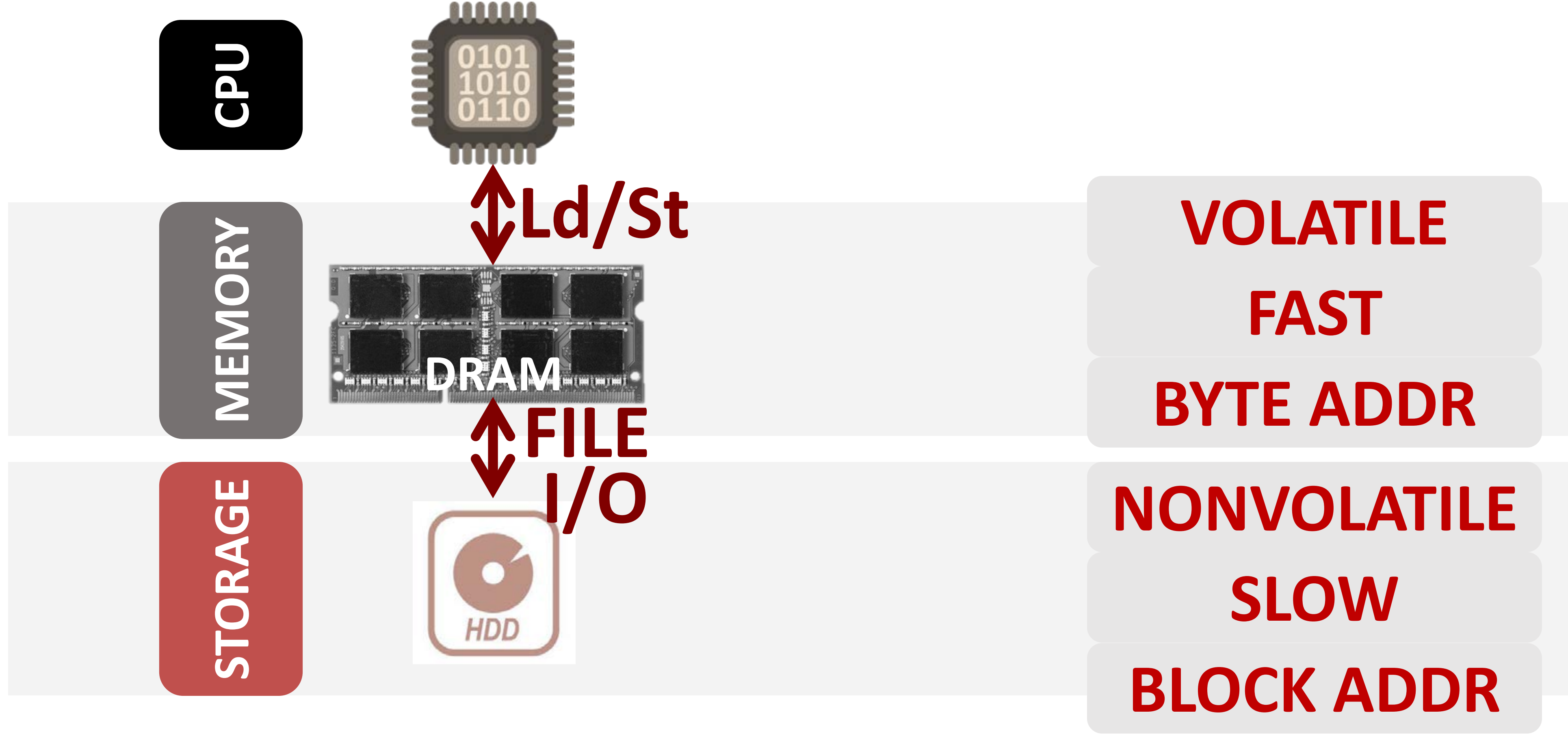
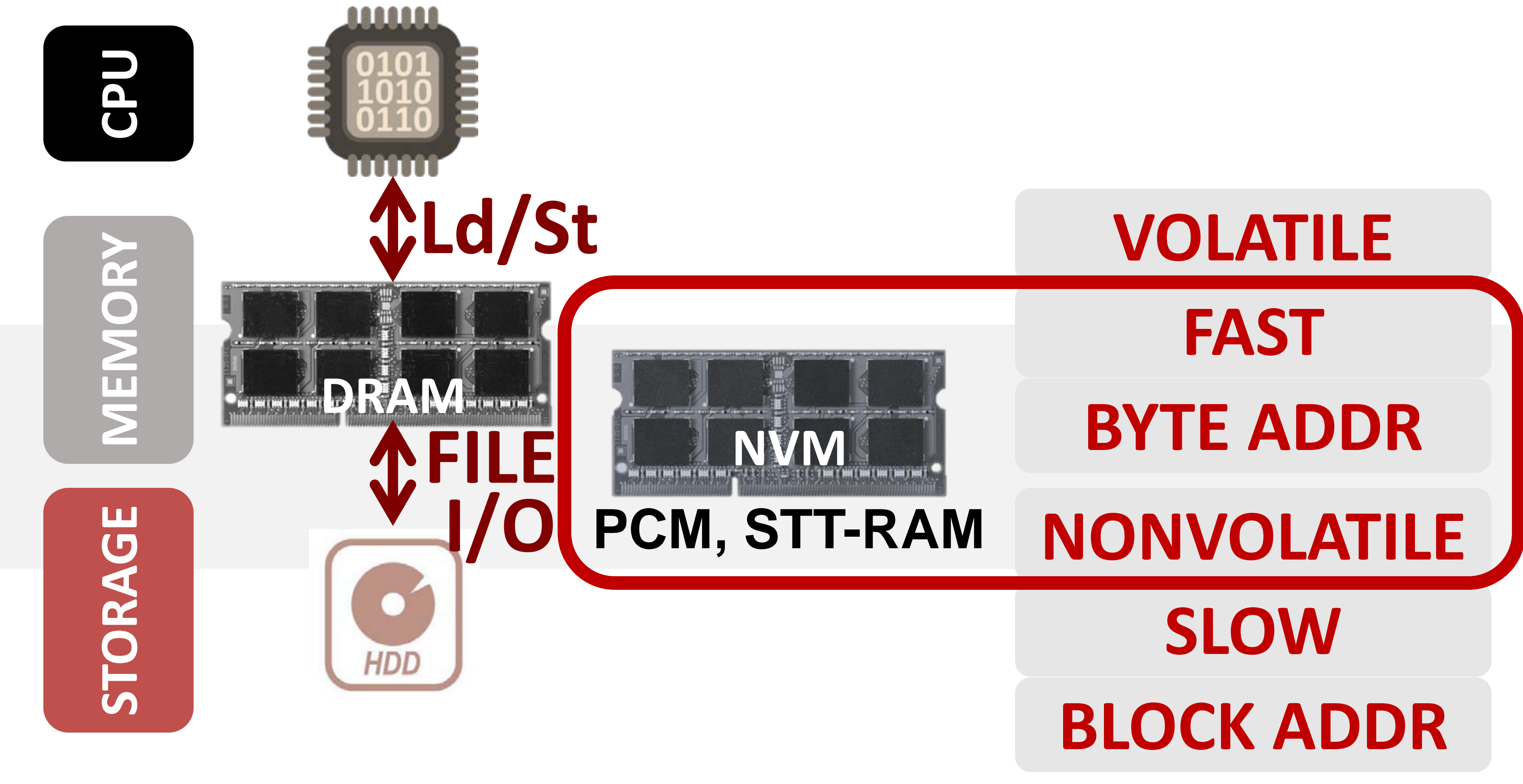
PMTEST
Testing Persistent Memory Applications

