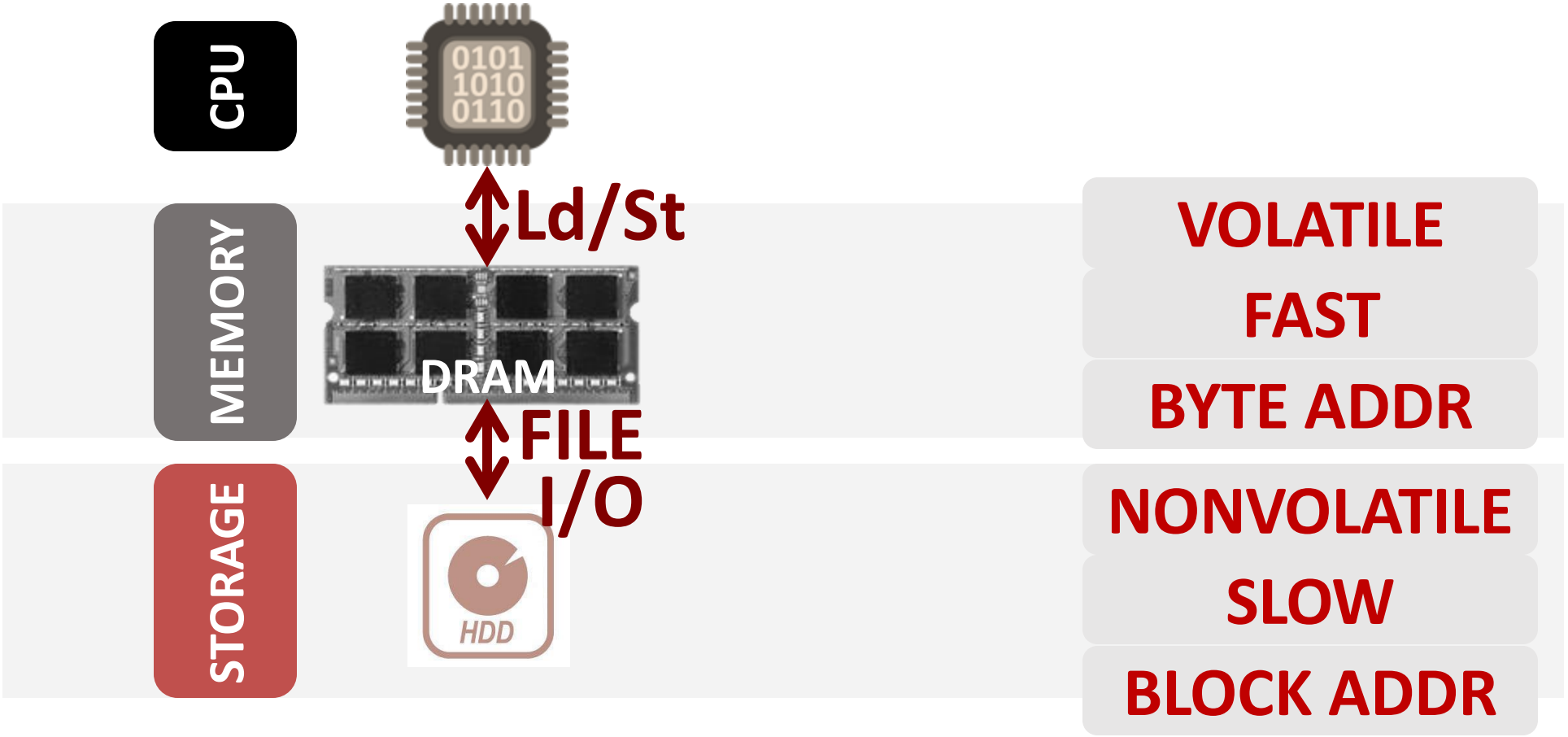
Rethinking System Support for Persistent Memory

