Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis

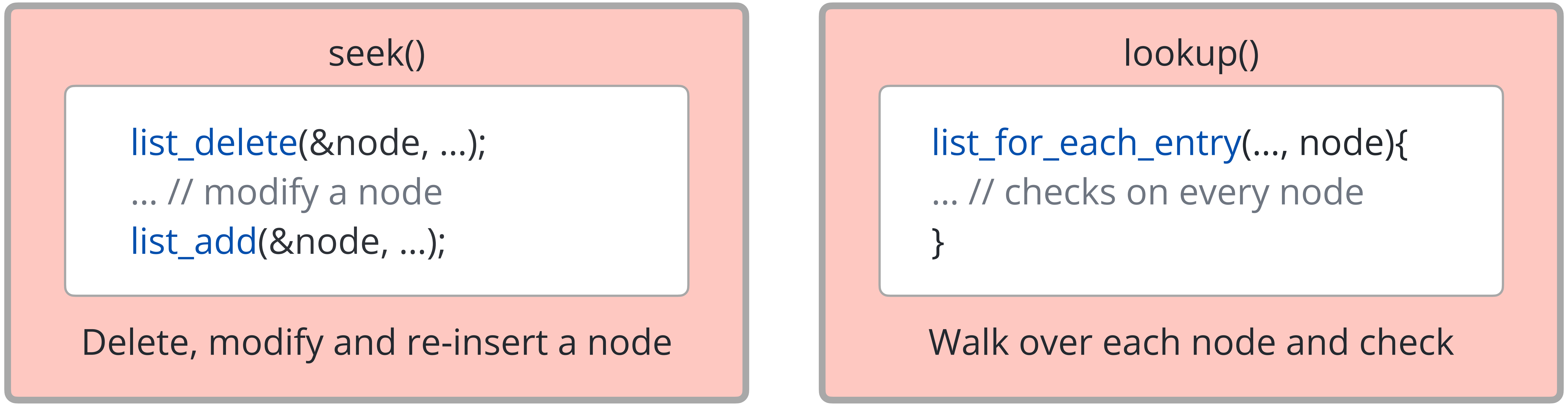
Sishuai Gong (Purdue University)

Deniz Altınbüken (Google Research)

Pedro Fonseca (Purdue University)

Petros Maniatis (Google Research)

A Linux kernel concurrency bug



seek()

```
list_delete(&node, …);
… // modify a node
list_add(&node, …);
```

Delete, modify and re-insert a node

lookup()

```
list_for_each_entry(…, node){
… // checks on every node
}
```

Walk over each node and check

A Linux kernel concurrency bug

Kernel thread 1—running seek()

```
❶ list_delete(&node, ...);
   ... // modify a node
   list_add(&node, ...);
```

Kernel thread 2—running lookup()

```
❷ list_for_each_entry(..., node){
   ... // checks on every node
   }
```

A Linux kernel concurrency bug

Kernel thread 1—running seek()

```
❶ list_delete(&node, ...);
   ... // modify a node
   list_add(&node, ...);
```

Kernel thread 2—running lookup()

```
❷ list_for_each_entry(..., node){
   ... // checks on ev
   }
```

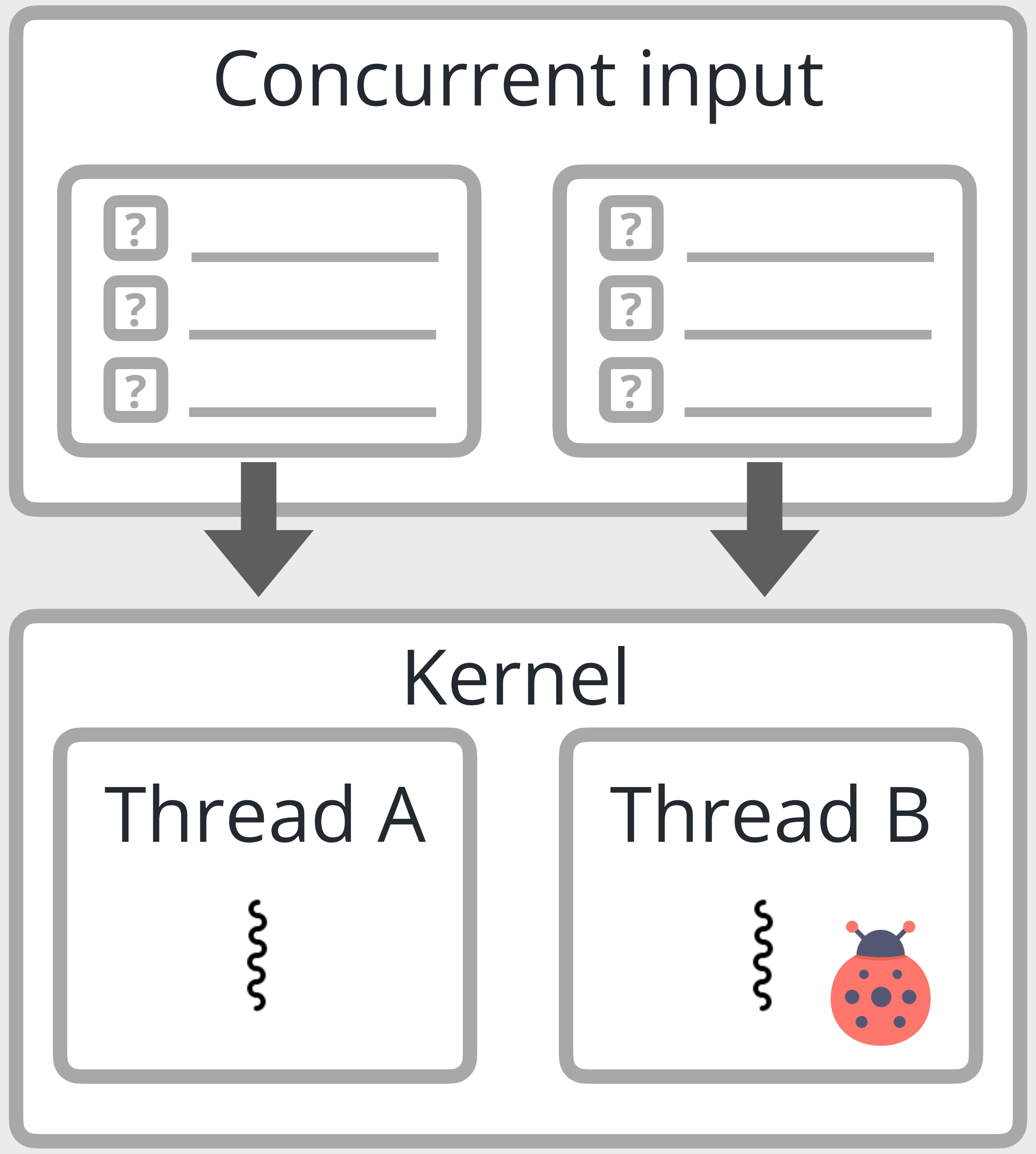
Kernel panic:
Null pointer deference

This bug existed in the kernel for over 14 years

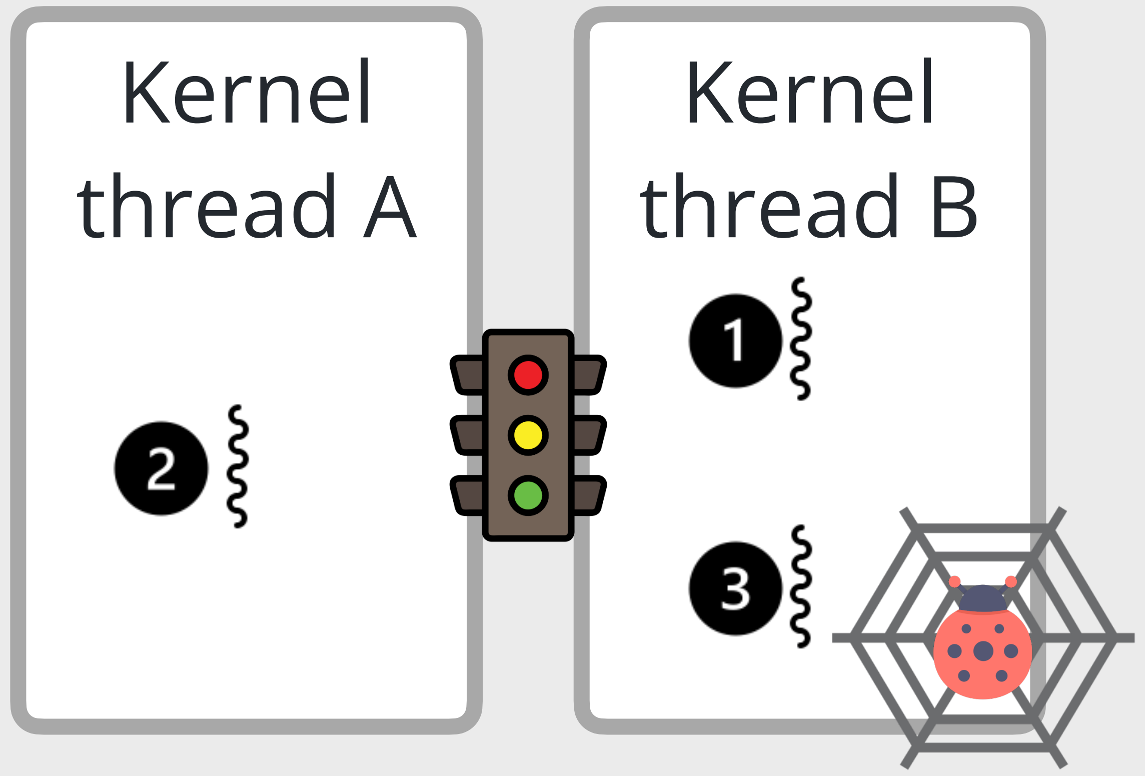
until Snowboard found it :)

Challenges in finding concurrency bugs

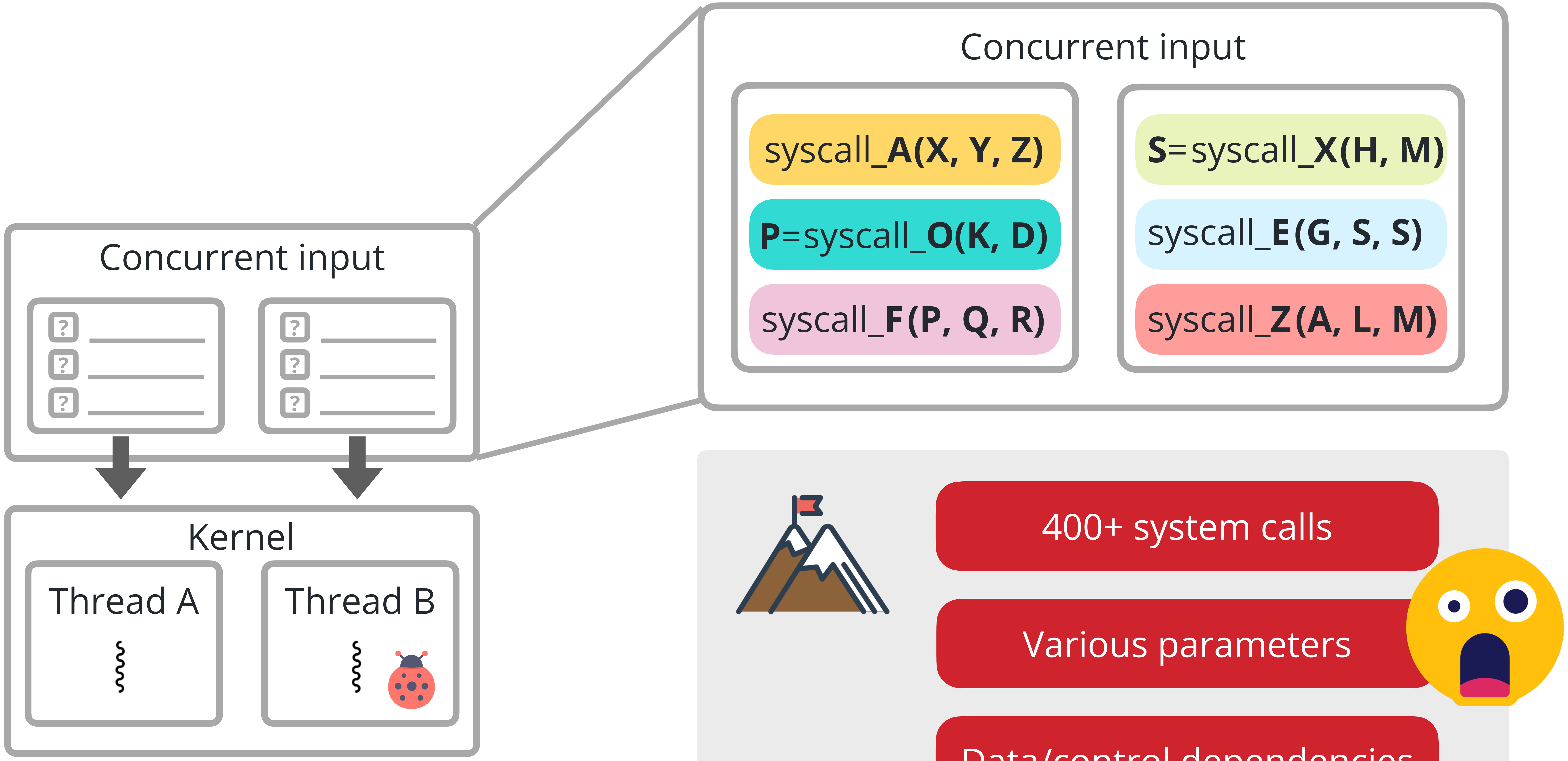
Challenge 1:
Find error-inducing concurrent inputs