Samira Khan

TWO-LEVEL STORAGE MODEL

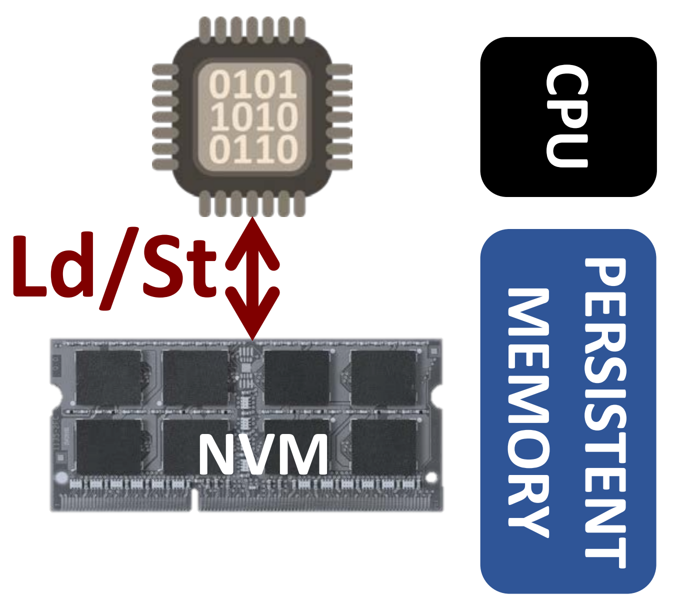


TWO-LEVEL STORAGE MODEL



Non-volatile memories combine characteristics of memory and storage

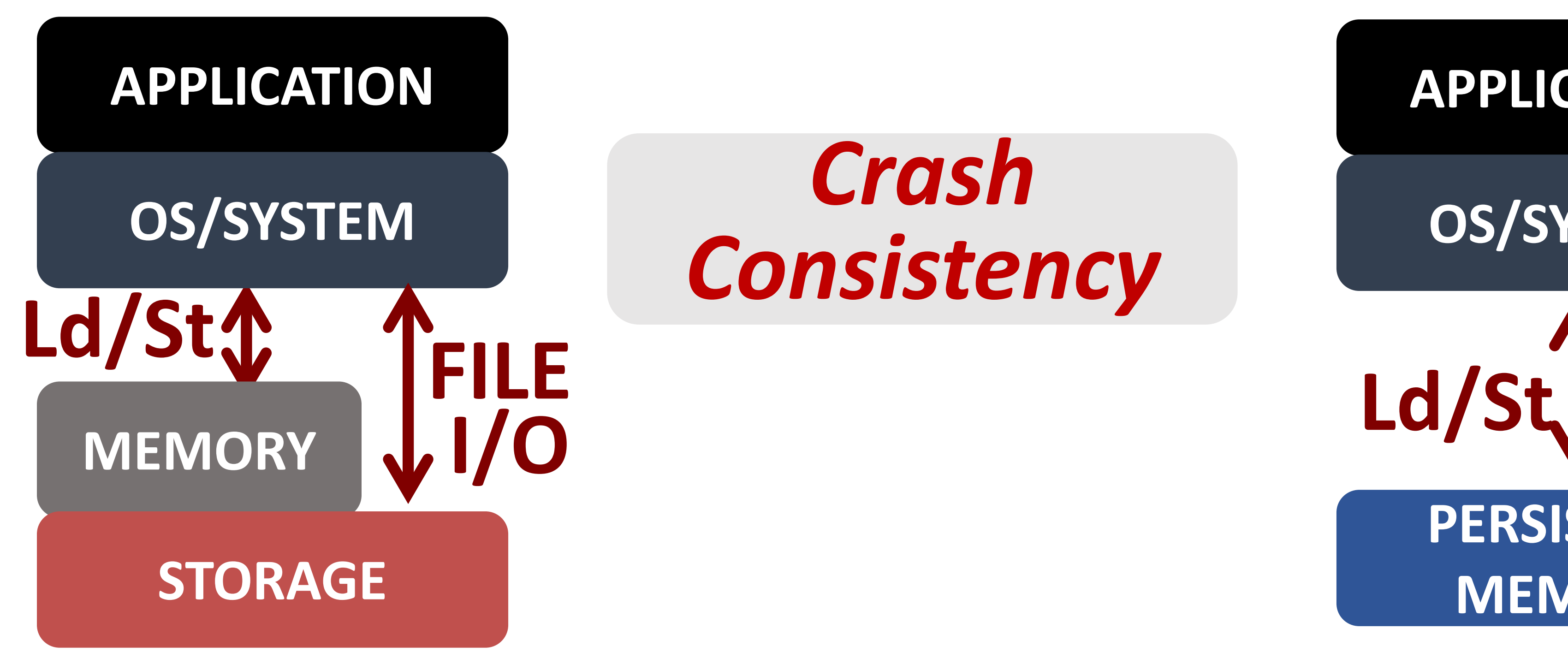
VISION: UNIFY MEMORY AND STORAGE



Provides an opportunity to manipulate persistent data directly in memory

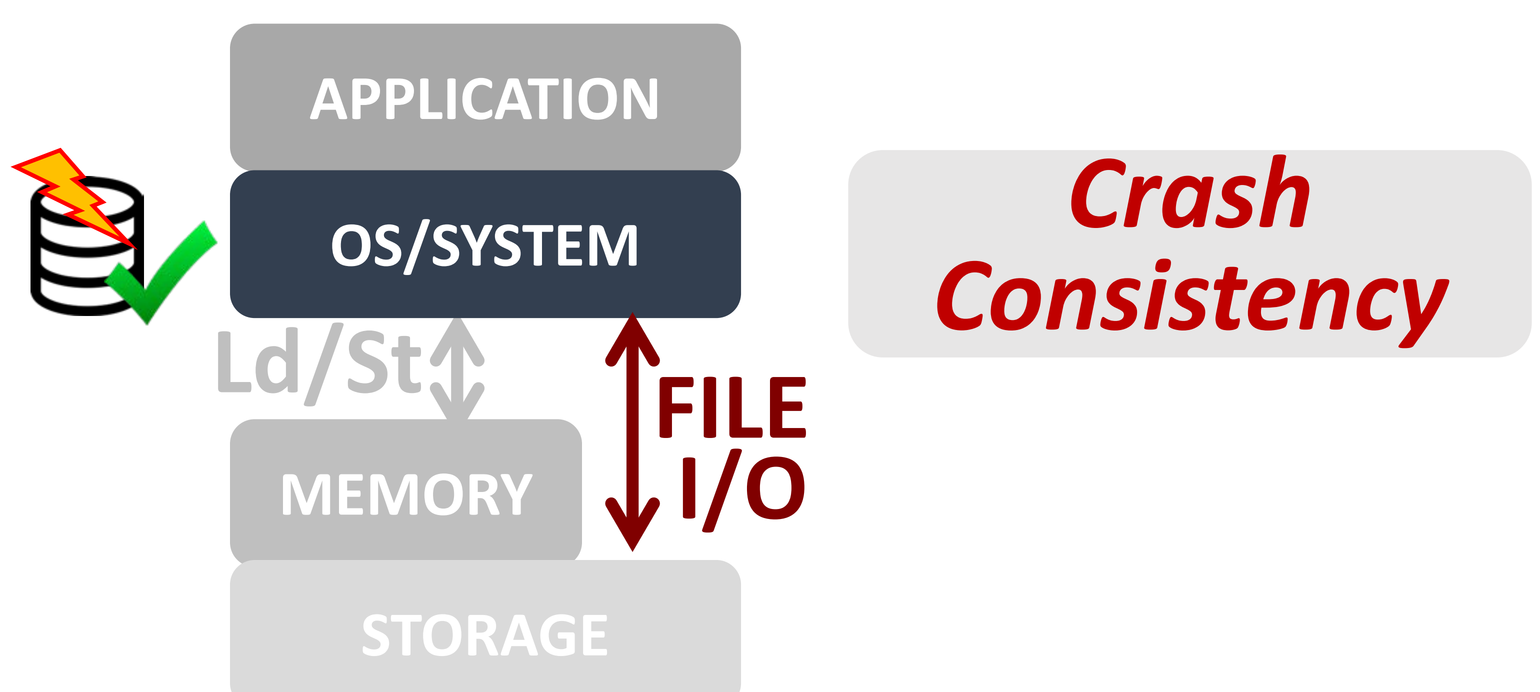
Avoids reading and writing back data to/from storage

CHALLENGE: NEED ALL STORAGE SYSTEM SUPPORTS

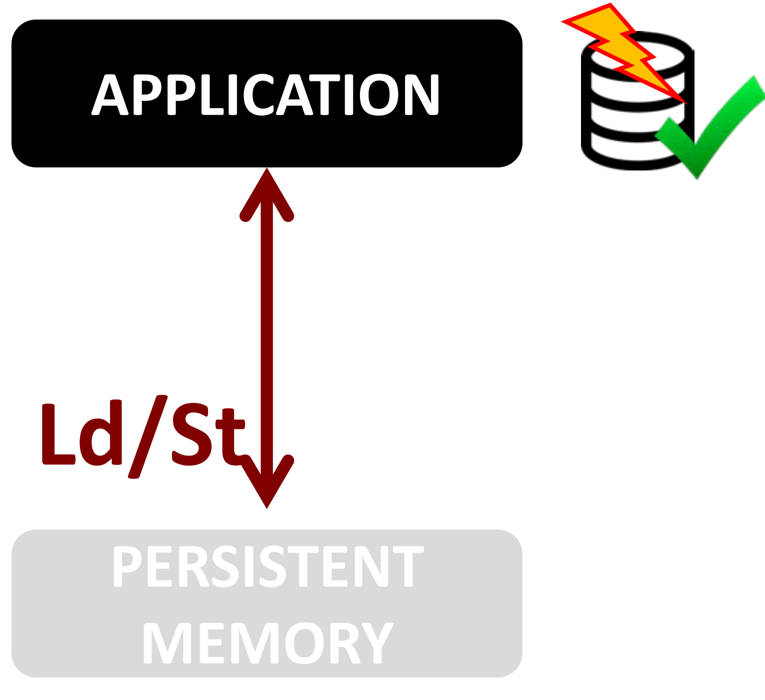


Overhead in OS/storage layer overshadows the benefit of nanosecond access latency of NVM

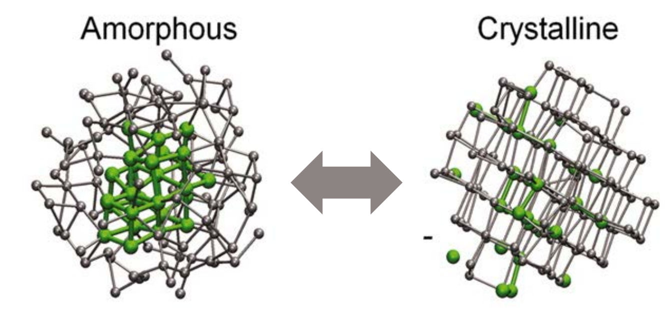
CHALLENGE: NEED ALL STORAGE SYSTEM SUPPORTS



Crash Consistency



Not the operating system,
Application layer is responsible for crash consistency in PM



NON-VOLATILE MEMORY

PERSISTENT MEMORY

Programming Persistent Memory Applications

CHALLENGE:
PM Programming is Hard!

Requirements and Key Ideas

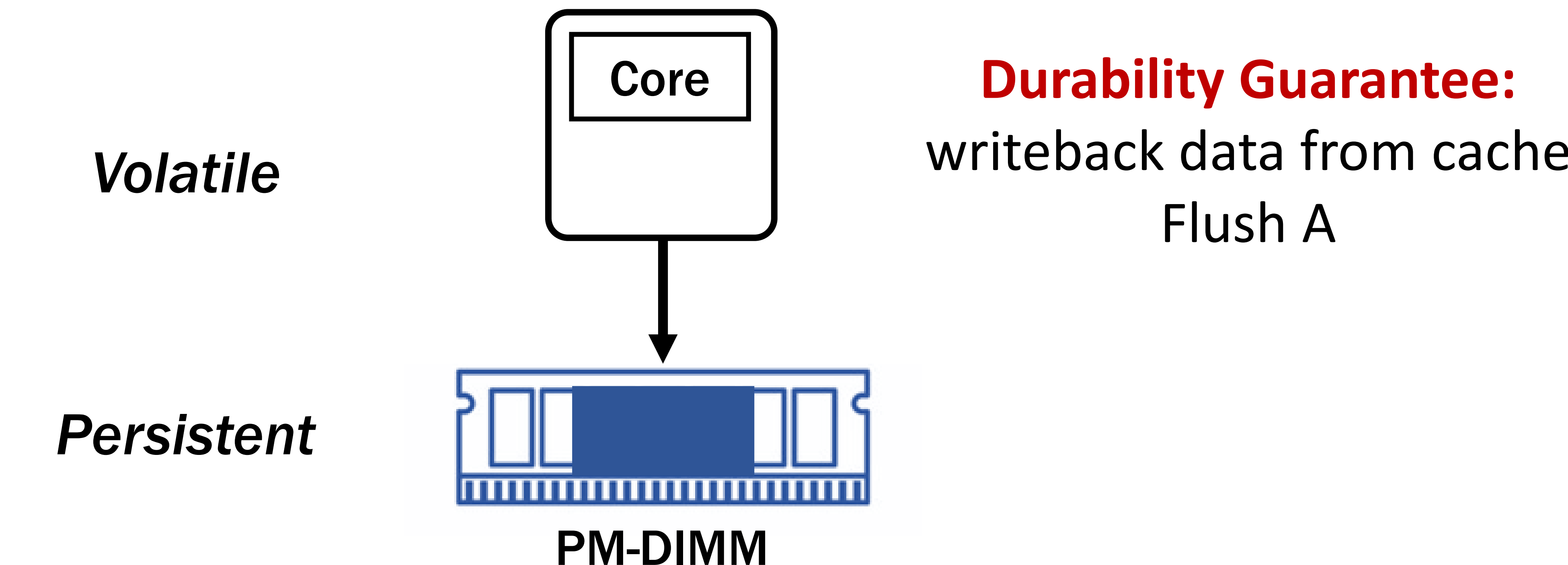
PMTEST: Interface and Mechanism

ASPLOS’19

Results and Conclusion

PERSISTENT MEMORY PROGRAMMING

- Support for crash consistency have two fundamental guarantees
  - **Durability**: writes become persistent in PM
  - **Ordering**: one write becomes persistent in PM before another



PERSISTENT MEMORY PROGRAMMING

- Support for crash consistency have two fundamental guarantees
  - **Durability**: writes become persistent in PM
  - **Ordering**: one write becomes persistent in PM before another

Volatile

Core

Cache

Persistent

B

A

PM-DIMM

Ordering Guarantee:

Write A before B

Writeback A

Barrier

Writeback B

PERSISTENT MEMORY PROGRAMMING

Expert

- Uses low-level primitives
- Understands the hardware
- Understands the algorithm

