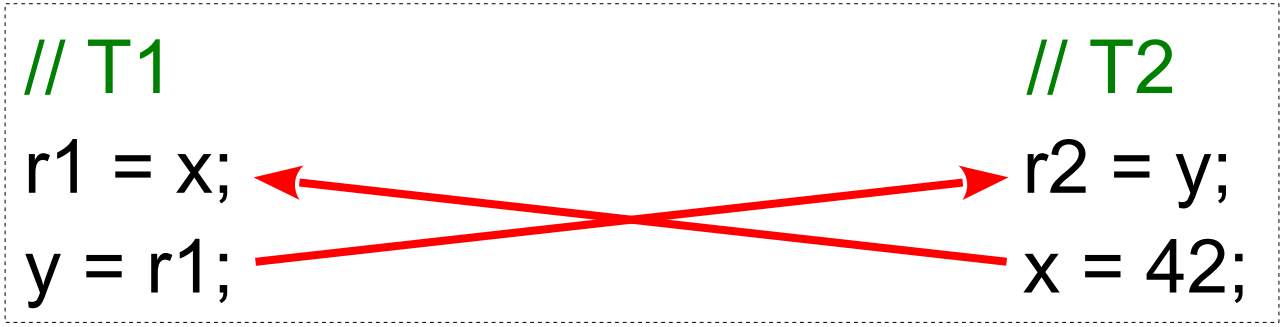
Towards Understanding the Costs of Avoiding Out-of-Thin-Air Results