Challenge 2:
Find error-inducing interleavings



Finding concurrent inputs is challenging



Concurrent input

Kernel

Thread A

Thread B

Concurrent input

syscall_A(X, Y, Z)

P=syscall_O(K, D)

syscall_F(P, Q, R)

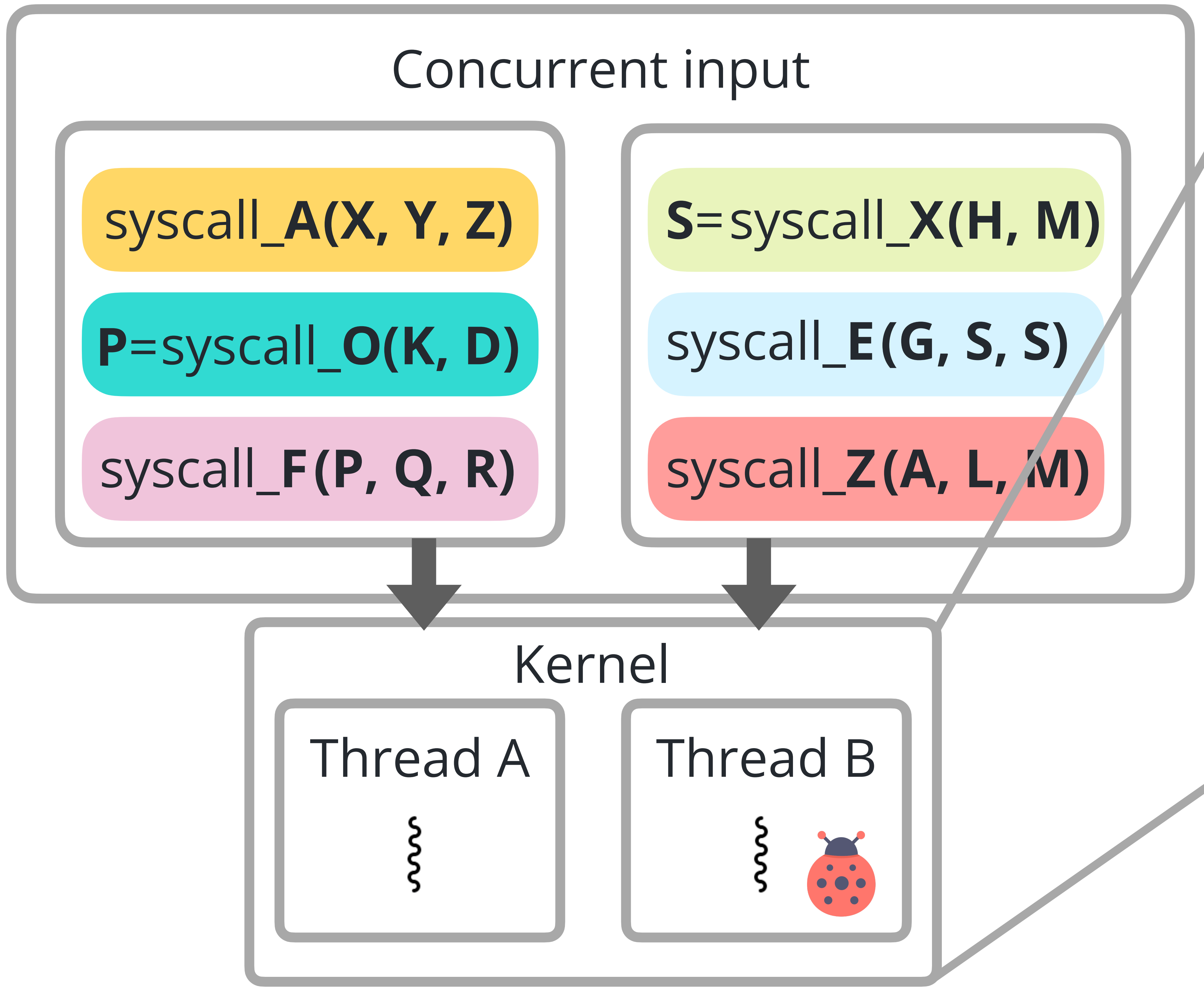
S=syscall_X(H, M)

syscall_E(G, S, S)

syscall_Z(A, L, M)

