AGAMOTTO: How Persistent is your Persistent Memory Application?

Ian Neal[1], Ben Reeves[1], Ben Stoler[1], Andi Quinn[1],
Youngjin Kwon[3], Simon Peter[2], Baris Kasikci[1]

1

The University of Texas at Austin Computer Science 2

KAIST School of Computing 3

Good News! PM has arrived!

Image Source: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

Good News! PM has arrived!

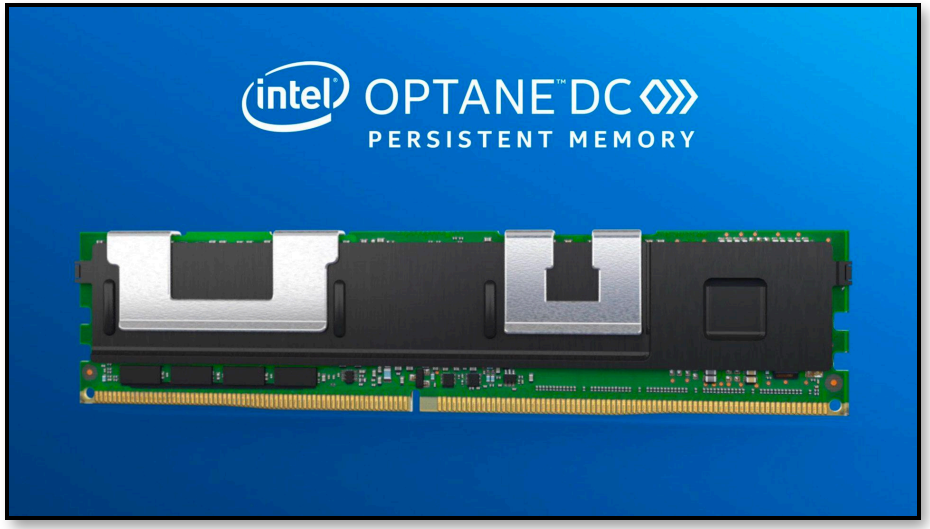


Image Source: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

Good News! PM has arrived!

- Great, fast new storage technology called persistent main memory (PM)
  - AKA non-volatile memory (NVM)

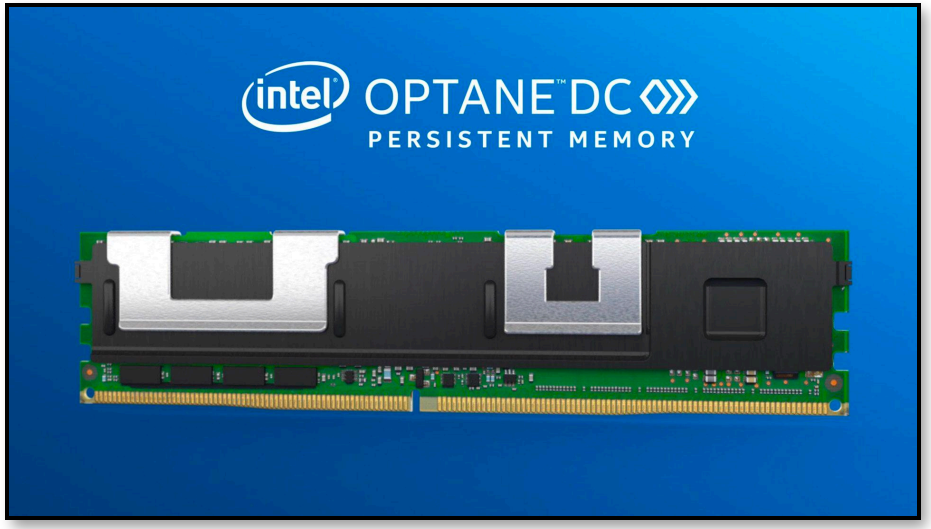


Image Source: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

Good News! PM has arrived!

- Great, fast new storage technology called persistent main memory (PM)
  - AKA non-volatile memory (NVM)
- Higher capacity, only 2−3x higher latency than DRAM

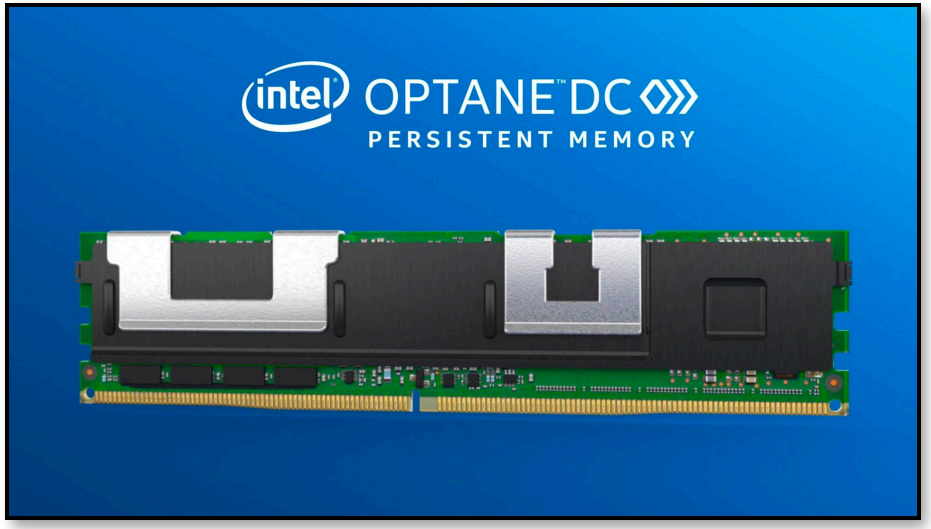


Image Source: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

Good News! PM has arrived!

- Great, fast new storage technology called persistent main memory (PM)
  - AKA non-volatile memory (NVM)
- Higher capacity, only 2−3x higher latency than DRAM
- Can be used to increase main memory capacity (volatile)
- ***Can be used as durable storage and can be used directly in user space!***

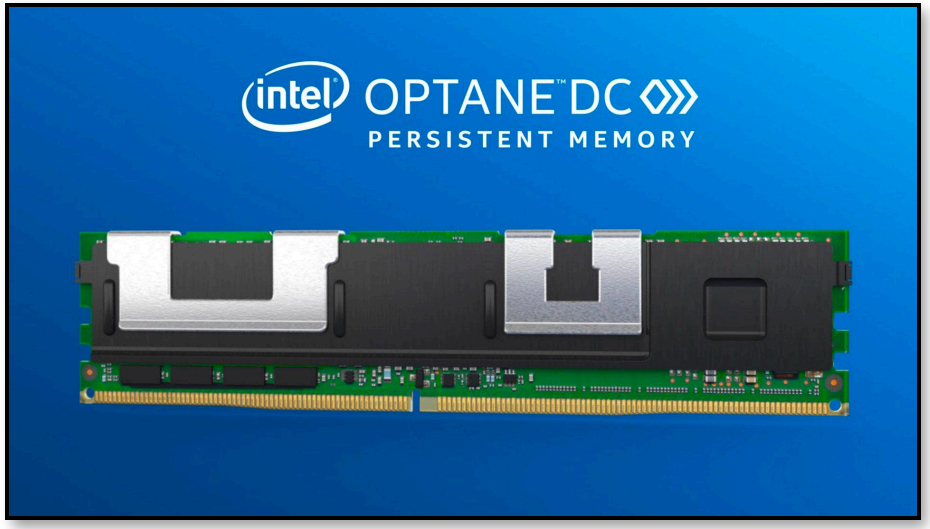
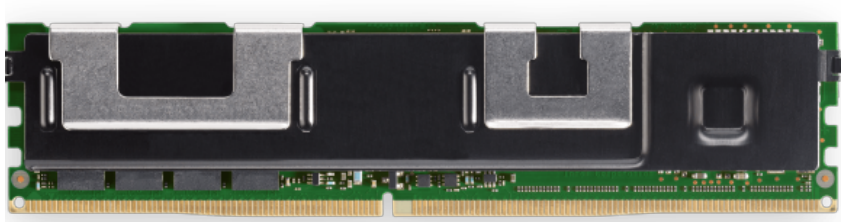


Image Source: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

PM Programming Background



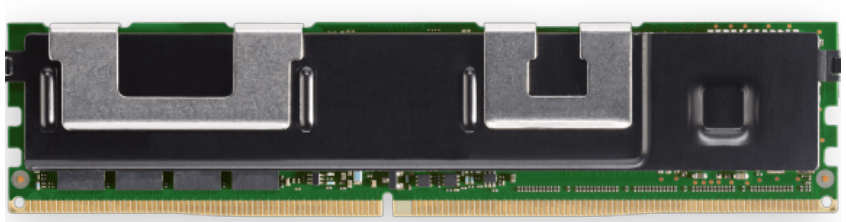
(Intel Optane PM DIMM)

CPU Cache

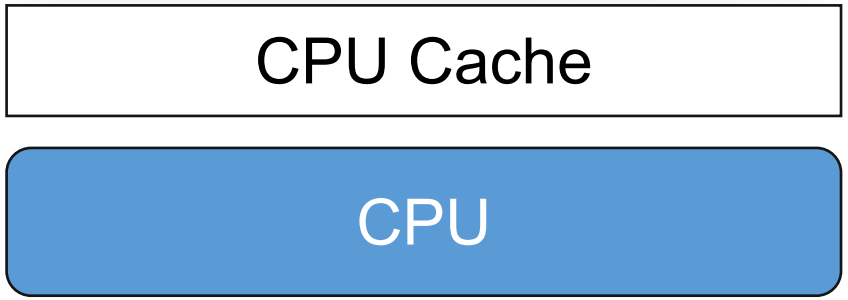
CPU

PM Programming Background

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable

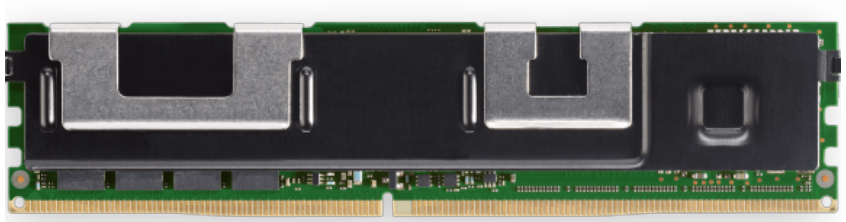


(Intel Optane PM DIMM)

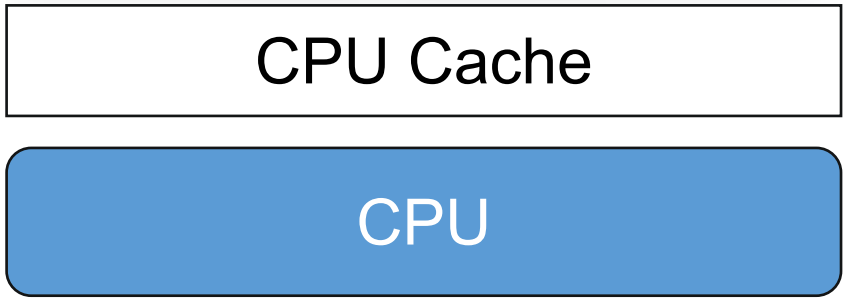


PM Programming Background

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable



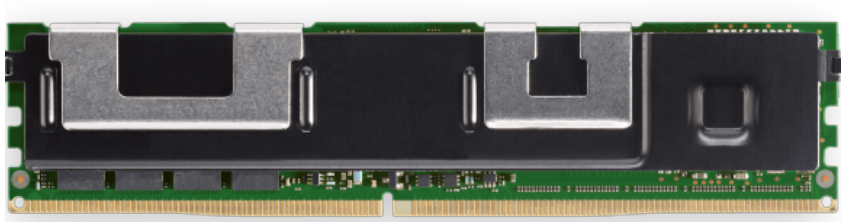
(Intel Optane PM DIMM)



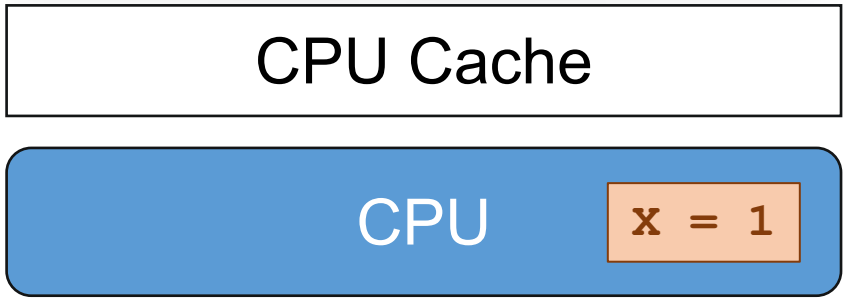

```
1. store 1 into X
```

PM Programming Background

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable



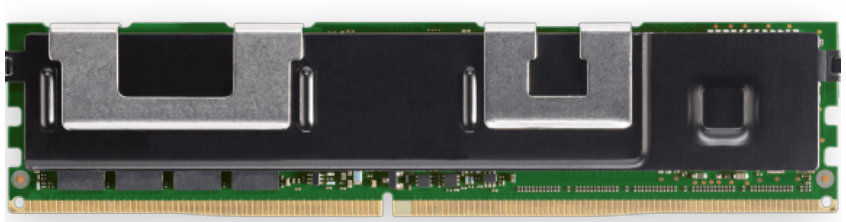
(Intel Optane PM DIMM)



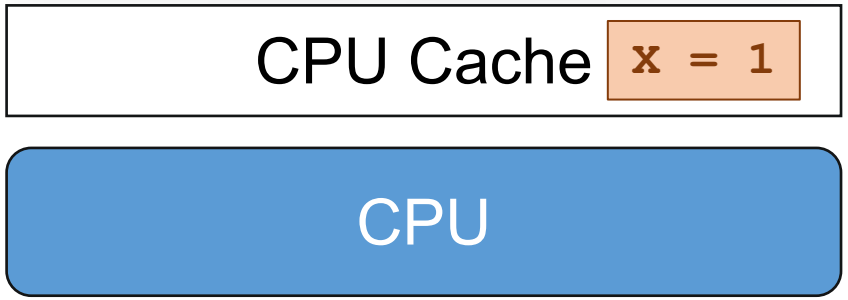

```
1. store 1 into X
```

PM Programming Background

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable



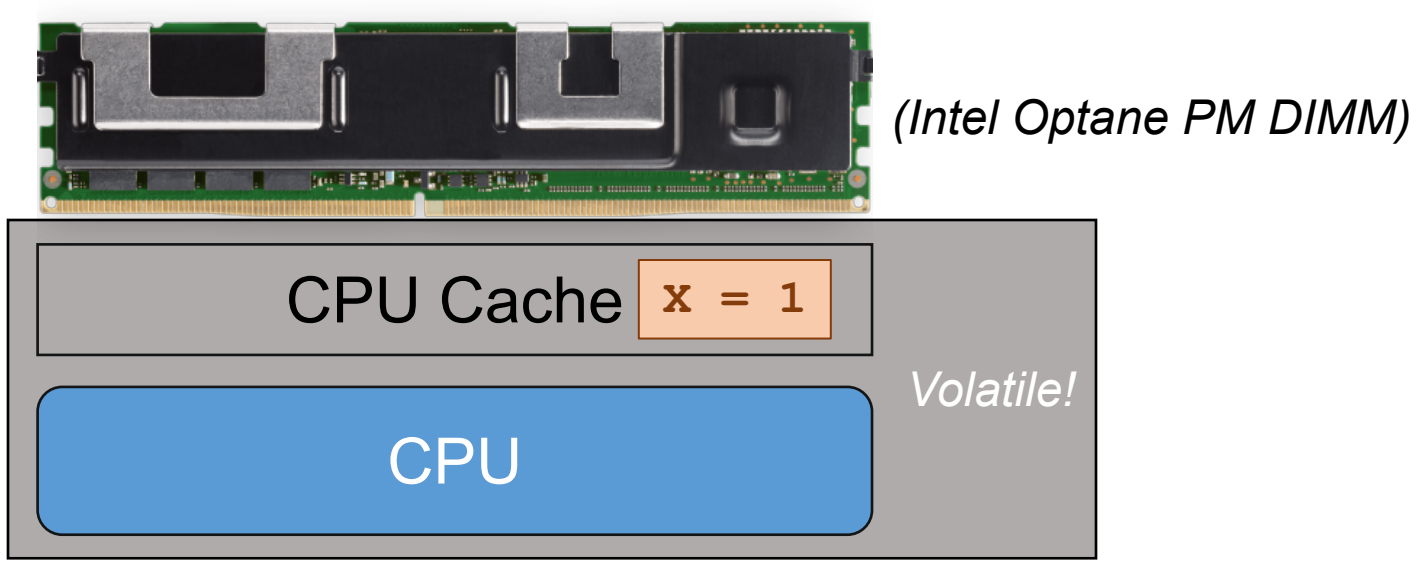
(Intel Optane PM DIMM)




```
1. store 1 into X
```

PM Programming Background

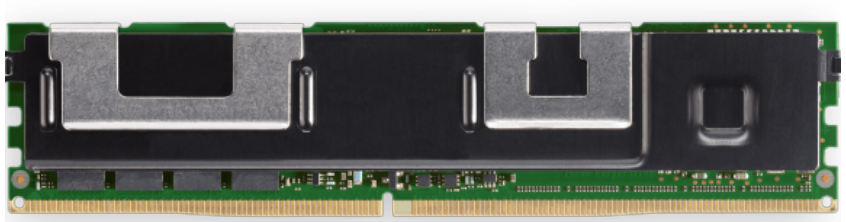
- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable



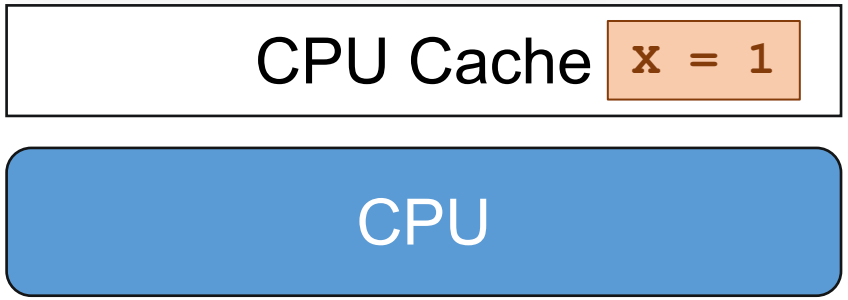

```
1. store 1 into X
```

PM Programming Background

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable



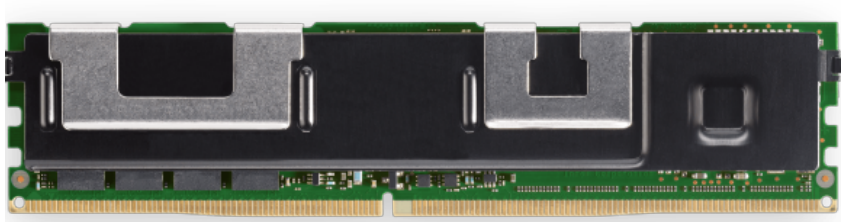
(Intel Optane PM DIMM)



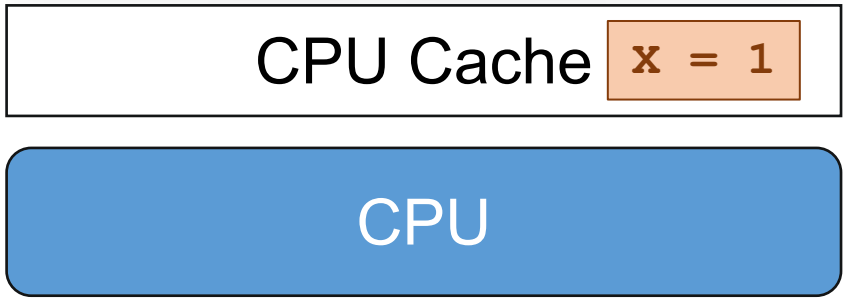

```
1. store 1 into X
```

PM Programming Background

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable



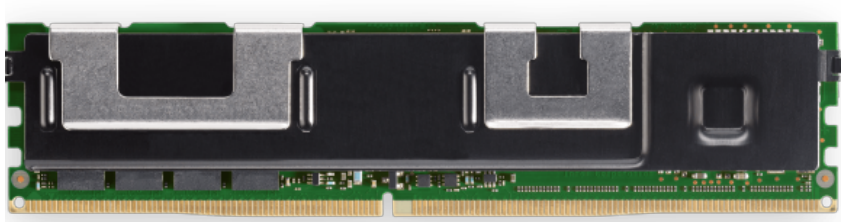
(Intel Optane PM DIMM)



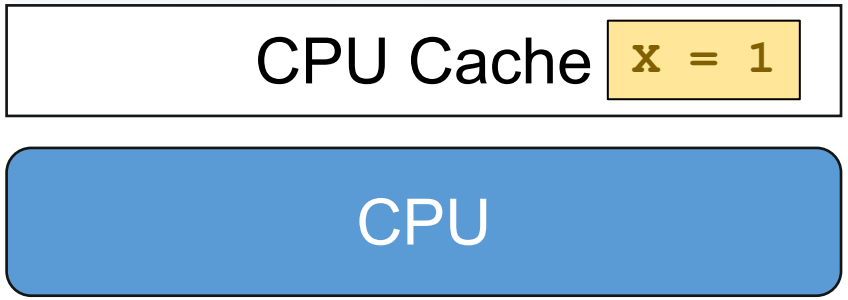

```
1. store 1 into X
2. flush
```

PM Programming Background

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable



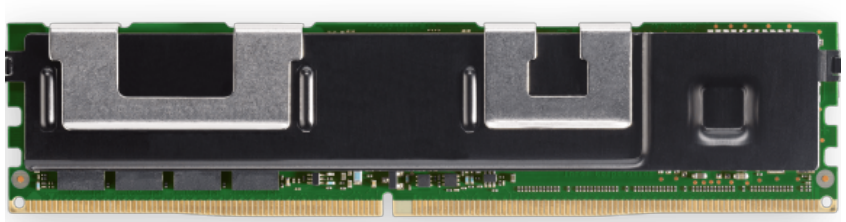
(Intel Optane PM DIMM)



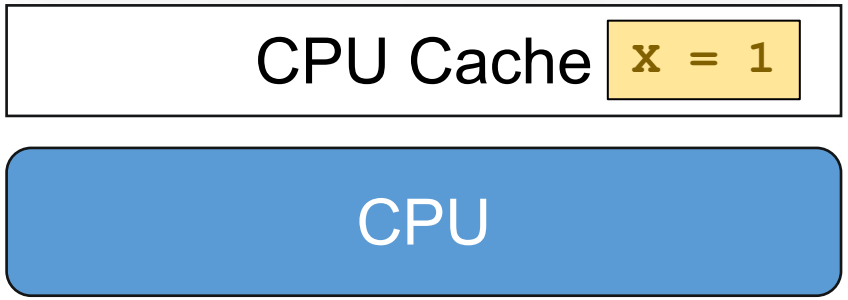

```
1. store 1 into X
2. flush
```

PM Programming Background

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable



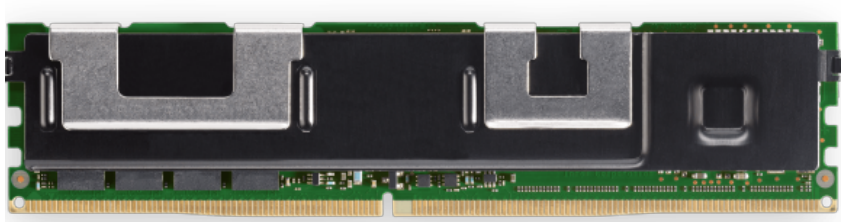
(Intel Optane PM DIMM)



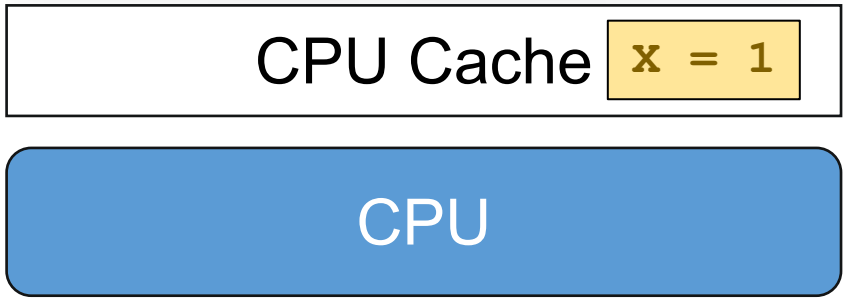

```
1. store 1 into X
2. flush ⌛
```

PM Programming Background

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable



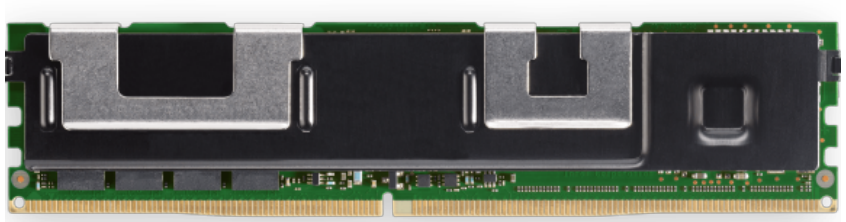
(Intel Optane PM DIMM)



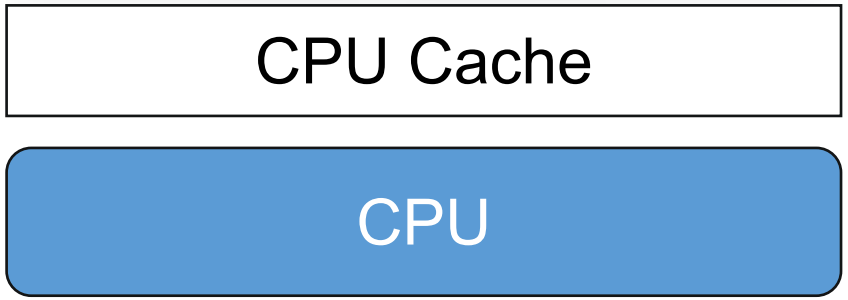

```
1. store 1 into X
2. flush ⧖
3. memory fence
```

PM Programming Background

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable



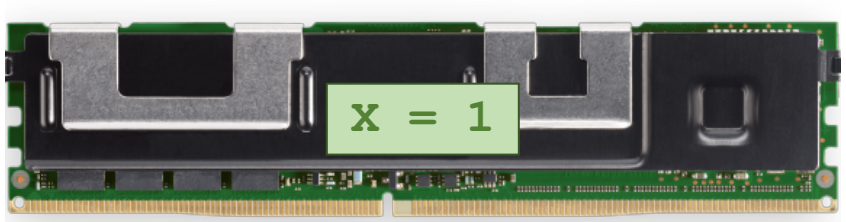
(Intel Optane PM DIMM)



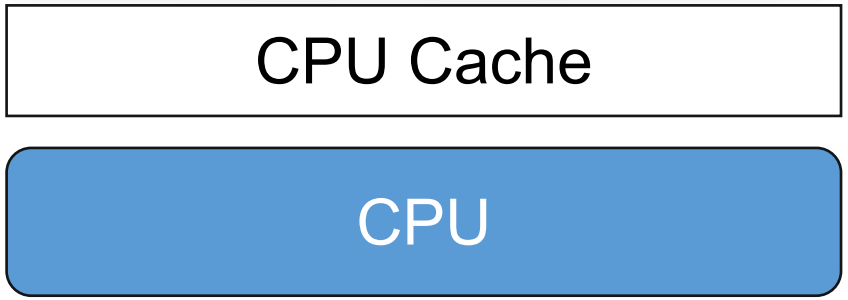

```
1. store 1 into X
2. flush
3. memory fence
```

PM Programming Background

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable



(Intel Optane PM DIMM)




```
1. store 1 into X
2. flush
3. memory fence
```

Problem: PM Programming Requires Care

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable
- Writing correct ***and*** efficient PM code is hard

```
1. store 1 into X
2. flush X
3. memory fence
```

Problem: PM Programming Requires Care

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable
- Writing correct ***and*** efficient PM code is hard

```
1. store 1 into X
2. flush X
3. memory fence
```

Problem: PM Programming Requires Care

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable
- Writing correct ***and*** efficient PM code is hard

```
1. store 1 into X
2. flush X
3. memory fence
```

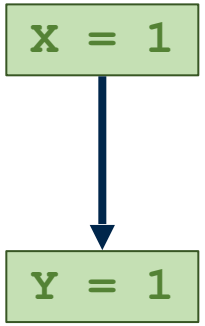
```
1. store 1 into X
2. flush X
3. memory fence
...
4. store 1 into Y
5. flush Y
6. memory fence
...
```

Problem: PM Programming Requires Care

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable
- Writing correct ***and*** efficient PM code is hard

```
1. store 1 into X
2. flush X
3. memory fence
```

```
1. store 1 into X
2. flush X
3. memory fence
...
4. store 1 into Y
5. flush Y
6. memory fence
...
```

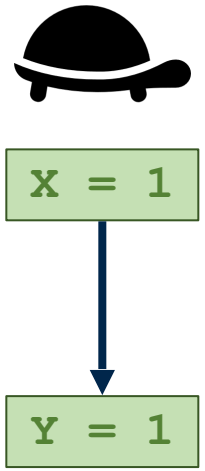


Problem: PM Programming Requires Care

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable
- Writing correct ***and*** efficient PM code is hard

```
1. store 1 into X
2. flush X
3. memory fence
```

```
1. store 1 into X
2. flush X
3. memory fence
...
4. store 1 into Y
5. flush Y
6. memory fence
...
```

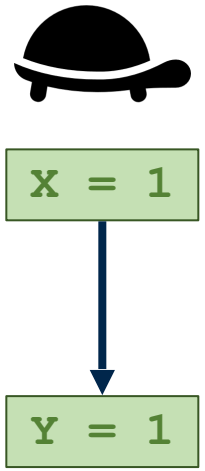


Problem: PM Programming Requires Care

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable
- Writing correct ***and*** efficient PM code is hard

```
1. store 1 into X
2. flush X
3. memory fence
```

```
1. store 1 into X
2. flush X
3. memory fence
...
4. store 1 into Y
5. flush Y
6. memory fence
...
```



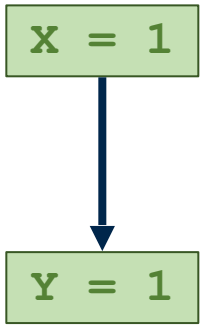

```
1. store 1 into X
2. store 1 into Y
...
3. flush X
4. flush Y
5. memory fence
...
```

Problem: PM Programming Requires Care

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable
- Writing correct ***and*** efficient PM code is hard

```
1. store 1 into X
2. flush X
3. memory fence
```

```
1. store 1 into X
2. flush X
3. memory fence
...
4. store 1 into Y
5. flush Y
6. memory fence
...
```




```
1. store 1 into X
2. store 1 into Y
...
3. flush X
4. flush Y
5. memory fence
...
```

Problem: PM Programming Requires Care

- Modern CPU architectures require special *persistence mechanisms* to ensure updates made durable
- Writing correct ***and*** efficient PM code is hard

```
1. store 1 into X
2. flush X
3. memory fence
```

```
1. store 1 into X
2. flush X
3. memory fence
...
4. store 1 into Y
5. flush Y
6. memory fence
...
```

X = 1

Y = 1

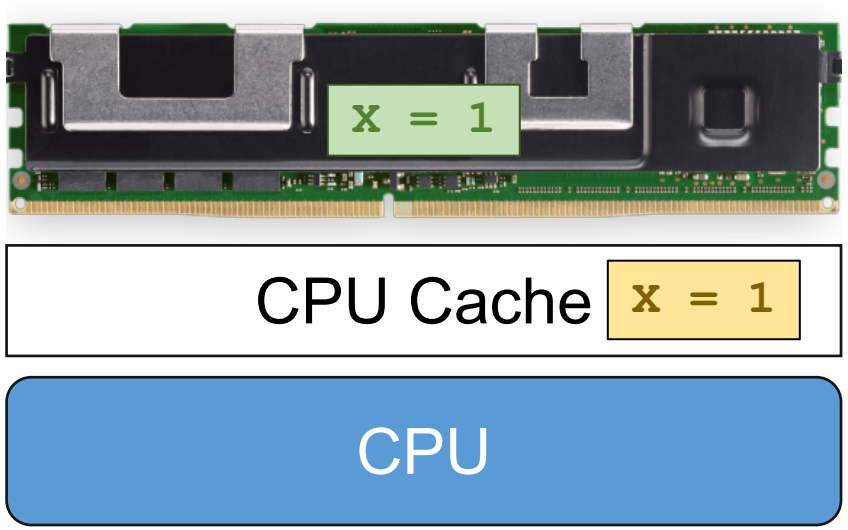
```
1. store 1 into X
2. store 1 into Y
...
3. flush X
4. flush Y
5. memory fence
...
```

X = 1

???

