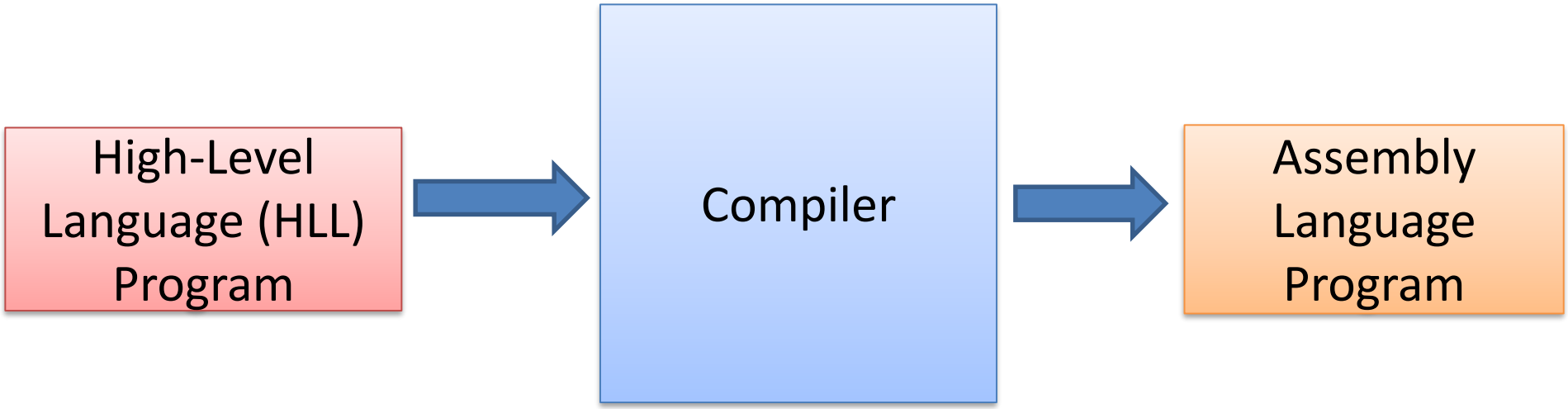
C11 Compiler Mappings: Exploration, Verification, and Counterexamples

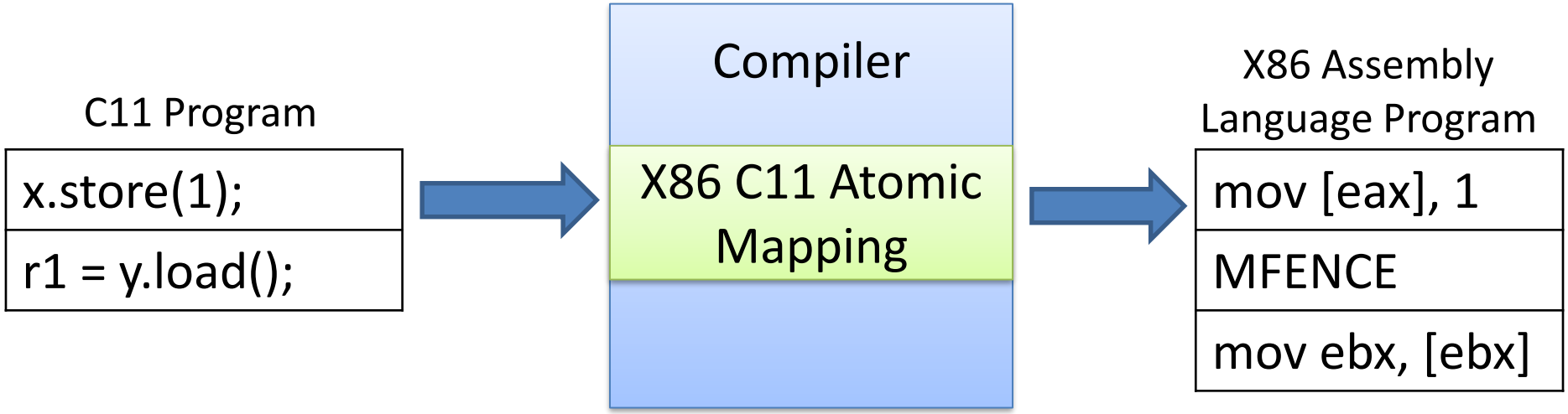
Yatin Manerkar
Princeton University
manerkar@princeton.edu
http://check.cs.princeton.edu
November 22nd, 2016

Compilers Must Uphold HLL Guarantees



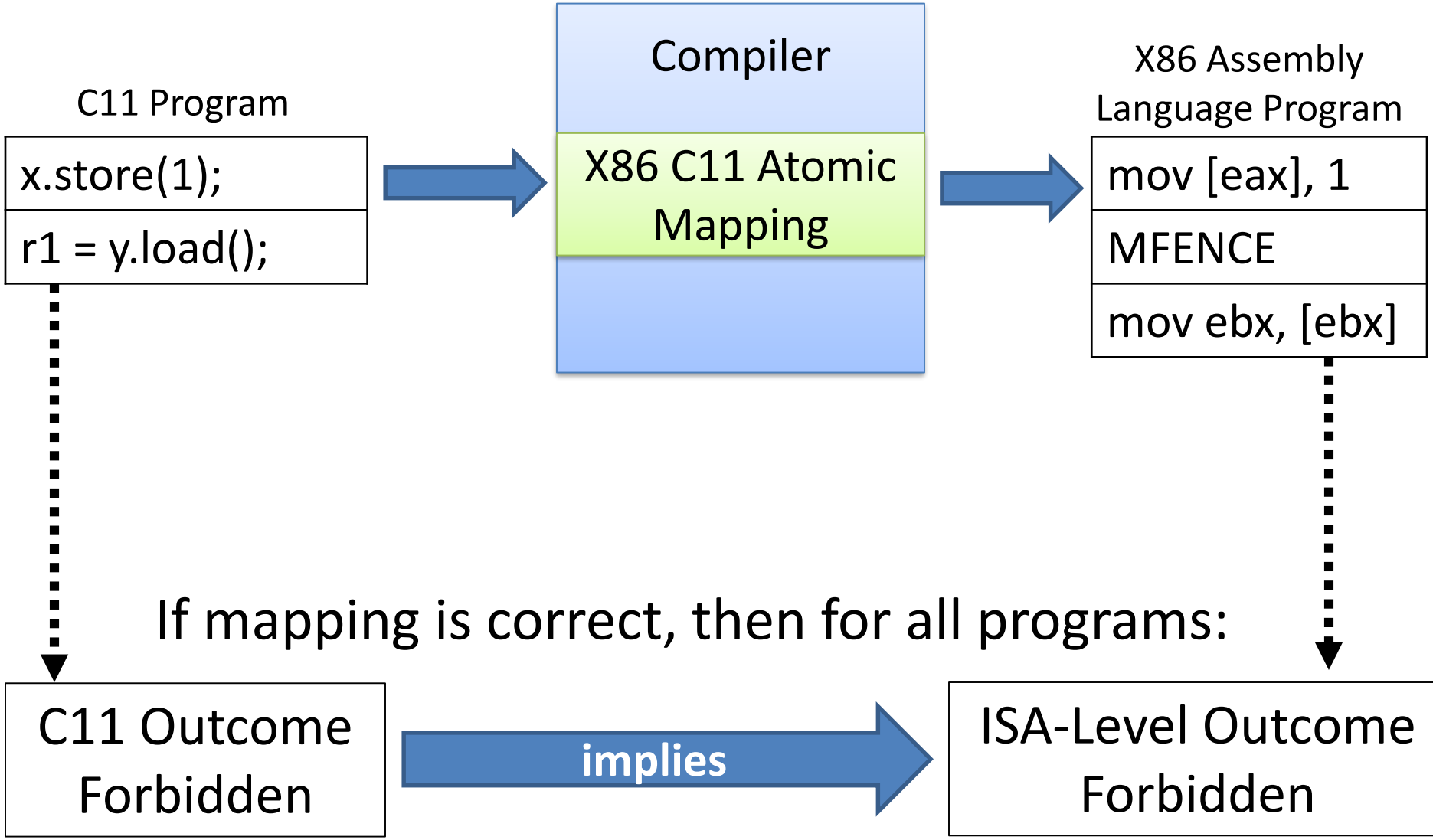
- Compiler translates HLL statements into assembly instructions
- Code generated by compiler must provide functionality required by HLL program