- 400+ system calls
- Various parameters
- Data/control dependencies

Finding concurrent inputs + interleavings is even more challenging



Concurrent input

syscall_**A(X, Y, Z)**

P=syscall_**O(K, D)**

syscall_**F(P, Q, R)**

S=syscall_**X(H, M)**

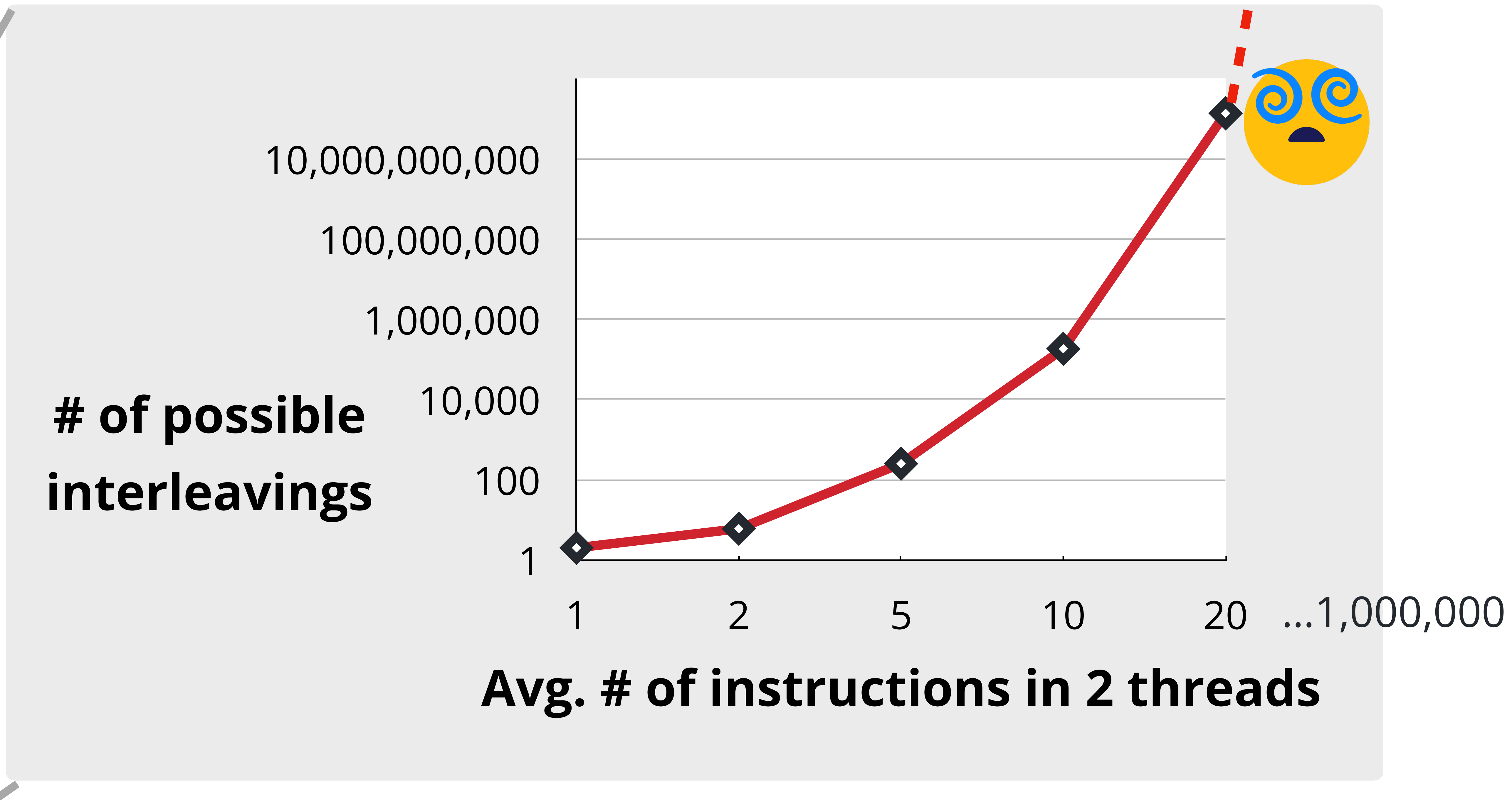
syscall_**E(G, S, S)**

syscall_**Z(A, L, M)**

Kernel

Thread A

Thread B

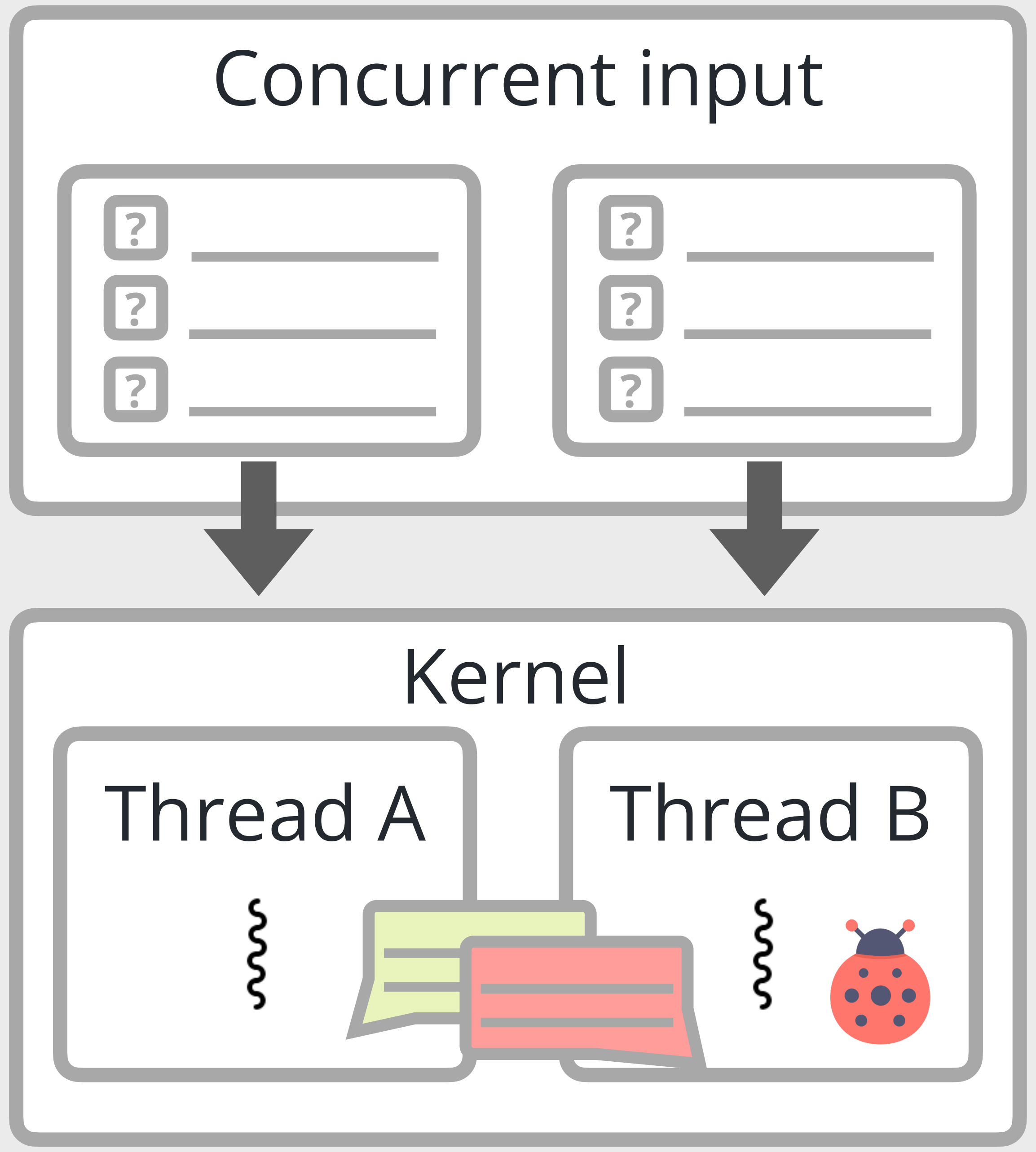


Too many possible interleavings

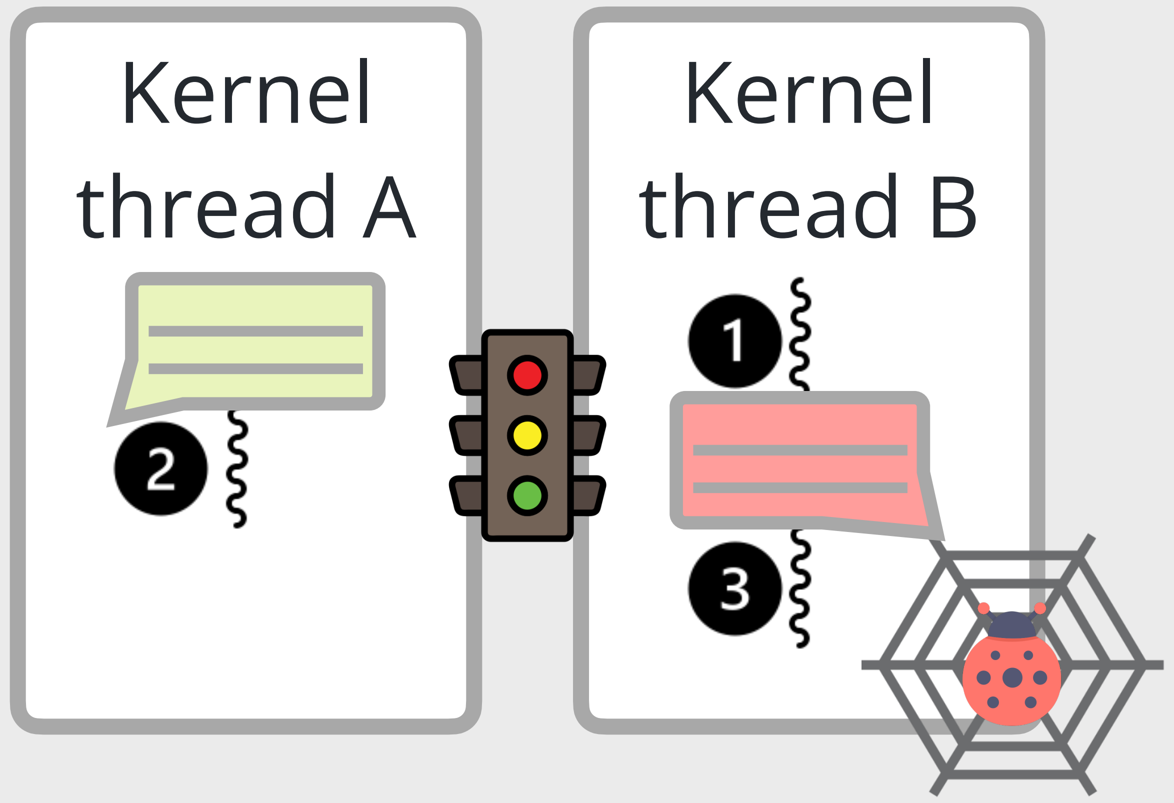
Only a few interleavings expose the bug

How does Snowboard find concurrency bugs?

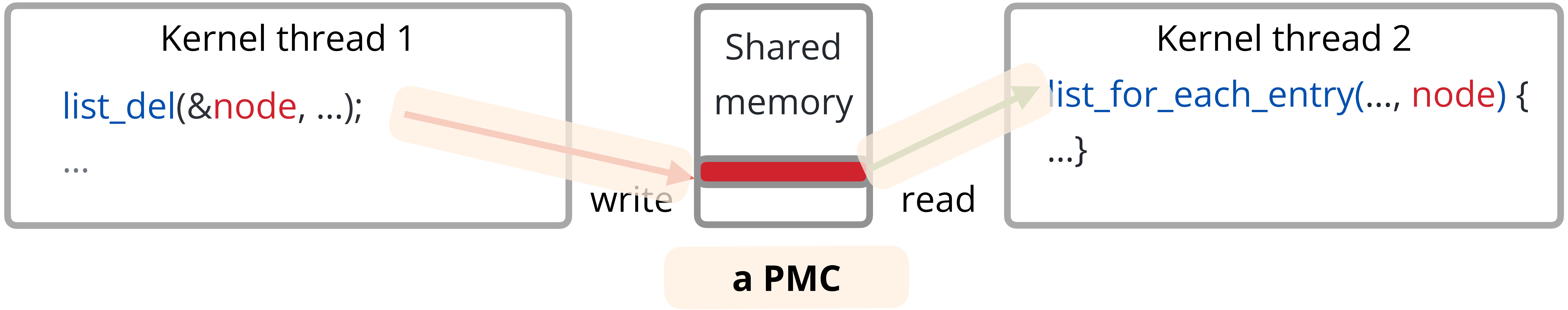
1. Predict thread interactions



2. Explore interaction interleavings



Potential memory communication



Kernel thread 1

```
list_del(&node, …);
…
```

write

Shared memory

read

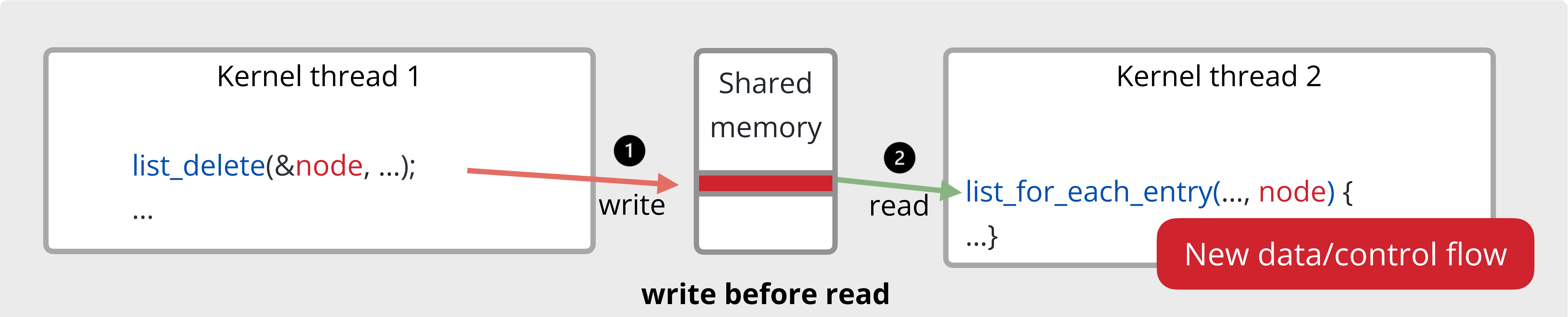
Kernel thread 2

```
list_for_each_entry(…, node) {
…}
```

a PMC

“write before read” or “write after read”?

PMC interleaving

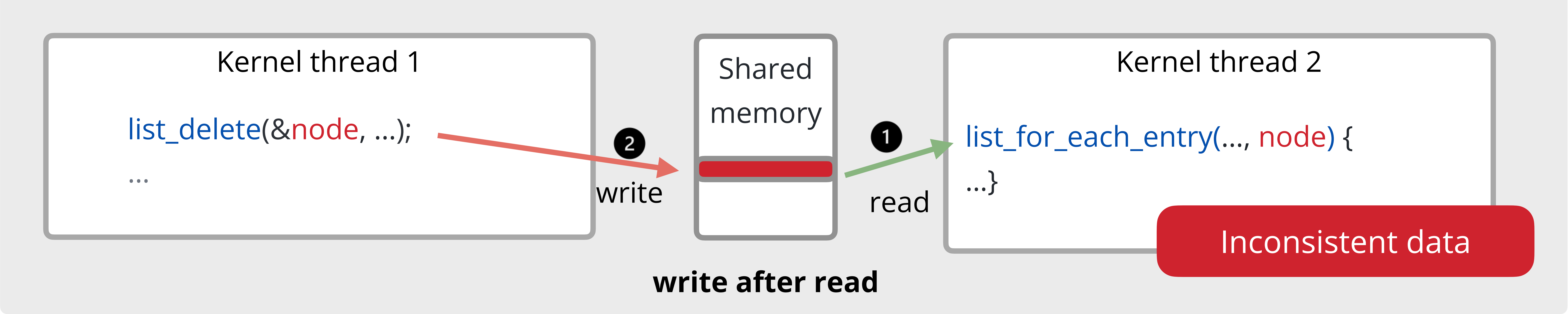


```
list_delete(&node, ...);
...
```

```
list_for_each_entry(..., node) {
...}
```