Y = 1

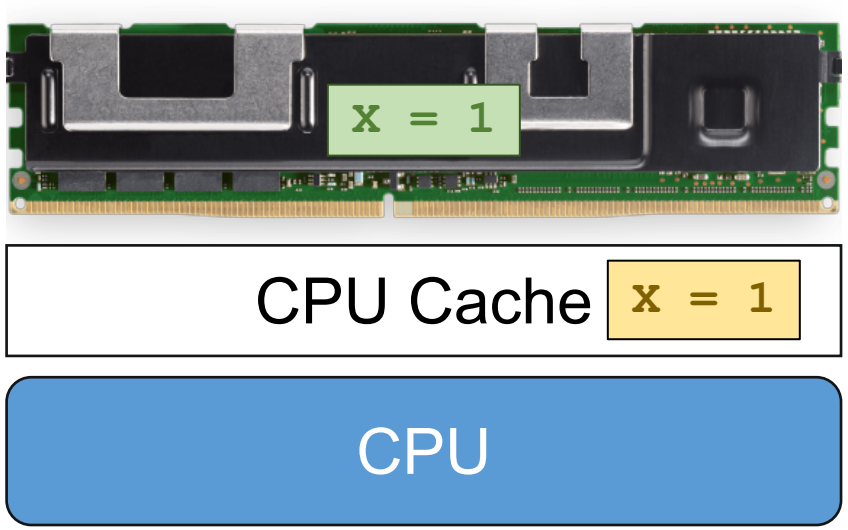
Problem: PM Programming Requires Care (cont.)



1. `store 1 into X`
2. `flush X`
3. `memory fence`

Problem: PM Programming Requires Care (cont.)

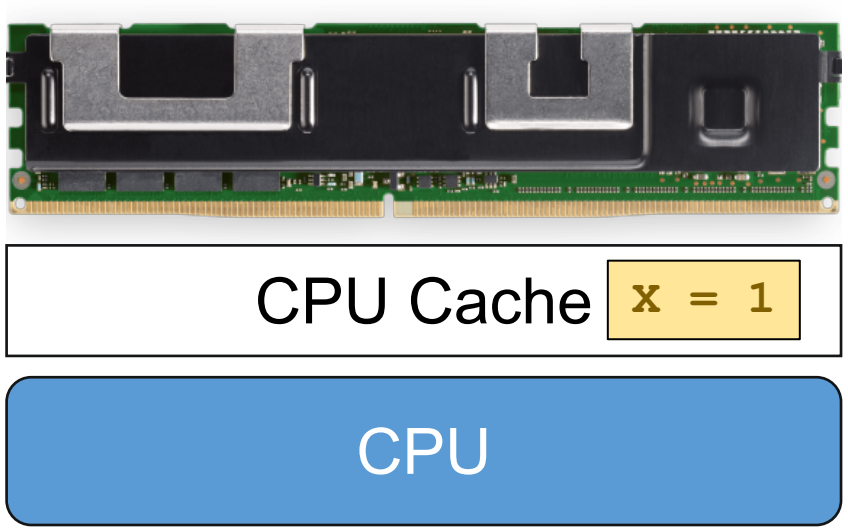
- Can have large consequences (data inconsistency, loss)




```
1. store 1 into X
2. flush X
3. memory fence
```

Problem: PM Programming Requires Care (cont.)

- Can have large consequences (data inconsistency, loss)

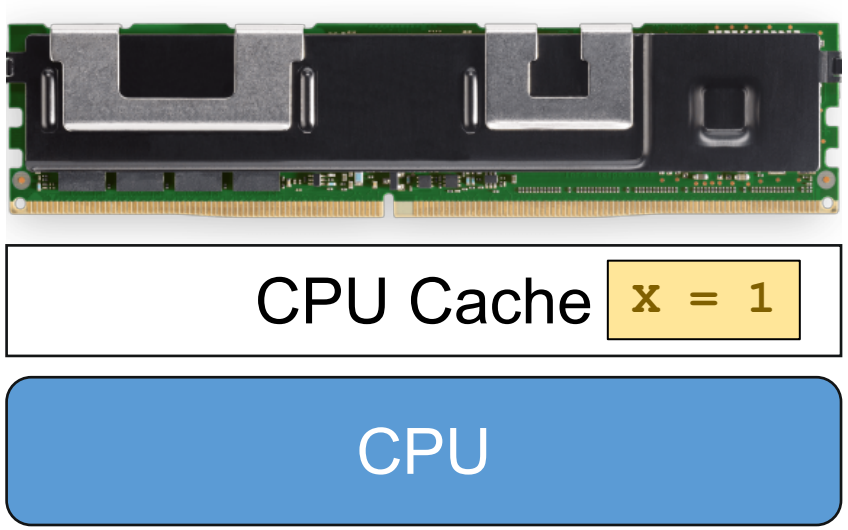



```
1. store 1 into X
```

```
3. memory fence
```

Problem: PM Programming Requires Care (cont.)

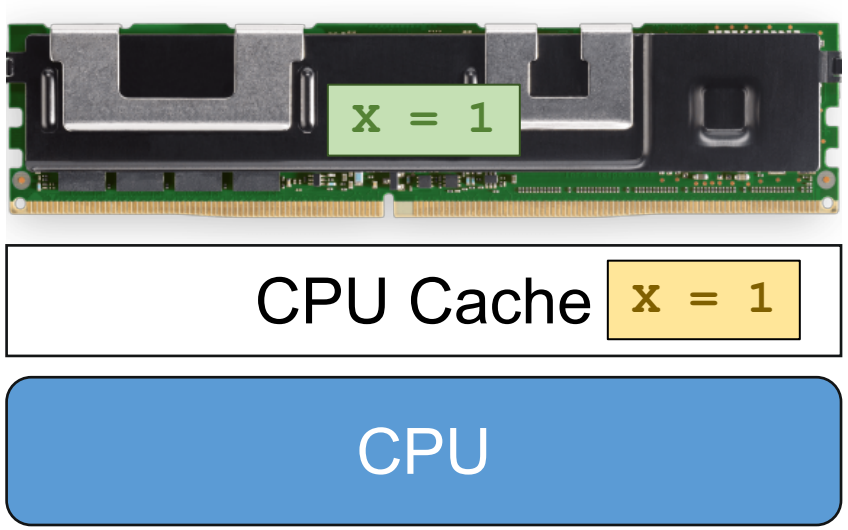
- Can have large consequences (data inconsistency, loss)




```
1. store 1 into X
2. flush X
```

Problem: PM Programming Requires Care (cont.)

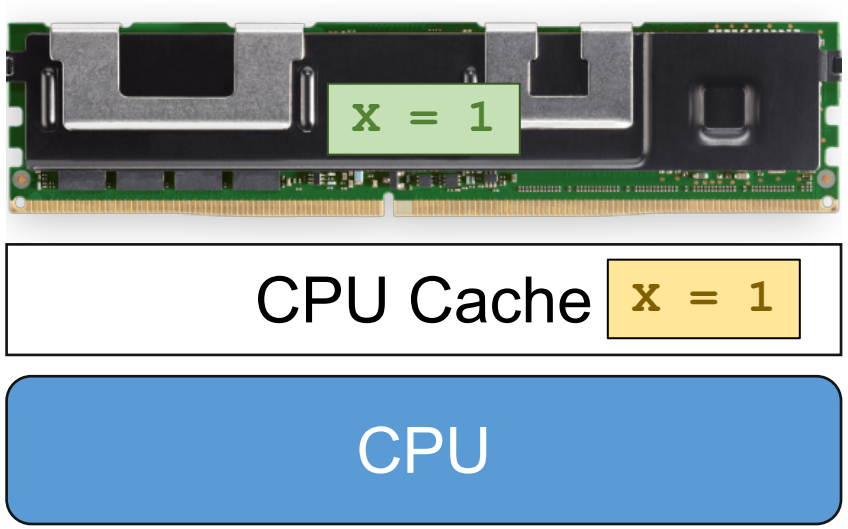
- Can have large consequences (data inconsistency, loss)




```
1. store 1 into X
2. flush X
3. memory fence
```

Problem: PM Programming Requires Care (cont.)

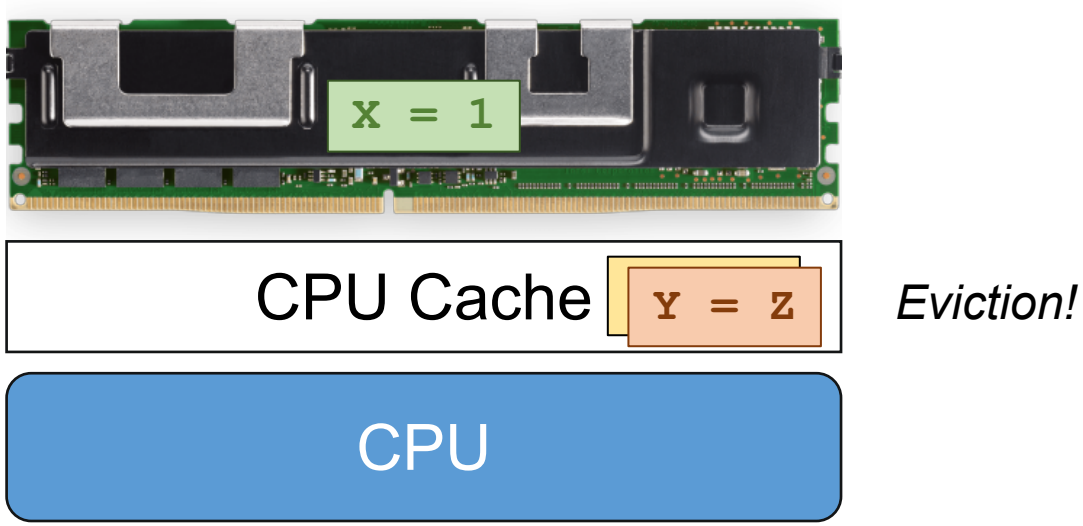
- Can have large consequences (data inconsistency, loss)
- Resulting bugs are hard to find (opaque, non-determinism)




```
1. store 1 into X
2. flush X
3. memory fence
```

Problem: PM Programming Requires Care (cont.)

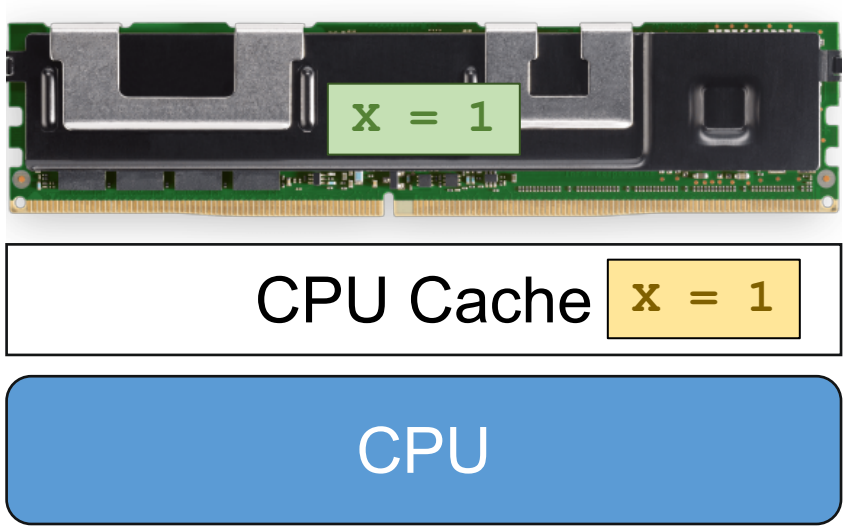
- Can have large consequences (data inconsistency, loss)
- Resulting bugs are hard to find (opaque, non-determinism)




```
1. store 1 into X
2. flush X
3. memory fence
```

Problem: PM Programming Requires Care (cont.)

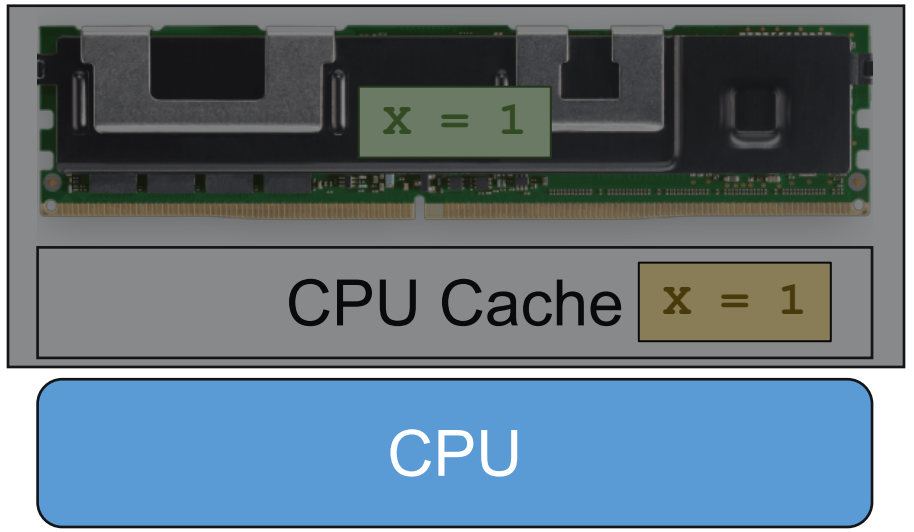
- Can have large consequences (data inconsistency, loss)
- Resulting bugs are hard to find (opaque, non-determinism)




```
1. store 1 into X
2. flush X
3. memory fence
```

Problem: PM Programming Requires Care (cont.)

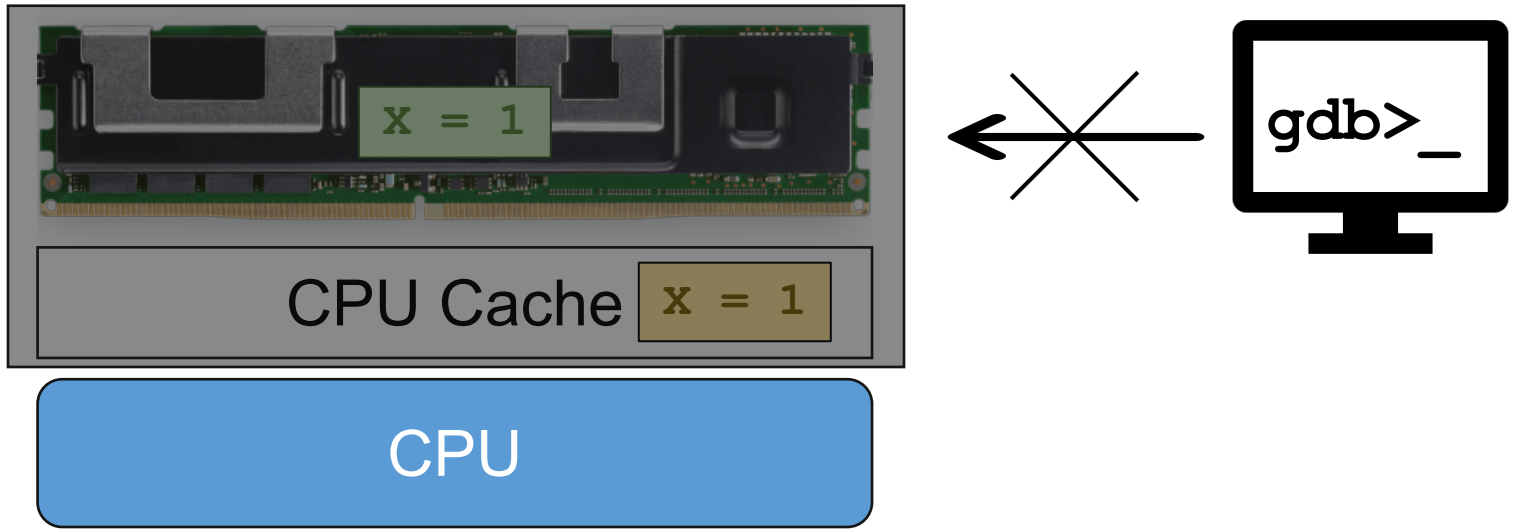
- Can have large consequences (data inconsistency, loss)
- Resulting bugs are hard to find (opaque, non-determinism)




```
1. store 1 into X
2. flush X
3. memory fence
```

Problem: PM Programming Requires Care (cont.)

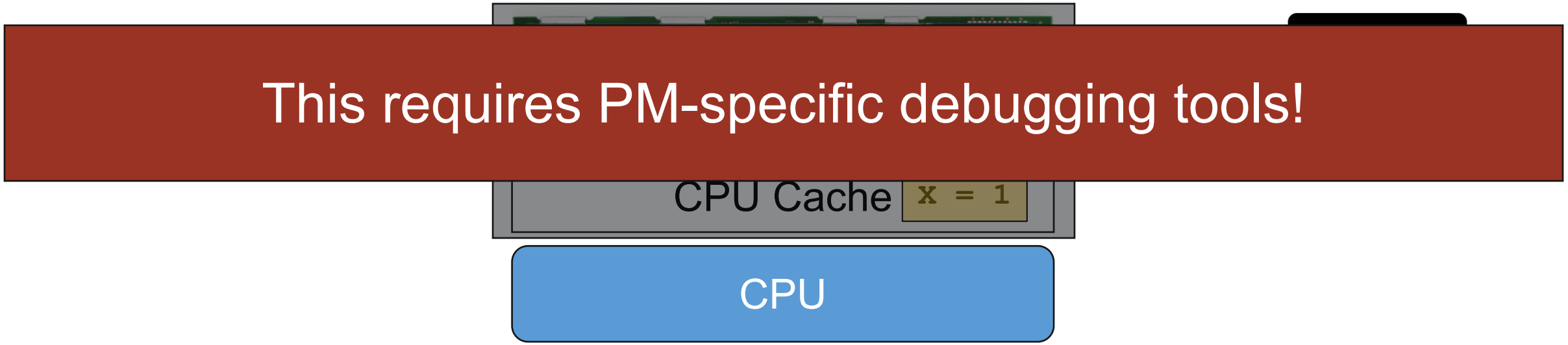
- Can have large consequences (data inconsistency, loss)
- Resulting bugs are hard to find (opaque, non-determinism)




```
1. store 1 into X
2. flush X
3. memory fence
```

Problem: PM Programming Requires Care (cont.)

- Can have large consequences (data inconsistency, loss)
- Resulting bugs are hard to find (opaque, non-determinism)




```
1. store 1 into X
2. flush X
3. memory fence
```

Prior Work

Prior Work

Yat: A Validation Framework for Persistent Memory Software

Philip Lantz, Subra

https://www.usenix.or

Development Topics & Technologies

intel

ors with Pmemcheck

Cross-Failure Bug Detection in Persistent Memory Programs

Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan
The International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020

Intel® Inspector

Deliver reliable applications. Locate and debug
errors in applications early in the design cycle
available as a stand-alone product or as part o
Studio.

PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs

Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan
International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019. [Paper] [Lightning Video] [Slides]

Prior Work

Yat: A Validation Framework for Persistent Memory Software

Philip Lantz, Subra

https://www.usenix.or

Development Topics & Technologies

intel

We want a tool that is ***automatic*** (low effort to use) and ***accurate*** (doesn't miss bugs).

Systems (ASPLOS), 2020

Intel® Inspector

Deliver reliable applications. Locate and debug errors in applications early in the design cycle available as a stand-alone product or as part o Studio.

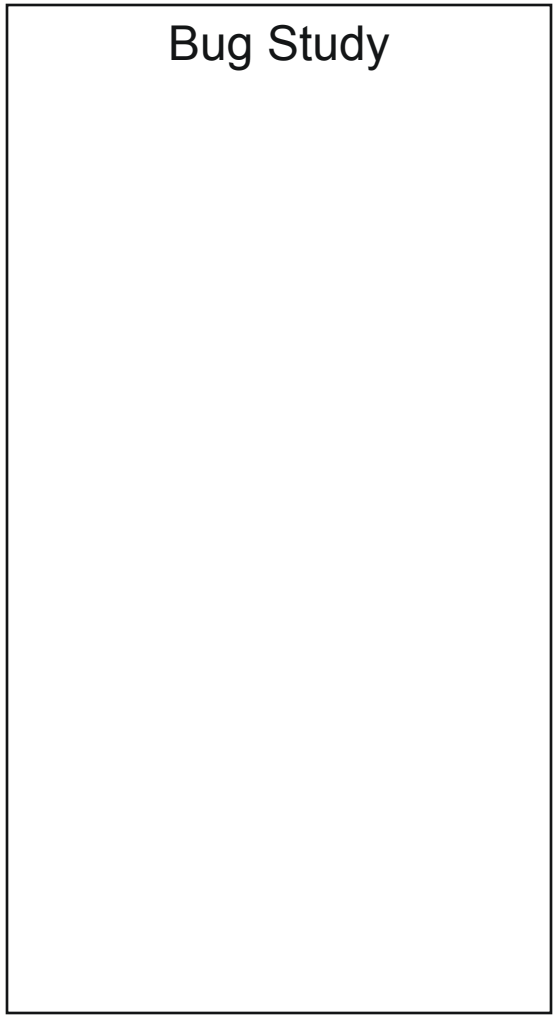
PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs

Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan

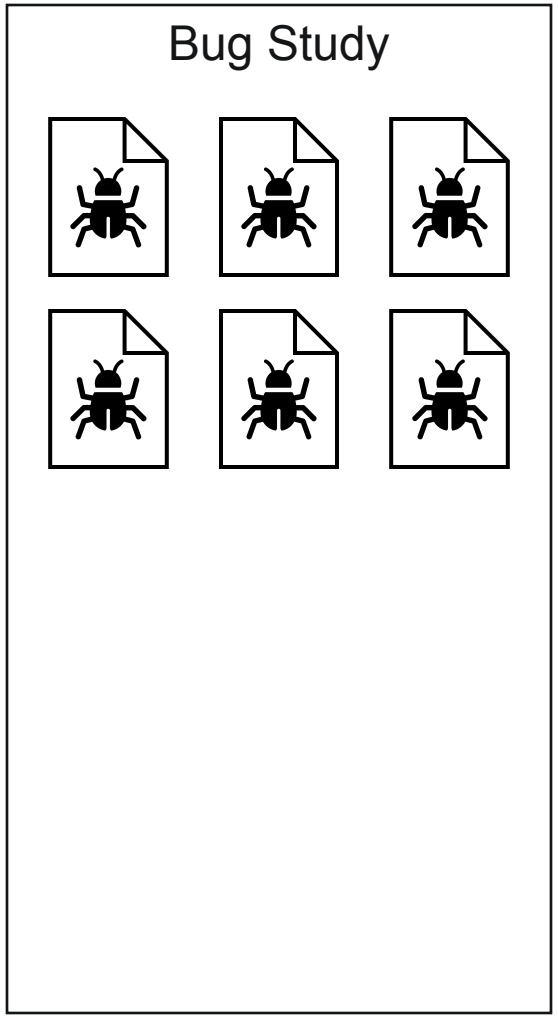
International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019. [Paper] [Lightning Video] [Slides]