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

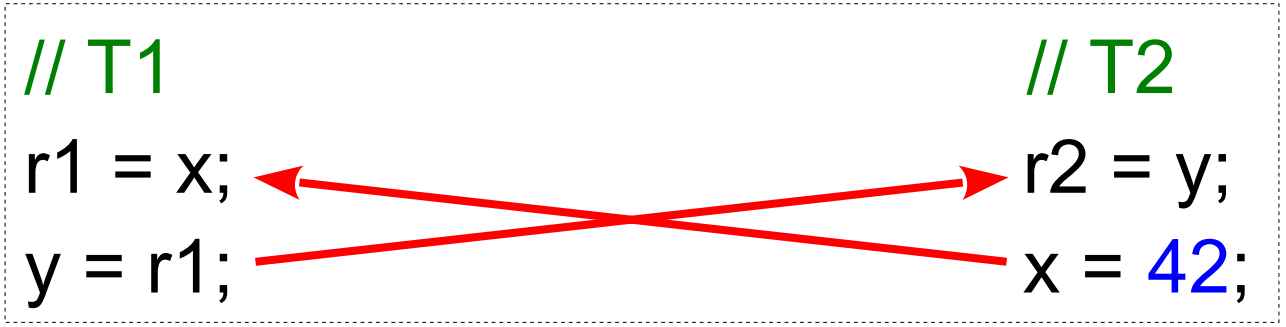


Load Buffering

r1 = r2 = 42 ✔

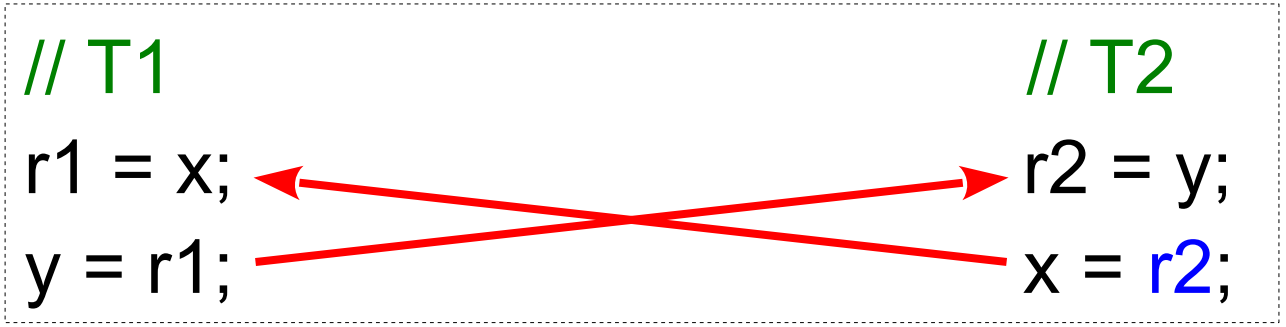
The Out-of-Thin-Air Problem

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



Load Buffering

r1 = r2 = 42 ✔



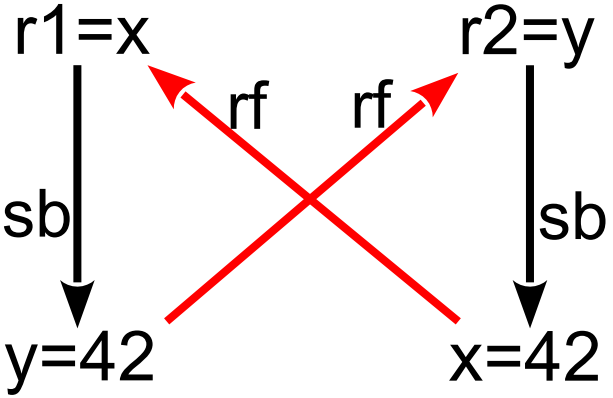
OOTA Example

r1 = r2 = 42??

The Out-of-Thin-Air Problem

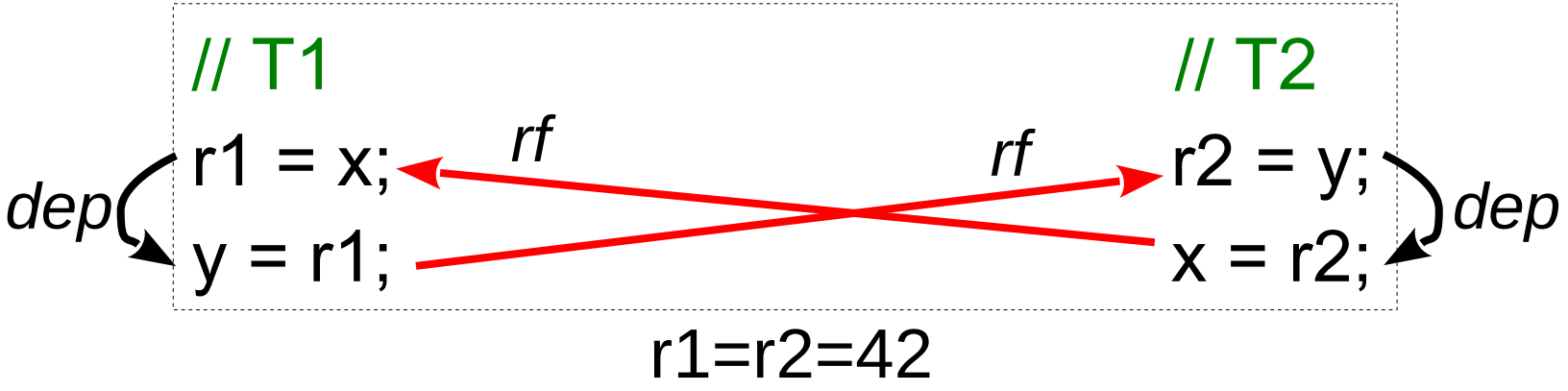
- Everything initialized to 0
- Loads & stores on x & y are C++ relaxed atomics

```
// T1                    // T2
r1 = x;                  r2 = y;
y = r1;                  x = 42;
```




```
// T1                    // T2
r1 = x;                  r2 = y;
y = r1;                  x = r2;
```