Compilers Must Uphold HLL Guarantees

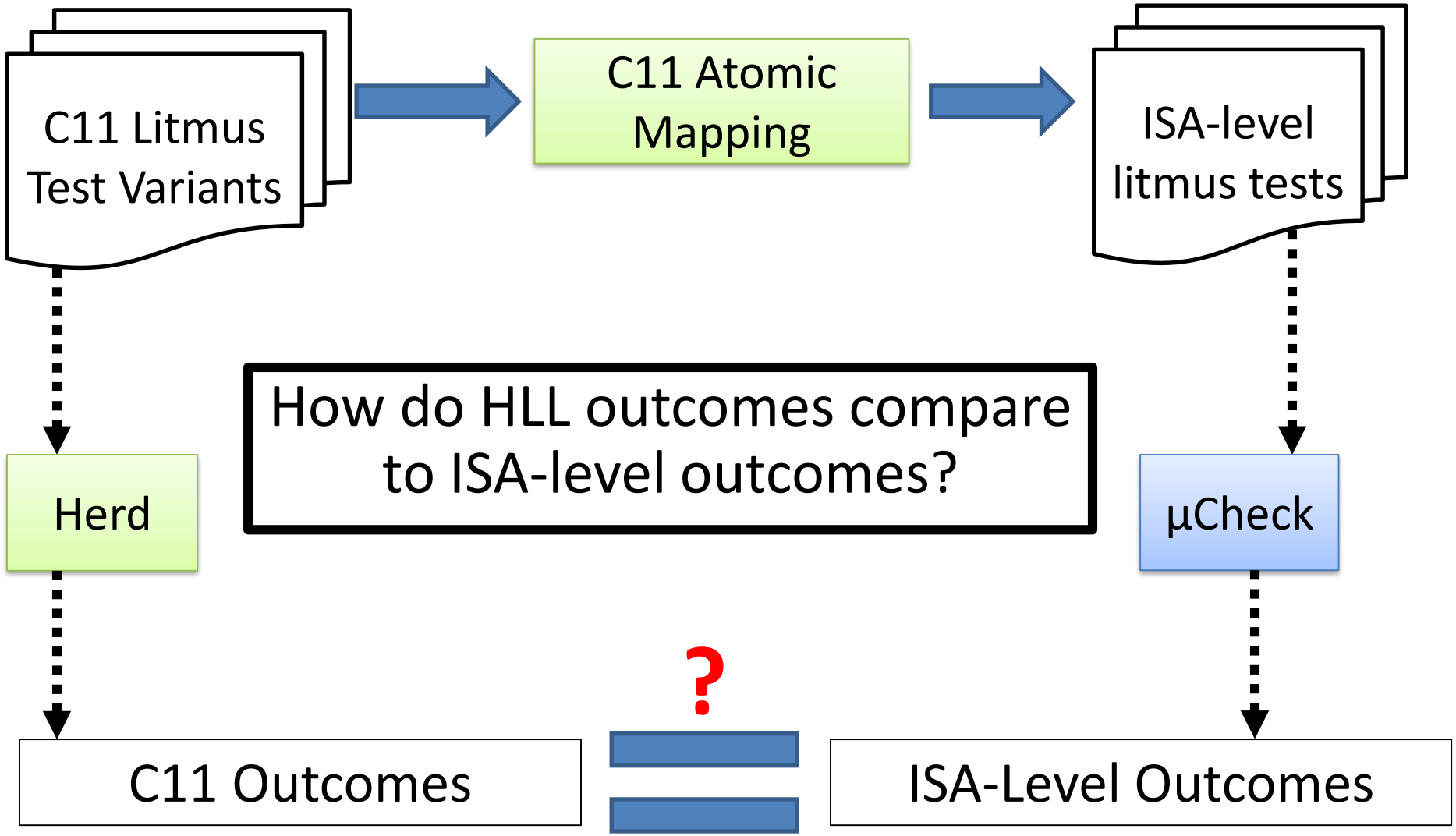


- C/C++11 standards introduced atomic operations
  - Portable, high-performance concurrent code
- Compiler uses **mapping** to translate from atomic ops to assembly instructions