Our Contributions

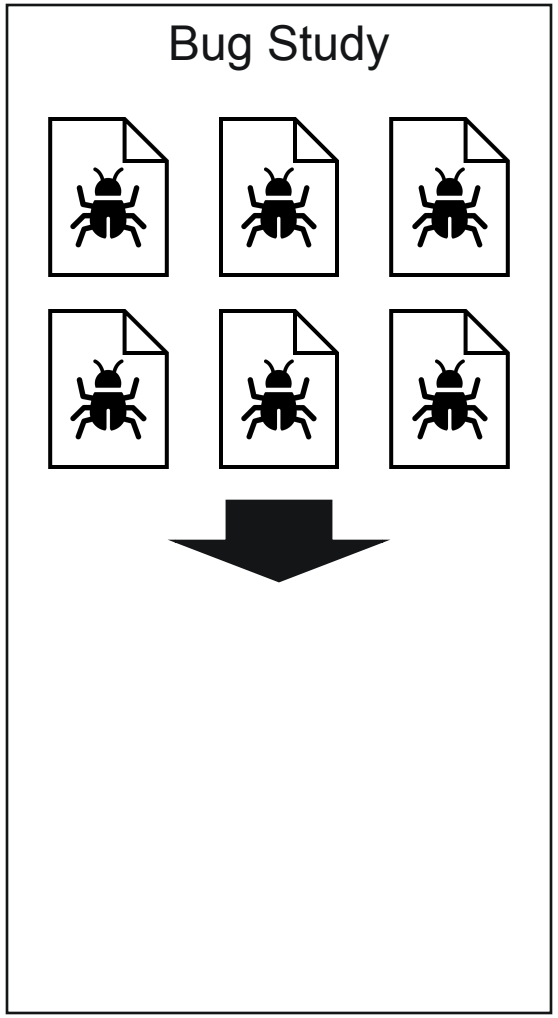
Our Contributions



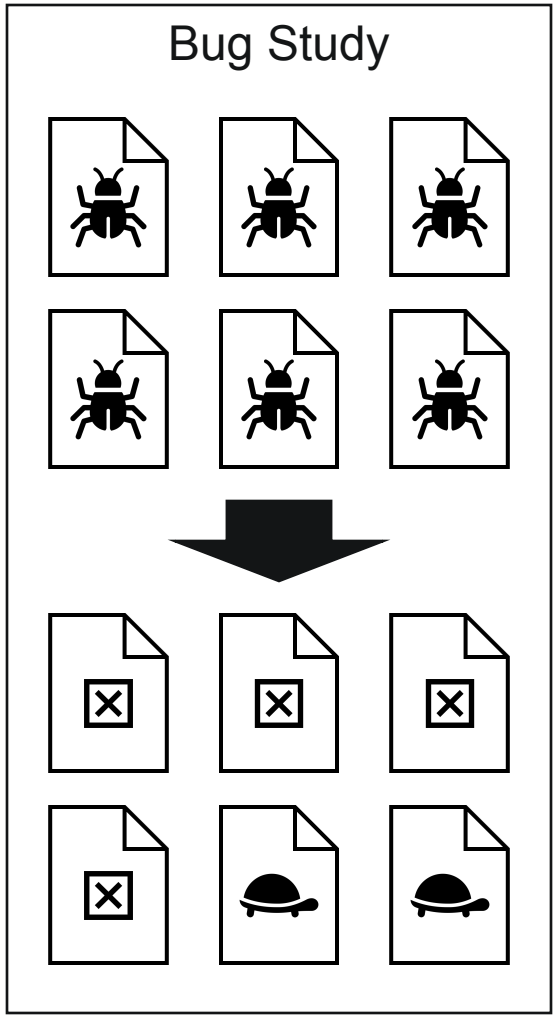
Our Contributions



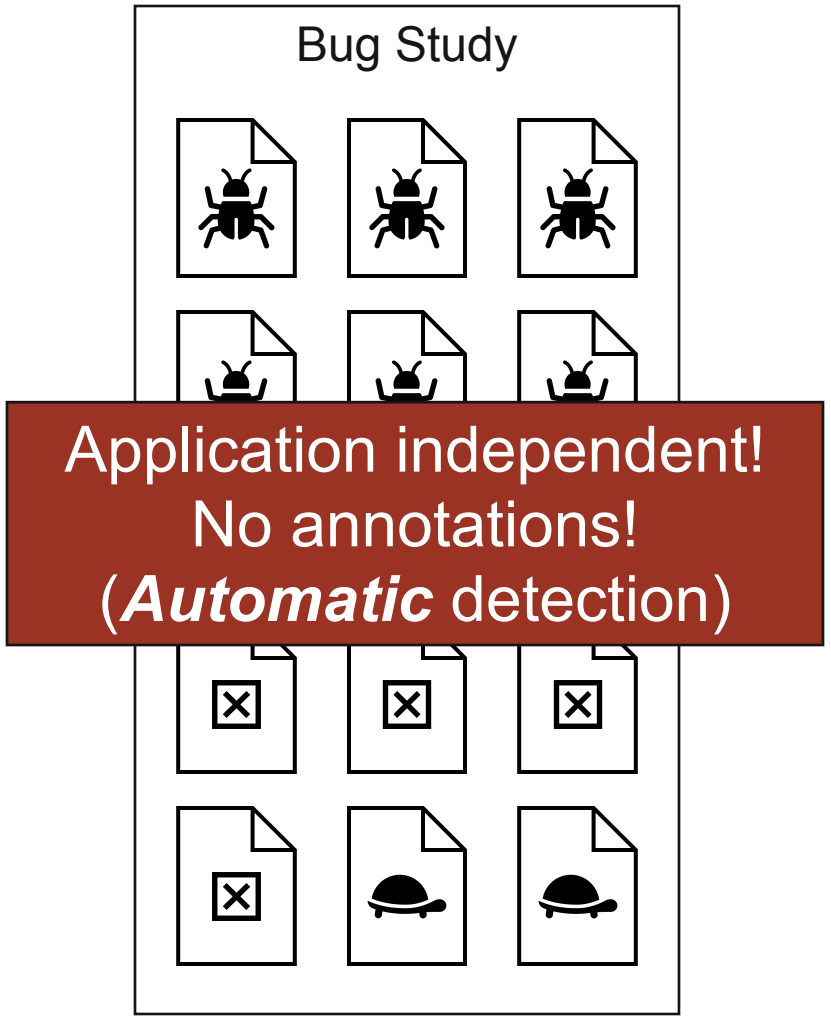
Our Contributions



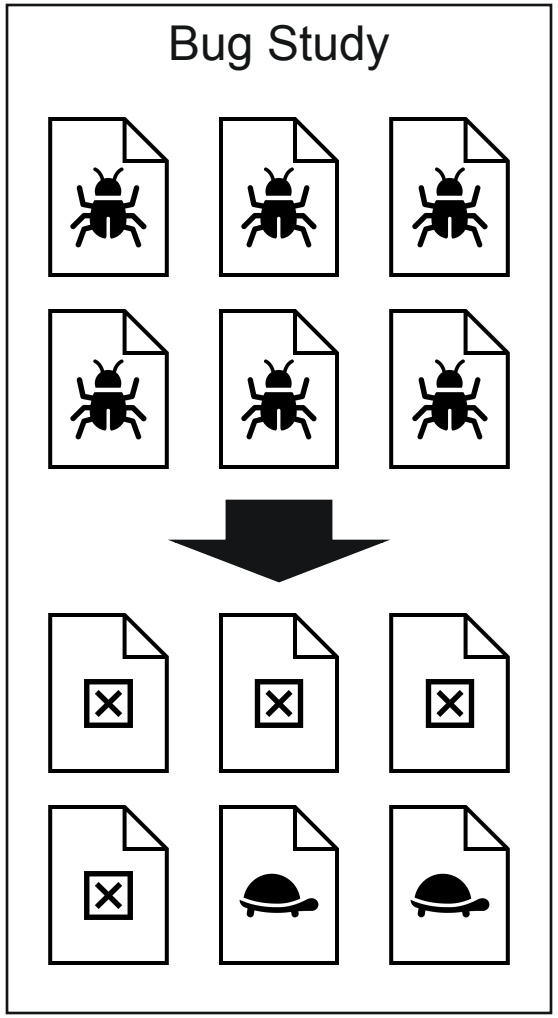
Our Contributions



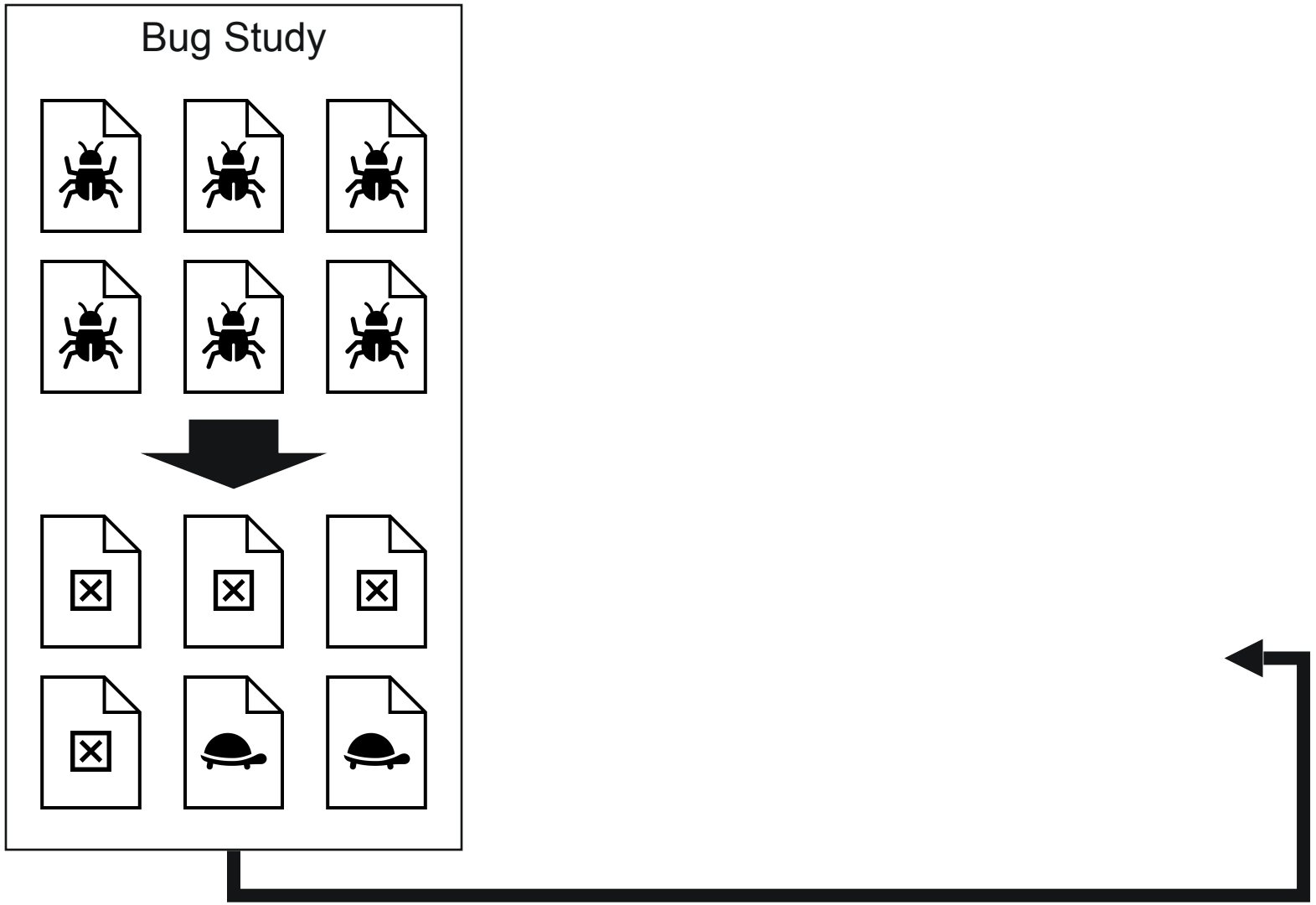
Our Contributions



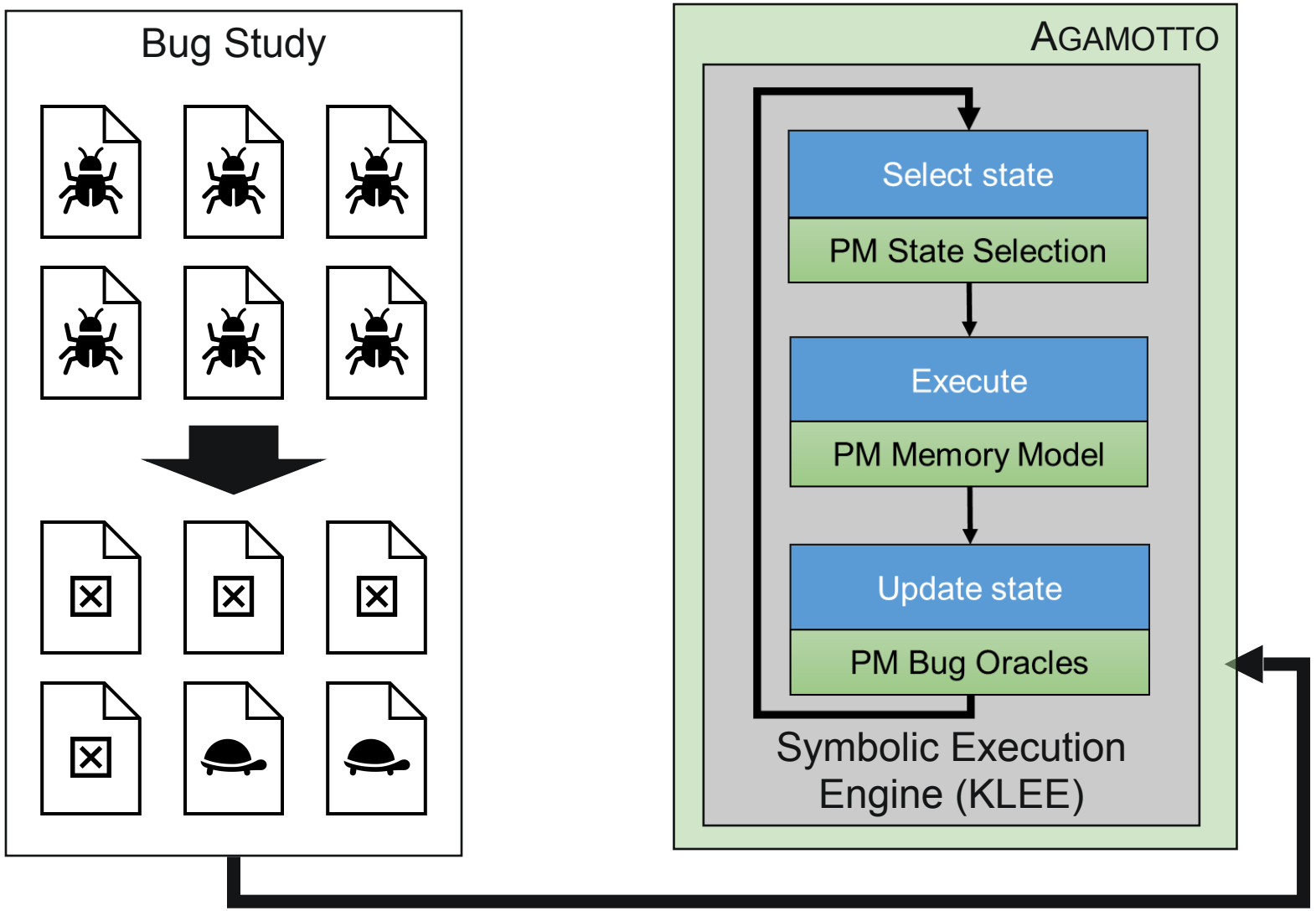
Our Contributions



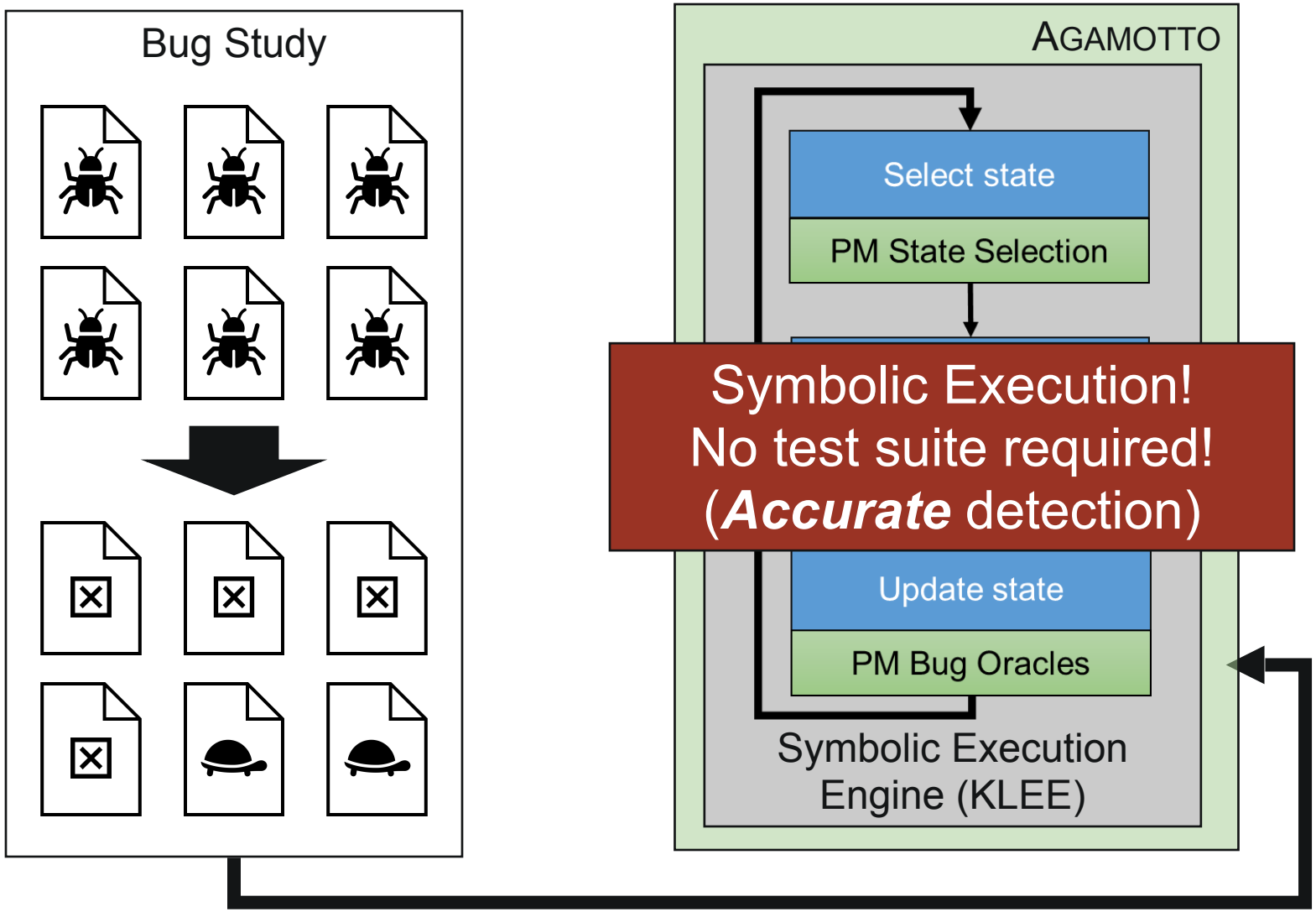
Our Contributions



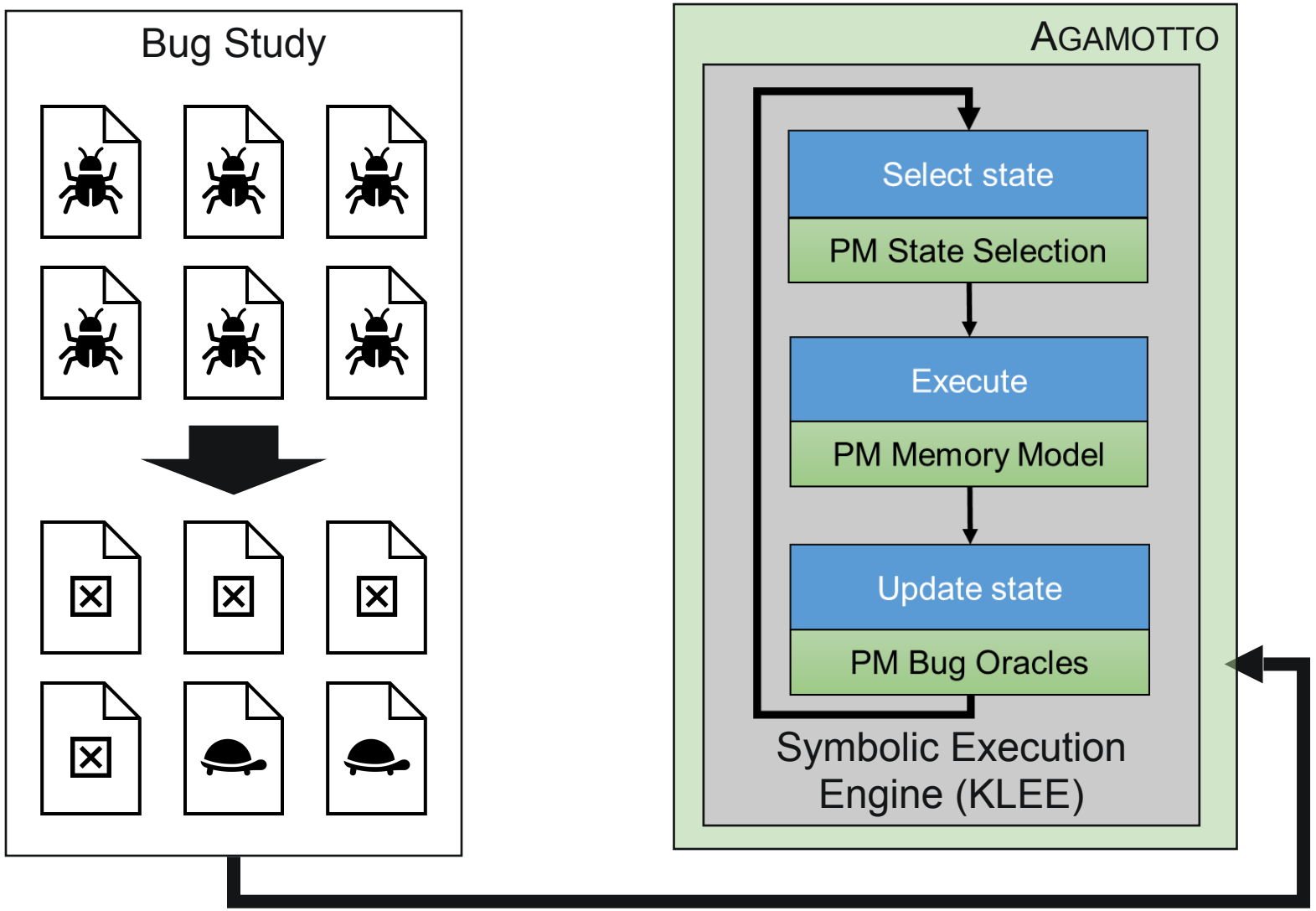
Our Contributions



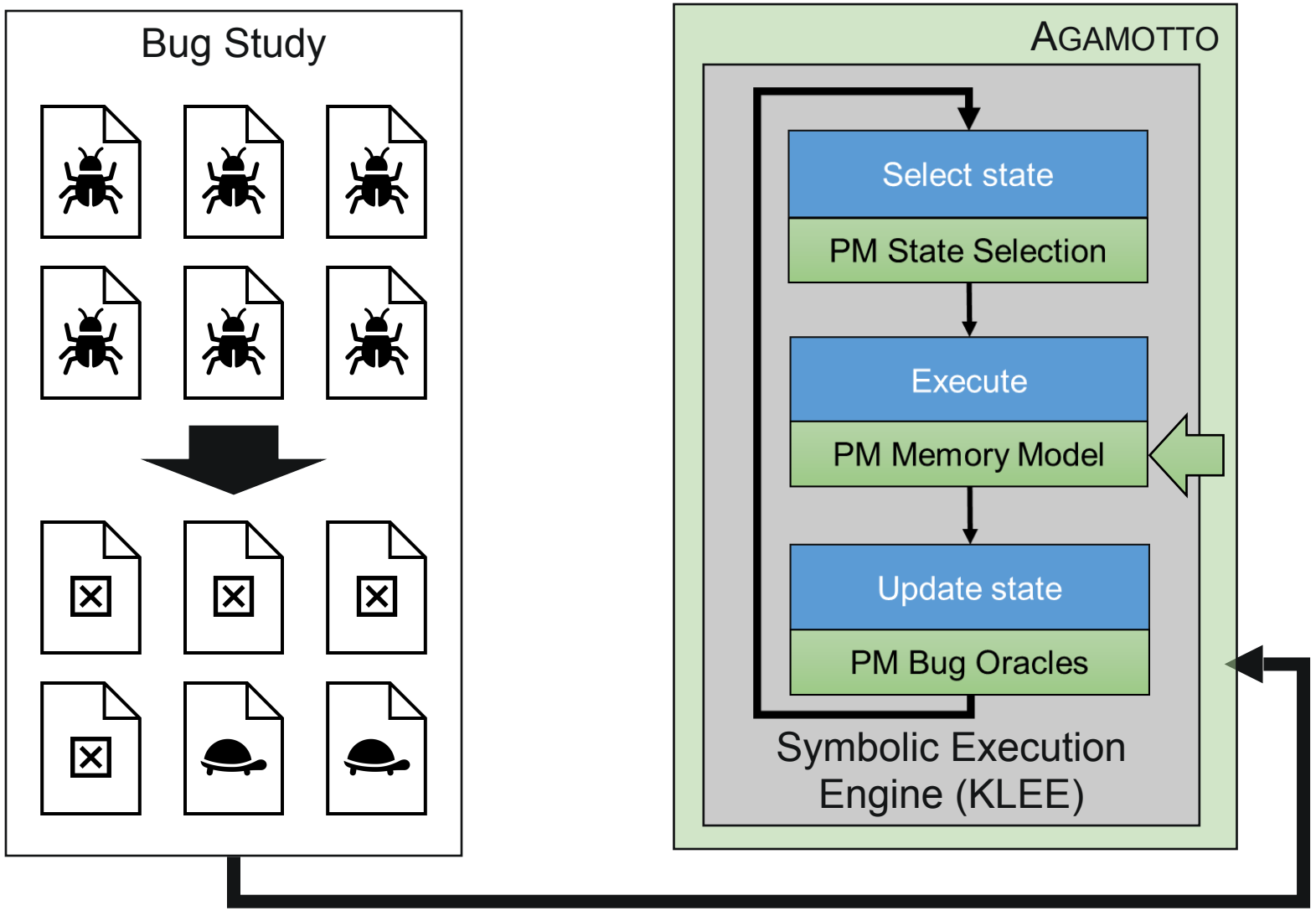
Our Contributions



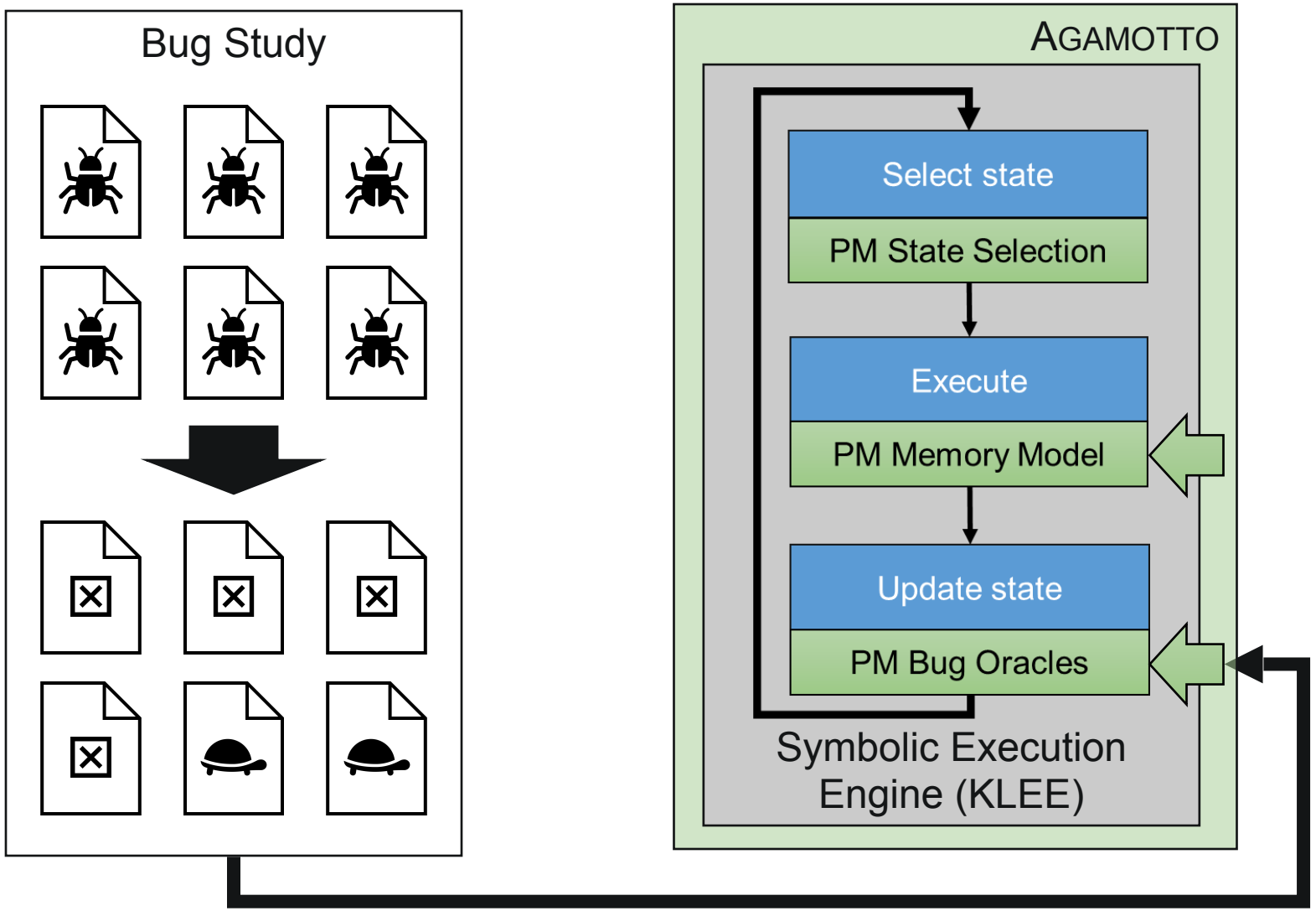
Our Contributions



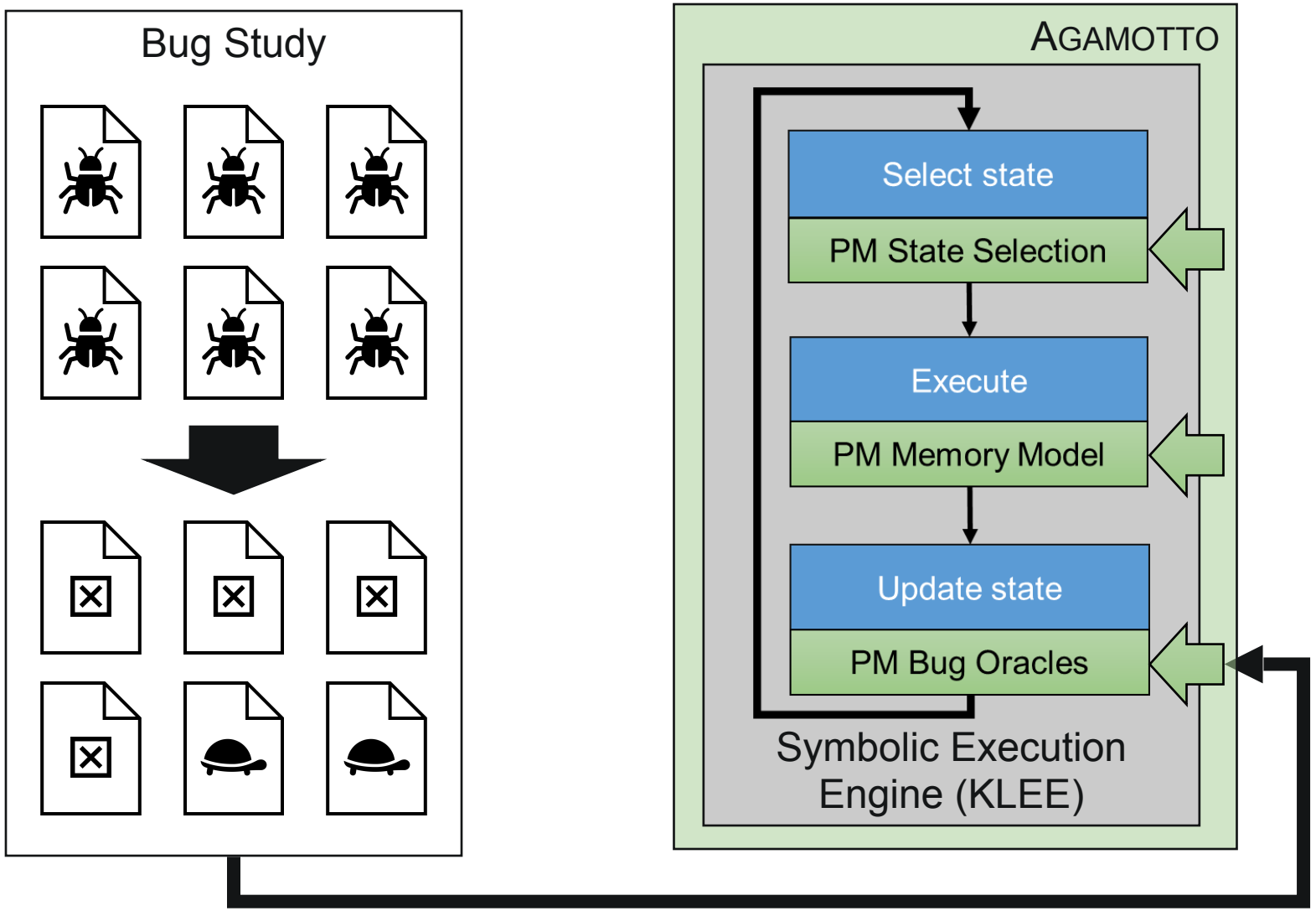
Our Contributions



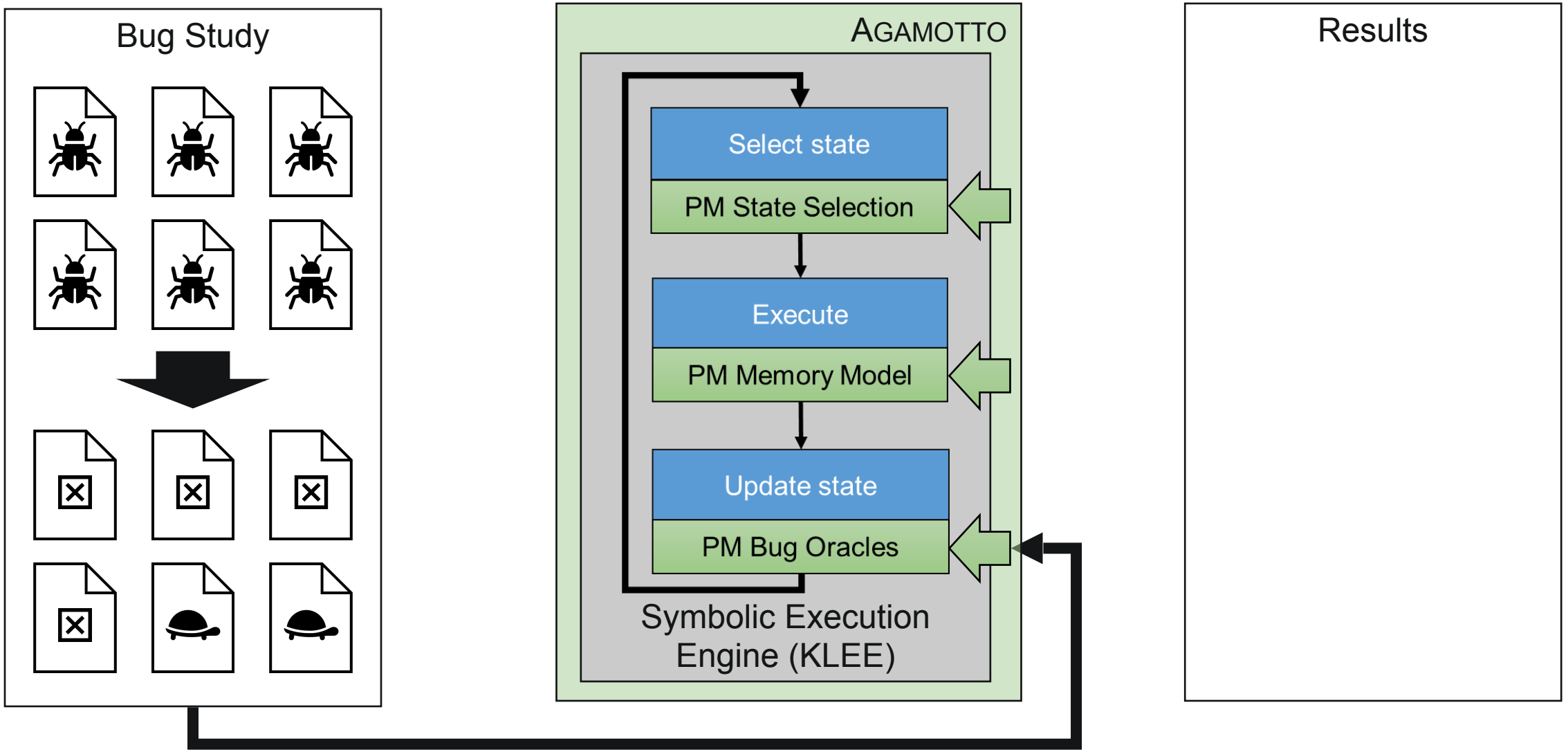
Our Contributions



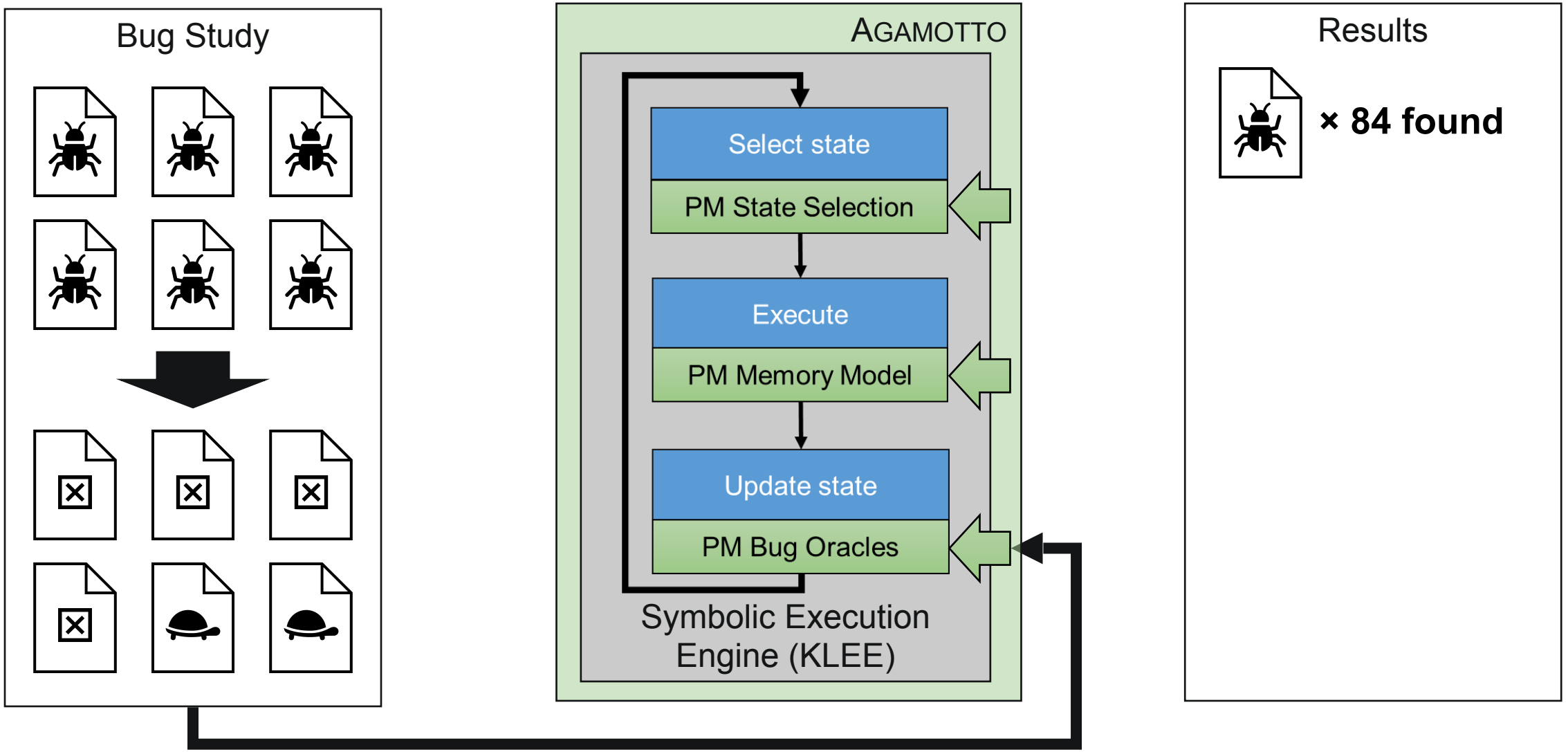
Our Contributions



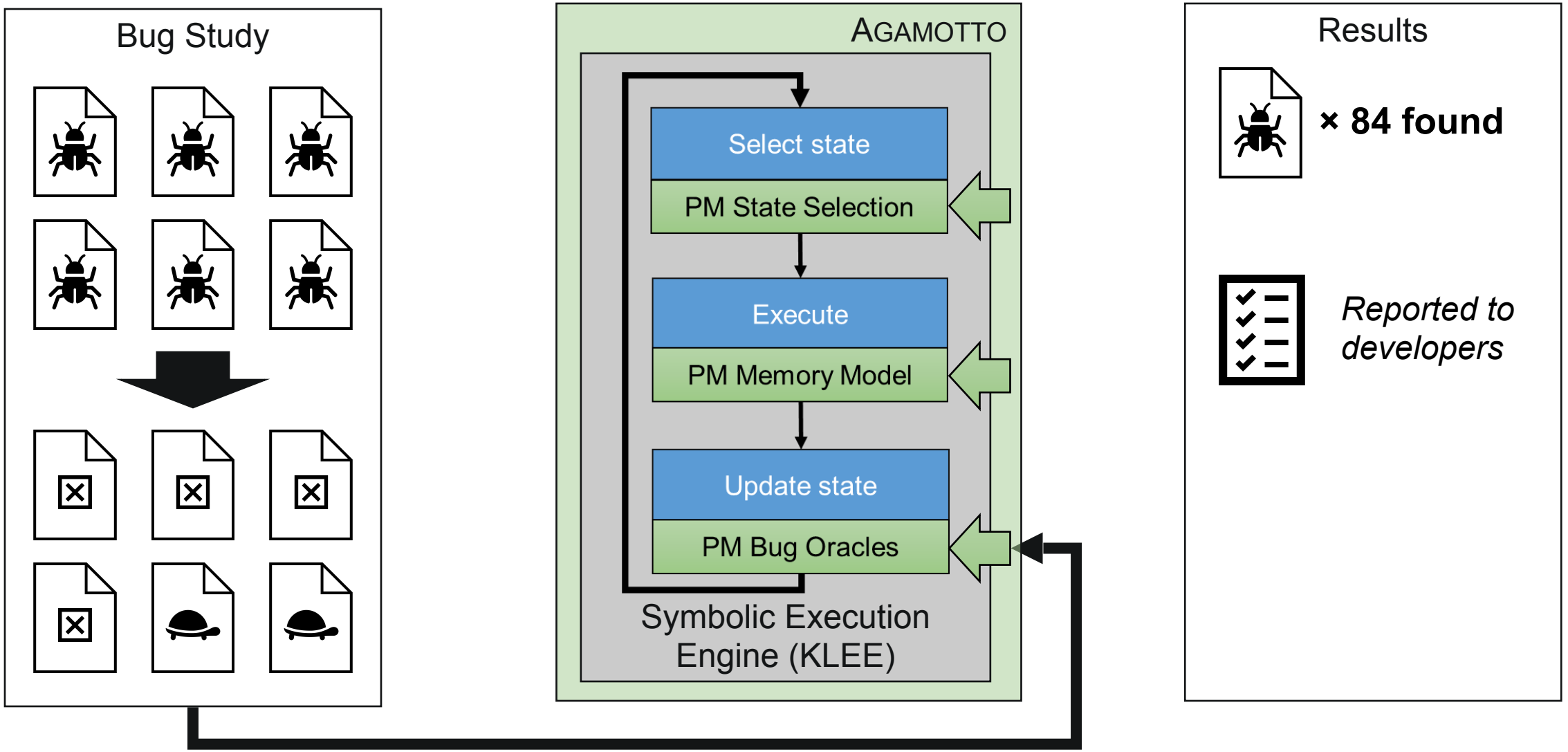
Our Contributions



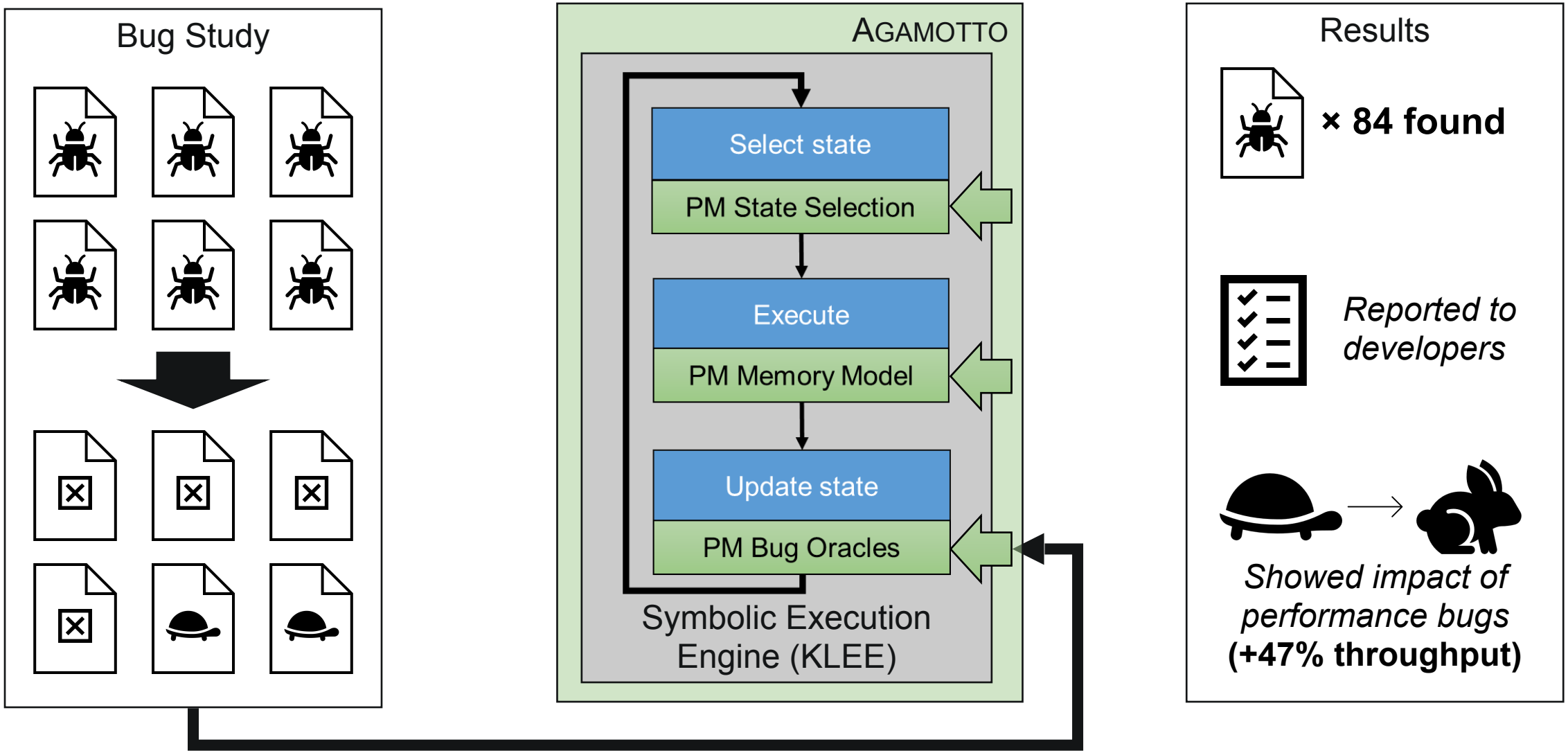
Our Contributions



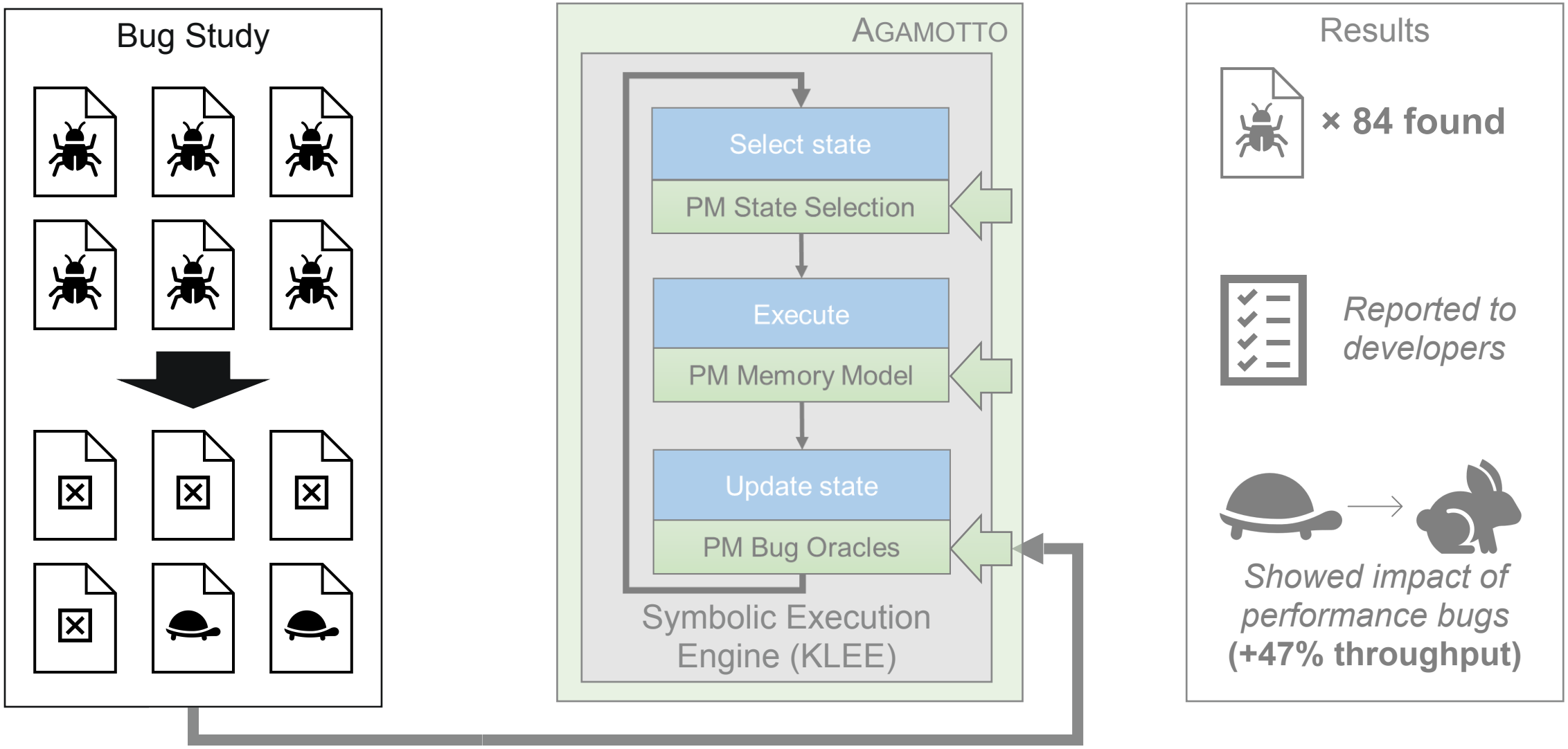
Our Contributions



Our Contributions



Overview


Bug Study
AGAMOTTO
Select state
PM State Selection
Execute
PM Memory Model
Update state
PM Bug Oracles
Symbolic Execution Engine (KLEE)
Results
× 84 found
Reported to developers
Showed impact of performance bugs
(+47% throughput)

PM Bug Study

PM Bug Study

- Surveyed 63 previously recorded bugs
  - Most from PMDK's issue tracker
  - Also analyzed bugs from PMTest, XFDetector

PM Bug Study

- Surveyed 63 previously recorded bugs
  - Most from PMDK's issue tracker
  - Also analyzed bugs from PMTest, XFDetector
- Identified two predominant patterns
  - Later use these to build bug detectors (i.e., bug *oracles)*

(1) Missing flush/fence

(1) Missing flush/fence

- Example: `oid` should be made durable, but never is

```
1   // oid is a pointer to PM
2   if (if_free != 0) {
3       *oid = NULL;
4       // BUG: missing flush and fence!
5   }
```

(1) Missing flush/fence

- Example: **`oid`** should be made durable, but never is

```
1   // oid is a pointer to PM
2   if (if_free != 0) {
3       *oid = NULL;
4       // BUG: missing flush and fence!
5   }
```

(1) Missing flush/fence

- Example: `oid` should be made durable, but never is

```
1  // oid is a pointer to PM
2  if (if_free != 0) {
3      *oid = NULL;
4      // BUG: missing flush and fence!
5  }
```

- Not making updates explicitly durable in PM

(1) Missing flush/fence

- Example: **`oid`** should be made durable, but never is

```
1   // oid is a pointer to PM
2   if (if_free != 0) {
3       *oid = NULL;
4       // BUG: missing flush and fence!
5   }
```

- Not making updates explicitly durable in PM
- Not tied to any application-specific logic

(1) Missing flush/fence

- Example: **`oid`** should be made durable, but never is

```
1   // oid is a pointer to PM
2   if (if_free != 0) {
3       *oid = NULL;
4       // BUG: missing flush and fence!
5   }
```

