Alan Stern, Rowland Institute at Harvard
Paul E. McKenney, Meta Platforms Kernel Team
Michael Wong, Codeplay
Maged Michael
Kangrejos, Copenhagen, Denmark, September 8, 2024

Lifetime-End Pointer Zap & How to Avoid OOTA Without Really Trying

Overview

This is just an overview, not a replacement for the papers themselves

- P2414R4 “Pointer lifetime-end zap proposed solutions”
  - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2414r4.pdf
- P3347R0 Invalid/Prospective Pointer Operations
  - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3347r0.pdf
  - Based on Davis Herring’s P2434R1 “Nondeterministic pointer provenance”
    - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2434r1.html
- P3064R2 “How to Avoid OOTA Without Really Trying”
  - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3064r2.pdf

Overview

- Lifetime-end pointer zap
- Out-of-thin-air (OOTA) cycles
- Where are we on OOTA?
- Leverage restrictions:
  - Real computer systems
  - Speculate properly or not at all
  - Existing restrictions for volatile atomics
  - No invention or repurposing of atomic loads
  - Tooling looks at object code
- Future directions

Lifetime-End Pointer Zap

Problem Restatement (C11, 1/2)

```
struct node_t* _Atomic top;

void list_push(value_t v)
{
  struct node_t *newnode = (struct node_t *) malloc(sizeof(*newnode));
  Struct node_t *next = atomic_load(&top);

  set_value(newnode, v);
  do {
    set_next(newnode, next);
    // newnode’s next pointer may have become invalid
  } while (!atomic_compare_exchange_weak(&top, &next, newnode));
}
```

Problem Restatement (C11, 2/2)

```
void list_pop_all()
{
  struct node_t *p = atomic_exchange(&top, NULL);

  while (p) {
    struct node_t *next = p->next;

    foo(p);
    free(p);
    p = next;
  }
}
```

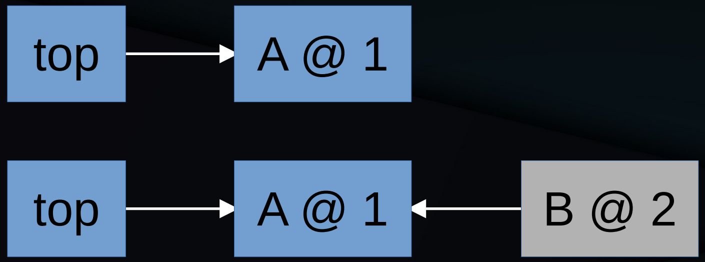
Problem Illustration (C11)

Freelist

Initial State

Problem Illustration (C11)

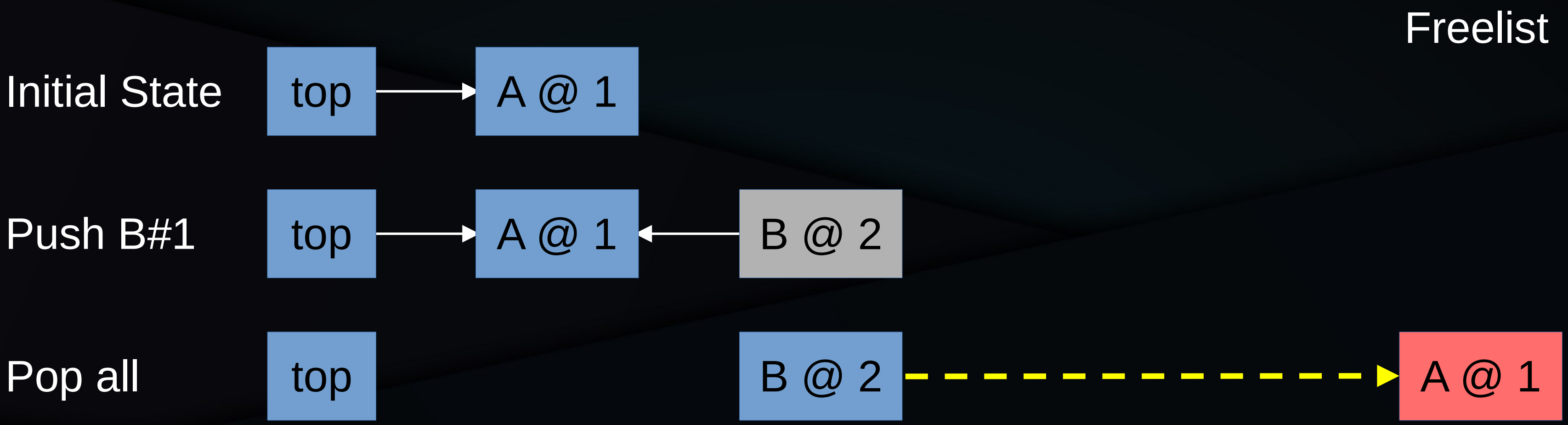
Freelist



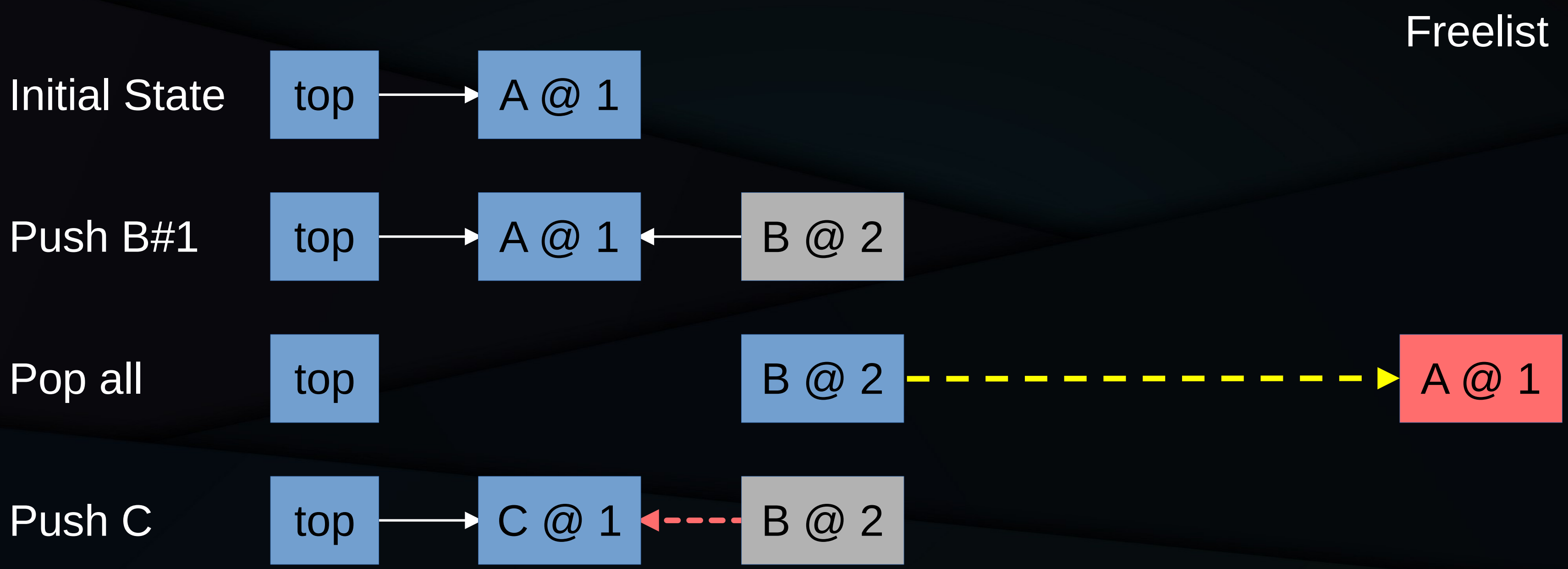
Initial State

Push B#1

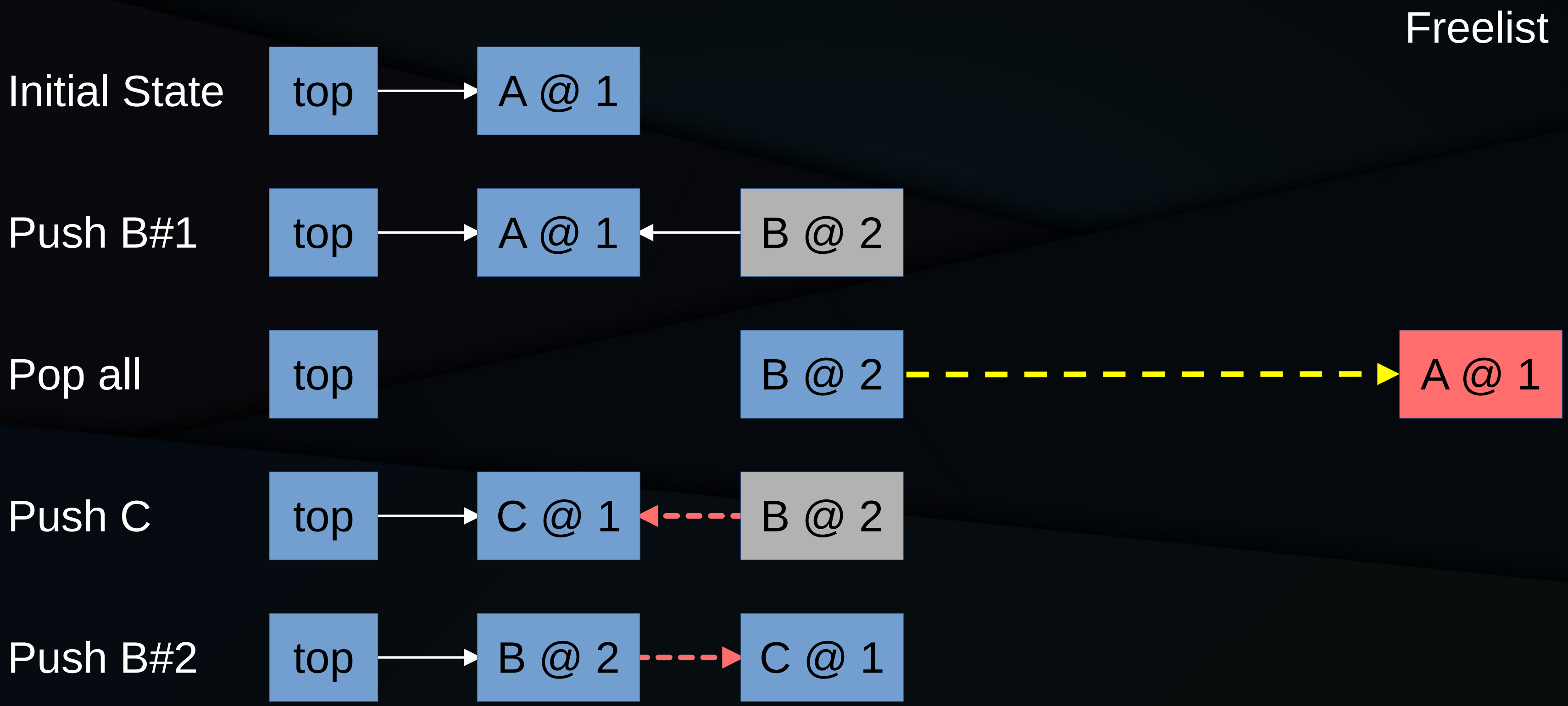
Problem Illustration (C11)



Problem Illustration (C11)



