OZZ: Identifying Kernel Out-of-Order Concurrency Bugs with In-Vivo Memory Access Reordering

Authors: Dae R. Jeong, Yewon Cho, Byoungyoung Lee,
Insik Shin, Youngjin Kwon

Presented by **Jiyang Wang**

2025–5-13

Background: Out-of-order execution

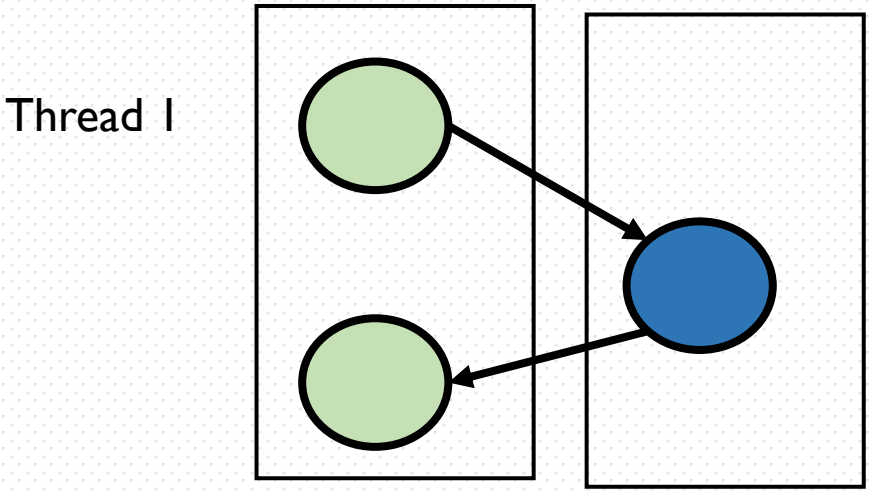
- **Why exist Out-of-order execution?**
  - **Reduce pipeline stalls**
  - **Improve cache utilization**

Background: Out-of-order execution

- **Why exist Out-of-order execution?**
  - **Reduce pipeline stalls**
  - **Improve cache utilization**
- **Different with thread interleaving**

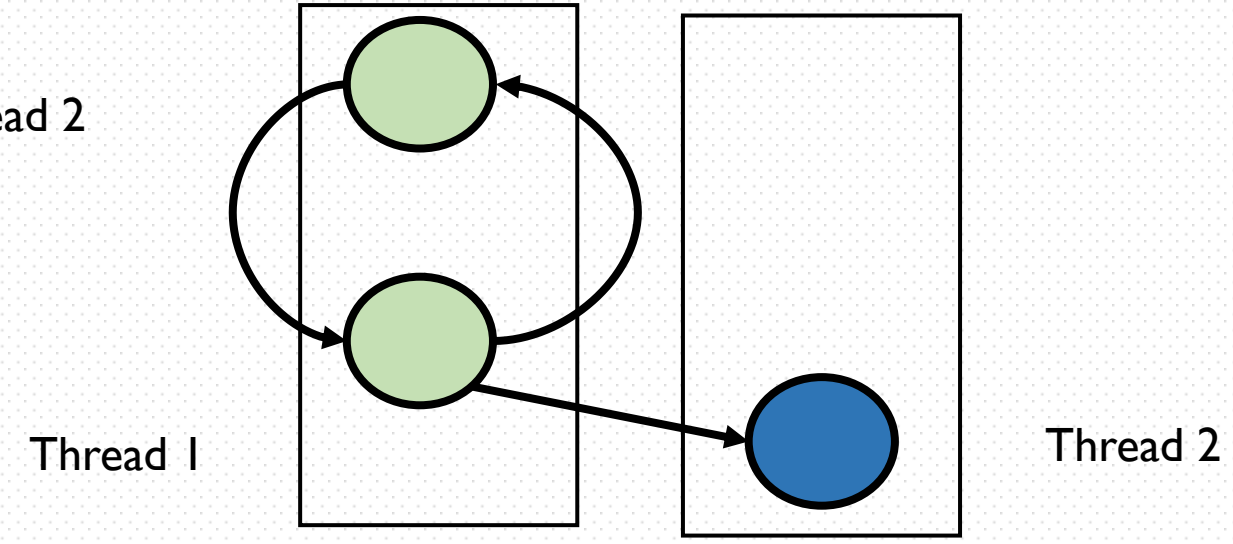
Thread interleaving

Caused by OS/scheduler



Out-of-order execution

Caused by processor



Background: Out-of-order execution

- How to prevent Out-of-order execution?

Memory barrier!

Type	Memory barrier API	Precedent accesses	Subsequent accesses
Full	smp_mb()	loads/stores	loads/stores
Load	smp_rmb()	loads	loads
Store	smp_wmb()	stores	stores
Release	smp_store_release(&a, v)	load/stores	store to &a
Acquire	smp_load_acquire(&a)	load from &a	loads/stores
Relaxed	READ_ONCE()/WRITE_ONCE()	none	none

Background: Out-of-order execution

❑How to prevent Out-of-order execution?

Obey Linux Kernel Memory Model!

- It defines reordering cases that would not occur

❖Data dependency

load-store

```
int r1;
r1 = X;
Y = r1 + 5;
```

Background: Out-of-order execution

❑How to prevent Out-of-order execution?

Obey Linux Kernel Memory Model!

- It defines reordering cases that would not occur

❖**Data dependency** ***load-store***

❖**Control dependency** ***load-store***

```
if (X == 0) {
    Y = 2;
}
```

← Affect the result of *if condition*

← A store operation inside the if statement

Background: Out-of-order execution

❑How to prevent Out-of-order execution?

Obey Linux Kernel Memory Model!

- It defines reordering cases that would not occur

❖**Data dependency** ***load-store***

❖**Control dependency** ***load-store***

❖**Address dependency** ***load-load / load-store***

X : load the value *i* ⟵ should use READ_ONCE() or atomic_read()

Y : load or store the value *arr[i]*

Background: Out-of-order execution

❑Harm of Out-of-order execution

```
1  /******* Thread A *******/
2  /* kernel/watch_queue.c */
3  void post_one_notification()
4    buf = &pipe->bufs[head];
5    buf->len = len;
6    buf->ops = &wq_pipe_ops;
7  + smp_wmb();
8    head += 1;
9  }
10
```

```
11 /******* Thread B *******/
12 /* fs/pipe.c */
13 void pipe_read() {
14   if (head > tail) {
15 +   smp_rmb();
16     buf = &pipe->bufs[tail];
17     len = buf->len;
18     buf->ops->confirm();
19   }
20 }
```

Background: Out-of-order execution

❑Harm of Out-of-order execution

```
1  /******* Thread A *******/
2  /* kernel/watch_queue.c */
3  void post_one_notification()
4    buf = &pipe->bufs[head];
5    buf->len = len;
④ 6    buf->ops = &wq_pipe_ops;
7  + smp_wmb();
① 8    head += 1;
9  }
10
```

```
11 /******* Thread B *******/
12 /* fs/pipe.c */
13 void pipe_read() {
② 14   if (head > tail) {
15 +    smp_rmb();
16       buf = &pipe->bufs[tail];
17       len = buf->len;
③ 18     buf->ops->confirm();
19     }
20 }
```

pipe_read() access uninitialized function !

Background: Out-of-order execution

❑Harm of Out-of-order execution

```
1  /******* Thread A *******/
2  /* kernel/watch_queue.c */
3  void post_one_notification()
4    buf = &pipe->bufs[head];
5    buf->len = len;
6    buf->ops = &wq_pipe_ops;
7  + smp_wmb();
8    head += 1;
9  }
10
```

② ③

```
11 /******* Thread B *******/
12 /* fs/pipe.c */
13 void pipe_read() {
14   if (head > tail) {
15 +   smp_rmb();
16     buf = &pipe->bufs[tail];
17     len = buf->len;
18     buf->ops->confirm();
19   }
20 }
```

④ ①

pipe_read() access uninitialized function !

Background: Out-of-order execution

- **Hard to identify Out-of-order execution**
  - Manual investigation kernel code is difficult
  - Different processors reorder differently (ARM more aggressive than x86_64)
  - Existing testing tools (e.g. concurrency fuzzers) is impractical
    - They assume memory accesses happened in order
    - Control thread interleaving may impose an ordered execution
  - *In-vitro* testing is insufficient
    - Lost runtime contexts when analyzing behavior
  - Most of data race detector short in comprehending the Out-of-order execution
    - What memory accesses should not be reordered
    - What will be the result of reordering

Key idea of in-vivo OEMU

- **Processor do Out-of-order execution**
  - **store-store, store-load, load-load**
  - **~~load-store~~**
    - **Theoretically can cause, but provides little improve in practice**

Key idea of in-vivo OEMU

- **Processor do Out-of-order execution**
  - Delay committing the ***store*** operation
  - Run a ***Load*** operation too early

Key idea of in-vivo OEMU

❑Processor do Out-of-order execution

- ❖Delay committing the ***store*** operation
- ❖Run a ***Load*** operation too early

- **Exactly emulating processor's behavior?**

☹ **Require simulate the full architecture, too expensive !**

Key idea of in-vivo OEMU

❑Processor do Out-of-order execution

- ❖Delay committing the ***store*** operation
- ❖Run a ***Load*** operation too early

- **Exactly emulating processor's behavior?**

☹ **Require simulate the full architecture, too expensive !**

- **Controlling Out-of-order execution explicitly and deterministically ?**

☹ **Execution order of instruction is decided in the processors !**

Key idea of in-vivo OEMU

❑Processor do Out-of-order execution

- ❖Delay committing the ***store*** operation
- ❖Run a ***Load*** operation too early

- **Exactly emulating processor's behavior**

☹ **Require simulate the full architecture, too expensive !**

- **Controlling Out-of-order execution explicitly and deterministically**

☹ **Execution order of instruction is decided in the processors !**

We can change the order of memory access in the instructions

Key idea of in-vivo OEMU

❑Processor do Out-of-order execution

- ❖Delay committing the ***store*** operation
- ❖Run a ***Load*** operation too early

delayed store operation

versioned load operation

System call interface	Description
delay_store_at(*I*)	When an instruction *I* is executed, its store operation will be delayed.
read_old_value_at(*I*)	When an instruction *I* is executed, its load operation will read an old value.

Systems calls to instruct OEMU to control Out-of-order execution

We can change the order of memory access in the instructions

Key idea of in-vivo OEMU

❑In-vivo emulation

```
// Original source code
x = 1;
r1 = y;
```

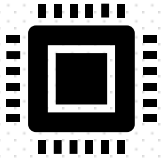
During compilation

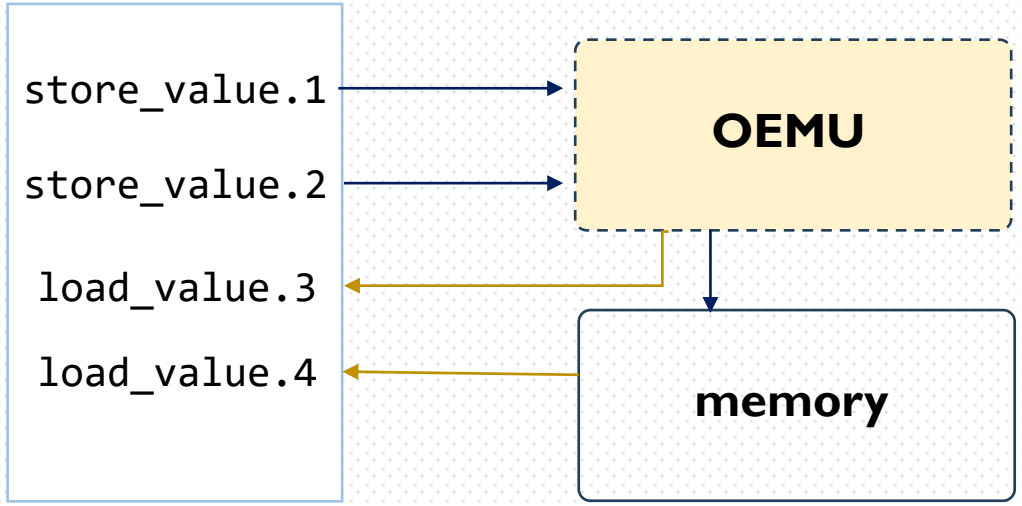
```
// Transformed code
store_value(&x, 1);
r1 = load_value(&y);
```