write before read

Interleavings of the PMC can lead to concurrency issues

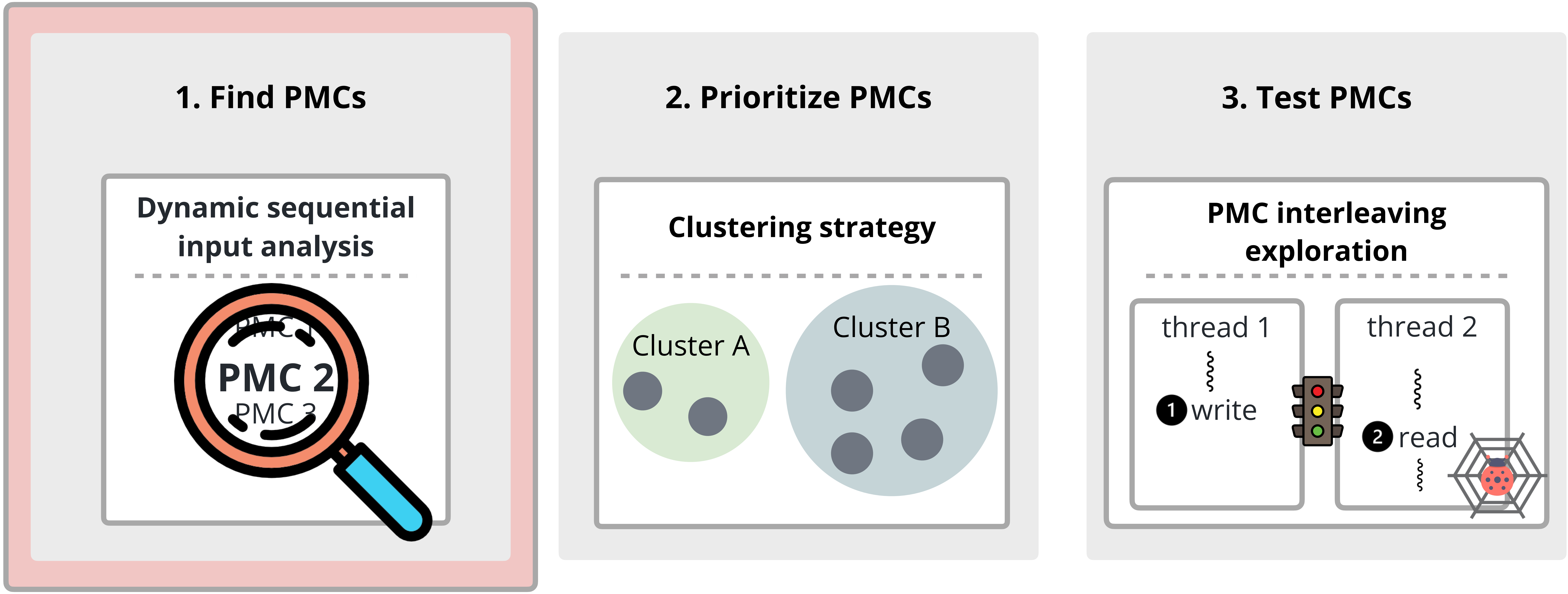


```
list_delete(&node, ...);
...
```

```
list_for_each_entry(..., node) {
...}
```

write after read

Snowboard finds concurrency bugs by testing PMCs



1. Find PMCs

Dynamic sequential input analysis

2. Prioritize PMCs

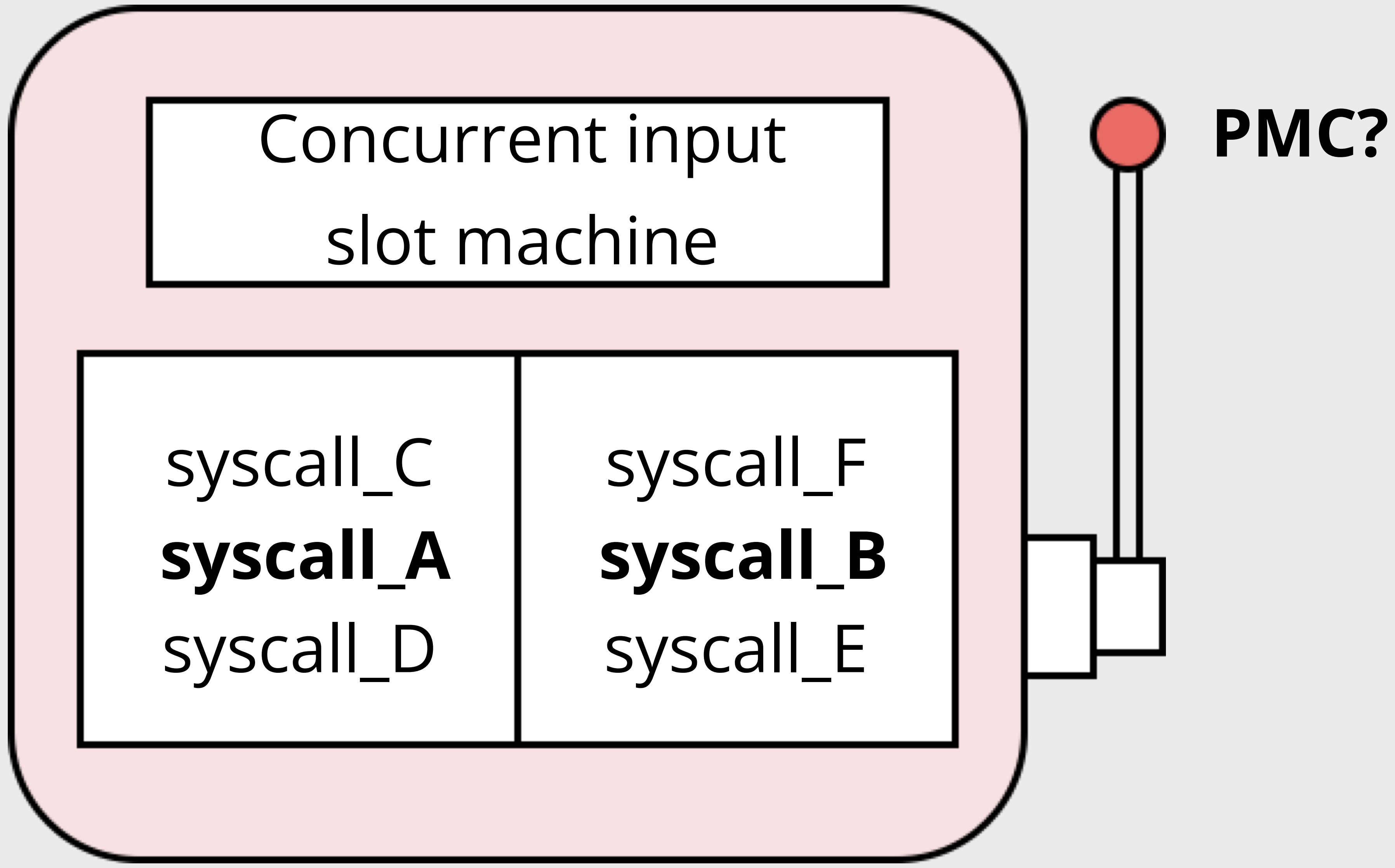
Clustering strategy

3. Test PMCs

PMC interleaving exploration

Find kernel PMCs—Possible approaches

Approach 1: Brute-force search



Concurrent input slot machine

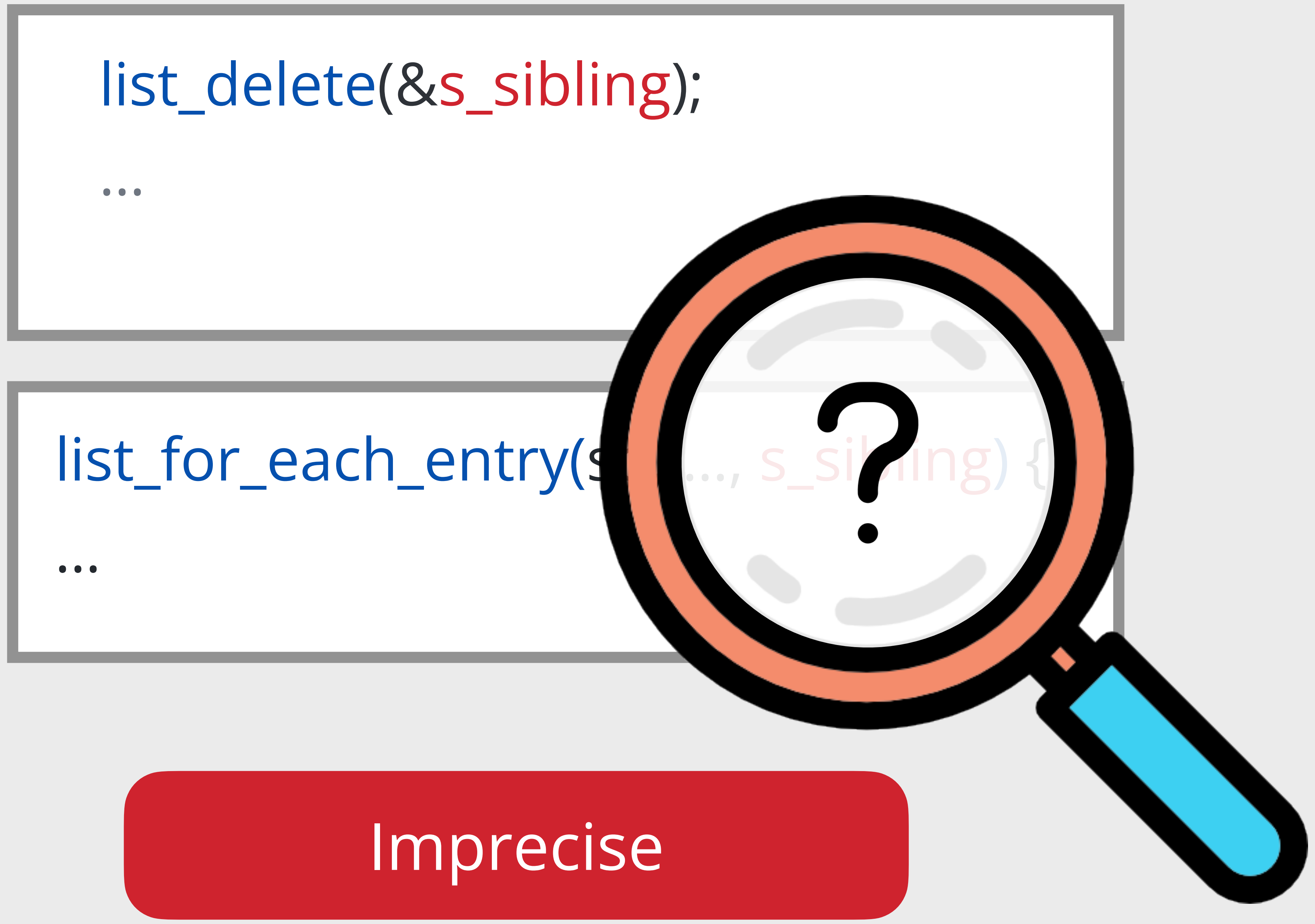
syscall_C
syscall_A
syscall_D

syscall_F
syscall_B
syscall_E

PMC?

Does not scale

Approach 2: Static analysis

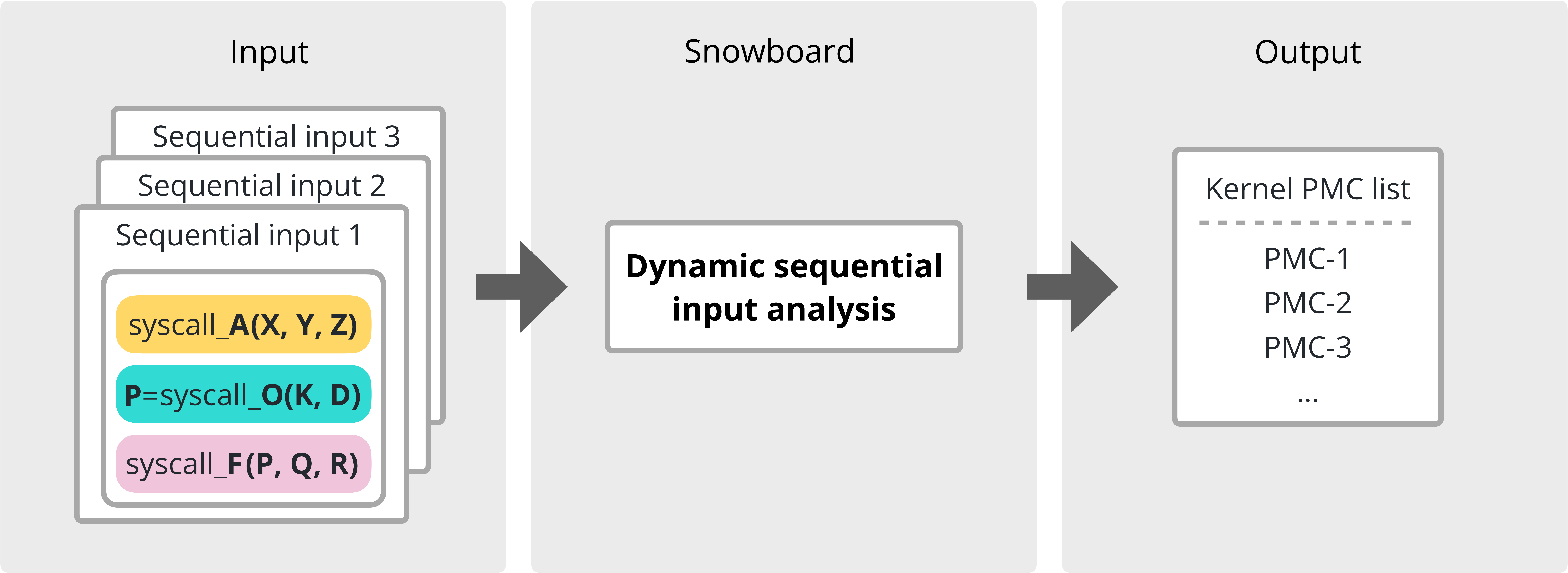


```
list_delete(&s_sibling);
...
```

```
list_for_each_entry(s..., s_sibling) {
...
```