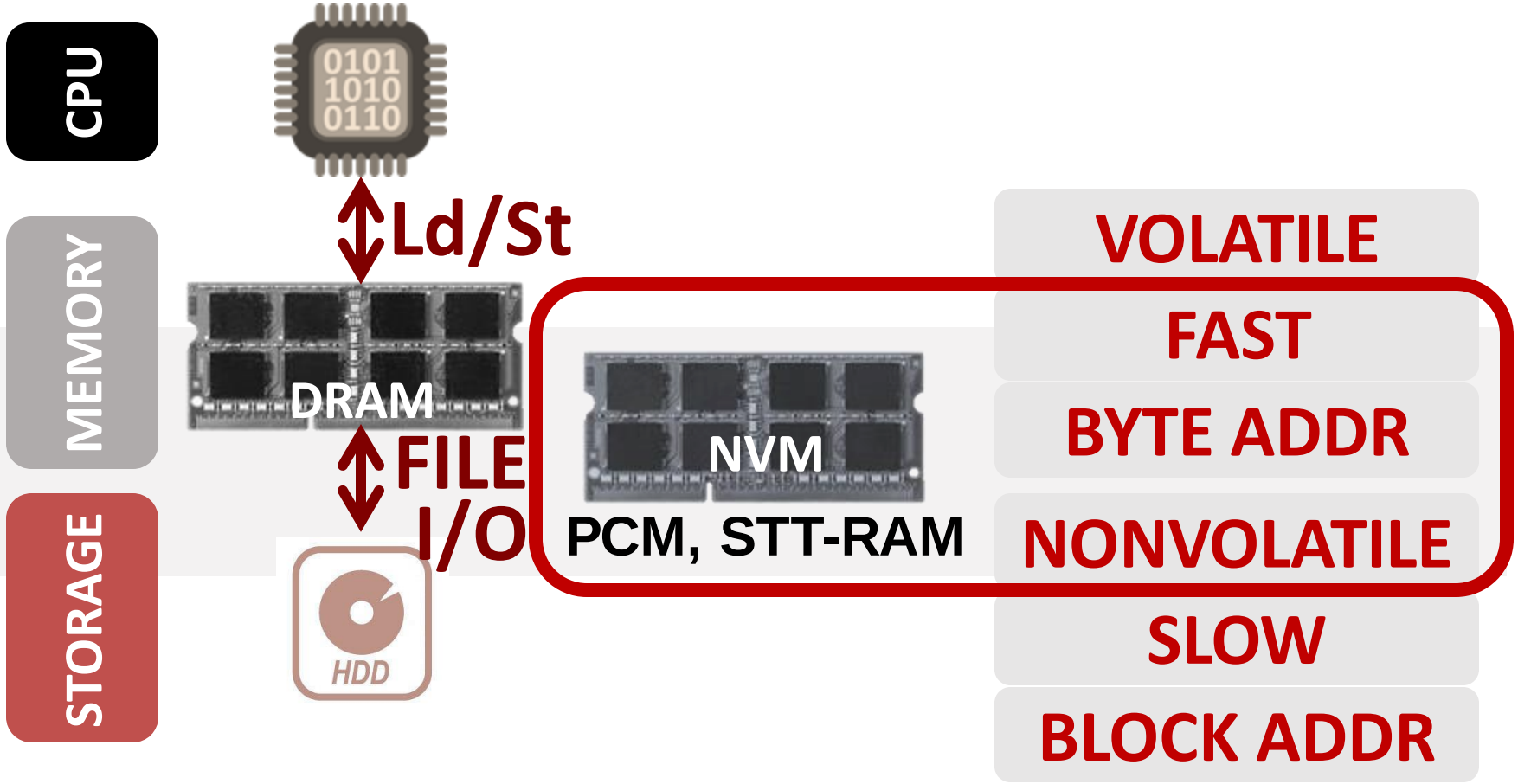
Samira Khan

University of Virginia

TWO-LEVEL STORAGE MODEL

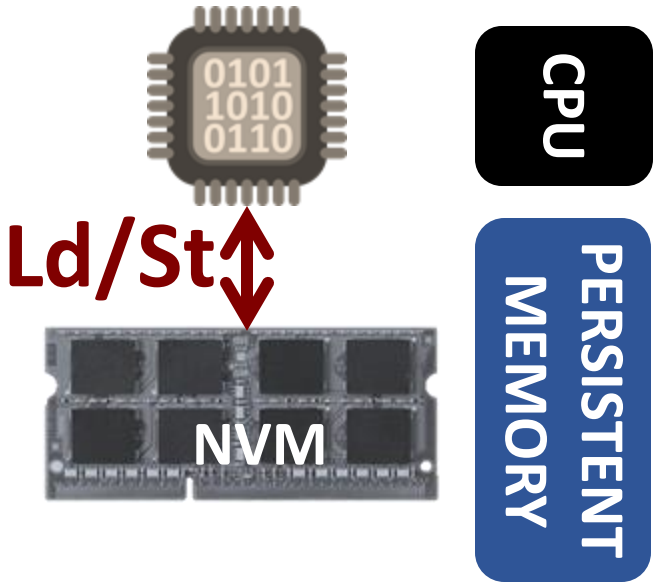


TWO-LEVEL STORAGE MODEL



Non-volatile memories combine characteristics of memory and storage