Callback functions

Variable on the stack

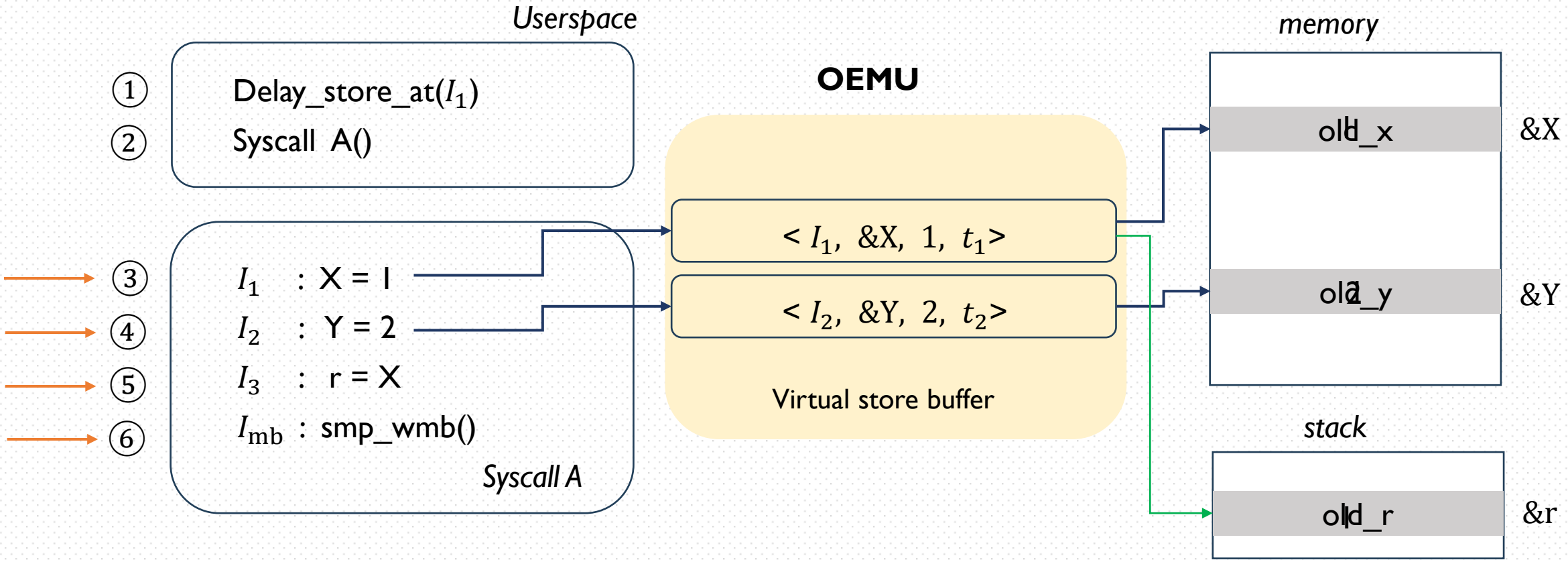
- Functions are executed sequentially
- Functions commute with OEMU to reorder memory access





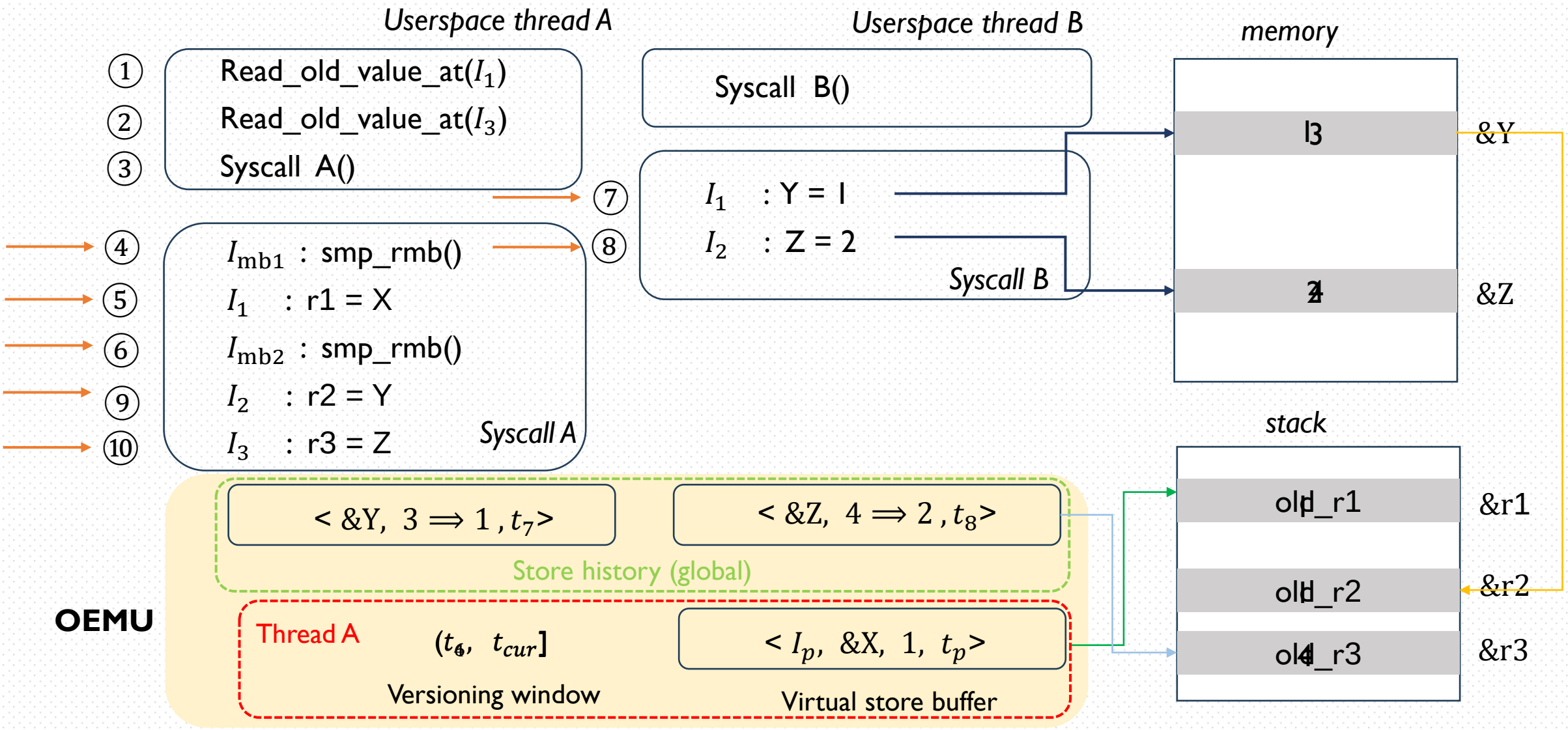
Reorder while kernel is running. Thus, can use all bug-detecting oracles

OEMU: Delayed store operation



- Virtual store buffer is a **per-thread**, **temporary** storage
- Flush when full, encounter memory barrier, interrupt on kernel …

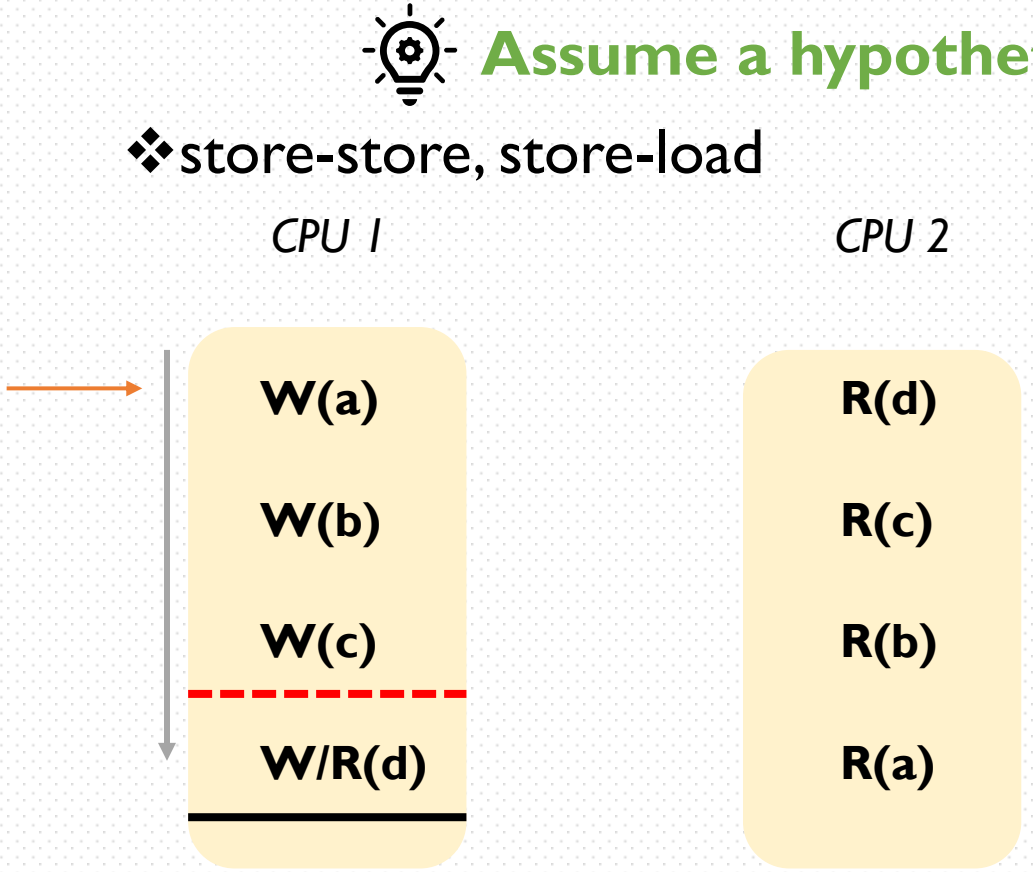
OEMU: Versioned load operation



Ozz: Key idea

Assume a hypothetical memory barrier is missing

❖store-store, store-load



: real memory barrier

: hypothetical memory barrier

Ozz: Key idea

Assume a hypothetical memory barrier is missing

❖store-store, store-load

CPU 1	Committed?	*CPU 2*
W(a)	N	R(d)
W(b)	N	R(c)
W(c)	N	R(b)
W/R(d)	Y	R(a)