- Not making updates explicitly durable in PM
- Not tied to any application-specific logic
- Accounts for **79%** of the bugs in our survey (50/63)

(1) Missing flush/fence

- Example: **`oid`** should be made durable, but never is

```
1  // oid is a pointer to PM
2  if (if_free != 0) {
3      *oid = NULL;
4      // BUG: missing flush and fence!
5  }
```

- Not making updates explicitly durable in PM
- Not tied to any application-specific logic
- Accounts for **79%** of the bugs in our survey (50/63)
- Not guaranteed to be false-positive free
  - We don’t discover any in our testing

(2) Redundant flush/fence

(2) Redundant flush/fence

- Example: overuse of flush/fence instructions

```
1  // oid is a pointer to PM
2  if (if_free != 0) {
3      *oid = NULL;
4      FLUSH(*oid);
5      FENCE();
6  }
7  // BUG: oid is already persistent!
8  FLUSH(*oid);
9  FENCE();
```

(2) Redundant flush/fence

- Example: overuse of flush/fence instructions

```
1  // oid is a pointer to PM
2  if (if_free != 0) {
3      *oid = NULL;
4      FLUSH(*oid);
5      FENCE();
6  }
7  // BUG: oid is already persistent!
8  FLUSH(*oid);
9  FENCE();
```

(2) Redundant flush/fence

- Example: overuse of flush/fence instructions

```
1  // oid is a pointer to PM
2  if (if_free != 0) {
3      *oid = NULL;
4      FLUSH(*oid);
5      FENCE();
6  }
7  // BUG: oid is already persistent!
8  FLUSH(*oid);
9  FENCE();
```

(2) Redundant flush/fence

- Example: overuse of flush/fence instructions

```
1  // oid is a pointer to PM
2  if (if_free != 0) {
3      *oid = NULL;
4      FLUSH(*oid);
5      FENCE();
6  }
7  // BUG: oid is already persistent!
8  FLUSH(*oid);
9  FENCE();
```

- Unnecessary use of durability mechanisms

(2) Redundant flush/fence

- Example: overuse of flush/fence instructions

```
1  // oid is a pointer to PM
2  if (if_free != 0) {
3      *oid = NULL;
4      FLUSH(*oid);
5      FENCE();
6  }
7  // BUG: oid is already persistent!
8  FLUSH(*oid);
9  FENCE();
```

- Unnecessary use of durability mechanisms
- Is a performance issue regardless of program logic

(2) Redundant flush/fence

- Example: overuse of flush/fence instructions

```
1  // oid is a pointer to PM
2  if (if_free != 0) {
3      *oid = NULL;
4      FLUSH(*oid);
5      FENCE();
6  }
7  // BUG: oid is already persistent!
8  FLUSH(*oid);
9  FENCE();
```

- Unnecessary use of durability mechanisms
- Is a performance issue regardless of program logic
- Accounts for **11%** of the bugs in our survey (7/63)

Bug Study Conclusions

Bug Study Conclusions

- Two patterns, **90%** of the bugs (57/63), *are application independent!*
  - Can automated their detection

Bug Study Conclusions

- Two patterns, **90%** of the bugs (57/63), *are application independent!*
  - Can automated their detection

- Application-specific bugs, **~10%** of the bugs (6/63), are still important
  - Allow developers to define their own patterns!