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

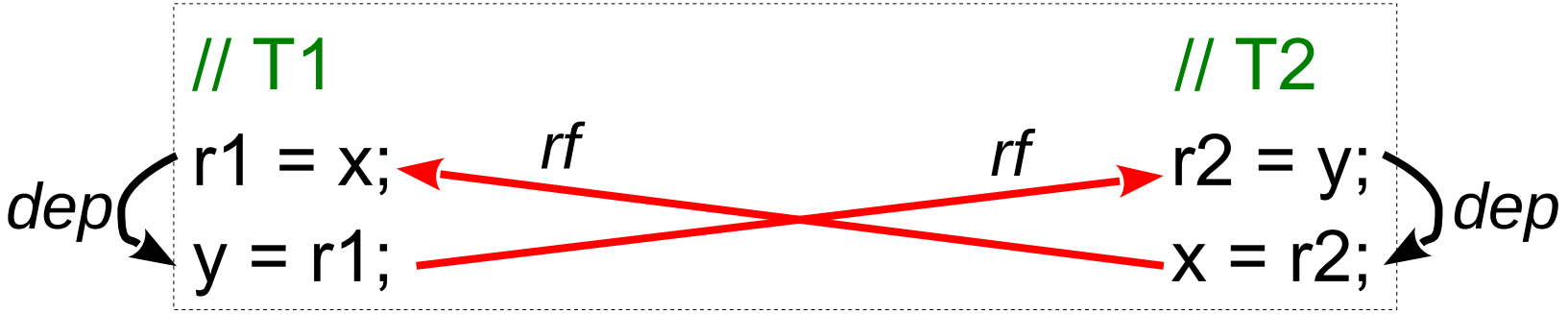
- Two approaches
  – enforce slightly stronger memory models to forbid OOTA results
- LLVM-based implementations
- Initial evaluations on their runtime overheads

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

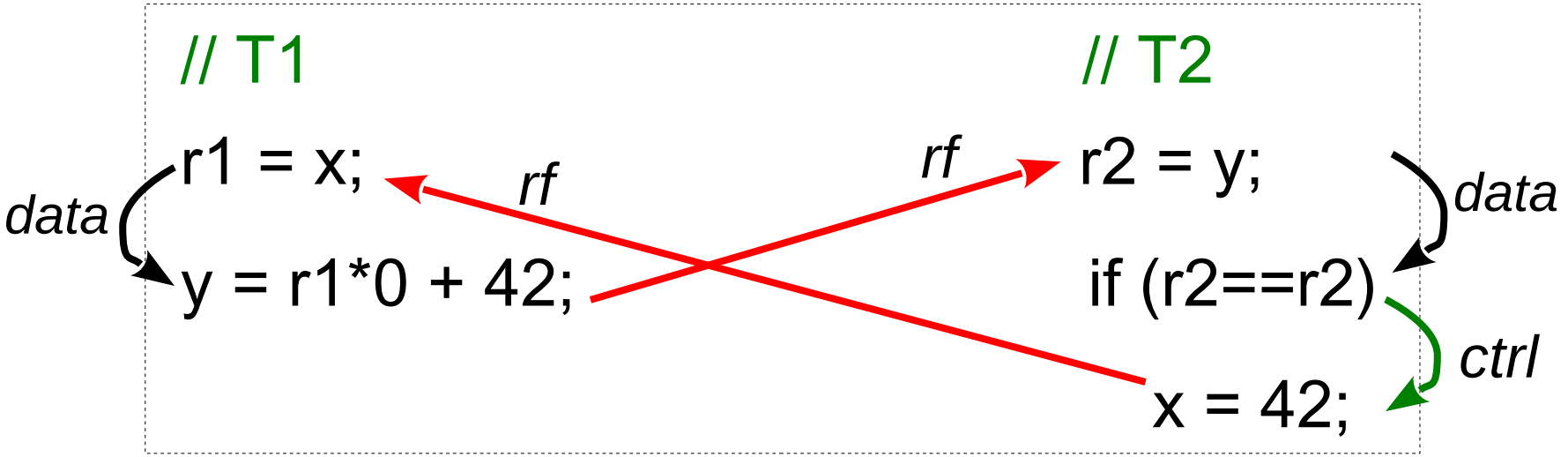
Dependency-Preserving Approach

- 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*** $\xrightarrow{dep}$ ***S***
  - require *dependency* U *rf* is acyclic
  - our dependency is close to hardware dependency
    → only need to preserve dependencies in compilers