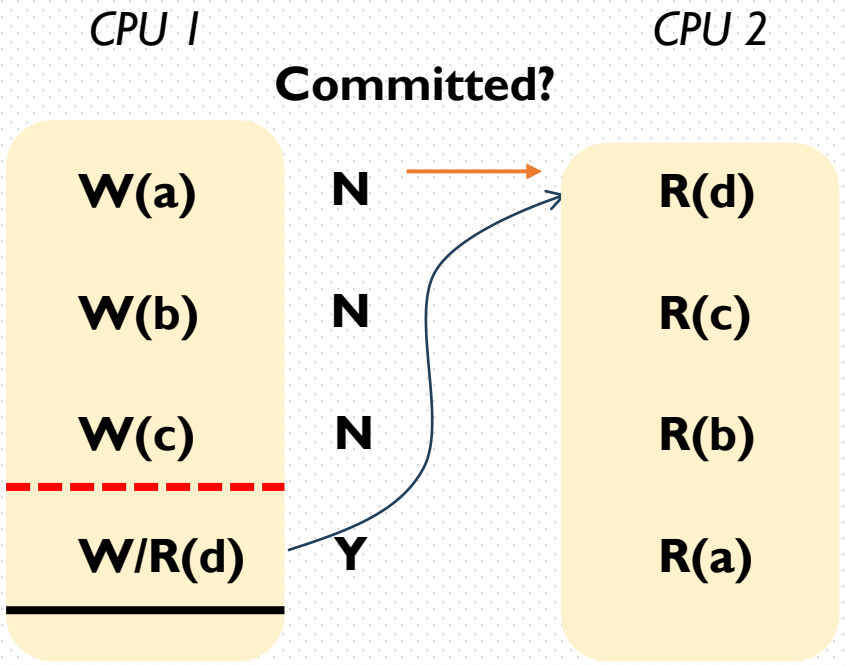
① reordering

——— : real memory barrier - - - - - : hypothetical memory barrier

Ozz: Key idea

Assume a hypothetical memory barrier is missing

❖store-store, store-load



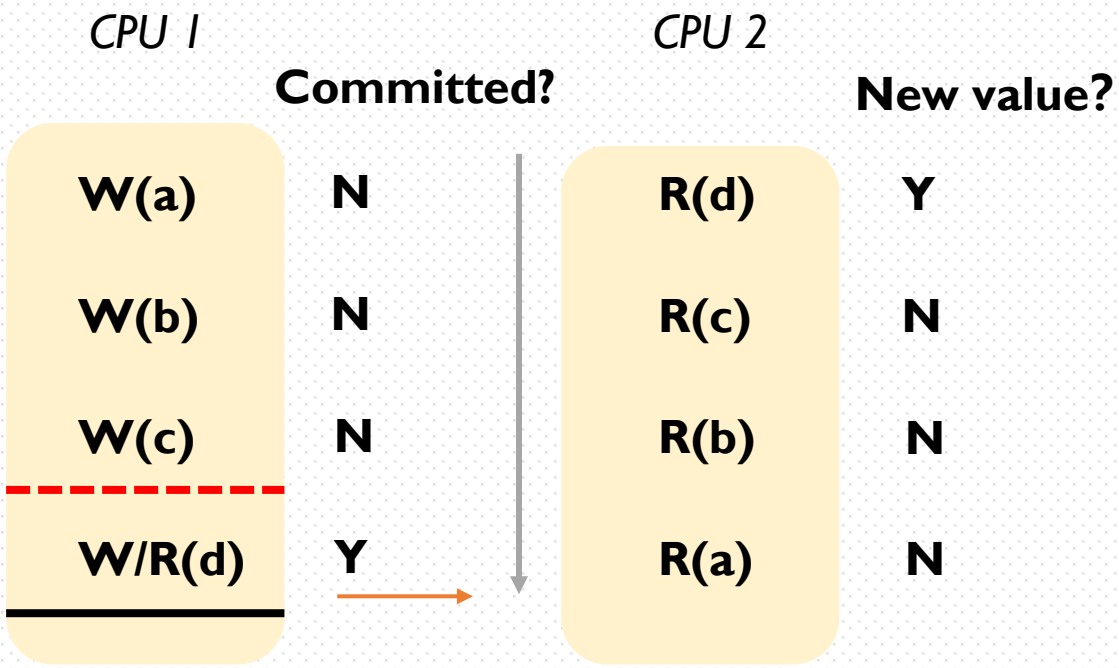
① reordering ② interleaving

——— : real memory barrier - - - - - : hypothetical memory barrier

Ozz: Key idea

Assume a hypothetical memory barrier is missing

❖store-store, store-load



① reordering ② interleaving ③ Testing

——— : real memory barrier ------ : hypothetical memory barrier

Ozz: Key idea

Assume a hypothetical memory barrier is missing

❖store-store, store-load

CPU 1	Committed?	*CPU 2*	New value?
W(a)	Y	R(d)	Y
W(b)	Y	R(c)	N
W(c)	Y	R(b)	N
W/R(d)	Y	R(a)	N

