Krace: Data Race Fuzzing

for Kernel File Systems

Meng Xu, Sanidhya Kashyap, Hanqing Zhao, Taesoo Kim

May 1, 2020

Let's talk about data race

Definition: Two memory accesses from different threads such that

1. They access the same memory location
2. At least one of them is a write operation
3. They may interleave without restrictions (i.e., locks, orderings, etc)

The classic race condition example

`counter = 0`

Thread 1	Thread 2

`for(i=0; i<50000; i++) {`
`  counter++;`
`}`
	`for(i=0; i<50000; i++) {`
	`  counter++;`
	`}`

What is the value of `counter` when both threads terminate?

Any value between 50,000 to 100,000

The classic race condition example

`counter = 0`

Thread 1	Thread 2

`for(i=0; i<50000; i++) {`
`  lock(mutex);`
`  counter++;`
`  unlock(mutex);`
`}`
	`for(i=0; i<50000; i++) {`
	`  lock(mutex);`
	`  counter++;`
	`  unlock(mutex);`
	`}`

What is the value of `counter` when both threads terminate?

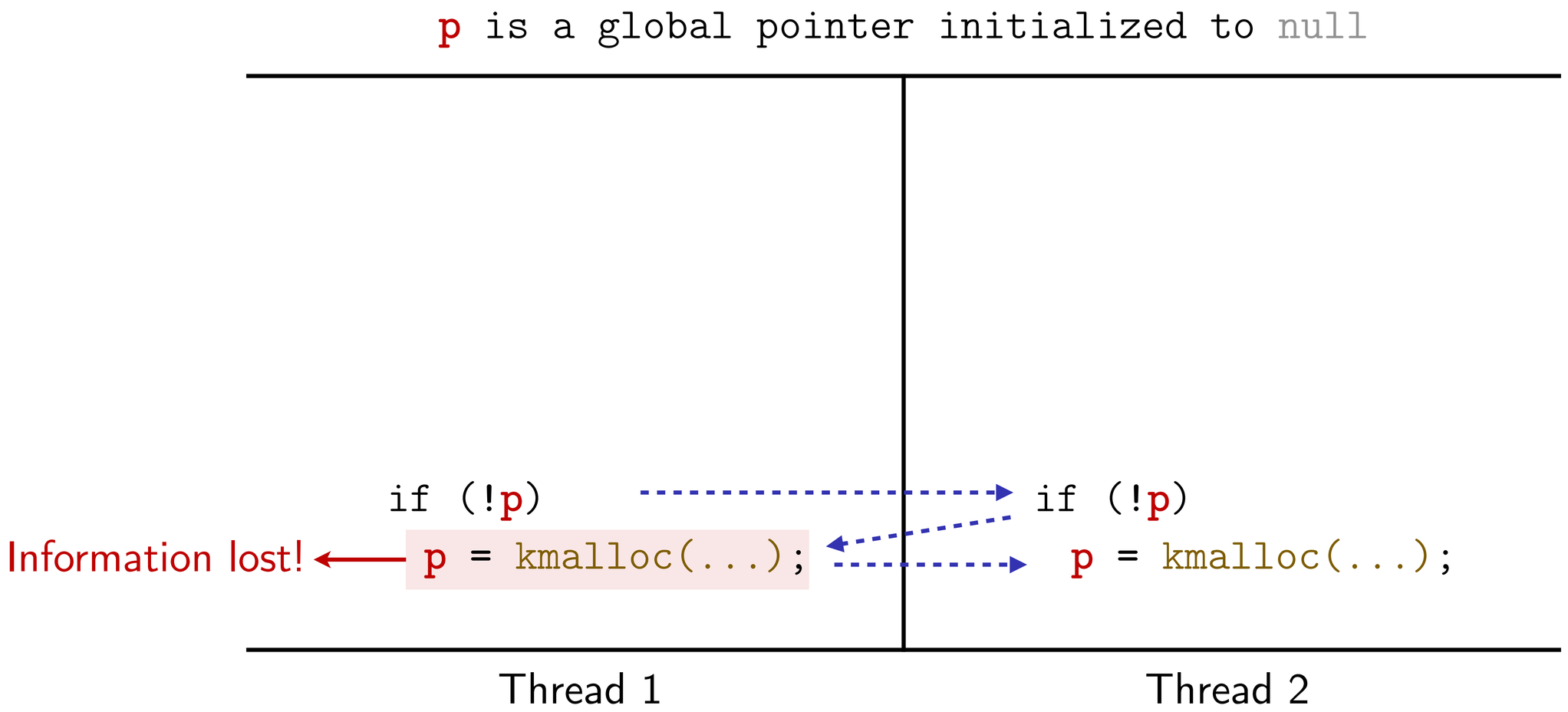
100,000

High level of concurrency in the Linux kernel

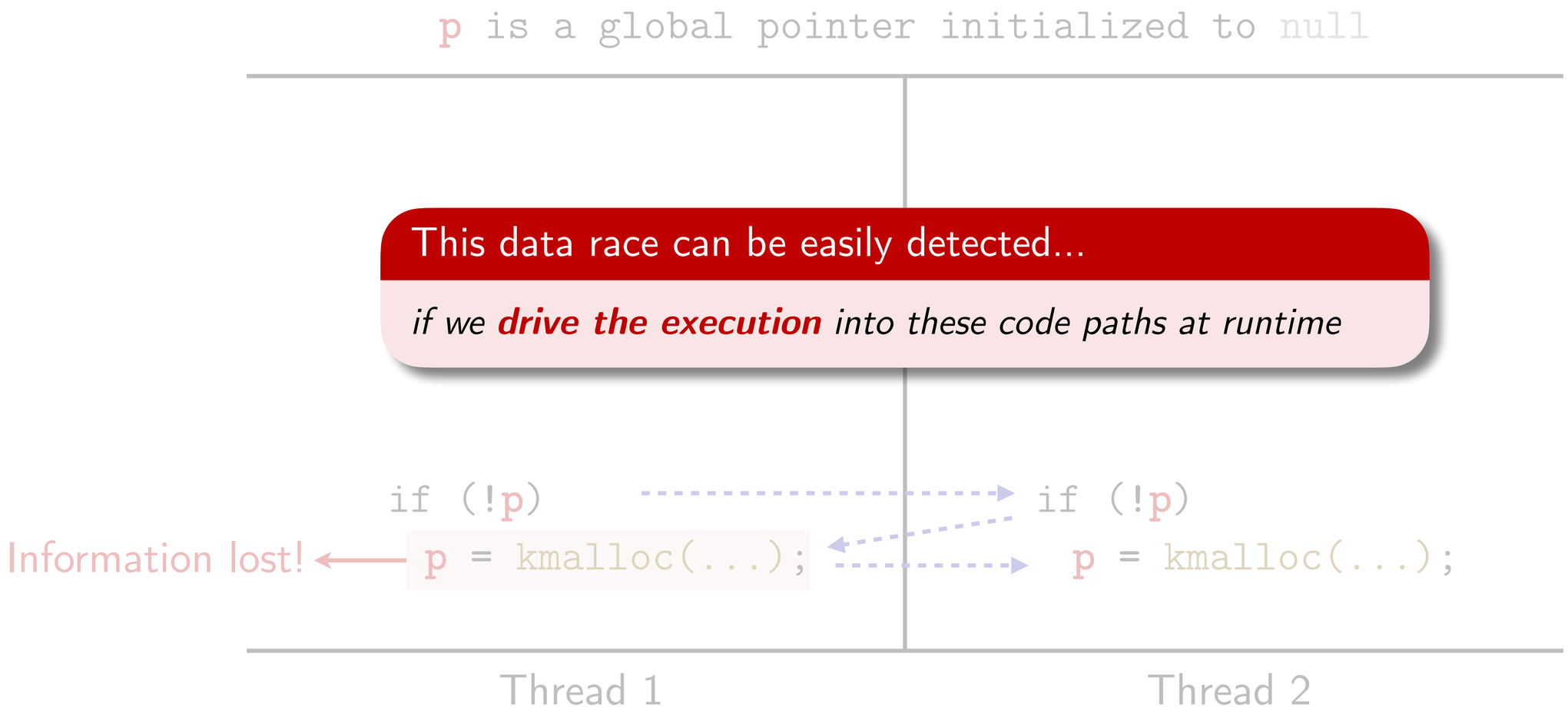
```
1  struct btrfs_fs_info {
2      /* work queues */
3      struct btrfs_workqueue *workers;
4      struct btrfs_workqueue *delalloc_workers;
5      struct btrfs_workqueue *flush_workers;
6      struct btrfs_workqueue *endio_workers;
7      struct btrfs_workqueue *endio_meta_workers;
8      struct btrfs_workqueue *endio_raid56_workers;
9      struct btrfs_workqueue *endio_repair_workers;
10     struct btrfs_workqueue *rmw_workers;
11     struct btrfs_workqueue *endio_meta_write_workers;
12     struct btrfs_workqueue *endio_write_workers;
13     struct btrfs_workqueue *endio_freespace_worker;
14     struct btrfs_workqueue *submit_workers;
15     struct btrfs_workqueue *caching_workers;
16     struct btrfs_workqueue *readahead_workers;
17     struct btrfs_workqueue *fixup_workers;
18     struct btrfs_workqueue *delayed_workers;
19     struct btrfs_workqueue *scrub_workers;
20     struct btrfs_workqueue *scrub_wr_completion_workers;
21     struct btrfs_workqueue *scrub_parity_workers;
22     struct btrfs_workqueue *qgroup_rescan_workers;
23     /* background threads */
24     struct task_struct *transaction_kthread;
25     struct task_struct *cleaner_kthread;
26 };
```

22 threads run
in the background!

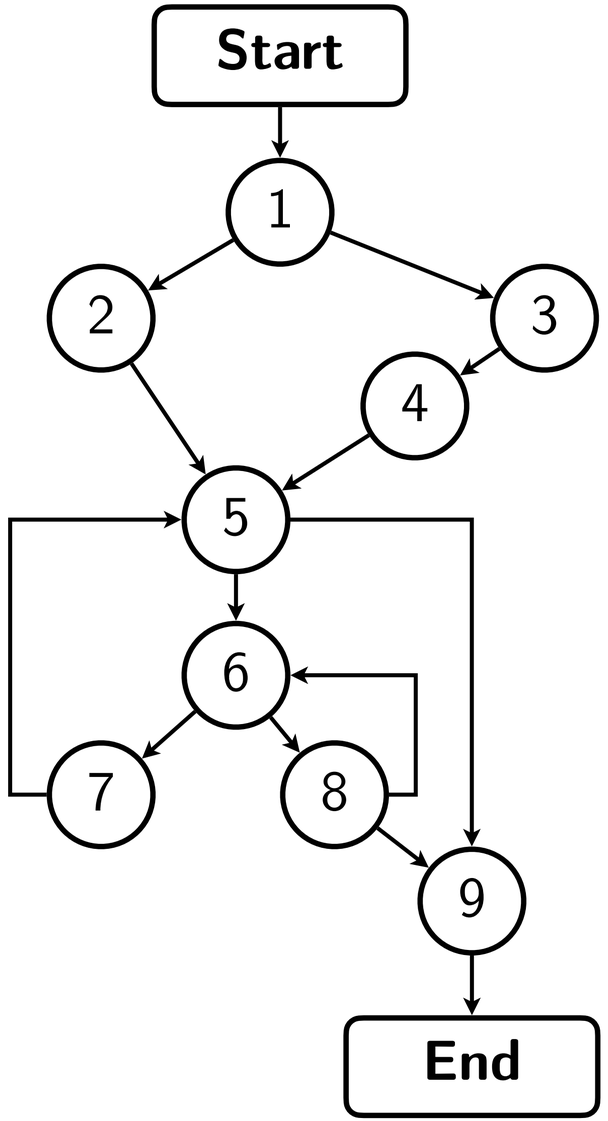
A data race in the kernel



A data race in the kernel

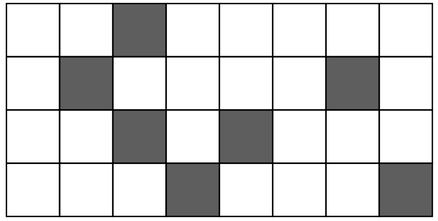


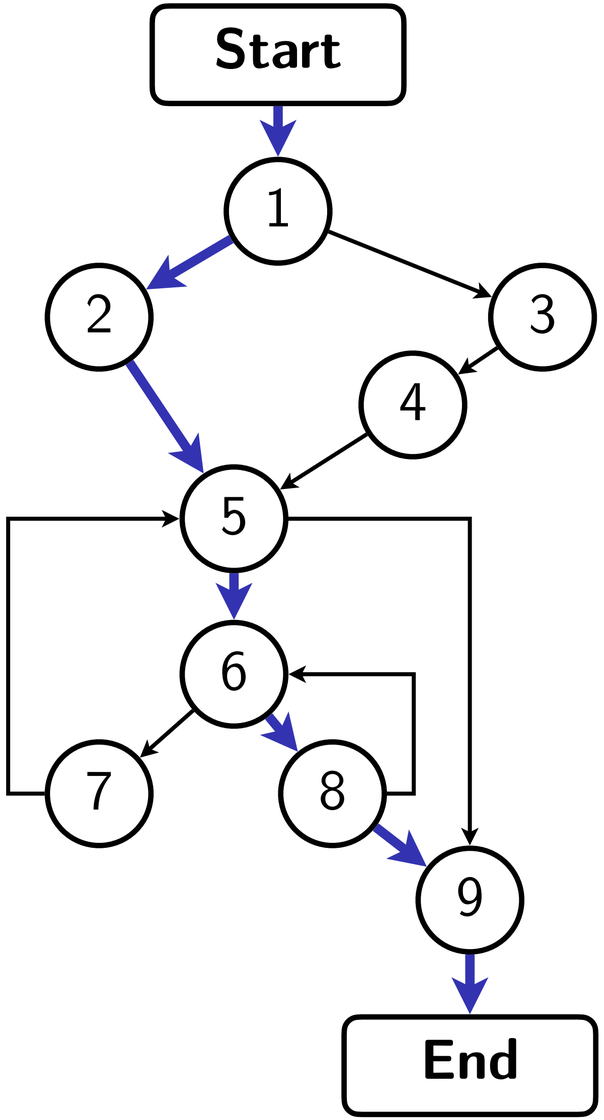
Fuzzing as a way to explore the program



Code coverage as an approximation

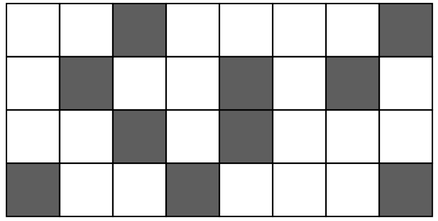
```
open(“some-file”, O_READ, ...)
```

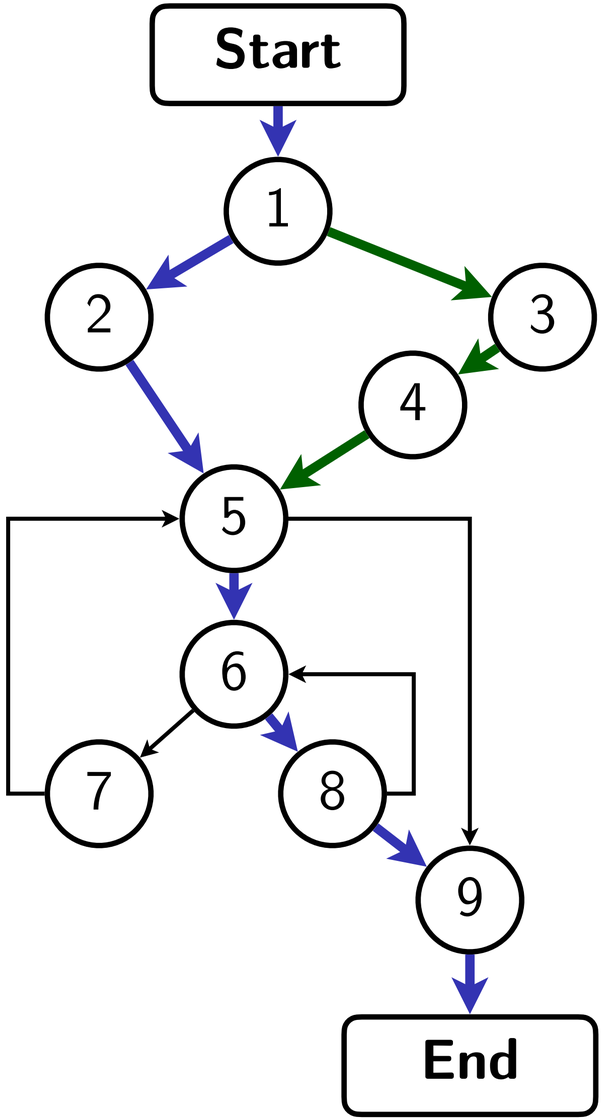




Code coverage as an approximation

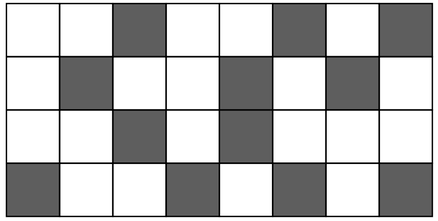
```
open(“some-file”, O_READ, ...)
open(“some-file”, O_WRITE, ...)
```

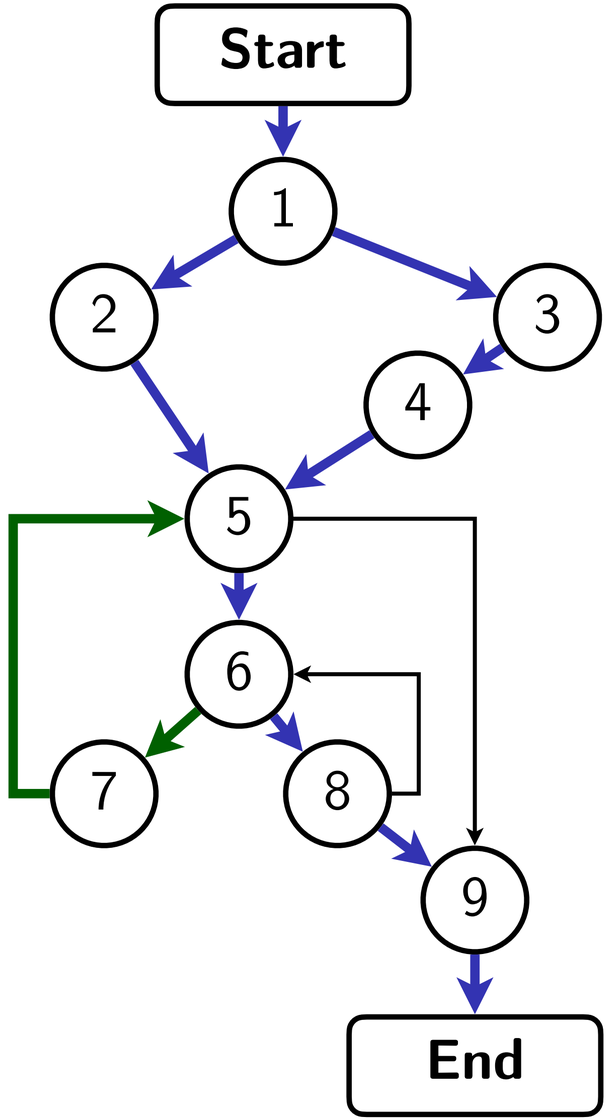




Code coverage as an approximation

```
open(“some-file”, O_READ, ...)
open(“some-file”, O_WRITE, ...)
open(“new-file”, O_READ, ...)
```

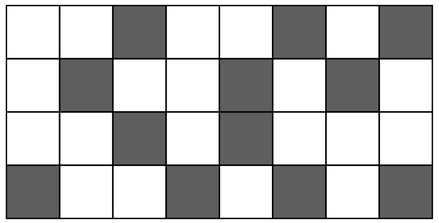


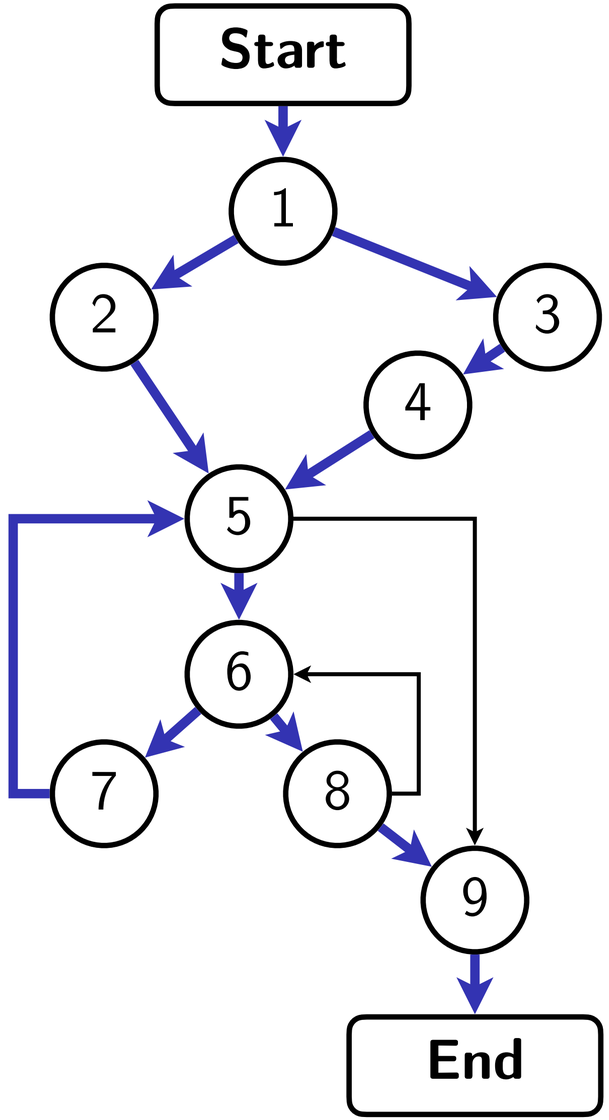


Code coverage as an approximation

```
open(“some-file”, O_READ, ...)
open(“some-file”, O_WRITE, ...)
open(“new-file”, O_READ, ...)
                    ⋮
                  20 trials
                    ⋮
open(“some-file”, O_RDWR, ...)
```

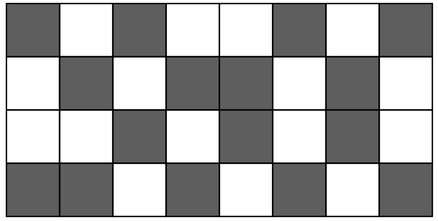
Coverage growth stalled!