Problem Illustration (C11)



Freelist

Initial State: top → A @ 1

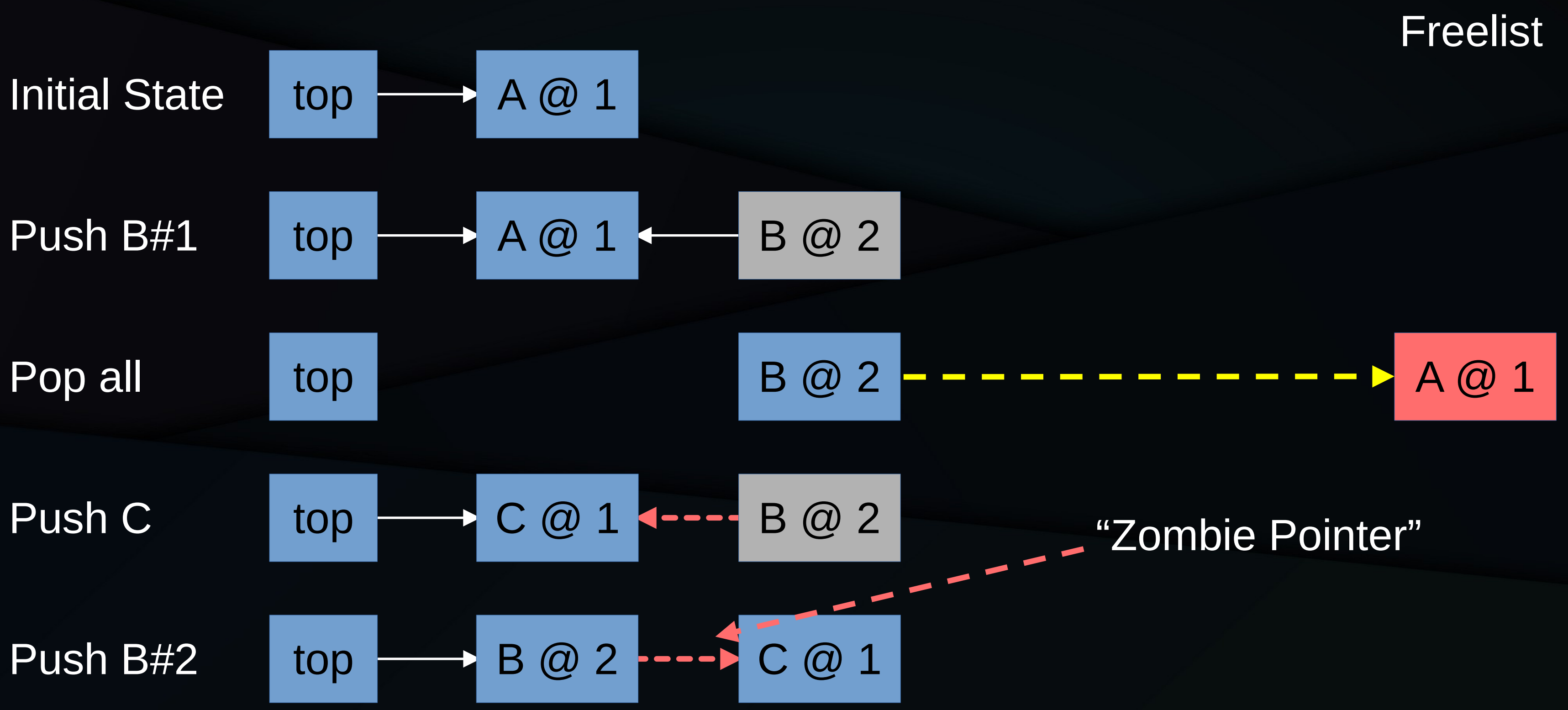
Push B#1: top → A @ 1 ← B @ 2

Pop all: top; B @ 2 → A @ 1

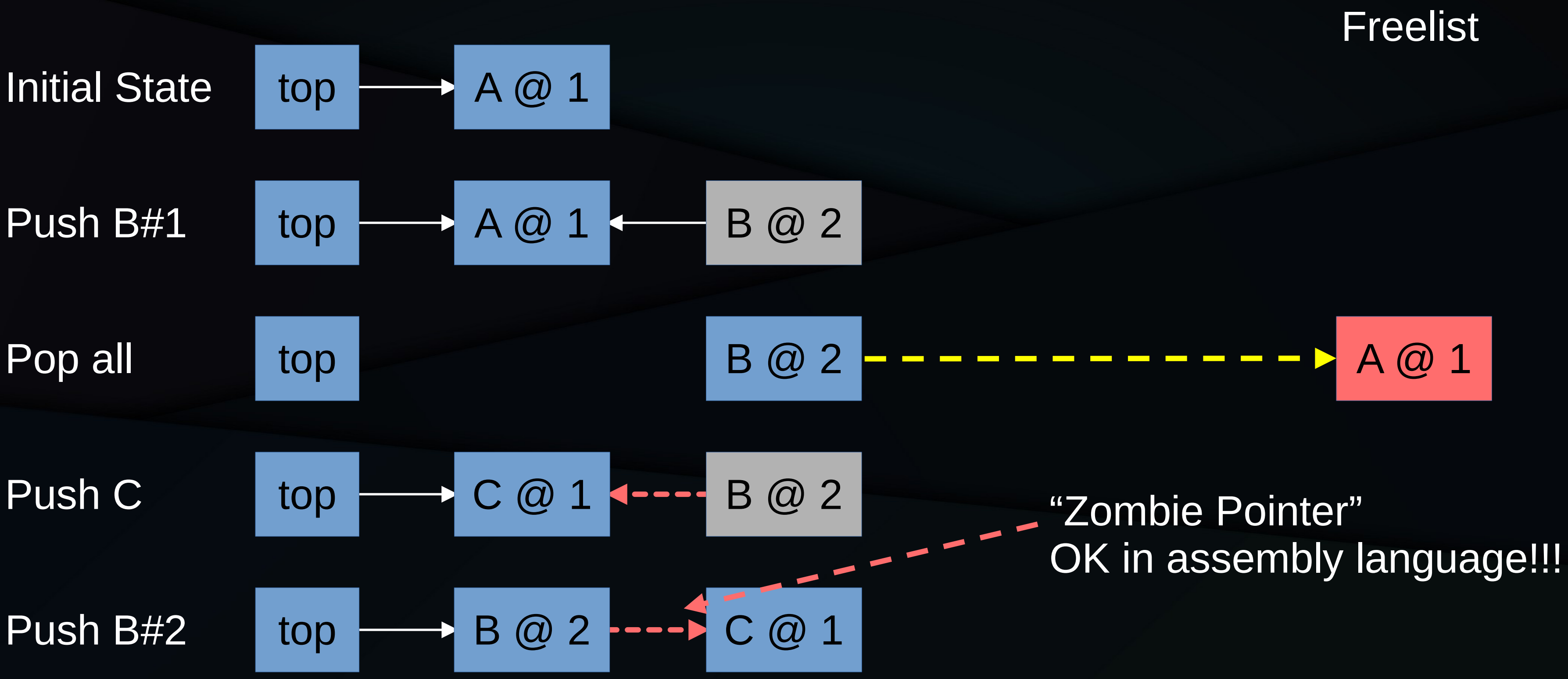
Push C: top → C @ 1 ← B @ 2

Push B#2: top → B @ 2 → C @ 1

Problem Illustration (C11)

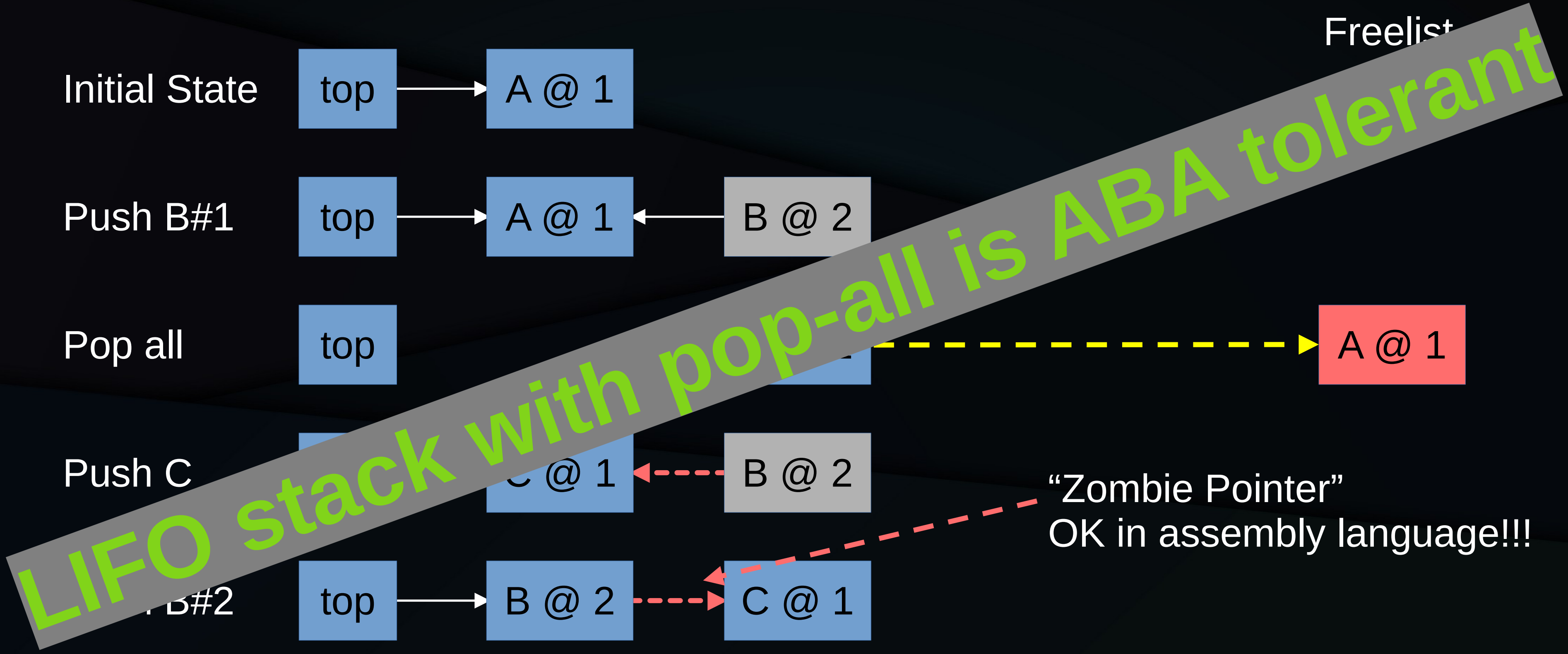


Problem Illustration (C11)



“Zombie Pointer”
OK in assembly language!!!

Problem Illustration (C11)



Freelist

Initial State
top
A @ 1

Push B#1
top
A @ 1
B @ 2

Pop all
top
A @ 1

Push C
C @ 1
B @ 2

B#2
top
B @ 2
C @ 1

“Zombie Pointer”
OK in assembly language!!!

LIFO stack with pop-all is ABA tolerant

This is Real and Isn’t Going Away

- LIFO stack described by Treiber in 1986
  - Written in IBM BAL, avoiding issues with compilers
- LIFO stack alluded to in early 1970s
- LIFO stack implemented in Rust library
  - Though with `pop()`, not `pop_all()`.
- Used in heavily production in many languages

OK, OK, What is New Since 2023???