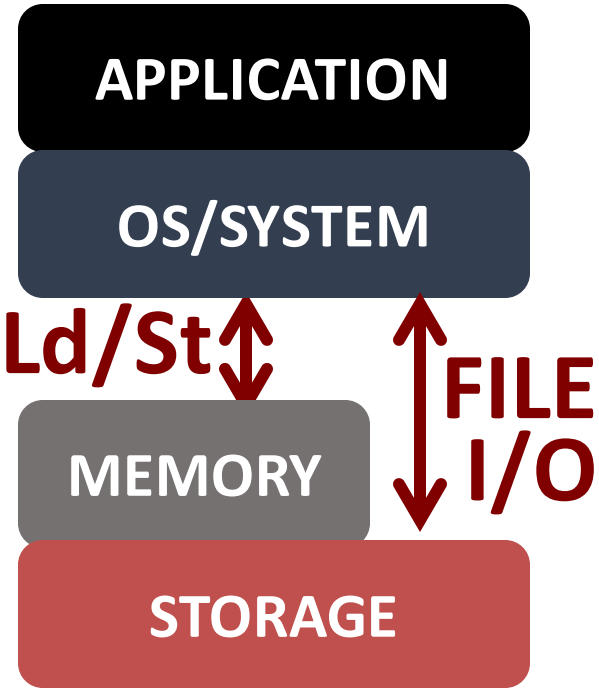
VISION: UNIFY MEMORY AND STORAGE



Provides an opportunity to manipulate persistent data directly in memory

Avoids reading and writing back data to/from storage

CHALLENGE: MEMORY & STORAGE SYSTEM SUPPORT