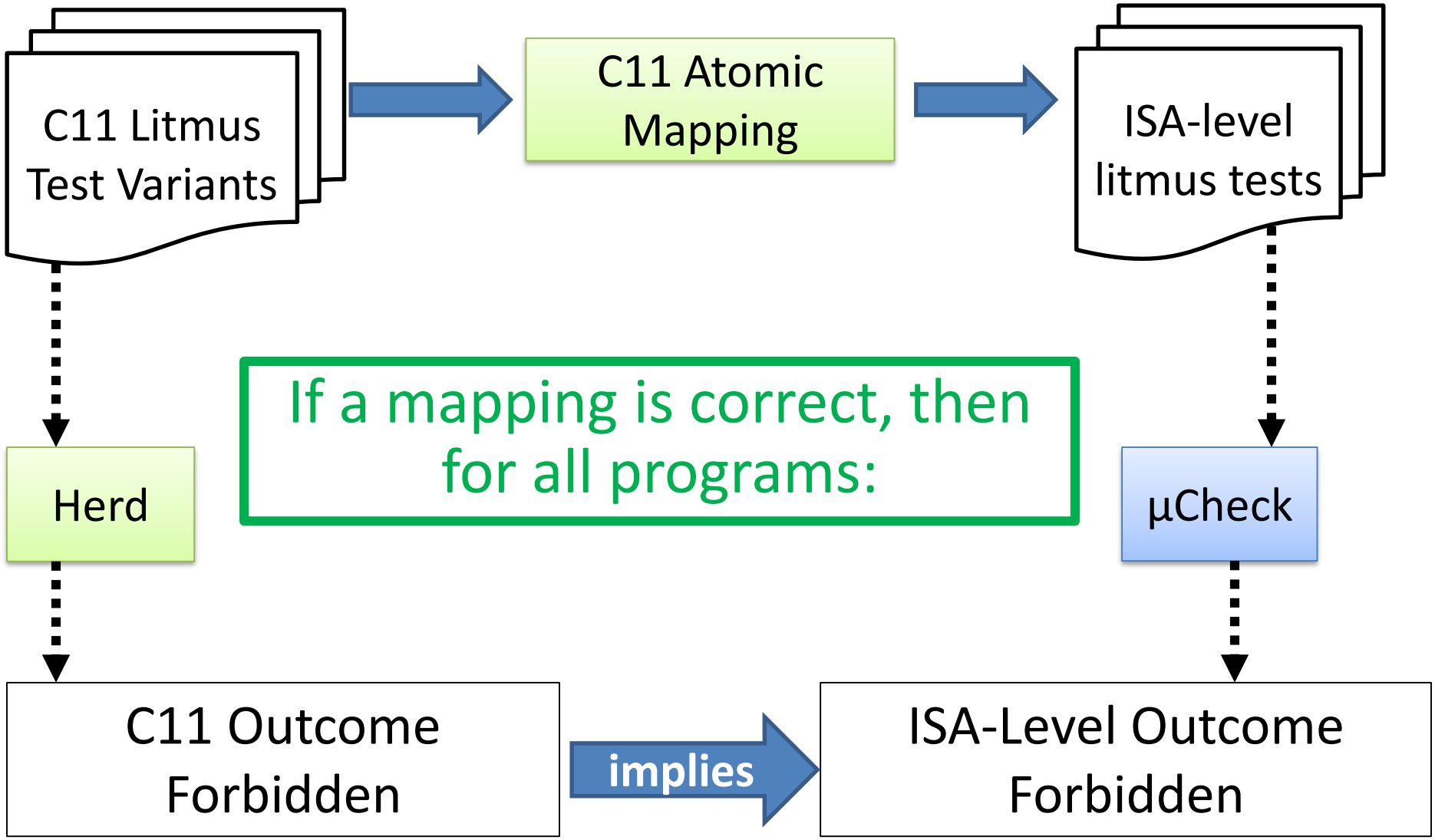
Compilers Must Uphold HLL Guarantees



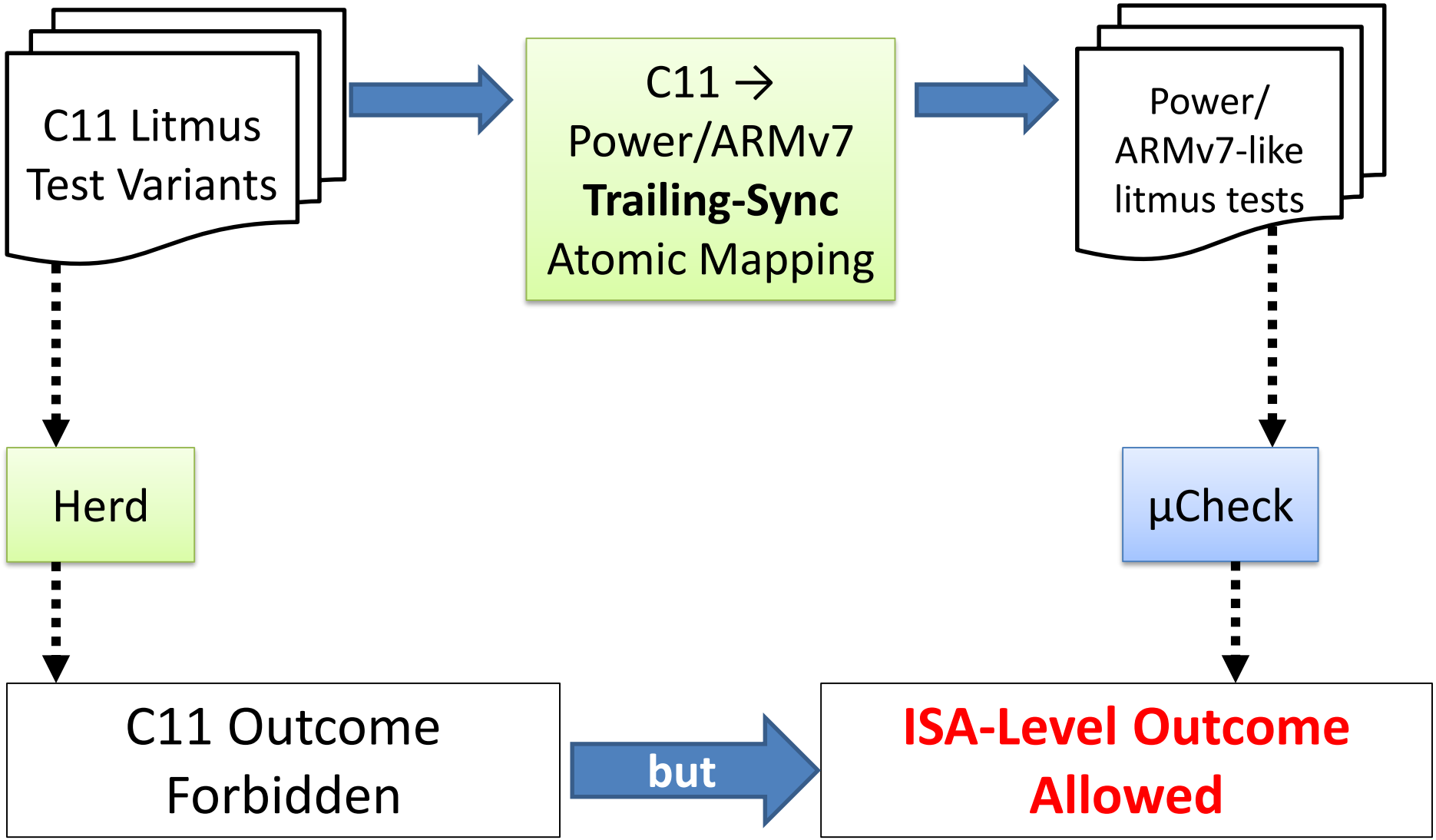
Exploring Mappings with TriCheck



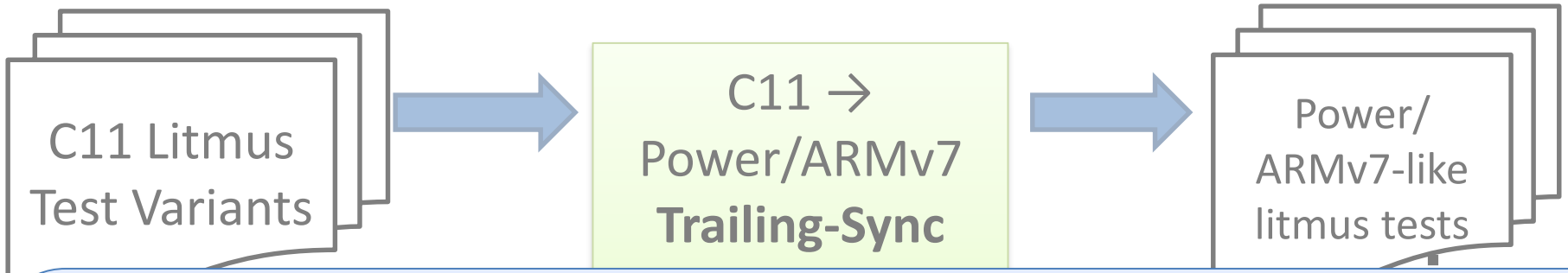
Exploring Mappings with TriCheck



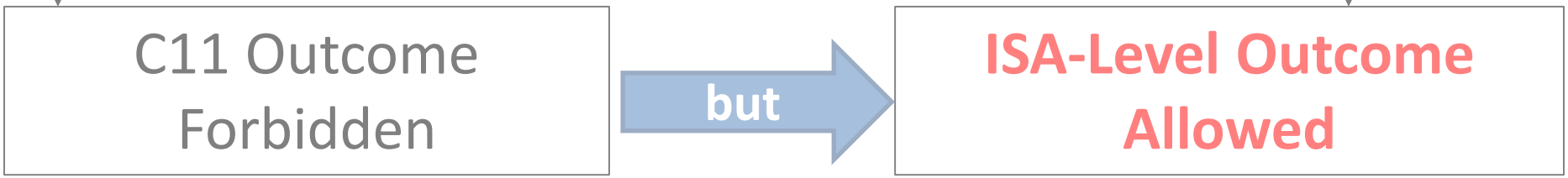
Counterexamples Detected!



Counterexamples Detected!



- Counterexample implies mapping is flawed
- But mapping previously proven correct [Batty et al. POPL 2012]
- Must be an error in the proof!



Outline

- Introduction
- Background on C11 model and mappings
- IRIW Counterexample and Analysis
- Loophole in Proof of Batty et al.
- IBM XL C++ Bugs
- Conclusions and Future Work

C11 Memory Model

- C11 memory model specifies a C11 program's allowed and forbidden outcomes
- Axiomatic model defined in terms of program executions
  - Executions that satisfy C11 axioms are **consistent**
  - Executions that do not satisfy axioms are **forbidden**
  - Outcome only allowed if consistent execution exists
- C11 axioms defined in terms of various relations on an execution

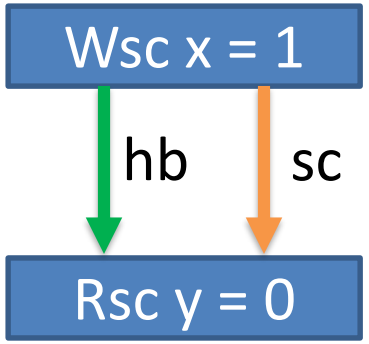
C11 `atomic` operations

- Used to write portable, high-performance concurrent code
- Atomic ops can have different memory orders
  - `seq_cst`, `acquire`, `release`, `relaxed` …
  - Stronger guarantees: easier correctness, lower performance
  - Weaker guarantees: harder correctness, higher performance
- Example (`y` is an atomic variable):

```
y.store(1, memory_order_release);
int b = y.load(memory_order_acquire);
```

Relevant C11 Memory Model Relations

- Happens-before (hb) = $(sb \cup sw)^+$
  - Transitive closure of statement order and synchronization order
- Total order on SC operations (sc)
  - Must be acyclic
  - sc edges must not be in opposite direction to hb edges (sc must be "consistent with" hb)
  - SC read operations cannot read from overwritten writes



Power and ARMv7 Compiler Mappings

- Trailing-sync mapping:
  - [Boehm 2011][Batty et al. POPL 2012]

C/C++ Atomic	Power Mapping	ARMv7 Mapping
Load Acquire	`ld; cmp; bc; isync`	`ldr; teq; beq; isb`
Load Seq Cst	`ld; sync`	`ldr; dmb ish`
Store Release	`lwsync; st`	`dmb ish; str`
Store Seq Cst	`lwsync; st; sync`	`dmb ish; str; dmb ish`

Power lwsync and ARMv7 dmb prior to releases ensure that prior accesses are made visible before the release

Power and ARMv7 Compiler Mappings

- Trailing-sync mapping:
  - [Boehm 2011][Batty et al. POPL 2012]

C/C++ Atomic	Power Mapping	ARMv7 Mapping
Load Acquire	ld; cmp; bc; isync	ldr; teq; beq; isb
Load Seq Cst	ld; sync	ldr; dmb ish
Store Release	lwsync; st	dmb ish; str
Store Seq Cst	lwsync; st; sync	dmb ish; str; dmb ish