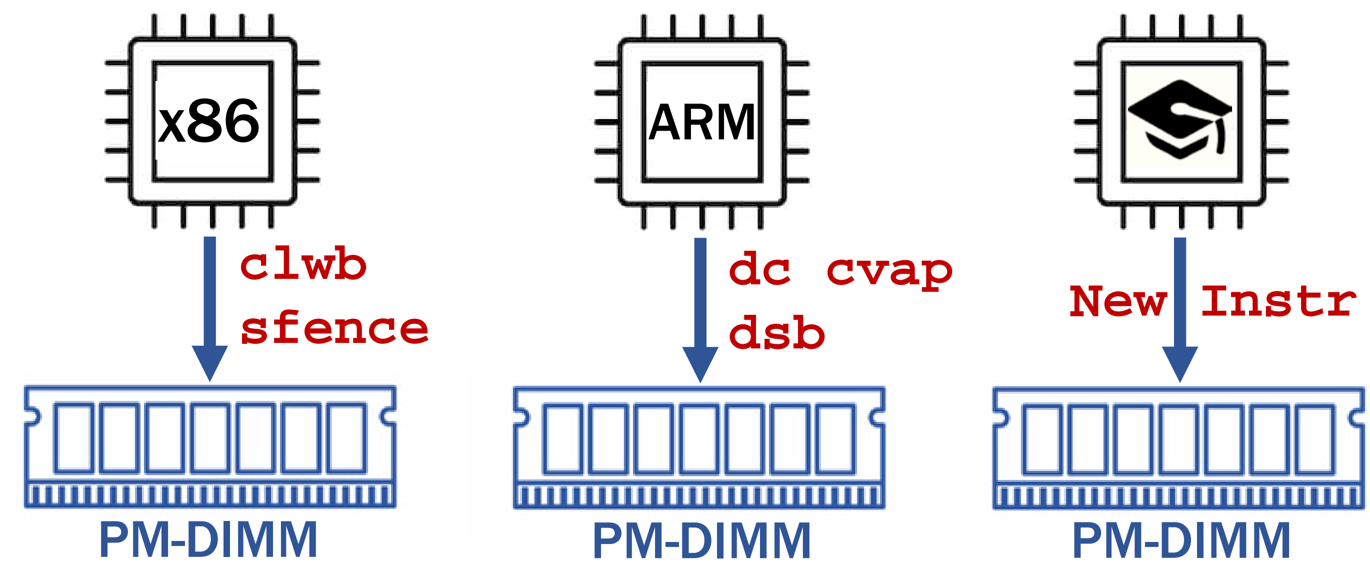
Normal

- Uses a high-level interface
- Does not need to know details of hardware or algorithm

Two different ways to program persistent applications

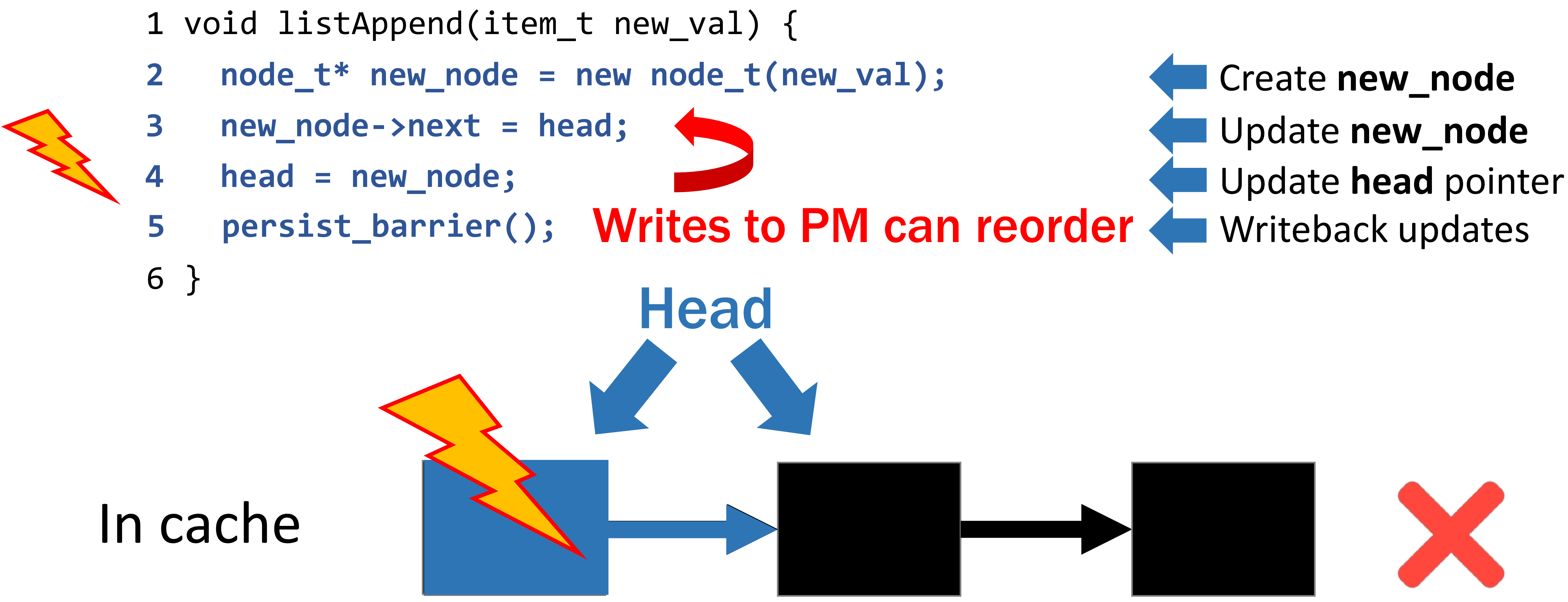
PERSISTENT MEMORY PROGRAMMING (LOW-LEVEL)

- Hardware provides low-level primitives for crash consistency
- Exposes instructions for cache flush and barriers
  - **`sfence`, `clwb`** from x86
  - **`dc cvap`** from ARM
  - Academic proposals, e.g., ofence, dfence.



[Kiln’13, ThyNVM’15, DPO’16, JUSTDOLogging’16, ATOM’17, HOPS’17, etc.]

PROGRAMMING USING LOW-LEVEL PRIMITIVES



PROGRAMMING USING LOW-LEVEL PRIMITIVES

```
1 void listAppend(item_t new_val) {
2   node_t* new_node = new node_t(new_val);
3   new_node->next = head;
4   head = new_node;
4   persist_barrier();
5   persist_barrier();
6 }
```

Enforce writeback before changing **head**

Head

In cache

In PM

Ensuring crash consistency with low-level primitives is HARD!

PERSISTENT MEMORY PROGRAMMING

Expert

- Uses low-level primitives
- Understands the hardware
- Understands the algorithm

Normal

- Uses a high-level interface
- Does not need to know details of hardware or algorithm

PERSISTENT MEMORY PROGRAMMING (HIGH-LEVEL)

- Libraries provide transactions on top of low-level primitives
  - Intel’s PMDK
  - Academic proposals

```
AtomicBegin {
      Append a new node;
} AtomicEnd;
```

Uses logging mechanisms to atomically commit the updates

[NV-Heaps’11, Mnemosyne’11, ATLAS’14, REWIND’15, NVL-C’16, NVThreads’17 LSNVMM’17, etc.]

PROGRAMMING USING TRANSACTIONS

```
1 void ListAppend(item_t new_val) {
2   TX_BEGIN {
3     node_t *new_node = makeNode(new_val);
4     TX_ADD(list.head, sizeof(node_t*));
5     List.head = new_node;
6     List.length++;
7   } TX_END
8 }
```

- Create `new_node`
- backup `head`
- Update `head`
- Update `length`

`length` is not backed up before update!

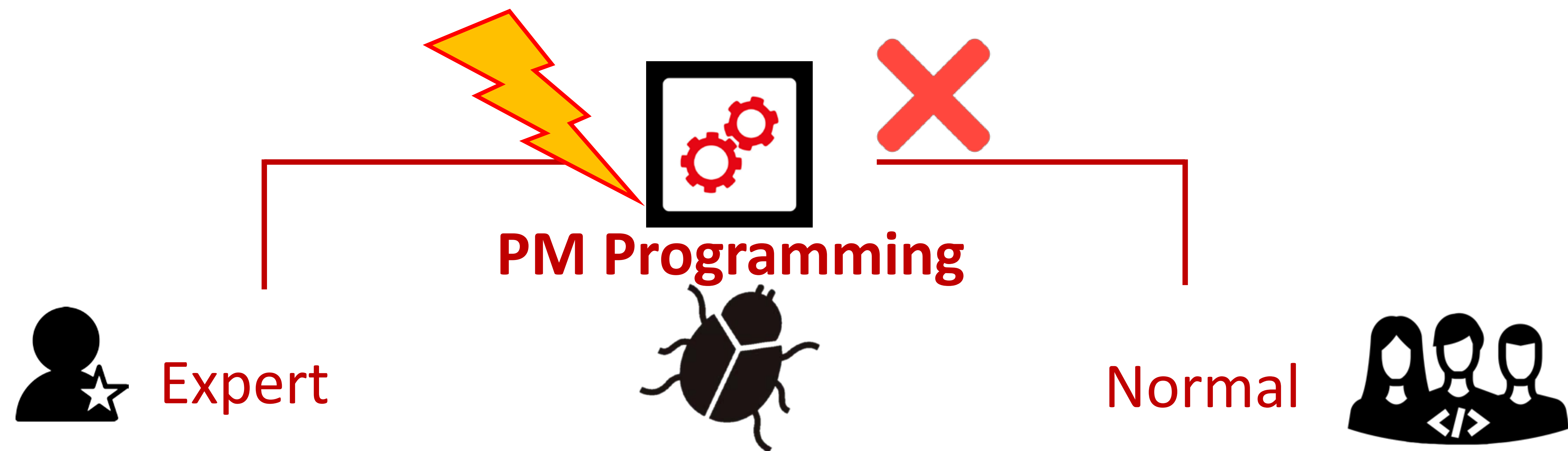
PROGRAMMING USING TRANSACTIONS

```
1 void ListAppend(item_t new_val) {
2   TX_BEGIN {
3     node_t *new_node = makeNode(new_val);
4     TX_ADD(list.head, sizeof(node_t*));
5     List.head = new_node;
6     TX_ADD(list.length, sizeof(unsigned));
      list.length++;
7   } TX_END
8 }
```

Backup `length` before update ✓

Ensuring crash consistency with transactions is still **HARD**!