C and C++: Pointer Provenance

- Pointers contain bits and also “provenance”
    - Compiler may assume that pointers from two different calls to the allocator are unequal
- Provenance may be erased
    - Conversion to integer, I/O, optimization frontiers
- Davis Herring C++ proposal (P2434R1) provides “angelic provenance”

C++: Angelic Provenance

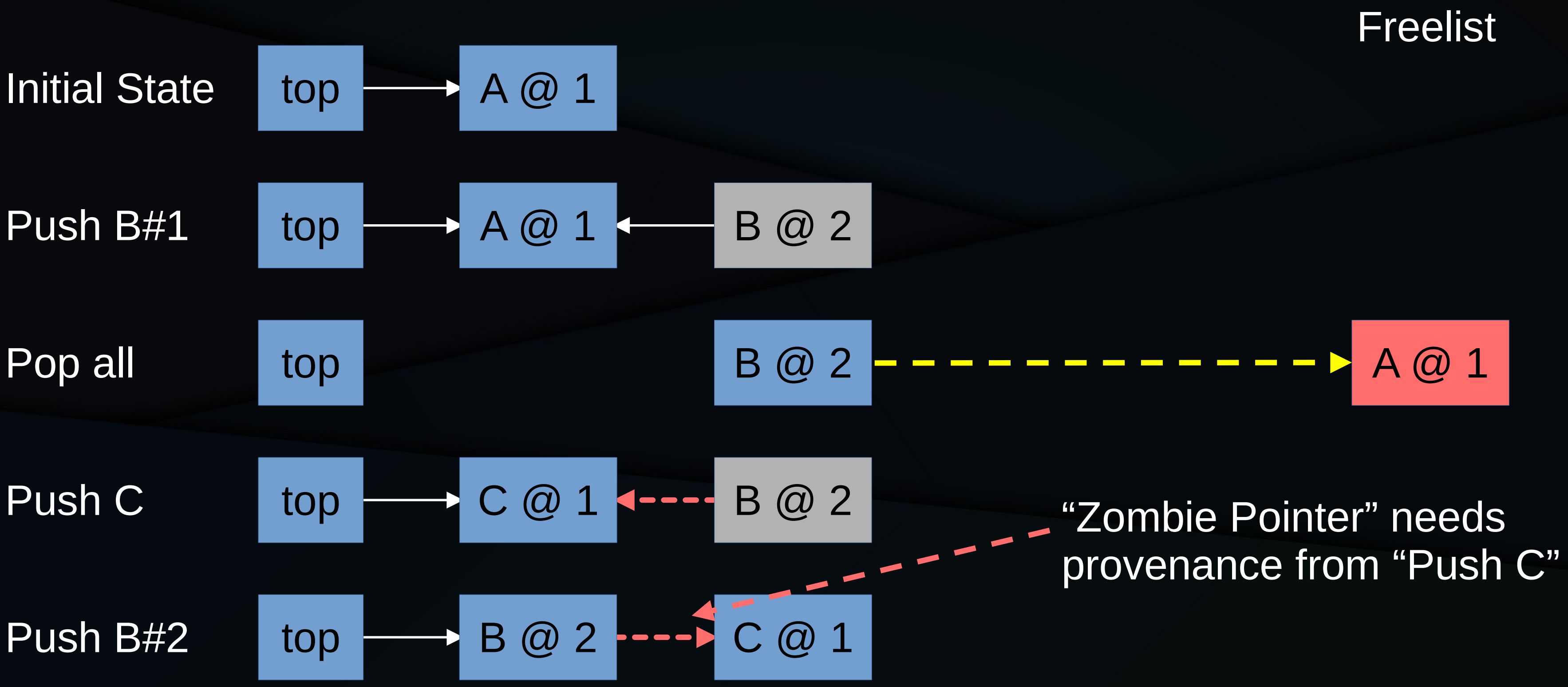
- Davis Herring P2434R1 (“Nondeterministic pointer provenance”) restricts provenance restoration
  - Conversion from integer, I/O, optimization frontiers
  - Pointer provenance remains “provisional” until comparison or dereference
  - At which point, the compiler must choose provenance (if any) that allows the program to be well-formed

C++: Angelic Provenance

- Davis Herring P2434R1 (“Nondeterministic pointer provenance”) restricts provenance restoratio[illegible]
  - Conversion from integer, I/O, opt[illegible] frontiers
  - Pointer provenance rem[illegible] [illegible]ovisional” until comparison or d[illegible]ce
  - At whic[illegible] the compiler must choose provenance (if a[illegible]hat allows the program to be well-formed

Not enough for LIFO stack...

Problem Illustration (C11)



Freelist

Initial State
top
A @ 1

Push B#1
top
A @ 1
B @ 2

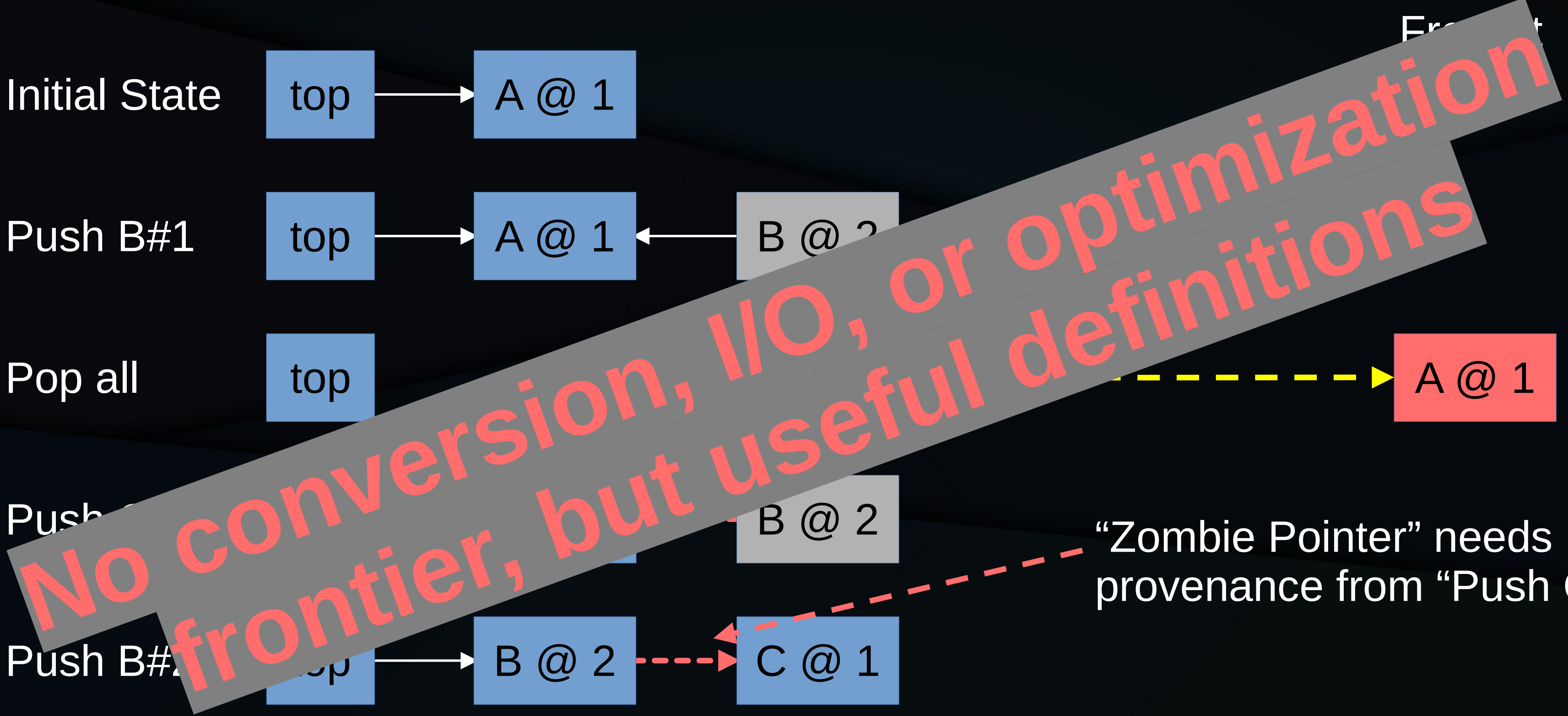
Pop all
top
B @ 2
A @ 1

Push C
top
C @ 1
B @ 2

“Zombie Pointer” needs provenance from “Push C”

Push B#2
top
B @ 2
C @ 1

Problem Illustration (C11)



No conversion, I/O, or optimization frontier, but useful definitions

“Zombie Pointer” needs provenance from “Push C”

What Else Is Needed?

- P2414R4 (“Pointer lifetime-end zap proposed solutions”): Provisional provenance results from:
  - Conversions to/from atomic<T *>
    - Including old pointer referenced by successful CAS operations
  - usable_ptr<T>
  - make_ptr_prospective() “identity” function
  - Volatile accesses involving pointers
- P3347R0 (“Pointer lifetime-end zap proposed solutions: Tighten IDB for invalid and prospective pointers”)
  - Glvalue-to-rvalue conversions from invalid pointers must produce value bits consistent with those of the lvalue

What Else Is Needed?

- P2414R4 (“Pointer lifetime-end zap proposed solutions”): Provisional provenance results from:
  - Conversions to/from atomic<T *>
    - Including old pointer in successful CAS operations
  - usable_ptr<T>
  - make_ptr_prospective() “identity” fun
  - Volatile accesses involving
- P3347R0 (“Pointer zap proposed solutions: Tighten IDB for invalid and prospective pointers
  - Glvalue-to-rvalue conversions from invalid pointers must produce value bits consistent with those of the lvalue

Not all that much!!!