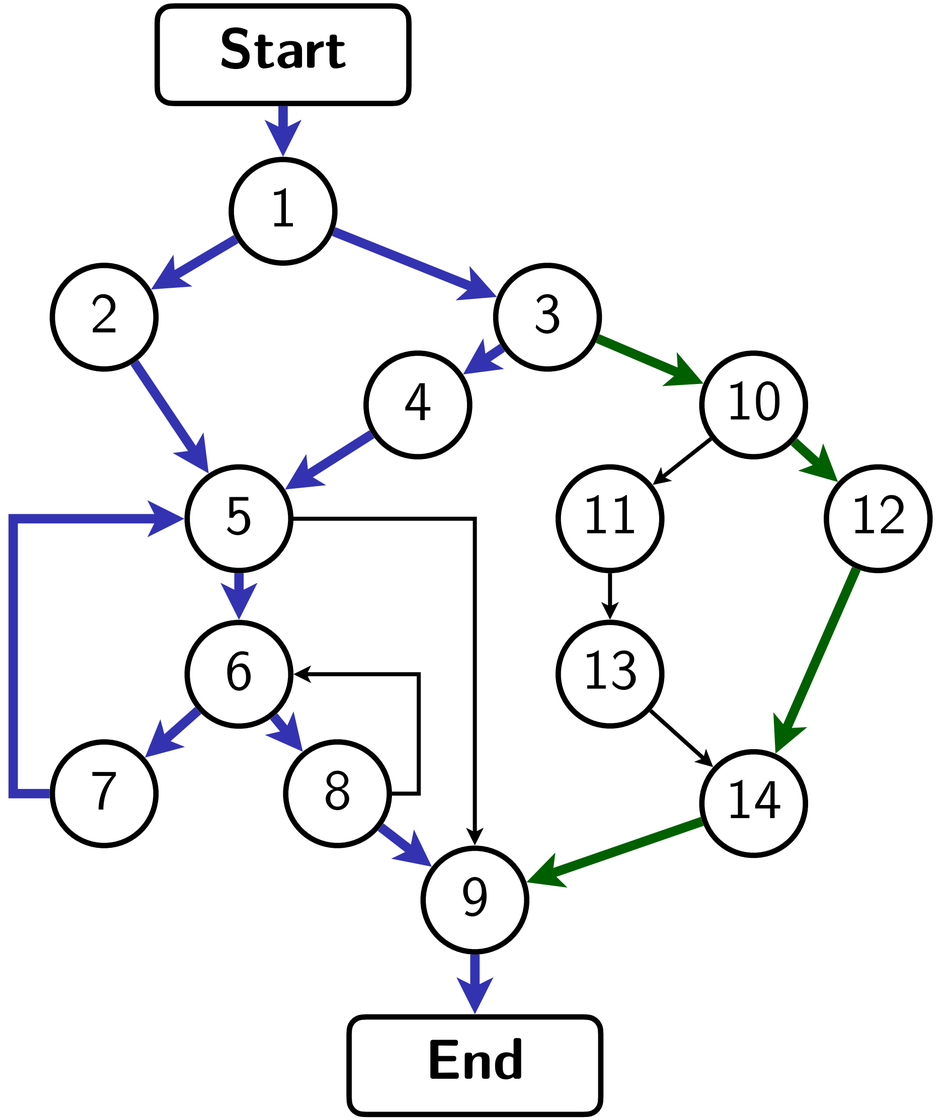


Code coverage as an approximation

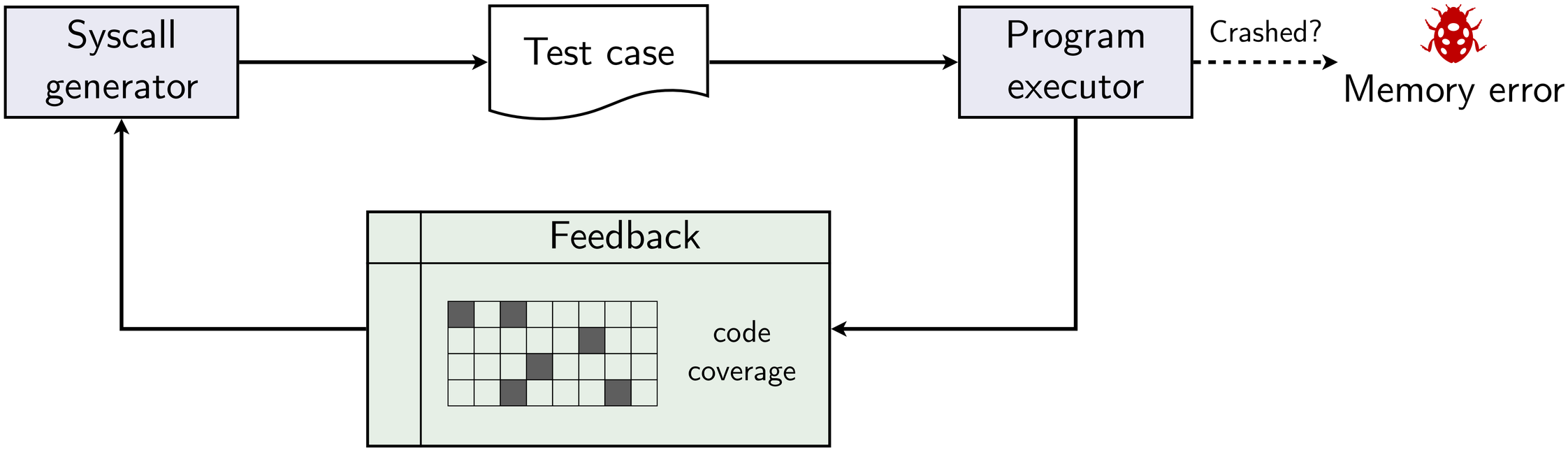
```
open(“some-file”, O_READ, ...)
open(“some-file”, O_WRITE, ...)
open(“new-file”, O_READ, ...)
                  ⋮
                  20 trials
                  ⋮
open(“some-file”, O_RDWR, ...)
```



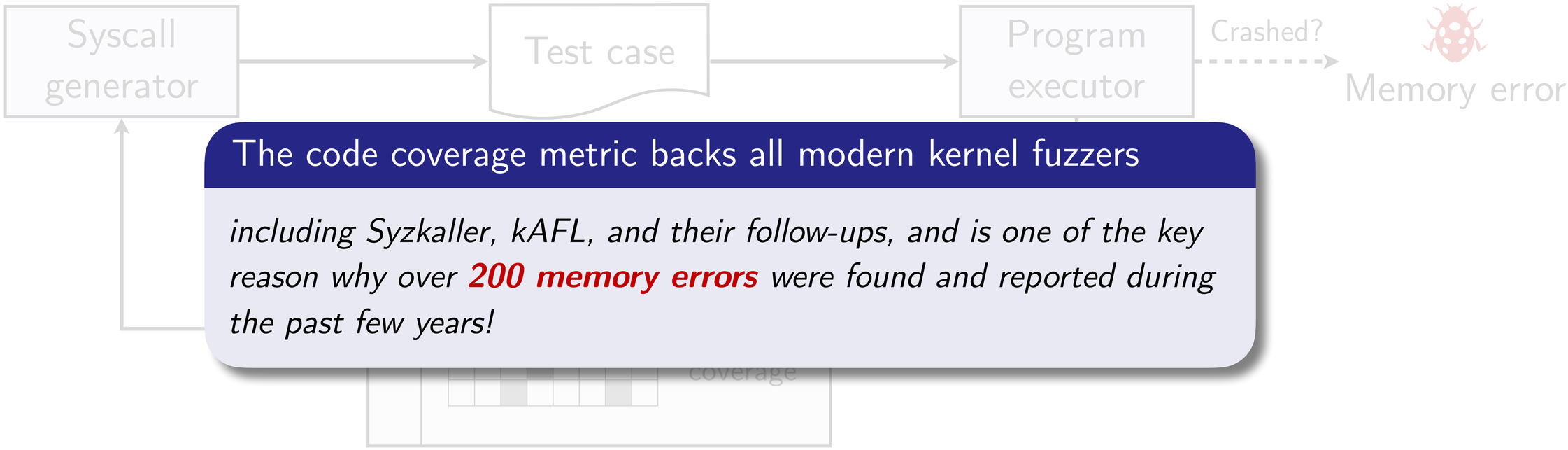
```
rename(“new-file”, “old-file”)
```



The conventional fuzzing process



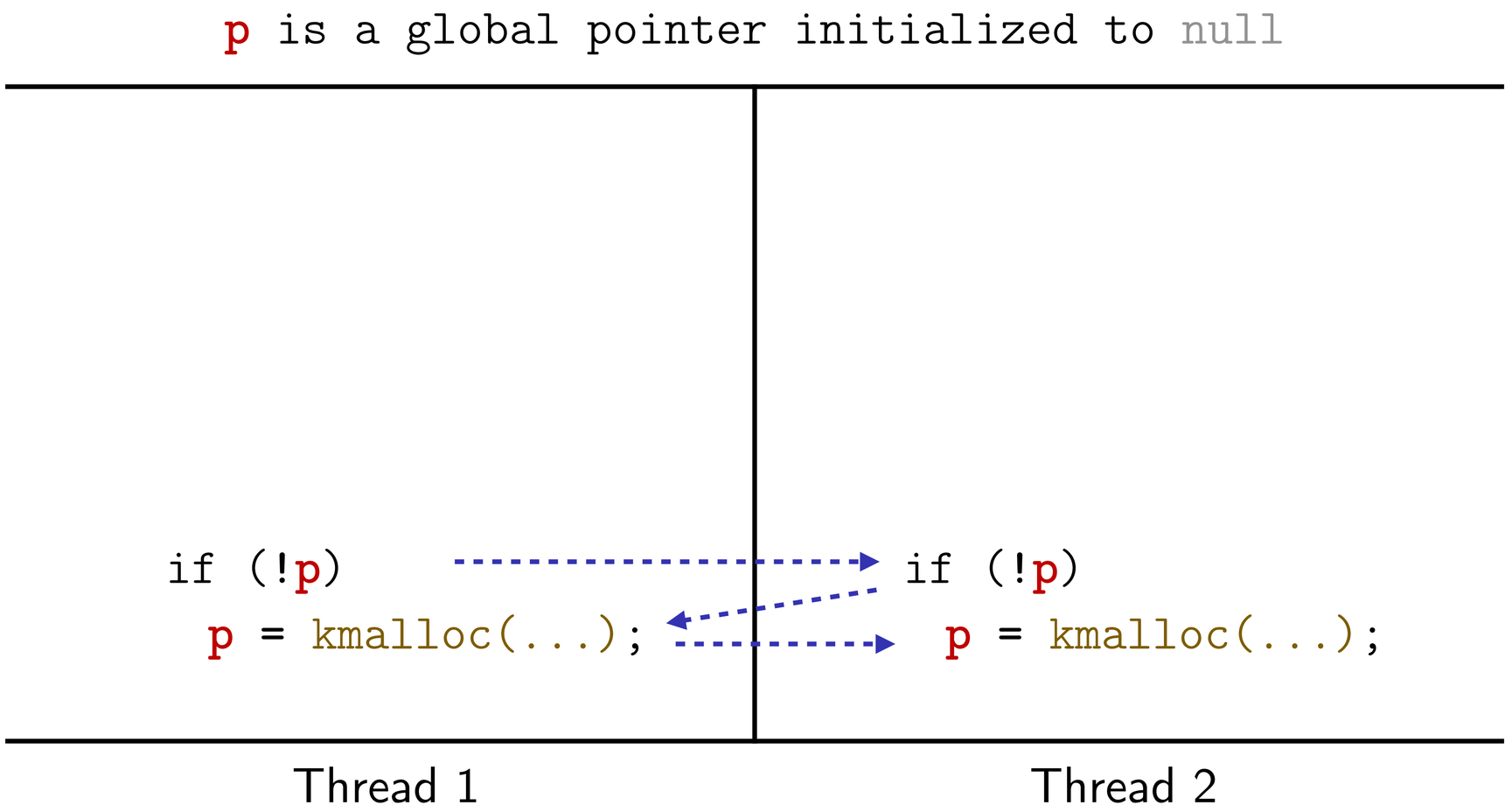
The conventional fuzzing process



The code coverage metric backs all modern kernel fuzzers

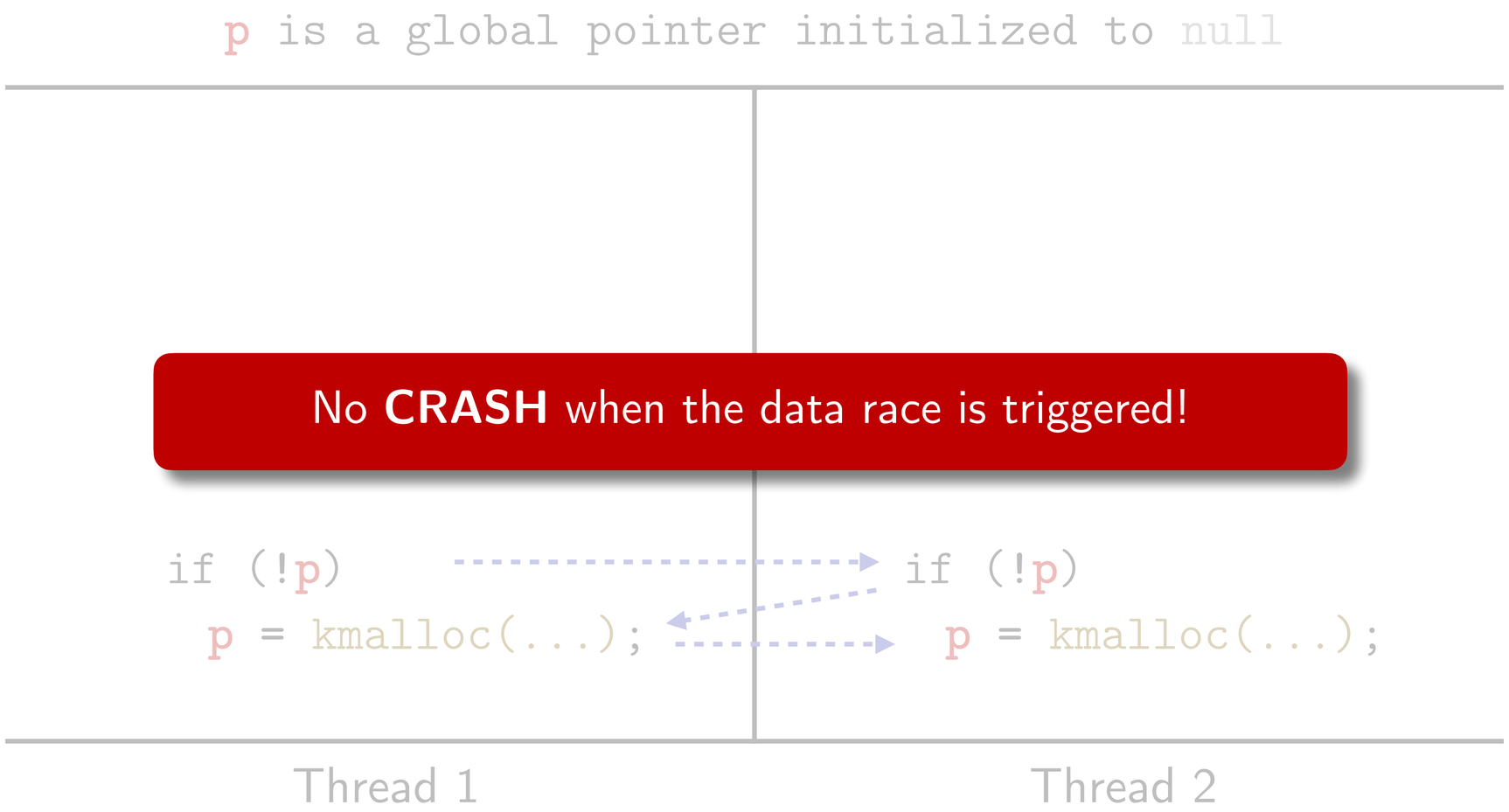
including Syzkaller, kAFL, and their follow-ups, and is one of the key reason why over ***200 memory errors*** *were found and reported during the past few years!*

Back to our data race example



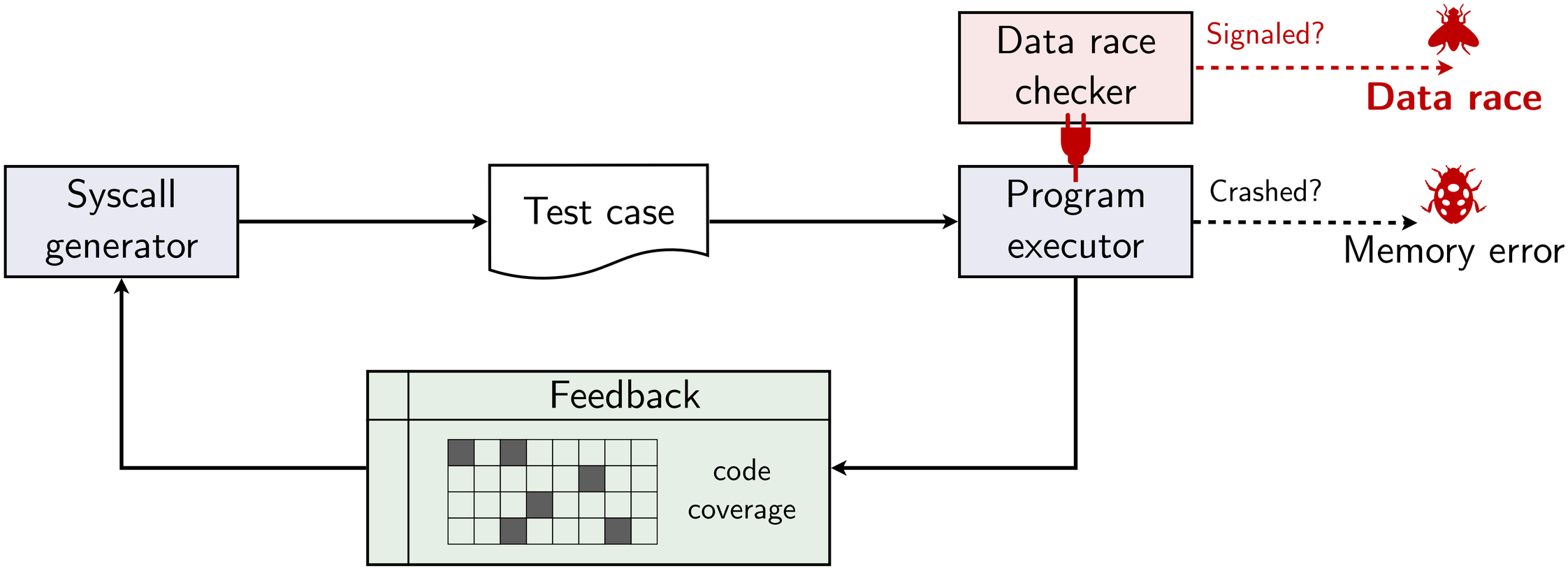
*Assume sequential consistency.

Back to our data race example



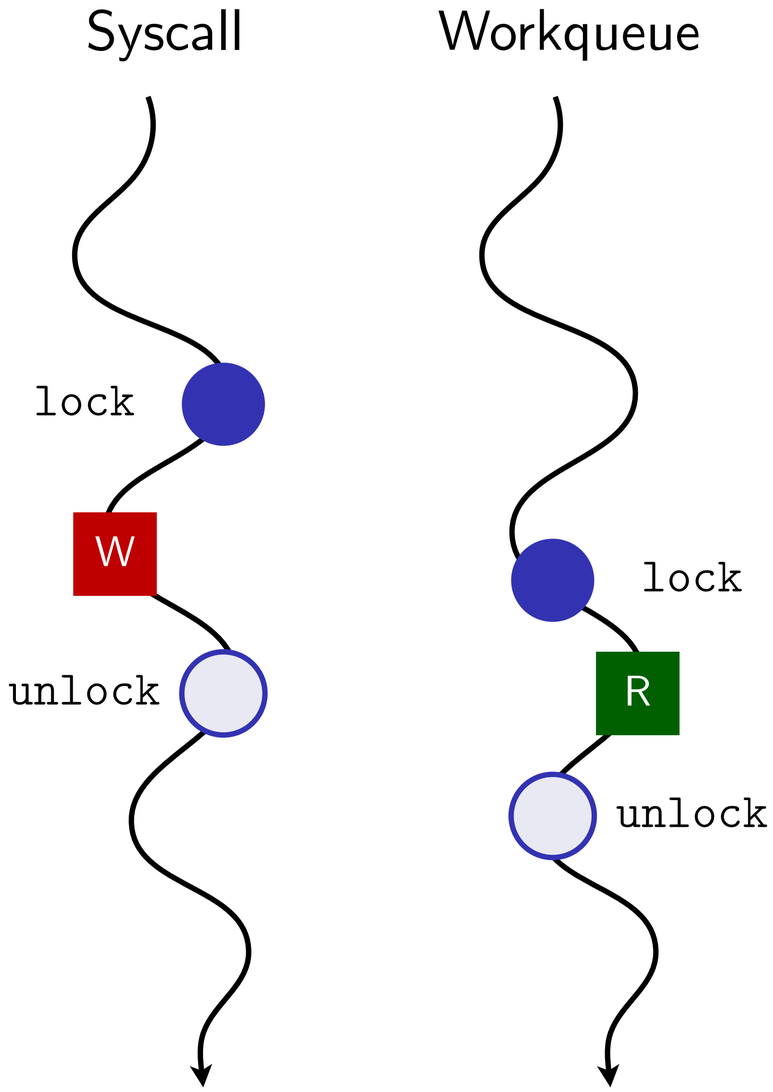
*Assume sequential consistency.

Bring out data races explicitly with a checker



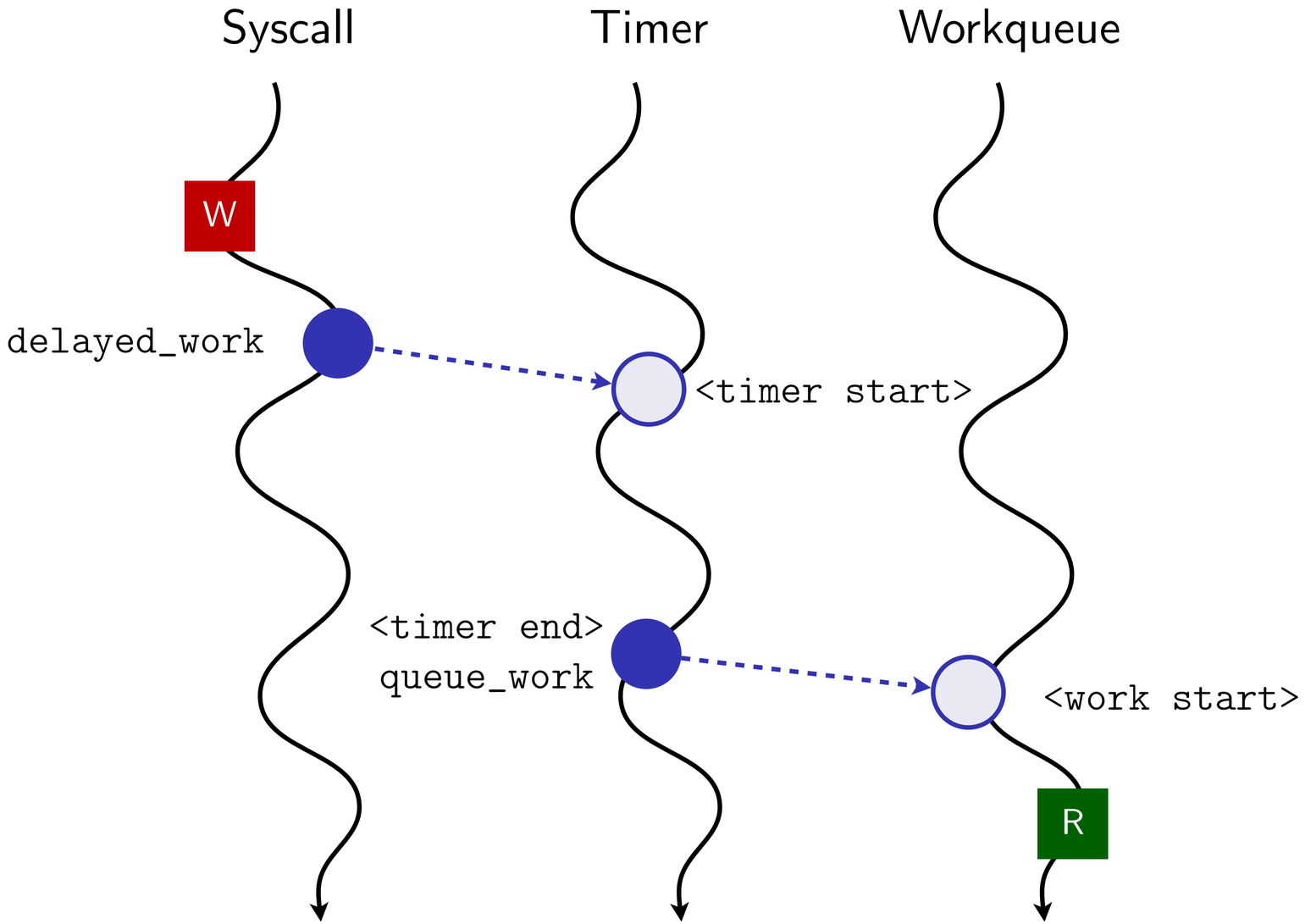
Checking data races - locking

- **Fork-style**
  - Work queues
  - Kernel threads
  - RCU callbacks
  - Timer functions
  - Software-based interrupts
  - Inter-processor interrupts
- **Join-style**
  - Wait_* (e.g., wait_event)
  - Semaphores
- **Publisher-subscriber**
  - RCU pointer operations