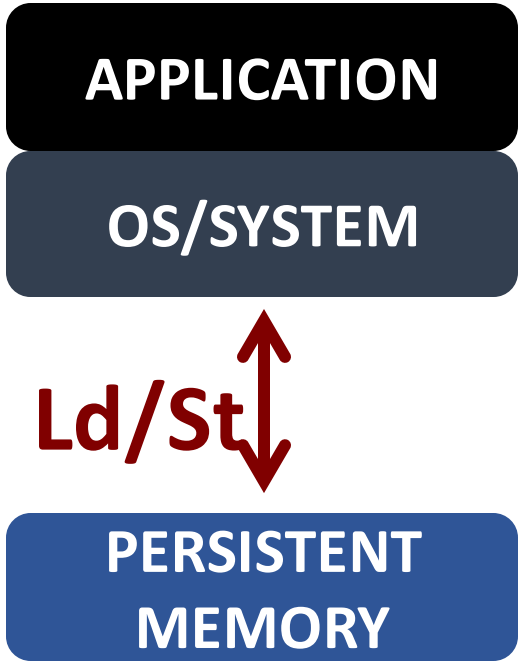
Crash Consistency

Availability

Compression

Integrity Check

Encryption

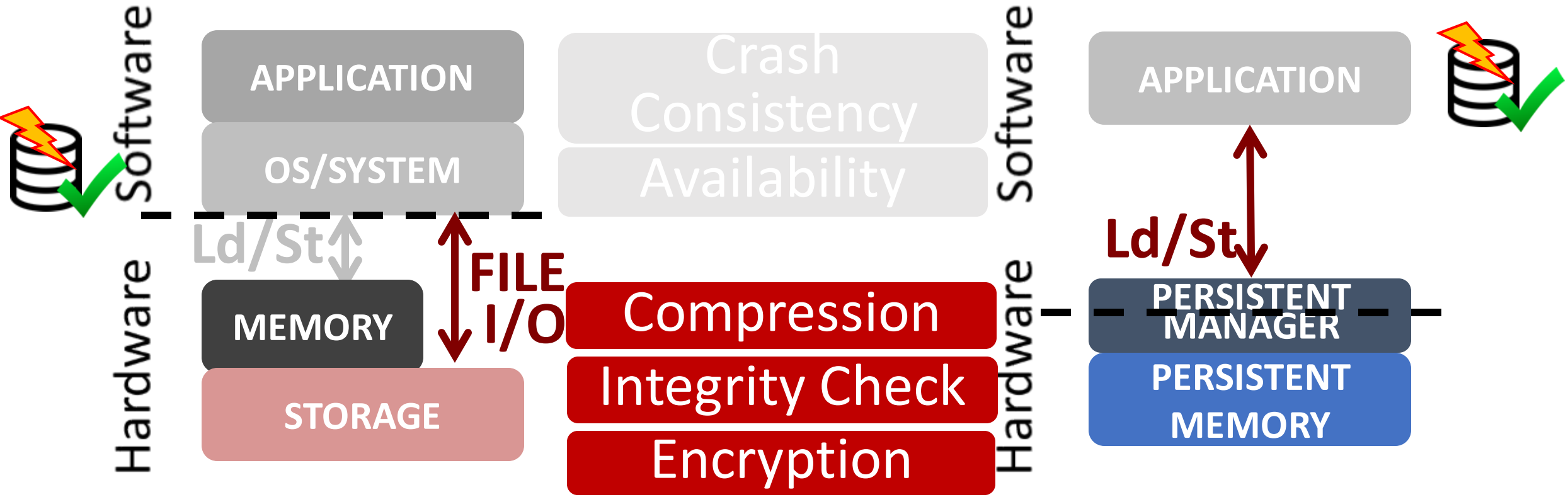


Overhead in OS/storage layer overshadows the benefit of nanosecond access latency of NVM