Status in C++ Committee

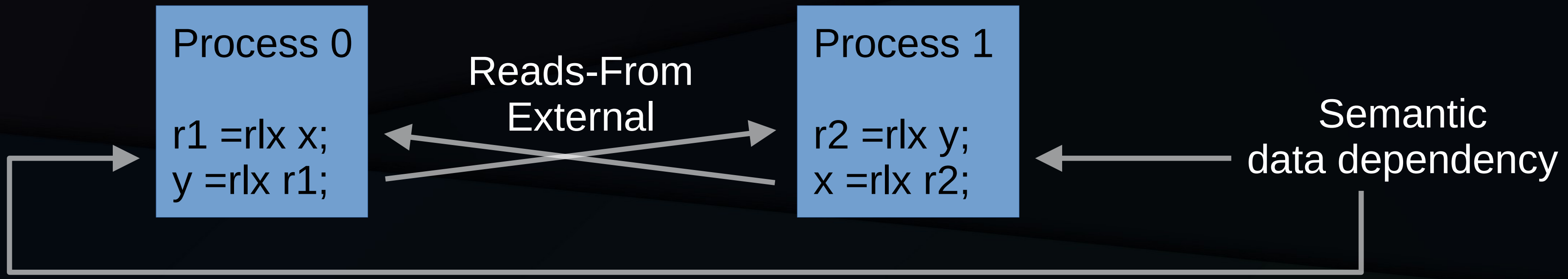
- All progressing through C++ committee:
  - P2414R4 “Pointer lifetime-end zap proposed solutions”
  - P3347R0 Invalid/Prospective Pointer Operations
  - Davis Herring’s P2434R1 “Nondeterministic pointer provenance”
- No guarantees, but best progress thus far

Pointer-Zap Discussion

OOTA Cycles

OOTA Cycles

- Self-satisfying load-buffering cycle, x==y==42



Process 0

r1 =rlx x;
y =rlx r1;

Reads-From
External

Process 1

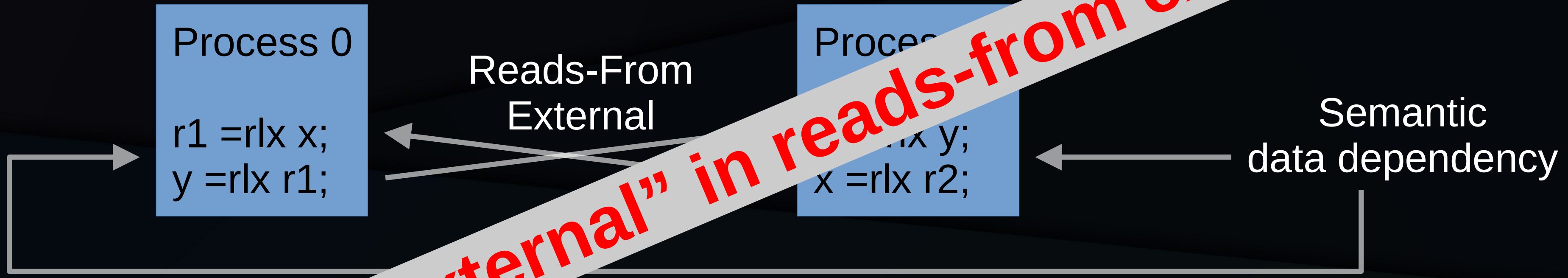
r2 =rlx y;
x =rlx r2;

Semantic
data dependency

Relaxed load or store denoted by “=rlx”

OOTA Cycles

- Self-satisfying load-buffering cycle, x==y==42



Process 0

r1 =rlx x;
y =rlx r1;

Reads-From
External

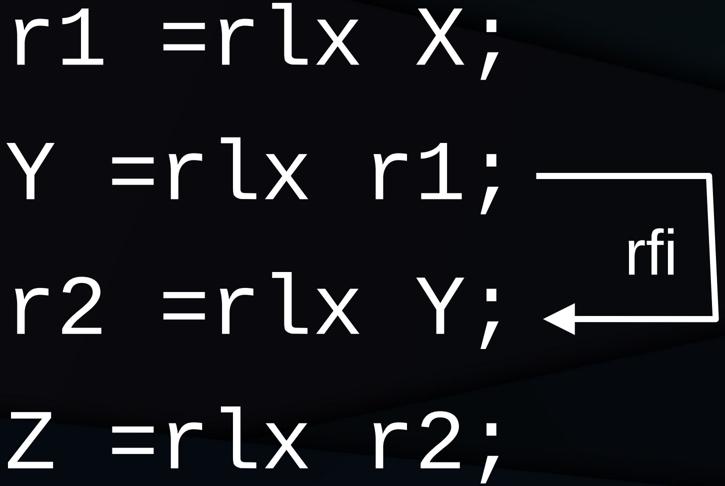
Proces

lx y;
x =rlx r2;

Semantic
data dependency

Why “external” in reads-from external?

OOTA Cycles: Reads-From Internal



```
r1 =rlx X;
Y =rlx r1;
r2 =rlx Y;
Z =rlx r2;
```

OOTA Cycles: Reads-From Internal

```
r1 =rlx X;
Y =rlx r1;
r2 =rlx Y;
Z =rlx r2;
```

→

```
r1 =rlx X;
Z =rlx r1;
Y =rlx r1;
r2 = r1;
```

OOTA Cycles: Reads-From Internal

```
r1 =rlx X;
Y =rlx r1;
r2 =rlx Y;
Z =rlx r2;
```

```
r1 =rlx X;
Z =rlx r1;
Y =rlx r1;
r2 = r1;
```

Compiler eliminated the read from Y so that
the store to Z can now occur before the store to Y

OOTA Cycles: Reads-From Internal

```
r1 =rlx X;
Y =rlx r1;
r2 =rlx Y;
Z =rlx r2;
```

```
r1 =rlx X;
Z =rlx     ;
Y      lx r1;
r2 = r1;
```

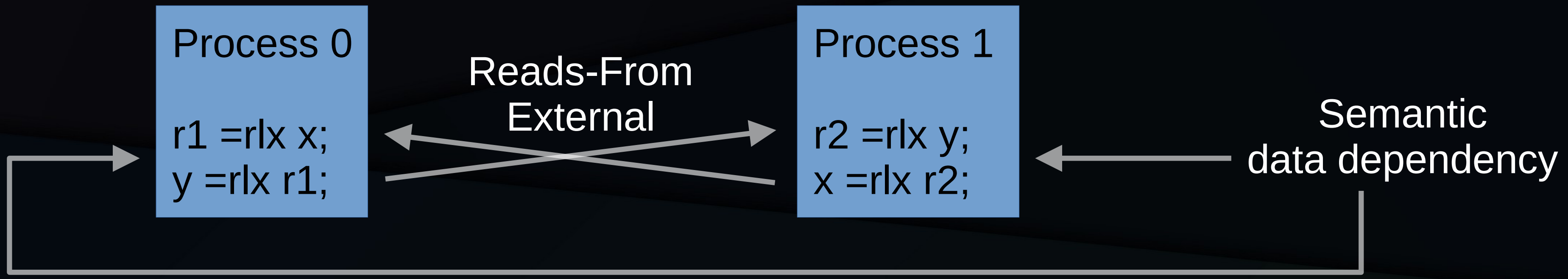
Hence "external" in reads-from external

piler eliminated the read from Y so that
the store to Z can now occur before the store to Y

See Appendix D.3 ("Why rfe Instead of Tried-And-True rf?") of P3064R1

OOTA Cycles, Original Diagram

- Self-satisfying load-buffering cycle, x==y==42



Process 0

r1 =rlx x;
y =rlx r1;

Reads-From
External

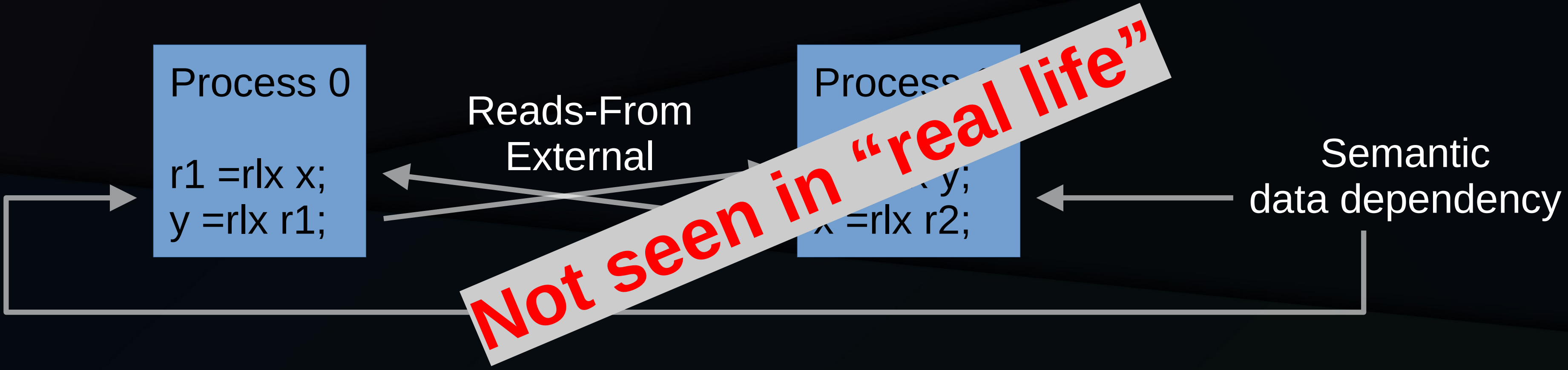
Process 1

r2 =rlx y;
x =rlx r2;

Semantic
data dependency

OOTA Cycles, Original Diagram

- Self-satisfying load-buffering cycle, x==y==42



Process 0

r1 =rlx x;
y =rlx r1;

Reads-From
External

Process

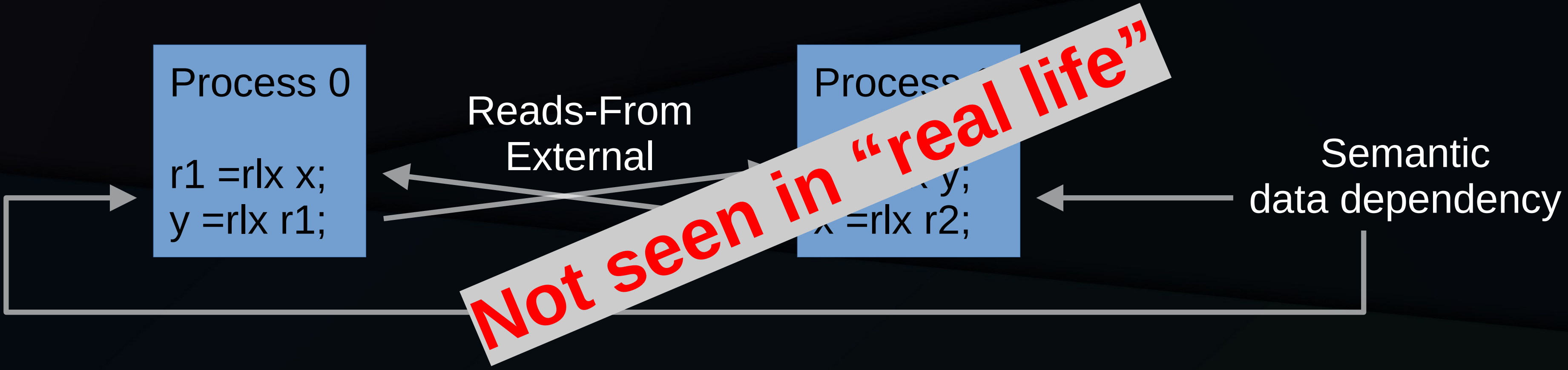
y;
x =rlx r2;

Semantic
data dependency

Not seen in “real life”

OOTA Cycles, Original Diagram

- Self-satisfying load-buffering cycle, x==y==42



Process 0

r1 =rlx x;
y =rlx r1;

Reads-From
External

Process

y;
x =rlx r2;

Semantic
data dependency

Not seen in “real life”

Why???

Where Are We on OOTA?