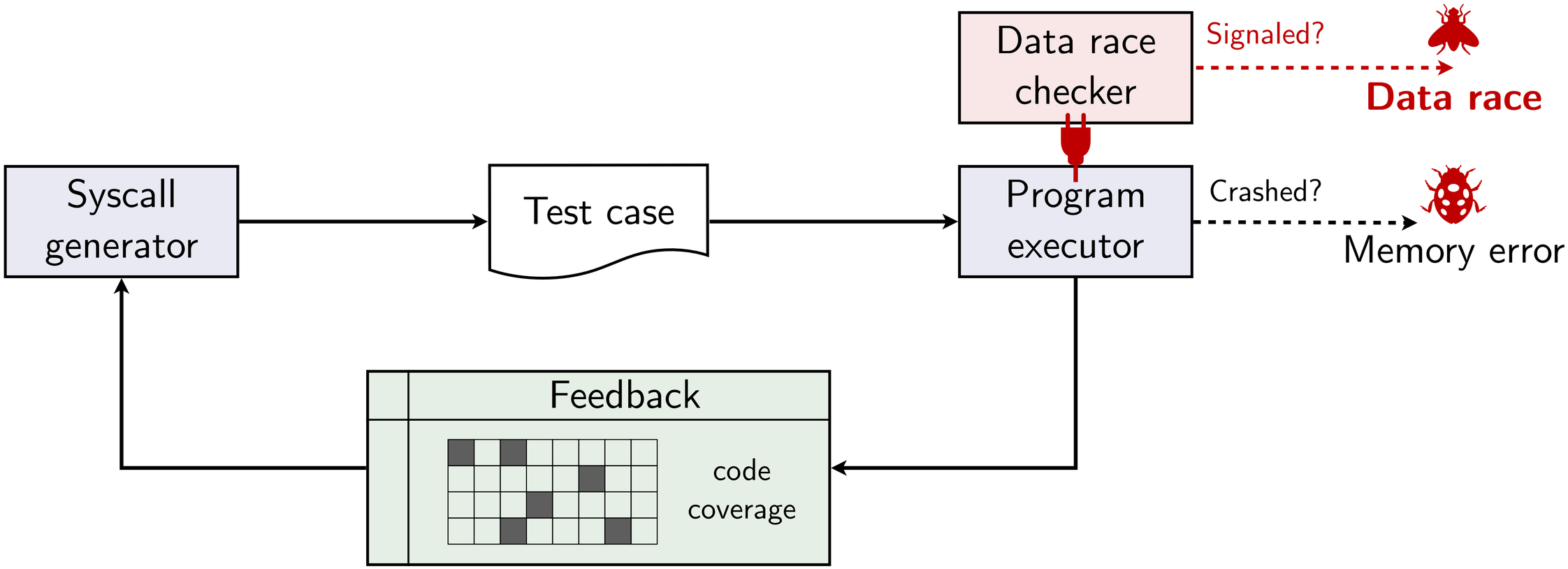


Checking data races - ordering (causality)

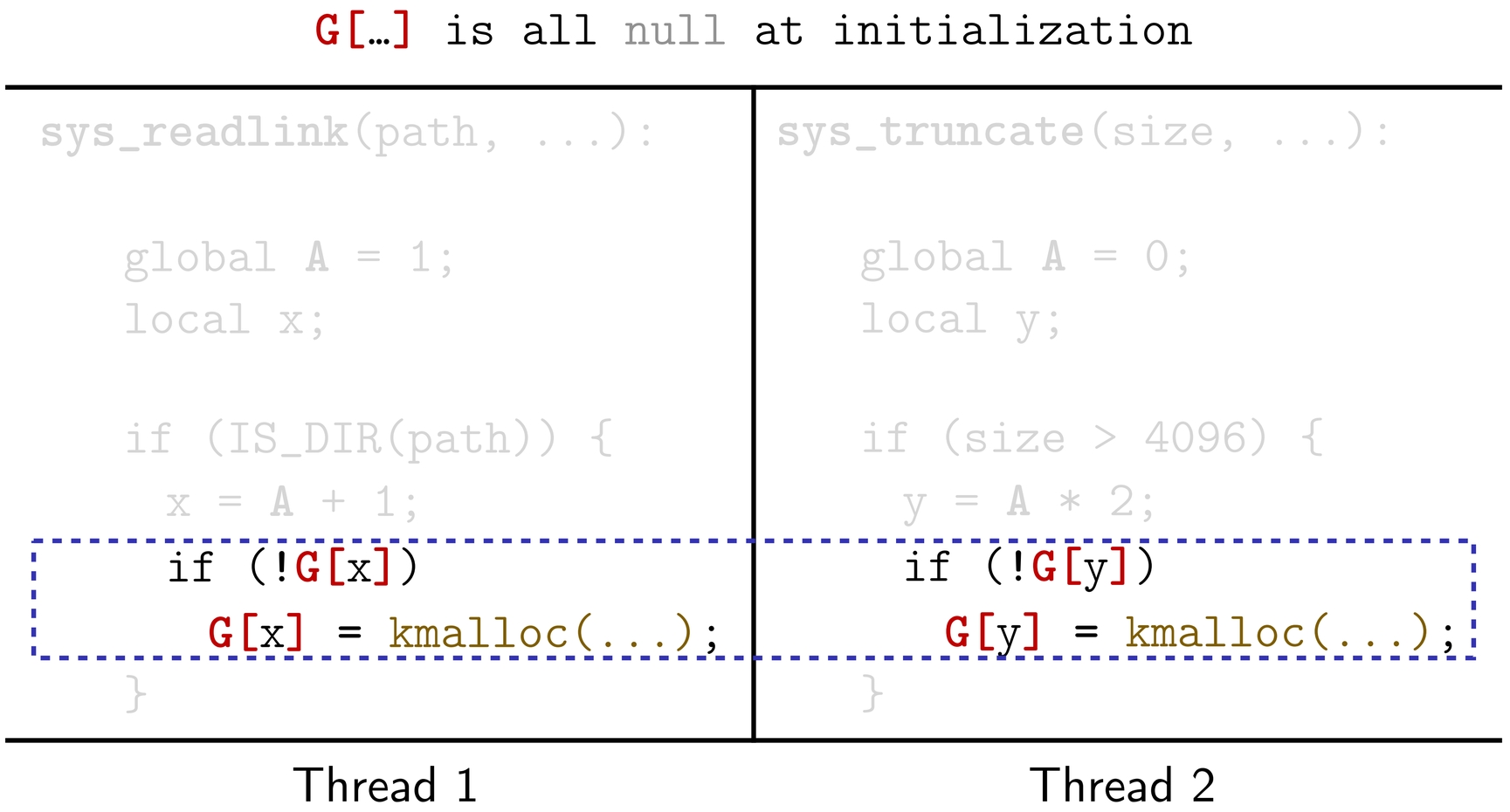
- **Fork-style**
  - Work queues
  - Kernel threads
  - RCU callbacks
  - Timer functions
  - Software-based interrupts
  - Inter-processor interrupts
- **Join-style**
  - Wait_* (e.g., wait_event)
  - Semaphores
- **Publisher-subscriber**
  - RCU pointer operations



Bring out data races explicitly with a checker



A slightly complicated data race



*Assume sequential consistency.

A slightly complicated data race

`G[…]` is all `null` at initialization

Thread 1	Thread 2
`sys_readlink(path, ...):`	`sys_truncate(size, ...):`
`global A = 1;`	`global A = 0;`
`local x;`	`local y;`
`if (IS_DIR(path)) {`	`if (size > 4096) {`
`x = A + 1;`	`y = A * 2;`
`if (!G[x])`	`if (!G[y])`
`G[x] = kmalloc(...);`	`G[y] = kmalloc(...);`
`}`	`}`

*Assume sequential consistency.

Case simplified

Thread 1	Thread 2
`A = 1;`	`A = 0;`
`x = A + 1;`	`y = A * 2;`

Can we reach x == y?

Case simplified

Thread 1	Thread 2
`A = 1;`	`A = 0;`
`x = A + 1;`	`y = A * 2;`

Can we reach x == y?

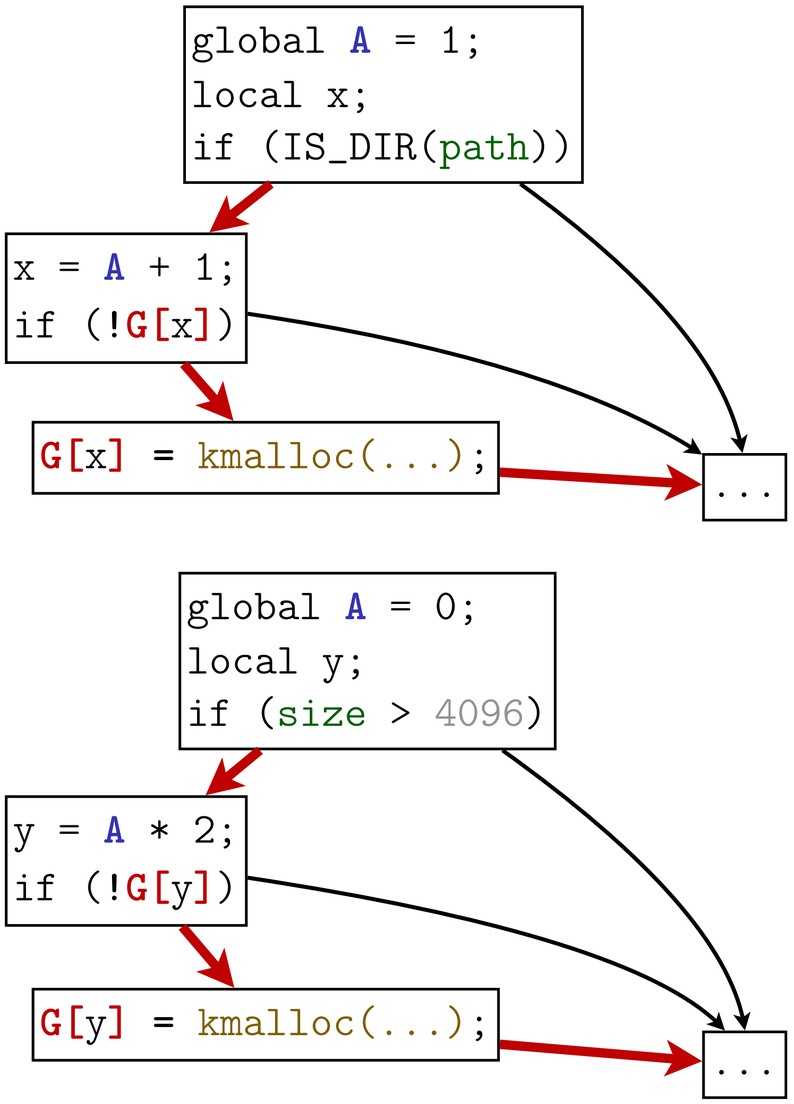
`A = 1;`
`x = A + 1;`
`A = 0;`
`y = A * 2;`
	`A = 1;`
	`A = 0;`
	`x = A + 1;`
	`y = A * 2;`
		`A = 1;`
		`A = 0;`
		`y = A * 2;`
		`x = A + 1;`

x = 2, y = 0	*x = 1, y = 0*	*x = 1, y = 0*

`A = 0;`
`y = A * 2;`
`A = 1;`
`x = A + 1;`
	`A = 0;`
	`A = 1;`
	`y = A * 2;`
	`x = A + 1;`
		`A = 0;`
		`A = 1;`
		`x = A + 1;`
		`y = A * 2;`

x = 2, y = 0	***x = 2, y = 2***	***x = 2, y = 2***

All interleavings yield to the same code coverage!



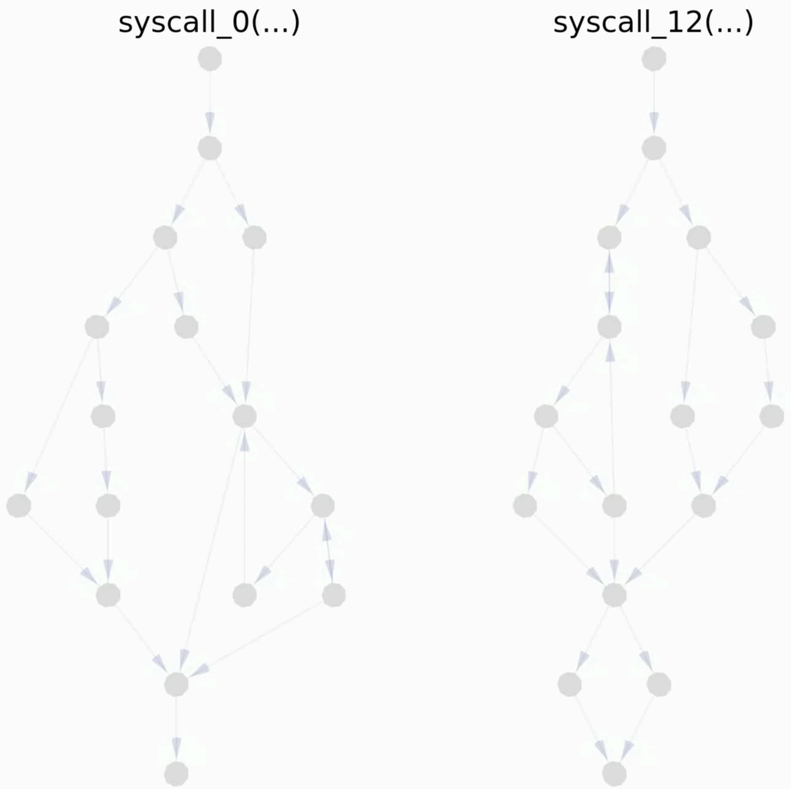
`A = 1;`
`x = A + 1;`
`A = 0;`
`y = A * 2;`
	`A = 1;`
	`A = 0;`
	`x = A + 1;`
	`y = A * 2;`
		`A = 1;`
		`A = 0;`
		`y = A * 2;`
		`x = A + 1;`

x = 2, y = 0	*x = 1, y = 0*	*x = 1, y = 0*

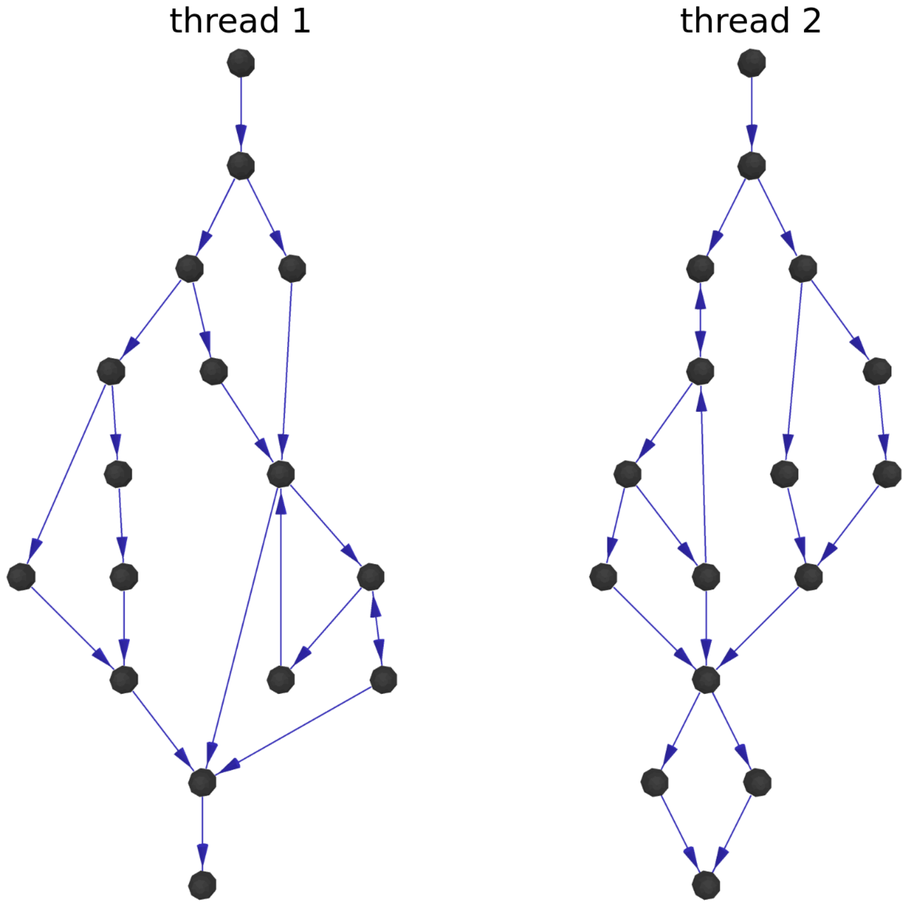
`A = 0;`
`y = A * 2;`
`A = 1;`
`x = A + 1;`
	`A = 0;`
	`A = 1;`
	`y = A * 2;`
	`x = A + 1;`
		`A = 0;`
		`A = 1;`
		`x = A + 1;`
		`y = A * 2;`