CHALLENGE: MEMORY & STORAGE SYSTEM SUPPORT

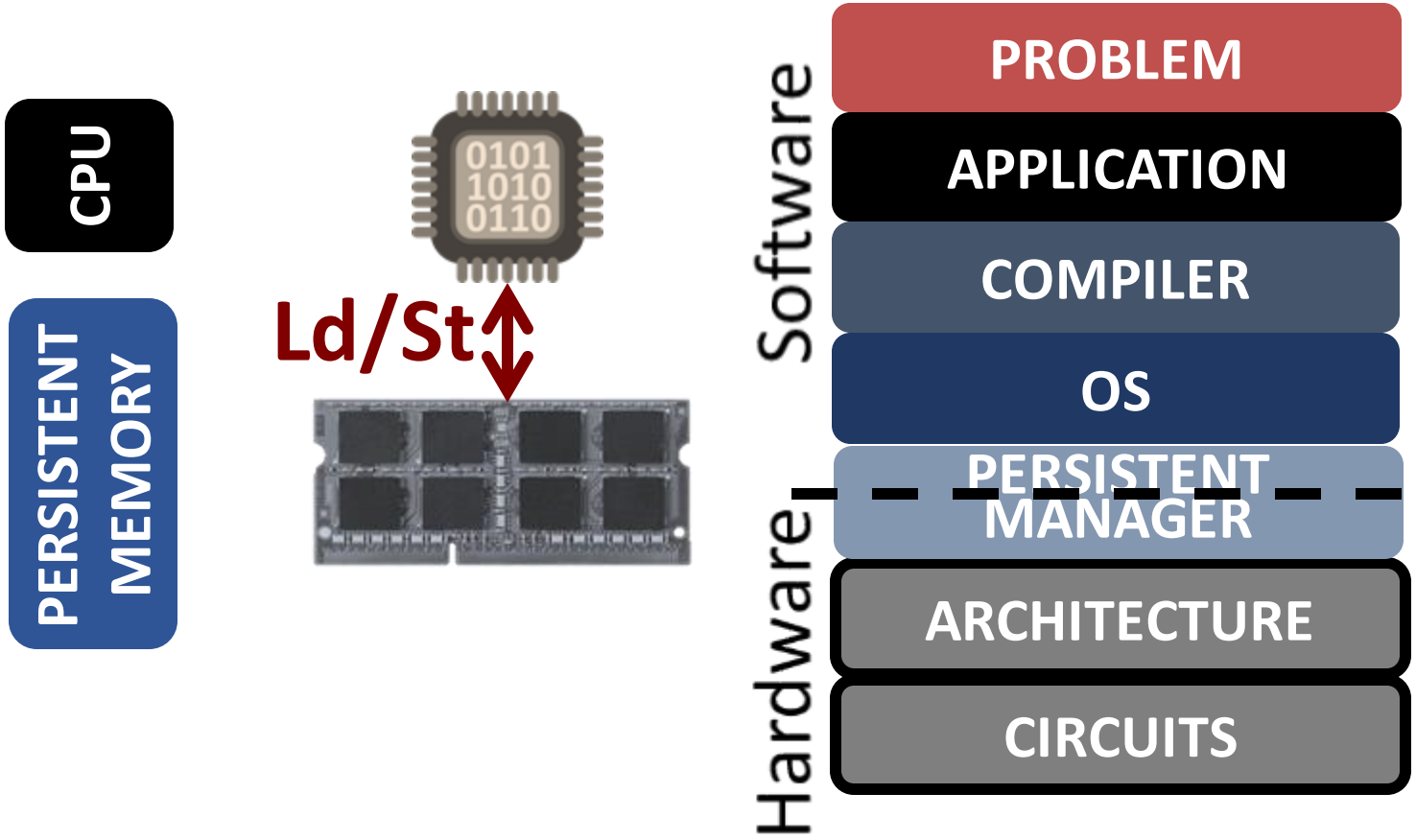
Not the operating system,
Application layer is responsible for crash consistency in PM

CHALLENGE: MEMORY & STORAGE SYSTEM SUPPORT



Not the operating system,
hardware is responsible for many system support in PM

GOAL: END-TO-END SYSTEM FOR PERSISTENT MEMORY



How to write consistent code?