Where Are We on OOTA? (TL;DR)

Process 0

r1 =rlx x;
y =rlx r1;

Reads-From
External

Process 1

r2 =rlx y;
x =rlx r2;

Where Are We on OOTA? (TL;DR)

Process 0

```
r1 =rlx x;
y =rlx r1;
```

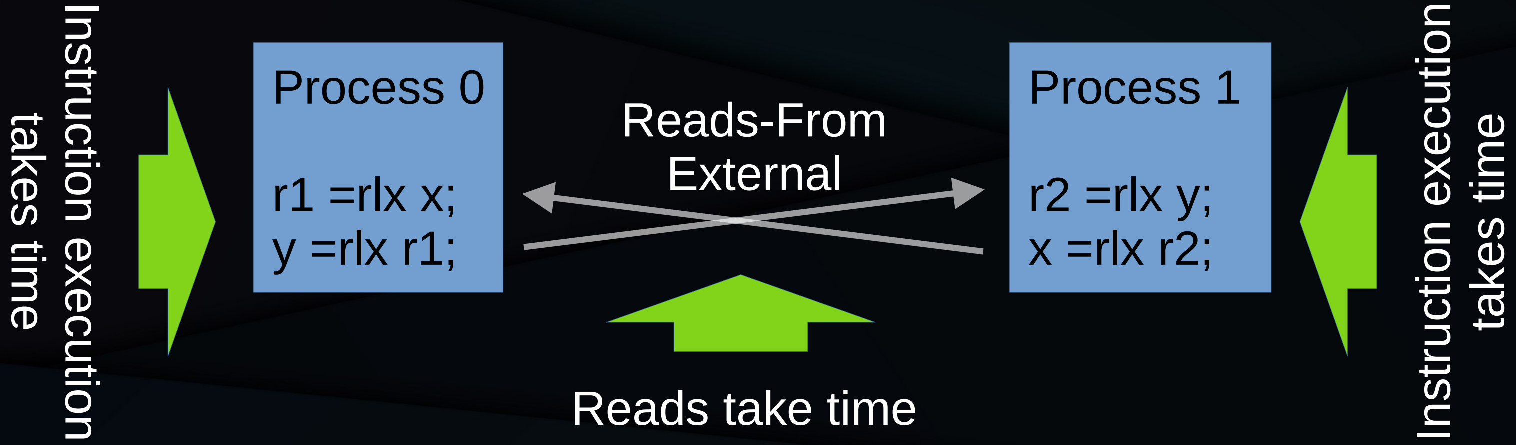
Process 1

```
r2 =rlx y;
x =rlx r2;
```

Reads-From External

Reads take time

Where Are We on OOTA? (TL;DR)



Instruction execution takes time

Process 0

r1 =rlx x;
y =rlx r1;

Reads-From External

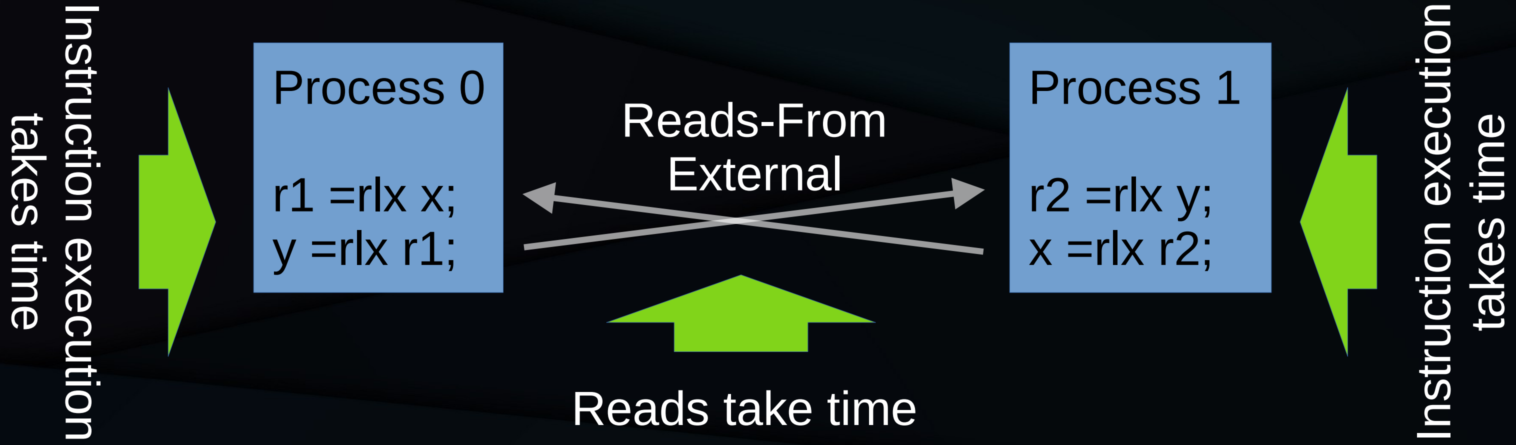
Reads take time

Process 1

r2 =rlx y;
x =rlx r2;

Instruction execution takes time

Where Are We on OOTA? (TL;DR)



Instruction execution takes time

Process 0

```
r1 =rlx x;
y =rlx r1;
```

Reads-From External

Reads take time

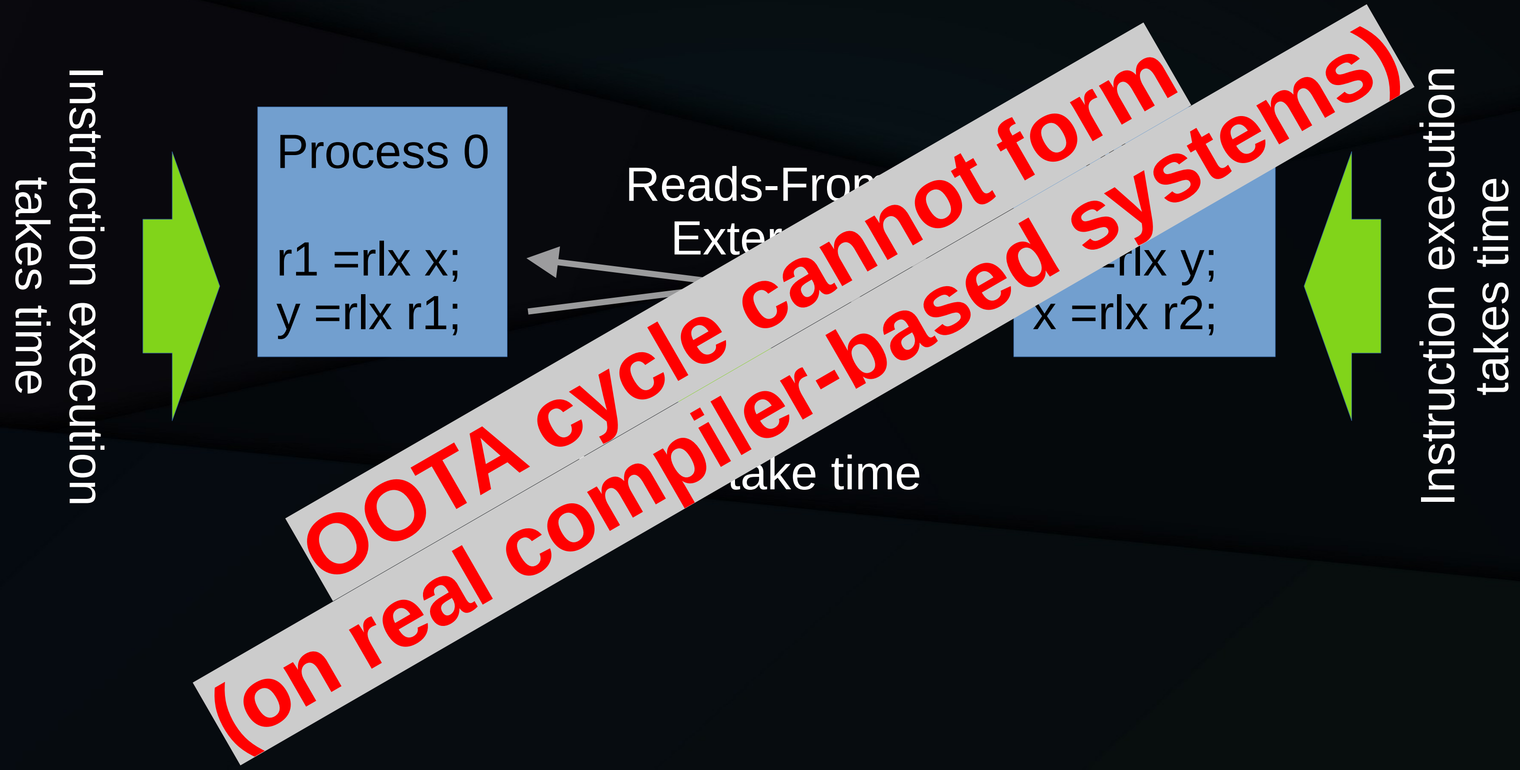
Process 1

```
r2 =rlx y;
x =rlx r2;
```

Instruction execution takes time

To form an OOTA cycle, at least one step must go backwards in time!!!

Where Are We on OOTA? (TL;DR)



OOTA cycle cannot form
(on real compiler-based systems)

To form an OOTA cycle, at least one step must go backwards in time!!!

Where Are We on OOTA?

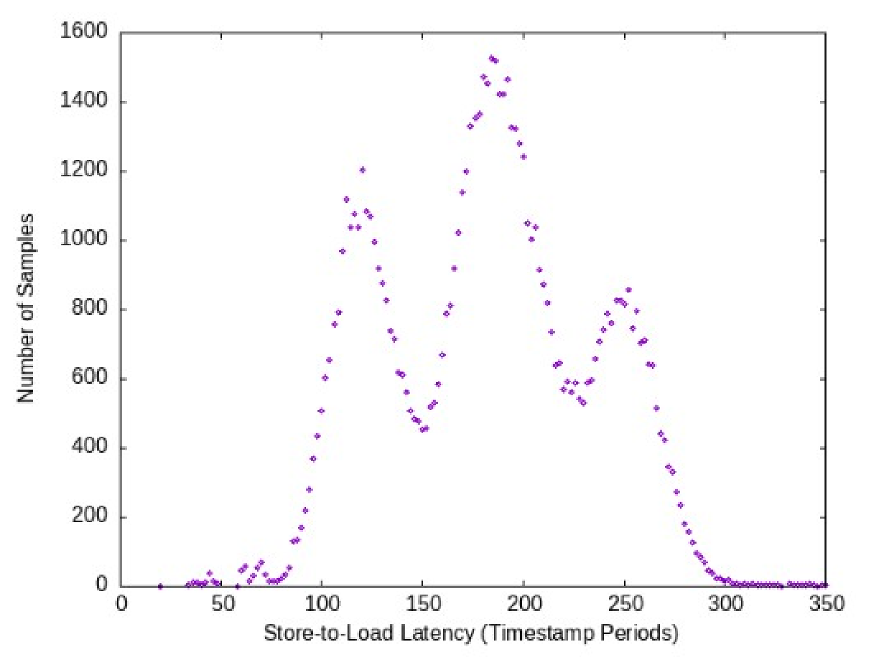
- Generalized “OOTA Cycle” (Section 2.2.2)
- Fundamental property of semantic dependency (Sections 5.3 and 6.1)
- Demonstrate OOTA-freedom under restrictions (Sections 6.2 and 6.3 for demonstration, 4.4 for restrictions)