PERSISTENCE MEMORY PROGRAMMING IS HARD

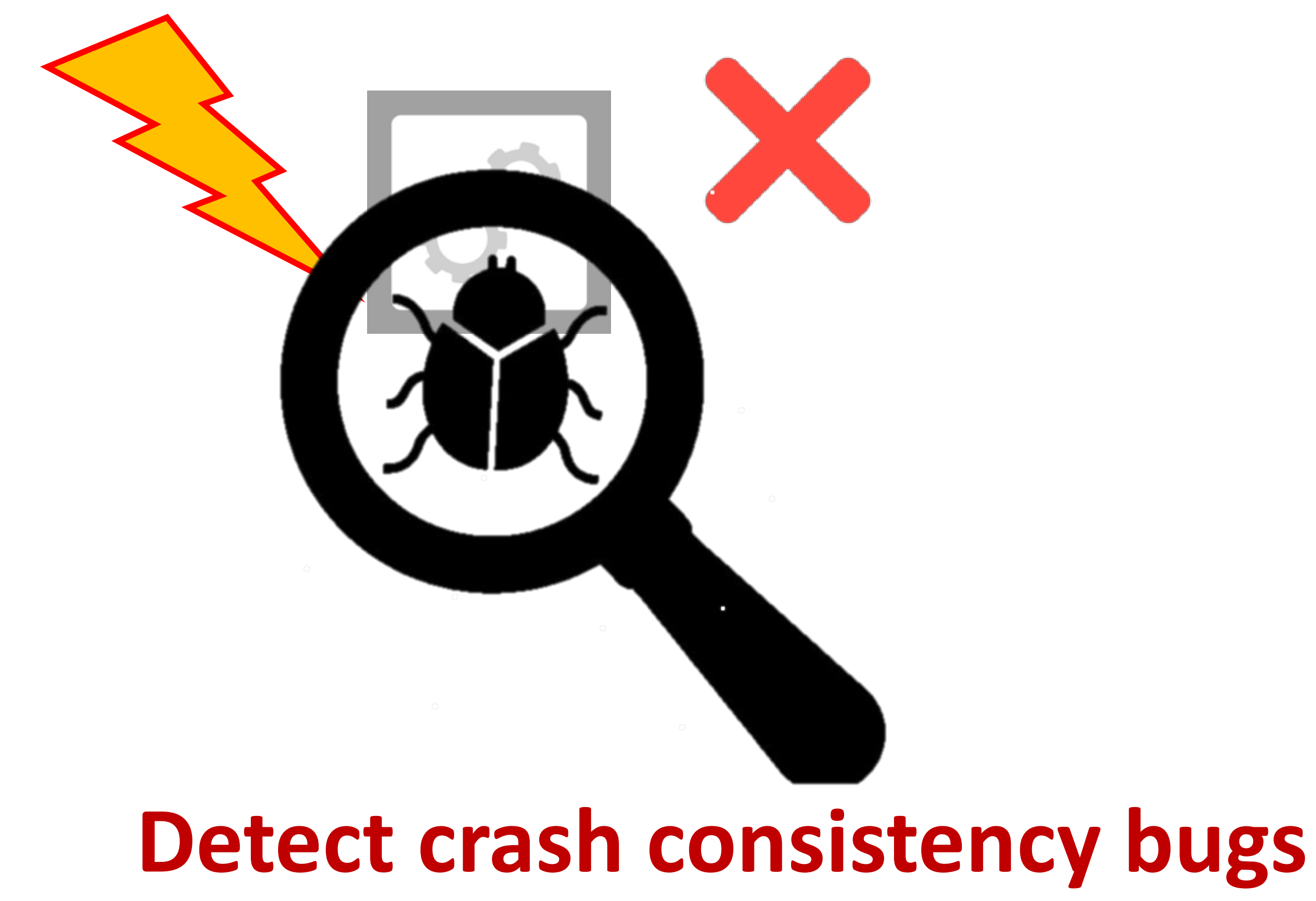


- Uses low-level primitives
- Understands the hardware
- Understands the algorithm

- Uses a high-level interface
- Does not need to know details of hardware or algorithm

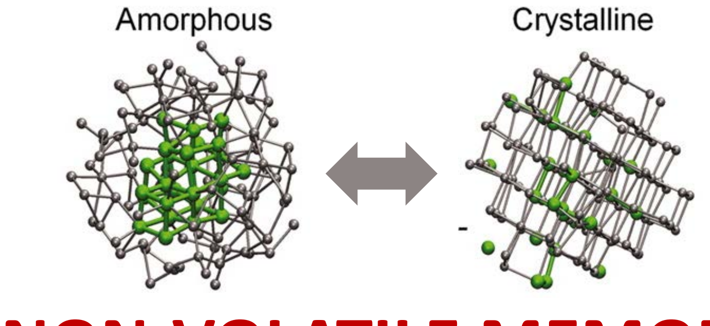
Both expert and normal programmers can make mistakes

PERSISTENT MEMORY PROGRAMMING IS HARD



Detect crash consistency bugs

We need a tool to detect crash consistency bugs!



NON-VOLATILE MEMORY

PERSISTENT MEMORY

Programming Persistent Memory Applications

CHALLENGE: PM Programming is Hard!

Requirements and Key Ideas

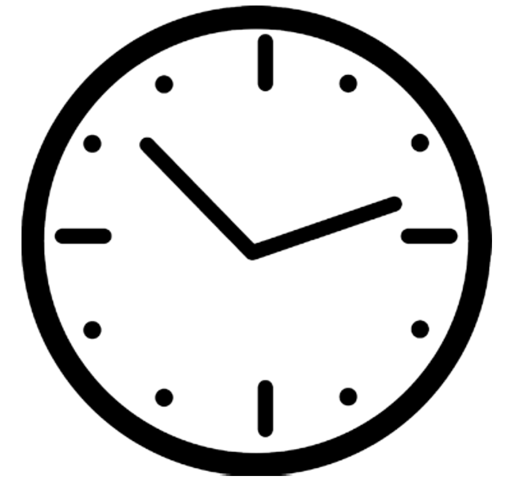
PMTEST: Interface and Mechanism

ASPLOS'19

Results

REQUIREMENTS OF THE TOOL

Fast



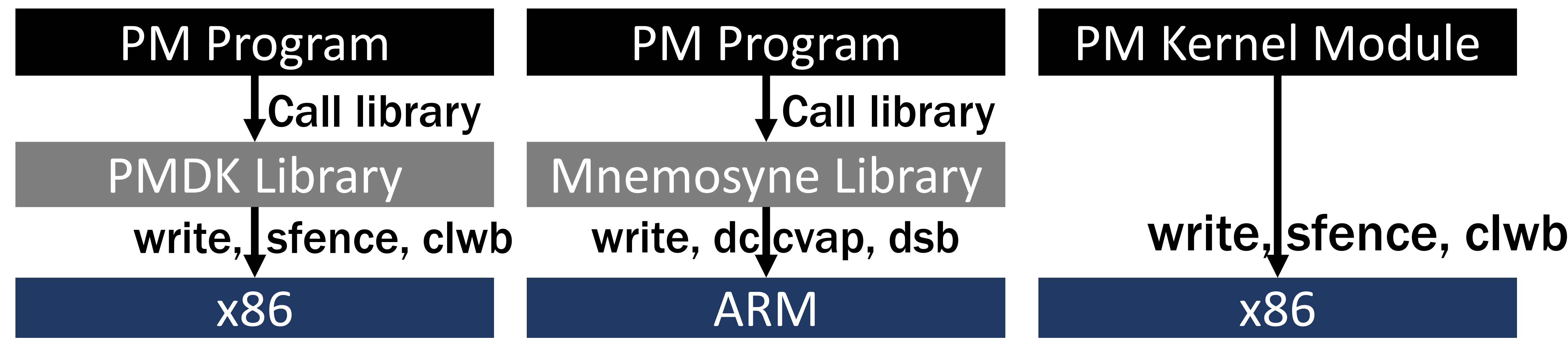
Flexible

[PMDK, NV-Heaps’11, Mnemosyne’11, ATLAS’14, REWIND’15, NVL-C’16, NVThreads’17 LSNVMM’17, etc.]

[PMFS’14, BPFS’09, NOVA’16, NOVA-Fortis’17, [illegible] etc.]

PMTEST KEY IDEAS: FLEXIBLE

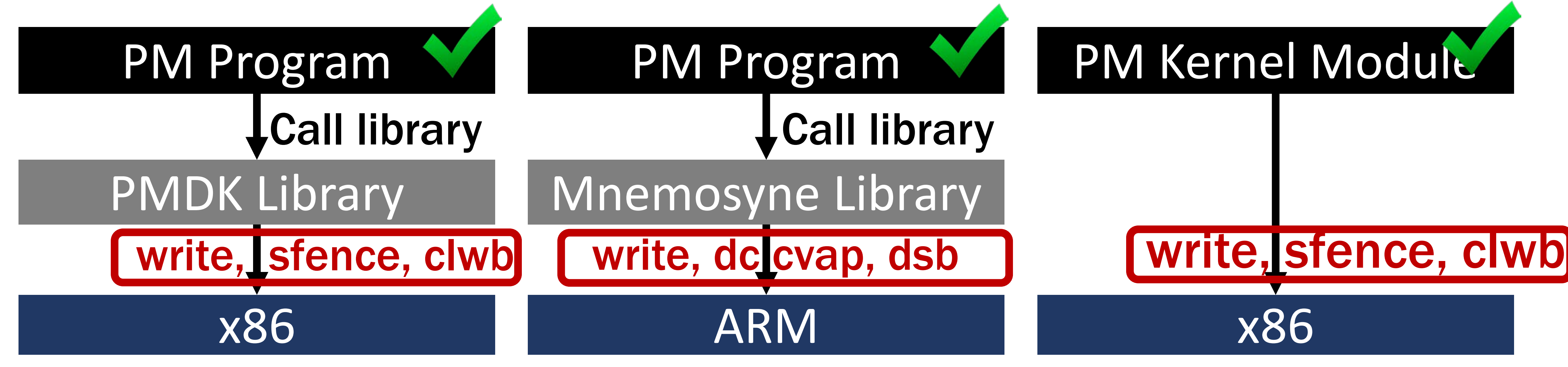
- Many different programming models and hardware primitives available



The challenge is to support **different** hardware and software models

PMTEST KEY IDEAS: FLEXIBLE

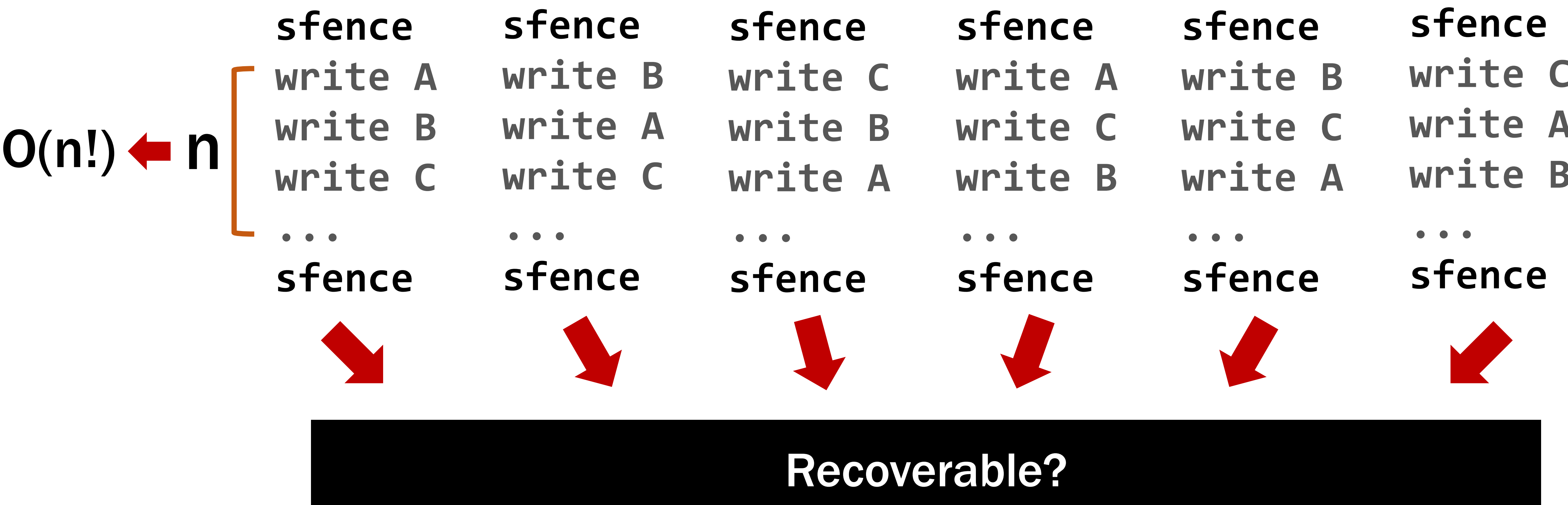
Operations that maintain crash consistency are similar:
ordering and durability guarantees