Imprecise

Find kernel PMCs—Our approach



Input

Sequential input 3

Sequential input 2

Sequential input 1

syscall_**A(X, Y, Z)**

P=syscall_**O(K, D)**

syscall_**F(P, Q, R)**

Snowboard

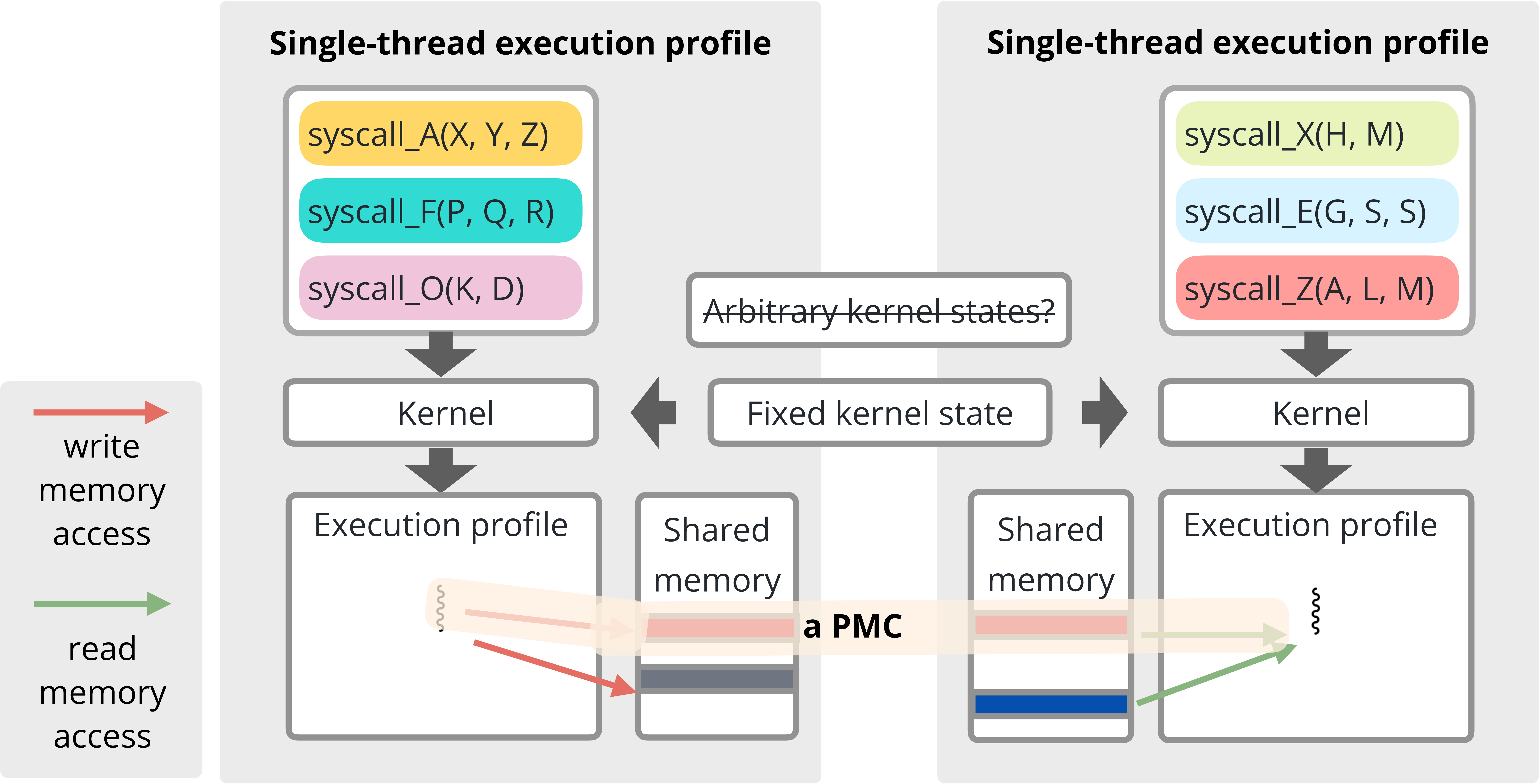
Dynamic sequential input analysis

Output

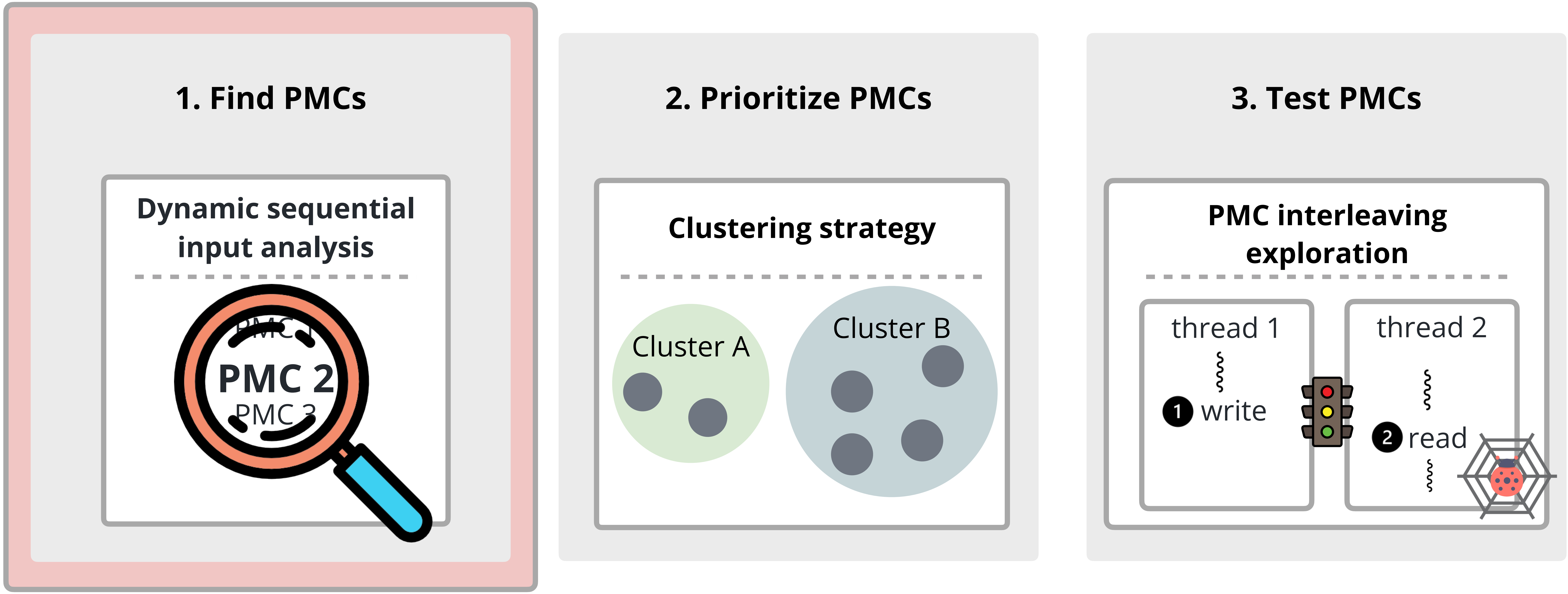
Kernel PMC list

PMC-1
PMC-2
PMC-3
...

Dynamic sequential input analysis



Snowboard finds concurrency bugs by testing PMCs



1. Find PMCs

Dynamic sequential input analysis

2. Prioritize PMCs

Clustering strategy

3. Test PMCs

PMC interleaving exploration

Prioritize PMCs

Kernel PMC list

PMC-1

PMC-2

PMC-3

...

Why do we need to prioritize PMCs?