Leverage Restrictions

Real Computer Systems

Real Computer Systems: Store-to-Load

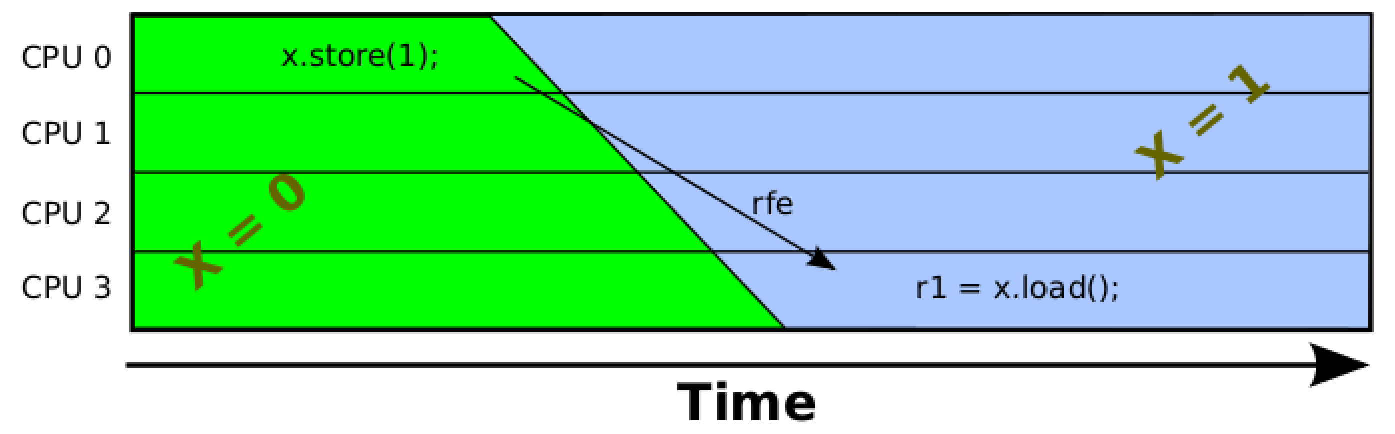
- Store-to-load links are temporal*



* The event that is logically first must happen before the other event in real-world time
Dual-socket Intel(R) Xeon(R) Gold 6138 CPUs @ 2.00 GHz, 80 hardware threads total: Measure beginning of store to end of load

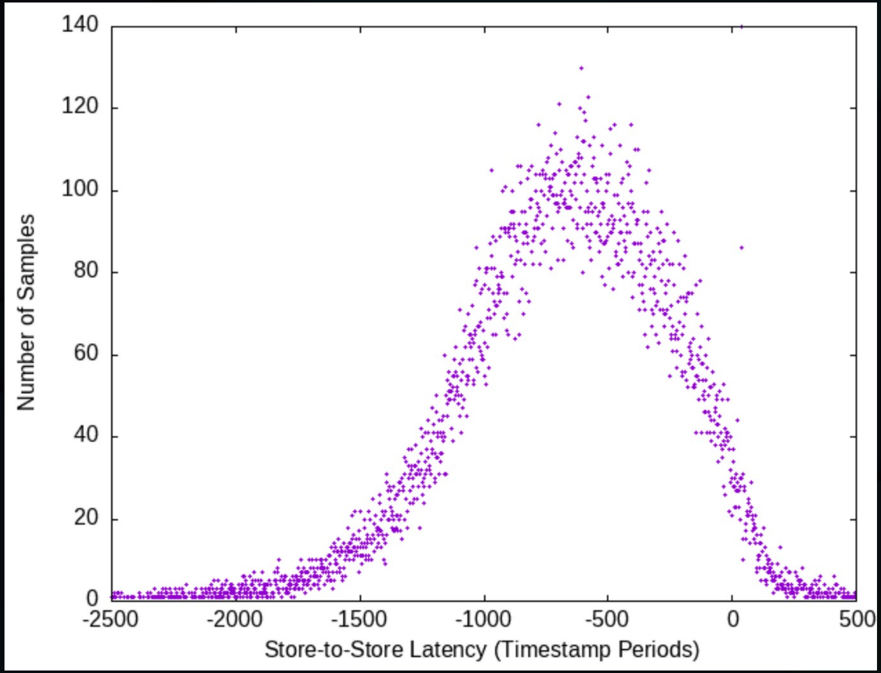
Real Computer Systems: Store-to-Load

- Store-to-load links are temporal: HW view



Real Computer Systems: Store-to-Store

- Store-to-store links are atemporal*

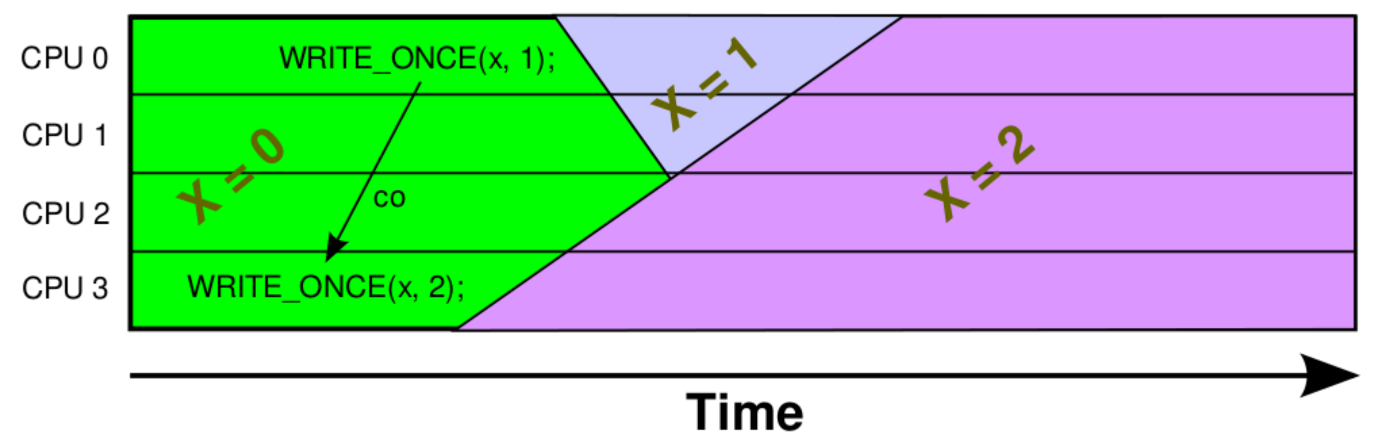


* The event which is logically first can happen after the other event in real-world time

Dual-socket Intel(R) Xeon(R) Gold 6138 CPUs @ 2.00 GHz, 80 hardware threads total: Measure beginning of winning store to end of store

Real Computer Systems: Store-to-Store

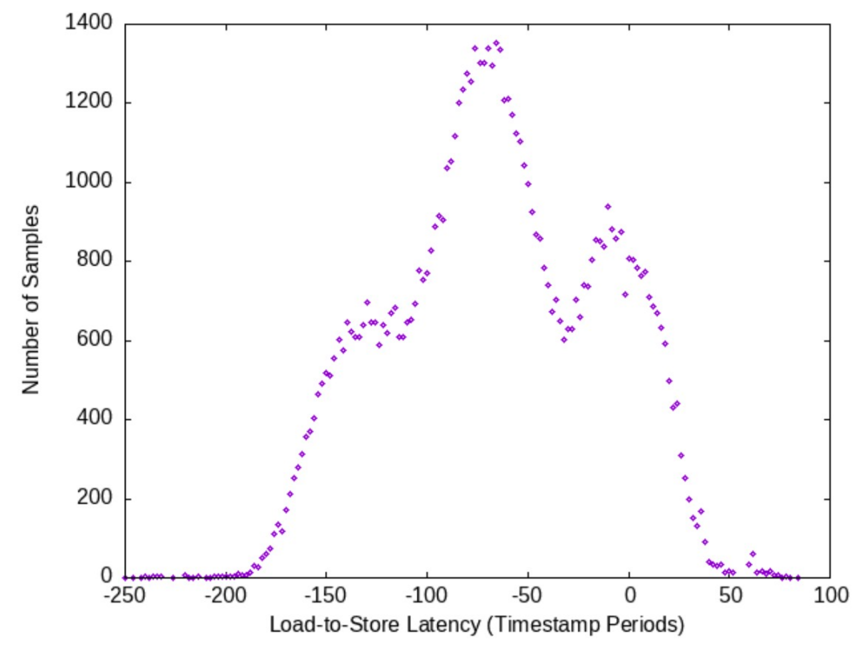
- Store-to-store links are atemporal: HW view



“co” is “modification order” in the C++ memory model

Real Computer Systems: Load-to-Store

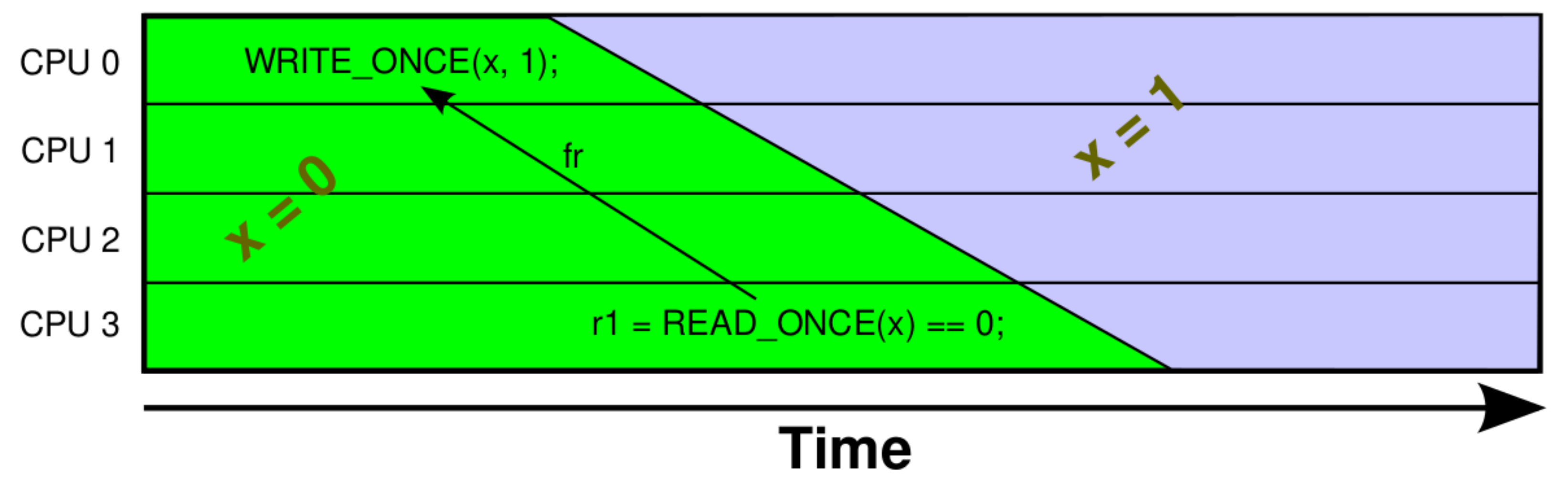
- Load-to-store links are atemporal



Dual-socket Intel(R) Xeon(R) Gold 6138 CPUs @ 2.00 GHz, 80 hardware threads total: Measure beginning of load to end of store

Real Computer Systems: Load-to-Store

- Load-to-store links are atemporal: HW view



“fr” is “from reads”, which connects a read to a write that happened too late to affect the value loaded