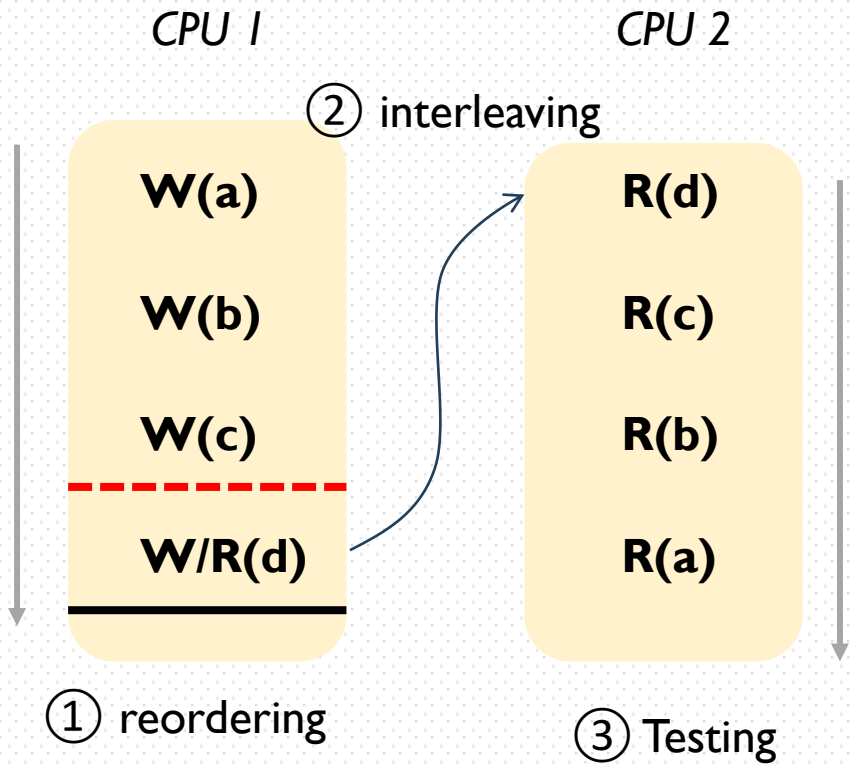
① reordering ② interleaving ③ Testing

——— : real memory barrier ------ : hypothetical memory barrier

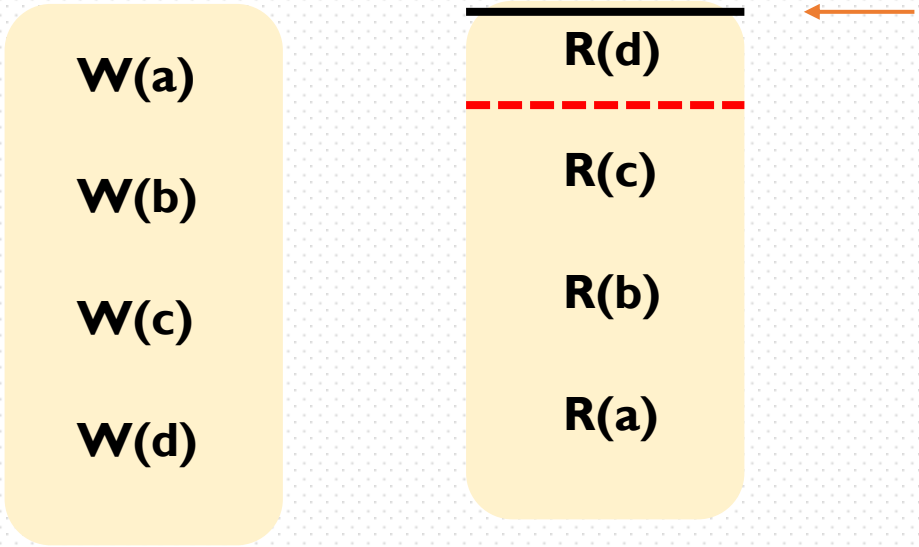
Ozz: Key idea

Assume a hypothetical memory barrier is missing

❖store-store, store-load



❖load-load



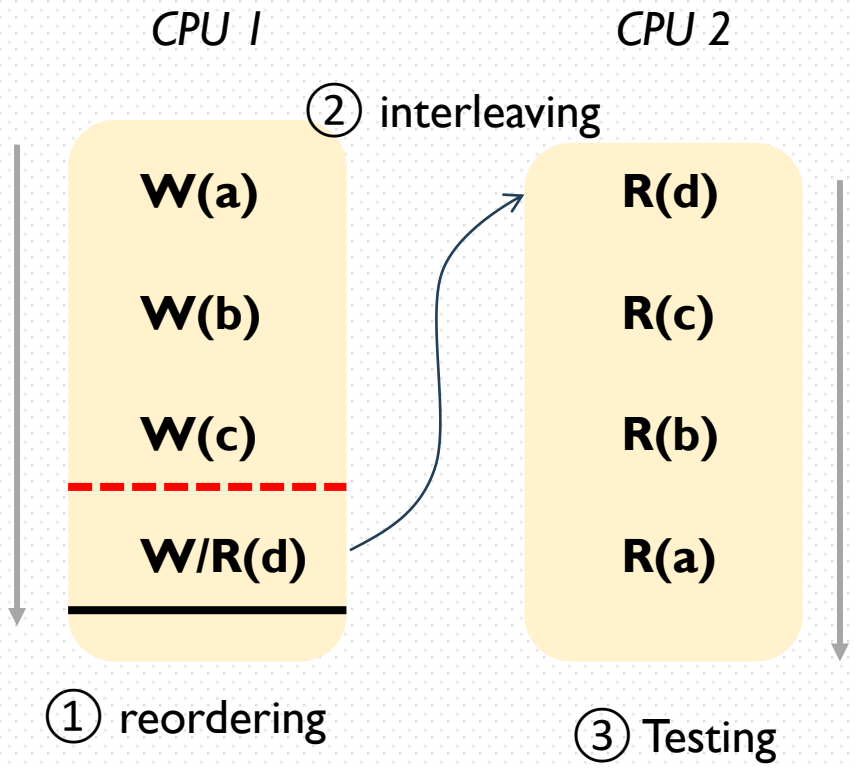
: real memory barrier

: hypothetical memory barrier

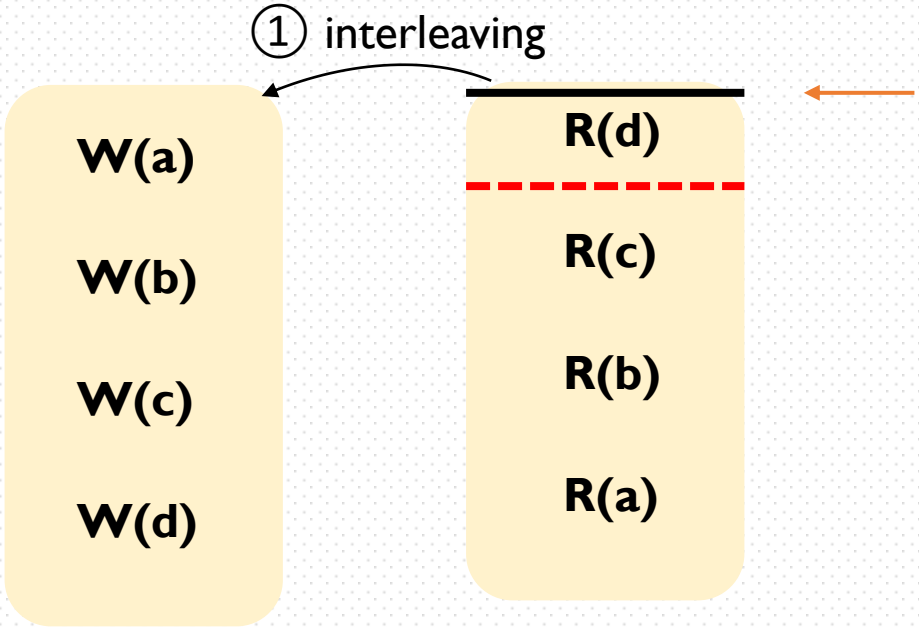
Ozz: Key idea

Assume a hypothetical memory barrier is missing

❖store-store, store-load



❖load-load



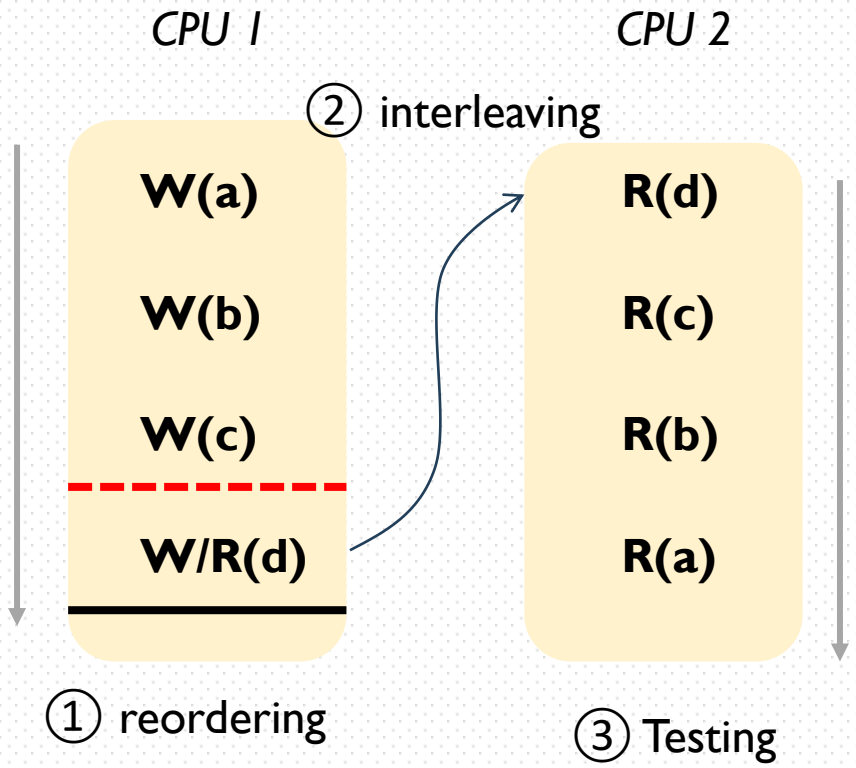
: real memory barrier

: hypothetical memory barrier

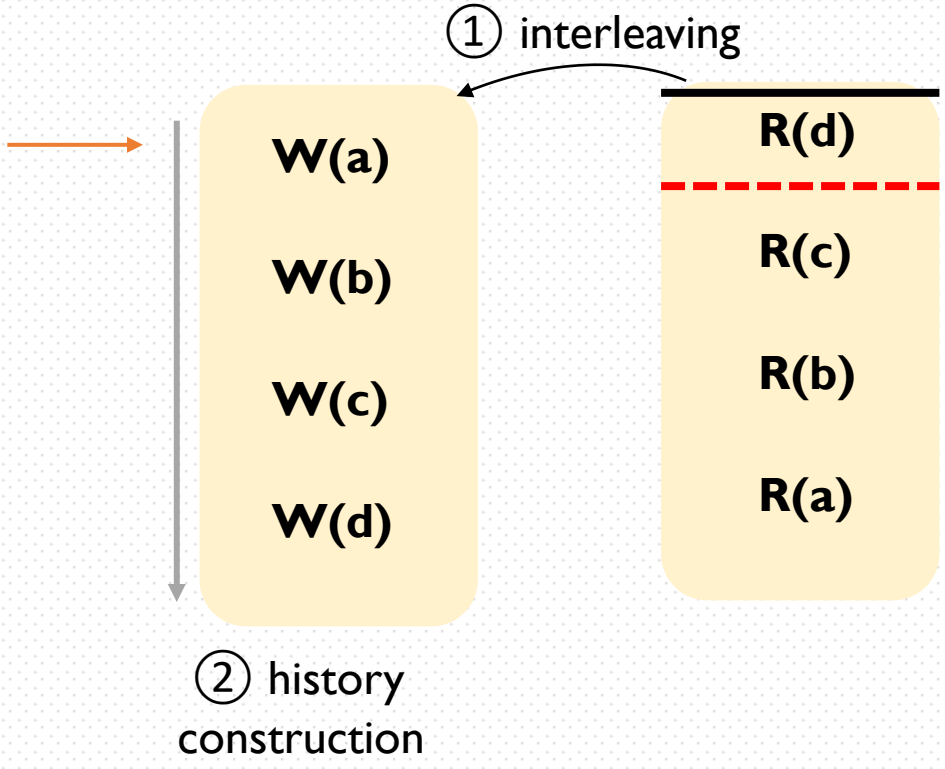
Ozz: Key idea

Assume a hypothetical memory barrier is missing

❖store-store, store-load



❖load-load



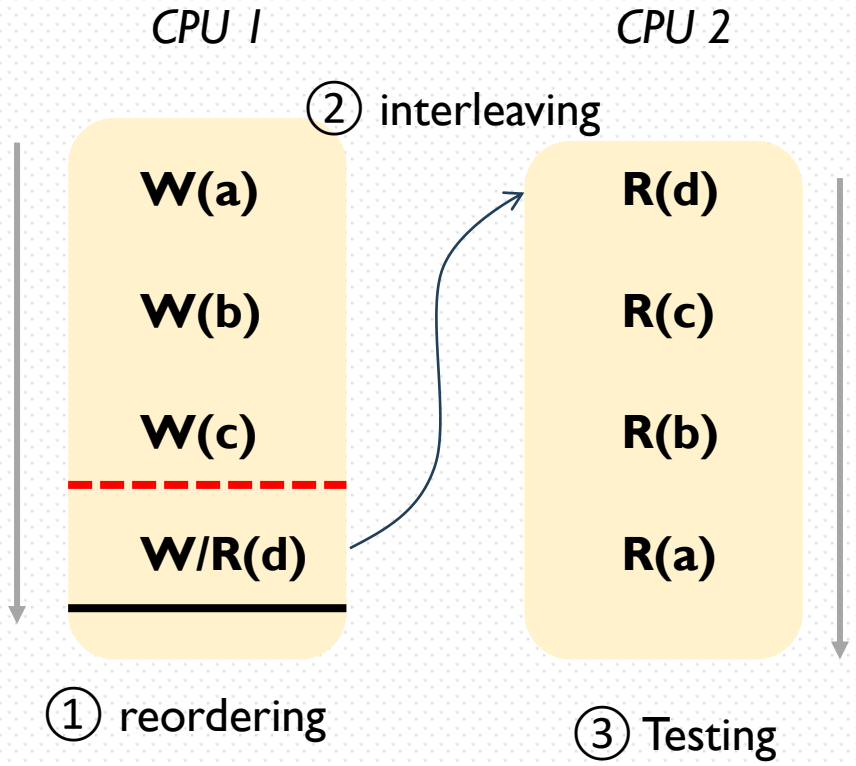
: real memory barrier

: hypothetical memory barrier

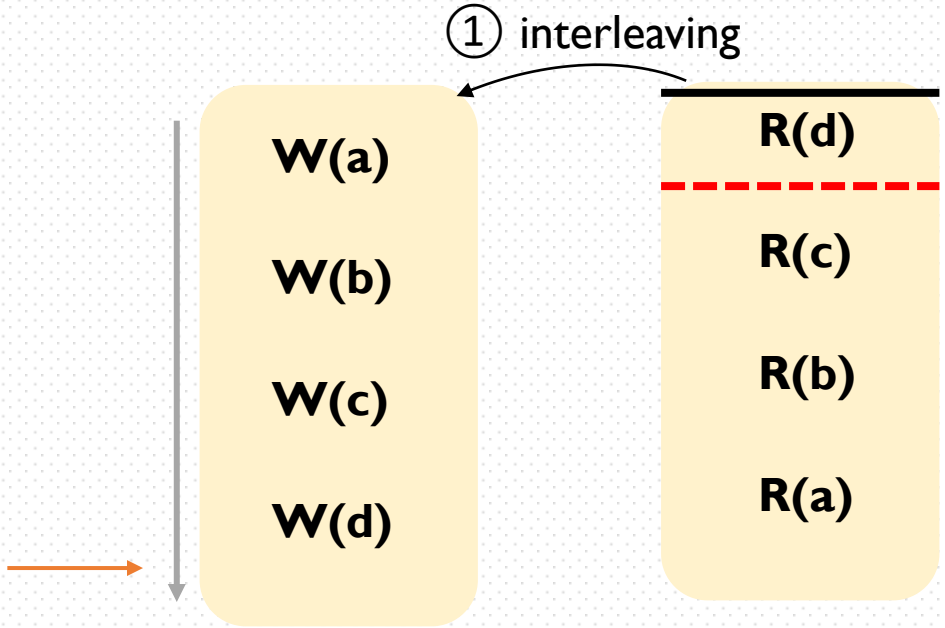
Ozz: Key idea

Assume a hypothetical memory barrier is missing

❖store-store, store-load



❖load-load



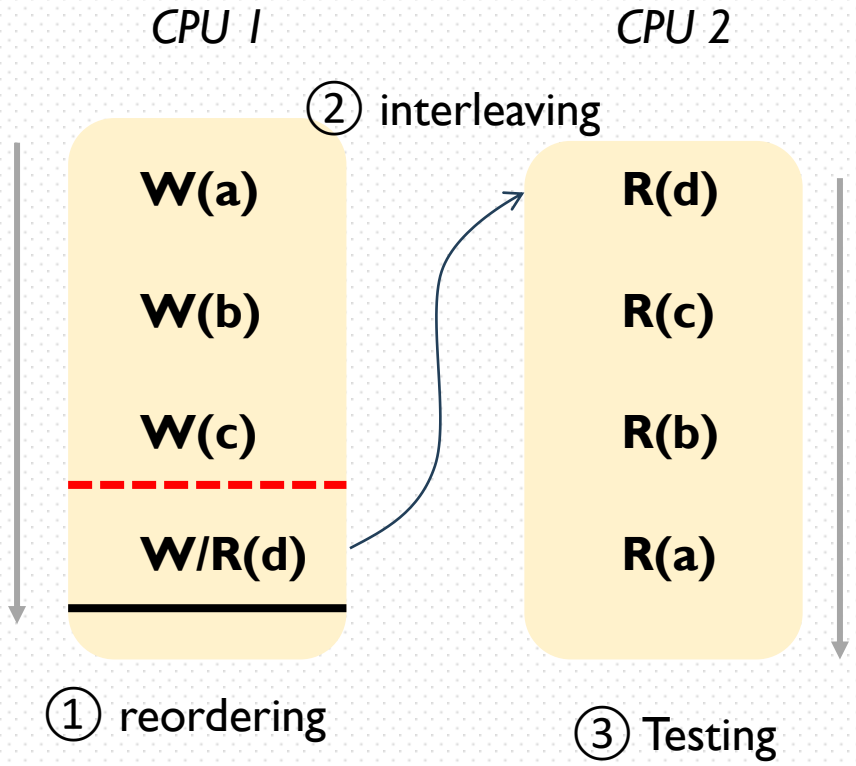
: real memory barrier

: hypothetical memory barrier

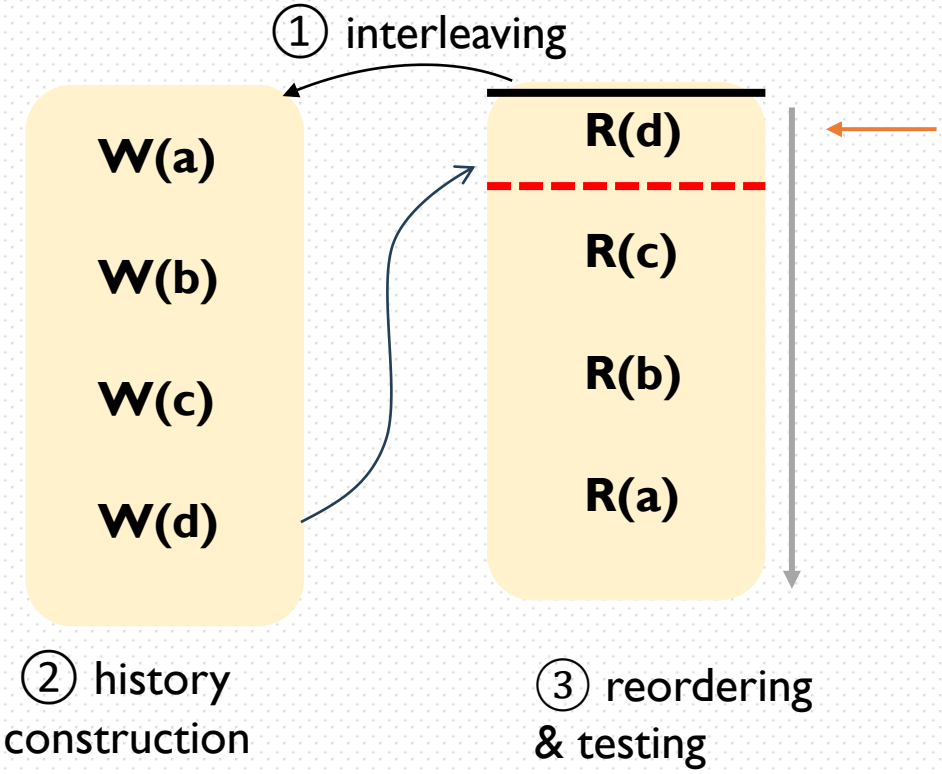
Ozz: Key idea

Assume a hypothetical memory barrier is missing

❖store-store, store-load



❖load-load



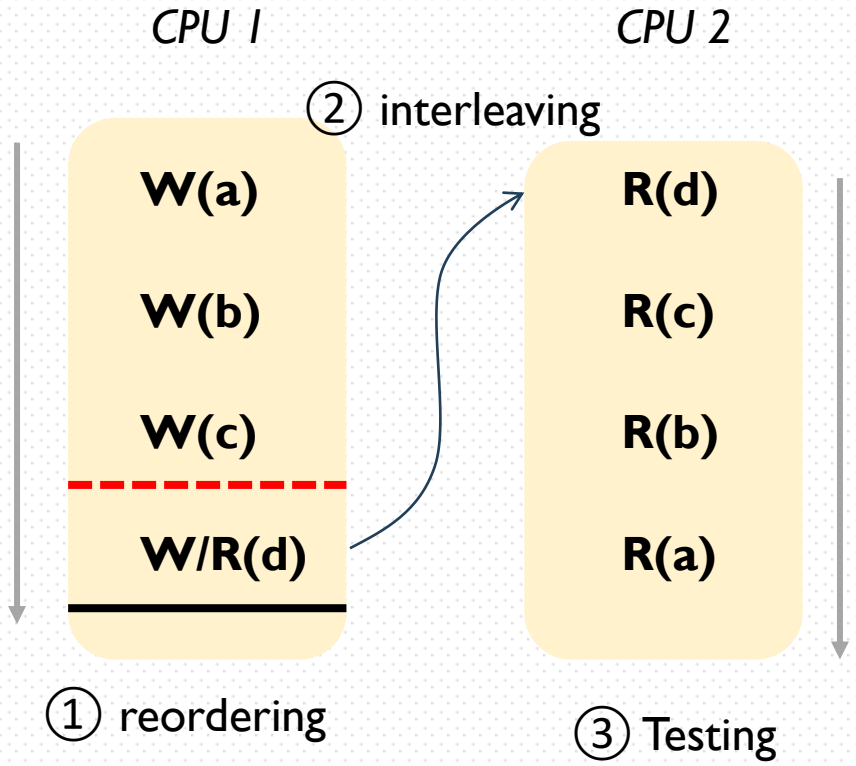
——— : real memory barrier

- - - - - : hypothetical memory barrier

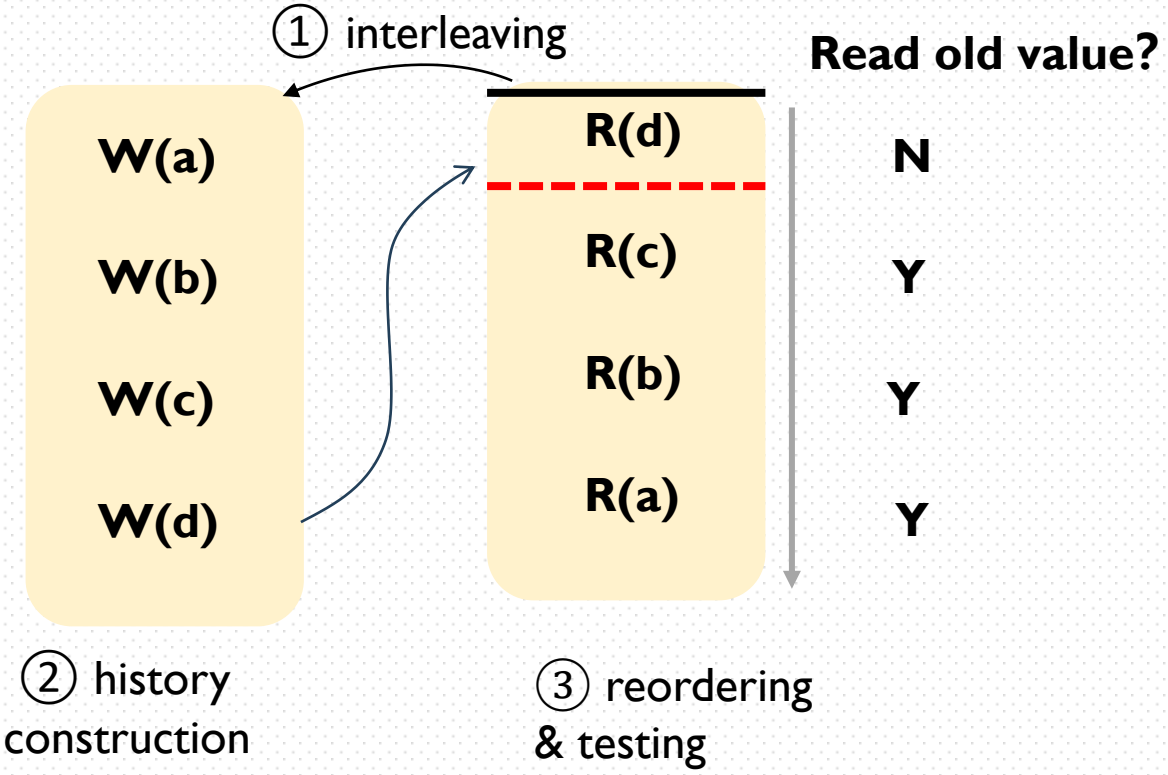
Ozz: Key idea

Assume a hypothetical memory barrier is missing

❖store-store, store-load



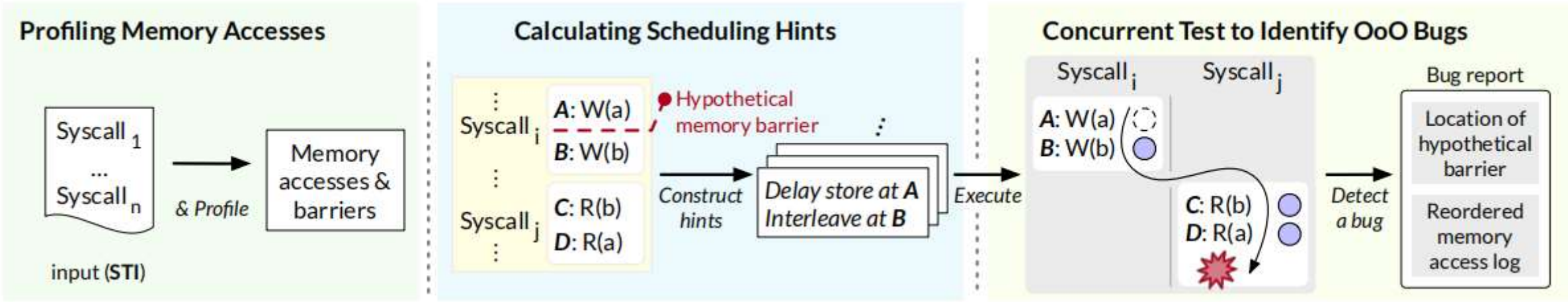
❖load-load



: real memory barrier

: hypothetical memory barrier

Ozz: Workflow

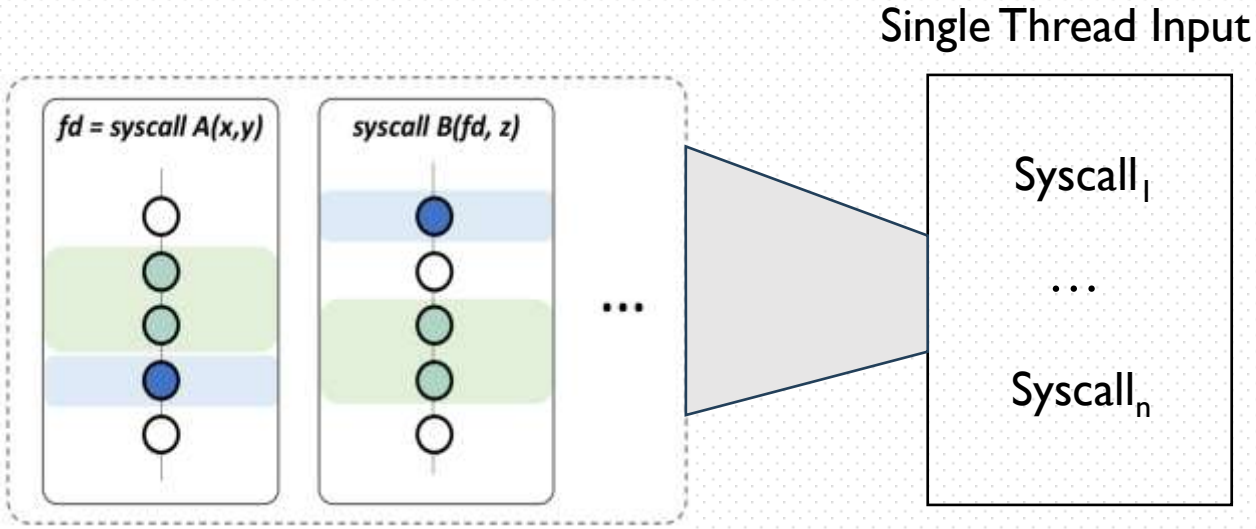


① Profiles memory accesses and memory barriers

② Calculate where is the hypothetical memory barrier, where doing the schedule, what memory access to reorder

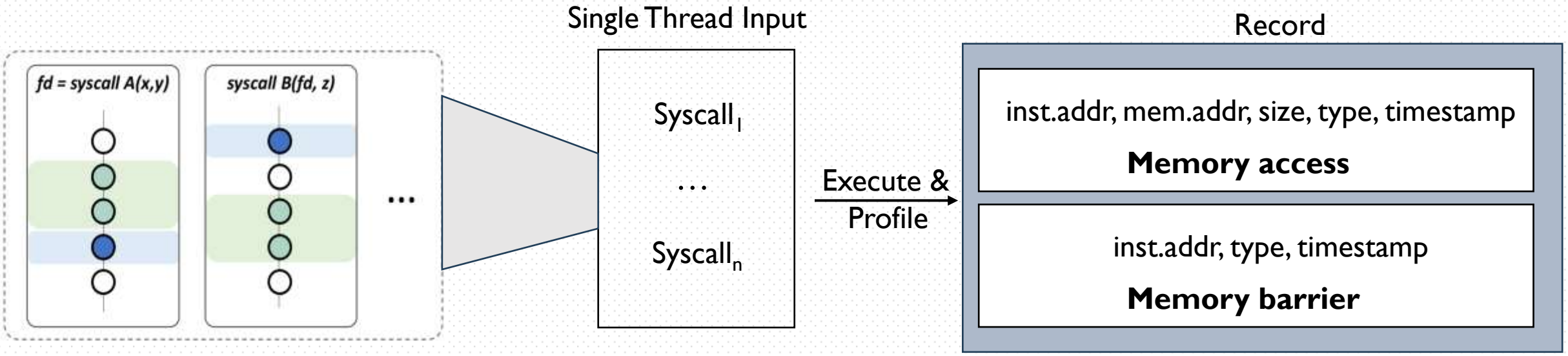
③ Use result of ② to test and observer Out-of-order bugs

Ozz: Profiling



- Ozz make STIs preserve necessary resource dependencies
- Ozz inserts callback function during LLVM compiler pass

Ozz: Profiling



- ❖Ozz make STIs preserve necessary resource dependencies
- ❖Ozz inserts callback function during LLVM compiler pass
- ❖Ozz do the execution and profiling and get the information

Ozz: Scheduling

Algorithm 1: Calculating scheduling hints

Input : S_i, S_j: Sequences of memory access and memory barriers executed by two system calls