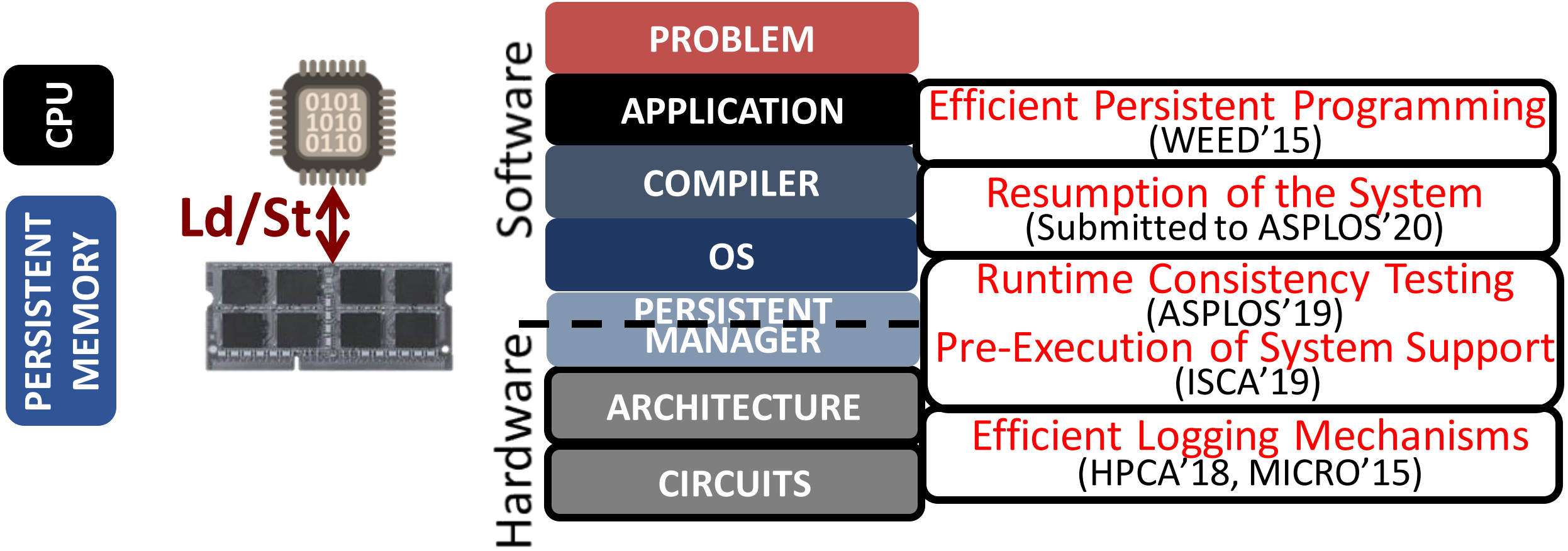
How to test the code is correct?

How to recover and resume application and OS?

How to provide efficient hardware support?