Our key idea is to test for these **two fundamental guarantees** which in turn can cover all hardware-software variations

PMTEST KEY IDEAS: FAST

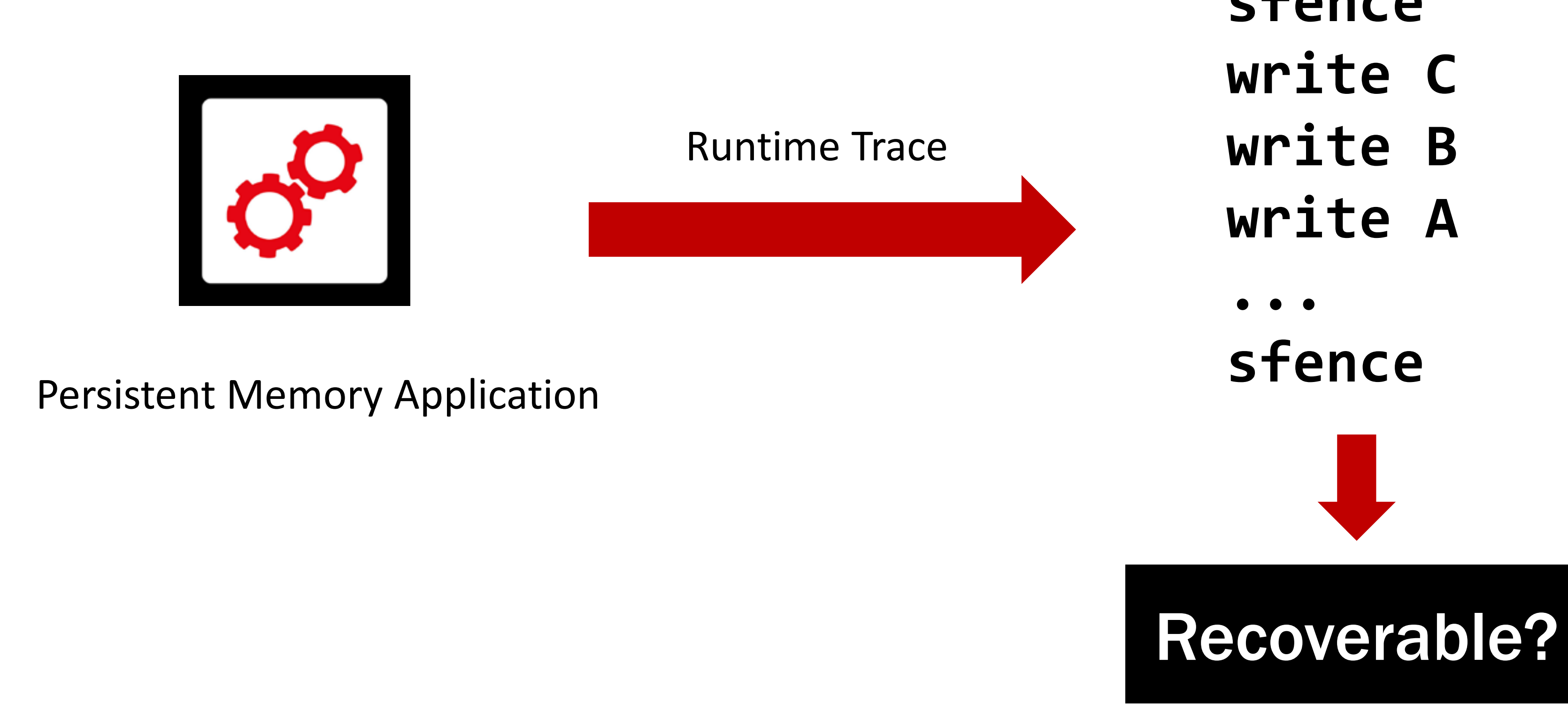
- Prior work [Yat'14] uses exhaustive testing



Exhaustive testing is time consuming and not practical

PMTest KEY IDEAS: FAST

- Reduce test time by using only one dynamic trace



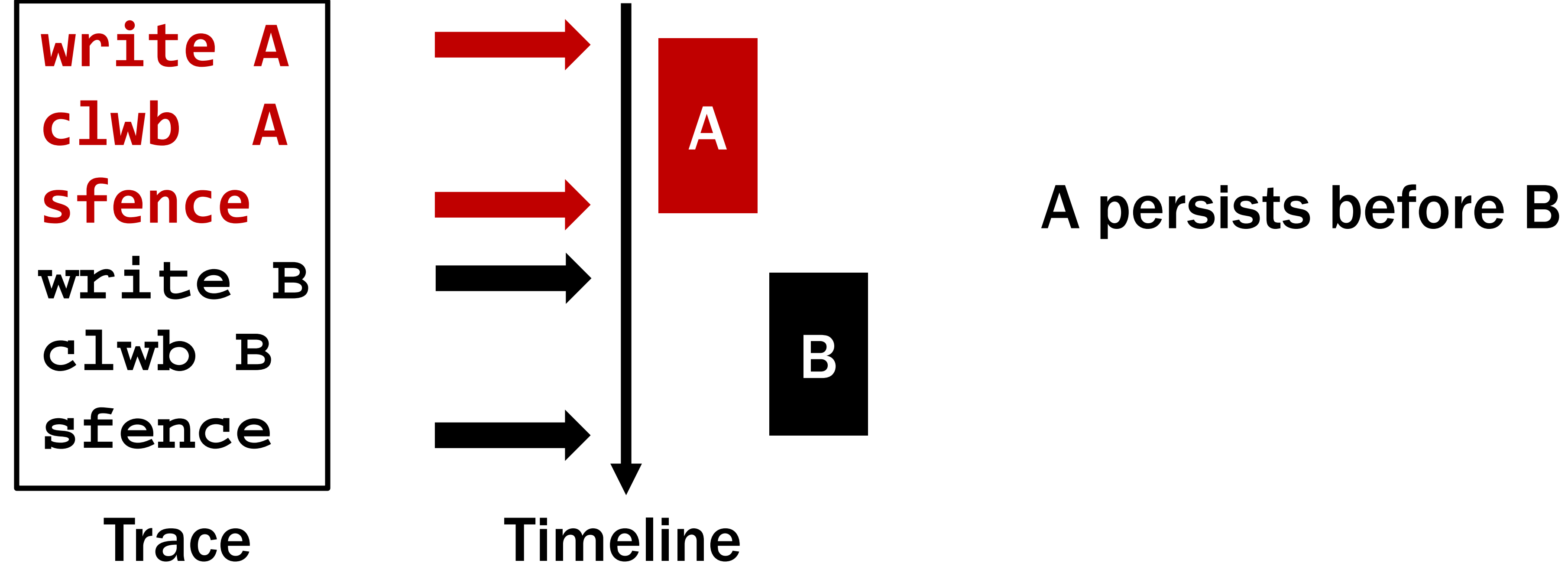

```
sfence
write C
write B
write A
...
sfence
```

Recoverable?

A significant improvement over O(n!) testing

PMTEST KEY IDEAS: FAST

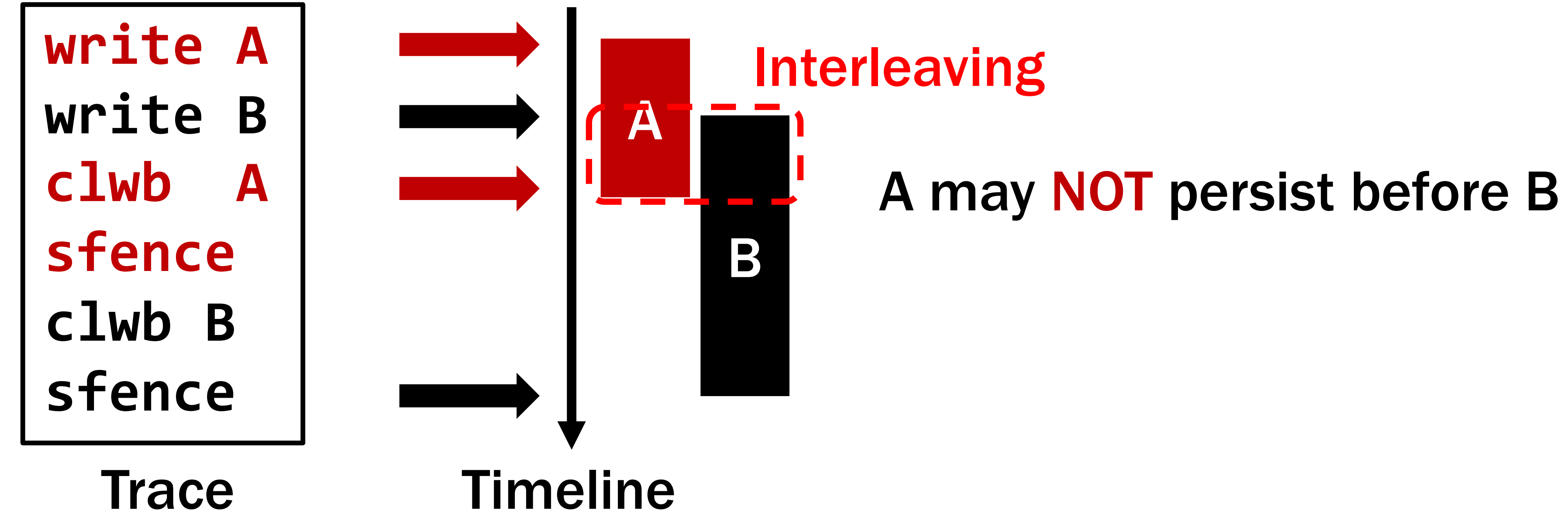
- PMTest infers the **persistence interval** from PM operation trace
  ➡ The interval in which a write can possibly become persistent



A disjoint interval indicates that no re-ordering in the hardware will lead to a case where A does not persist before B