Power ctrlisync/sync and ARMv7 ctrlisb/dmb after acquires enforce that subsequent accesses are made visible after the acquire

Use of sync/dmb for SC loads helps enforce the required C11 total order on SC operations

Power and ARMv7 Compiler Mappings

- Trailing-sync mapping:
  - [Boehm 2011][Batty et al. POPL 2012]

C/C++ Atomic	Power Mapping	ARMv7 Mapping
Load Acquire	`ld; cmp; bc; isync`	`ldr; teq; beq; isb`
Load Seq Cst	`ld; sync`	`ldr; dmb ish`
Store Release	`lwsync; st`	`dmb ish; str`
Store Seq Cst	`lwsync; st; sync`	`dmb ish; str; dmb ish`

Power sync and ARMv7 dmb after SC stores ("trailing-sync") prevent reordering with subsequent SC loads

Ostensibly, this ordering can also be enforced by putting fences before SC loads…

Power and ARMv7 Compiler Mappings

- Leading-sync mapping:
  - [McKenney and Silvera 2011]

C/C++ Atomic	Power Mapping	ARMv7 Mapping
Load Acquire	`ld; cmp; bc; isync`	`ldr; teq; beq; isb`
Load Seq Cst	`sync; ld; cmp; bc; isync`	`dmb ish; ldr; teq; beq; isb`
Store Release	`lwsync; st`	`dmb ish; str`
Store Seq Cst	`sync; st`	`dmb ish; str`

Leading-sync mapping places these fences *before* SC loads

Only translations of SC atomics change between the two mappings

Both Mappings are Currently Invalid

- Both supposedly proven correct [Batty et al. POPL 2012]
- We discovered two counterexamples to trailing-sync mappings on Power and ARMv7
  - Isolated the proof loophole that allowed flaw
- Vafeiadis et al. found counterexamples for leading-sync mapping, and have proposed solution

Outline

- Introduction
- Background on C11 model and mappings
- IRIW Counterexample and Analysis
- Loophole in Proof of Batty et al.
- IBM XL C++ Bugs
- Conclusions and Future Work

IRIW Trailing-Sync Counterexample

T0	T1	T2	T3
x.store(1, seq_cst);	y.store(1, seq_cst);	r1 = x.load(acquire);	r3 = y.load(acquire);
		r2 = y.load(seq_cst);	r4 = x .load(seq_cst);

Outcome: r1 = 1, r2 = 0, r3 = 1, r4 = 0

- Variant of IRIW (Independent-Reads-Independent-Writes) litmus test
- IRIW corresponds to two cores observing stores to different addresses in different orders
- At least one of first loads on T2 and T3 is an acquire; all other accesses are SC

IRIW Counterexample Compilation

T0	**T1**	**T2**	**T3**
x.store(1, seq_cst);	y.store(1, seq_cst);	r1 = x.load(acquire);	r3 = y.load(acquire);
		r2 = y.load(seq_cst);	r4 = x .load(seq_cst);

Outcome: r1 = 1, r2 = 0, r3 = 1, r4 = 0

With **trailing sync** mapping, effectively compiles down to

C0	C1	C2	C3
St x = 1	St y = 1	r1 = Ld x	r3 = Ld y
		ctrlisync/ctrlisb	ctrlisync/ctrlisb
		r2 = Ld y	r4 = Ld x

Allowed by Power model and hardware [Alglave et al. TOPLAS 2014]

Allowed by ARMv7 model [Alglave et al. TOPLAS 2014]

IRIW Counterexample Compilation

T0	**T1**	**T2**	**T3**
x.store(1, seq_cst);	y.store(1, seq_cst);	r1 = x.load(acquire);	r3 = y.load(acquire);
		r2 = y.load(seq_cst);	r4 = x .load(seq_cst);

Outcome: r1 = 1, r2 = 0, r3 = 1, r4 = 0

With **trailing sync** mapping, effectively compiles down to

C0	C1	C2	C3
St x = 1	St y = 1	r1 = Ld x	r3 = Ld y
		ctrlisync/ctrlisb	ctrlisync/ctrlisb
		r2 = Ld y	r4 = Ld x

ctrlisync/ctrlisb are not strong enough to forbid outcome

Allowed by Power model and hardware [Alglave et al. TOPLAS 2014]

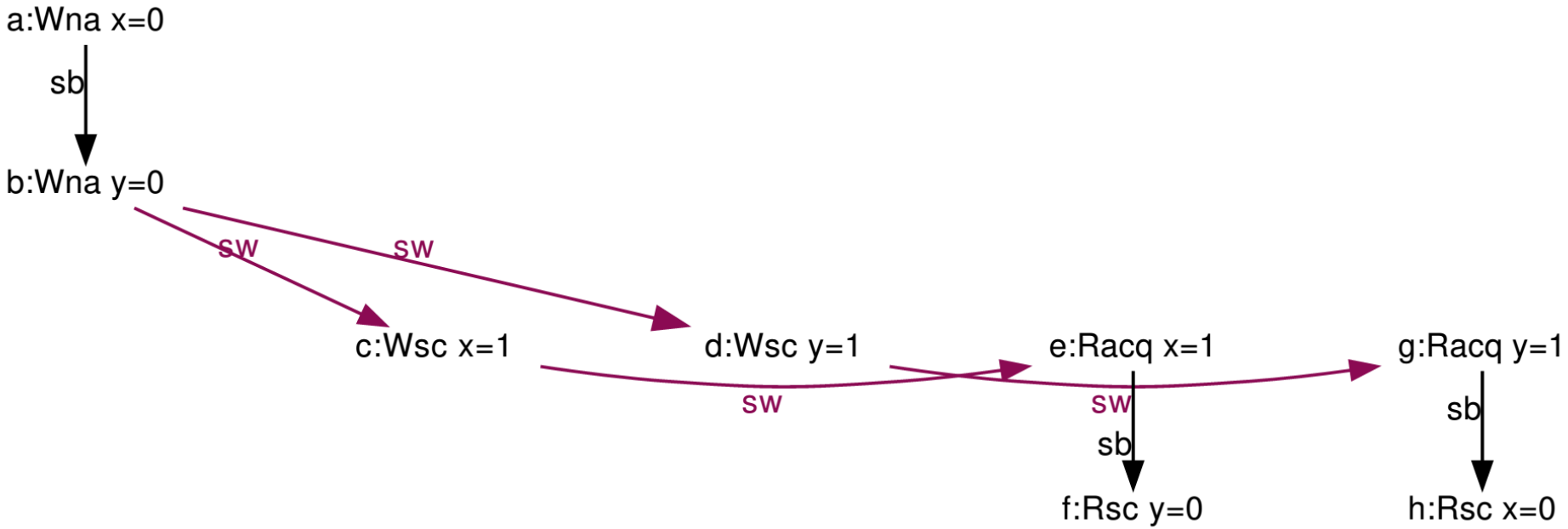
Allowed by ARMv7 model [Alglave et al. TOPLAS 2014]

IRIW Trailing-Sync Counterexample

T0	T1	T2	T3
x.store(1, seq_cst);	y.store(1, seq_cst);	r1 = x.load(acquire);	r3 = y.load(acquire);
		r2 = y.load(seq_cst);	r4 = x .load(seq_cst);