Output : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

▷ `Step 1: Filter out memory accesses`

$S_i, S_j = filter\_out(S_i, S_j)$

for $k \in \{i, j\}$ **do**

  for $barrier\_type \in \{st, ld\}$ **do**

    ▷ `Step 2: Group memory accesses between memory barriers of the same type`

    $G_t, g = \emptyset, \emptyset$

    for $s \in S_k$ **do**

      if *s is a memory access* **then**

        $g = g \cup \{s\}$

      else if *s is a barrier &* *type of s = barrier_type* **then**

        $G_t = G_t \cup \{g\}$

        $g = \emptyset$

    ▷ `Step 3: Construct scheduling hints`

    $H_{ij} = \emptyset$

    for $g \in G_t$ **do**

      if $barrier\_type = st$ **then** $sched = g.last$

      else $sched = g.first$

      while $g \neq \emptyset$ **do**

        $h.sched = sched$

        $h.reorder = g \setminus sched$

        $H_{ij} = H_{ij} \cup \{h\}$

        if $barrier\_type = st$ **then** $g = g \setminus \{g.last\}$

        else $g = g \setminus \{g.first\}$

$H_{ij}.sort(key : len(h.reorder))$

return H_{ij}

Ozz: Scheduling

Algorithm 1: Calculating scheduling hints

Input : S_i, S_j: Sequences of memory access and memory barriers executed by two system calls

Output : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

```
▷ Step 1: Filter out memory accesses
1  S_i, S_j = filter_out(S_i, S_j)
2  for k ∈ {i, j} do
3      for barrier_type ∈ {st, ld} do
           ▷ Step 2: Group memory accesses between
           memory barriers of the same type
4          G_t, g = ∅, ∅
5          for s ∈ S_k do
6              if s is a memory access then
7                  g = g ∪ {s}
8              else if s is a barrier &
9              type of s = barrier_type then
10                 G_t = G_t ∪ {g}
11                 g = ∅
           ▷ Step 3: Construct scheduling hints
12         H_ij = ∅
13         for g ∈ G_t do
14             if barrier_type = st then sched = g.last
15             else sched = g.first
16             while g ≠ ∅ do
17                 h.sched = sched
18                 h.reorder = g \ sched
19                 H_ij = H_ij ∪ {h}
20                 if barrier_type = st then g = g \ {g.last}
21                 else g = g \ {g.first}
22 H_ij.sort(key : len(h.reorder))
23 return H_ij
```

Algorithm 2: Algorithm of the $filter\_out()$ function

Input : S_i, S_j: Sequences of memory accesses and memory barriers executed by two system calls.