PMTEST KEY IDEAS: FAST

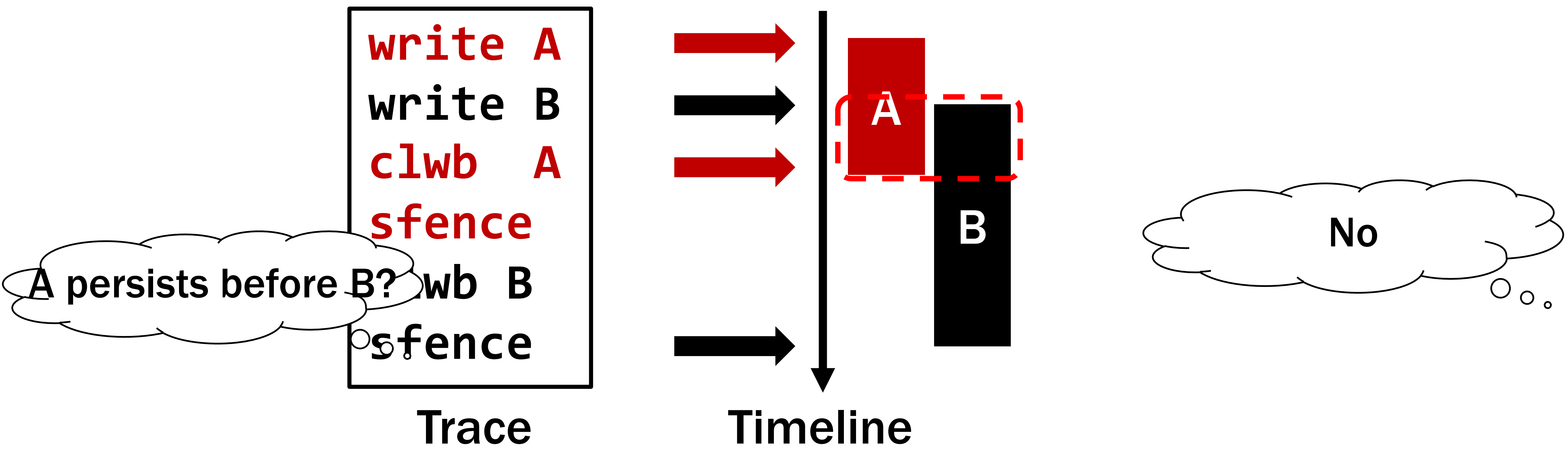
- PMTest infers the **persistence interval** from PM operation trace
  The interval in which a write can possibly become persistent



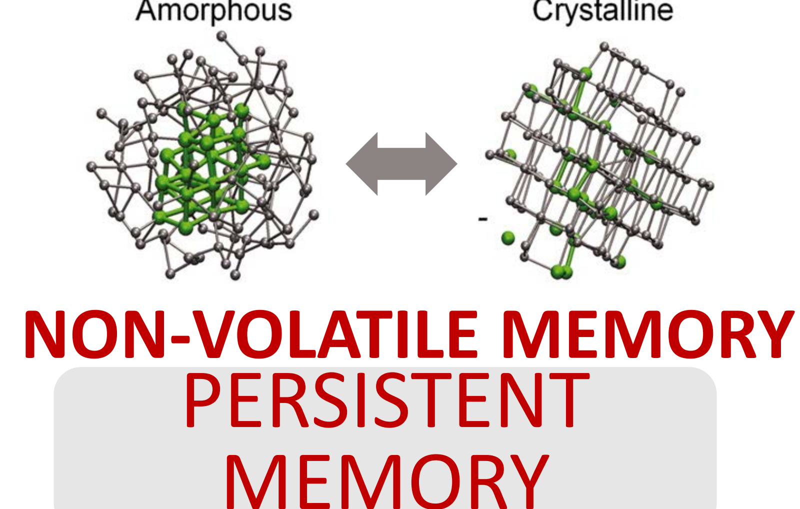
An overlapping interval indicates that there is a case where A does not persist before B

PMTest KEY IDEAS: FAST

- PMTest infers the **persistence interval** from PM operation trace
  The interval in which a write can possibly become persistent



Querying the trace can detect any violation
in ordering and durability guarantee at runtime



NON-VOLATILE MEMORY

PERSISTENT MEMORY

Programming Persistent Memory Applications

CHALLENGE: PM Programming is Hard!

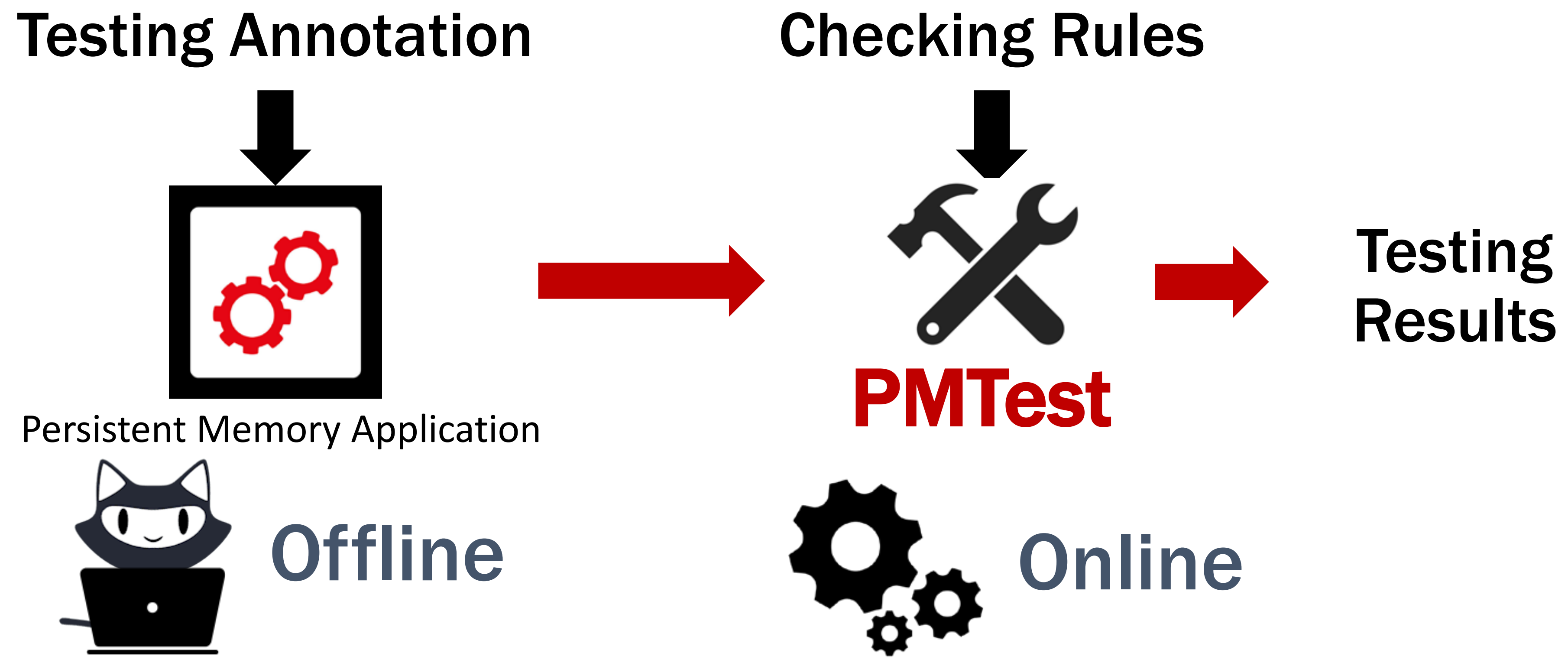
Requirements and Key Ideas

PMTEST: Interface and Mechanism

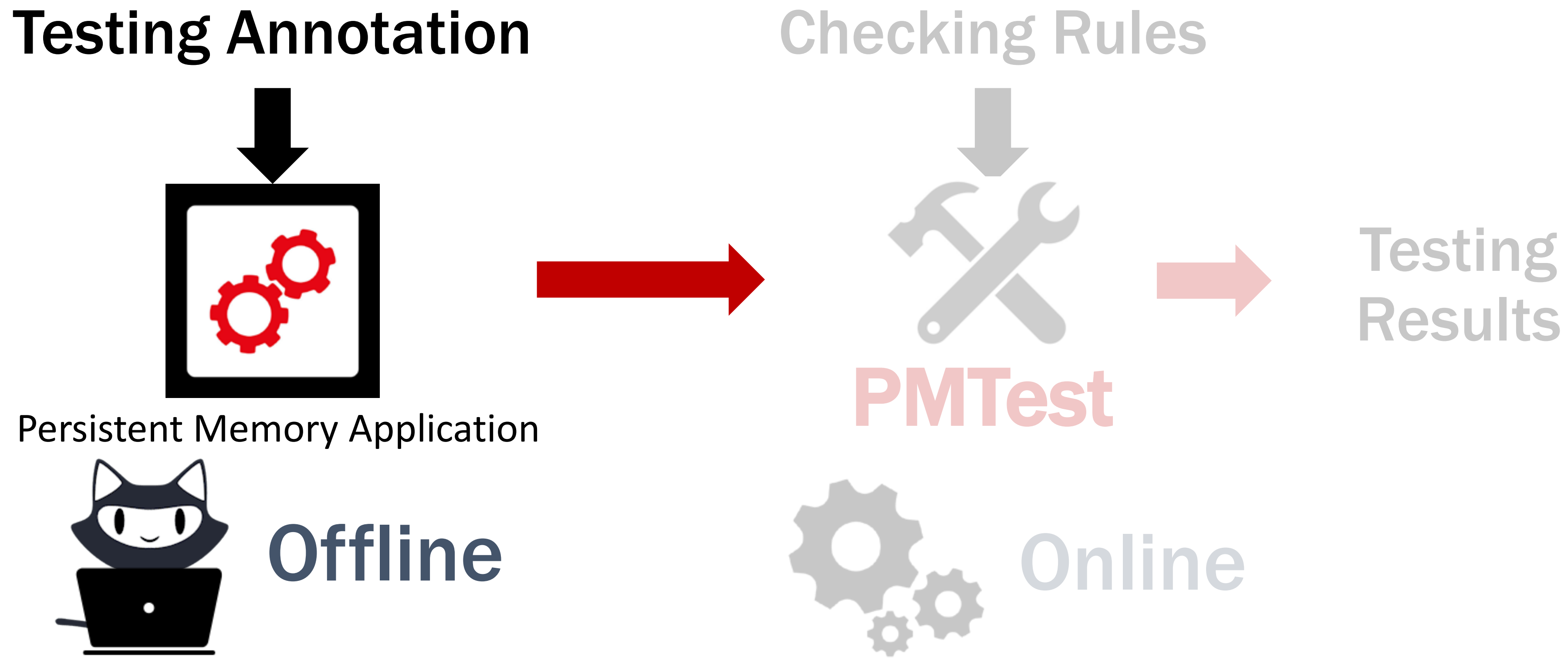
ASPLOS'19

Results and Conclusion

PMTest OVERVIEW



PMTEST OVERVIEW



PMTest INTERFACE

- Assertion-like low-level interface
- Check behavior vs. specification

- High-level interface
- Minimize programmer's effort
- Automatically inject low-level checkers

PMTest provides two different interfaces

PMTest LOW-LEVEL INTERFACE

- Two low-level checkers
  - **isOrderedBefore**(A, sizeA, B, sizeB)
    - ➡ Checks whether A is persisted before B **(Ordering guarantee)**
  - **IsPersisted**(A, sizeA)
    - ➡ Checks whether A has been written back to PM **(Durability guarantee)**

PMTest LOW-LEVEL INTERFACE

- Two low-level checkers
  - **isOrderedBefore**(A, sizeA, B, sizeB)

    ➡ Checks whether A is persisted before B **(Ordering guarantee)**
  - **IsPersisted**(A, sizeA)