Outcome: r1 = 1, r2 = 0, r3 = 1, r4 = 0

Happens-before edges from `c` → `f` and from `d` → `h` by transitivity

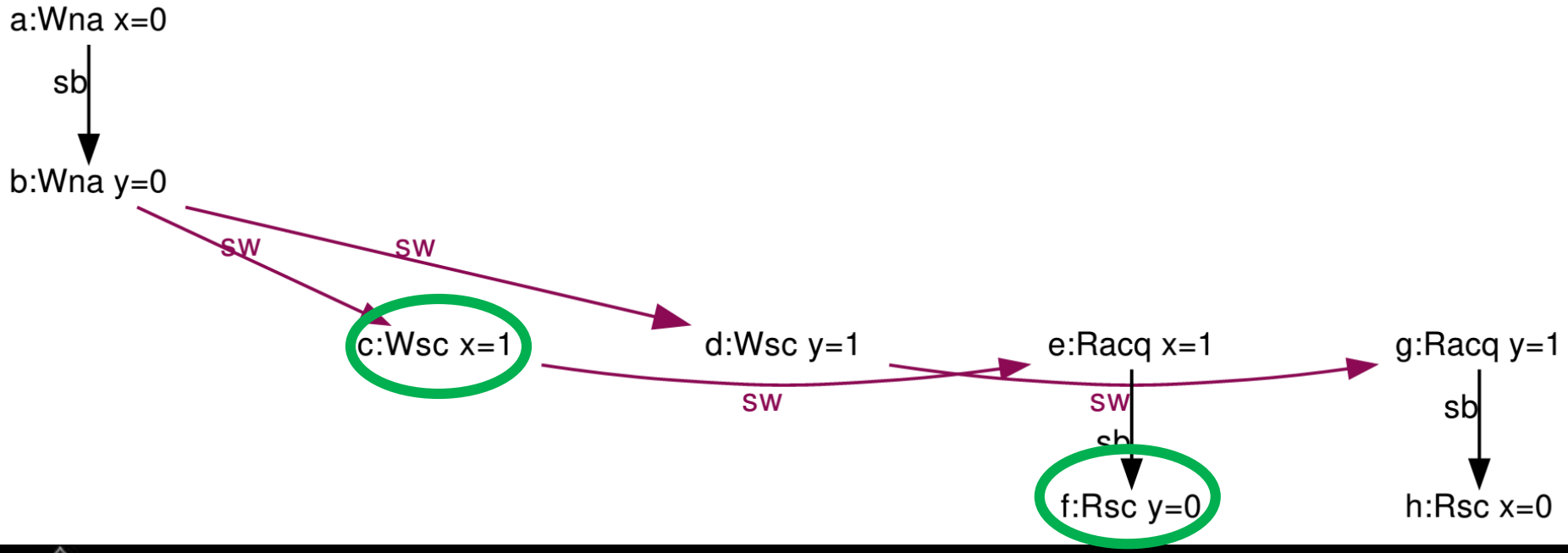


IRIW Trailing-Sync Counterexample

T0	T1	T2	T3
x.store(1, seq_cst);	y.store(1, seq_cst);	r1 = x.load(acquire);	r3 = y.load(acquire);
		r2 = y.load(seq_cst);	r4 = x .load(seq_cst);

Outcome: r1 = 1, r2 = 0, r3 = 1, r4 = 0

Happens-before edges from `c` → `f` and from `d` → `h` by transitivity

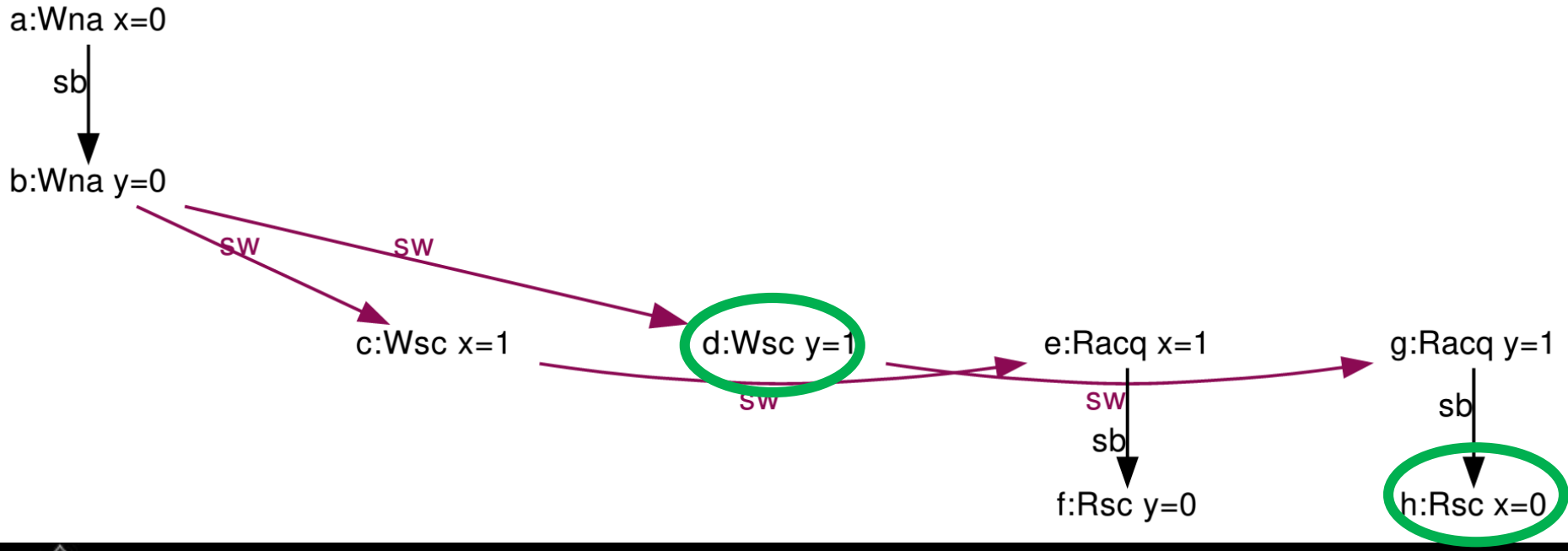


IRIW Trailing-Sync Counterexample

T0	T1	T2	T3
x.store(1, seq_cst);	y.store(1, seq_cst);	r1 = x.load(acquire);	r3 = y.load(acquire);
		r2 = y.load(seq_cst);	r4 = x .load(seq_cst);

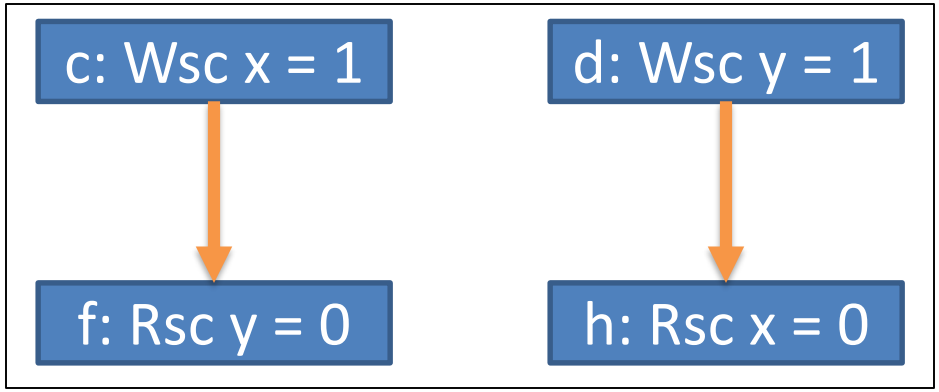
Outcome: r1 = 1, r2 = 0, r3 = 1, r4 = 0

Happens-before edges from `c` → `f` and from `d` → `h` by transitivity



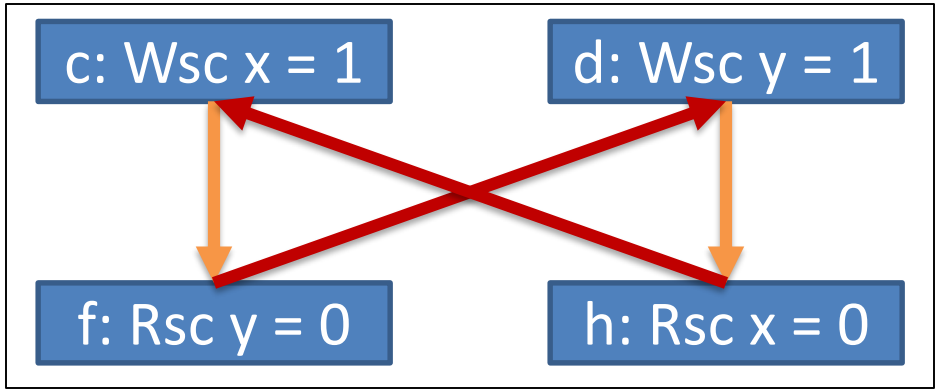
IRIW Trailing-Sync Counterexample

- SC order must contain edges from `c` → `f` and from `d` → `h` to match direction of hb edges
- Shown below as `sc_hb` edges