Output : S'_i, S'_j: Sequences of memory accesses and memory barriers in which irrelevant memory accesses are filtered.

```
1  shared_mem = ∅
2  for (a_i, a_j) ∈ S_i × S_j do
3      if either a_i or a_j is not a memory access then
4          continue
5      o = shared_memory_location(a_i, a_j)
6      if o ≠ ∅ then
7          shared_mem = shared_mem ∪ {o}
8  for k ∈ {i, j} do
9      for a ∈ S_k do
10         if a is not a memory access then
11             continue
12         if a.addr ∉ shared_mem then
13             S_k = S_k \ {a}
14 S'_i, S'_j = S_i, S_j
15 return S'_i, S'_j
```

① Ozz finds out memory locations shared between two memory accesses

Ozz: Scheduling

Algorithm 1: Calculating scheduling hints

Input : S_i, S_j: Sequences of memory access and memory barriers executed by two system calls

Output : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

```
▷ Step 1: Filter out memory accesses
1  S_i, S_j = filter_out(S_i, S_j)
2  for k ∈ {i, j} do
3      for barrier_type ∈ {st, ld} do
           ▷ Step 2: Group memory accesses between
           memory barriers of the same type
4          G_t, g = ∅, ∅
5          for s ∈ S_k do
6              if s is a memory access then
7                  g = g ∪ {s}
8              else if s is a barrier &
9              type of s = barrier_type then
10                 G_t = G_t ∪ {g}
11                 g = ∅
           ▷ Step 3: Construct scheduling hints
12         H_ij = ∅
13         for g ∈ G_t do
14             if barrier_type = st then sched = g.last
15             else sched = g.first
16             while g ≠ ∅ do
17                 h.sched = sched
18                 h.reorder = g \ sched
19                 H_ij = H_ij ∪ {h}
20                 if barrier_type = st then g = g \ {g.last}
21                 else g = g \ {g.first}
22 H_ij.sort(key : len(h.reorder))
23 return H_ij
```

Algorithm 2: Algorithm of the *filter_out*() function

Input : S_i, S_j: Sequences of memory accesses and memory barriers executed by two system calls.

Output : S'_i, S'_j: Sequences of memory accesses and memory barriers in which irrelevant memory accesses are filtered.

```
1  shared_mem = ∅
2  for (a_i, a_j) ∈ S_i × S_j do
3      if either a_i or a_j is not a memory access then
4          continue
5      o = shared_memory_location(a_i, a_j)
6      if o ≠ ∅ then
7          shared_mem = shared_mem ∪ {o}
8  for k ∈ {i, j} do
9      for a ∈ S_k do
10         if a is not a memory access then
11             continue
12         if a.addr ∉ shared_mem then
13             S_k = S_k \ {a}
14 S'_i, S'_j = S_i, S_j
15 return S'_i, S'_j
```

① Ozz finds out memory locations shared between two memory accesses

② Ozz excludes memory accesses don't visit *shared_mem*

Ozz: Scheduling

Algorithm 1: Calculating scheduling hints

Input : S_i, S_j: Sequences of memory access and memory barriers executed by two system calls

Output : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

```
▷ Step 1: Filter out memory accesses
1  S_i, S_j = filter_out(S_i, S_j)
2  for k ∈ {i, j} do
3      for barrier_type ∈ {st, ld} do
           ▷ Step 2: Group memory accesses between
           memory barriers of the same type
4          G_t, g = ∅, ∅
5          for s ∈ S_k do
6              if s is a memory access then
7                  g = g ∪ {s}
8              else if s is a barrier &
9              type of s = barrier_type then
10                 G_t = G_t ∪ {g}
11                 g = ∅
           ▷ Step 3: Construct scheduling hints
12         H_ij = ∅
13         for g ∈ G_t do
14             if barrier_type = st then sched = g.last
15             else sched = g.first
16             while g ≠ ∅ do
17                 h.sched = sched
18                 h.reorder = g \ sched
19                 H_ij = H_ij ∪ {h}
20                 if barrier_type = st then g = g \ {g.last}
21                 else g = g \ {g.first}
22 H_ij.sort(key : len(h.reorder))
23 return H_ij
```

$$k = i$$
$$barrier\_type = st$$

Initial: $G_t, g = \emptyset, \emptyset$

st_barrier1
memory1
memory2
memory3
ld_barrier1
memory4
memory5
st_barrier2
memory6
memory7
st_barrier3

Syscall S_i

Ozz: Scheduling

Algorithm 1: Calculating scheduling hints

Input : S_i, S_j: Sequences of memory access and memory barriers executed by two system calls

Output : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

```
▷ Step 1: Filter out memory accesses
1  S_i, S_j = filter_out(S_i, S_j)
2  for k ∈ {i, j} do
3      for barrier_type ∈ {st, ld} do
           ▷ Step 2: Group memory accesses between
           memory barriers of the same type
4          G_t, g = ∅, ∅
5          for s ∈ S_k do
6              if s is a memory access then
7                  g = g ∪ {s}
8              else if s is a barrier &
9              type of s = barrier_type then
10                 G_t = G_t ∪ {g}
11                 g = ∅
           ▷ Step 3: Construct scheduling hints
12         H_ij = ∅
13         for g ∈ G_t do
14             if barrier_type = st then sched = g.last
15             else sched = g.first
16             while g ≠ ∅ do
17                 h.sched = sched
18                 h.reorder = g \ sched
19                 H_ij = H_ij ∪ {h}
20                 if barrier_type = st then g = g \ {g.last}
21                 else g = g \ {g.first}
22 H_ij.sort(key : len(h.reorder))
23 return H_ij
```

$$k = i$$
$$barrier\_type = st$$

Initial: $G_t, g = \emptyset, \emptyset$

$$G_t = \emptyset$$
$$g = \{m1, m2, m3\}$$

st_barrier1
memory1
memory2
memory3
ld_barrier1
memory4
memory5
st_barrier2
memory6
memory7
st_barrier3

Syscall S_i

Ozz: Scheduling

Algorithm 1: Calculating scheduling hints

Input : S_i, S_j: Sequences of memory access and memory barriers executed by two system calls

Output : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

```
▷ Step 1: Filter out memory accesses
1  S_i, S_j = filter_out(S_i, S_j)
2  for k ∈ {i, j} do
3      for barrier_type ∈ {st, ld} do
           ▷ Step 2: Group memory accesses between
           memory barriers of the same type
4          G_t, g = ∅, ∅
5          for s ∈ S_k do
6              if s is a memory access then
7                  g = g ∪ {s}
8              else if s is a barrier &
9              type of s = barrier_type then
10                 G_t = G_t ∪ {g}
11                 g = ∅
           ▷ Step 3: Construct scheduling hints
12         H_ij = ∅
13         for g ∈ G_t do
14             if barrier_type = st then sched = g.last
15             else sched = g.first
16             while g ≠ ∅ do
17                 h.sched = sched
18                 h.reorder = g \ sched
19                 H_ij = H_ij ∪ {h}
20                 if barrier_type = st then g = g \ {g.last}
21                 else g = g \ {g.first}
22 H_ij.sort(key : len(h.reorder))
23 return H_ij
```

$$k = i$$
$$barrier\_type = st$$

Initial: $G_t, g = \emptyset, \emptyset$

$$G_t = \emptyset$$
$$g = \{m1, m2, m3\}$$

$$G_t = \{g1\}$$
$$g1 = \{m1, m2, m3, m4, m5\}$$
$$g = \emptyset$$

st_barrier1
memory1
memory2
memory3
ld_barrier1
memory4
memory5
st_barrier2
memory6
memory7
st_barrier3

Syscall S_i

Ozz: Scheduling

Algorithm 1: Calculating scheduling hints

Input : S_i, S_j: Sequences of memory access and memory barriers executed by two system calls

Output : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

```
▷ Step 1: Filter out memory accesses
S_i, S_j = filter_out(S_i, S_j)
for k ∈ {i, j} do
    for barrier_type ∈ {st, ld} do
        ▷ Step 2: Group memory accesses between
        memory barriers of the same type
        G_t, g = ∅, ∅
        for s ∈ S_k do
            if s is a memory access then
                g = g ∪ {s}
            else if s is a barrier &
            type of s = barrier_type then
                G_t = G_t ∪ {g}
                g = ∅
        ▷ Step 3: Construct scheduling hints
        H_ij = ∅
        for g ∈ G_t do
            if barrier_type = st then sched = g.last
            else sched = g.first
            while g ≠ ∅ do
                h.sched = sched
                h.reorder = g \ sched
                H_ij = H_ij ∪ {h}
                if barrier_type = st then g = g \ {g.last}
                else g = g \ {g.first}
H_ij.sort(key : len(h.reorder))
return H_ij
```

$$k = i$$
$$barrier\_type = st$$

Initial: $G_t, g = \emptyset, \emptyset$

$$G_t = \emptyset$$
$$g = \{m1, m2, m3\}$$

$$G_t = \{g1\}$$
$$g1 = \{m1, m2, m3, m4, m5\}$$
$$g = \emptyset$$

$$G_t = \{g1, g2\}$$
$$g1 = \{m1, m2, m3, m4, m5\}$$
$$g2 = \{m6, m7\}$$
$$g = \emptyset$$

st_barrier1
memory1
memory2
memory3
ld_barrier1
memory4
memory5
st_barrier2
memory6
memory7
st_barrier3

Syscall S_i

Ozz: Scheduling

Algorithm 1: Calculating scheduling hints

Input : S_i, S_j: Sequences of memory access and memory barriers executed by two system calls

Output : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

▷ Step 1: Filter out memory accesses

1 $S_i, S_j = filter\_out(S_i, S_j)$
2 **for** $k \in \{i, j\}$ **do**
3 **for** $barrier\_type \in \{st, ld\}$ **do**
 ▷ Step 2: Group memory accesses between memory barriers of the same type
4 $G_t, g = \emptyset, \emptyset$
5 **for** $s \in S_k$ **do**
6 **if** *s is a memory access* **then**
7 $g = g \cup \{s\}$
8 **else if** *s is a barrier &*
9 *type of s* $= barrier\_type$ **then**
10 $G_t = G_t \cup \{g\}$
11 $g = \emptyset$
 ▷ Step 3: Construct scheduling hints
12 $H_{ij} = \emptyset$
13 **for** $g \in G_t$ **do**
14 **if** $barrier\_type = st$ **then** $sched = g.last$
15 **else** $sched = g.first$
16 **while** $g \neq \emptyset$ **do**
17 $h.sched = sched$
18 $h.reorder = g \setminus sched$
19 $H_{ij} = H_{ij} \cup \{h\}$
20 **if** $barrier\_type = st$ **then** $g = g \setminus \{g.last\}$
21 **else** $g = g \setminus \{g.first\}$
22 $H_{ij}.sort(key : len(h.reorder))$
23 **return** H_{ij}

$$barrier\_type = st$$
$$g = \{m1, m2, m3, m4, m5\}$$
$$sched = m5$$

First Test:

$$H_{ij} = H_{ij} \cup \{h1\}$$
$$h1.shed = m5$$
$$h1.reorder = \{m1, m2, m3, m4\}$$

st_barrier1
memory1
memory2
memory3
ld_barrier1
memory4
---
memory5
st_barrier2
memory6
memory7
st_barrier3