1 Too many PMCs in the kernel

e.g., we identified 161B PMCs in Linux

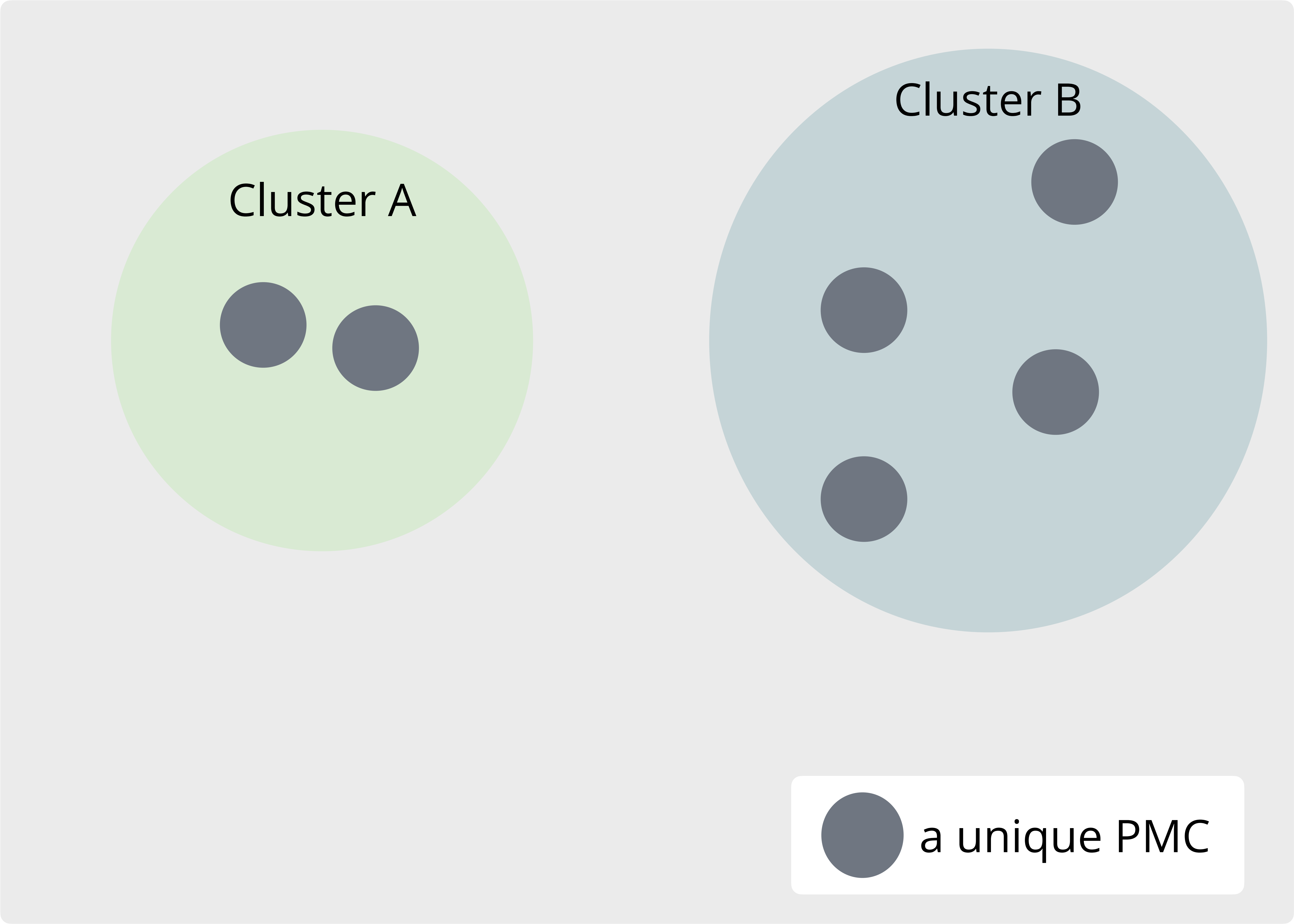
2 Testing PMCs is expensive

e.g., controlling kernel interleavings is expensive

Clustering strategy

❶ **Cluster similar PMCs**

Since testing similar execution is less rewarding



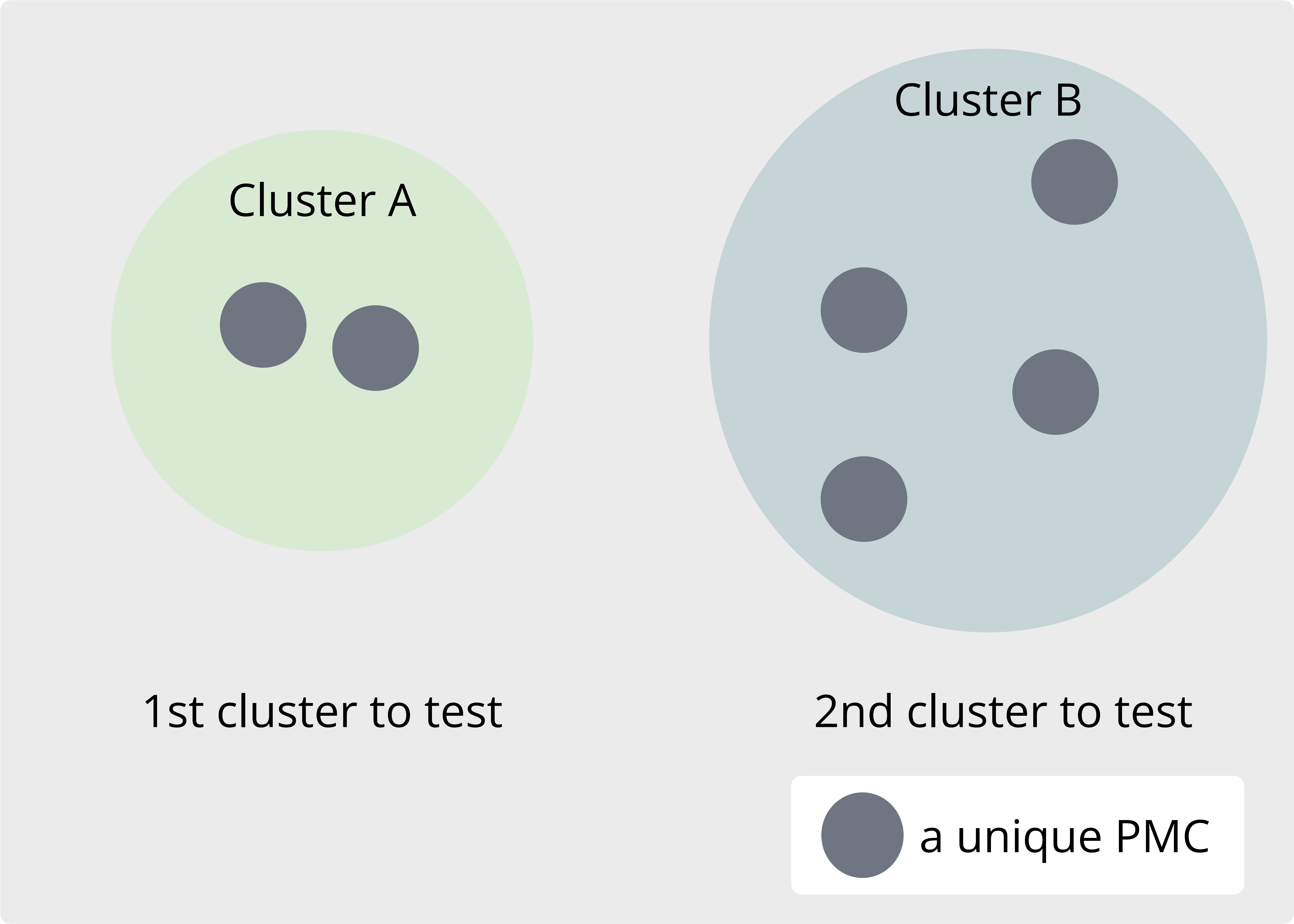
Clustering strategy

1. **Cluster similar PMCs**

   Since testing similar execution is less rewarding

2. **Prioritize small clusters**

   Since these are less likely to be tested



Clustering strategy

1. **Cluster similar PMCs**

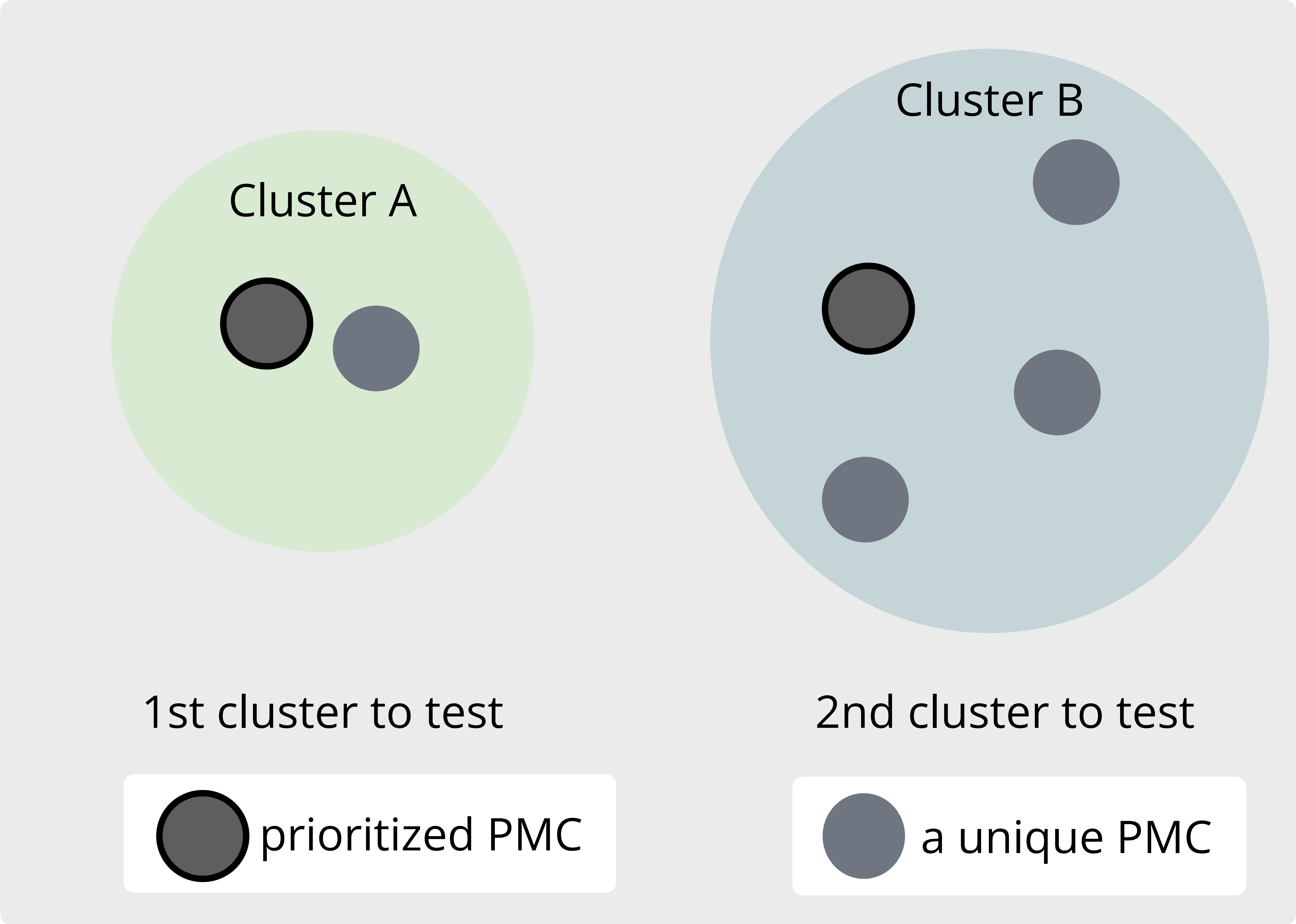
   Since testing similar execution is less rewarding

2. **Prioritize small clusters**

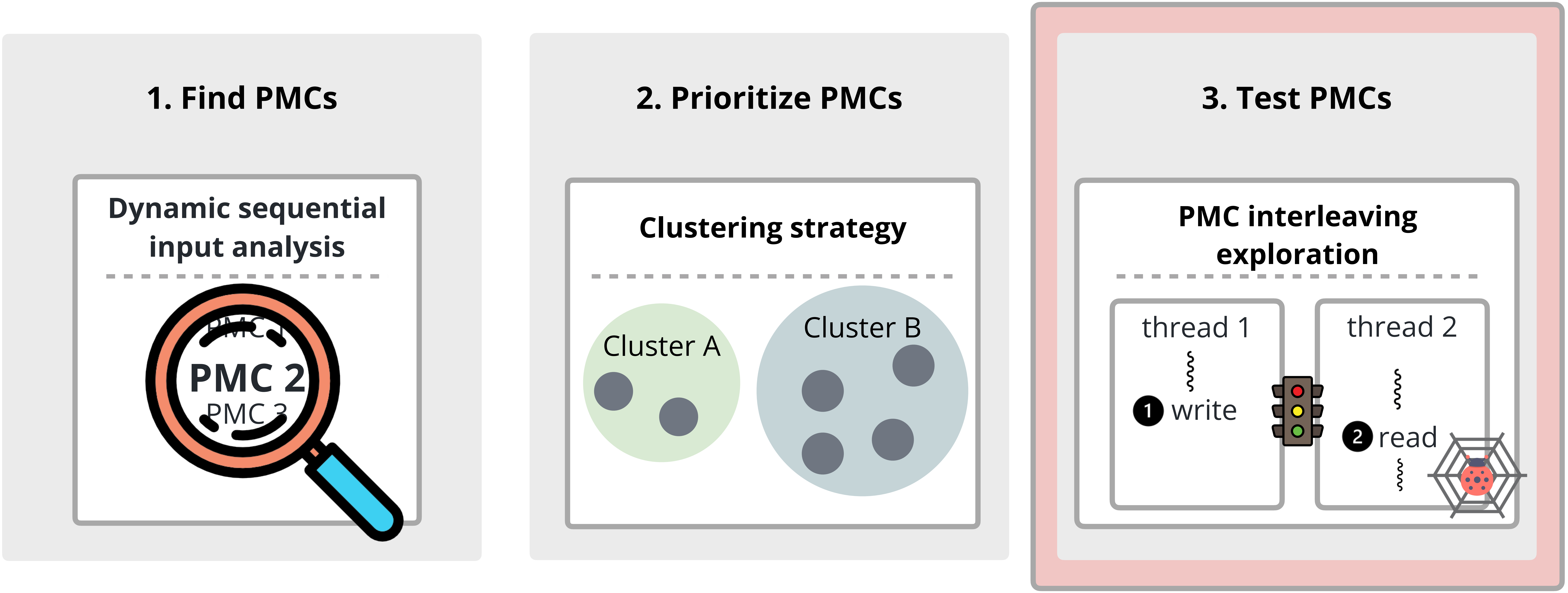
   Since these are less likely to be tested