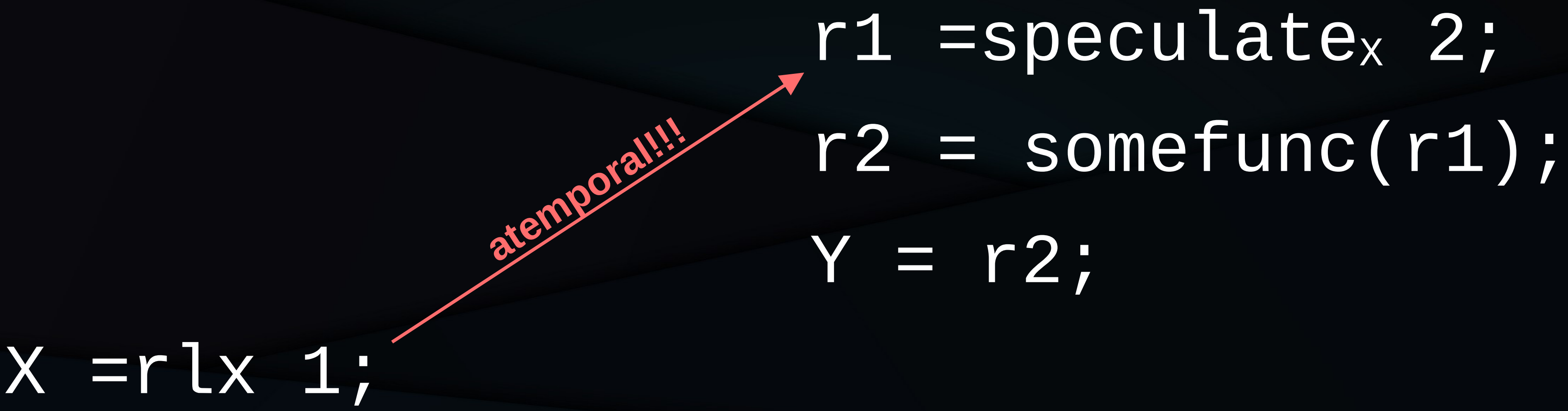
Real Computer Systems: Summary

- Load-to-store links: Atemporal
- Store-to-store links: Atemporal
- Store-to-load links: Temporal
  - And thus have ordering properties on the cheap

See Appendix A ("Interthread Communications") of P3064R1

Speculate Properly or Not At All

Speculate Properly or Not At All



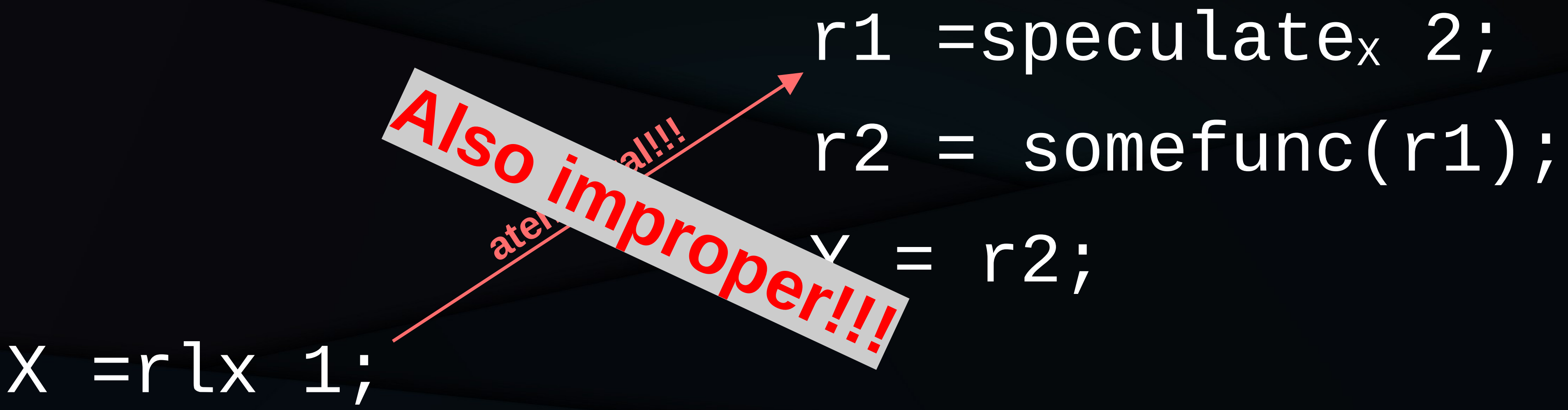
r1 =speculate$_x$ 2;

r2 = somefunc(r1);

Y = r2;

X =rlx 1;

Speculate Properly or Not At All



r1 =speculate_{x} 2;

r2 = somefunc(r1);

Y = r2;

X =rlx 1;

Also improper!!!

Don’t just guess! Guess and then check!!!

Speculate Properly or Not At All

```
X =rlx 1;
```

temporal!!!

```
r1 =speculate_x 2;
r2 = somefunc(r1);
~~Y = r2;~~
r3 =rlx X; // 1, not 2!
if (r1 != r3)
    r2 = somefunc(r3);
Y = r2;
```

Speculate Properly or Not At All

```
X =rlx 1;
```

temporal!!!

```
r1 =speculate_x 2;
r2 = somefun      );
~~Y = r2;~~
r3              1, not 2!
               r3)
    r2 = somefunc(r3);
Y = r2;
```

Speculation must be checked against the value from an actual load!!!

See Section 5.2 (“Instruction Ordering”) and Section 7.1 (“Hardware Architecture and Design”) of P3064R1

Existing Restrictions on Volatile Atomics

Existing Restrictions on Volatile Atomics

- Compiler may not:
  - Reorder accesses
  - Invent, duplicate, or repurpose accesses
  - Merge or fuse accesses
  - Omit accesses
- Relax restrictions for non-volatile atomics?

No Atomic-Load Invention/Repurposing

No Atomic-Load Invention

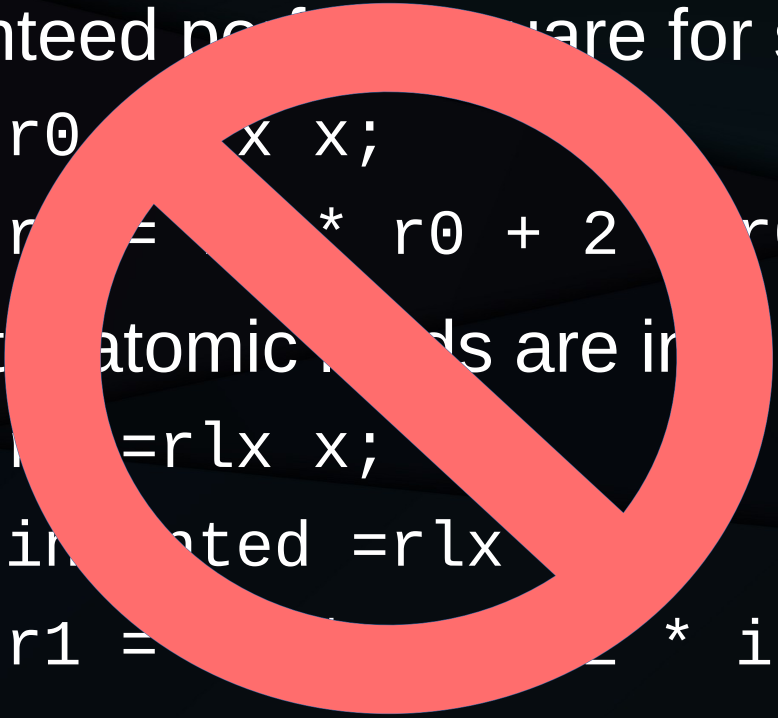
- Guaranteed perfect square for small X:

```
int r0 =rlx x;
int r1 = r0 * r0 + 2 * r0 + 1;
```

- But not if atomic loads are invented!!!

```
int r0 =rlx x;
int invented =rlx x;
int r1 = r0 * r0 + 2 * invented + 1;
```

No Atomic-Load Invention



- Guaranteed pe are for small X:

```
int r0       x x;
int r    =        * r0 + 2      r0 + 1;
```

- But not atomic ds are in nted!!!

```
int r    =rlx x;
int in    nted =rlx
int r1 =                     * invented + 1;
```

No Atomic-Load Repurposing

- Guaranteed perfect square for small X:

```
r2 =rlx x;
do_something(r2); // No synchronization or stores to x
int r0 =rlx x;
int r1 = r0 * r0 + 2 * r0 + 1;
```

- But not if atomic loads are repurposed!!!

```
r2 =rlx x;
do_something(r2); // No synchronization or stores to x
int r0 =rlx x;
int r1 = r0 * r0 + 2 * r2 + 1;
```

No Atomic-Load Repurposing

- Guaranteed per re for small X:

```
r2 =r
do_ ing(r2); // N chronization or stores to x
ir 0 x;
i r1 = r 0 + 2 * r0 ;
```

- But if atomic lo re repurp d!!!

```
r rlx x;
do ething(r2); / chronization or stores to x
int lx x;
int r1 r2 + 1;
```

Instead, Merge the Atomic Loads

- Guaranteed perfect square for small X:

```
r2 =rlx x;
do_something(r2); // No synchronization or stores to x
int r0 =rlx x;
int r1 = r0 * r0 + 2 * r0 + 1;
```

- And that guarantee is maintained for merged loads:

```
r0 =rlx x;
do_something(r0); // No synchronization or stores to x
int r1 = r0 * r0 + 2 * r0 + 1;
```

Instead, Merge the Atomic Loads

- Guaranteed perfect square for small X:

```
r2 =rlx x;
do_something(r2); // No synchronizati[illegible] to x
int r0 =rlx x;
int r1 = r0 * r0 + 2 * r0[illegible]
```

- And that guarantee is n[illegible]ged loads:

```
r0 =rlx x;
do_so[illegible] No synchronization or stores to x
[illegible]r0 + 2 * r0 + 1;
```

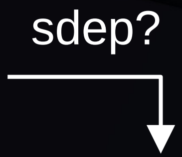
If do_something() contains synchronization, then must keep both atomic loads

Atomic Loads and Memory Ordering

```
r1 =rlx X;

r2 =rlx Y;

Z =rlx (r1 == r2);
```



```
X =rlx 1;
```

Note: X, Y, and Z boolean and initially zero

See Section 7.2 (“Constraints of the Standard”) and Appendix D.4 (“Inventing Atomic Loads”), Listing 22 of P3064R1

Atomic Loads and Memory Ordering

```
r1a =rlx X;
r1b =rlx X; // Invented load
If (r1a != r1b) {
    Z =rlx 1;
    r2 =rlx Y;
} else {
    r2 =rlx Y;
    Z =rlx (r1b == r2);
}
```

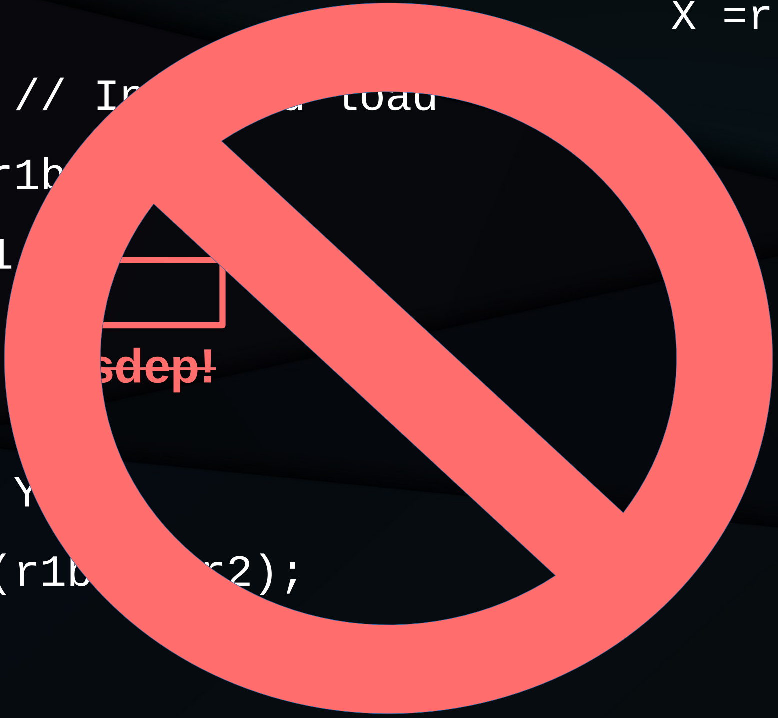
~~sdep!~~

```
X =rlx 1;
```

Note: X, Y, and Z boolean and initially zero

Inventing atomic load likely also invents hundreds-of-cycles cache miss!!!

