- Data races in normal accesses
  - must forbid OOTA results produced by normal accesses
Dependency-Preserving Approach

- Core idea (borrowed from hardware)
  - define a notion of dependency at the language level
  - if a load $L$ may cause a store $S$ to happen, $L \xrightarrow{\text{dep}} S$
  - require dependency $U rf$ is acyclic
  - our dependency is close to hardware dependency
  - only need to preserve dependencies in compilers

```
// T1
r1 = x;
y = r1;
```

```
// T2
r2 = y;
x = r2;
```

$\text{dep U rf cycle, } r1=r2=42 \times$
Example of Dependency Notion

\[
\begin{align*}
// T1 \\
r1 &= x; \\
y &= r1 \times 0 + 42; \\
\end{align*}
\]

\[
\begin{align*}
// T2 \\
r2 &= y; \\
\text{if (r2==r2)} \\
x &= 42; \\
\end{align*}
\]

\[
r1 = r2 = 42x
\]

- More details in our paper
  - when the address of a store depends on some load
  - when a store is conditionally executed...
An LLVM-Based Implementation

Input:
C/C++ Source Code

1. Clang Front End
   Generate
   Unoptimized IR

2. LLVM IR
   Optimizer

Optimize
IR

3. LLVM Backend
   Code Generator
   Generate
   Machine Code

Output:
Object Code
Preserving Dependencies at IR Level

• Focus on a select set of 35 IR passes
  – overhead with only these passes enabled is only 1.8% (over -O3)

• Our implementation
  – disable all other IR passes
  – audit selected passes
  – modify those that can break dependencies
Modified IR Pass

- Modified instcombine, simplifycfg, loop-unrolling…
  - also modified passes that perform store-store reordering, e.g., dead store elimination…
  - more details in our paper

```
r2 = (r1 == r1);
if (r2) ...
```

✗

```
r2 = true;
if (true) ...
```

```
r1 = x;
if (r1) y = 1;
else y = 1;
```

✗

```
r1 = x;
y = 1;
```
Preserve Backend Dependencies

- AArch64 backend
  - more relevant than x86 (relatively strong memory model)
- Modifications
  - data dependencies
    - SelectionDAG-based instruction selection pass (modified)
  - control dependencies
    - codegenprepare (modified)
  - branchfolding (disabled)
Dependency-Preserving Evaluation

- Benchmarks
  - SPEC CPU2006 C/C++ programs
- Baseline
  - stock LLVM 3.8 with O3 option enabled
- Processor
  - Cortex-A72 core (ARMv8) on a Firefly RK3399 board
- Compiler configurations
  - partial optimization (unmodified LLVM with only the selected 35 IR passes enabled)
  - our dependency-preserving compiler
Single-Threaded Runs

- Dependency Preserving — **3.1%** on average & **17.6%** maximum
- Partial Optimization — **1.8%** on average (room for optimizations)
- Speedup on some benchmarks
  - disabling “BranchFolding” pass alone → **-1.5% ~ 1.5%**
Load-Store-Order-Preserving Approach
Load-Store-Order-Order-Preserving Approach

- Targeted towards C/C++-like languages
  - racy operations are labeled as atomics
  - racy non-atomic accesses → undefined semantics
    - OOTA involves racy operations
    - already exclude OOTA for non-atomic accesses
- Only atomics can produce OOTA results
  - relatively rare (especially relaxed atomics)
Load-Store-Order-Preserving Approach

- Core idea: preserve load-store ordering for **atomics**
  - forbid \( sb \cup rf \) cycle
    - effectively forbid \( dep \cup rf \) cycle
  - does not affect normal accesses & single-threaded code
Load-Store-Order-Preserving Implementation

- IR-level passes
  - NO atomic load-store reordering

- AArch64 backend
  - need to add sufficient constraints to enforce load-store ordering

```
r1 = x.load(relaxed);
y.store(0, relaxed);
```

```
ldr w1, [x8]
str wzr, [x9]
```
Preserving Load-Store Ordering in ARMv8

- 6 alternative strategies
  - relaxed loads → acquire loads
  - relaxed stores → release stores
  - insert “DMB LD” fence before relaxed stores
  - add a bogus conditional branch after relaxed loads
  - add a bogus load after relaxed loads
  - taint existing the address of existing stores if any, otherwise add bogus conditional branch
Load-Store-Order-Order-Preserving Evaluation

- Benchmarks
  - 43 concurrent data structures
  - from C++ Libcds library, Facebook Folly library...
  - concurrent queues, hashtables, synchronization...
- Baseline
  - stock LLVM 3.8 with O3 option enabled
- Processor
  - two Cortex-A72 cores on Firefly RK3399 board
Overhead on Multi-Threaded Runs

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquire Load</td>
<td>0.4%</td>
<td>27.5%</td>
</tr>
<tr>
<td>Release Store</td>
<td>3.6%</td>
<td>82.6%</td>
</tr>
<tr>
<td>DMB LD Fence</td>
<td>-0.1%</td>
<td>32.0%</td>
</tr>
<tr>
<td><strong>Bogus Conditional Branch</strong></td>
<td><strong>-0.3%</strong></td>
<td><strong>6.3%</strong></td>
</tr>
<tr>
<td>Bogus Load</td>
<td>2.6%</td>
<td>42.9%</td>
</tr>
<tr>
<td>Extra Dependencies to Store</td>
<td>1.3%</td>
<td>23.2%</td>
</tr>
</tbody>
</table>

- Run with two threads
  - each thread in a single core
- “Bogus Conditional Branch”
  - no overhead on average
- Speedup in some benchmarks
  - possibly due to contention
Conclusion

● Initial evaluation on runtime overheads
  – two approaches that can disallow OOTA results

● Further evaluation needed
  – results generalize across different CPUs, e.g., ARMv7, Power, etc?
  – any applications that make more extensive use of relaxed atomics than concurrent data structures?
  – do full applications change the results for bogus branches by putting additional pressure on branch predictor?
Questions?