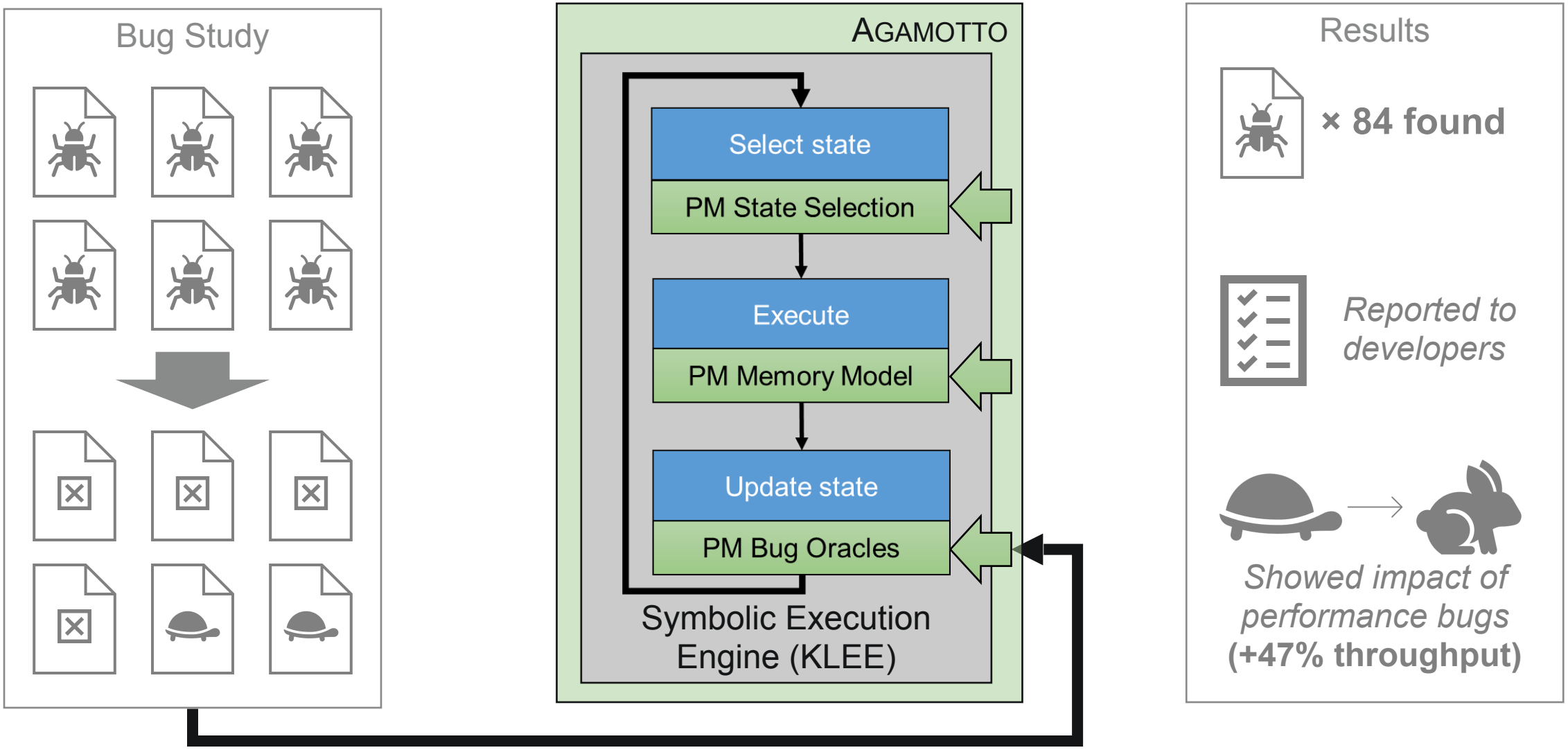
Bug Study Conclusions

- Two patterns, **90%** of the bugs (57/63), *are application independent!*
  - Can automated their detection

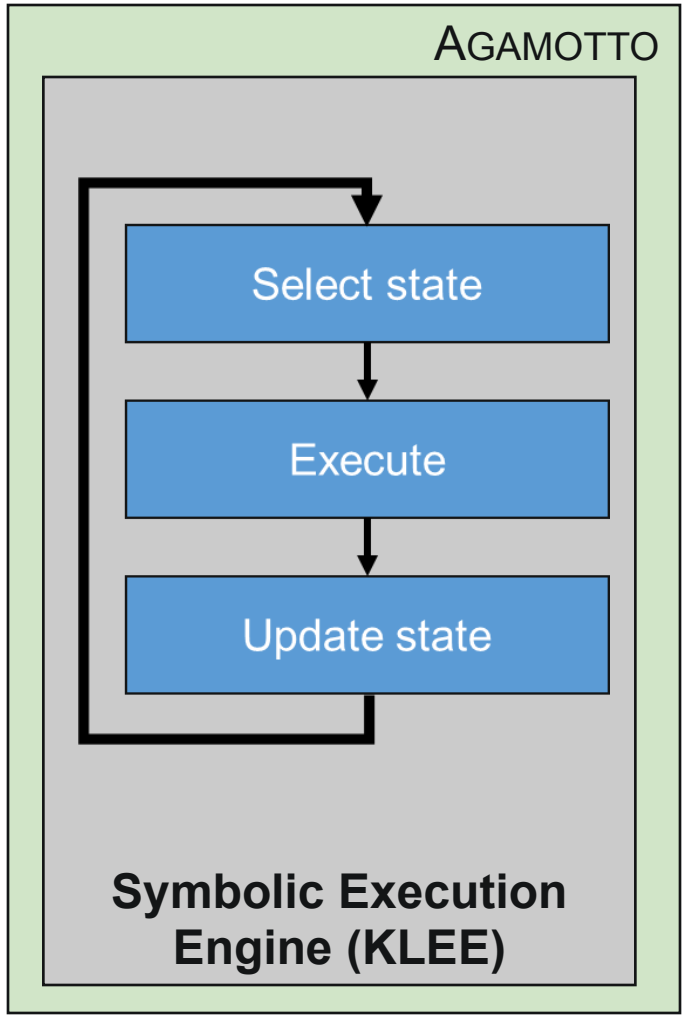
Pattern-based detection allows us to avoid source code modifications!

  - Allow developers to define their own patterns!

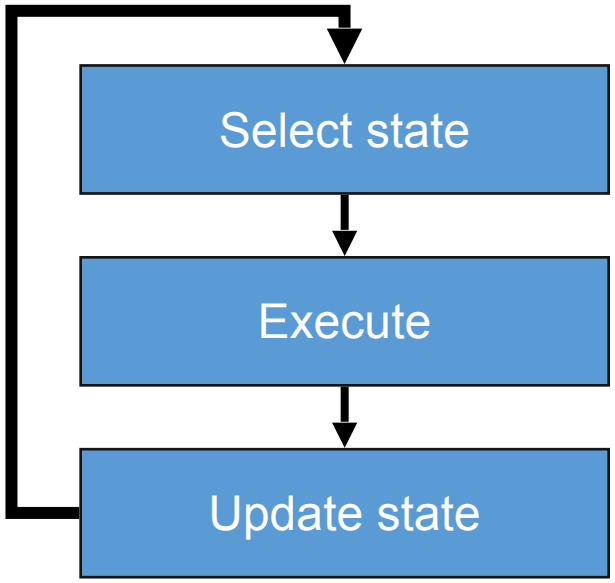
Overview


Bug Study
AGAMOTTO
Select state
PM State Selection
Execute
PM Memory Model
Update state
PM Bug Oracles
Symbolic Execution Engine (KLEE)
Results
× 84 found
Reported to developers
Showed impact of performance bugs
(+47% throughput)

Overview


AGAMOTTO
Select state
Execute
Update state
Symbolic Execution Engine (KLEE)

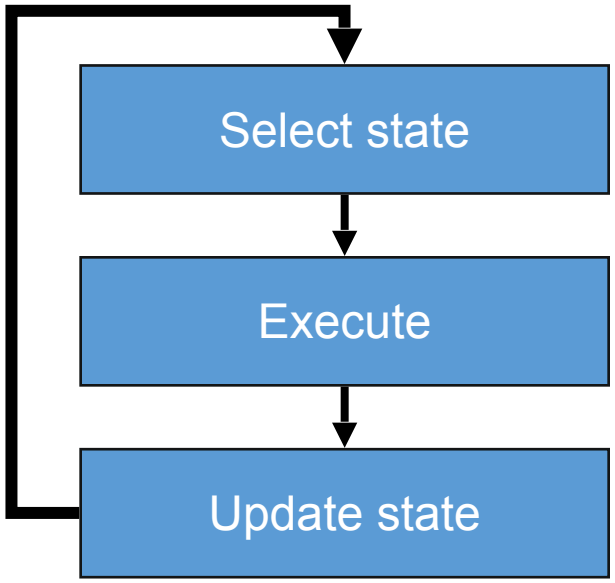
How Symbolic Execution Works




```
do_read = ?
```

```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

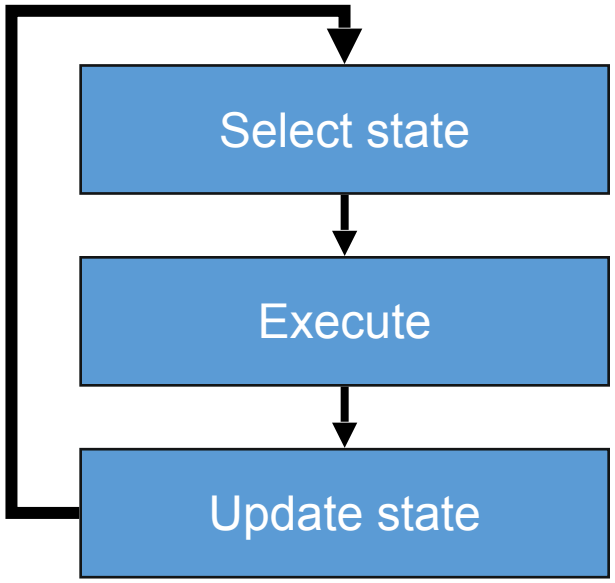
How Symbolic Execution Works



do_read = ?

```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

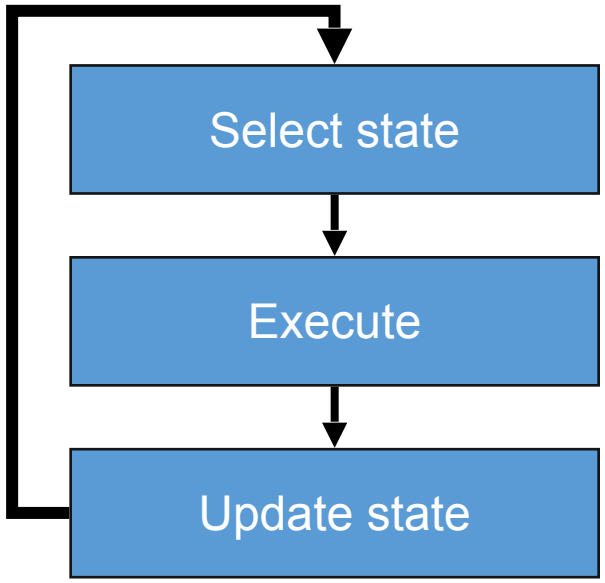
How Symbolic Execution Works



do_read = λ

```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

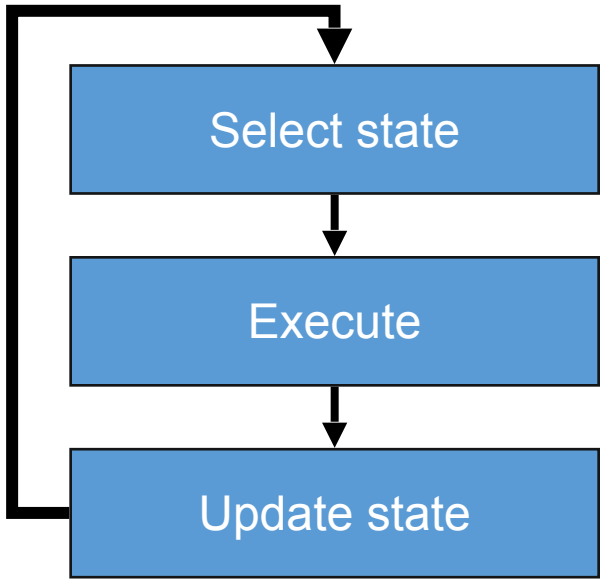
How Symbolic Execution Works



do_read = λ

```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

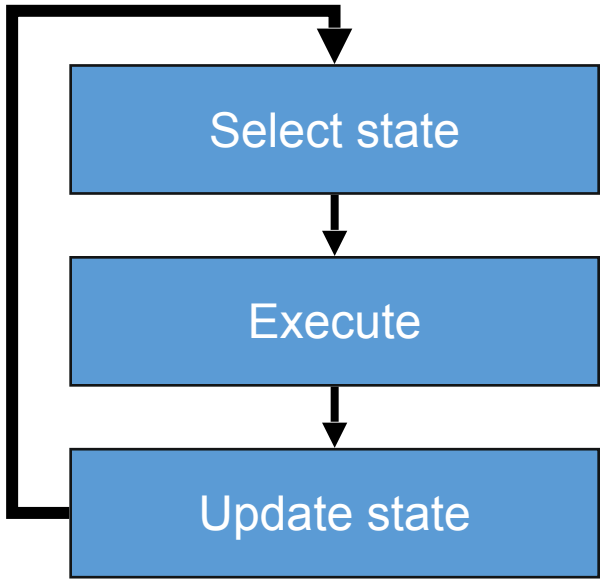
How Symbolic Execution Works



do_read = λ

```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

How Symbolic Execution Works

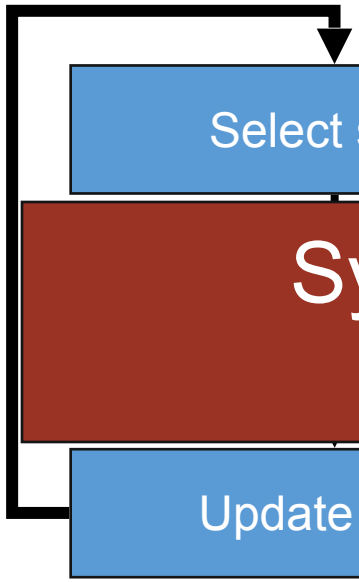


do_read = λ≠0

do_read = 0

```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

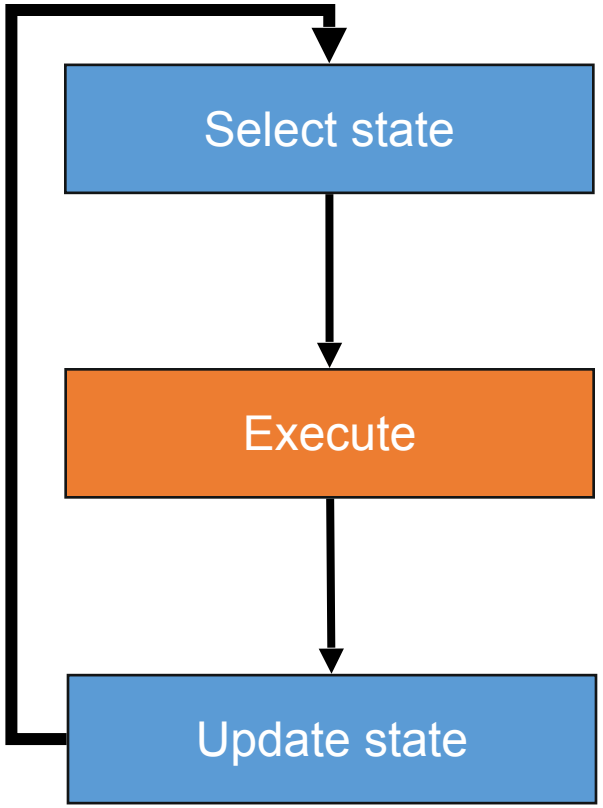
How Symbolic Execution Works




```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
9        pbuf[x] = a
10       clwb(pbuf[x])
11       // BUG: Missing sfence!
12 exit(0)
```

Symbolic execution gives us high code coverage!
(no need for extensive test suites)

How does AGAMOTTO work?

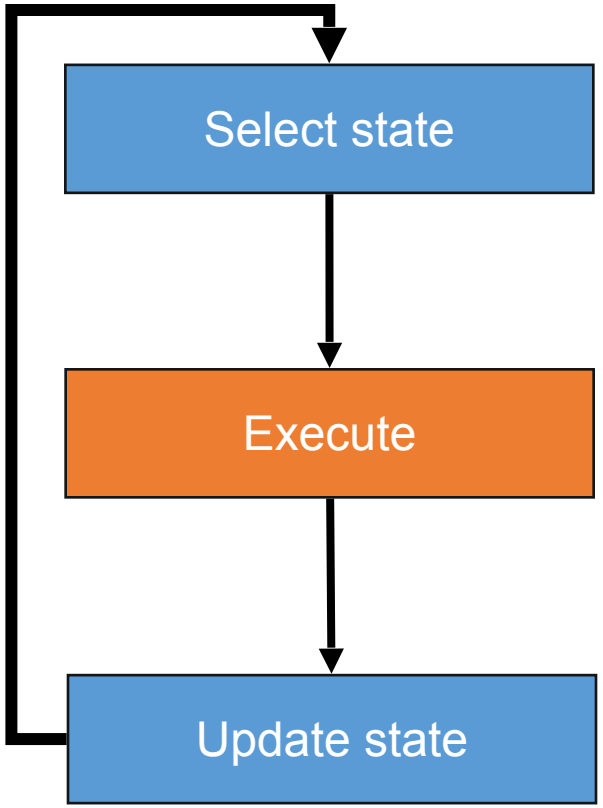


do_read = λ≠0

do_read = 0

```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

How does AGAMOTTO work?

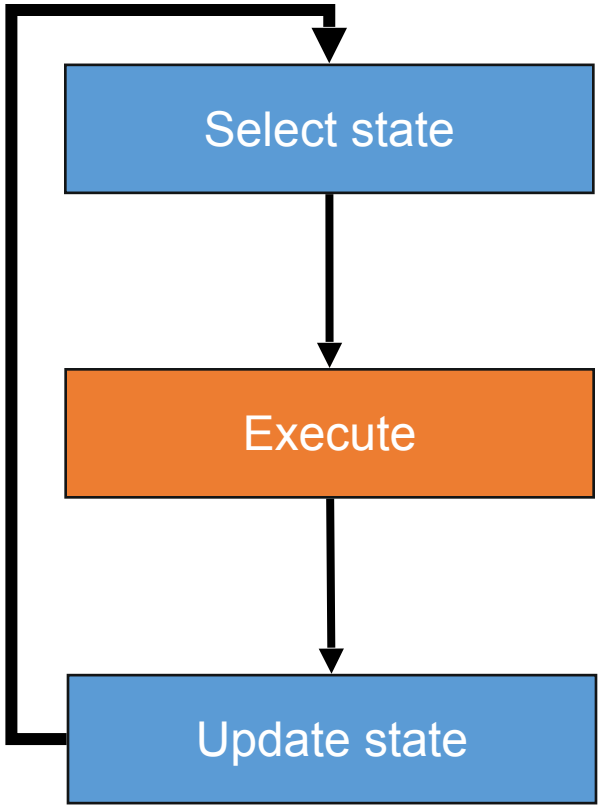



```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

do_read = λ≠0

do_read = 0

How does AGAMOTTO work?

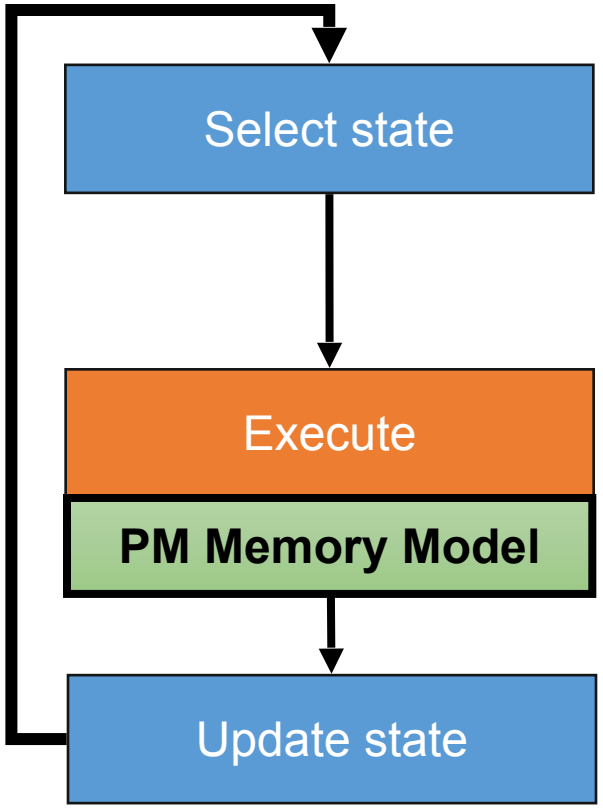



```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

do_read = λ≠0

do_read = 0

How does AGAMOTTO work?




```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

do_read = λ≠0

pbuf[x]: clean

do_read = 0

pbuf[x]: dirty

(1) PM Memory Model

(1) PM Memory Model

- Augment symbolic memory model with PM state
  - Allocated from calls to `mmap` from pmem files

(1) PM Memory Model

- Augment symbolic memory model with PM state
  - Allocated from calls to **`mmap`** from pmem files

(1) PM Memory Model

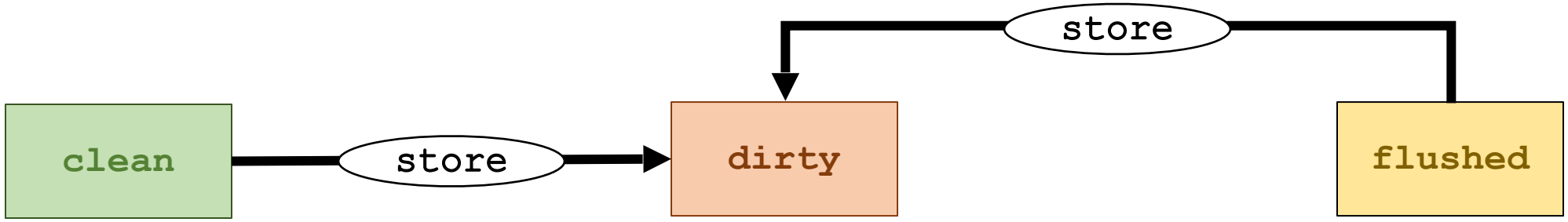
- Augment symbolic memory model with PM state
  - Allocated from calls to **`mmap`** from pmem files

(1) PM Memory Model

- Augment symbolic memory model with PM state
  - Allocated from calls to **`mmap`** from pmem files

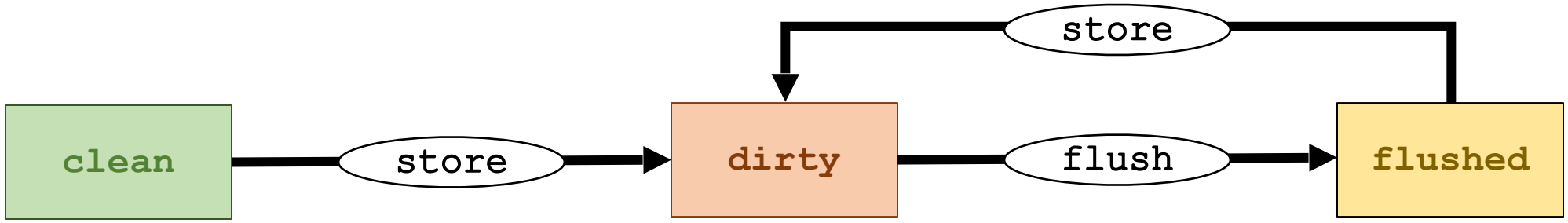
(1) PM Memory Model

- Augment symbolic memory model with PM state
  - Allocated from calls to **`mmap`** from pmem files



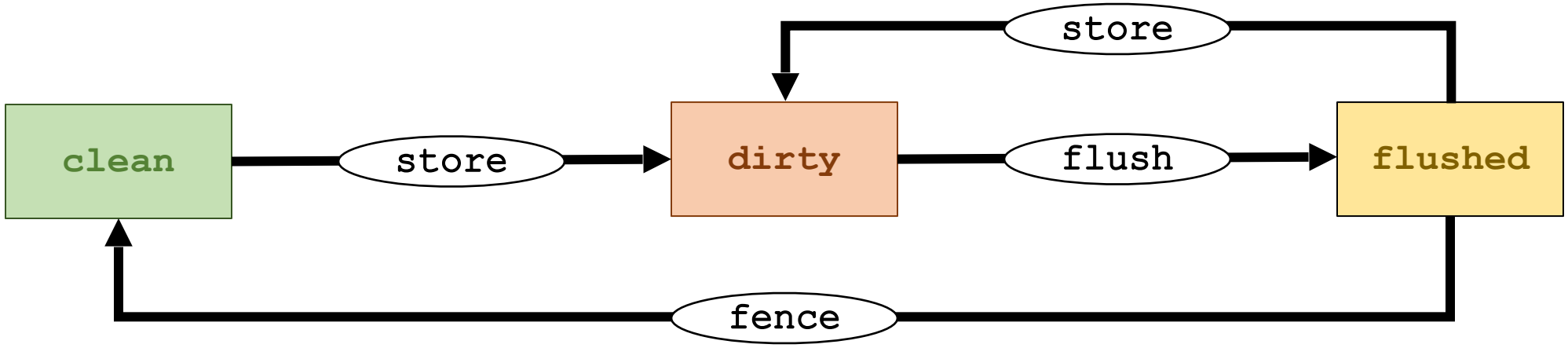
(1) PM Memory Model

- Augment symbolic memory model with PM state
  - Allocated from calls to **`mmap`** from pmem files

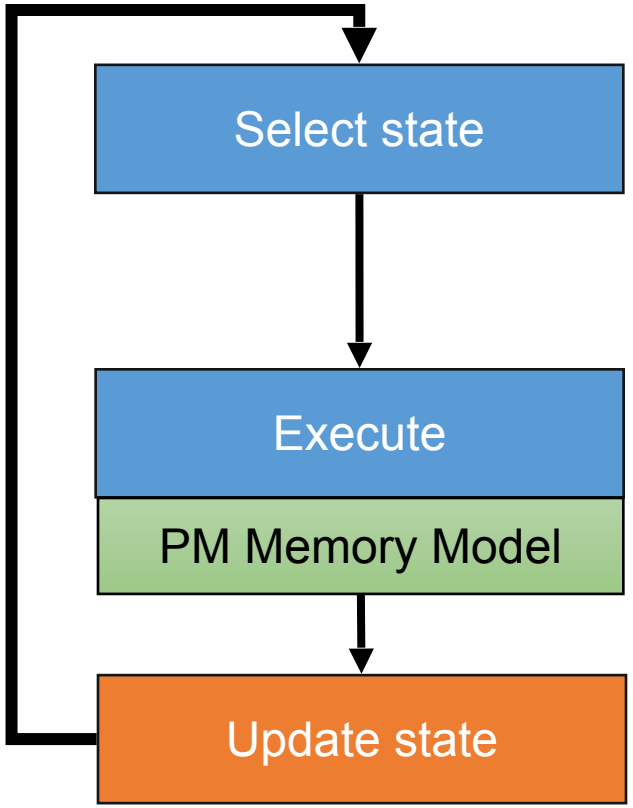


(1) PM Memory Model

- Augment symbolic memory model with PM state
  - Allocated from calls to **`mmap`** from pmem files



How does AGAMOTTO work?




```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

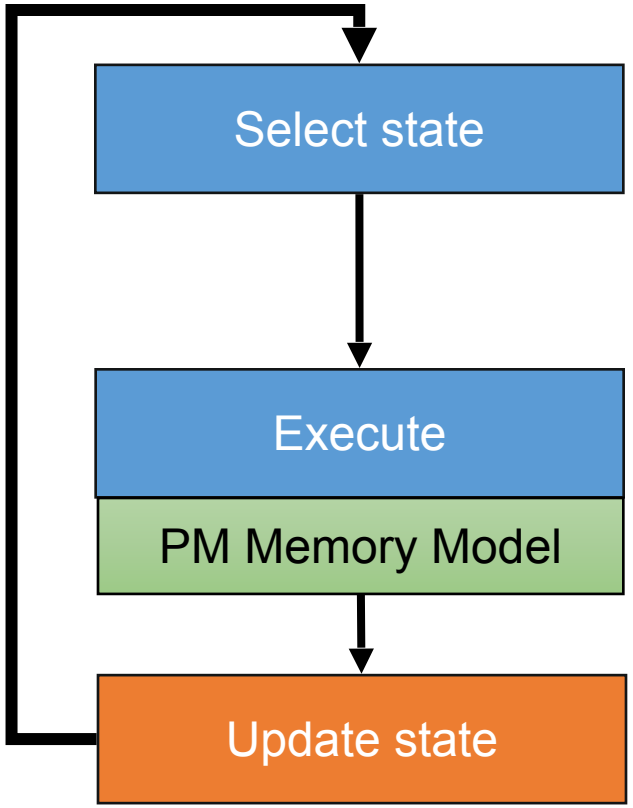
do_read = λ≠0

pbuf[x]: clean

do_read = 0

pbuf[x]: flushed

How does AGAMOTTO work?