x = 2, y = 0	***x = 2, y = 2***	***x = 2, y = 2***

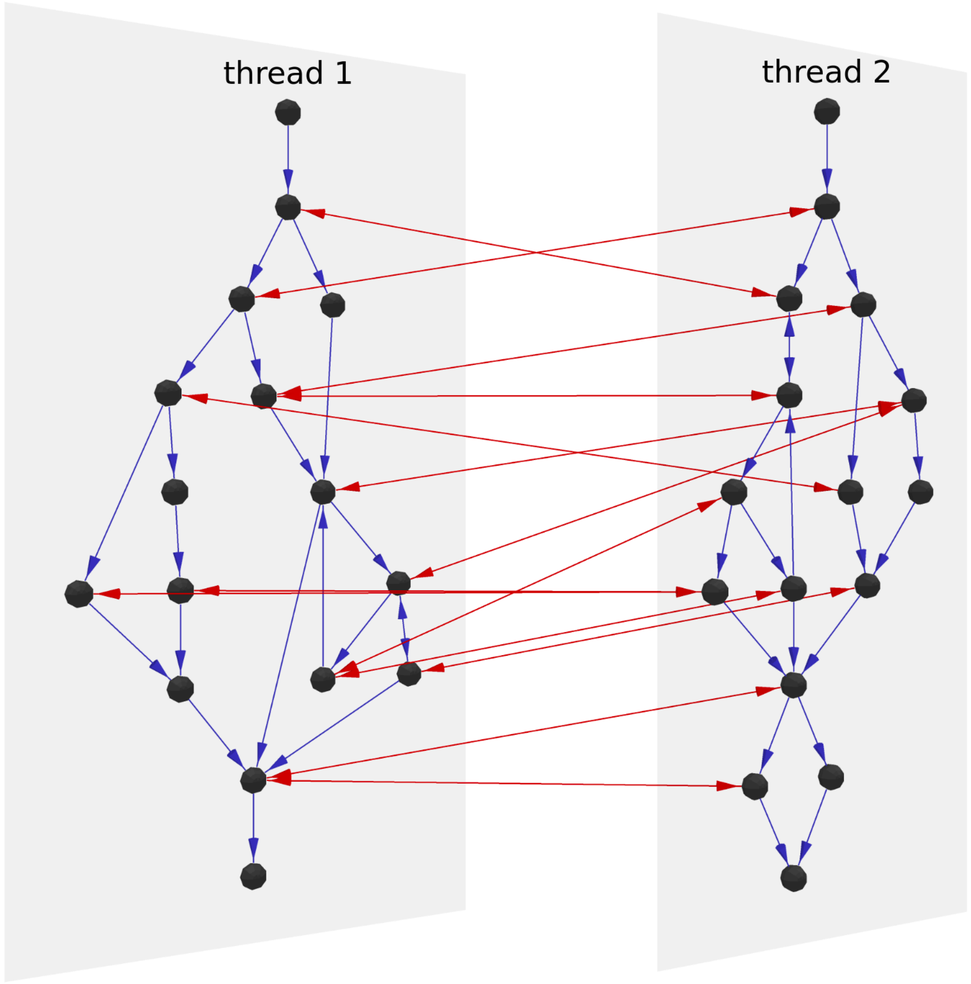
Incompleteness of CFG edge coverage



A multi-dimensional view of coverage in fuzzing

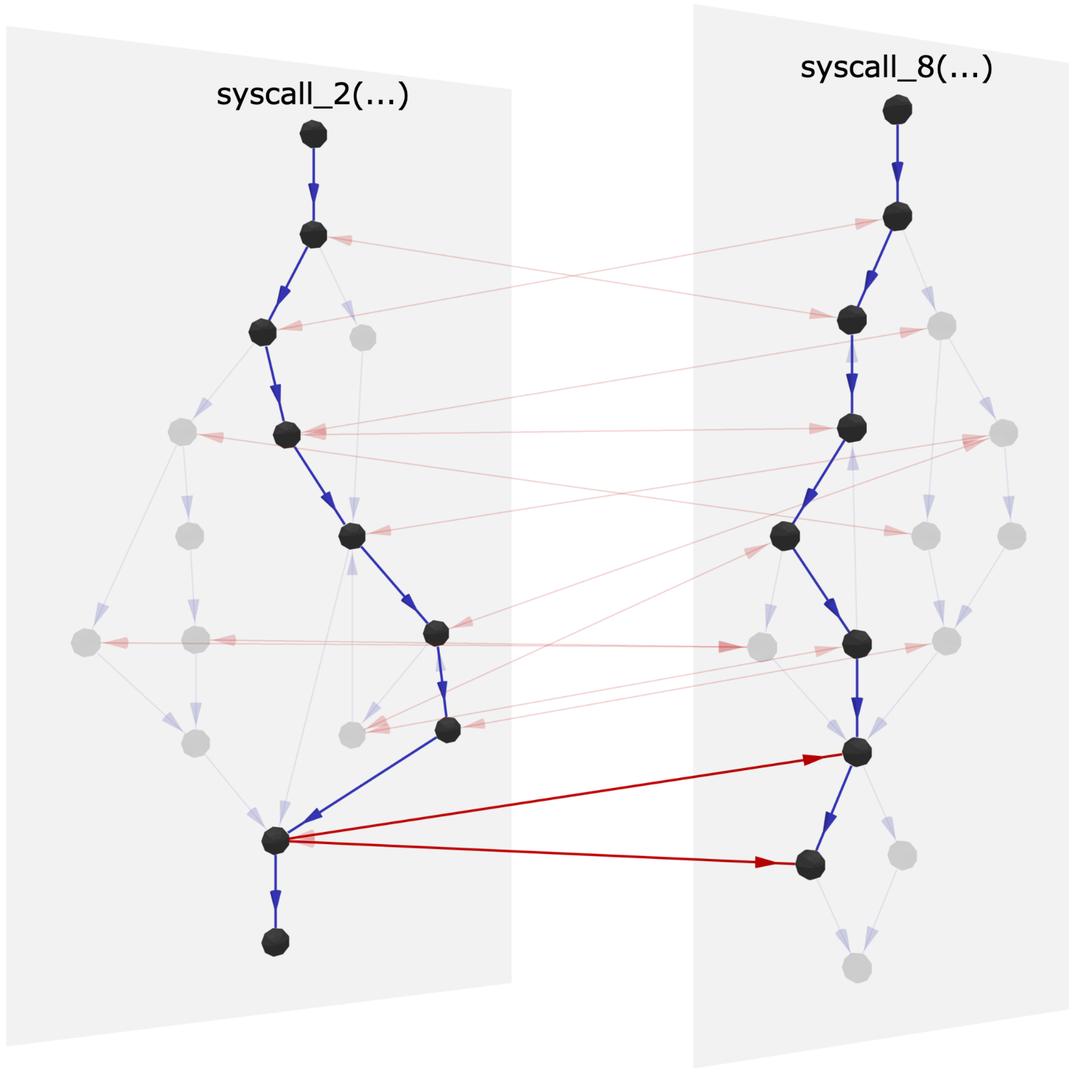


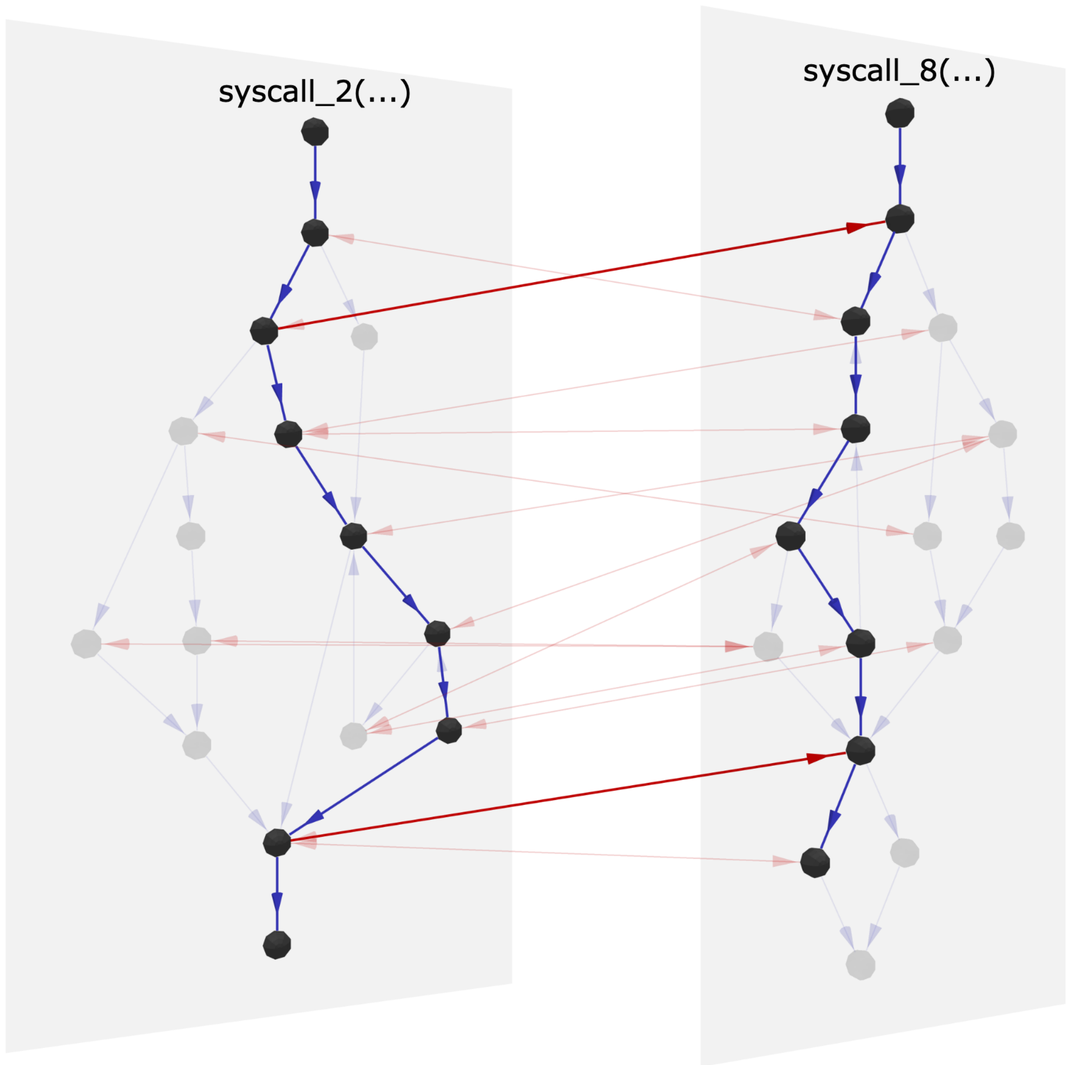
Edge-coverage only



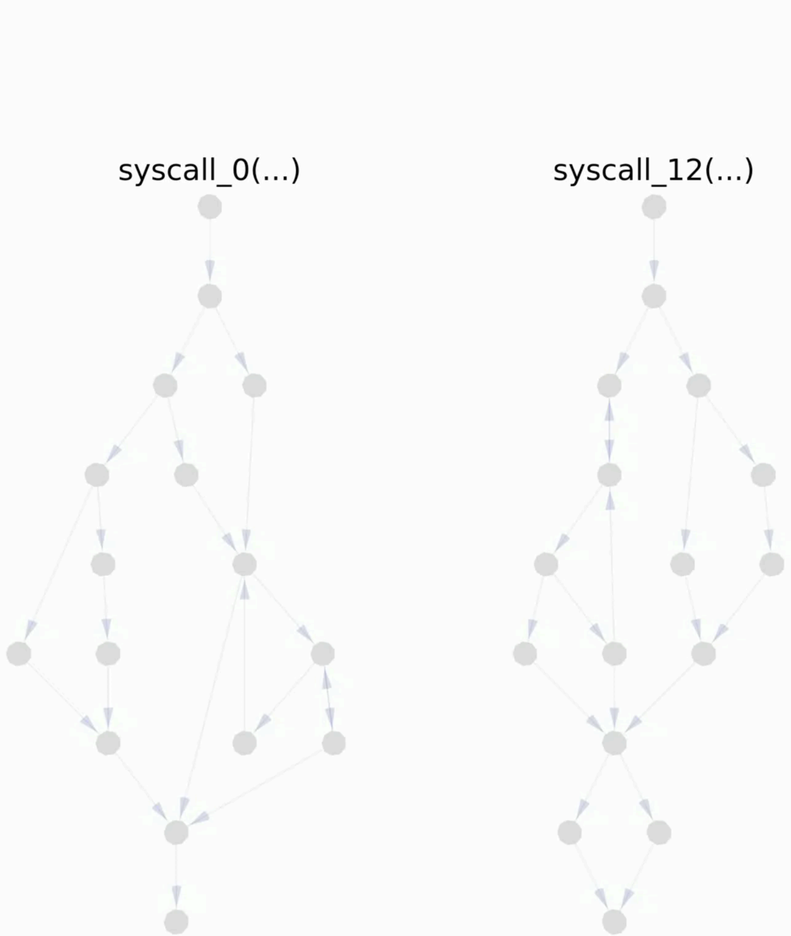
Krace

Visualizing the concurrency dimension

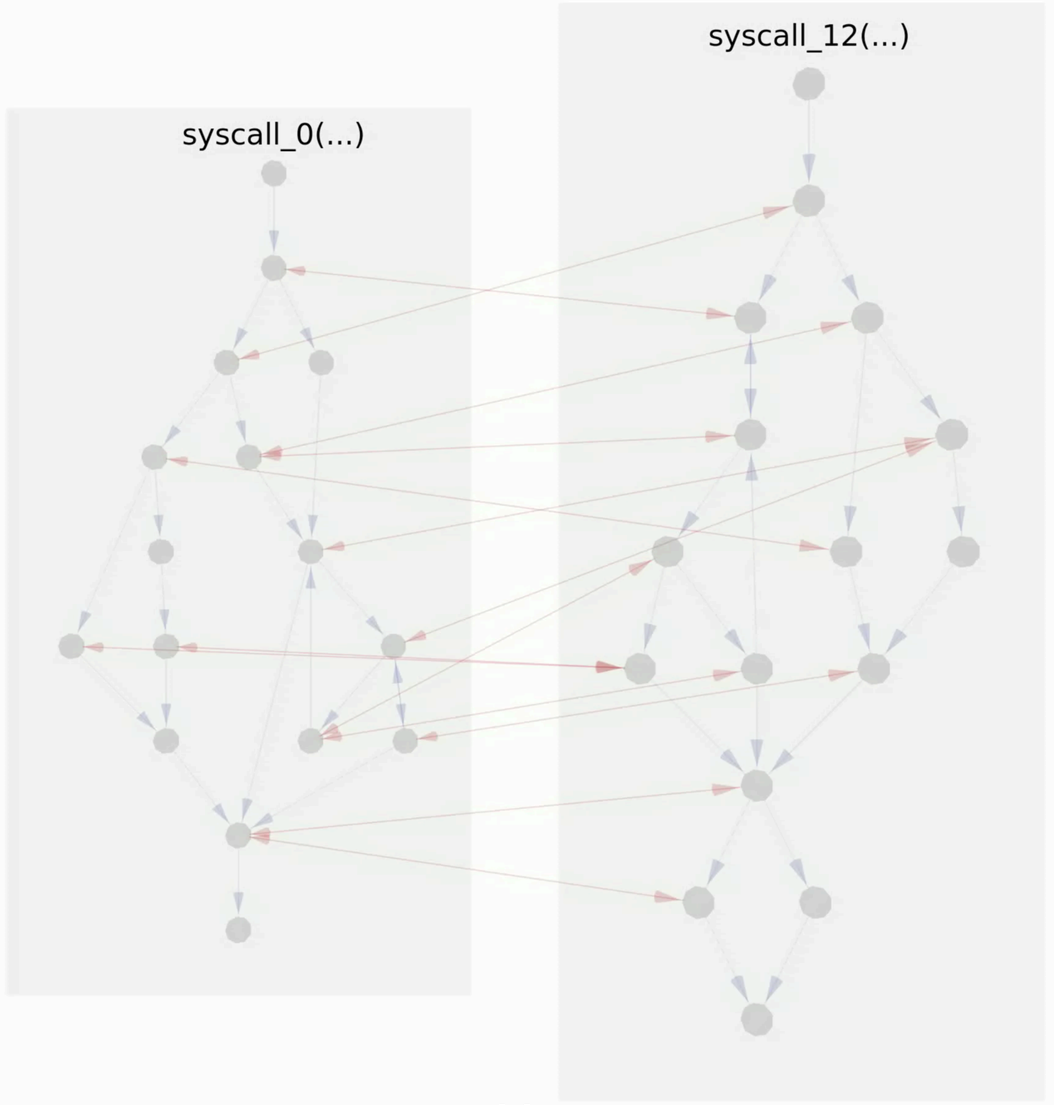




Visualizing the concurrency dimension

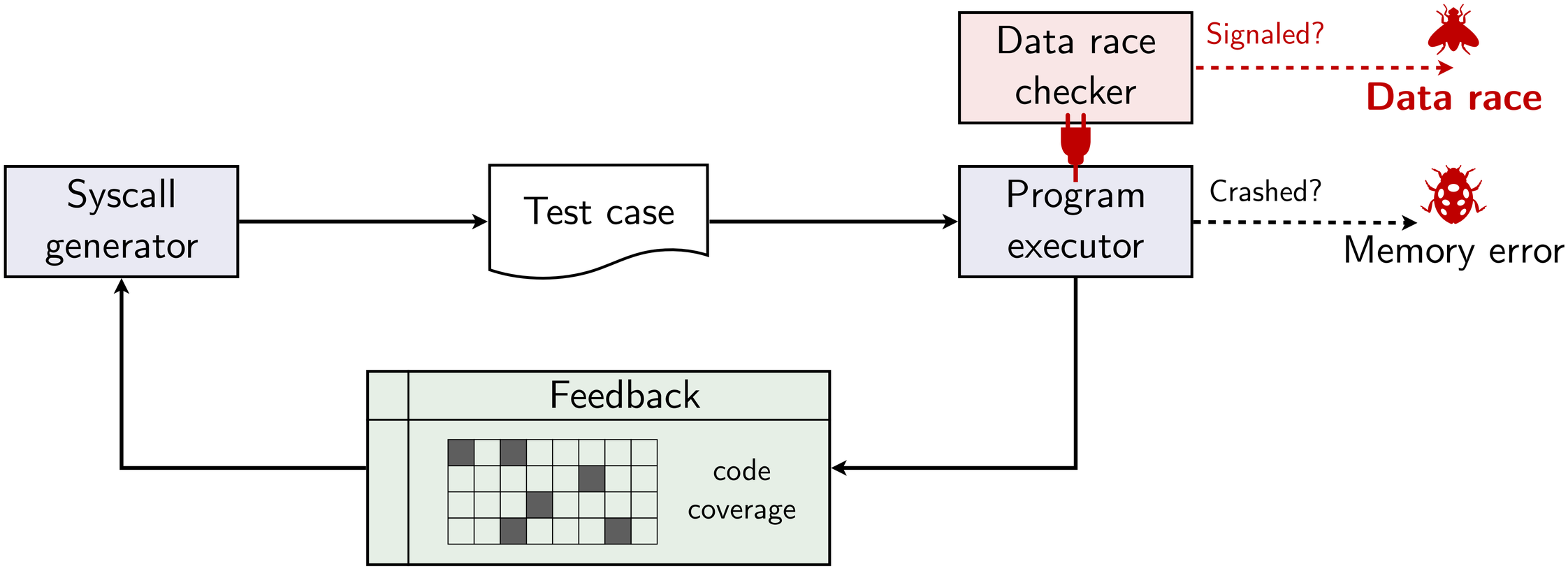


Edge-coverage only

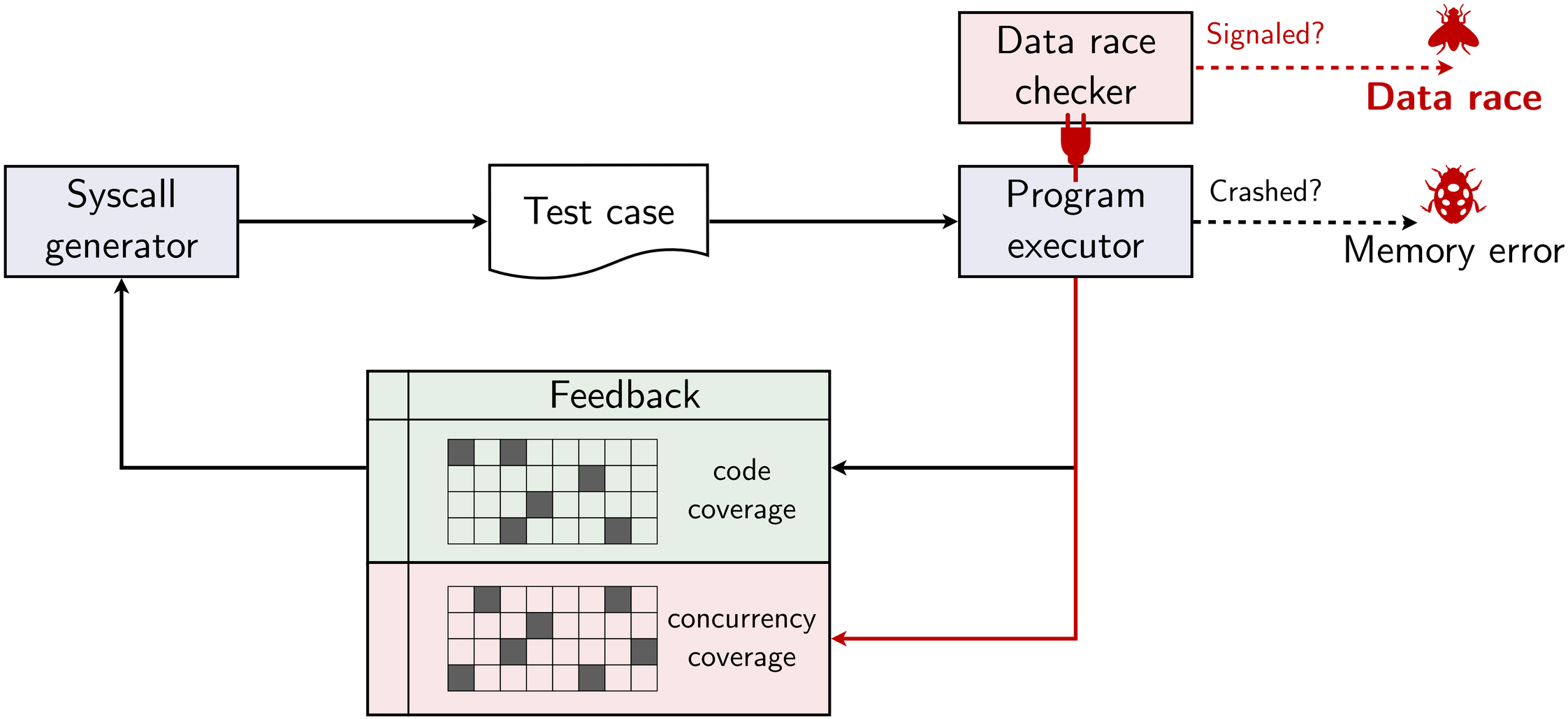


Krace

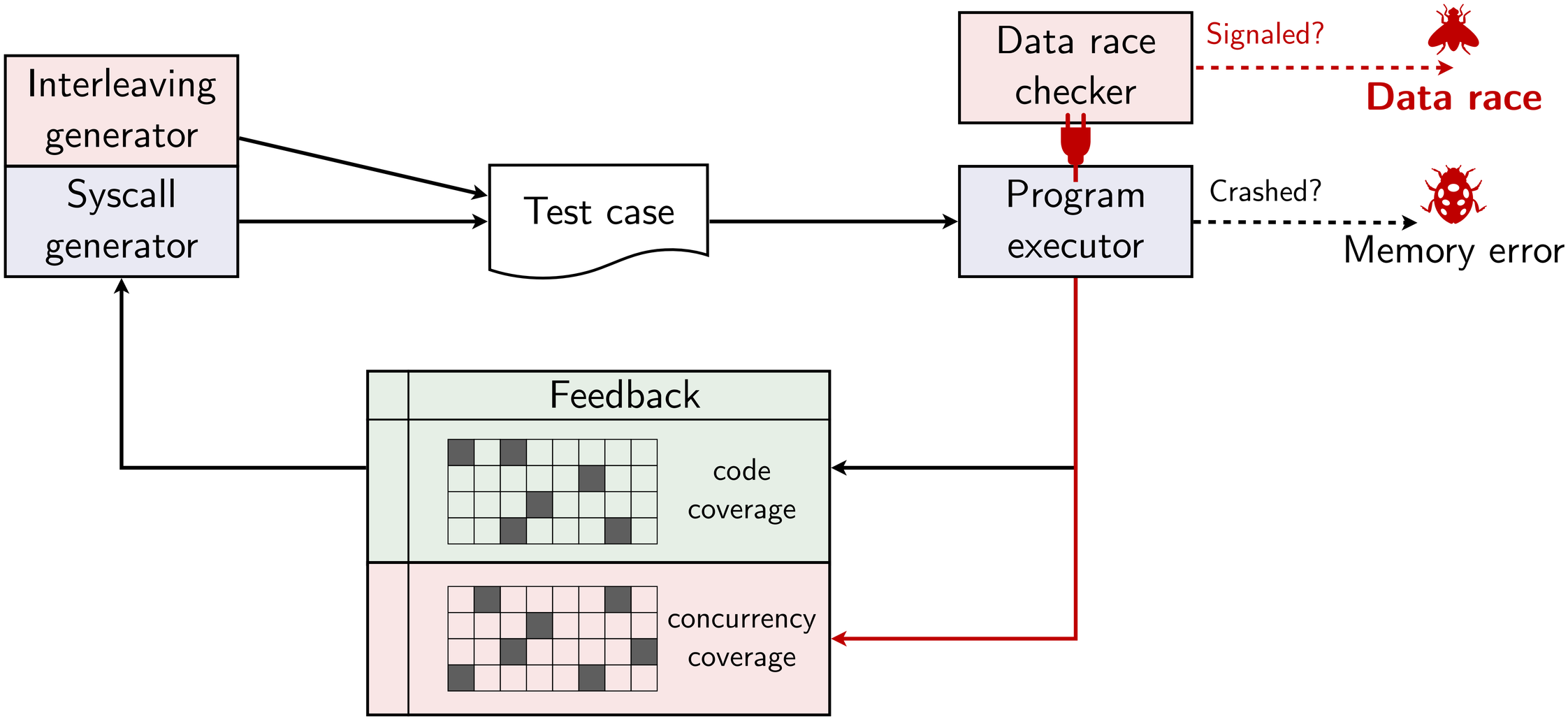
Bring fuzzing to the concurrency dimension



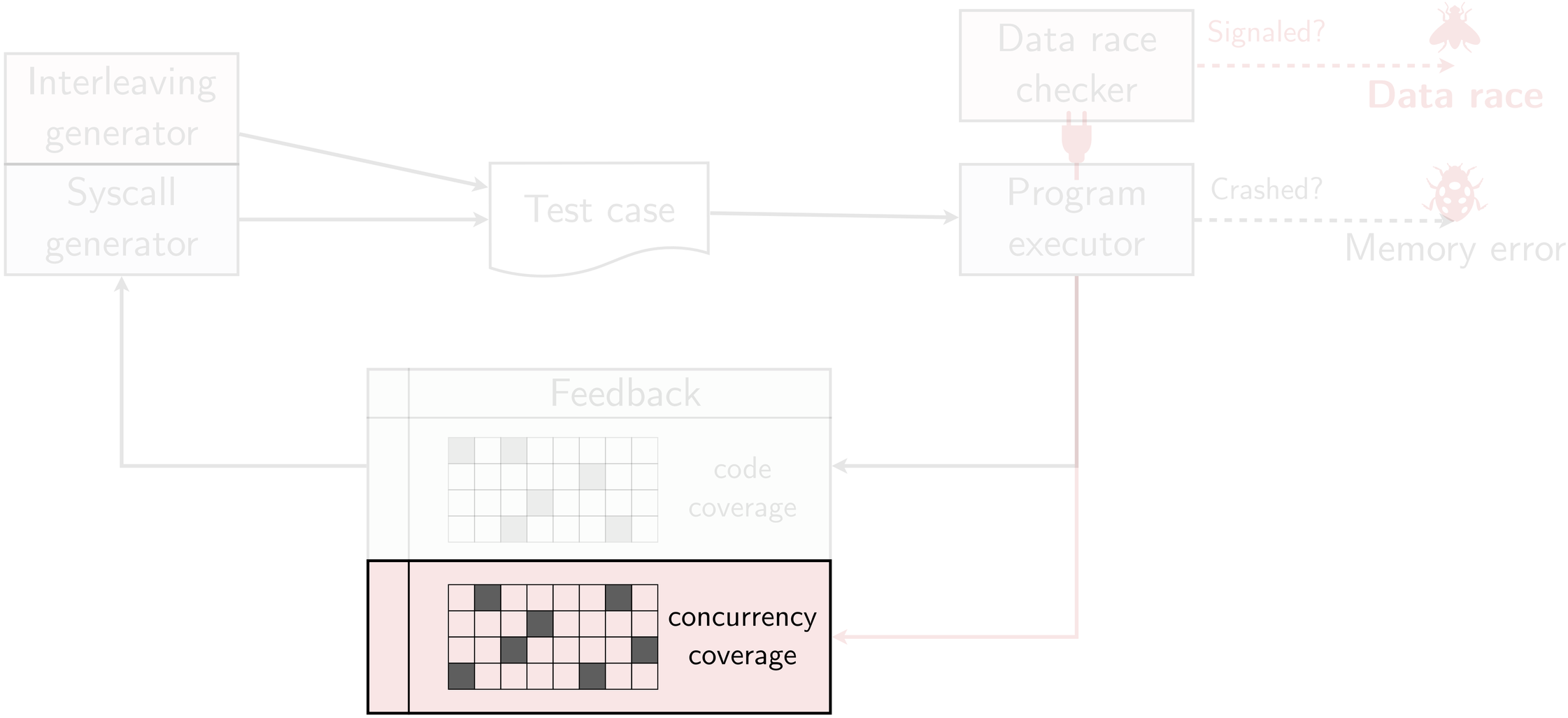
Bring fuzzing to the concurrency dimension



Bring fuzzing to the concurrency dimension



Concurrency coverage tracking



A straw-man solution

sys_readlink(path, ...):

```
i1  global A = 1;
i2  local x;

i3  if (IS_DIR(path)) {
i4    x = A + 1;
i5    if (G[x])
i6      kmalloc(...);
    }
```

Thread 1

sys_truncate(size, ...):

```
i7  global A = 0;
i8  local y;

i9  if (size > 4096) {
i10   y = A * 2;
i11   if (G[y])
i12     kmalloc(...);
    }
```

Thread 2

A straw-man solution

```
sys_readlink(path, ...):

i1  global A = 1;
i2  local x;

i3  if (IS_DIR(path)) {
i4    x = A + 1;
i5    if (G[x])
i6      kmalloc(...);
    }
```

Thread 1

```
i1 global A = 1;
        i7 global A = 0;
        i8 local y;
i2 local x;
i3 if (IS_DIR(path)) {
        i9 if (size > 4096) {
i4   x = A + 1;
        i10   y = A * 2;
i5   if(G[x])
        i11   if (G[y])
        i12     kmalloc(...);
              }
i6     kmalloc(...);
   }
```