    ➡ Checks whether A has been written back to PM **(Durability guarantee)**
- Help check if implementation meets specification for
  - Programs/kernel modules based on low-level primitives
  - PM libraries

EXAMPLE

```
void hashMapRemove() {
   ...
   remove(buckets->bucket[hash]);
   count--;
   persist_barrier();
   ...
   hashmap_rebuild();


}
```

Check if **count** has been persisted *before* rebuilding

Check if all updates have been persisted in rebuilding

PMTest helps the programmers to reason about the code

*This example is inspired by *hashmap_atomic* from PMDK

PMTest **LOW-LEVEL** INTERFACE

- Two low-level checkers
  - **isOrderedBefore**(A, sizeA, B, sizeB)
    - ➡ Check whether A is persisted before B **(Ordering guarantee)**
  - **IsPersisted**(A, sizeA)
    - ➡ Check whether A has been written back to PM **(Durability guarantee)**
- Help check if implementation meets specification for
  - Programs/kernel modules based on low-level primitives
  - PM libraries
- Further enables **high-level** checkers to automate testing

PMTest **HIGH-LEVEL** INTERFACE

- Currently provides high-level checkers for PMDK transactions
- Automatically detects **crash consistency** bugs

```
void ListAppend(item_t new_val) {
  TX_CHECKER_START; //Start of TX checker
  TX_BEGIN {
    node_t *new_node = makeNode(new_val);
    TX_ADD(list.head, sizeof(node_t*));
    List.head = new_node;
    List.length++;
  } TX_END
  TX_CHECKER_END; //End of TX checker
}
```

Automatically check if there is a backup before update

Automatically check if all updates have been persisted

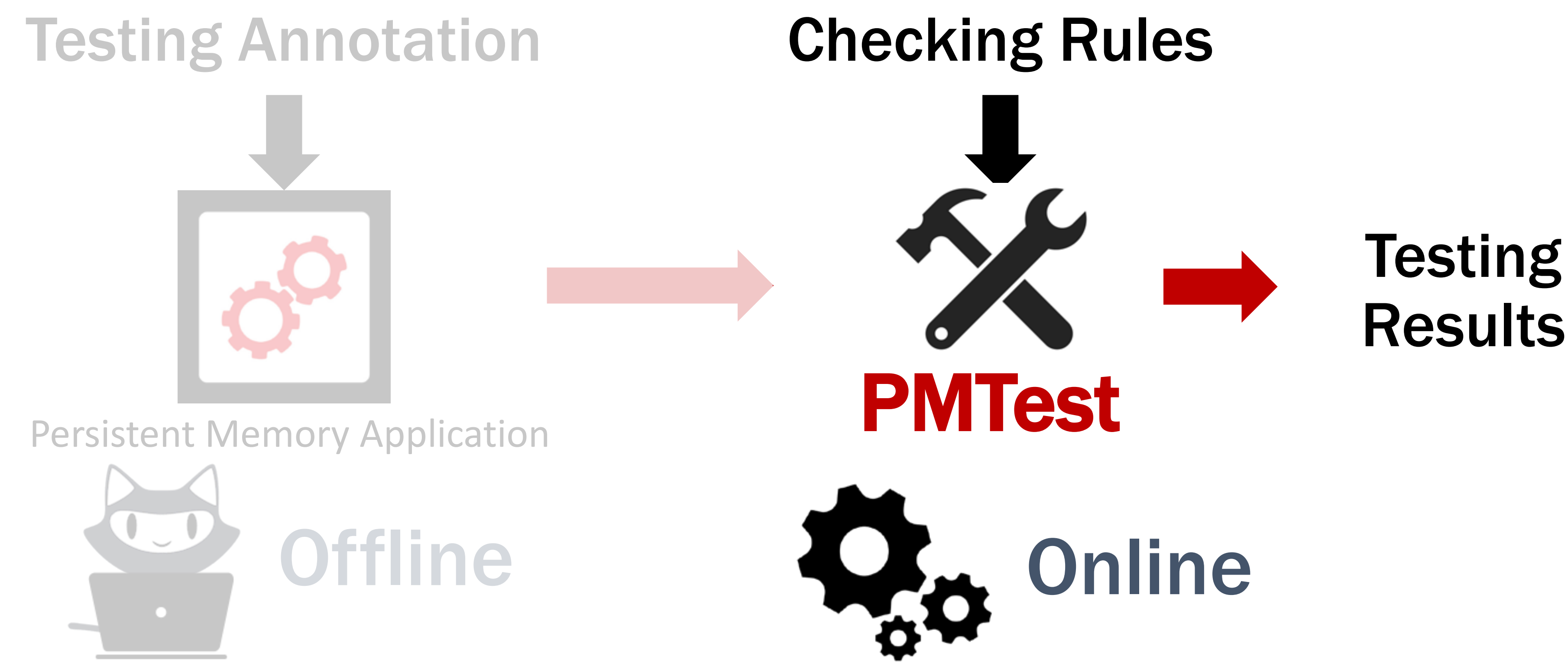
* This example does not include initialization and communication with PMTest

PMTEST HIGH-LEVEL INTERFACE

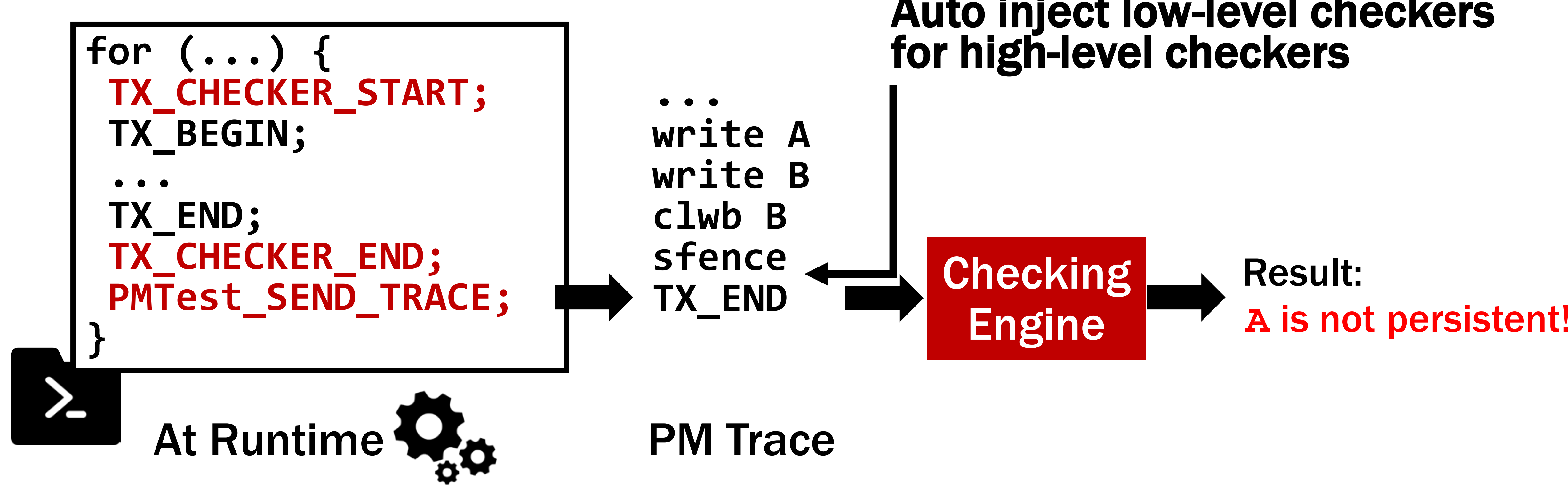
- Currently provides high-level checkers for PMDK transactions
- **Automatically** detects **crash consistency** bugs
  - If all updates have been persisted at the end of the transaction
  - If there is a backup before update during the transaction
- **Automatically** detects **performance** bugs
  - Redundant log/backup
  - Duplicated writeback/flush operations (for all programs)

High-level checkers minimize programmer's effort

PMTEST OVERVIEW



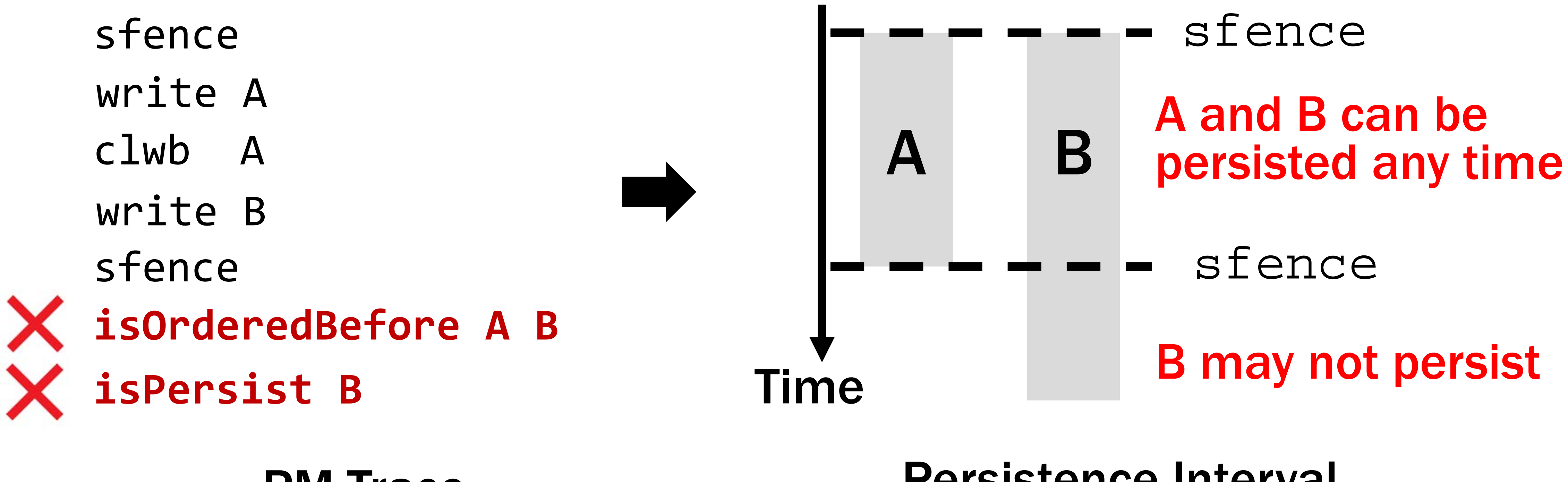
PMTEST CHECKING MECHANISM



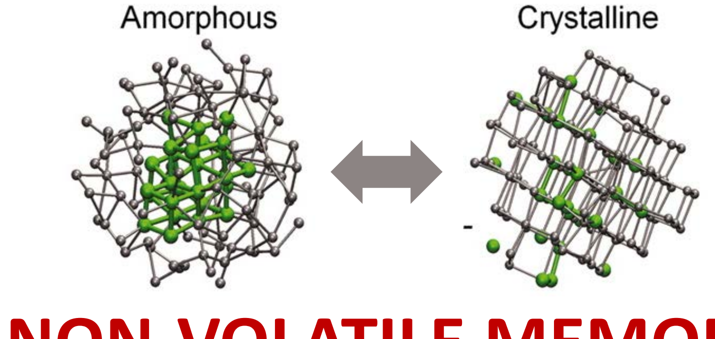
The checking engine tests the trace

CHECKING ENGINE ALGORITHM

- Infer the **persistence interval** in which a write can become persistent
- Check the interval against the low-level checkers



Our interval-based check enables faster testing



NON-VOLATILE MEMORY

PERSISTENT MEMORY

Programming Persistent Memory Applications

CHALLENGE: PM Programming is Hard!