3. **Sample a PMC from each cluster**

   Since the rest of the PMCs are similar



Snowboard finds concurrency bugs by testing PMCs



1. Find PMCs

Dynamic sequential input analysis

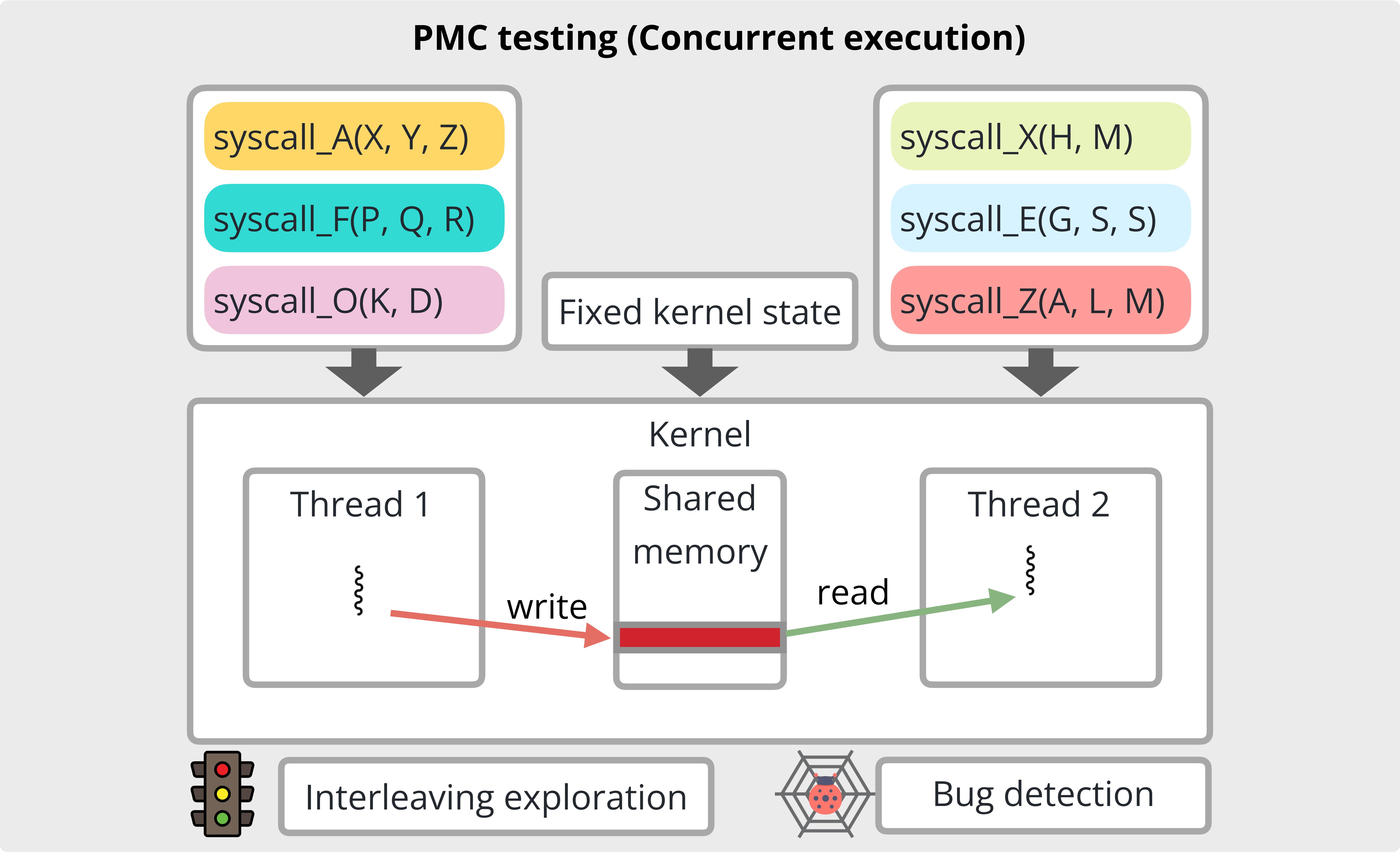
2. Prioritize PMCs

Clustering strategy

3. Test PMCs

PMC interleaving exploration

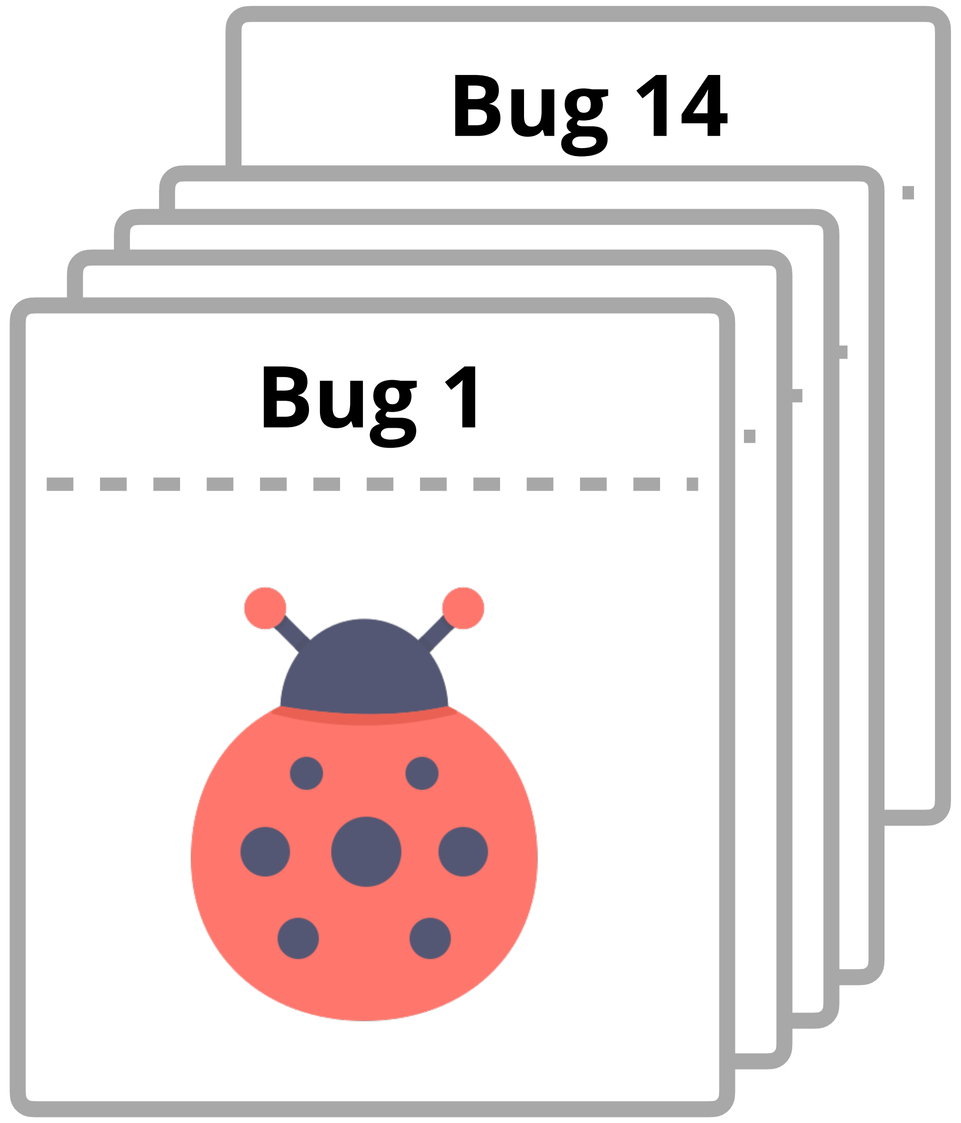
Test PMCs



PMC testing (Concurrent execution)
syscall_A(X, Y, Z)
syscall_F(P, Q, R)
syscall_O(K, D)
Fixed kernel state
syscall_X(H, M)
syscall_E(G, S, S)
syscall_Z(A, L, M)
Kernel
Thread 1
Shared memory
Thread 2
write
read
Interleaving exploration
Bug detection

Evaluation

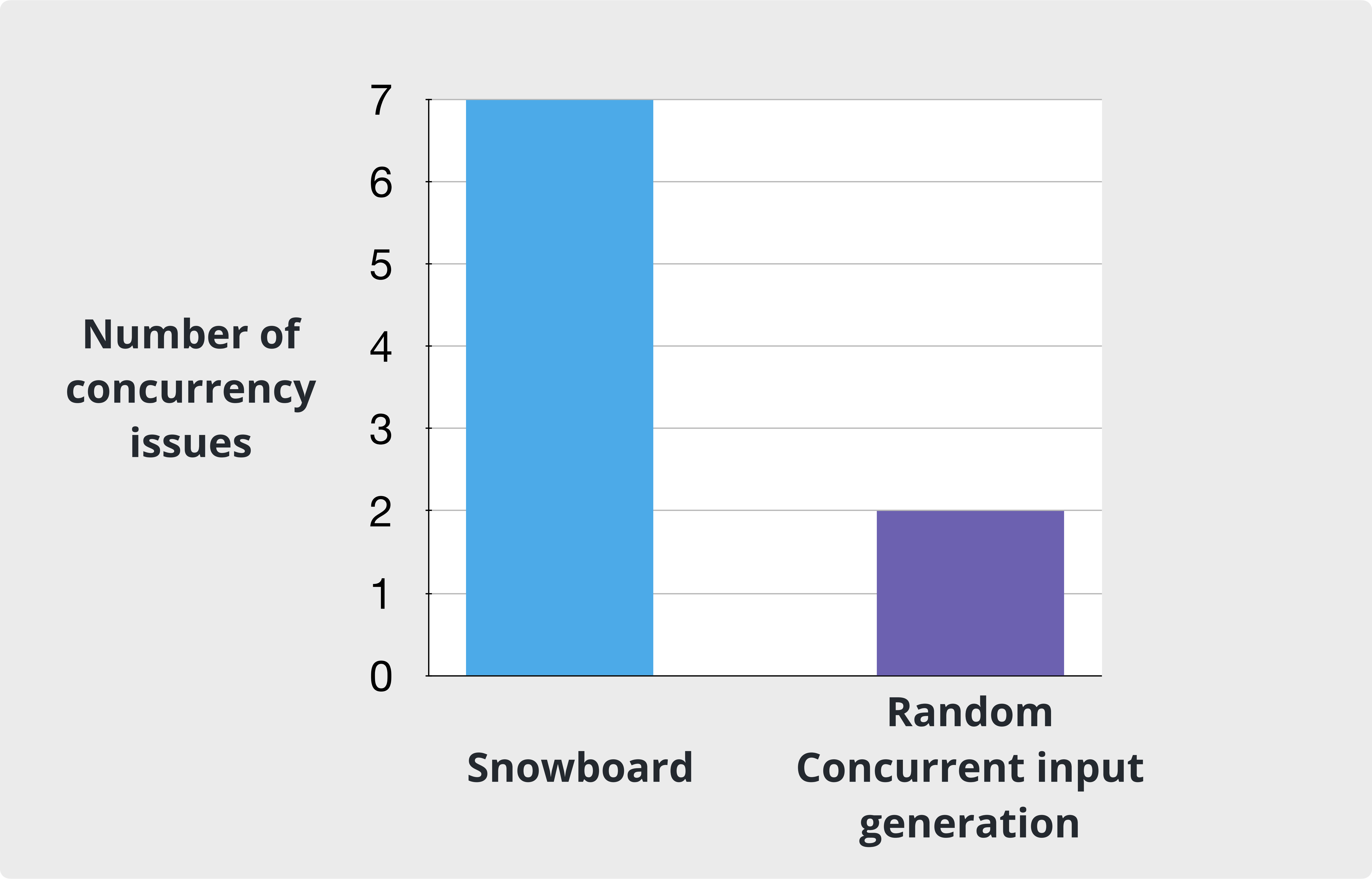
We applied Snowboard to recent Linux kernel releases