A possible interleaving

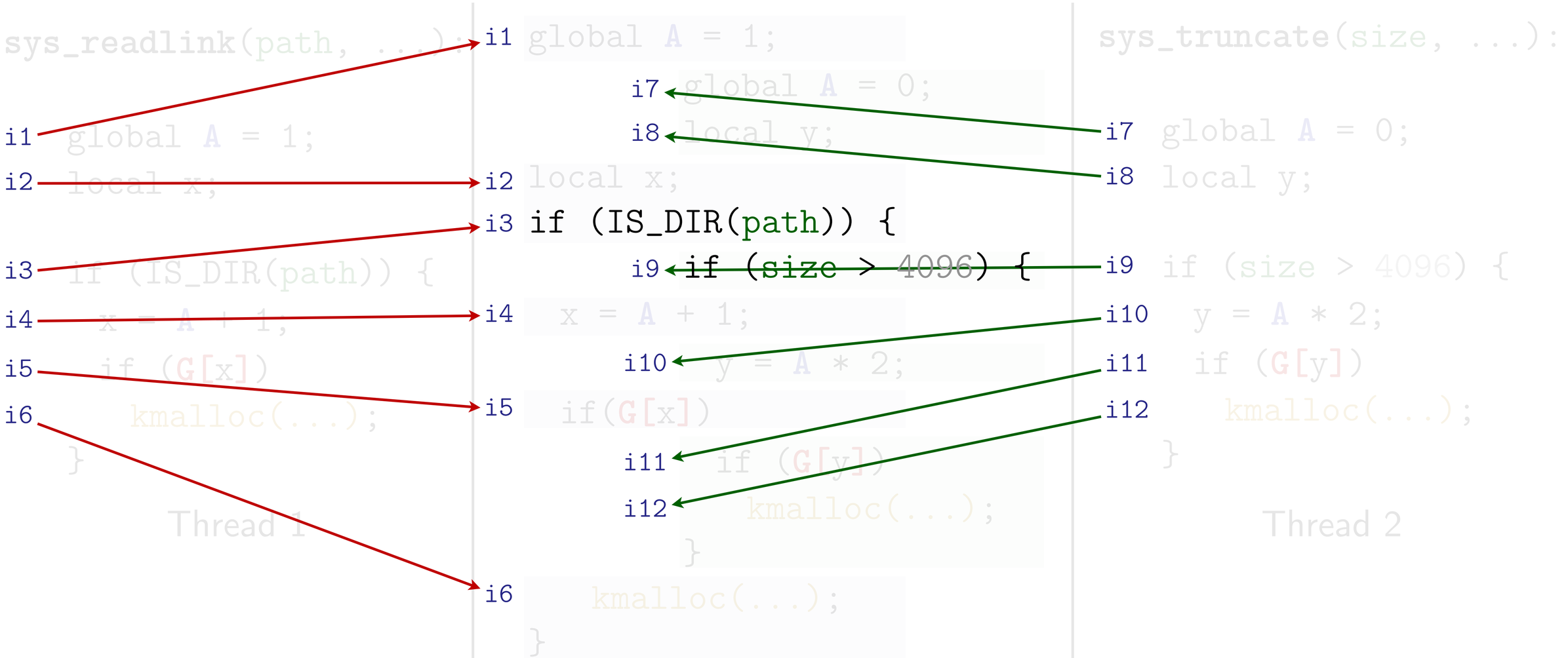
```
sys_truncate(size, ...):

i7  global A = 0;
i8  local y;

i9  if (size > 4096) {
i10   y = A * 2;
i11   if (G[y])
i12     kmalloc(...);
    }
```

Thread 2

A straw-man solution



Hash(i1, i7, i8, i2, ***i3***, ***i9***, i4, i10, i5, i11, i12, i6) = 7825

Hash(i1, i7, i8, i2, ***i9***, ***i3***, i4, i10, i5, i11, i12, i6) = 1356

A straw-man solution

```
sys_readlink(path, ...):
i1  global A = 1;
i2  local x;
i3  if (IS_DIR(path)
i4    x = A + 1;
i5    if (G[x])
i6      kmalloc(...);
    }
```

Thread 1

```
i1 global A = 1;
                    i3 global A = 0;
                    i4 local y;
                    i5 if (size > 4096) {
                    i6   y = A * 2;
                    i7   if (G[y])
i6 kmalloc(...);    i8     kmalloc(...);
}                   }
```

A possible interleaving

```
sys_truncate(size, ...):
global A = 0;
local y;
if (size > 4096) {
  y = A * 2;
  if (G[y])
    kmalloc(...);
```

Thread 2

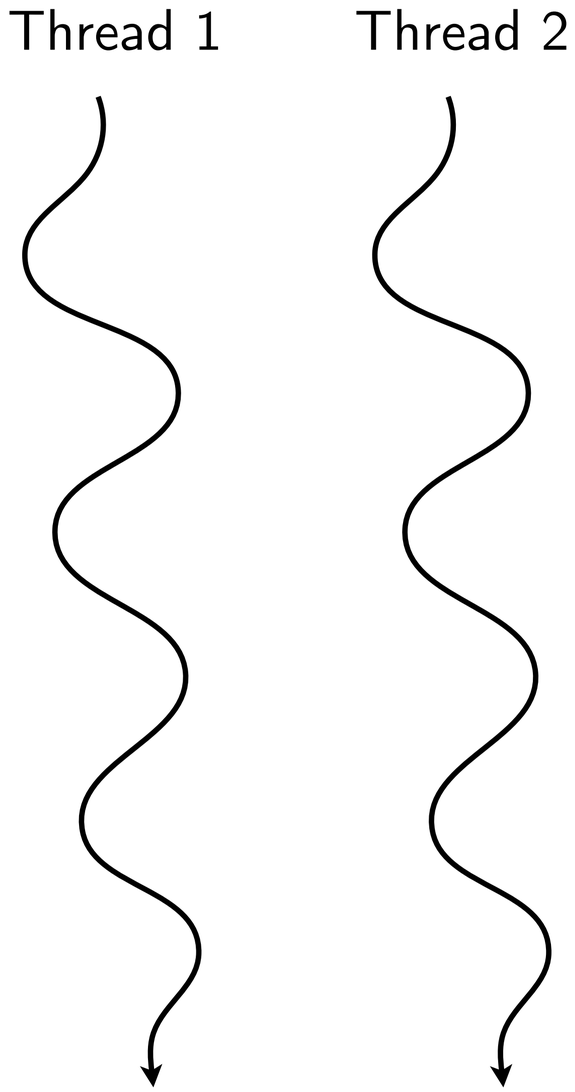
Number of possible interleavings of two threads

If two threads have m and n instructions respectively, then the number interleavings between them is given by:

$$\frac{(m+n)!}{m! \times n!}$$

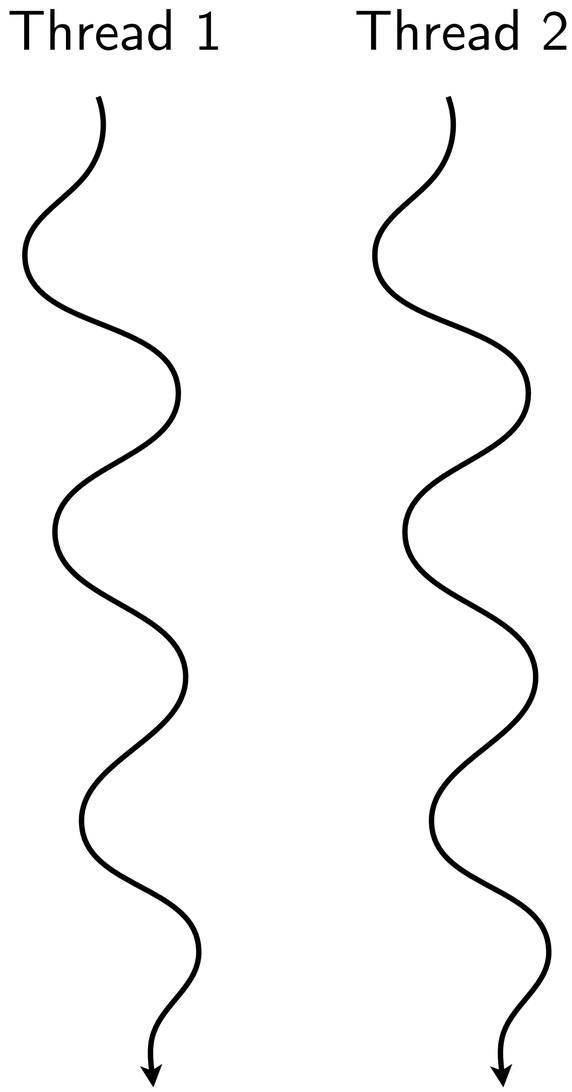
$m = n = 2$	$m = n = 4$	$m = n = 8$	$m = n = 16$
6	70	$13K$	$601M$

Observations on practical interleaving tracking



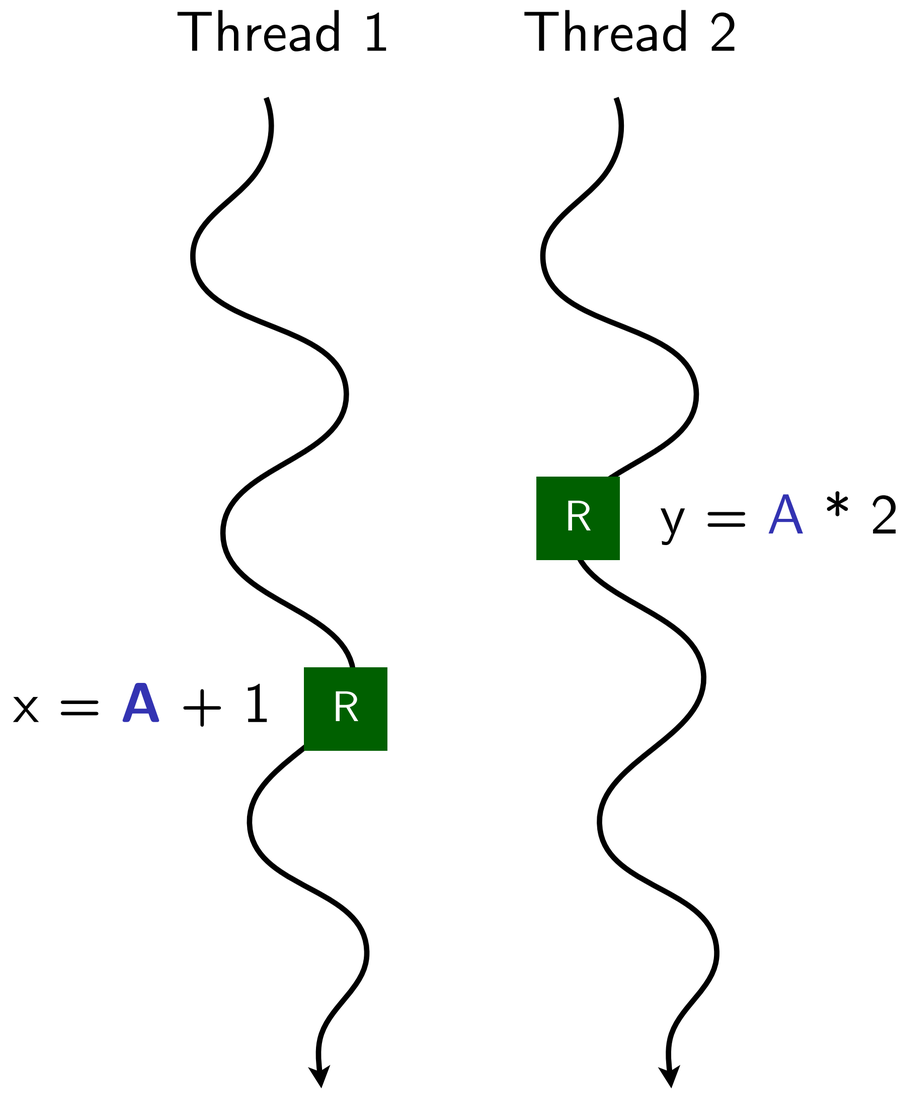
Observations on practical interleaving tracking

- Only interleaved accesses to shared memory matters
  - In an extreme case where two threads do not shared memory, they interleaving does not matter at all.



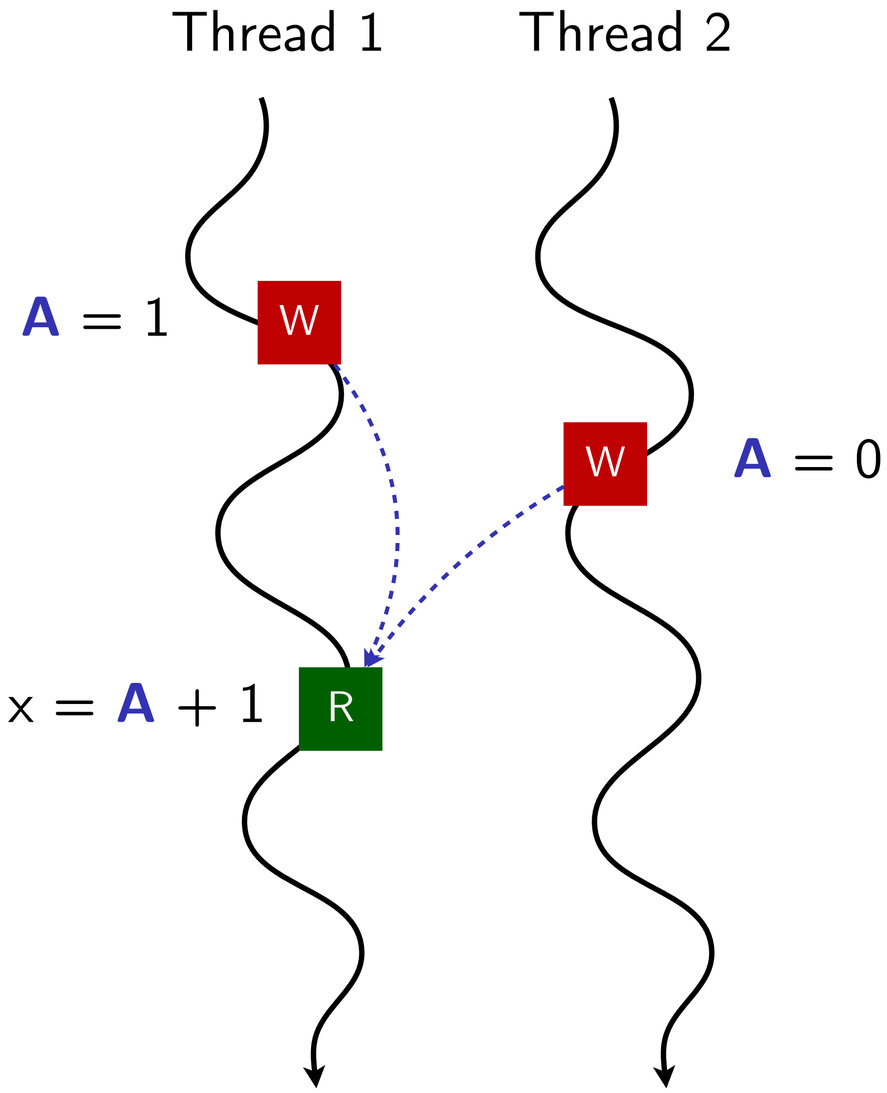
Observations on practical interleaving tracking

- Only interleaved accesses to shared memory matters
  - In an extreme case where two threads do not shared memory, they interleaving does not matter at all.
- Only interleaved **read-write** accesses to shared memory locations matters
  - In an extreme case where two threads only read from shared memory, they interleaving does not matter at all.



Observations on practical interleaving tracking

- Only interleaved accesses to shared memory matters
  - In an extreme case where two threads do not shared memory, they interleaving does not matter at all.
- Only interleaved **read-write** accesses to shared memory locations matters
  - In an extreme case where two threads only read from shared memory, they interleaving does not matter at all.
- Thread interleaving alters the **def-use relation** of memory locations!



Observations on practical interleaving tracking

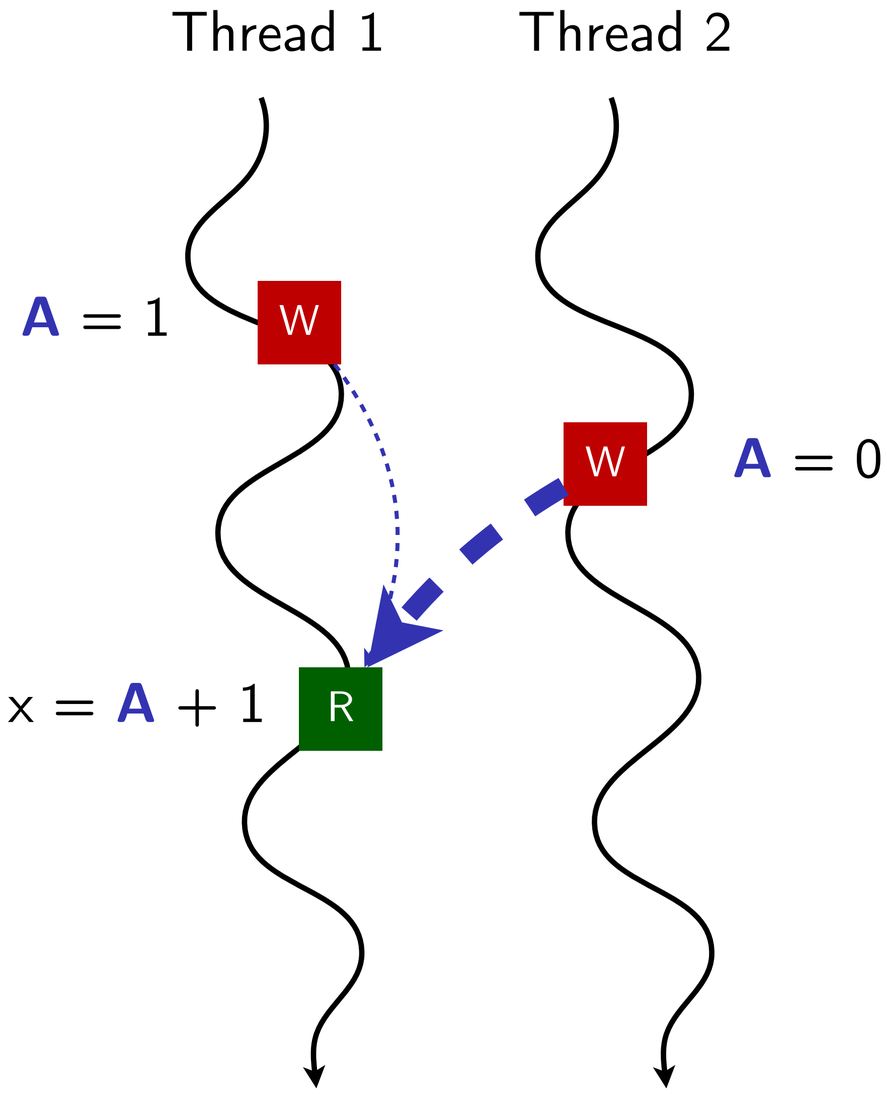
- Only interleaved accesses to shared memory matters
  - In an extreme case where two threads do not shared memory, they interleaving does not matter at all.

Interleaving approximation

Track cross-thread write-to-read (def-to-use) edges!

- Only interleaved accesses to shared memory matters
  - In an extreme case where two threads only read from shared memory, they interleaving does not matter at all.

- Thread interleaving alters the **def-use relation** of memory locations!

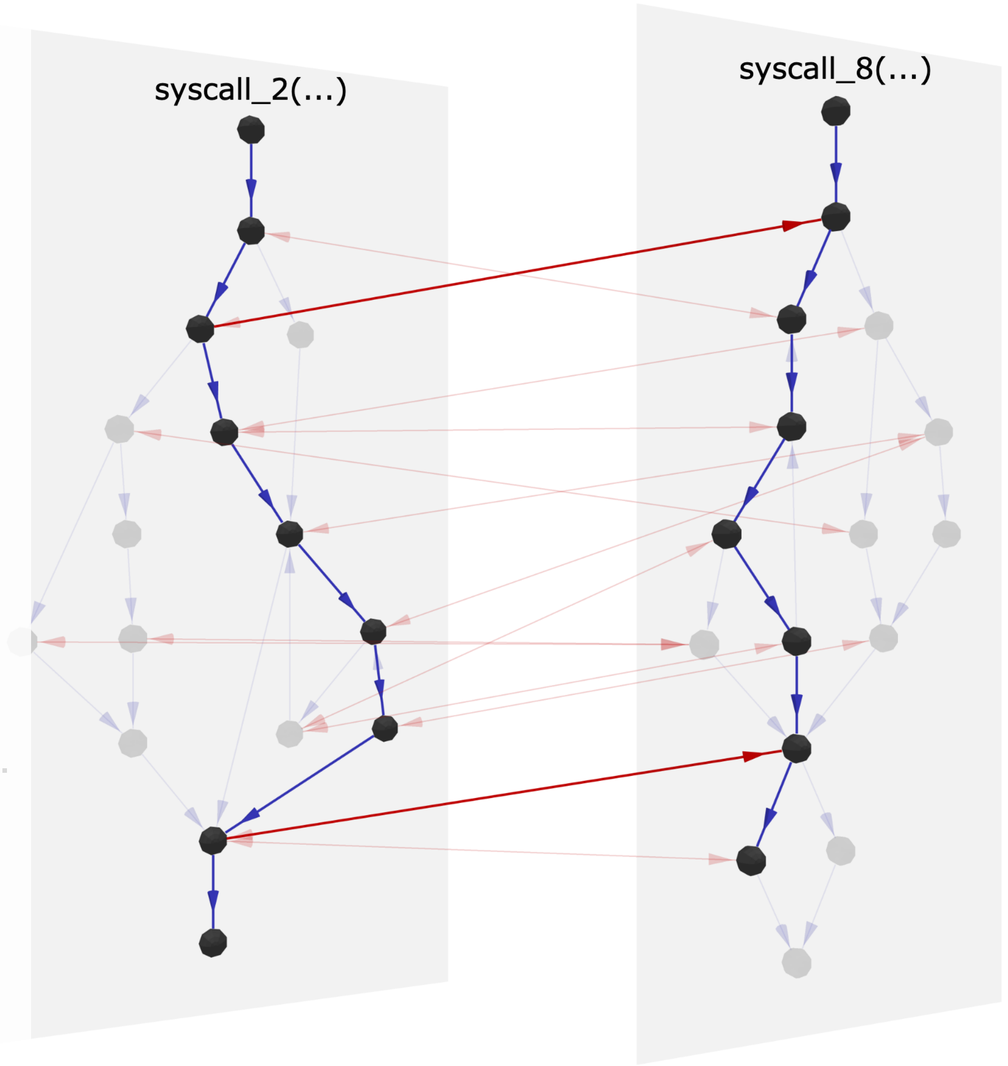


Observations on practical interleaving tracking

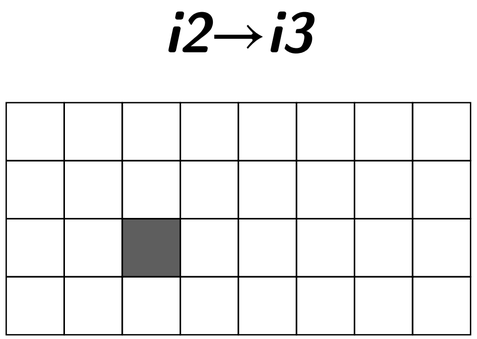
- Only interleaved accesses to shared memory matters
  - In an extreme case where two threads do not shared memory, they interleaving does not matter at all.
- Only interleaved accesses to shared memory matters
  - In an extreme case where two threads only read from shared memory, they interleaving does not matter at all.
- Thread interleaving alters the **def-use relation** of memory locations!