IRIW Trailing-Sync Counterexample

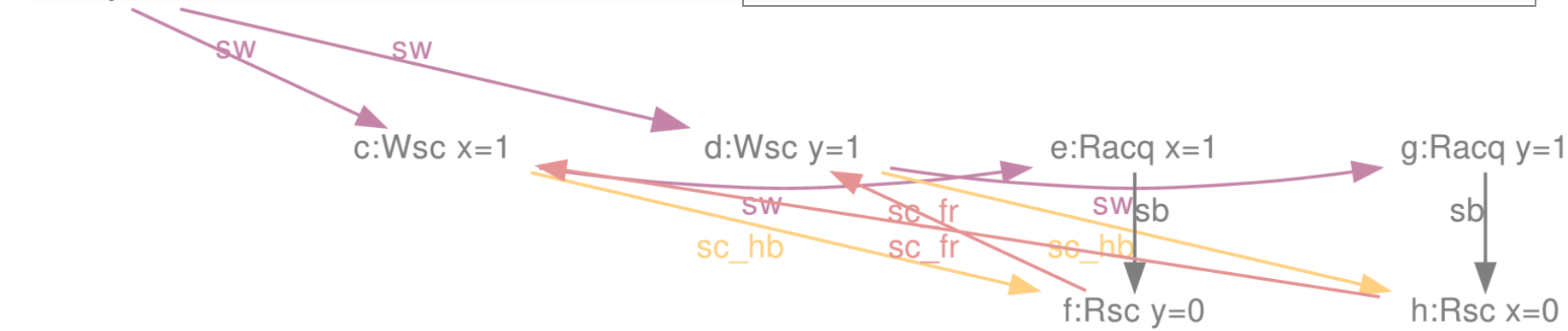
- SC reads `f` and `h` must read from non-SC writes `b` and `a` before they are overwritten
- The SC order must contain `f`→`d` and `h`→`c` to satisfy this condition



IRIW Trailing-Sync Counterexample

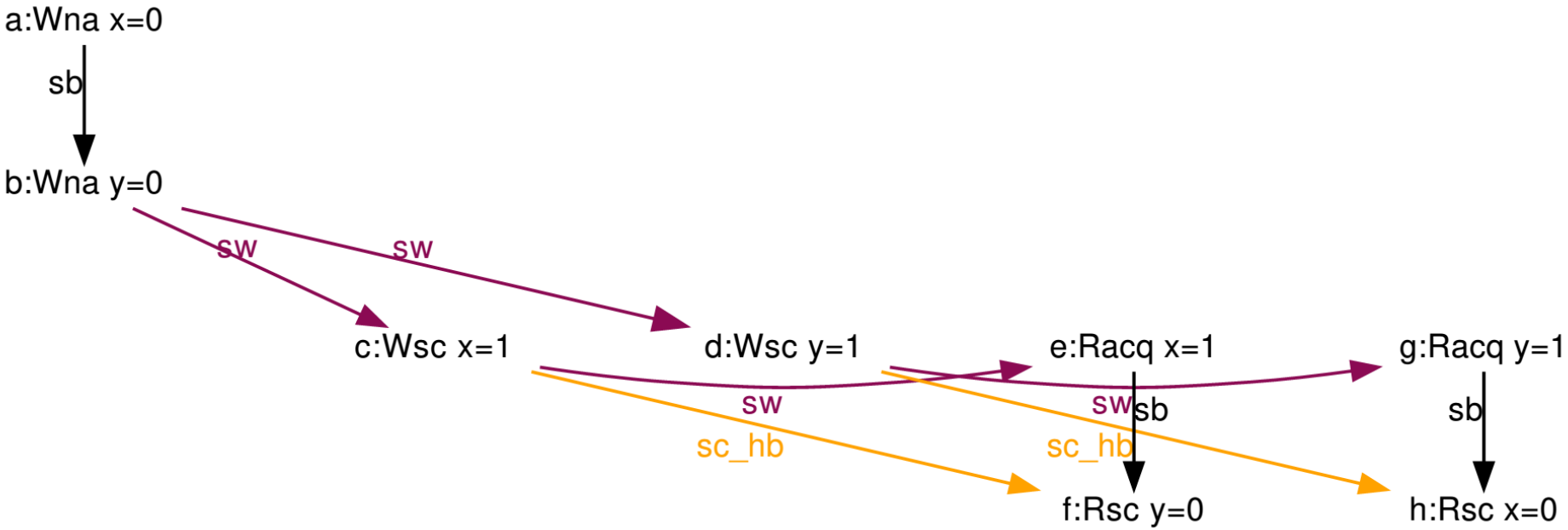
- SC reads `f` and `h` must read from non-SC writes `b` and `a` before they are overwritten

- Cycle in the SC order
- Outcome is **forbidden** as there is no corresponding consistent execution
- But compiled code **allows** the behaviour!



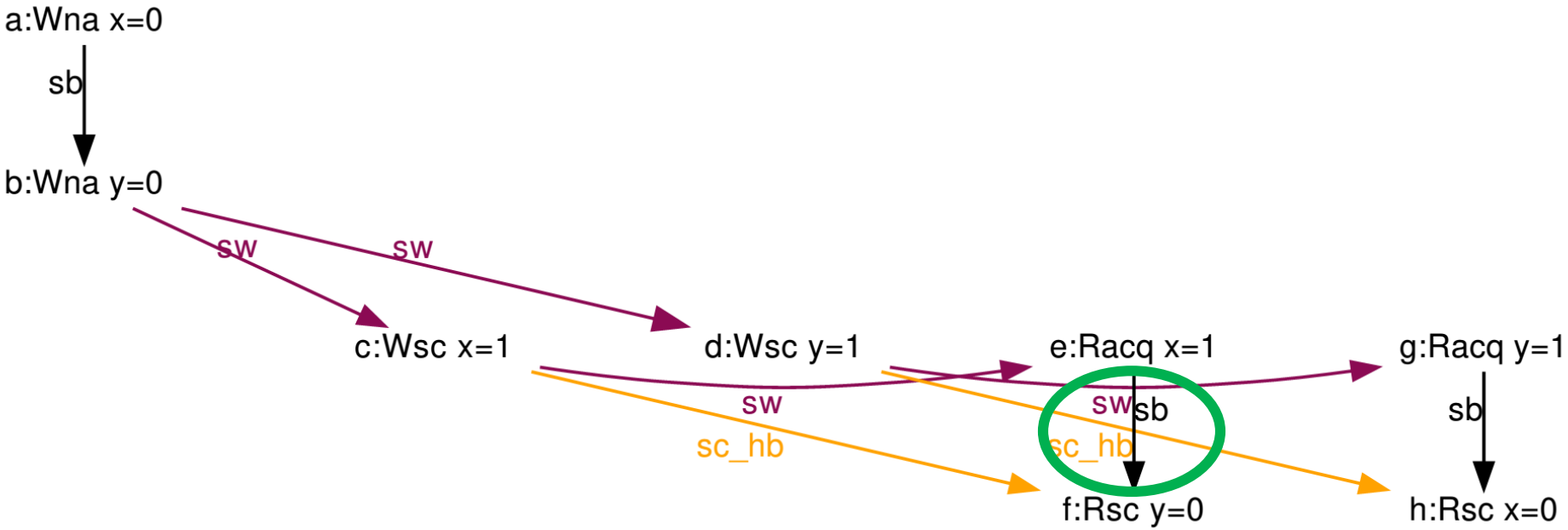
What went wrong?

- SC axioms required SC order to contain edges from `c` → `f` and from `d` → `h` to match direction of hb edges
- This requires a `sync/dmb ish` between `e` and `f` as well as between `g` and `h` on Power and ARMv7
- These fences are **NOT** provided by trailing-sync mapping



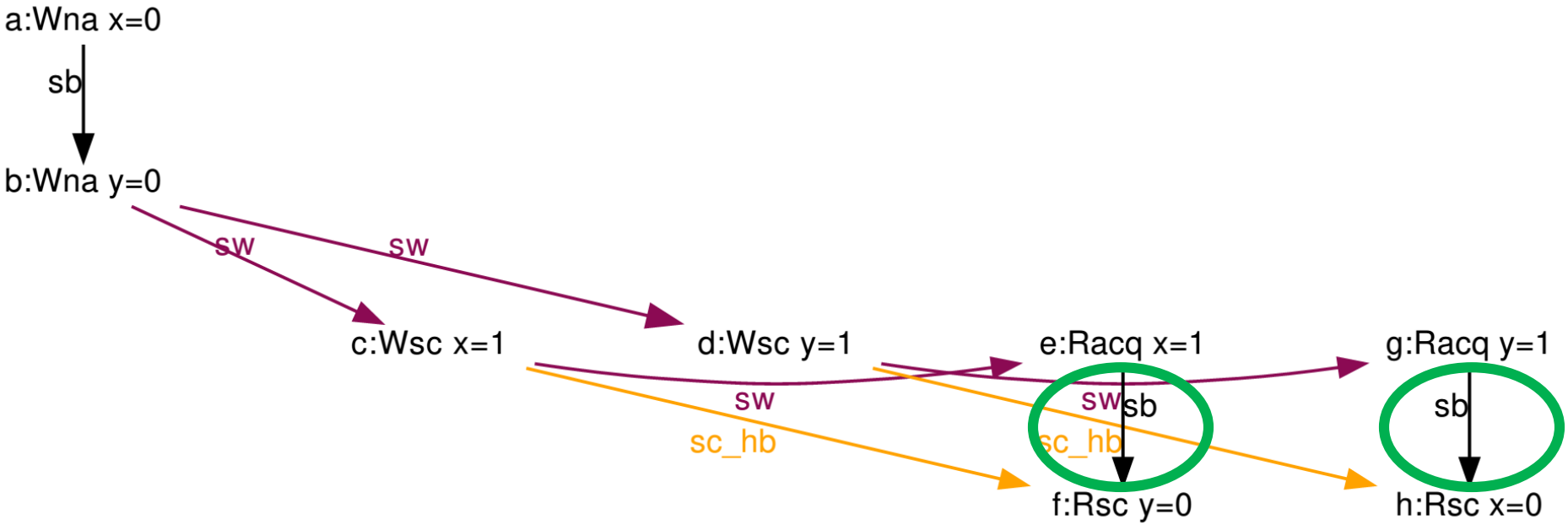
What went wrong?