```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

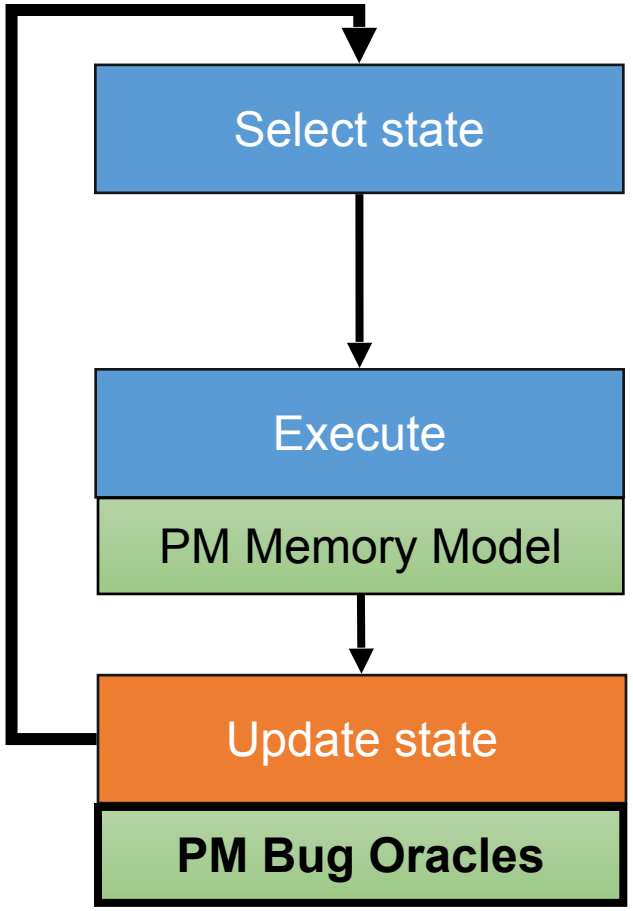
do_read = λ≠0

pbuf[x]: clean

do_read = 0

pbuf[x]: flushed

How does AGAMOTTO work?




```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

do_read = λ≠0

pbuf[x]: clean

do_read = 0

pbuf[x]: flushed

(2) PM Bug Detection

(2) PM Bug Detection

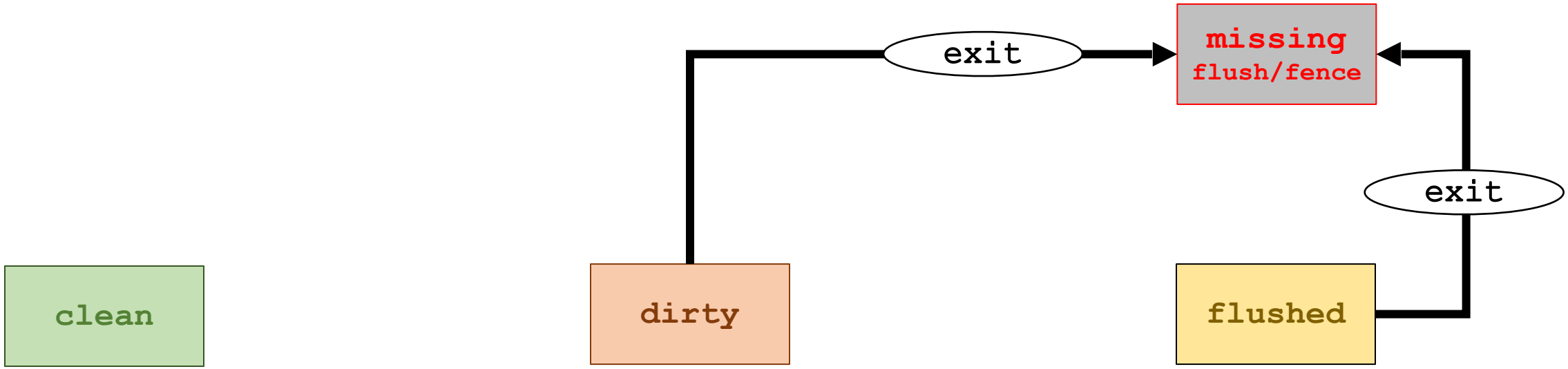
- Augment PM state transitions with bug signals

(2) PM Bug Detection

- Augment PM state transitions with bug signals

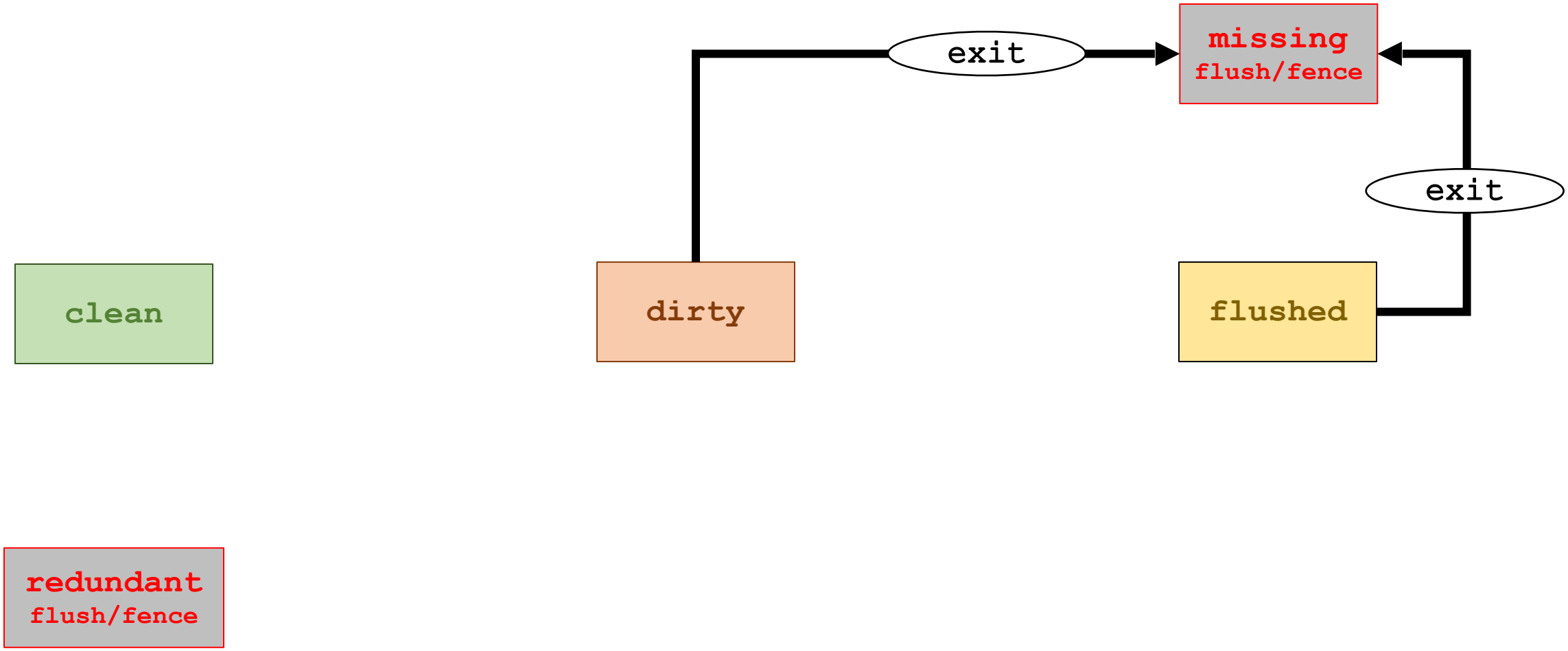
(2) PM Bug Detection

- Augment PM state transitions with bug signals



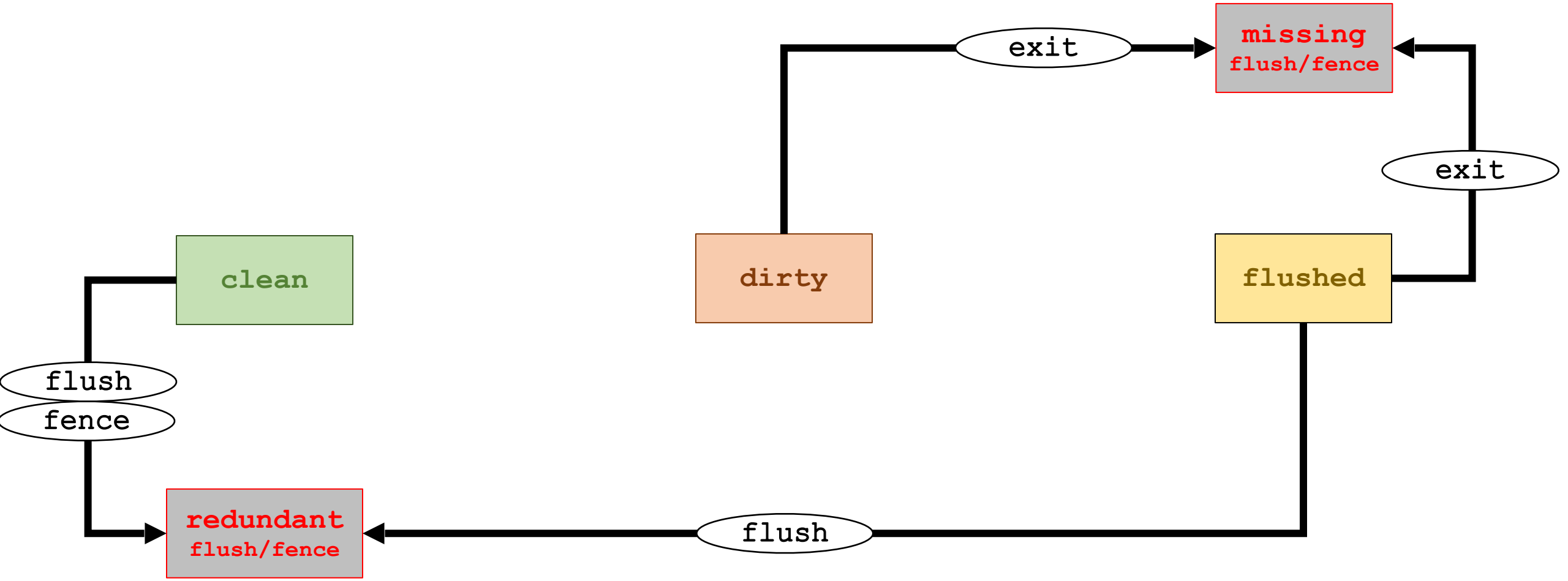
(2) PM Bug Detection

- Augment PM state transitions with bug signals



(2) PM Bug Detection

- Augment PM state transitions with bug signals



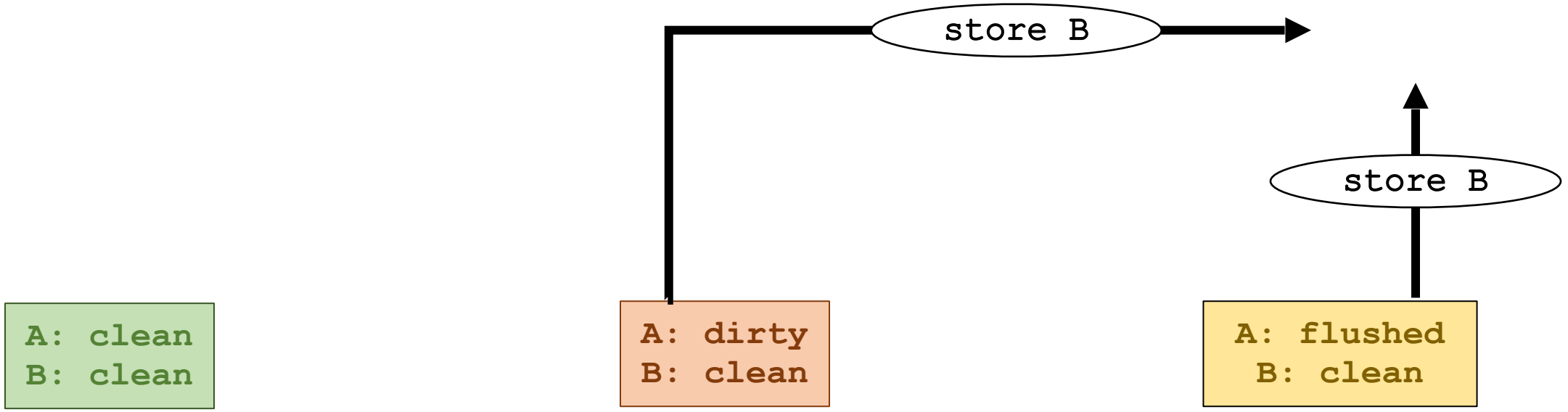
(2) PM Bug Detection

(2) PM Bug Detection

- Custom oracles = custom state & signals
  - Example: ordering bug (**A** must be durable **before B**)

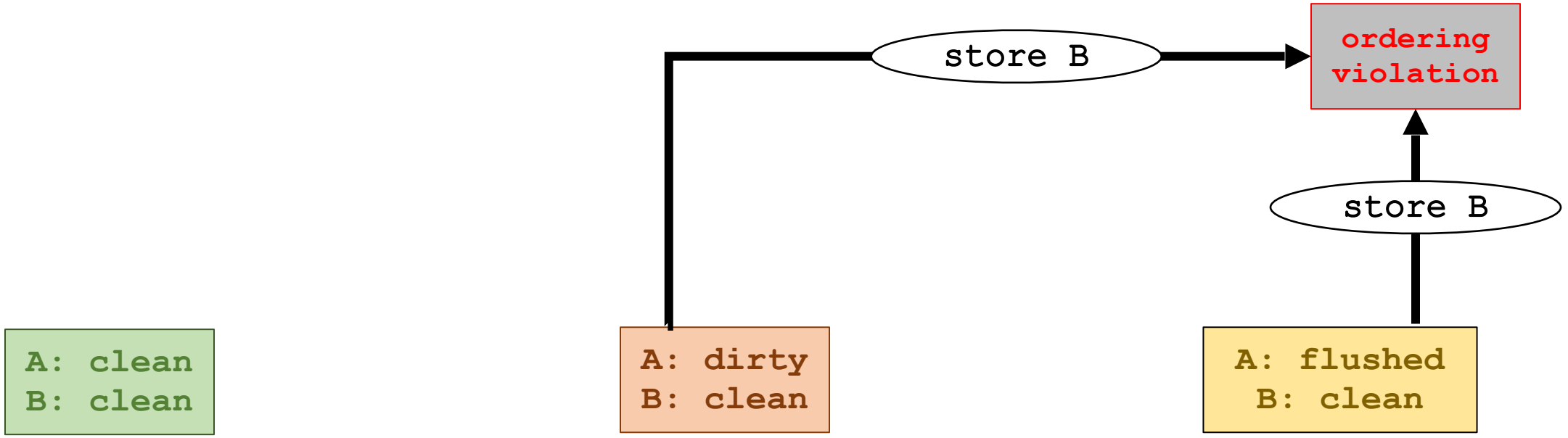
(2) PM Bug Detection

- Custom oracles = custom state & signals
  - Example: ordering bug (**A** must be durable **before B**)



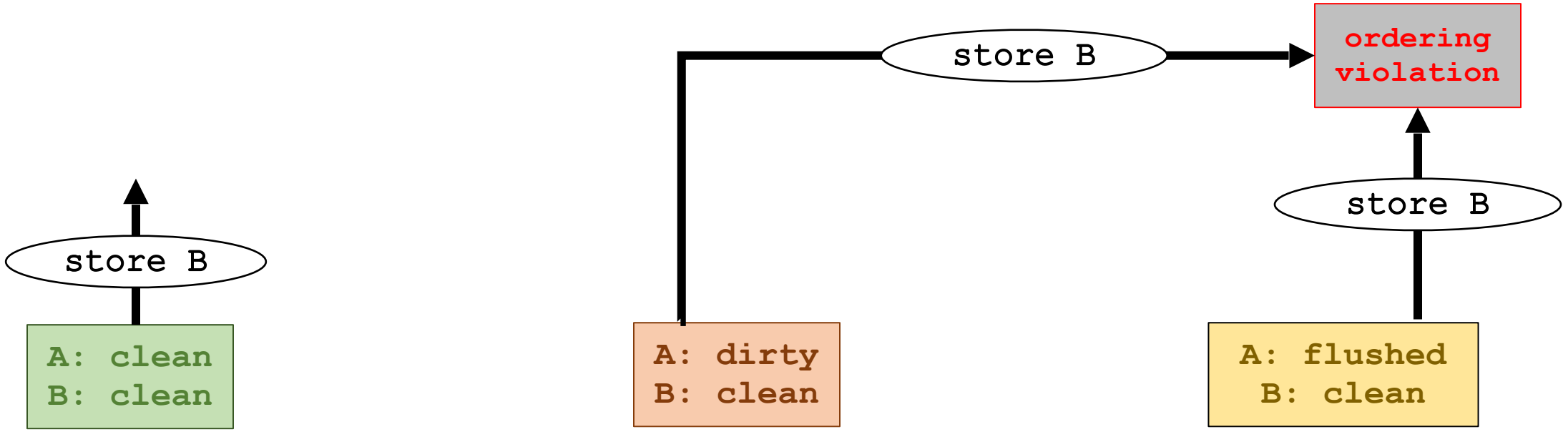
(2) PM Bug Detection

- Custom oracles = custom state & signals
  - Example: ordering bug (**A** must be durable **before B**)



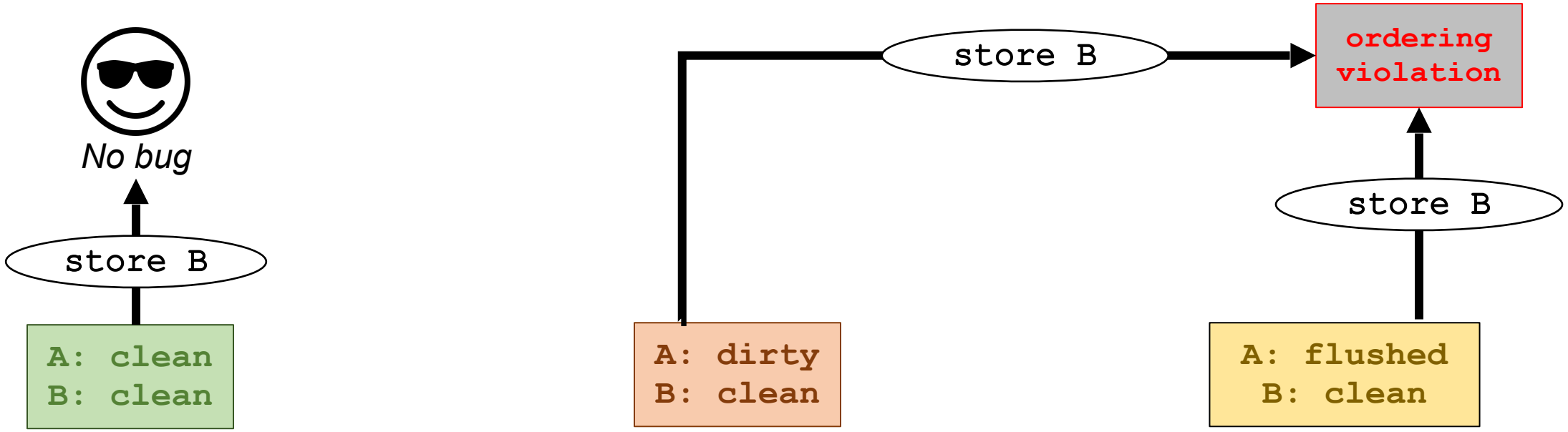
(2) PM Bug Detection

- Custom oracles = custom state & signals
  - Example: ordering bug (**A** must be durable **before B**)



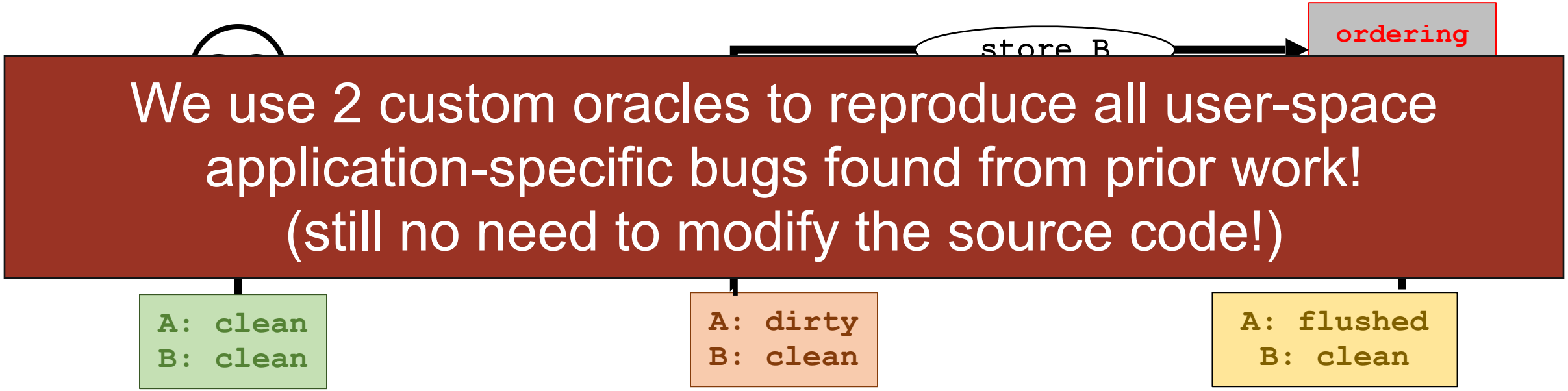
(2) PM Bug Detection

- Custom oracles = custom state & signals
  - Example: ordering bug (**A** must be durable **before B**)

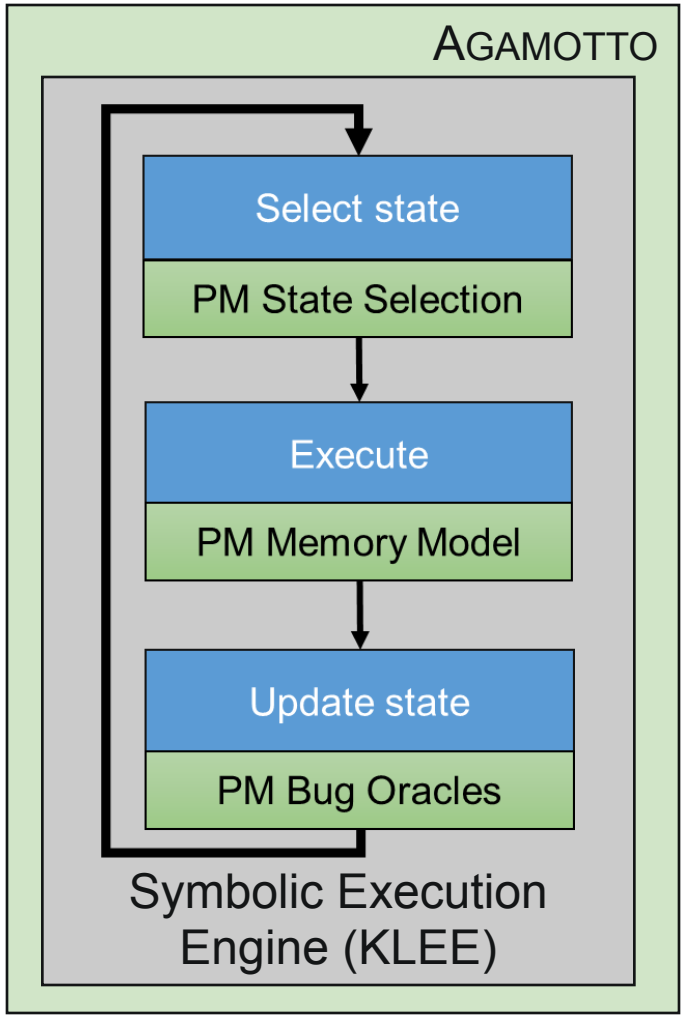


(2) PM Bug Detection

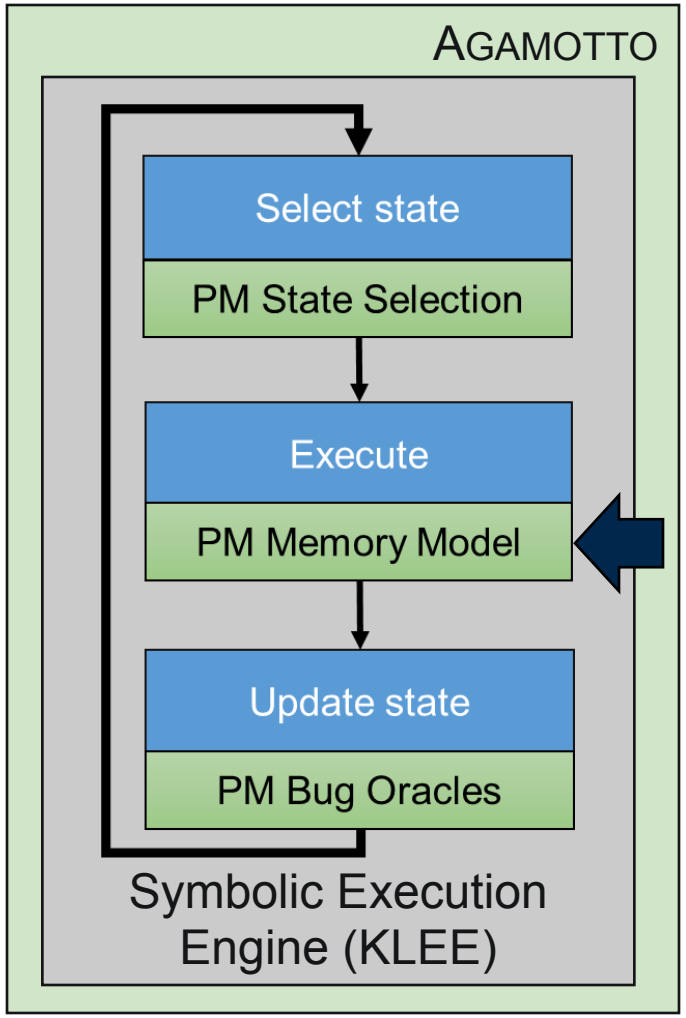
- Custom oracles = custom state & signals
  - Example: ordering bug (**A** must be durable **before B**)



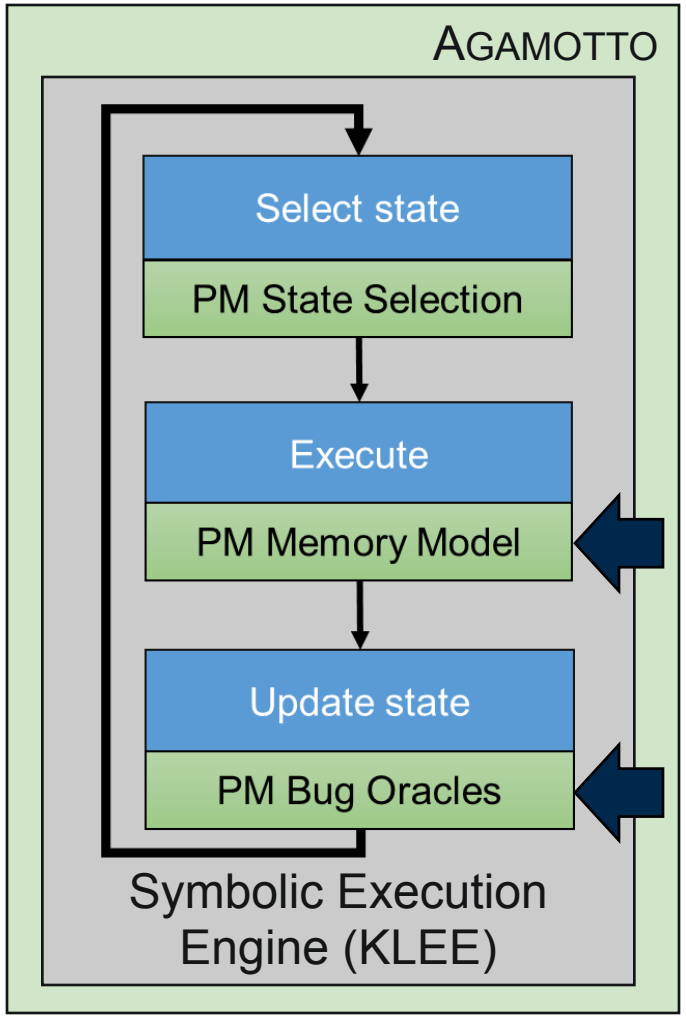
What else?


AGAMOTTO
Select state
PM State Selection
Execute
PM Memory Model
Update state
PM Bug Oracles
Symbolic Execution Engine (KLEE)

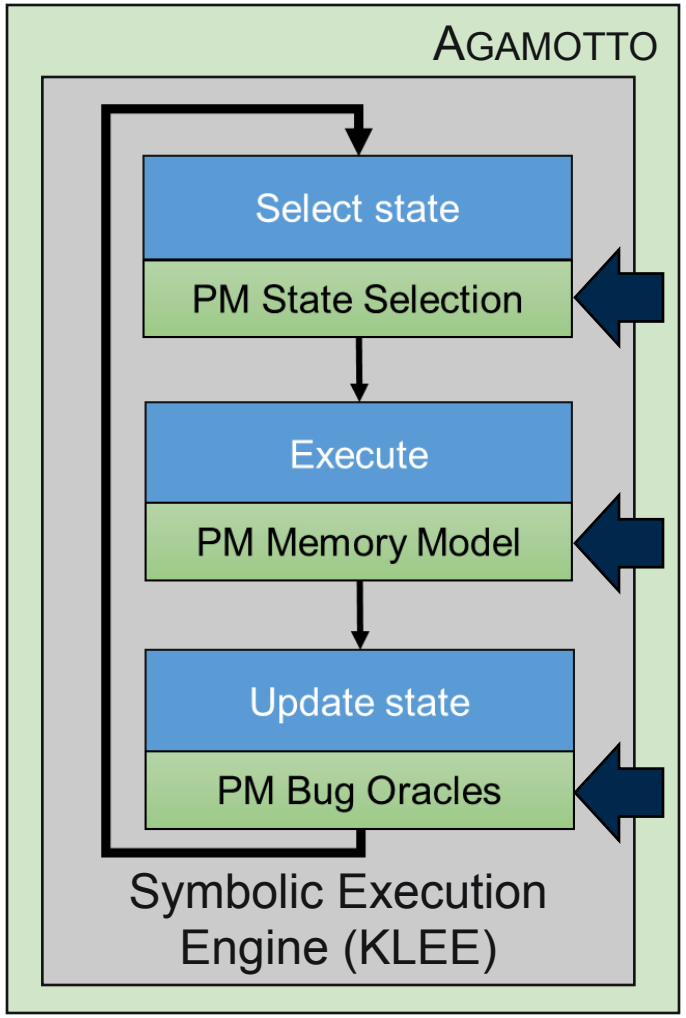
What else?


AGAMOTTO
Select state
PM State Selection
Execute
PM Memory Model
Update state
PM Bug Oracles
Symbolic Execution Engine (KLEE)

What else?


AGAMOTTO
Select state
PM State Selection
Execute
PM Memory Model
Update state
PM Bug Oracles
Symbolic Execution Engine (KLEE)

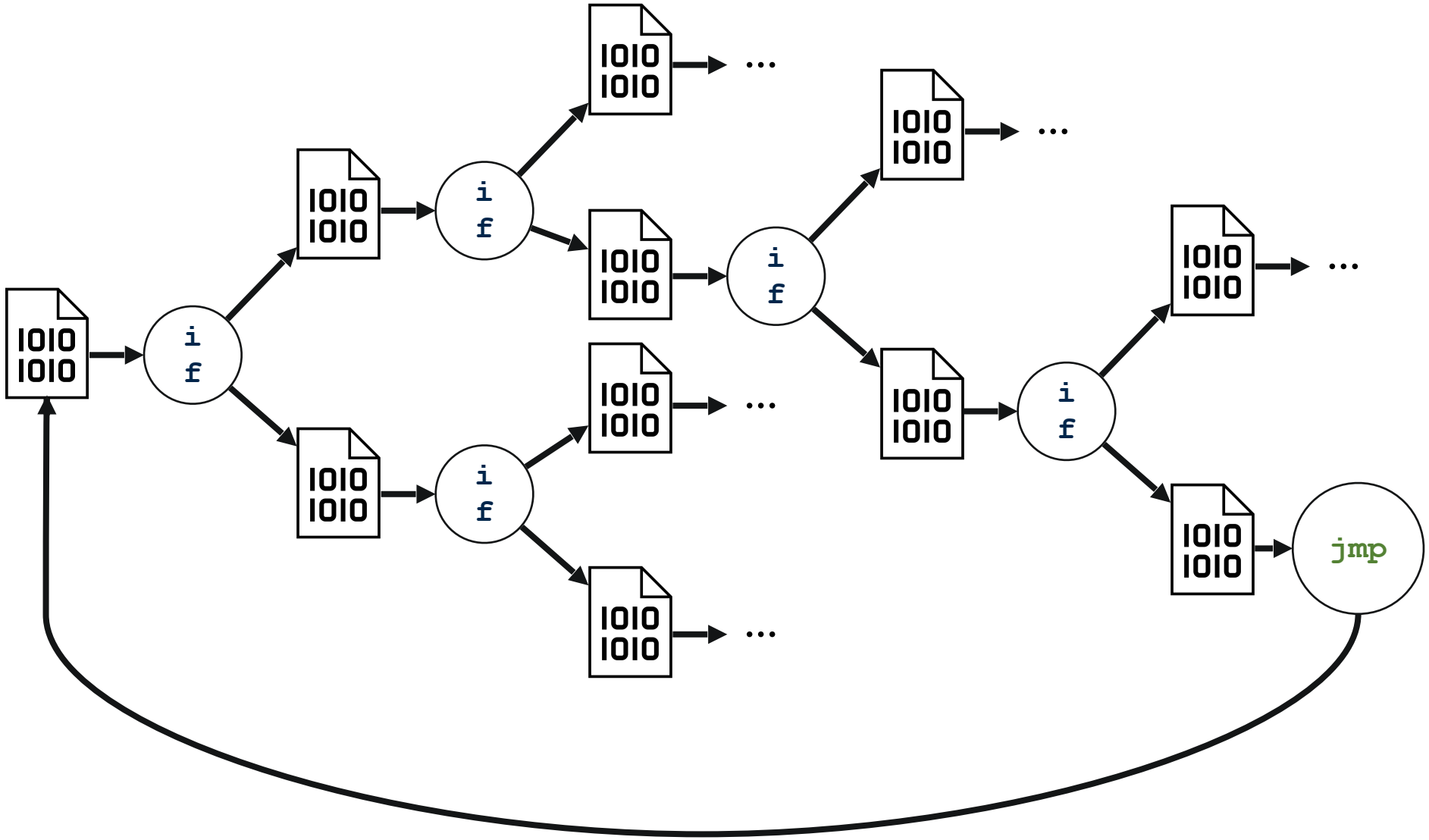
What else?



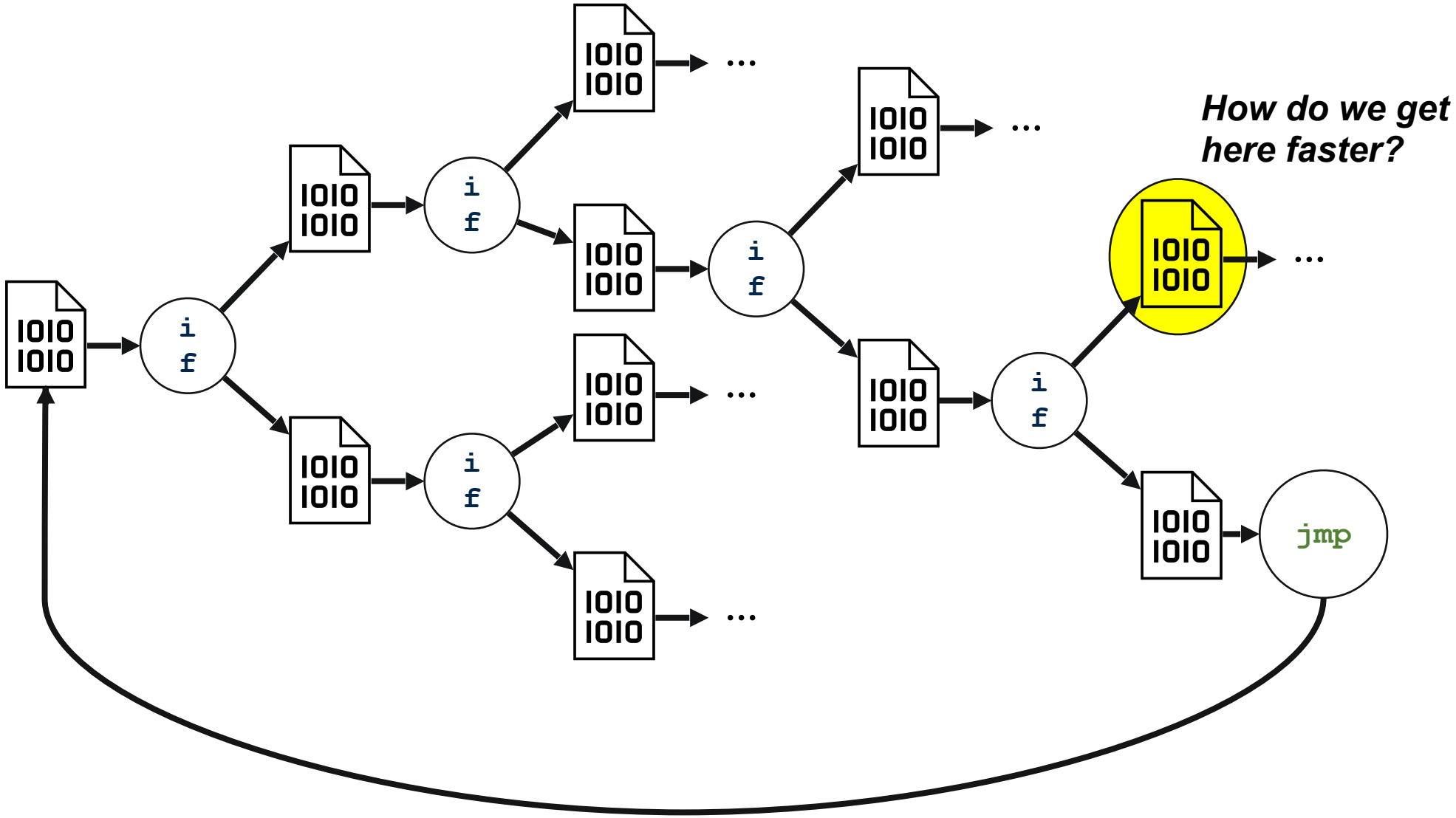
Why? To handle "State Space Explosion"!

Problem: State Space Explosion

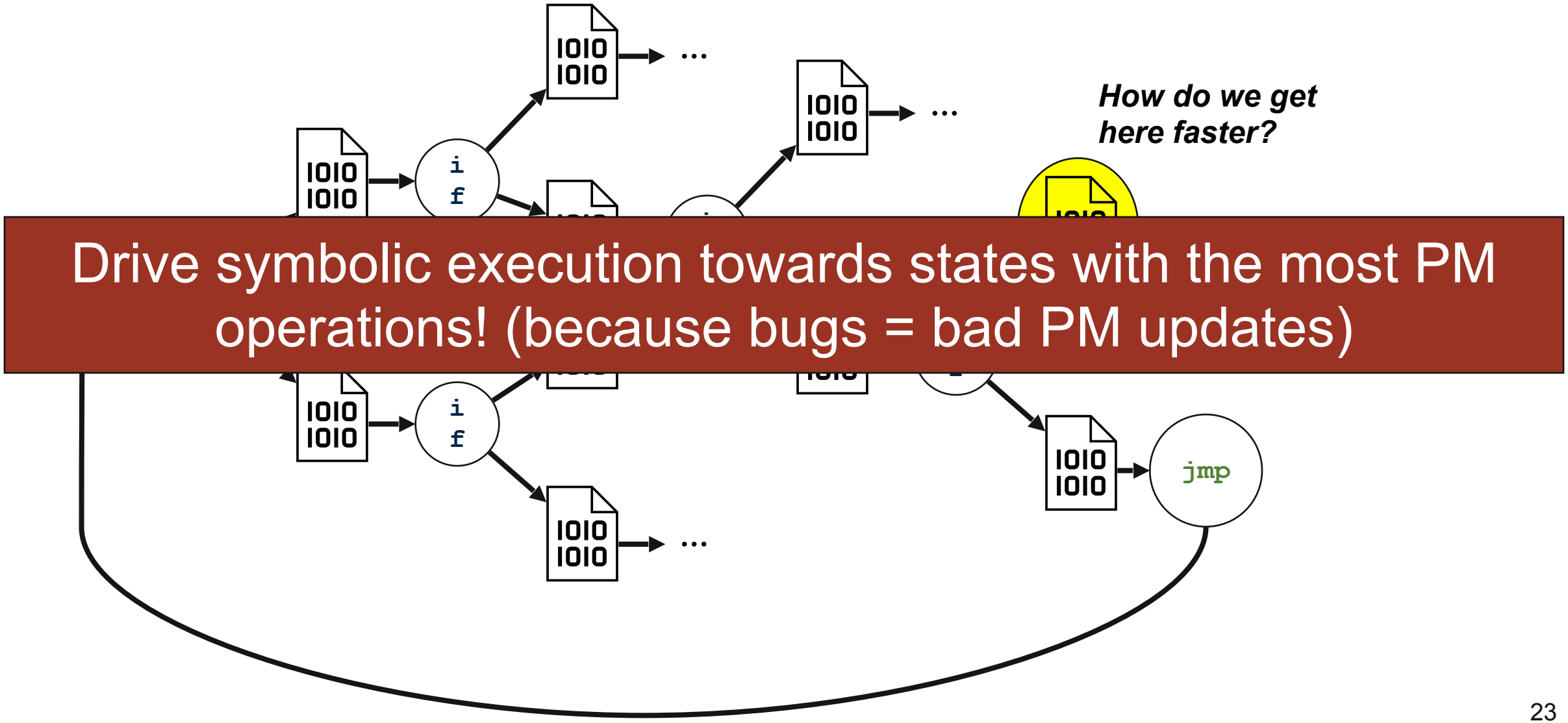
Problem: State Space Explosion



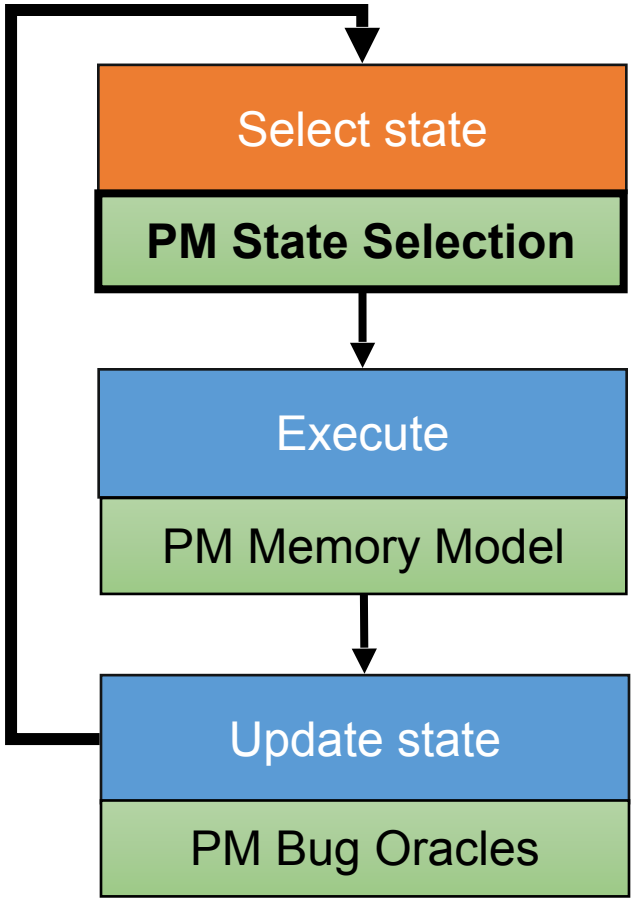
Problem: State Space Explosion



Problem: State Space Explosion



(3) PM State Selection