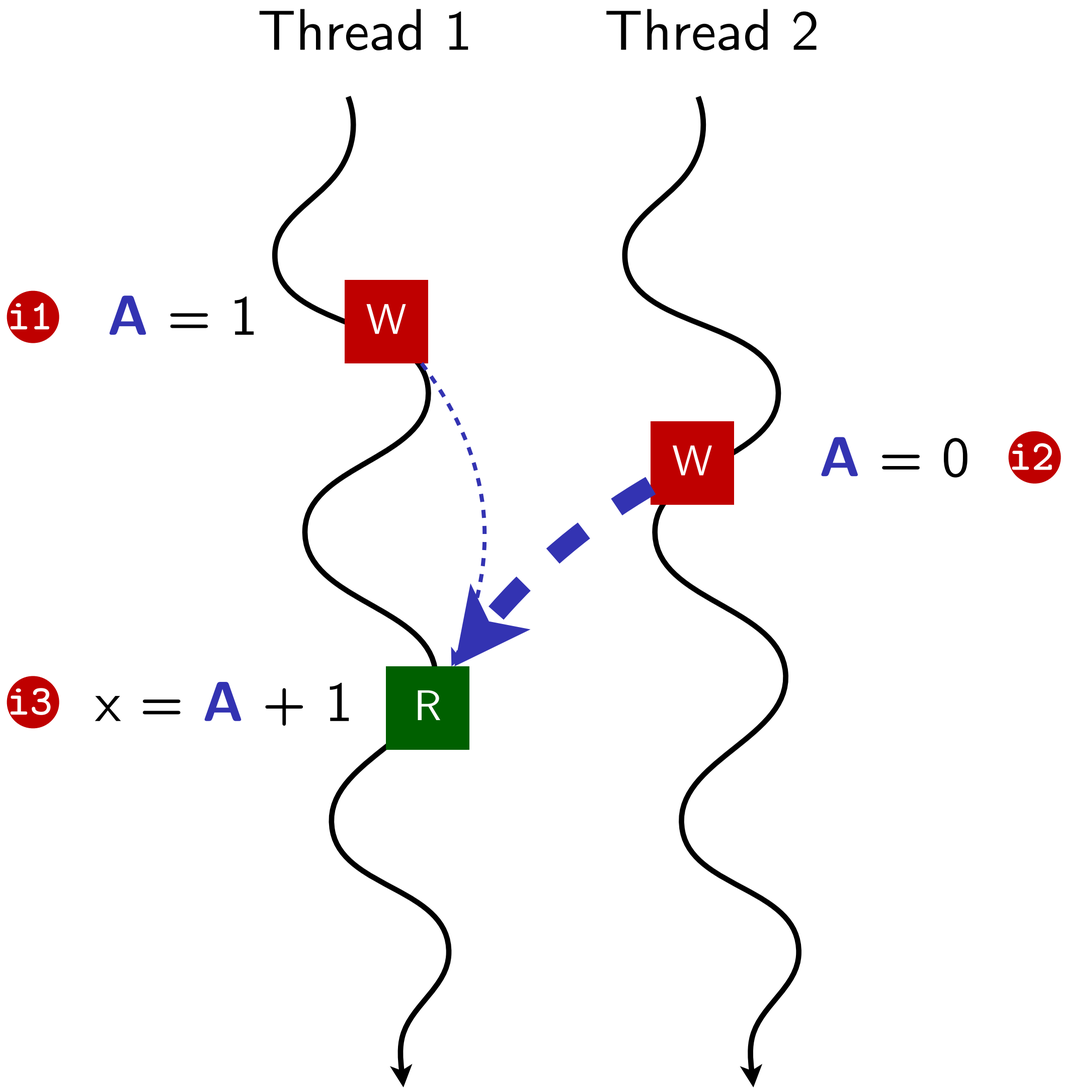
Interleaving approximation

Track cross-thread write-to-read (def-to-use) edges!



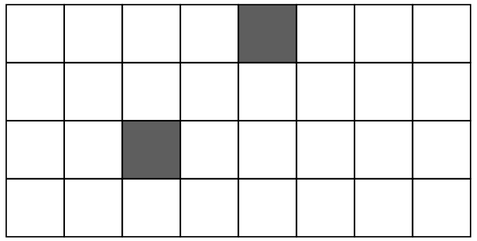
Aliased-instruction coverage

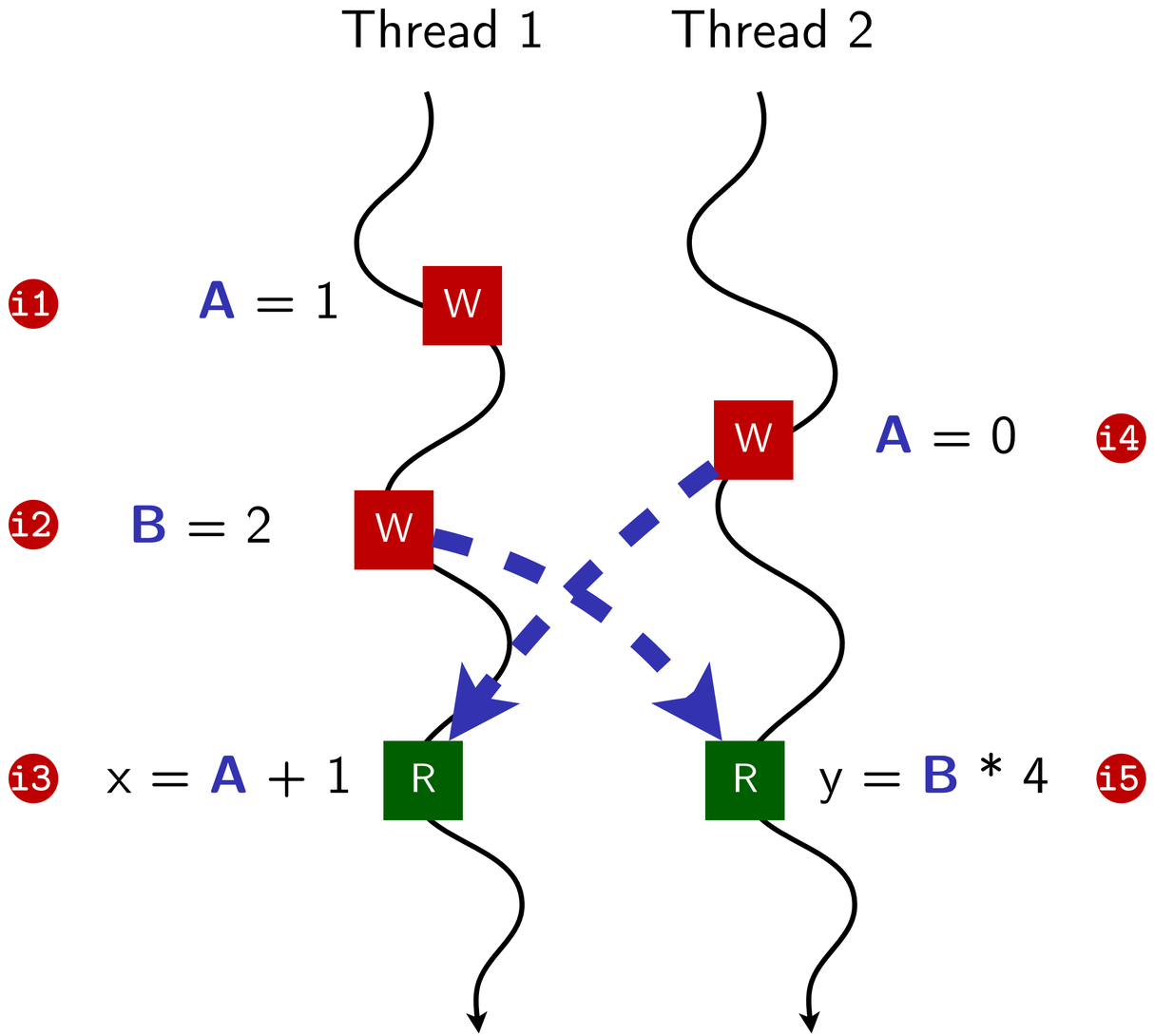




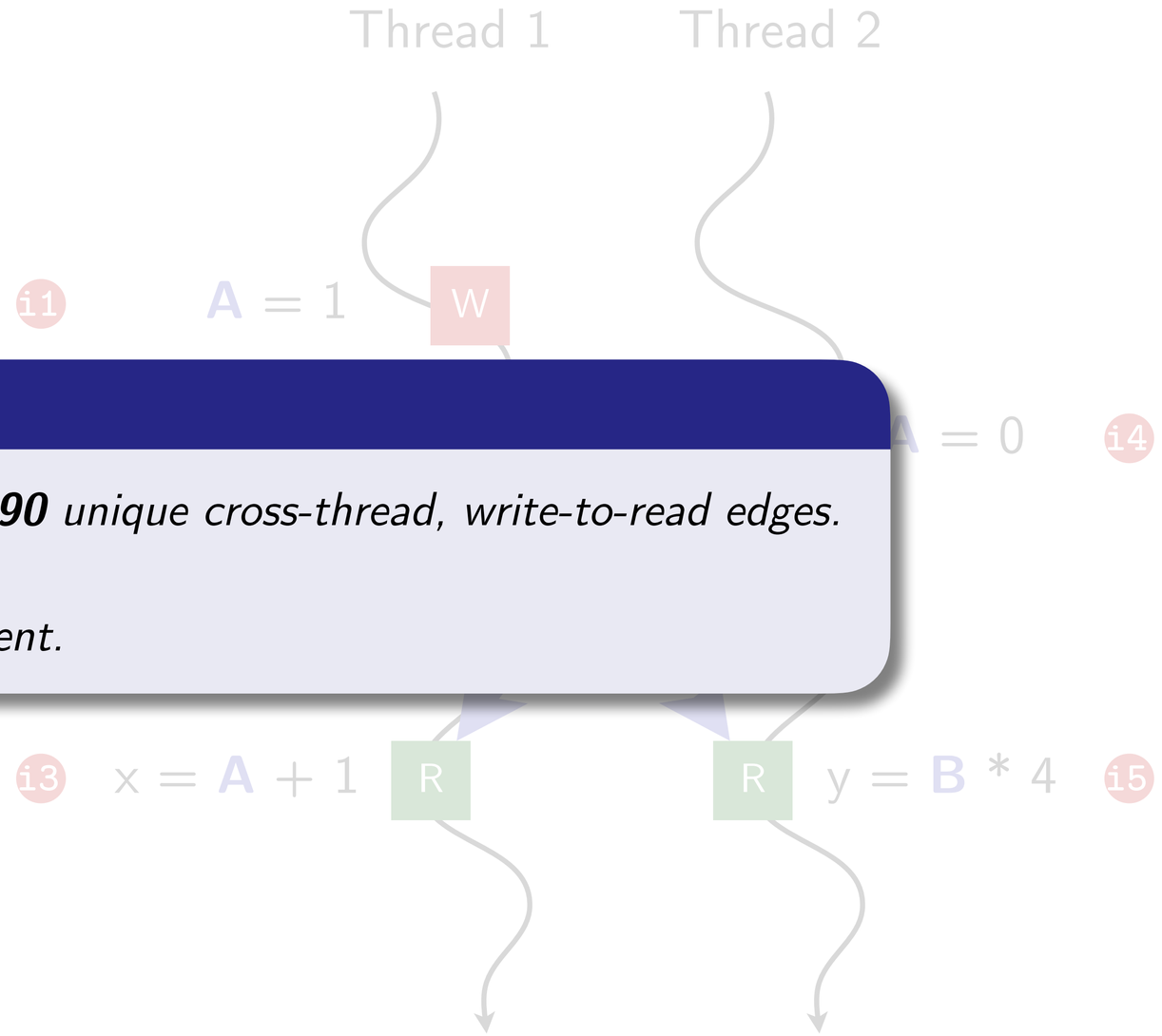
Aliased-instruction coverage

i2→i5, i4→i3





Aliased-instruction coverage

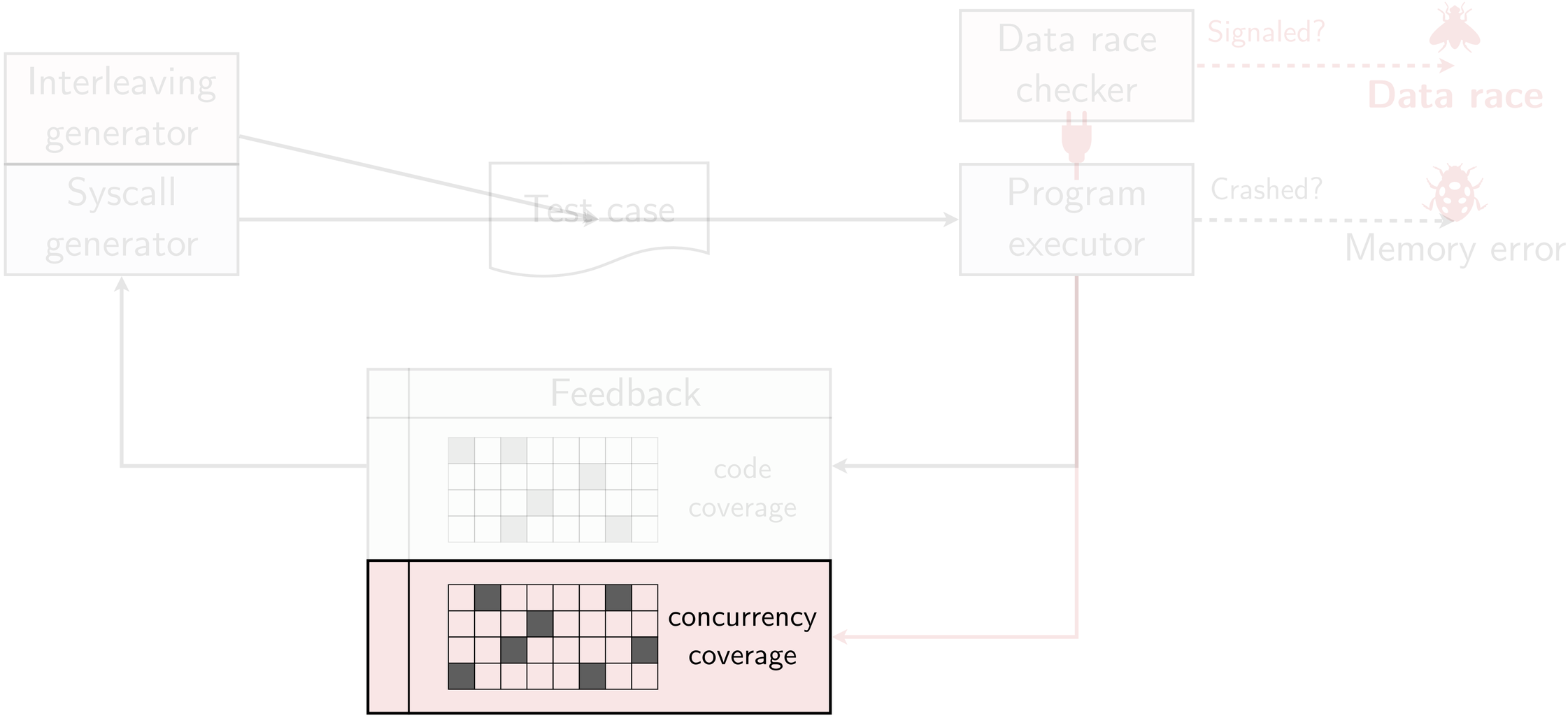


Concurrency coverage bitmap size

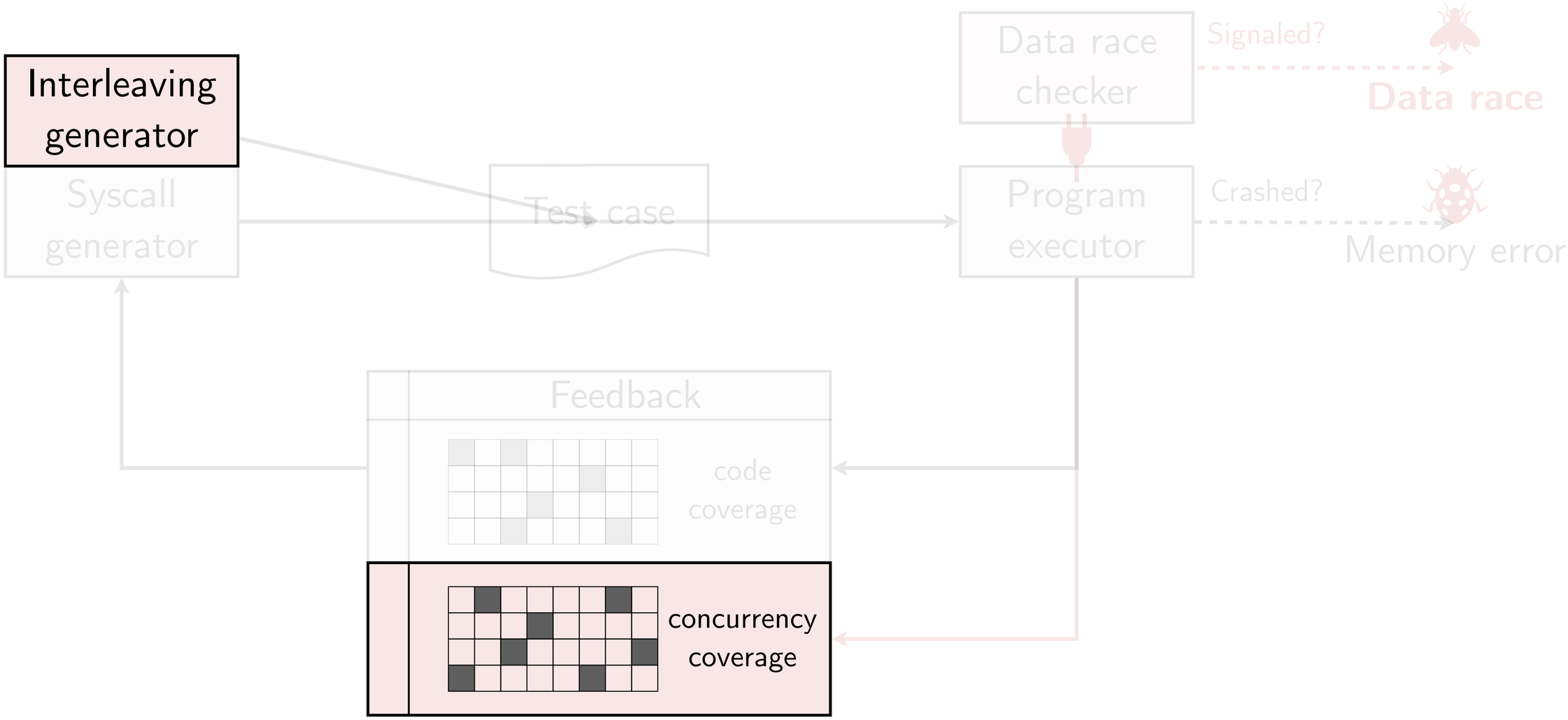
*During our experiment, we observed **63,590** unique cross-thread, write-to-read edges.*

→ *a bitmap size of **128KB** will be sufficient.*

Concurrency coverage tracking

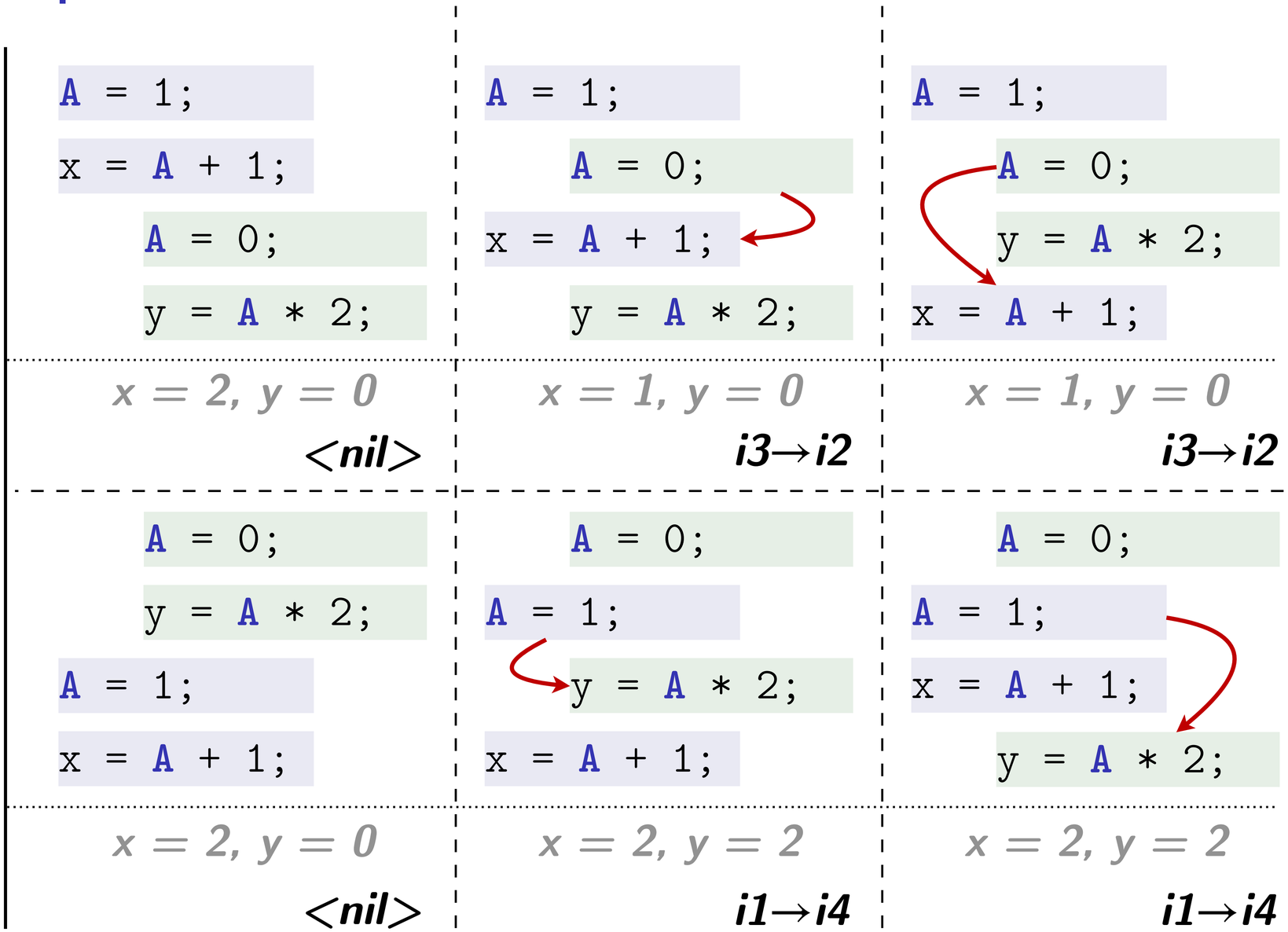


Interleaving exploration

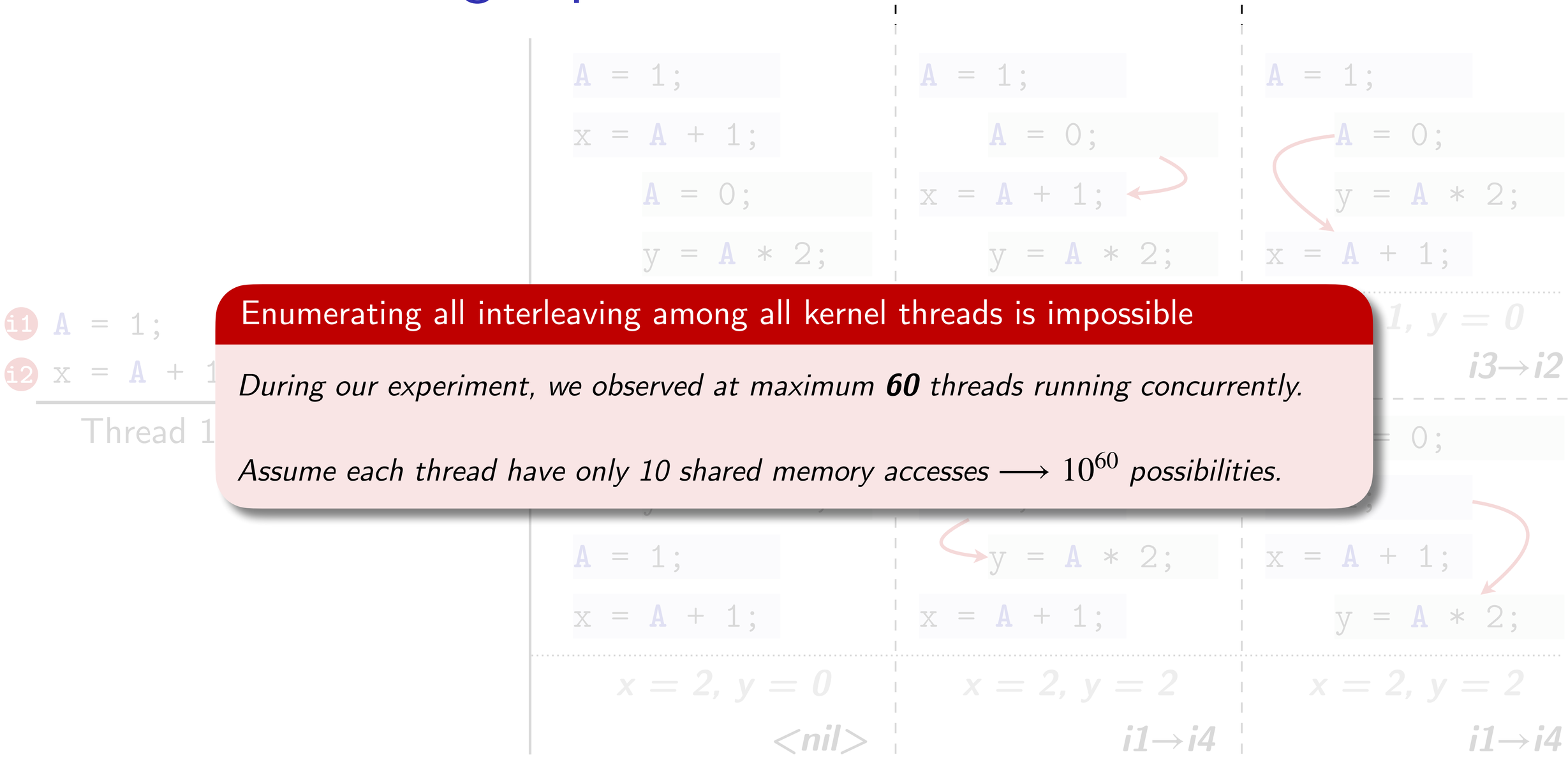


Active interleaving exploration - ideal case

Thread 1	Thread 2
i1 `A = 1;`	`A = 0;` i3
i2 `x = A + 1;`	`y = A * 2;` i4



Active interleaving exploration - ideal case



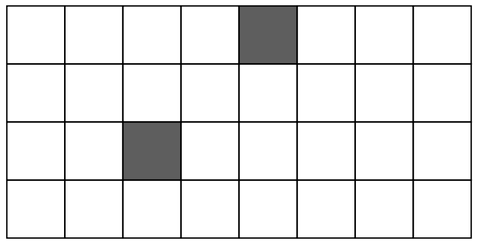
Enumerating all interleaving among all kernel threads is impossible