Requirements and Key Ideas

PMTEST: Interface and Mechanism

ASPLOS'19

Results and Conclusion

METHODOLOGY

Platform

CPU: 8-core Skylake 2.1GHz, OS: Ubuntu 14.04, Linux kernel 4.4
Memory: 64GB DDR4
NVM: 64GB Battery-backed NVDIMM

Workloads

Micro-benchmarks (from PMDK)

- C-Tree
- B-Tree
- RB-Tree
- HashMap

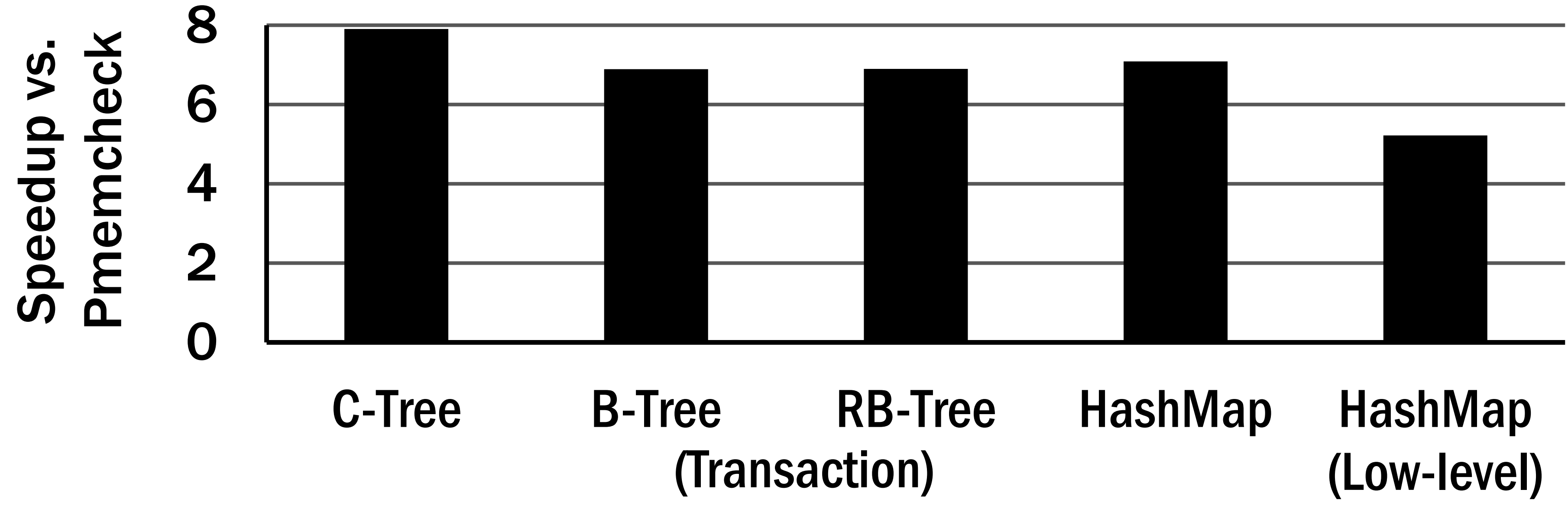
Real-world workloads

- PM-optimized file system
  - Intel's PMFS (kernel module)
- PM-optimized database
  - Redis (PMDK Library)
  - Memcached (Mnemosyne Library)

Baselines

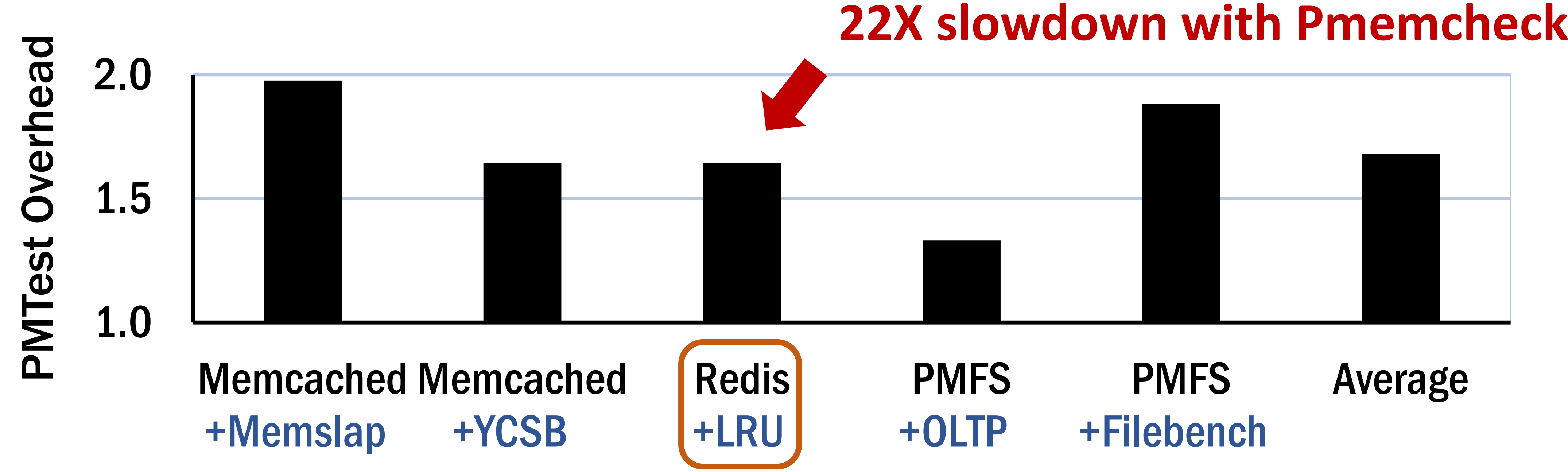
- No testing tool
- With Intel's **Pmemcheck** (only for PMDK-based programs)

MICRO-BENCHMARK



PMTest is 7.1X faster than Pmemcheck

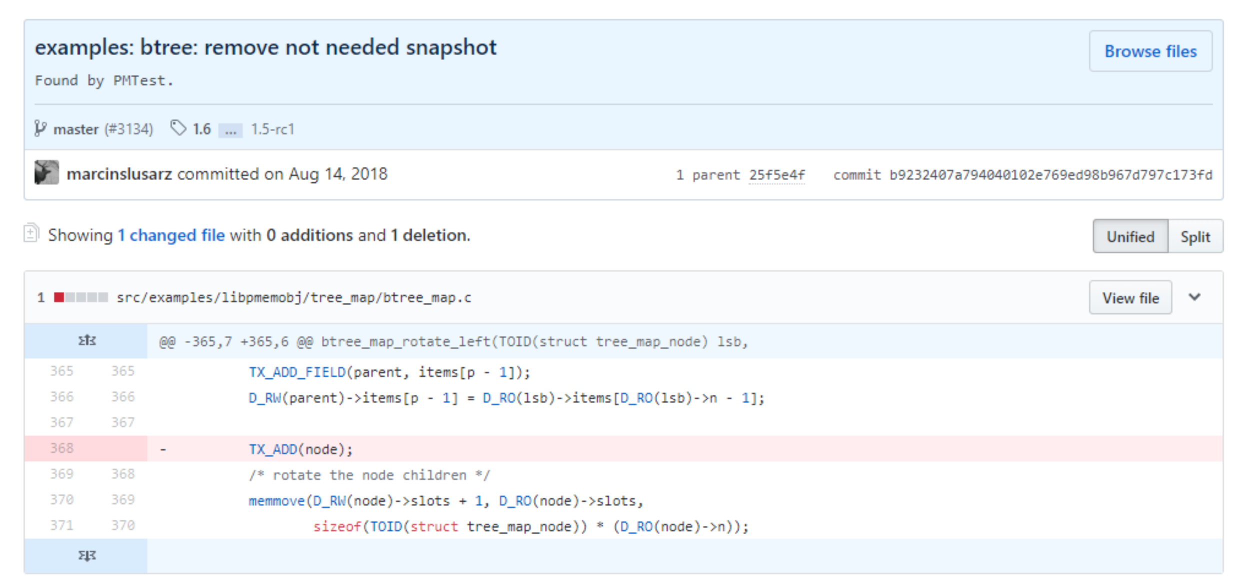
REAL-WORLD WORKLOADS

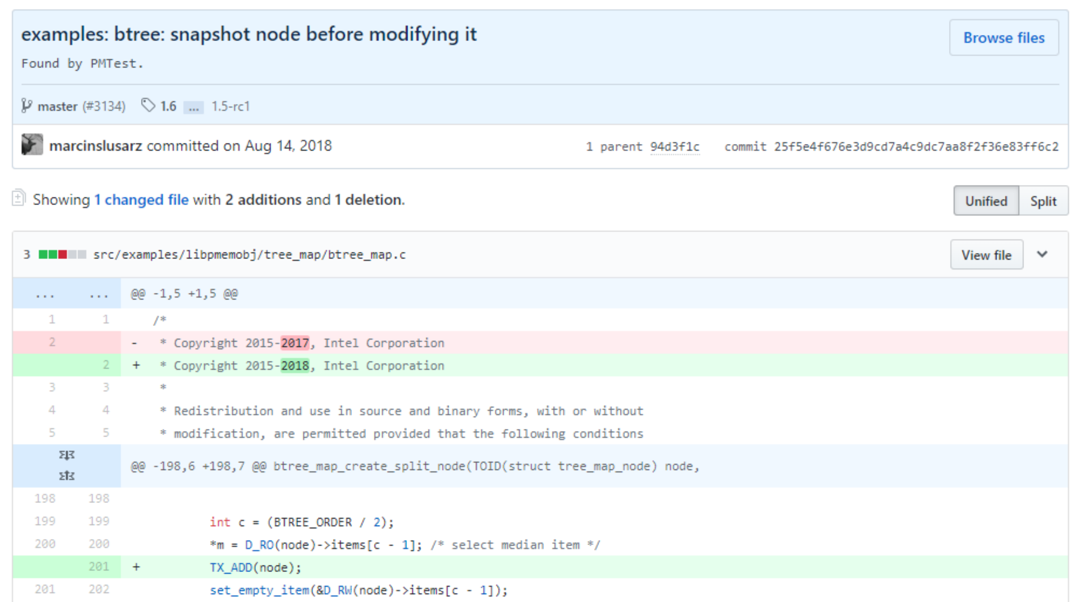


PMTest has < 2X overhead in real-world workloads

BUG DETECTION

- Validated with
  - 42 synthetic bugs injected to micro-benchmarks
  - 3 existing bugs from commit history
- New bugs found
  - 1 crash consistency bug in PMDK applications
  - 1 performance bug in PMFS
  - 1 performance bug in PMDK applications





CONCLUSION

- It is hard to guarantee crash consistency in persistent memory applications

pmtest.persistentmemory.org

- Our tool PMTest is **fast** and **flexible**
  - Flexible: Supports kernel modules, custom PM programs, transaction-based programs
  - Fast: Incurs < 2X overhead in real-workload applications

- PMTest has detected 3 new bugs in PMFS and PMDK applications

PMTEST
Testing Persistent Memory Applications

Samira Khan