Atomic Loads and Memory Ordering



```
r1a =rlx X;
r1b =rlx X; // In         load
If (r1a != r1b
    Z =rlx 1
    r2 =rlx
} else {
    r2 =rlx Y
    Z =rlx (r1b     r2);
}
```

~~sdep!~~

```
X =rlx 1;
```

Note: X, Y, and Z boolean and initially zero

See Appendix D.4 (“Inventing Atomic Loads”), Listing 23 of P3064R1

Non-Volatile Atomics Optimizations?

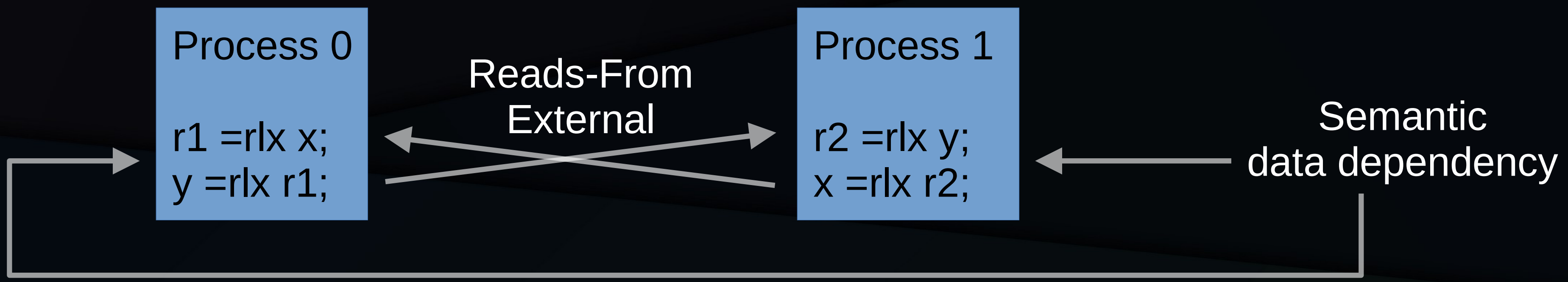
- Looking only at relaxed operations:
  - **Reorder loads/stores from/to different objects**
  - **Merge back-to-back loads to same object**
  - **Drop loads whose values are unused**
  - **Discard first of back-to-back stores to same object**
  - **Fuse loads from (or stores to) adjacent objects if this results in a machine-word-sized/aligned access**
  - **But no invented, duplicated, or repurposed loads!!!**

Tooling Looks at Object Code

See Section 7.3 (“Semantic Dependencies and Tooling”) and Appendix C (“But What About Tooling?”), P3064R1

OOTA Cycles, Original Diagram

- Self-satisfying load-buffering cycle, x==y==42



Process 0

r1 =rlx x;
y =rlx r1;

Reads-From
External

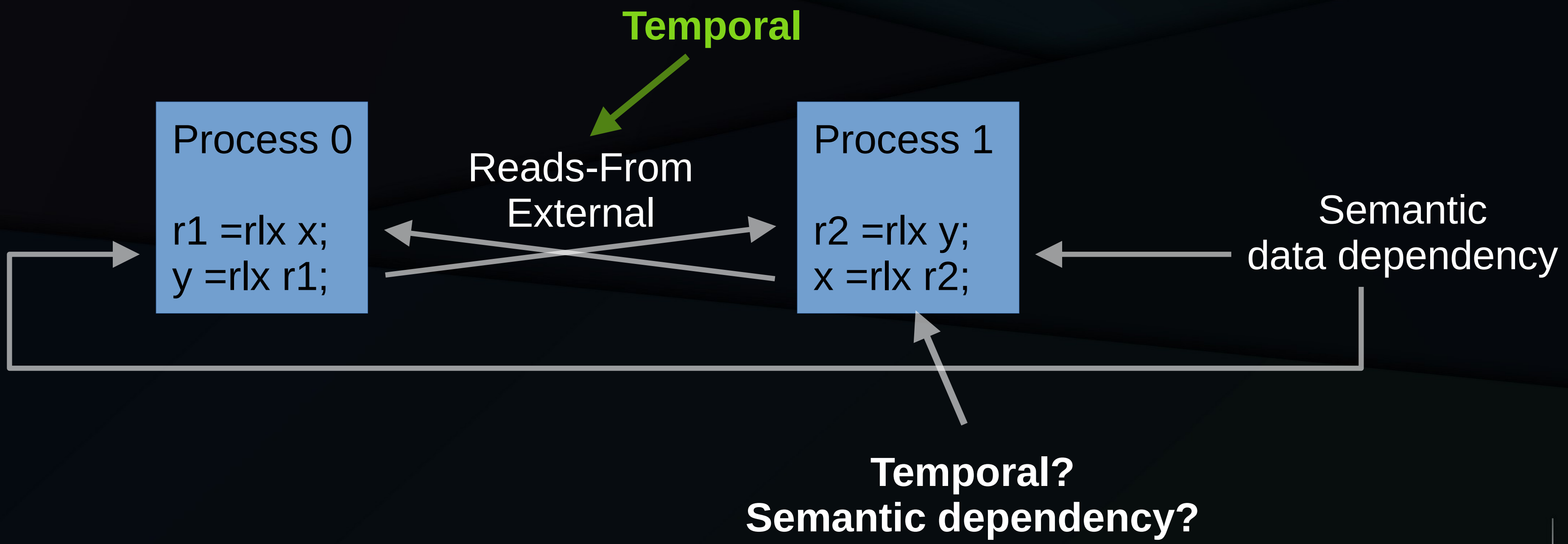
Process 1

r2 =rlx y;
x =rlx r2;

Semantic
data dependency

OOTA Cycles, Original Diagram

- Self-satisfying load-buffering cycle, x==y==42



Temporal

Process 0

r1 =rlx x;
y =rlx r1;

Reads-From
External

Process 1

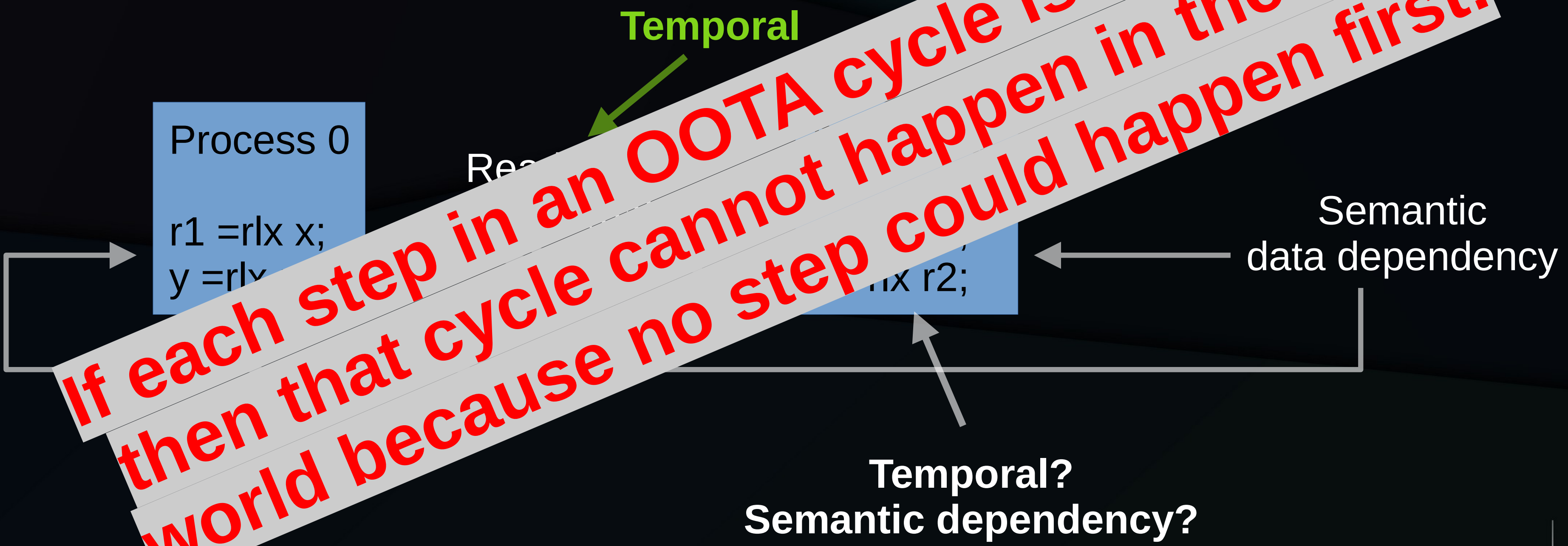
r2 =rlx y;
x =rlx r2;

Semantic
data dependency

Temporal?
Semantic dependency?

OOTA Cycles, Original Diagram

- Self-satisfying load-buffering cycl =42



If each step in an OOTA cycle is temporal, then that cycle cannot happen in the real world because no step could happen first!

Semantic Dependencies are Tricky

- At source-code level, semantic dependencies:
  - Are not strict functions of source code (Section 2)
  - Can be many-to-one (Section 2 and Appendix D.2)
  - Depend on partially defined executions (Section 3)
  - Depend on compilers and their users (Section 4)
- Current paper assumes local analysis (no global cross-thread optimizations)

Semantic Dependencies in Code?

- Semantic dependencies are temporal:
  - Instructions take time to execute
  - Speculation must be checked against actual load

Semantic Dependencies in Code?

- Semantic dependencies are temporal:
  - Instructions take time to execute
  - Speculation must be checked against actual load
- Compiler optimizations break dependencies:
  - But HW memory models respect dependencies
  - Thus look at object code (seL4 verification approach)
  - Also look at other compiler-produced artifacts

See Sections 5-7, P3064R1

Semantic Dependencies in Code?

- Semantic dependencies are temporal:
  - Instructions take time to execute
  - Speculation must be chec
- Compiler optimiz ncies:
  - But HW dependencies
  - Th (seL4 verification approach)
  - compiler-produced artifacts

If compiler optimizes dependency away, it was not semantic. Otherwise, executing dependency’s code will take time.

See Sections 5-7, P3064R1

Where Are We on OOTA? (Reprise)

- Generalized “OOTA Cycle” (Section 2.2.2)
- Fundamental property of semantic dependency (Sections 5.3 and 6.1)
- Demonstrate OOTA-freedom under restrictions (Sections 6.2 and 6.3 for demonstration, 4.4 for restrictions)
  - The main restriction is: No invented, duplicated, or repurposed atomic loads

Future Directions

- From compilers to (some) JITs, interpreters, and link-time optimizations (LTO)
- Compilers doing (some) global analysis given volatile atomics
- Identify absolute semantic dependencies inherent in source code
- Non-shared-memory communication

Discussion