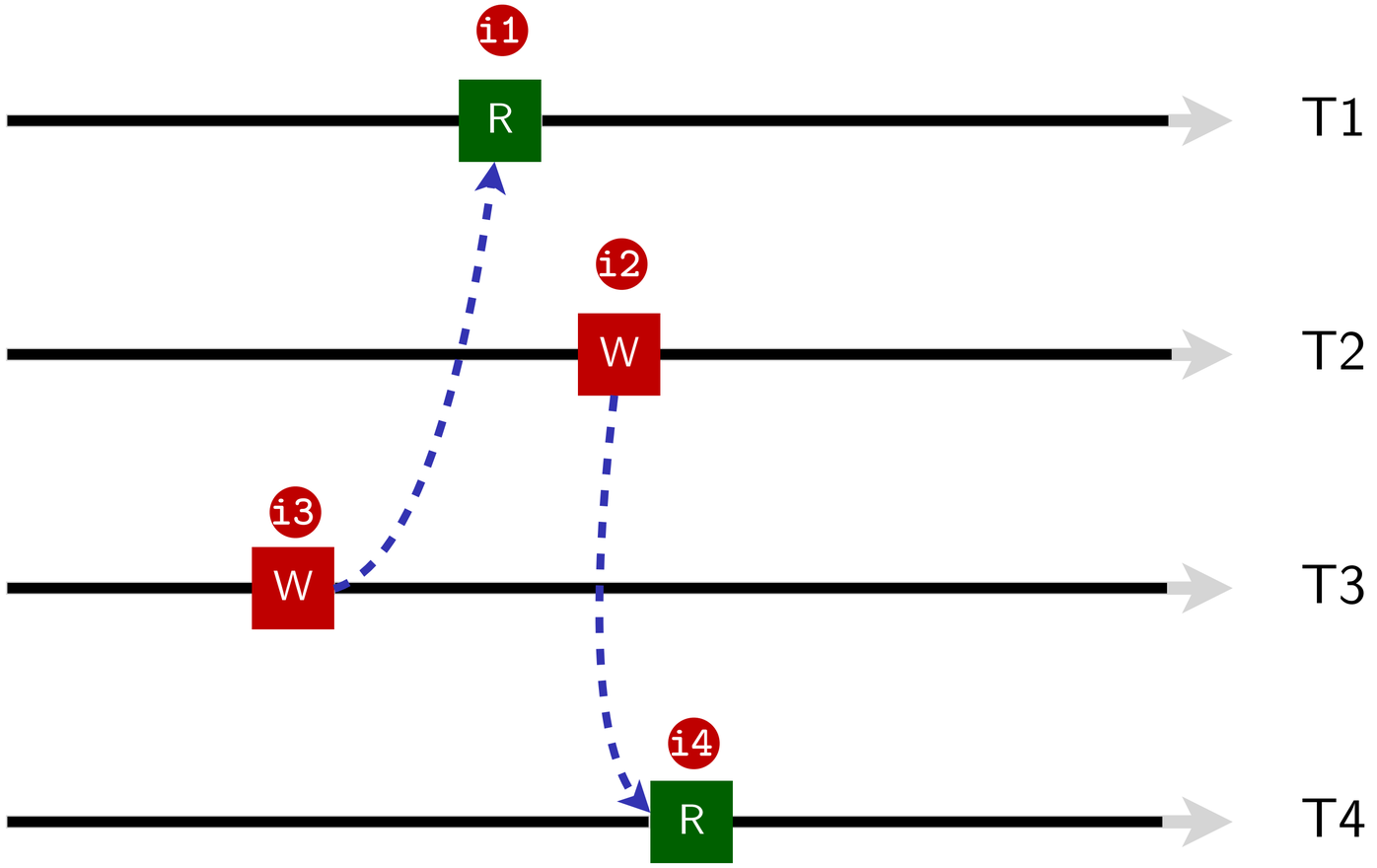
*During our experiment, we observed at maximum **60** threads running concurrently.*

Assume each thread have only 10 shared memory accesses $\longrightarrow 10^{60}$ *possibilities.*

Active interleaving exploration through delay injection

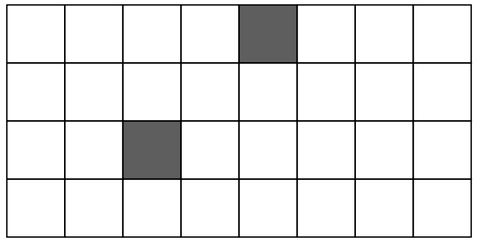
Concurrency coverage

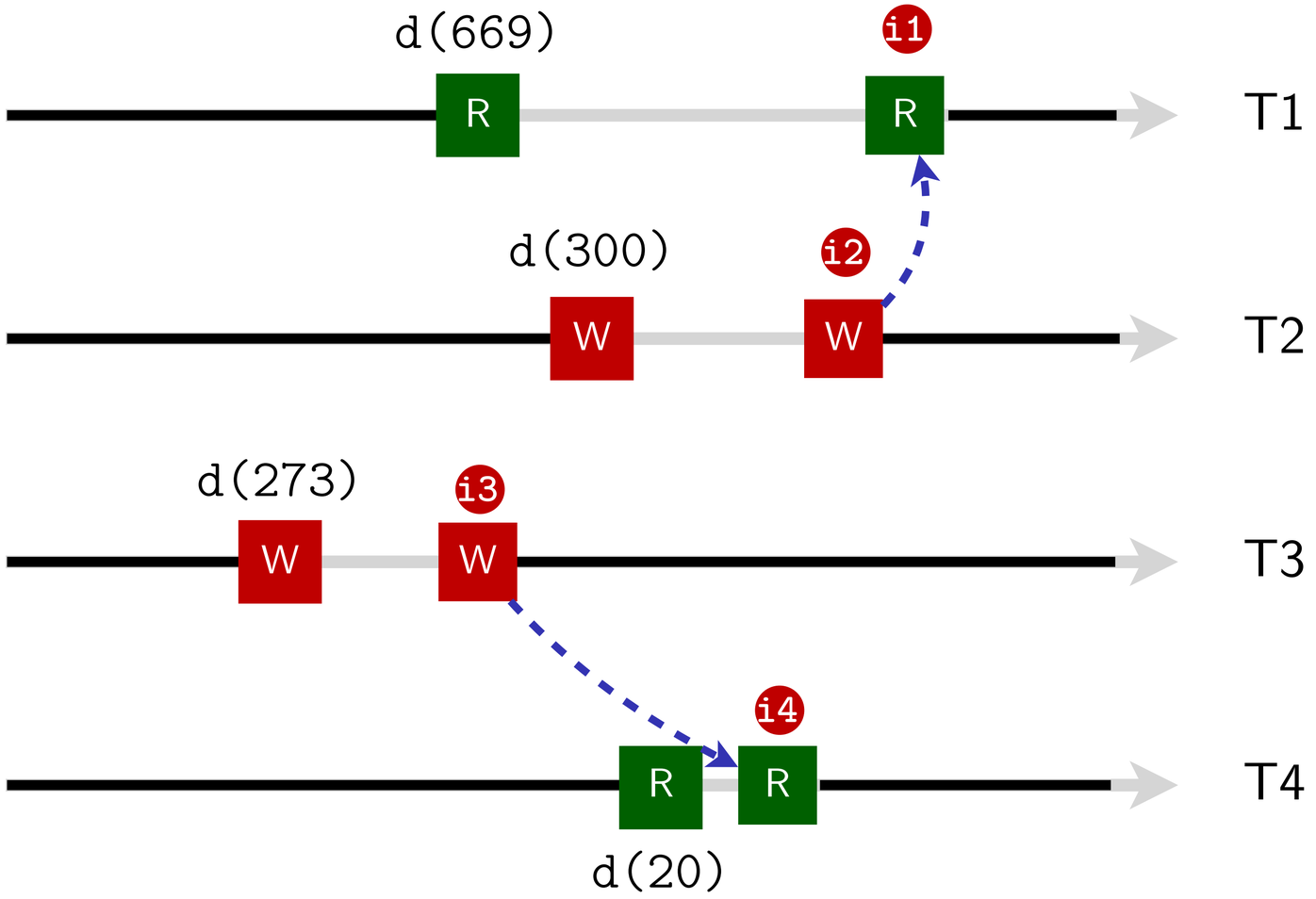




Active interleaving exploration through delay injection

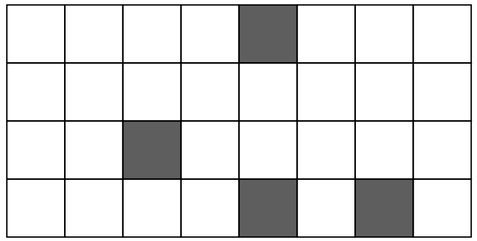
Concurrency coverage

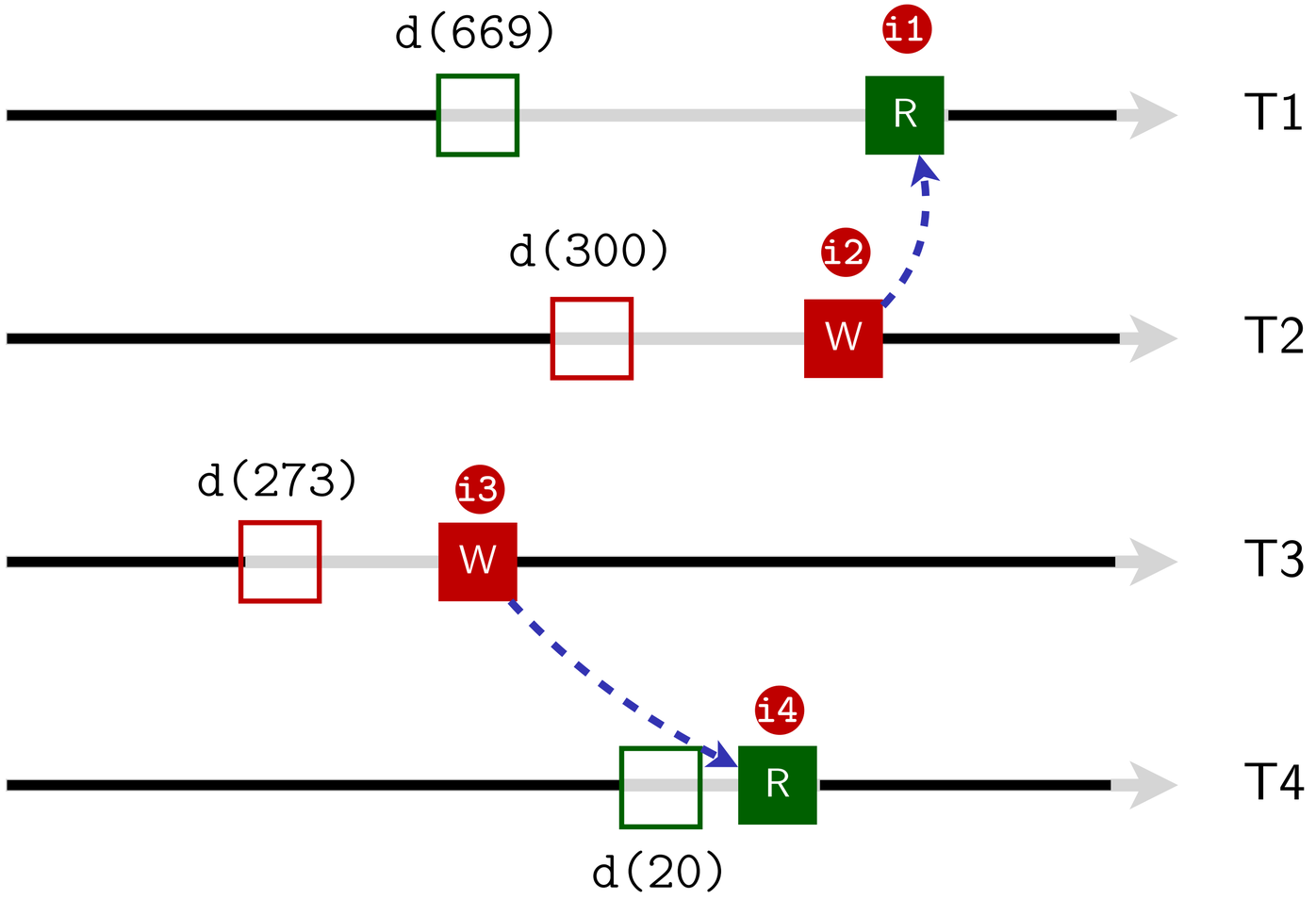




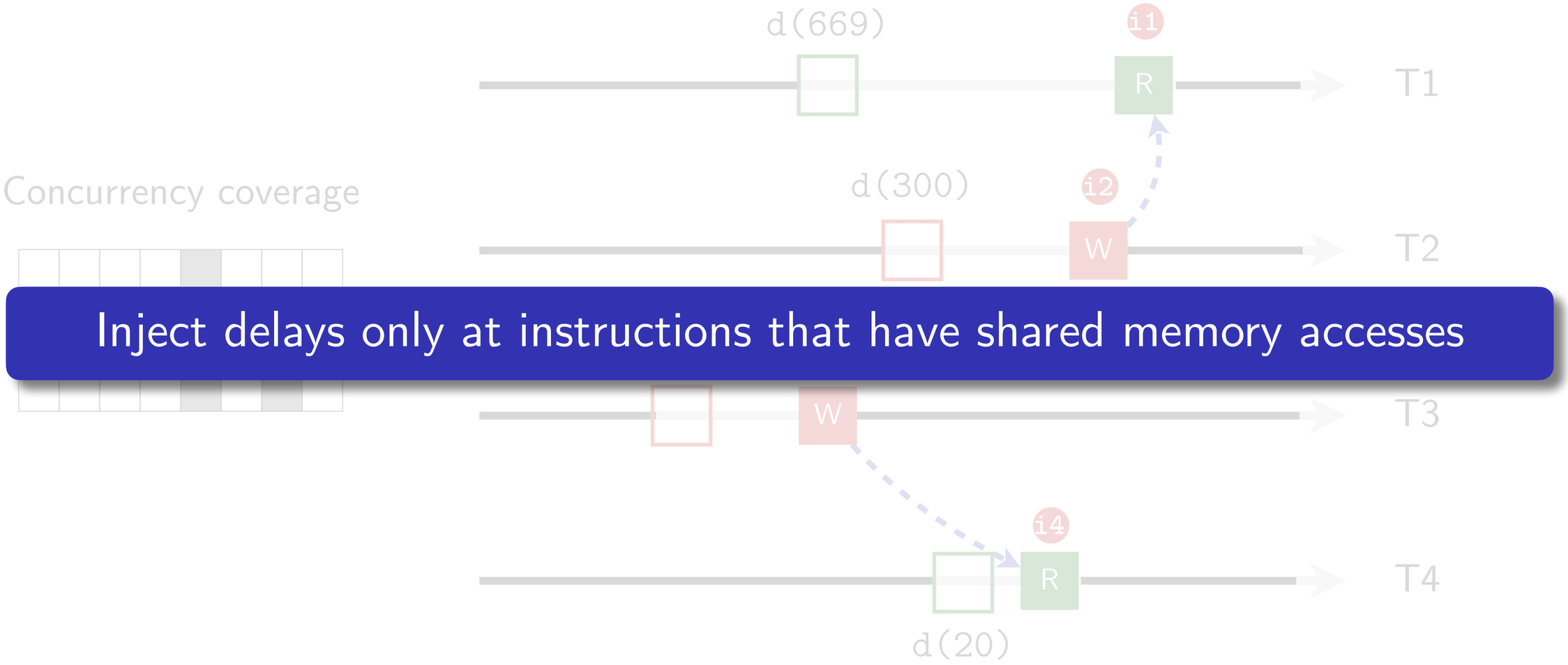
Active interleaving exploration through delay injection

Concurrency coverage

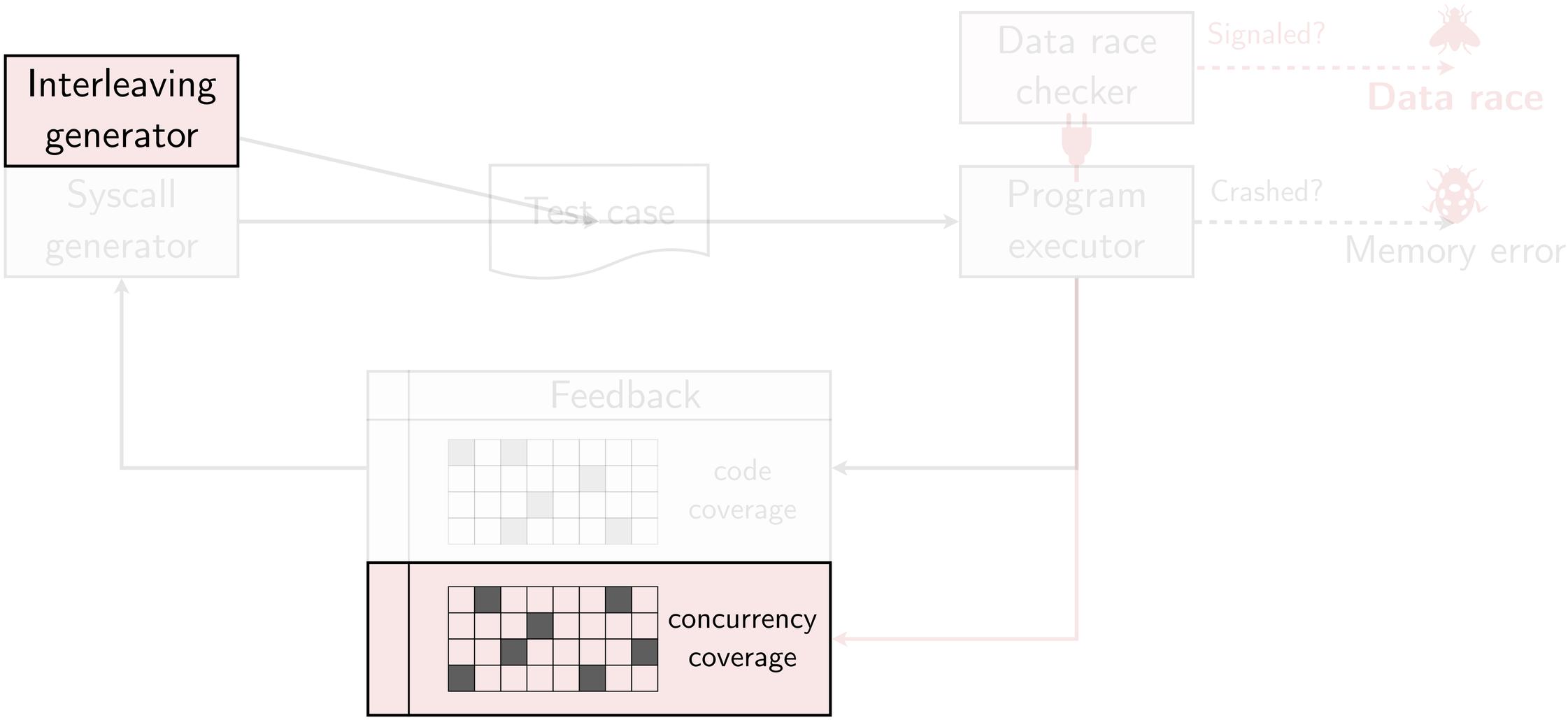




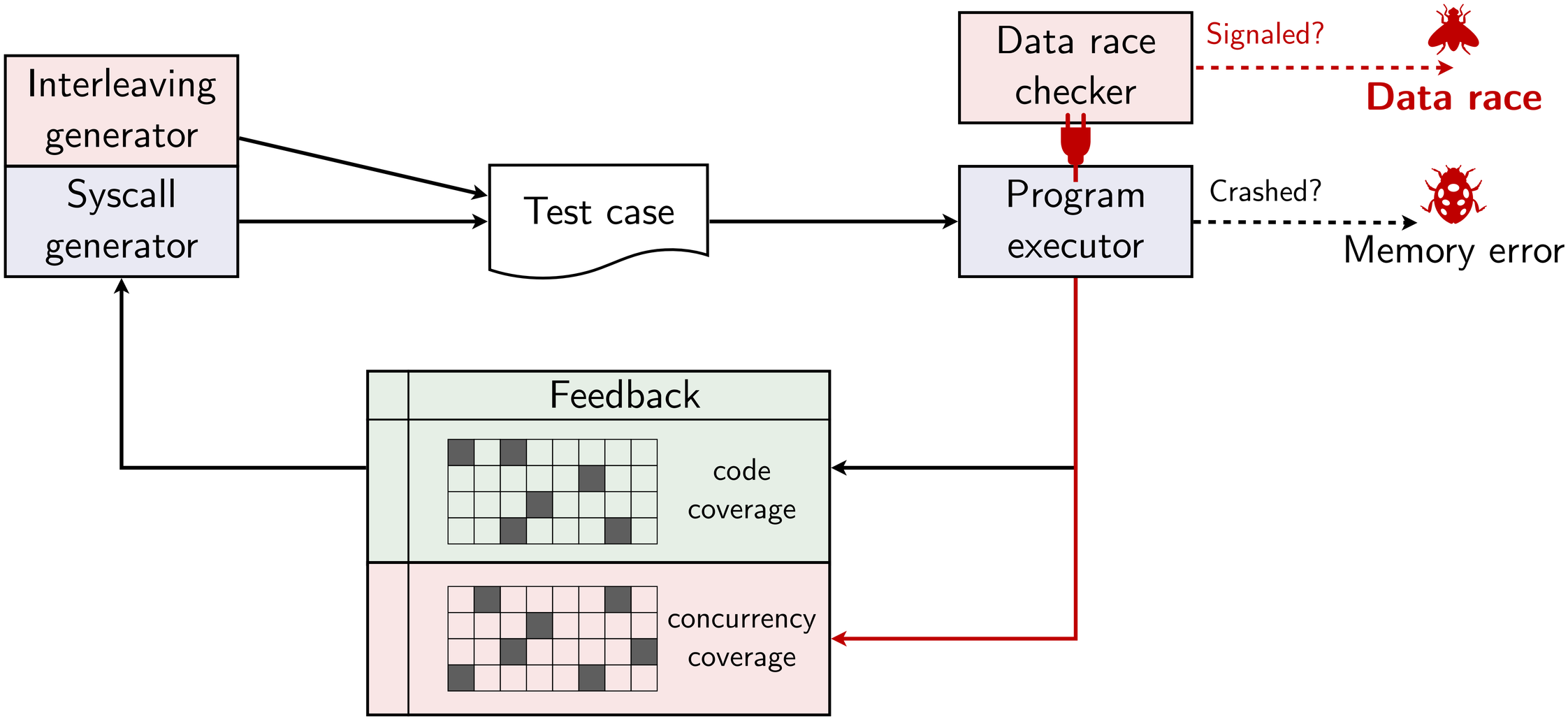
Active interleaving exploration through delay injection