```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

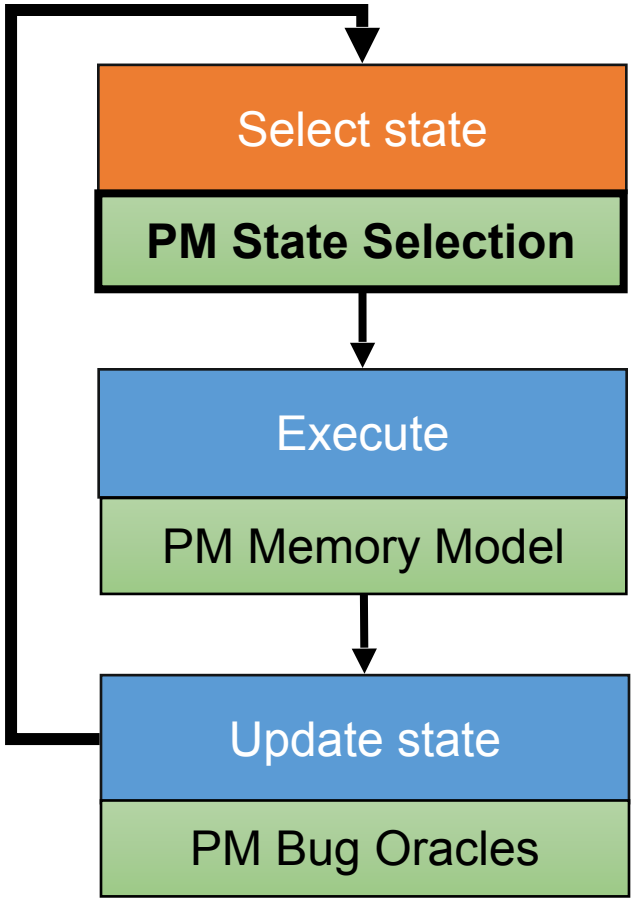
`do_read = λ≠0`

`pbuf[x]: clean`

`do_read = 0`

`pbuf[x]: dirty`

(3) PM State Selection




```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

do_read = λ≠0

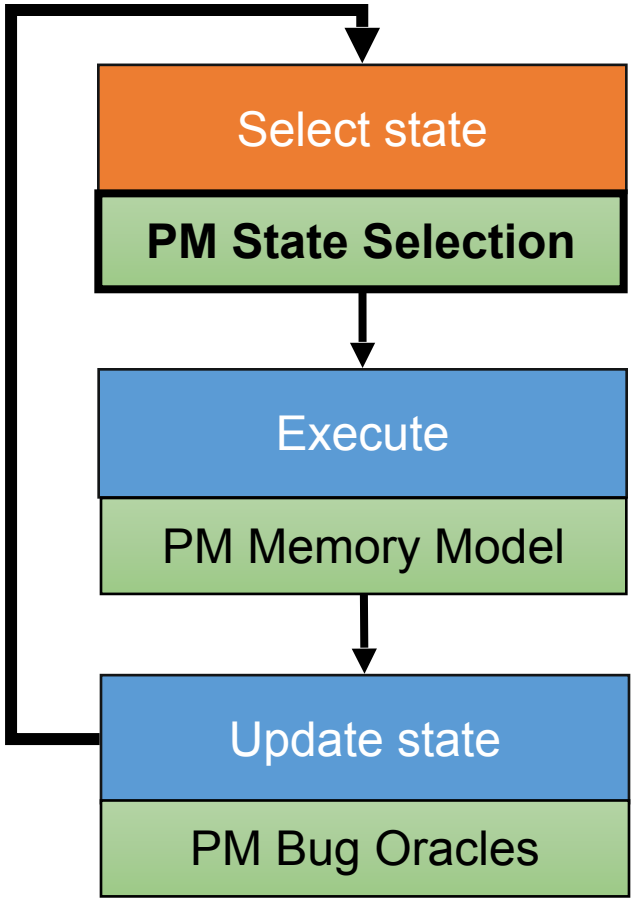
pbuf[x]: clean

do_read = 0

pbuf[x]: dirty

1. *Identify PM instructions (static analysis)*

(3) PM State Selection



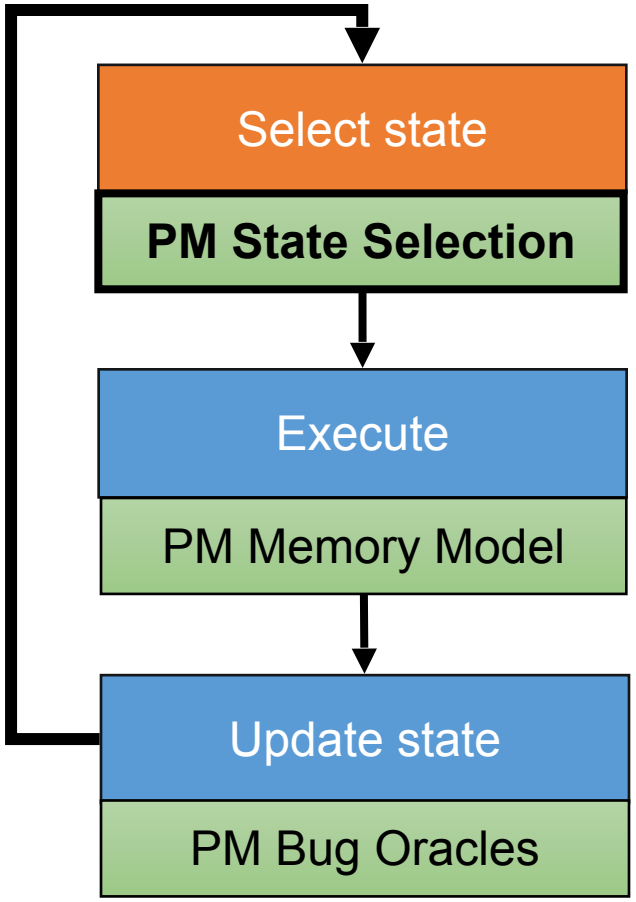

```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5        a = pbuf[x]
6        foo ()
7  else
8        a = ...
9        pbuf[x] = a
10       clwb(pbuf[x])
11       // BUG: Missing sfence!
12 exit(0)
```

do_read = λ≠0 — pbuf[x]: clean

do_read = 0 — pbuf[x]: dirty

1. *Identify PM instructions (static analysis)*

(3) PM State Selection



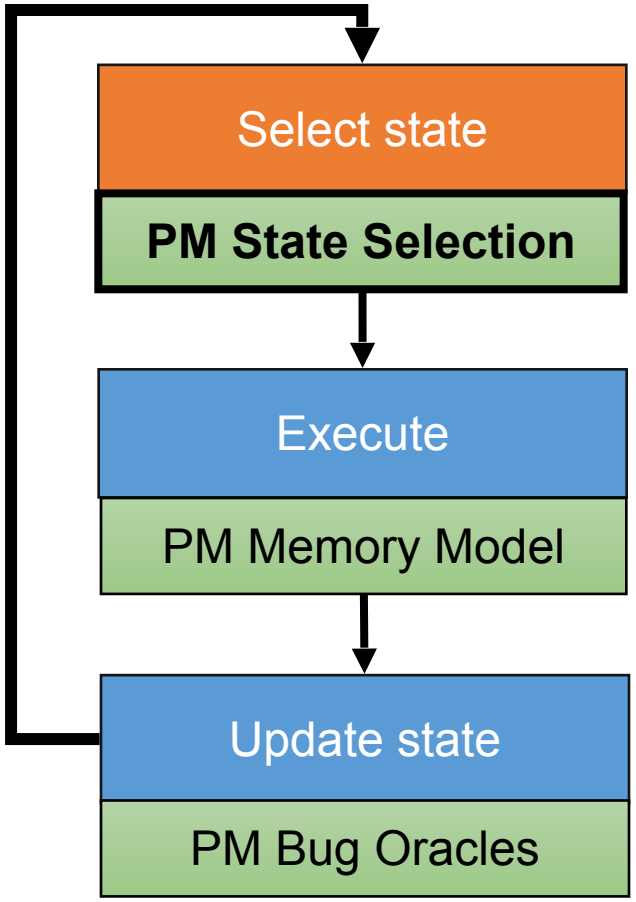

```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

`do_read = λ≠0` → `pbuf[x]: clean`

`do_read = 0` → `pbuf[x]: dirty`

1. *Identify PM instructions (static analysis)*
2. *Assign unit weight*

(3) PM State Selection



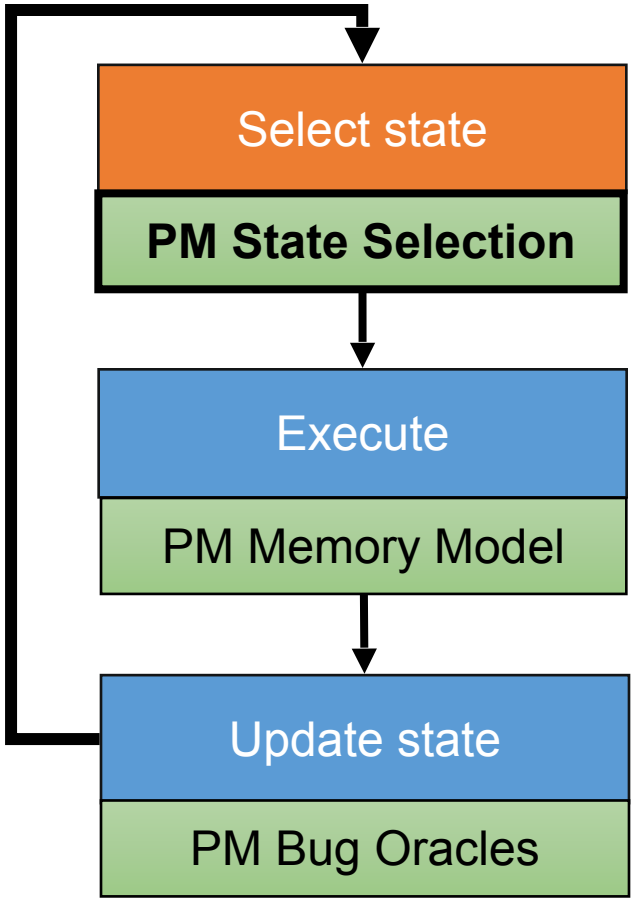

```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5        a = pbuf[x]
6        foo ()
7  else
8        a = ...
9        pbuf[x] = a
10       clwb(pbuf[x])
11       // BUG: Missing sfence!
12 exit(0)
```

`do_read = λ≠0` — `pbuf[x]: clean`

`do_read = 0` — `pbuf[x]: dirty`

1. *Identify PM instructions (static analysis)*
2. *Assign unit weight*
3. *Back-propagate priorities*

(3) PM State Selection




```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

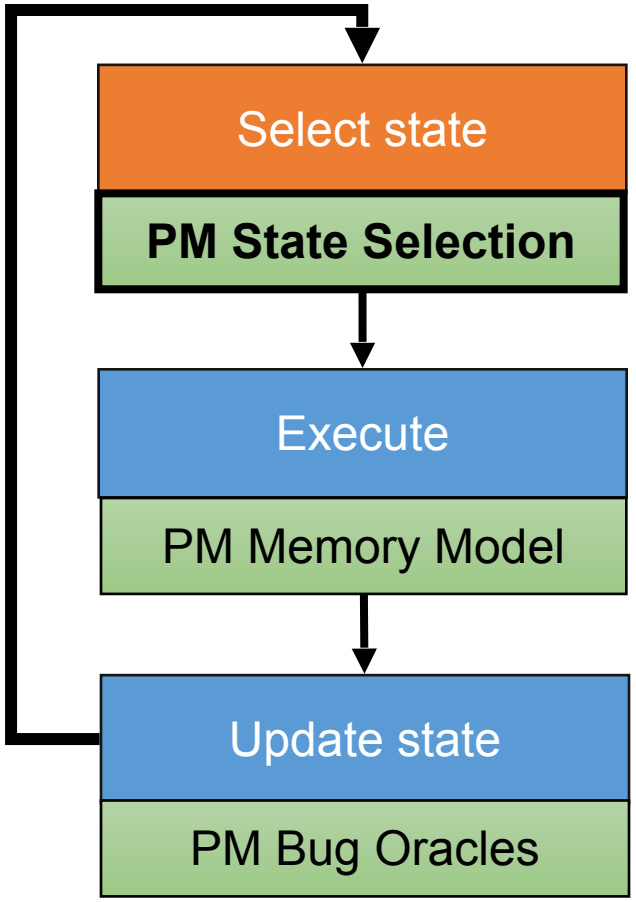
`do_read = λ≠0` — `pbuf[x]: clean`

`do_read = 0` — `pbuf[x]: dirty`

Priority: 1

1. *Identify PM instructions (static analysis)*
2. *Assign unit weight*
3. *Back-propagate priorities*

(3) PM State Selection




```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5       a = pbuf[x]
6       foo ()
7  else
8       a = ...
9       pbuf[x] = a
10      clwb(pbuf[x])
11      // BUG: Missing sfence!
12 exit(0)
```

`do_read = λ≠0` — `pbuf[x]: clean`

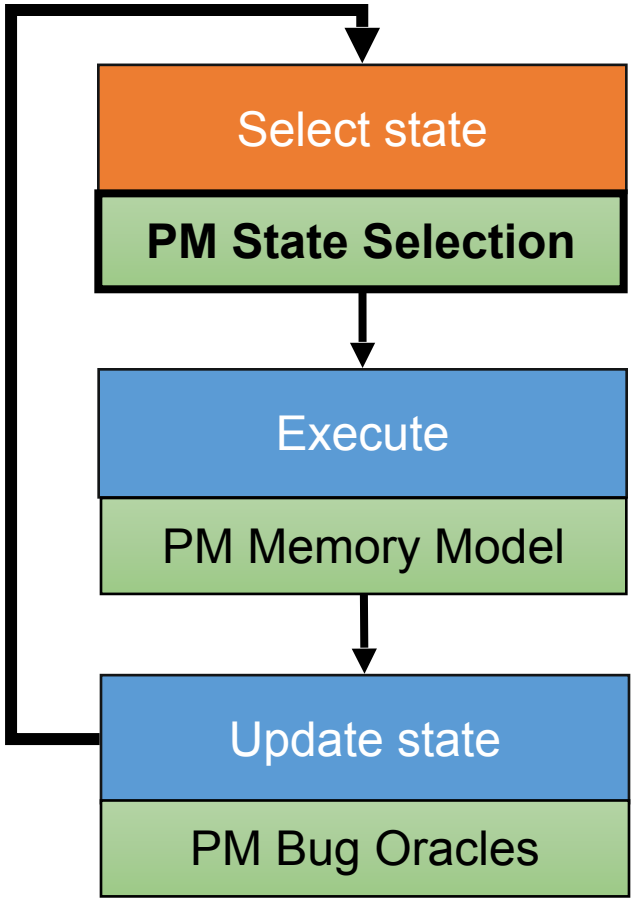
`do_read = 0` — `pbuf[x]: dirty`

Priority: 2

Priority: 1

1. *Identify PM instructions (static analysis)*
2. *Assign unit weight*
3. *Back-propagate priorities*

(3) PM State Selection