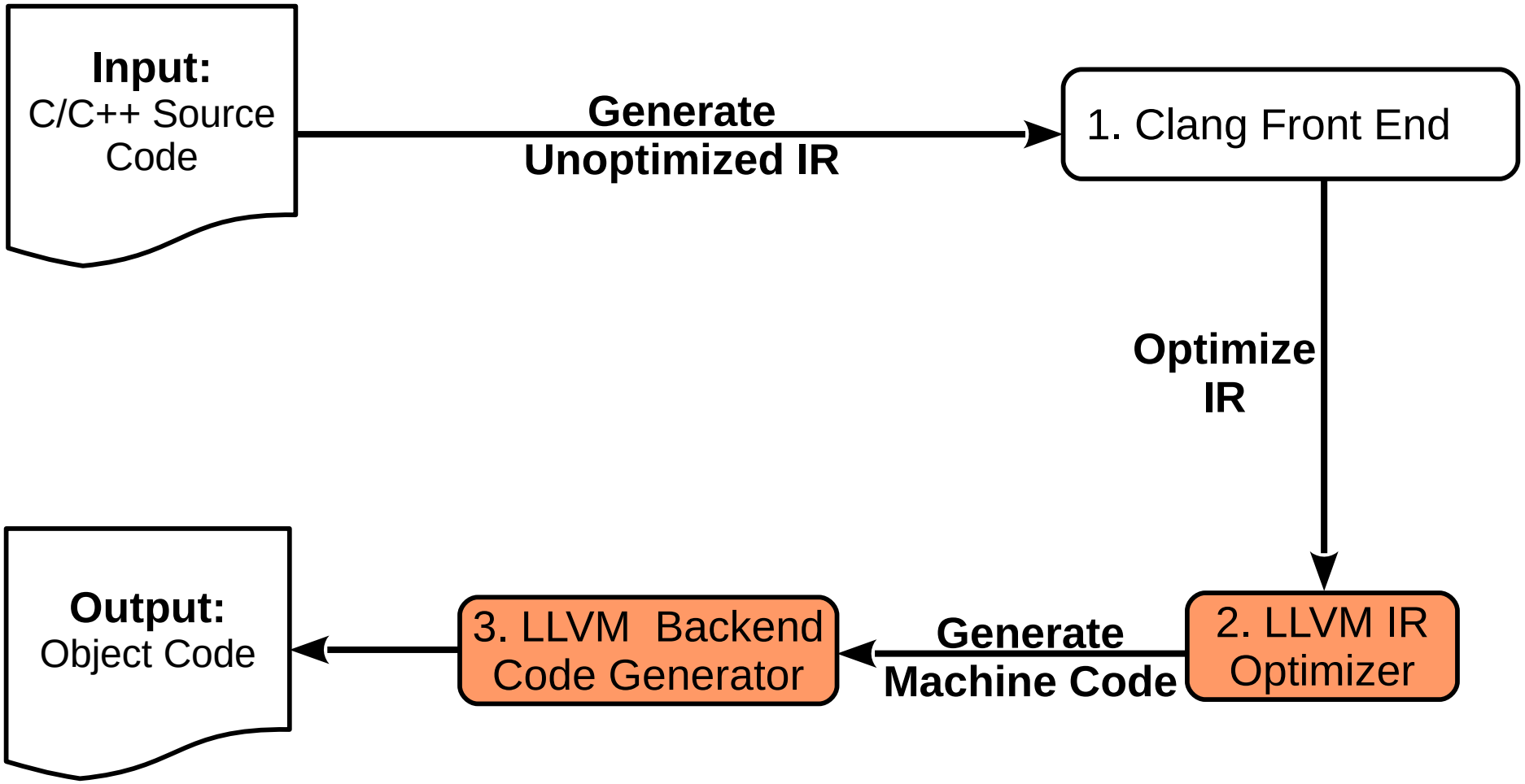
dep U rf cycle, r1=r2=42 ✗

Example of Dependency Notion



- 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



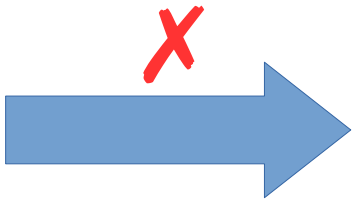
Preserving Dependencies at IR Level

- 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

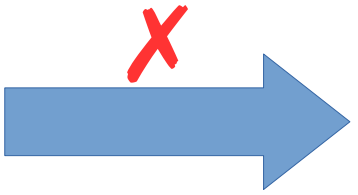
- Modified instcombine, simplifycfg, loop-unrolling…
  – also modified passes that perform store-store reordering, e.g.,dead store elimination…
  – more details in our paper