- SC axioms required SC order to contain edges from `c` → `f` and from `d` → `h` to match direction of hb edges
- This requires a `sync/dmb ish` between `e` and `f` as well as between `g` and `h` on Power and ARMv7
- These fences are **NOT** provided by trailing-sync mapping



What went wrong?

- SC axioms required SC order to contain edges from `c` → `f` and from `d` → `h` to match direction of hb edges
- This requires a `sync/dmb ish` between `e` and `f` as well as between `g` and `h` on Power and ARMv7
- These fences are **NOT** provided by trailing-sync mapping



Outline

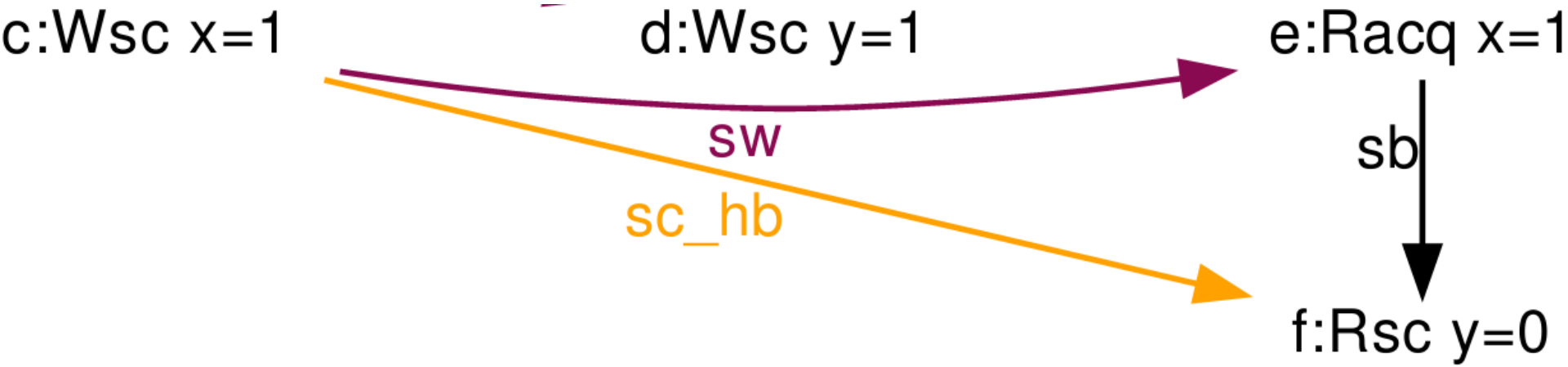
- Introduction
- Background on C11 model and mappings
- IRIW Counterexample and Analysis
- Loophole in Proof of Batty et al.
- IBM XL C++ Bugs
- Conclusion

Loophole in Batty et al. proof [POPL 2012]

- Lemma in proof states that SC order for a given Power trace is an arbitrary linearization of
$$(po_t^{sc} \cup co_t^{sc} \cup fr_t^{sc} \cup erf_t^{sc})^*$$
- This is the transitive closure of program order and coherence edges **directly** between SC accesses
- Proof clause checking C11 axiom that sc and hb edges match direction states that having SC order be arbitrary linearization of above relation is sufficient

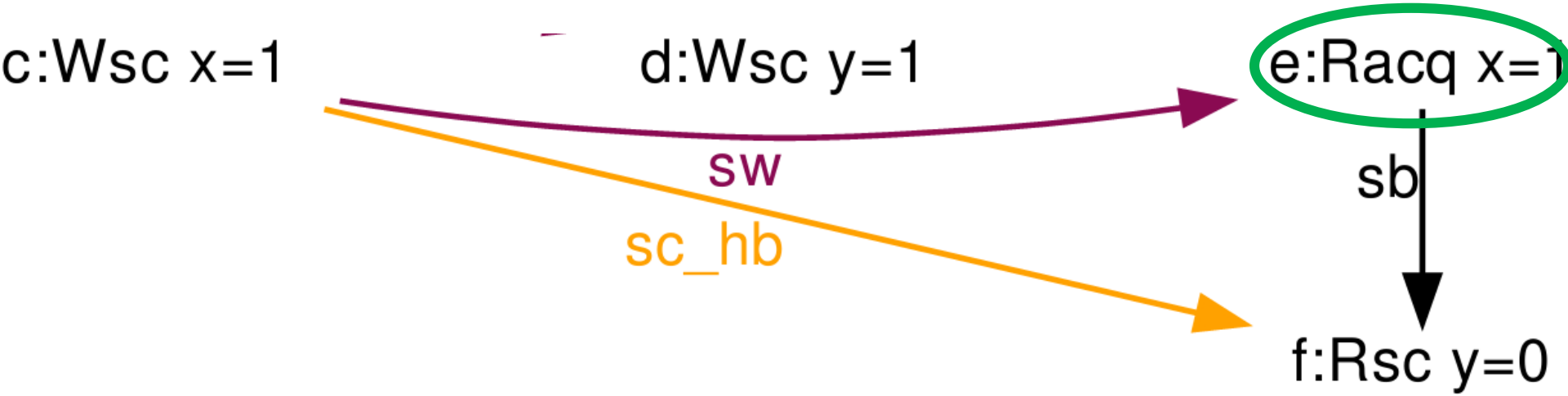
Loophole in Batty et al. proof [POPL 2012]

- This claim is **false** in certain scenarios
- hb edges can arise between SC accesses through the transitive composition of edges to and from a non-SC **intermediate** access
- Occurs in IRIW counterexample:



Loophole in Batty et al. proof [POPL 2012]

- This claim is **false** in certain scenarios
- hb edges can arise between SC accesses through the transitive composition of edges to and from a non-SC **intermediate** access
- Occurs in IRIW counterexample:



Loophole in Batty et al. proof [POPL 2012]

- SC order must be in same direction as these hb edges, but an arbitrary linearization of $(po_t^{sc} \cup co_t^{sc} \cup fr_t^{sc} \cup erf_t^{sc})^*$ may not satisfy this condition
- Result: Proof does not guarantee that sc and hb edges match direction between two accesses, and is incorrect
  - confirmed by Batty et al.