```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5        a = pbuf[x]
6        foo ()
7  else
8        a = ...
9        pbuf[x] = a
10       clwb(pbuf[x])
11       // BUG: Missing sfence!
12 exit(0)
```

do_read = λ≠0 — pbuf[x]: clean

do_read = 0 — pbuf[x]: dirty

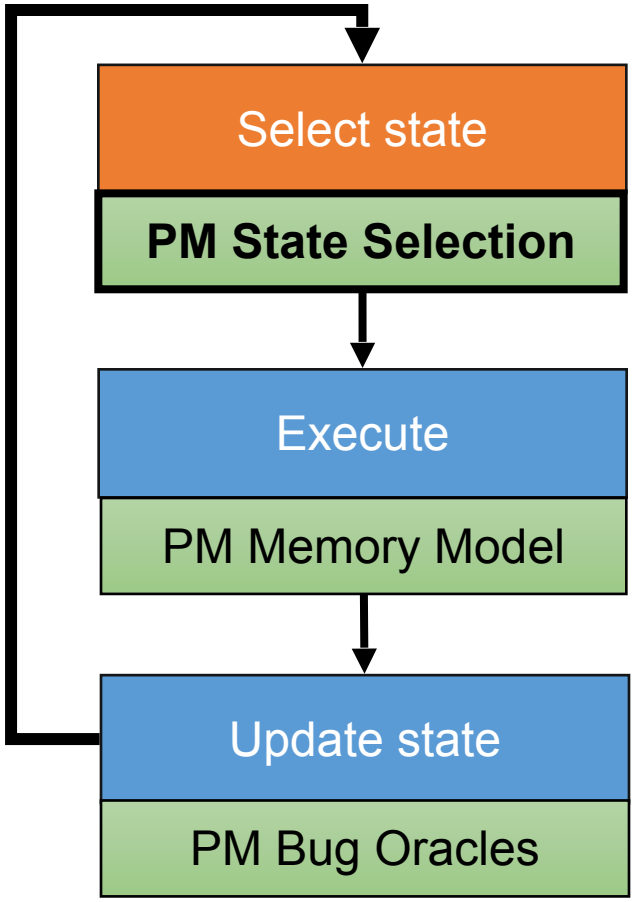
Priority: 2

Priority: 2

Priority: 1

1. *Identify PM instructions (static analysis)*
2. *Assign unit weight*
3. *Back-propagate priorities*

(3) PM State Selection




```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5        a = pbuf[x]
6        foo ()
7  else
8        a = ...
9        pbuf[x] = a
10       clwb(pbuf[x])
11       // BUG: Missing sfence!
12 exit(0)
```

do_read = λ≠0 — pbuf[x]: clean

do_read = 0 — pbuf[x]: dirty

Priority: 0

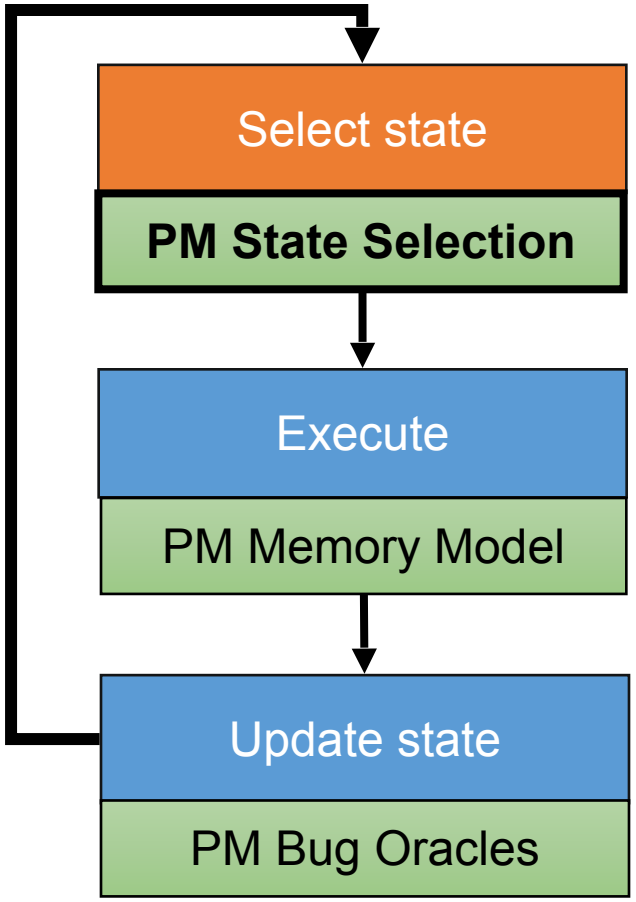
Priority: 2

Priority: 2

Priority: 1

1. *Identify PM instructions (static analysis)*
2. *Assign unit weight*
3. *Back-propagate priorities*

(3) PM State Selection




```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
5        a = pbuf[x]
6        foo ()
7  else
8        a = ...
9        pbuf[x] = a
10       clwb(pbuf[x])
11       // BUG: Missing sfence!
12 exit(0)
```

do_read = λ≠0 — pbuf[x]: clean

do_read = 0 — pbuf[x]: dirty

Priority: 0 (line 5)

Priority: 0 (line 6)

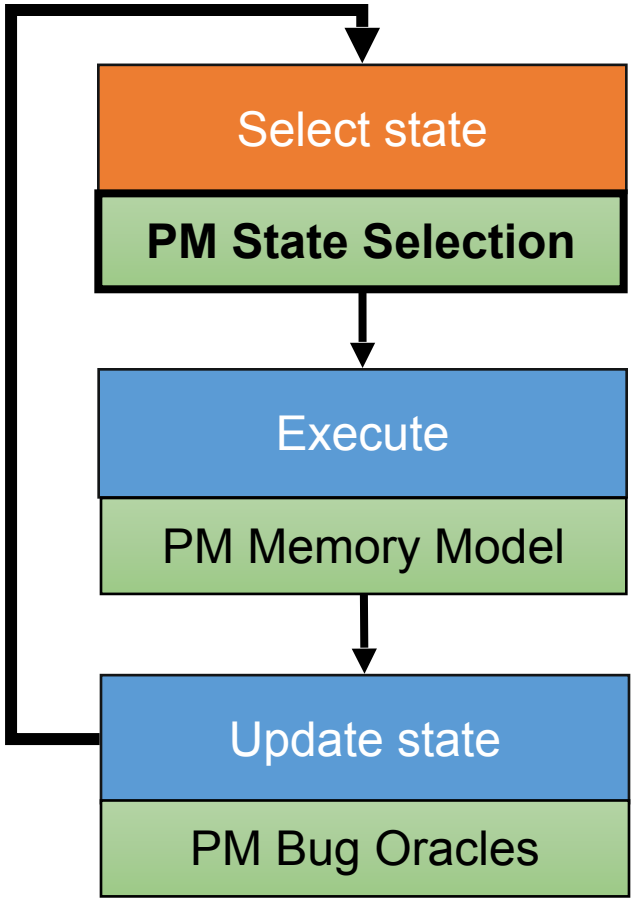
Priority: 2 (line 8)

Priority: 2 (line 9)

Priority: 1 (line 10)

1. *Identify PM instructions (static analysis)*
2. *Assign unit weight*
3. *Back-propagate priorities*

(3) PM State Selection



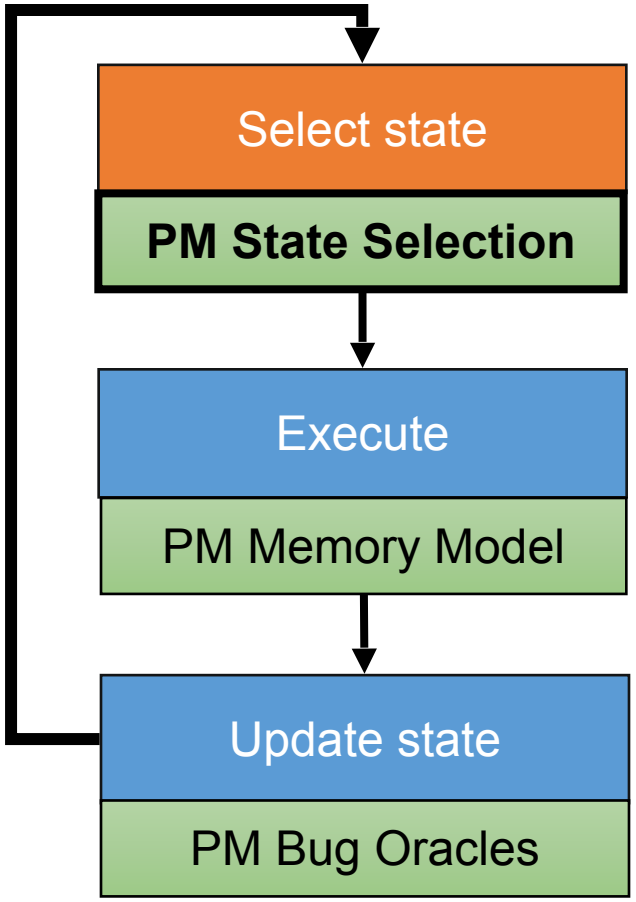

```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)                          Priority: 2
5      a = pbuf[x]                       Priority: 0
6      foo ()                            Priority: 0
7  else
8      a = ...                           Priority: 2
9      pbuf[x] = a                       Priority: 2
10     clwb(pbuf[x])                     Priority: 1
11     // BUG: Missing sfence!
12 exit(0)
```

do_read = λ≠0 — pbuf[x]: clean

do_read = 0 — pbuf[x]: dirty

1. *Identify PM instructions (static analysis)*
2. *Assign unit weight*
3. *Back-propagate priorities*

(3) PM State Selection



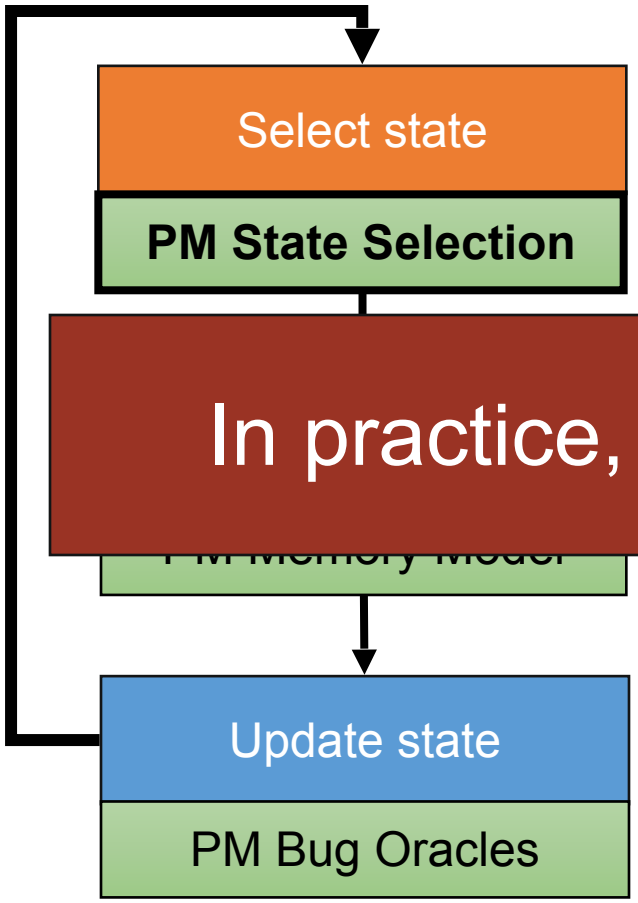

```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...                          Priority: 2
4  if (do_read)                           Priority: 2
5        a = pbuf[x]                      Priority: 0
6        foo ()                           Priority: 0
7  else
8        a = ...                          Priority: 2
9        pbuf[x] = a                      Priority: 2
10       clwb(pbuf[x])                    Priority: 1
11       // BUG: Missing sfence!
12 exit(0)
```

do_read = λ≠0 — pbuf[x]: clean

do_read = 0 — pbuf[x]: dirty

1. *Identify PM instructions (static analysis)*
2. *Assign unit weight*
3. *Back-propagate priorities*

(3) PM State Selection




```
1  char *pbuf = mmap(<PM file>);
2  ...
3  do_read = ...
4  if (do_read)
```

Priority: 2

Priority: 2

In practice, ***<10%*** of instructions have non-zero priority!

```
do_read = 0
```

pbuf[x]: dirty

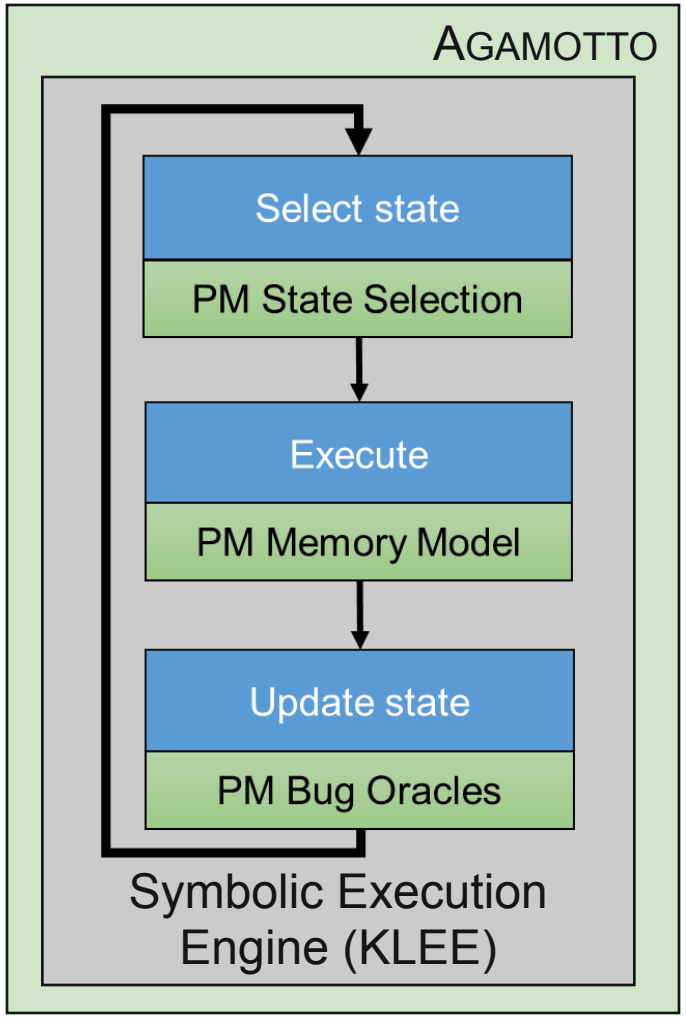
```
9      pbuf[x] = a
10     clwb(pbuf[x])
11     // BUG: Missing sfence!
12 exit(0)
```

Priority: 2

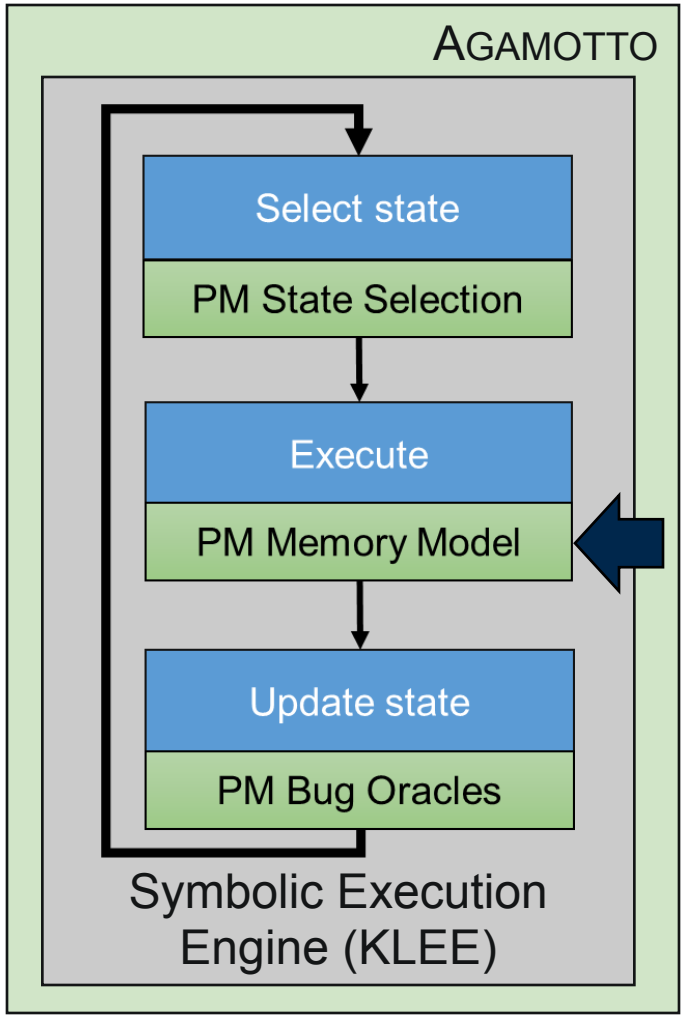
Priority: 1

1. *Identify PM instructions (static analysis)*
2. *Assign unit weight*
3. *Back-propagate priorities*

Design Summary

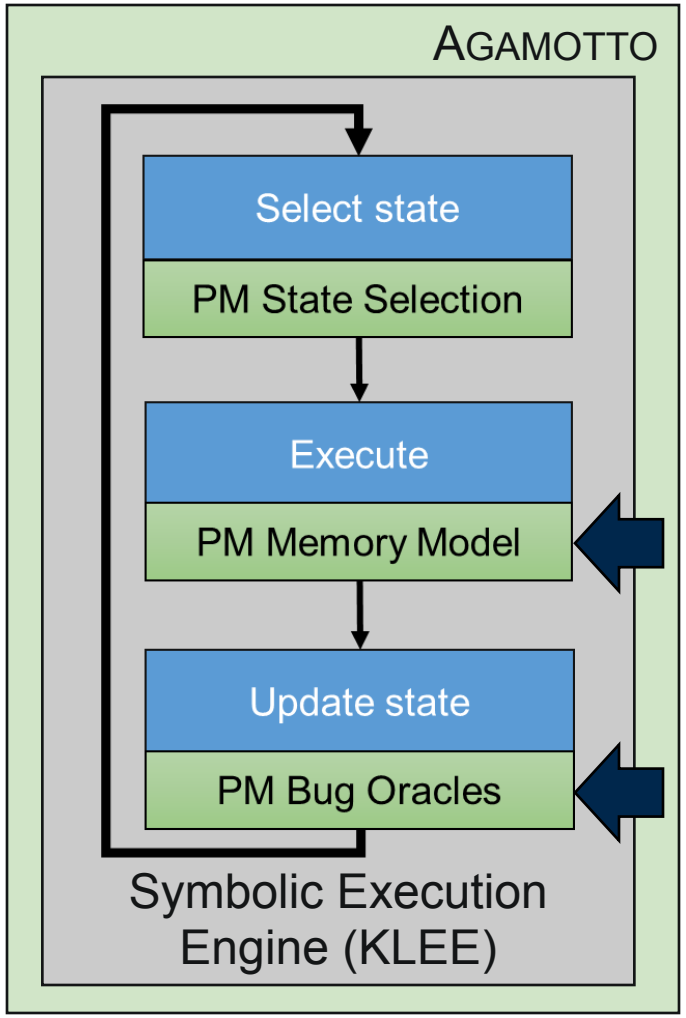


Design Summary



1. Create a model for PM and add to symbolic states

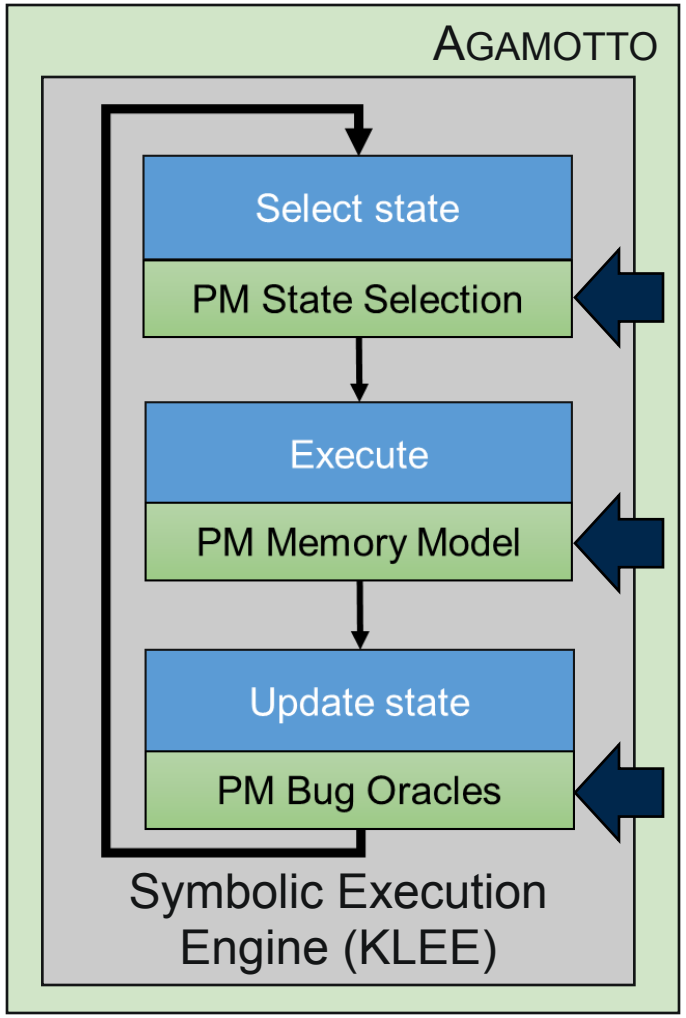
Design Summary



1. Create a model for PM and add to symbolic states

2. Add rules to detect application-independent bug signals

Design Summary

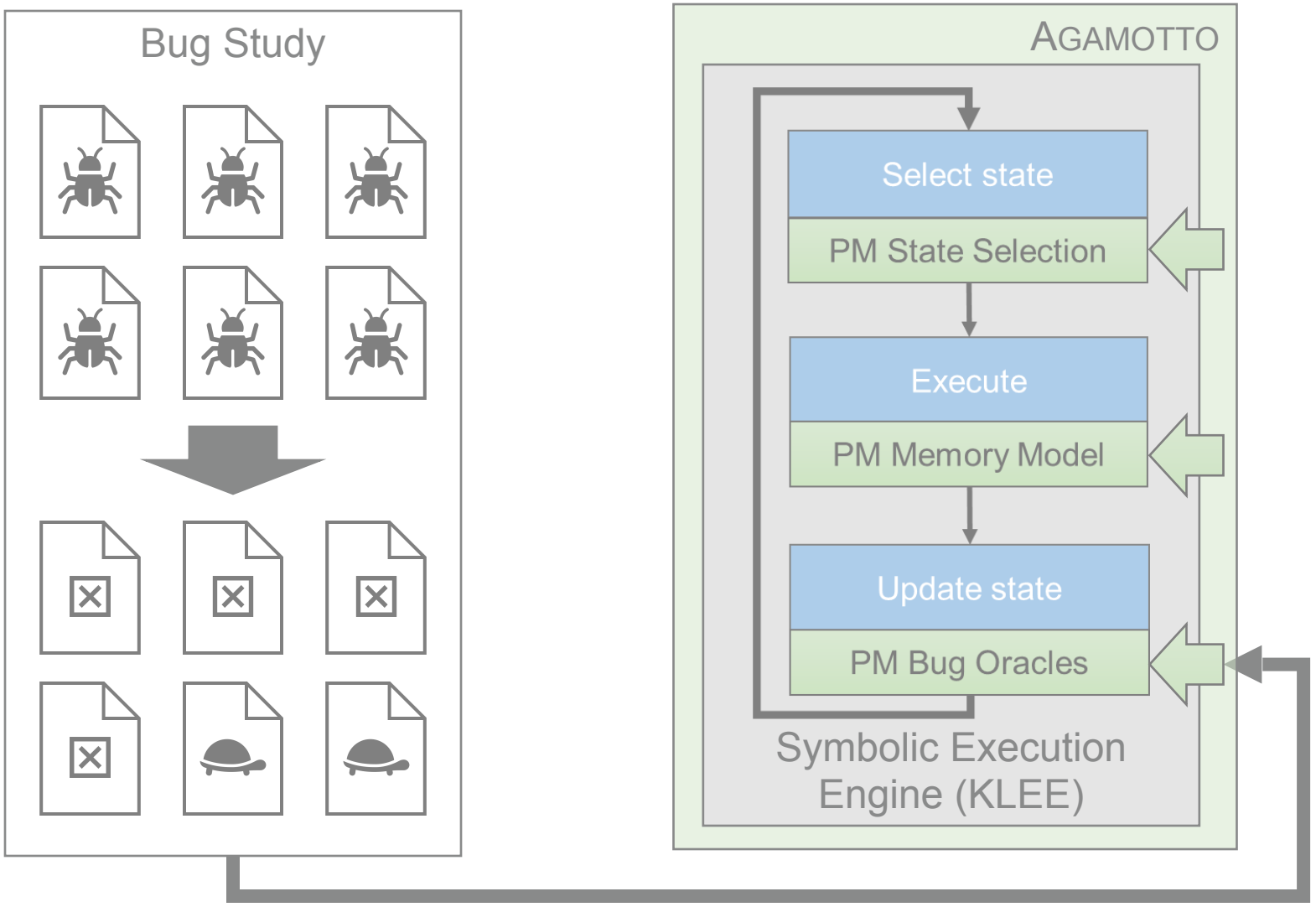


3. Select states based on PM priority to maximize chances of finding bugs

1. Create a model for PM and add to symbolic states

2. Add rules to detect application-independent bug signals

Overview



Results

× 84 found

Reported to developers

Showed impact of performance bugs
(+47% throughput)

Effectiveness

Effectiveness

- **84** new bugs found so far (across 5 systems)
  - 14 correctness bugs, 70 performance bugs
  - PMTest, XFDetector found 3, 4 new bugs

Effectiveness

- **84** new bugs found so far (across 5 systems)
  - 14 correctness bugs, 70 performance bugs
  - PMTest, XFDetector found 3, 4 new bugs

- PM Search Strategy:
  - KLEE: after 1 hour, found **30%** (25/84) bugs
  - AGAMOTTO: finds ***all*** bugs in <40 minutes

Effectiveness

- **84** new bugs found so far (across 5 systems)
  - 14 correctness bugs, 70 performance bugs
  - PMTest, XFDetector found 3, 4 new bugs

- PM Search Strategy:
  - KLEE: after 1 hour, found **30%** (25/84) bugs
  - AGAMOTTO: finds ***all*** bugs in <40 minutes

- Performance study on RECIPE (SOSP ‘19):
  - **23–47%** overall throughput increase in key-value store workloads!

Helpfulness: Developer Testimonials

Helpfulness: Developer Testimonials

- PMDK developers confirmed our findings about performance issues.

Helpfulness: Developer Testimonials

- PMDK developers confirmed our findings about performance issues.

- Oracle's developers confirmed they were aware of some of the issues we reported.
  - “Resources for software development are always in short supply…your email may be the push that gets us to do something about it. Thank you.”

Helpfulness: Developer Testimonials

- PMDK developers confirmed our findings about performance issues.

- Oracle's developers confirmed they were aware of some of the issues we reported.
  - “Resources for software development are always in short supply…your email may be the push that gets us to do something about it. Thank you.”

- RECIPE’s authors confirmed and started patching all the bugs we reported.
  - “These are some really good finds, since it was difficult to debug our own code without having a proper tool.”

Conclusion

Conclusion

- Performed a study of bugs and found **2 application-independent bug patterns**

Conclusion

- Performed a study of bugs and found **2 application-independent bug patterns**

- Built AGAMOTTO, a symbolic-execution tool which **automatically** (no source code modifications) and **accurately** (low false negatives) finds PM bugs

Conclusion

- Performed a study of bugs and found **2 application-independent bug patterns**

- Built AGAMOTTO, a symbolic-execution tool which **automatically** (no source code modifications) and **accurately** (low false negatives) finds PM bugs

- We find **84 new PM bugs** and show the potential impact of performance bugs, increasing throughput by **23–47%**

Conclusion

- Performed a study of bugs and found **2 application-independent bug patterns**

- Built AGAMOTTO, a symbolic-execution tool which **automatically** (no source code modifications) and **accurately** (low false negatives) finds PM bugs

- We find **84 new PM bugs** and show the potential impact of performance bugs, increasing throughput by **23–47%**

https://github.com/efeslab/agamotto

Email: iangneal@umich.edu

Conclusion

- Performed a study of bugs and found **2 application-independent bug patterns**

- Built AGAMOTTO, a symbolic-execution tool which **automatically** (no source code modifications) and **accurately** (low false negatives) finds PM bugs

- We find **84 new PM bugs** and show the potential impact of performance bugs, increasing throughput by **23–47%**

https://github.com/efeslab/agamotto

Email: iangneal@umich.edu

Trivia: AGAMOTTO's name was inspired by the "Eye of Agamotto," which is another name for the Time Stone in the Avengers franchise.