Syscall S_i

Ozz: Scheduling

Algorithm 1: Calculating scheduling hints

Input : S_i, S_j: Sequences of memory access and memory barriers executed by two system calls

Output : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

```
▷ Step 1: Filter out memory accesses
1  S_i, S_j = filter_out(S_i, S_j)
2  for k ∈ {i, j} do
3      for barrier_type ∈ {st, ld} do
           ▷ Step 2: Group memory accesses between
           memory barriers of the same type
4          G_t, g = ∅, ∅
5          for s ∈ S_k do
6              if s is a memory access then
7                  g = g ∪ {s}
8              else if s is a barrier &
9              type of s = barrier_type then
10                 G_t = G_t ∪ {g}
11                 g = ∅
           ▷ Step 3: Construct scheduling hints
12         H_ij = ∅
13         for g ∈ G_t do
14             if barrier_type = st then sched = g.last
15             else sched = g.first
16             while g ≠ ∅ do
17                 h.sched = sched
18                 h.reorder = g \ sched
19                 H_ij = H_ij ∪ {h}
20                 if barrier_type = st then g = g \ {g.last}
21                 else g = g \ {g.first}
22 H_ij.sort(key : len(h.reorder))
23 return H_ij
```

$$barrier\_type = st$$
$$g = \{m1, m2, m3, m4, m5\}$$
$$sched = m5$$

First Test:

$$H_{ij} = H_{ij} \cup \{h1\}$$
$$h1.shed = m5$$
$$h1.reorder = \{m1, m2, m3, m4\}$$

Second Test:

$$H_{ij} = H_{ij} \cup \{h1\} \cup \{h2\}$$
$$h2.shed = m5$$
$$h2.reorder = \{m1, m2, m3\}$$

...

st_barrier1

memory1

memory2

memory3

ld_barrier1

---

memory4

memory5

st_barrier2

memory6

memory7

st_barrier3

Syscall S_i

Ozz: Scheduling

Algorithm 1: Calculating scheduling hints

Input : S_i, S_j: Sequences of memory access and memory barriers executed by two system calls

Output : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

▷ Step 1: Filter out memory accesses

1 $S_i, S_j = filter\_out(S_i, S_j)$
2 **for** $k \in \{i, j\}$ **do**
3 **for** $barrier\_type \in \{st, ld\}$ **do**
 ▷ Step 2: Group memory accesses between memory barriers of the same type
4 $G_t, g = \emptyset, \emptyset$
5 **for** $s \in S_k$ **do**
6 **if** *s is a memory access* **then**
7 $g = g \cup \{s\}$
8 **else if** *s is a barrier &*
9 *type of s* $= barrier\_type$ **then**
10 $G_t = G_t \cup \{g\}$
11 $g = \emptyset$
 ▷ Step 3: Construct scheduling hints
12 $H_{ij} = \emptyset$
13 **for** $g \in G_t$ **do**
14 **if** $barrier\_type = st$ **then** $sched = g.last$
15 **else** $sched = g.first$
16 **while** $g \neq \emptyset$ **do**
17 $h.sched = sched$
18 $h.reorder = g \setminus sched$
19 $H_{ij} = H_{ij} \cup \{h\}$
20 **if** $barrier\_type = st$ **then** $g = g \setminus \{g.last\}$
21 **else** $g = g \setminus \{g.first\}$
22 $H_{ij}.sort(key : len(h.reorder))$
23 **return** H_{ij}

$$barrier\_type = ld$$
$$g = \{m1, m2, m3, m4, m5\}$$

$$sched = m1$$

First Test:

$$H_{ij} = H_{ij} \cup \{h1\}$$
$$h1.sched = m1$$
$$h1.reorder = \{m2, m3, m4, m5\}$$

ld_barrier1
memory1
---
memory2
memory3
st_barrier1
memory4
memory5
ld_barrier2
memory6
memory7
st_barrier3

Syscall S_i

Ozz: Scheduling

Algorithm 1: Calculating scheduling hints

Input : S_i, S_j: Sequences of memory access and memory barriers executed by two system calls

Output : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

```
▷ Step 1: Filter out memory accesses
S_i, S_j = filter_out(S_i, S_j)
for k ∈ {i, j} do
    for barrier_type ∈ {st, ld} do
        ▷ Step 2: Group memory accesses between
        memory barriers of the same type
        G_t, g = ∅, ∅
        for s ∈ S_k do
            if s is a memory access then
                g = g ∪ {s}
            else if s is a barrier &
            type of s = barrier_type then
                G_t = G_t ∪ {g}
                g = ∅
        ▷ Step 3: Construct scheduling hints
        H_ij = ∅
        for g ∈ G_t do
            if barrier_type = st then sched = g.last
            else sched = g.first
            while g ≠ ∅ do
                h.sched = sched
                h.reorder = g \ sched
                H_ij = H_ij ∪ {h}
                if barrier_type = st then g = g \ {g.last}
                else g = g \ {g.first}
    H_ij.sort(key : len(h.reorder))
return H_ij
```

$$barrier\_type = ld$$
$$g = \{m1, m2, m3, m4, m5\}$$

$$sched = m1$$

First Test:

$$H_{ij} = H_{ij} \cup \{h1\}$$
$$h1.shed = m1$$
$$h1.reorder = \{m2, m3, m4, m5\}$$

Second Test:

$$H_{ij} = H_{ij} \cup \{h1\} \cup \{h2\}$$
$$h2.shed = m1$$
$$h2.reorder = \{m3, m4, m5\}$$

...

ld_barrier1
memory1
memory2
---
memory3
st_barrier1
memory4
memory5
ld_barrier2
memory6
memory7
st_barrier3

Syscall S_i

Ozz: Scheduling

Algorithm 1: Calculating scheduling hints

Input : S_i, S_j: Sequences of memory access and memory barriers executed by two system calls

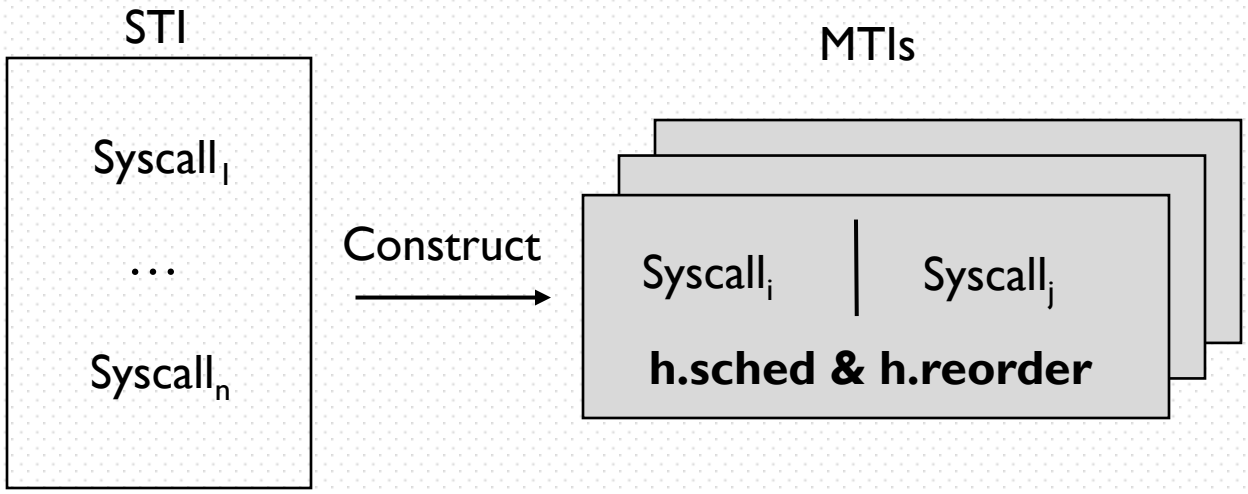
Output : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

▷ `Step 1: Filter out memory accesses`

$S_i, S_j = filter\_out(S_i, S_j)$
for $k \in \{i, j\}$ **do**
  for $barrier\_type \in \{st, ld\}$ **do**
    ▷ `Step 2: Group memory accesses between memory barriers of the same type`
    $G_t, g = \emptyset, \emptyset$
    for $s \in S_k$ **do**
      if *s is a memory access* **then**
        $g = g \cup \{s\}$
      else if *s is a barrier &* *type of s = barrier_type* **then**
        $G_t = G_t \cup \{g\}$
        $g = \emptyset$
    ▷ `Step 3: Construct scheduling hints`
    $H_{ij} = \emptyset$
    for $g \in G_t$ **do**
      if $barrier\_type = st$ **then** $sched = g.last$
      else $sched = g.first$
      while $g \neq \emptyset$ **do**
        $h.sched = sched$
        $h.reorder = g \setminus sched$
        $H_{ij} = H_{ij} \cup \{h\}$
        if $barrier\_type = st$ **then** $g = g \setminus \{g.last\}$
        else $g = g \setminus \{g.first\}$
$H_{ij}.sort(key : len(h.reorder))$
return H_{ij}

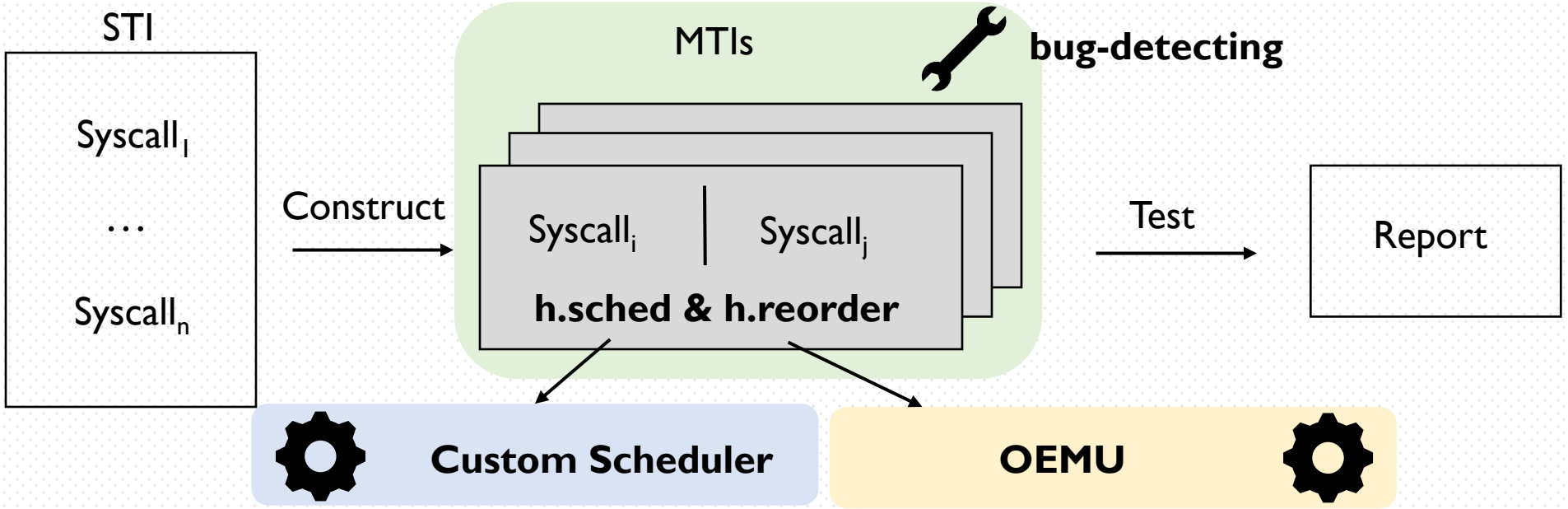
Greedy: When deviate more, bug arise with high possible.