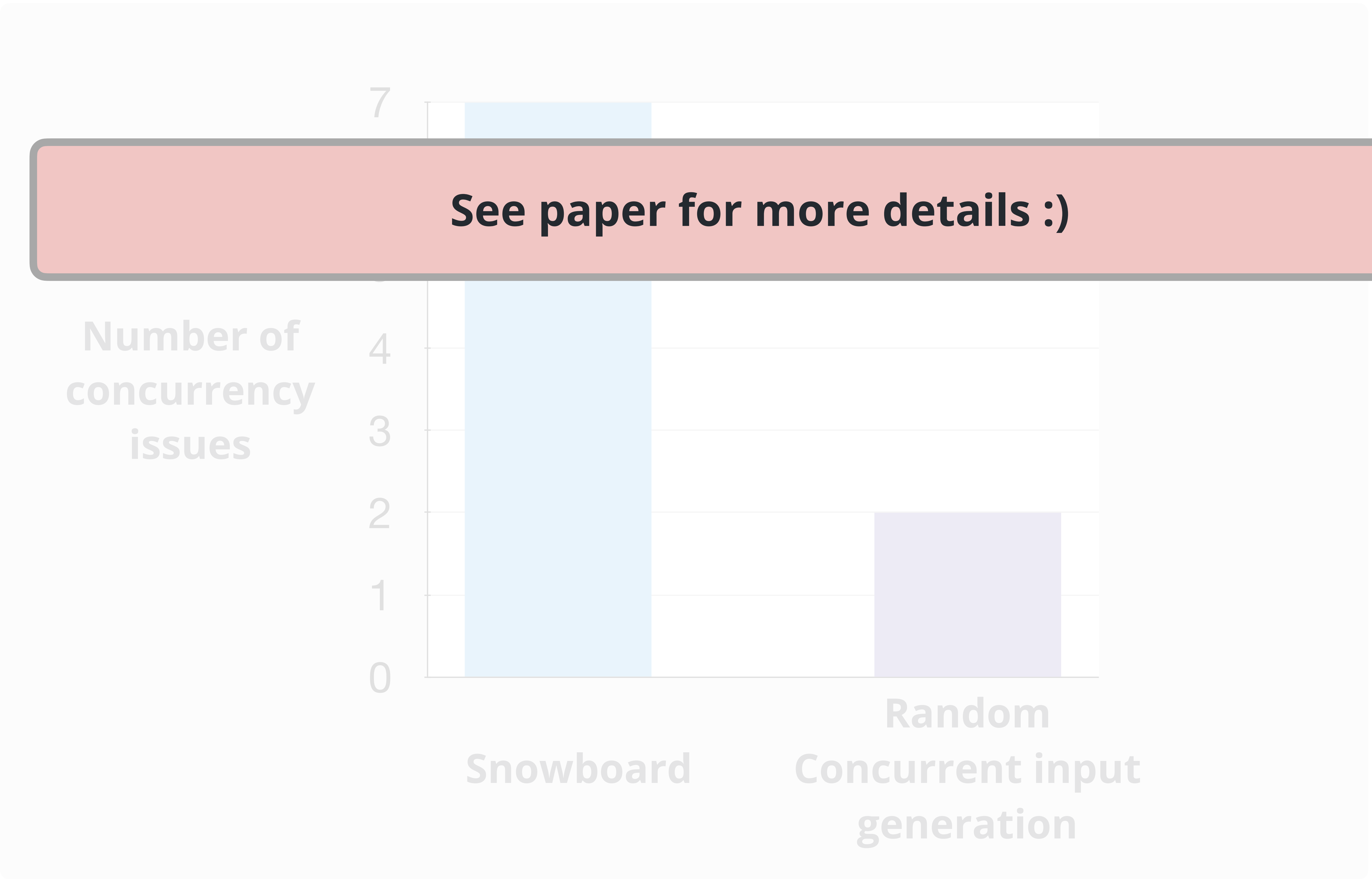
❶ Many bugs have serious impact (e.g. kernel panics, filesystem error).

❷ Some bugs existed for years.

Evaluation


Number of concurrency issues
7
6
5
4
3
2
1
0
Snowboard
Random Concurrent input generation

Evaluation



See paper for more details :)

Snowboard **Kernel concurrency bugs**

Potential memory communication (PMC)

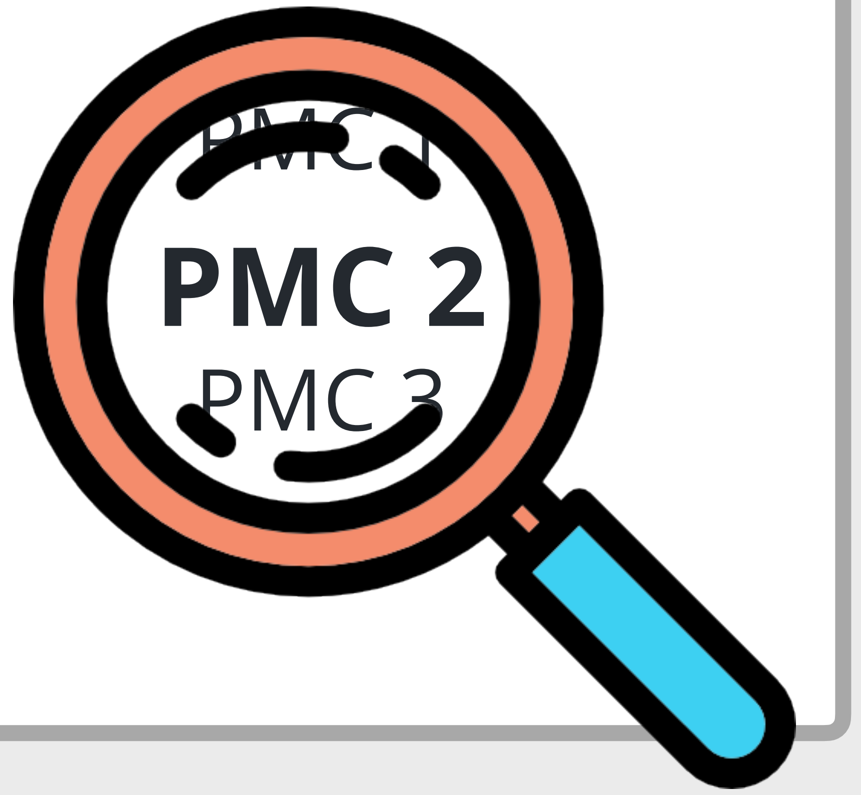
Pair of write and read accesses to shared resources

Effective in finding new concurrency bugs

https://github.com/rssys/snowboard

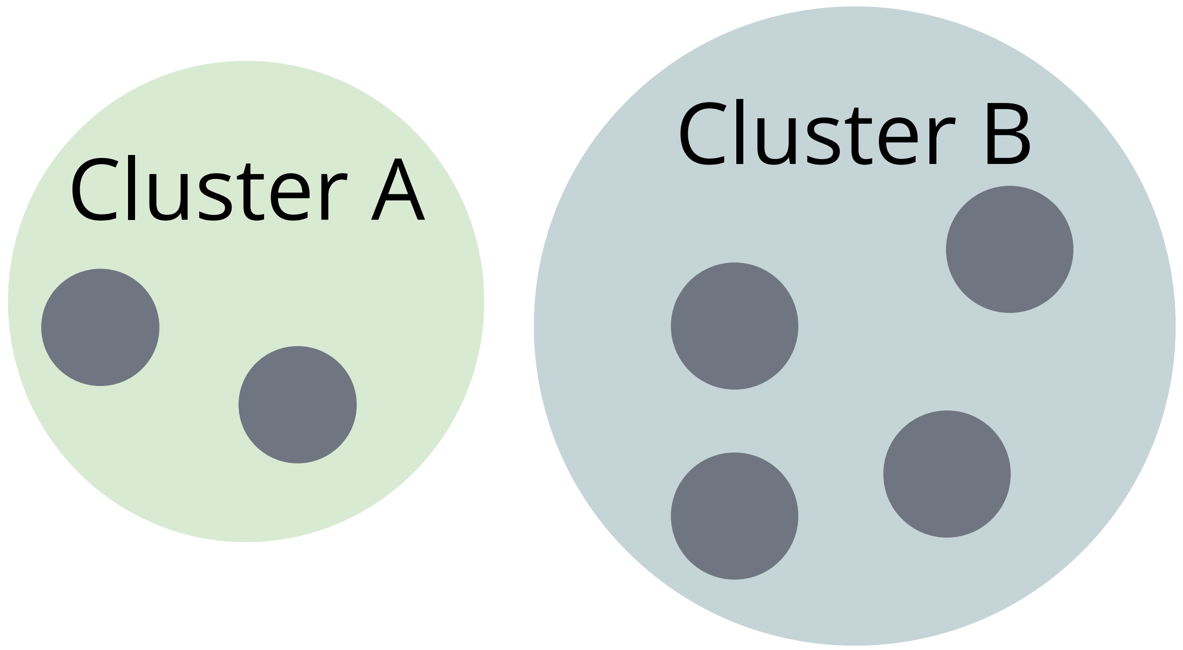
1. Find PMCs

Dynamic sequential input analysis



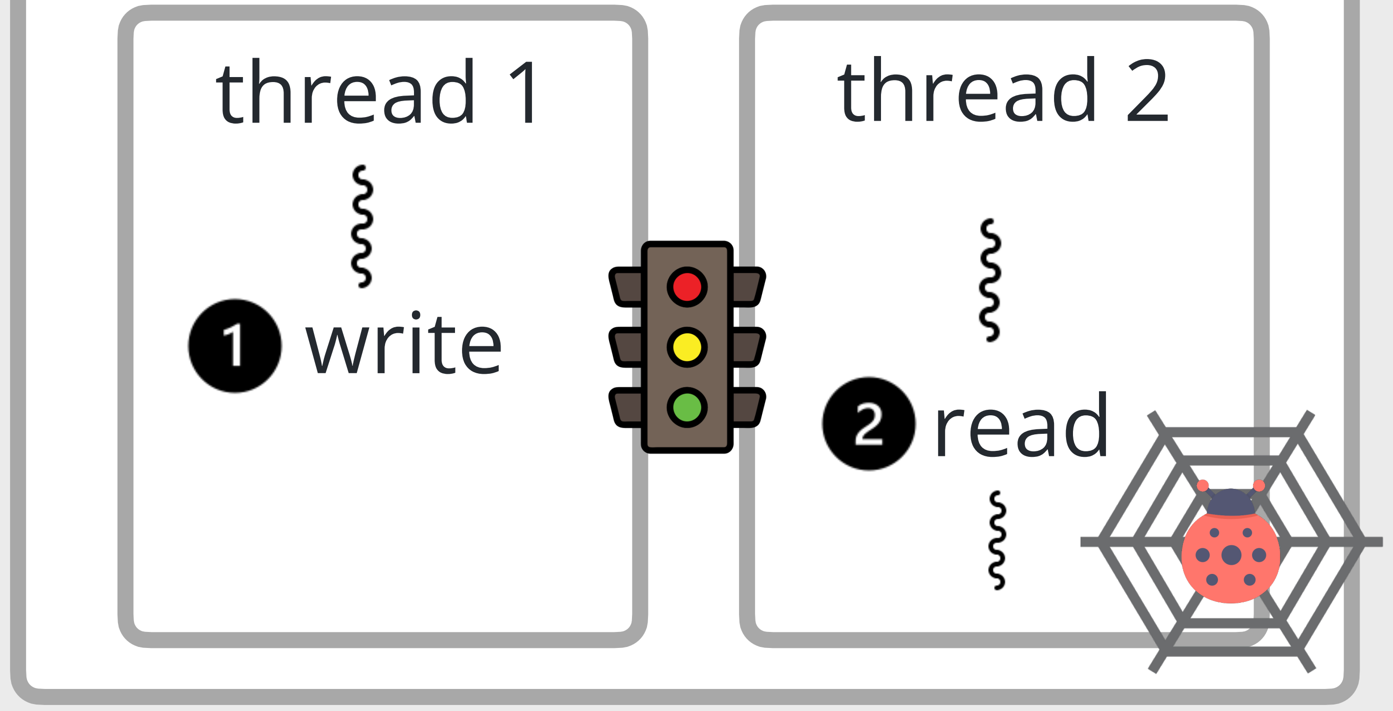
2. Prioritize PMCs

Clustering strategy



3. Test PMCs

PMC testing


thread 1
1 write
thread 2
2 read