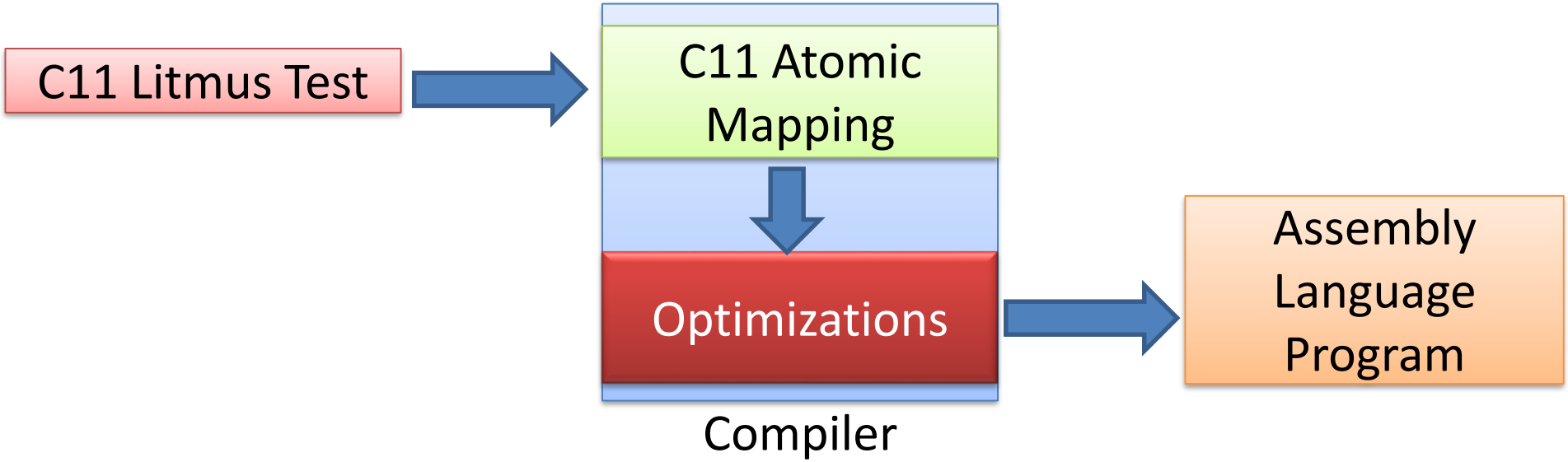
Current Compiler and Architecture State

- Neither GCC nor Clang implement exact flawed trailing-sync mapping
  - Use leading-sync mapping for Power
  - Use trailing-sync for ARMv7, but with stronger acquire mapping (`ld; dmb ish` or stronger)
  - Sufficient to disallow both our counterexamples
- Both counterexample behaviours observed on Power hardware [Alglave et al. TOPLAS 2014]
- ARMv7 model [Alglave et al. TOPLAS 2014] allows counterexample behaviours, but not observed on ARMv7 hardware

Outline

- Introduction
- Background on C11 model and mappings
- IRIW Counterexample and Analysis
- Loophole in Proof of Batty et al.
- IBM XL C++ Bugs
- Conclusion

What about optimizations?



- Even if mapping is correct, optimizations cannot introduce new outcomes
- Recent work on src-to-src opts and LLVM IR verification
  - [Vafeiadis et al. POPL 2015]
  - [Chakraborty and Vafeiadis CGO 2016]
- What about commercial compilers?

XL C++ Bugs Overview

- Visited IBM Yorktown Heights to check if XL C++ (v13.1.4) was vulnerable to trailing-sync counterexample
- XL C++ mapping close to leading-sync
- Often correct at lower optimization levels, but increasing optimizations to –O3 and –O4 generated incorrect code for multiple tests
- Bugs have since been fixed by compiler team
  - Caused by issues in code generator
  - Fixes in v13.1.5

Bug #1: Loss of SC Store Release Semantics

"Message-passing" litmus test (mp), relaxed store of x, all other accesses SC

T0	T1
x.store(1, relaxed);	r1 = y.load(seq_cst);
y.store(1, seq_cst);	r2 = x.load(seq_cst);

Outcome: r1 = 1, r2 = 0 (Forbidden by C++)

XL C++ with –O3 compiles to:

C0	C1
St x = 1	sync
sync	r1 = Ld y
St y = 1	ctrlisync (twice)
	sync
	r2 = Ld x
	ctrlisync (twice)

Forbidden

XL C++ with –O4 compiles to:

C0	C1
St x = 1	ctrlisync
ctrlisync	r1 = Ld y
St y = 1	sync
sync	ctrlisync
	r2 = Ld x
	sync

Allowed

Used **litmus** utility to exercise outcome of incorrect code

Bug #1: Loss of SC Store Release Semantics

"Message-passing" litmus test (mp), relaxed store of x, all other accesses SC

T0	T1
x.store(1, relaxed);	r1 = y.load(seq_cst);
y.store(1, seq_cst);	r2 = x.load(seq_cst);

Outcome: r1 = 1, r2 = 0 (Forbidden by C++)

Bug: Ctrlisync is not strong enough to ensure stores are observed in order

XL C++ with –O3 compiles to:

C0	C1
St x = 1	sync
sync	r1 = Ld y
St y = 1	ctrlisync (twice)
	sync
	r2 = Ld x
	ctrlisync (twice)

Forbidden

XL C++ with –O4 compiles to:

C0	C1
St x = 1	ctrlisync
ctrlisync	r1 = Ld y
St y = 1	sync
sync	ctrlisync
	r2 = Ld x
	sync

Allowed

Used **litmus** utility to exercise outcome of incorrect code

Bug #2: Incorrect Impl. of Releases

"Message-passing" litmus test (mp), with release-acquire atomics, relaxed store of x

T0	**T1**
x.store(1, relaxed);	r1 = y.load(acquire);
y.store(1, release);	r2 = x.load(acquire);

Outcome: r1 = 1, r2 = 0 (Forbidden by C++)

XL C++ with –O3 compiles to:

C0	**C1**
St x = 1	ctrlisync
St y = 1	r1 = Ld y
	ctrlisync
	r2 = Ld x

Allowed

Used **litmus** utility to exercise outcome of incorrect code

Bug #2: Incorrect Impl. of Releases

"Message-passing" litmus test (mp), with release-acquire atomics, relaxed store of x

T0	T1
x.store(1, relaxed);	r1 = y.load(acquire);
y.store(1, release);	r2 = x.load(acquire);

Outcome: r1 = 1, r2 = 0 (Forbidden by C++)

XL C++ with –O3 compiles to:

C0	C1
St x = 1	ctrlisync
St y = 1	r1 = Ld y
	ctrlisync
	r2 = Ld x

Allowed

Bug: No ordering enforcement between stores

Used **litmus** utility to exercise outcome of incorrect code

Bug #3: Reordering SC Loads and `syncs`

IRIW litmus test with two acquire loads, all other accesses SC

T0	T1	T2	T3
x.store(1, seq_cst);	y.store(1, seq_cst);	r1 = x.load(acquire);	r3 = y.load(acquire);
		r2 = y.load(seq_cst);	r4 = x .load(seq_cst);

Outcome: r1 = 1, r2 = 0, r3 = 1, r4 = 0 (Forbidden by C++)

XL C++ with –O3 compiles to:

C0	C1	C2	C3
St x = 1	St y = 1	ctrlisync	ctrlisync
		r1 = Ld x	r3 = Ld y
		sync	sync
		r2 = Ld y	r4 = Ld x
		ctrlisync	ctrlisync

Forbidden

XL C++ with –O4 compiles to:

C0	C1	C2	C3
ctrlisync	ctrlisync	ctrlisync	ctrlisync
St x = 1	St y = 1	r1 = Ld x	r3 = Ld y
		ctrlisync	ctrlisync
		r2 = Ld y	r4 = Ld x
		sync	sync

Allowed

Bug #3: Reordering SC Loads and syncs

IRIW litmus test with two acquire loads, all other accesses SC

T0	T1	T2	T3
x.store(1, seq_cst);	y.store(1, seq_cst);	r1 = x.load(acquire);	r3 = y.load(acquire);
		r2 = y.load(seq_cst);	r4 = x .load(seq_cst);

Outcome: r1 = 1, r2 = 0, r3 = 1, r4 = 0 (Forbidden by C++)

Bug: Ctrlisync is not enough to enforce required orderings

XL C++ with –O3 compiles to:

C0	C1	C2	C3
St x = 1	St y = 1	ctrlisync	ctrlisync
		r1 = Ld x	r3 = Ld y
		sync	sync
		r2 = Ld y	r4 = Ld x
		ctrlisync	ctrlisync

Forbidden

XL C++ with –O4 compiles to:

C0	C1	C2	C3
ctrlisync	ctrlisync	ctrlisync	ctrlisync
St x = 1	St y = 1	r1 = Ld x	r3 = Ld y
		ctrlisync	ctrlisync
		r2 = Ld y	r4 = Ld x
		sync	sync

Allowed

Future Work

- XL C++ bugs show that it is particularly hard to maintain C11 orderings across optimizations
- Need a top-to-bottom verification flow from HLL to assembly code, incorporating compiler optimizations
  - Avenue for future work

Conclusions

- TriCheck provides rapid exploration of different compiler mappings for architectures across C11 litmus test variants
- Using TriCheck, discovered two trailing-sync counterexamples for Power and ARMv7
  - Also discovered loophole in proof of mappings
  - Either C11 model or mappings must change to enable correct compilation
- Experiments with IBM XL C++ revealed bugs (since fixed) in their C11 implementation

C11 Compiler Mappings: Exploration, Verification, and Counterexamples

Yatin Manerkar

Princeton University

Tools and papers available at
http://check.cs.princeton.edu