Ozz: Test



- **Each STI is translated into multiple MTIs**
  - **MTI have the same set of syscalls with the STI**
  - **MTIs are annotated with a pair of syscalls to run concurrently and schedule hints**

Ozz: Test



- **Each STI is translated into multiple MTIs**
  - **MTI have the same set of syscalls with the STI**
  - **MTIs are annotated with a pair of syscalls to run concurrently and schedule hints**
- **Ozz run MTIs monitor bugs**
  - **Ozz leverage bug-detecting oracles during runtime**
  - **Report tells the reordered accesses and hypothetical memory barrier**

Setup

- **Hardware**
  - **Two-sockets 32 phsical-core Intel Xeon CPU E5-2683 v4 operating at 2.1GHz**
  - **512GB of RAM**
- **Host operating system**
  - **Ubuntu 20.04.4 kernel 5.4.143**
- **OZZ**
  - **based on *SYZKALLER* (**SOTA fuzzer developed by Google**)**
  - **32 VMs each is equipped with 4 vCPUS and 8G memory**
  - **Kernel :6.5-rc6 to 6.8 (***SYZKALLER* use the same kernel**)**

Evaluation: Real-world OoO bug

ID	Kernel version	Subsystem	Summary	Status
Bug #1	v6.7-rc8	RDS	KASAN: slab-out-of-bounds Read in rds_loop_xmit	Fixed
Bug #2	v6.5-rc6	watchqueue	BUG: unable to handle kernel NULL pointer dereference in _find_first_bit	Reported
Bug #3	v6.5-rc6	VMCI	general protection fault in add_wait_queue	Reported
Bug #4	v6.6-rc2	XDP	BUG: unable to handle kernel NULL pointer dereference in xsk_poll	Fixed
Bug #5	v6.6-rc2	TLS	BUG: unable to handle kernel NULL pointer dereference in tls_getsockopt	Fixed
Bug #6	v6.7-rc8	BPF	BUG: unable to handle kernel NULL pointer dereference in sk_psock_verdict_data_ready	Fixed
Bug #7	v6.5-rc7	XDP	BUG: unable to handle kernel NULL pointer dereference in xsk_generic_xmit	Fixed
Bug #8	v6.7-rc8	SMC	BUG: unable to handle kernel NULL pointer dereference in connect	Confirmed
Bug #9	v6.7-rc2	TLS	BUG: unable to handle kernel NULL pointer dereference in tls_setsockopt	Fixed
Bug #10	v6.8-rc1	SMC	KASAN: null-ptr-deref Write in fput	Confirmed
Bug #11	v6.8	GSM	BUG: unable to handle kernel NULL pointer dereference in gsm_dlci_config	Confirmed

- Ozz discovers 61 unique crashed, and 11 new OoO bugs
- *SYZKALLER* is impractical to identify
  - x86-64 does not reorder s-s or l-l
  - TCG does not reorder memory access

Evaluation: Real-world OoO bug

❑Improper adoption of memory barrier

```
1  /******* Thread A *******/
2  /* net/tls/tls_main.c */
3  int tls_init() {
4    ctx = kzalloc();
5    sk->data = ctx;
6    ctx->sk_proto =
7      READ_ONCE(sk->sk_prot);
8  + smp_wmb();
9    WRITE_ONCE(sk->sk_prot,
10                &tls_prots);
11 }
12
13 struct proto_ops tls_prots = {
14  .setsockopt = tls_setsockopt,
15 };
```

④ ① (④: lines 6–7; ①: line 9)

```
16 /******* Thread B *******/
17 /* net/core/socket.c */
18 int sock_common_setsockopt() {
19   struct sock *sk = sock->sk;
20   return READ_ONCE(sk->sk_prot)
21            ->setsockopt(sk);
22 }
23
24 /* net/tls/tls_main.c */
25 int tls_setsockopt() {
26   struct tls_context *ctx =
27     sk->data;
28   return ctx->sk_proto
29            ->setsockopt(sk);
30 }
```

② ③ (②: line 20; ③: line 28)

ctx->sk_proto() **uninitialized !**

Evaluation: Real-world OoO bug

❑Improper adoption of memory barrier

```
1  /******* Thread A *******/
2  /* net/tls/tls_main.c */
3  int tls_init() {
4    ctx = kzalloc();
5    sk->data = ctx;
6    ctx->sk_proto =
7      READ_ONCE(sk->sk_prot);
8  + smp_wmb();
9    WRITE_ONCE(sk->sk_prot,
10             &tls_prots);
11 }
12
13 struct proto_ops tls_prots = {
14  .setsockopt = tls_setsockopt,
15 };
```

```
16 /******* Thread B *******/
17 /* net/core/socket.c */
18 int sock_common_setsockopt() {
19   struct sock *sk = sock->sk;
20   return READ_ONCE(sk->sk_prot)
21            ->setsockopt(sk);
22 }
23
24 /* net/tls/tls_main.c */
25 int tls_setsockopt() {
26   struct tls_context *ctx =
27     sk->data;
28   return ctx->sk_proto
29            ->setsockopt(sk);
30 }
```

Developers caught the data race (load/store tearing)

However, these function suppress a data race detector from reporting

Evaluation: Real-world OoO bug

- Incorrect customized lock

```
1  /* net/rds/send.c */
2  int acquire_in_xmit()
3  {
4    int acquired =
5      !test_and_set_bit
6        (IN_XMIT, &cp_flags);
7    return acquired;
8  }
```

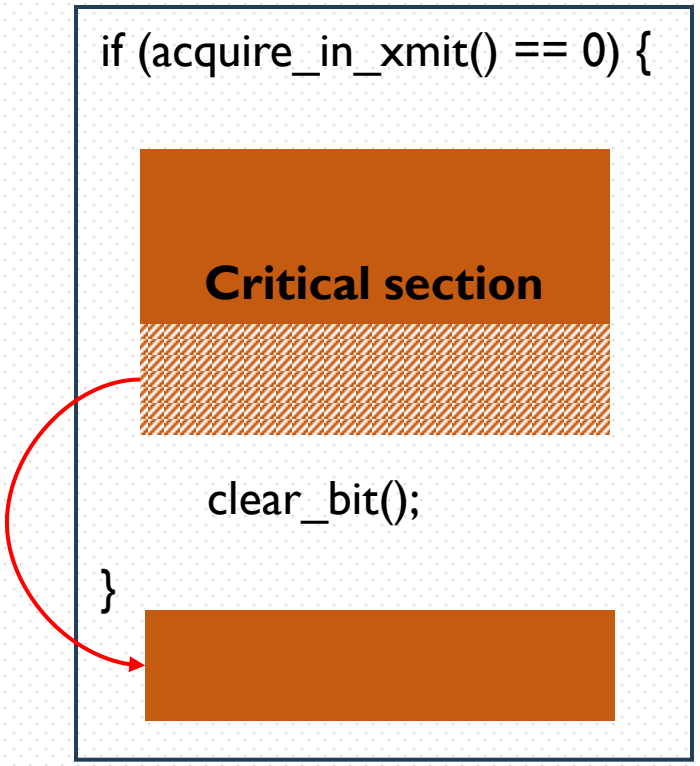
```
9  /* net/rds/send.c */
10 void release_in_xmit()
11 {
12 - clear_bit(IN_XMIT, &cp_flags);
13 + clear_bit_unlock(IN_XMIT,
14                    &cp_flags);
15 }
16
```

Evaluation: Real-world OoO bug

- Incorrect customized lock

```
1  /* net/rds/send.c */
2  int acquire_in_xmit()
3  {
4    int acquired =
5      !test_and_set_bit
6        (IN_XMIT, &cp_flags);
7    return acquired;
8  }
```

```
9  /* net/rds/send.c */
10 void release_in_xmit()
11 {
12 - clear_bit(IN_XMIT, &cp_flags);
13 + clear_bit_unlock(IN_XMIT,
14                    &cp_flags);
15 }
16
```



Evaluation: Known OoO bug

ID	Subsystem	Version	Reproduced?	# of tests	Type
#1 [120]	vlan	5.12-rc7	✓	342	S-S
#2 [31]	watchqueue	5.17-rc7	✓	23	S-S
#3 [103]	xsk	4.17-rc4	✓	47	S-S
#4 [101]	xsk	5.3-rc3	✓	12	S-S
#5 [30]	fs	6.1-rc1	✓	17	L-L
#6 [60]	sbitmap	5.1-rc1	×	-	S-S
#7 [78]	nbd	6.7-rc1	✓	17	L-L
#8 [50]	tls	6.7-rc1	✓*	42	S-S
#9 [106]	unix	5.0-rc7	✓	23	L-L

← Detect by a wrong value, not crash

- **8 OoO bugs can be reproduced, running tens of test run on average**
- **#6 is caused by thread migration**
  - **Ozz pins concurrent threads on specific CPUs**

Evaluation: Performance overhead

Benchmark suit : *LMBench* (evaluating various OS operations)

Tests	Linux (μs)	Linux w/ OEMU (μs)	Overhead
null	1.74	43.3	24.9×
stat	75.64	859.6	11.4×
open/close	128	1369.2	10.7×
File create	403.3	5623.5	13.9×
File delete	207.8	3363	16.2×
ctxsw 2p/0k	23.8	71.5	3.0×
pipe	59.3	610.1	10.3×
unix	173.8	2567.6	14.8×
fork	7590	145.6k	19.2×
mmap	133.8k	7896.1k	59.0×

- **Developers can opt to selectively enable OEMU (lockless implementations)**
- **OMEU has 7.9x lower throughout compared to *SYZKALLER*** (0.92 test/s **VS** 7.33 test/s)
  - **OMEU can control Out-of-order execution**
  - **Save the cost of buying new machines (ARM)**

Evaluation: Compared with OFence

- **OFence**
  - **Predifine likely-buggy patterns** (mamory barriers are not in pair)
  - **Using *static* pattern matching analysis**

Evaluation: Compared with OFence

❑OFence

- ❖**Predifine likely-buggy patterns** (mamory barriers are not in pair)
- ❖**Using *static* pattern matching analysis**

❑Result

- ❖**Ozz (**and *SYZKALLER*) **is limited in generating inputs of bugs found by OFence**
  - ➢ A submodule requires specific hardware to run, inhibiting dynamic testing in such submodules
- ❖**Only 3 out of 11 OoO bugs found by Ozz fit pattern of OFence**

Conclusion

- **Pros:**
  - The problem of Out-of-order execution is interesting
  - Change the order of memory access to emulate Out-of-order

- **Cons:**
  - Assum work in two thread and only one thread do Out-of-order execution
  - Can't deal with *load-store* reorder
  - Can't tell what type memory barrier is the best to insert

Q&A

Ozz: Scheduling

Algorithm 1: Calculating scheduling hints

Input : S_i, S_j: Sequences of memory access and memory barriers executed by two system calls

Output : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

```
▷ Step 1: Filter out memory accesses
1  S_i, S_j = filter_out(S_i, S_j)
2  for k ∈ {i, j} do
3      for barrier_type ∈ {st, ld} do
           ▷ Step 2: Group memory accesses between
           memory barriers of the same type
4          G_t, g = ∅, ∅
5          for s ∈ S_k do
6              if s is a memory access then
7                  g = g ∪ {s}
8              else if s is a barrier &
               type of s = barrier_type then
9                  G_t = G_t ∪ {g}
10                 g = ∅
           ▷ Step 3: Construct scheduling hints
12         H_ij = ∅
13         for g ∈ G_t do
14             if barrier_type = st then sched = g.last
15             else sched = g.first
16             while g ≠ ∅ do
17                 h.sched = sched
18                 h.reorder = g \ sched
19                 H_ij = H_ij ∪ {h}
20                 if barrier_type = st then g = g \ {g.last}
21                 else g = g \ {g.first}
22 H_ij.sort(key : len(h.reorder))
23 return H_ij
```

Algorithm 2: Algorithm of the *filter_out*() function

Input : S_i, S_j: Sequences of memory accesses and memory barriers executed by two system calls.

Output : S'_i, S'_j: Sequences of memory accesses and memory barriers in which irrelevant memory accesses are filtered.

```
1  shared_mem = ∅
2  for (a_i, a_j) ∈ S_i × S_j do
3      if either a_i or a_j is not a memory access then
4          continue
5      o = shared_memory_location(a_i, a_j)
6      if o ≠ ∅ then
7          shared_mem = shared_mem ∪ {o}
8  for k ∈ {i, j} do
9      for a ∈ S_k do
10         if a is not a memory access then
11             continue
12         if a.addr ∉ shared_mem then
13             S_k = S_k \ {a}
14 S'_i, S'_j = S_i, S_j
15 return S'_i, S'_j
```

① Ozz finds out memory locations shared between two memory accesses

② Ozz excludes memory accesses don't visit *shared_mem*