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



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

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



```
r1 = x;
y = 1;
```

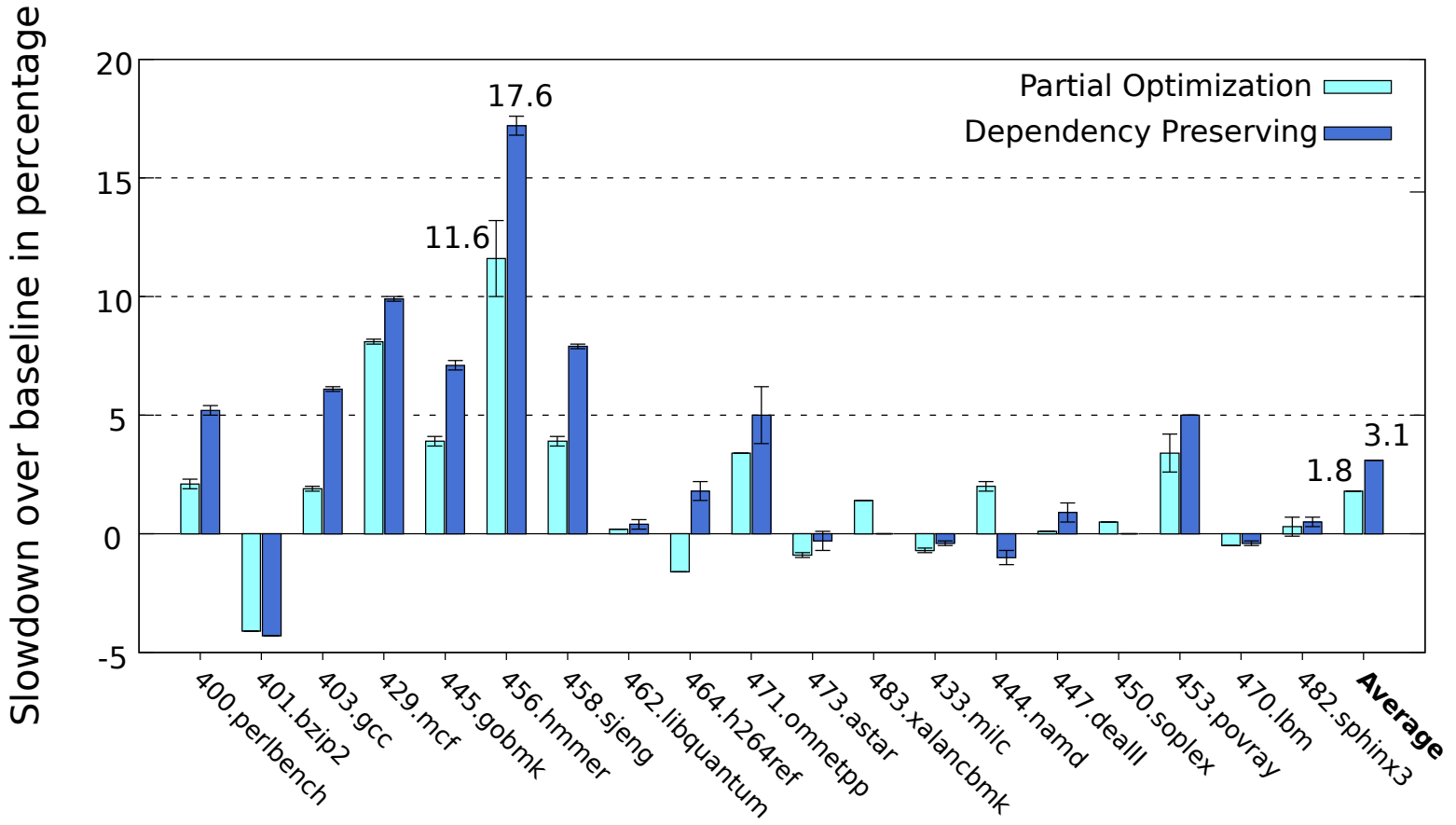
Preserve Backend Dependencies

- 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



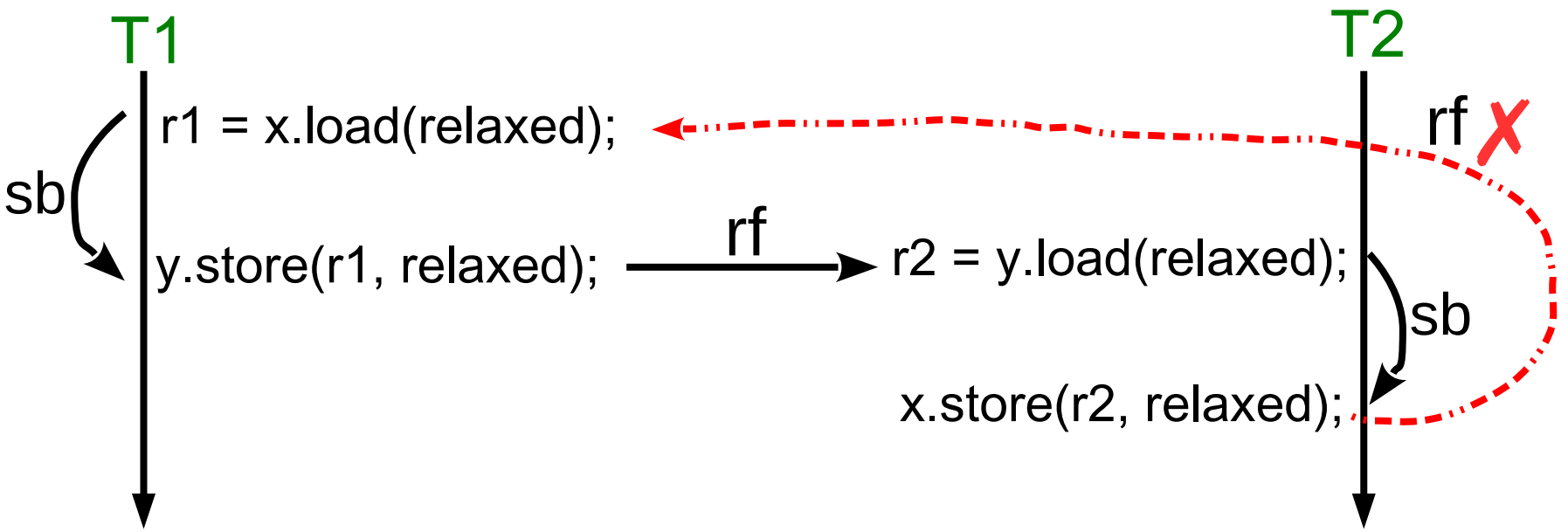
- Dependency Preserving — 3.1% on average & 17.6% maximum
- Partial Optimization — 1.8% on average (room for optimizations)
- Speedup on some benchmarks
  - disabling “BranchFolding” pass alone →-1.5% ~ 1.5%

Load-Store-Order-Preserving Approach

Load-Store-Order-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



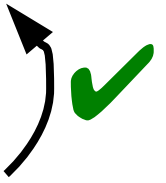
- Core idea: preserve load-store ordering for **atomics**
  - forbid *sb* U *rf* cycle
    - → effectively forbid *dep* U *rf* cycle
  - does not affect normal accesses & single-threaded code

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-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

Strategy	Average	Maximum
Acquire Load	0.4%	27.5%
Release Store	3.6%	82.6%
DMB LD Fence	-0.1%	32.0%
Bogus Conditional Branch	**-0.3%**	**6.3%**
Bogus Load	2.6%	42.9%
Extra Dependencies to Store	1.3%	23.2%

- 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?