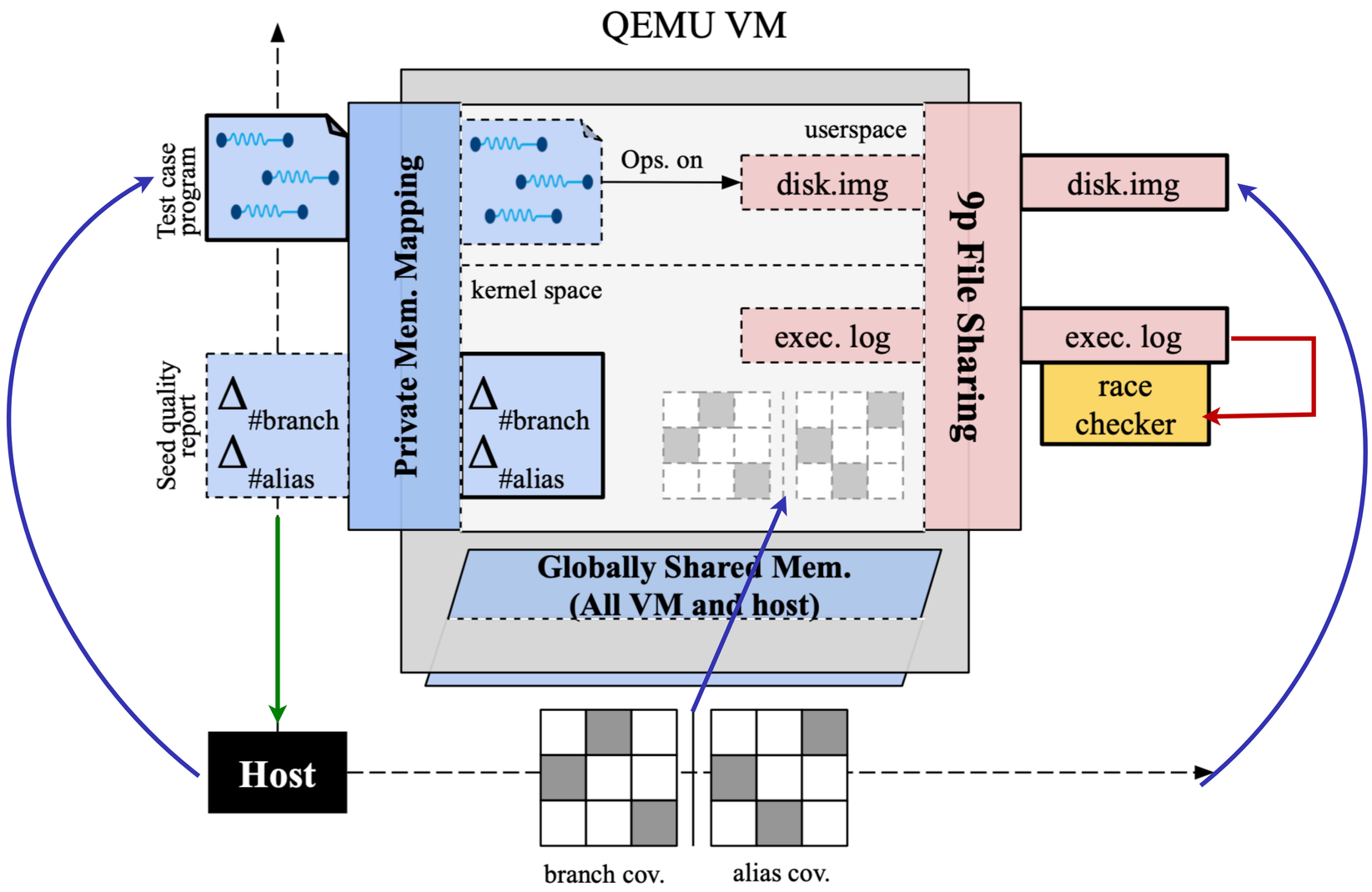
Interleaving exploration



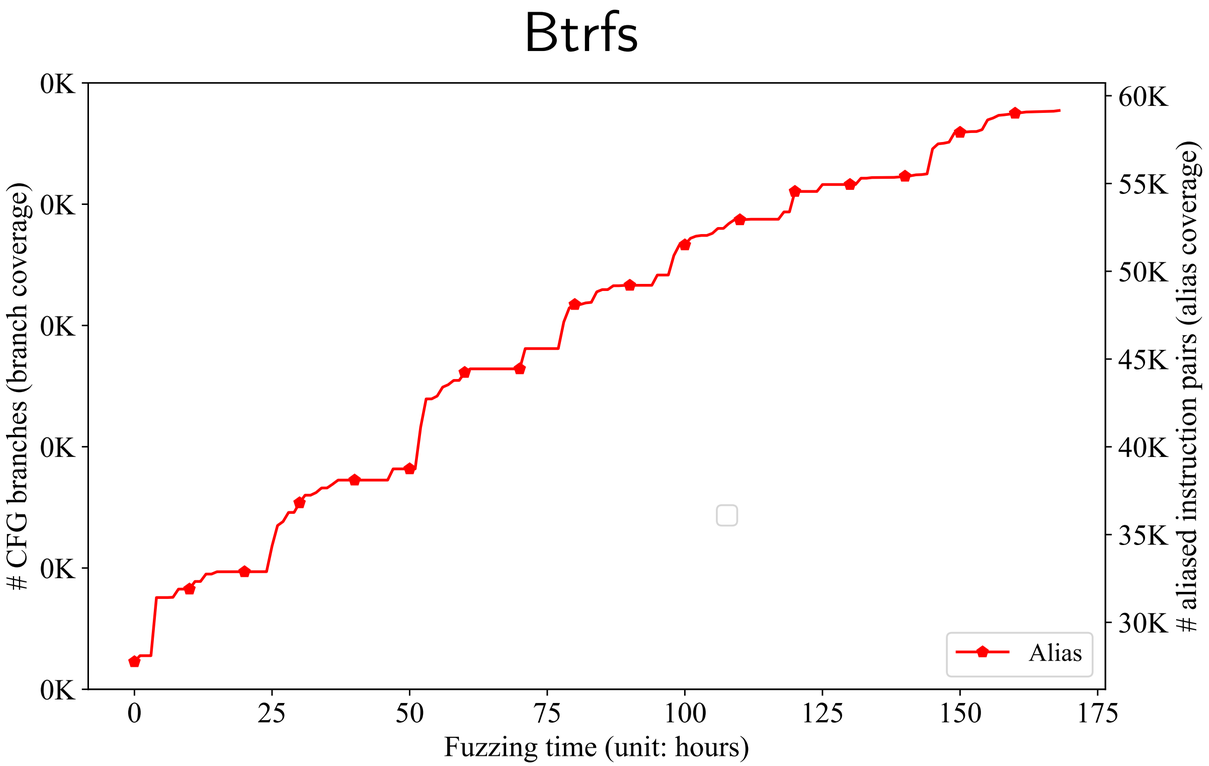
Bring them all together

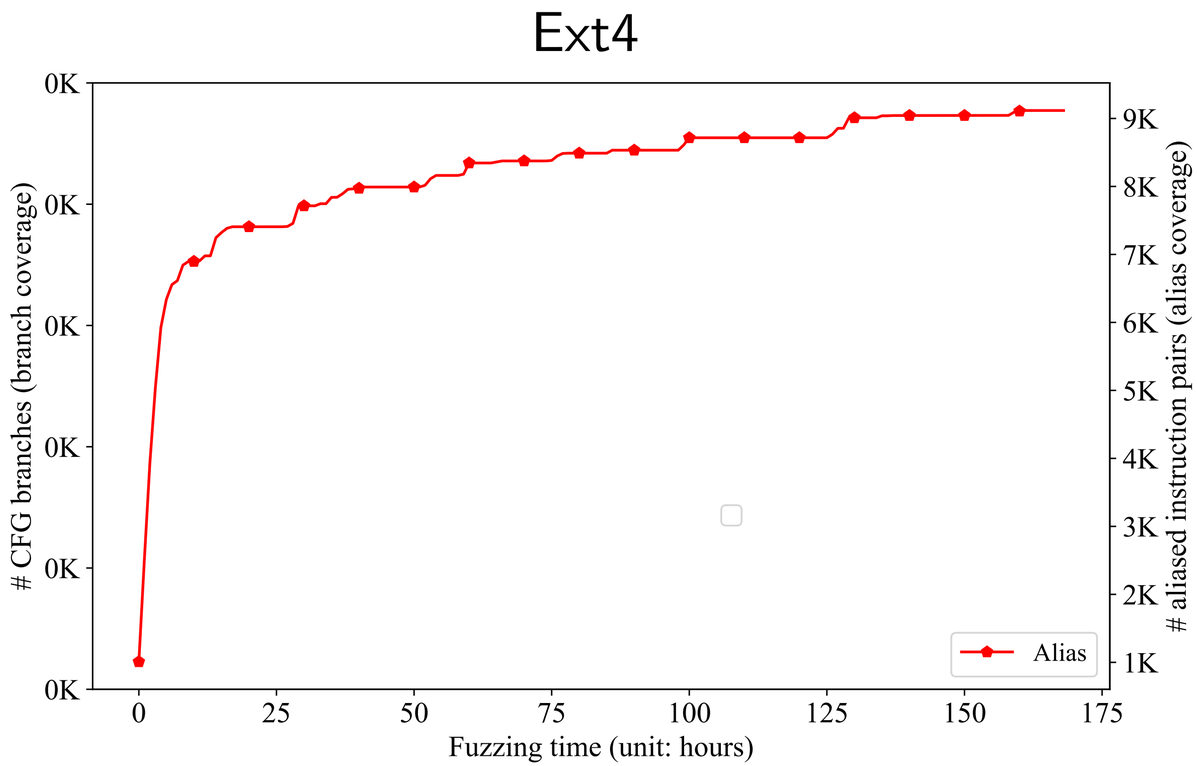


QEMU-based implementation



Alias coverage growth will be saturating

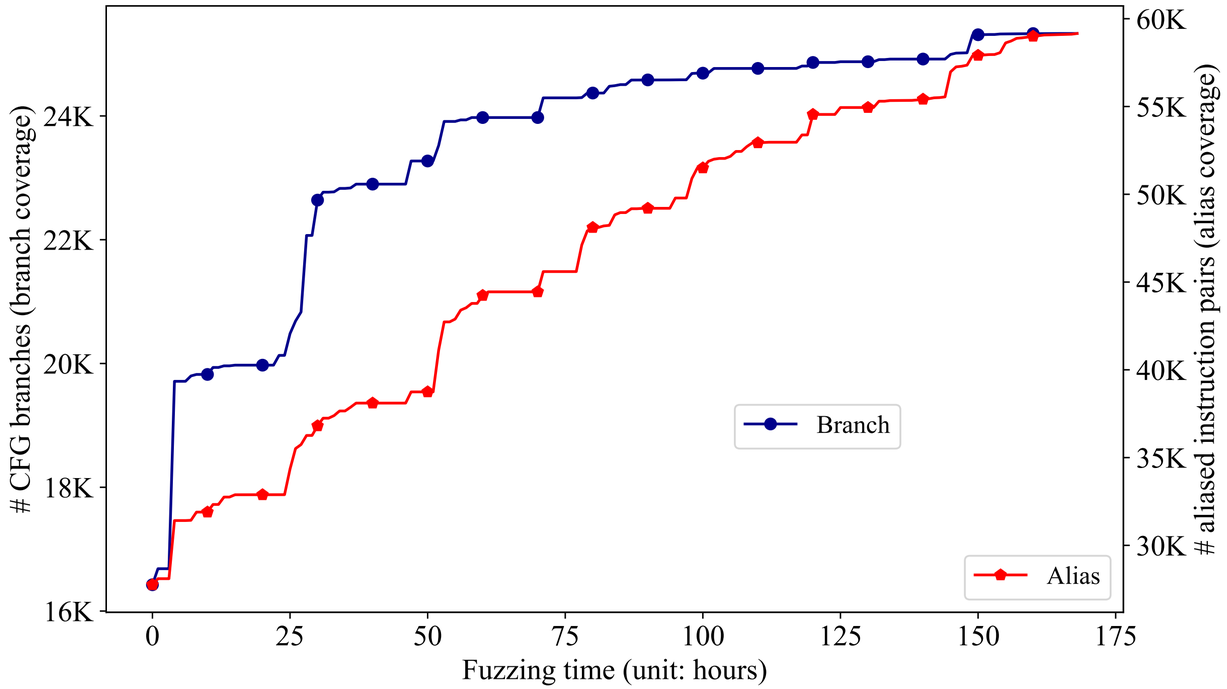




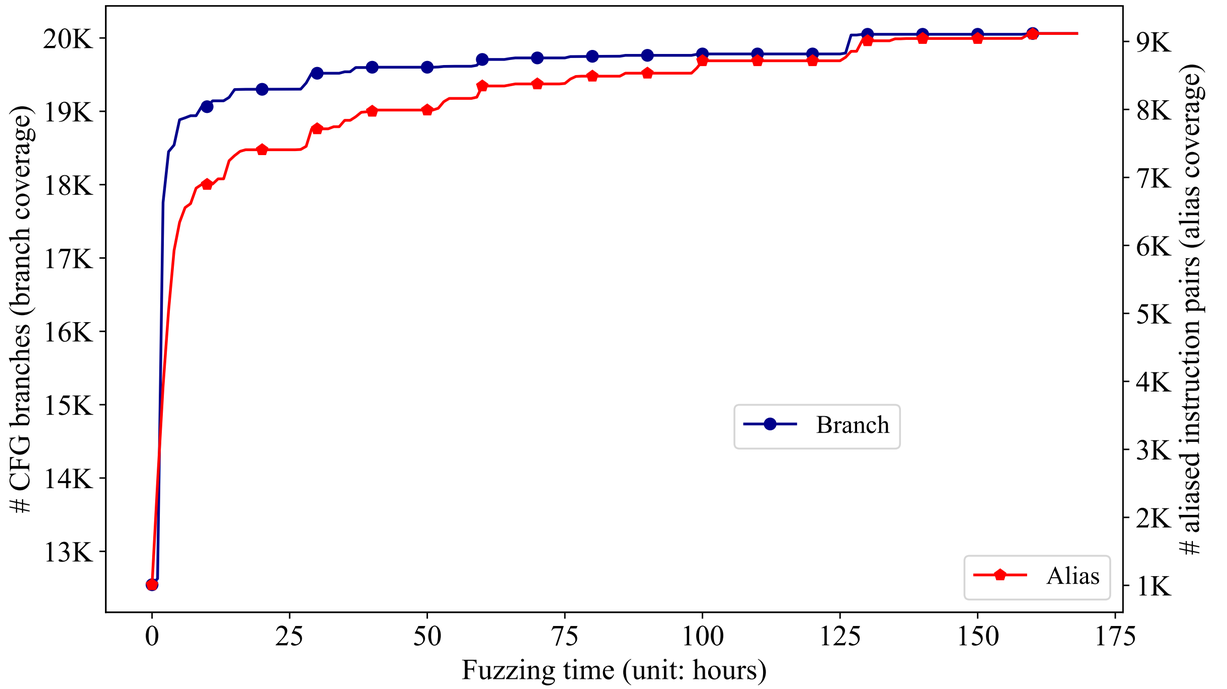
But file systems that are higher in concurrency level saturates much slower!

Edge and alias coverage goes generally in synchronization

Btrfs

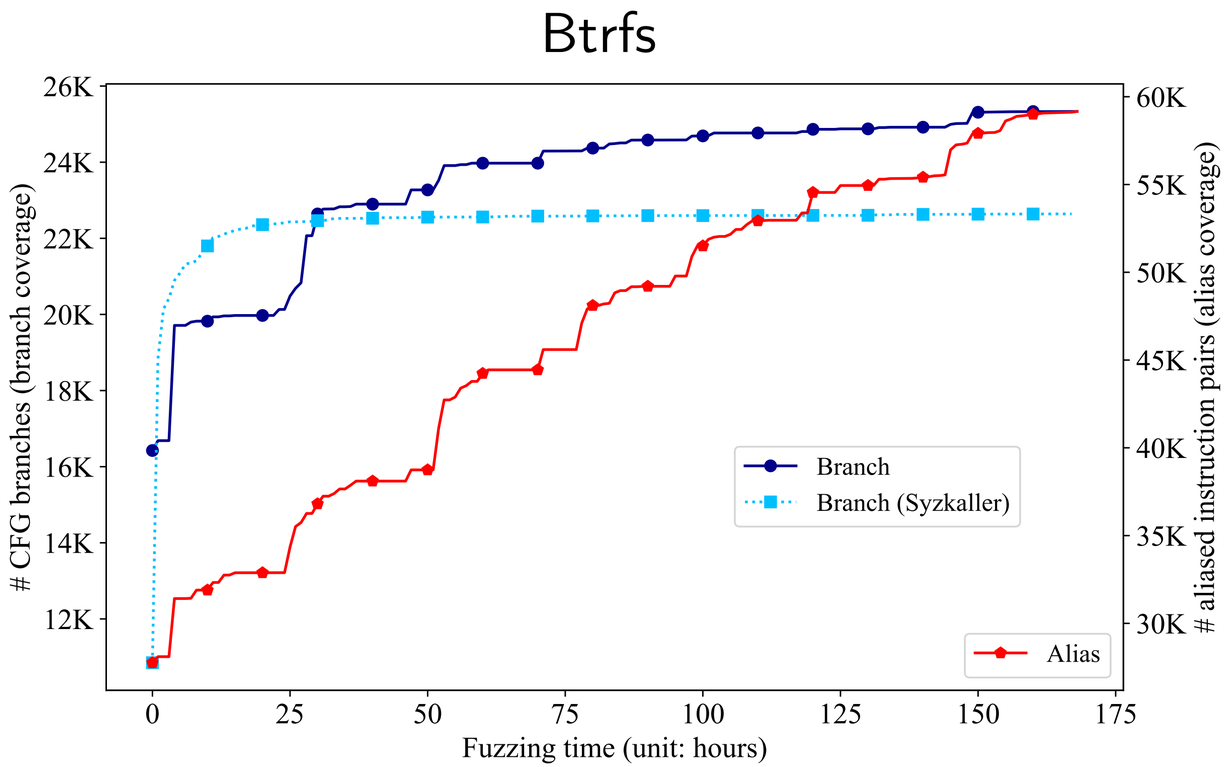


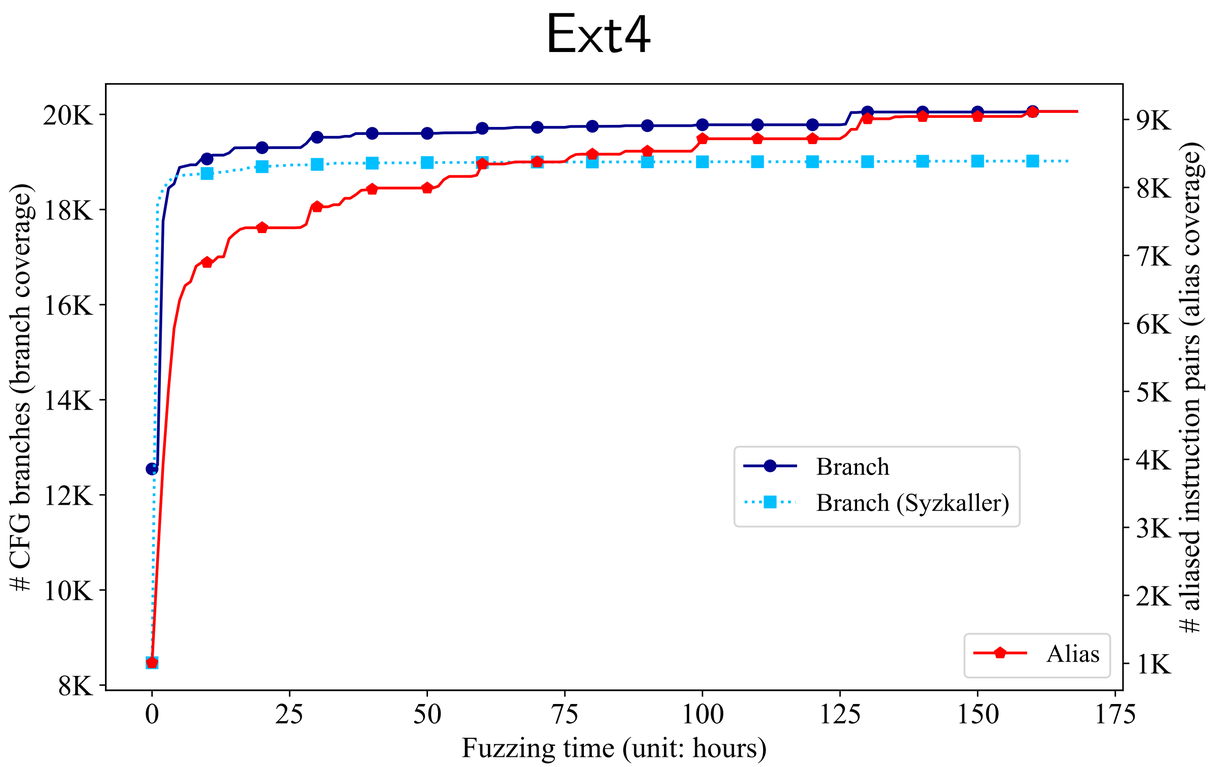
Ext4



But there will be time when the edge coverage saturates
but alias coverage keeps finding new thread interleaving

Slightly more branch coverage than Syzkaller





This maybe due to the fact that we give each seed more chances (if they make progresses in alias coverage)

Bugs found by Krace

File system	# data races	# harmful confirmed
Btrfs	11	8
Ext4	4	1
VFS	8	2
Total	23	11

Conclusion and contribution

Structured input	**Application**
[Google] Syzkaller [SP'19] Janus [ICSE'19] SLF	[CCS'17] SlowFuzz [ICSE'19] DifFuzz [VLDB'20] Apollo .....
Seed selection	**Coverage metric**
[CCS'16] AFLFast [ASE'18] FairFuzz [FSE'19] Fudge	[SP'18] Angora [RAID'19] Benchmark **[SP'20] Krace**

Conclusion and contribution

Structured input

- [Google] Syzkaller
- [SP'19] Janus
- [ICSE'19] SLF
- .....

Application

- [CCS'17] SlowFuzz
- [ICSE'19] DifFuzz
- [VLDB'20] Apollo
- .....

Seed selection

- [CCS'16] AFLFast
- [ASE'18] FairFuzz
- [FSE'19] Fudge
- .....

Coverage metric

- [SP'18] Angora
- [RAID'19] Benchmark
- **[SP'20] Krace**