A full stack support for persistent memory applications

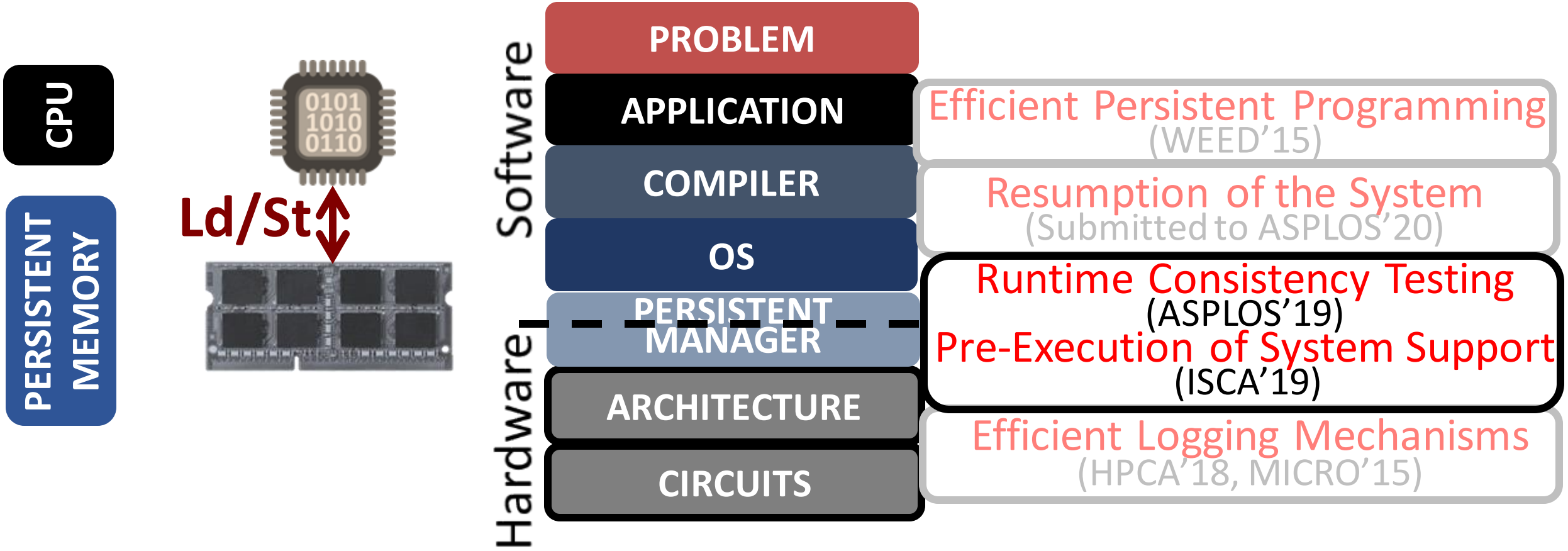
CURRENT WORKS SPAN THE WHOLE STACK



Programming and testing techniques for persistent memory applications

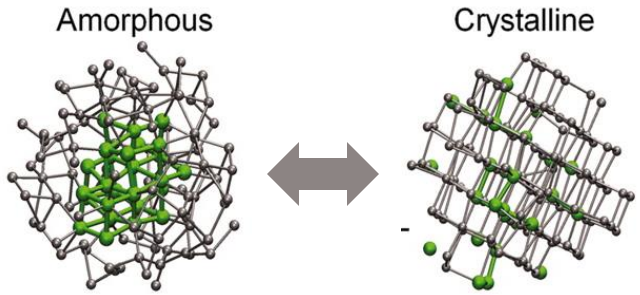
Efficient hardware and ISA support for persistent memory

CURRENT WORKS SPAN THE WHOLE STACK



Programming and testing techniques for persistent memory applications

Efficient logging and ISA support for persistent hardware



NON-VOLATILE MEMORY

PERSISTENT MEMORY

Unified Memory and Storage

Rethinking System Support

PMTEST: Testing for Correctness

ASPLOS'19

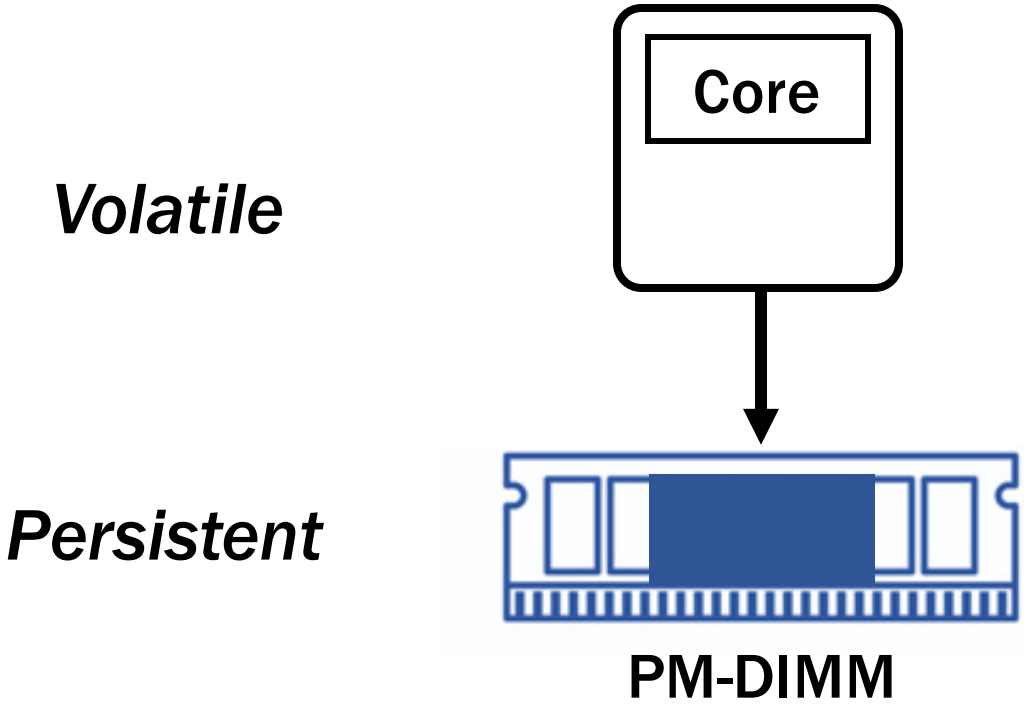
JANUS: Optimizing for Efficiency

ISCA'19

Conclusion

PERSISTENT MEMORY PROGRAMMING

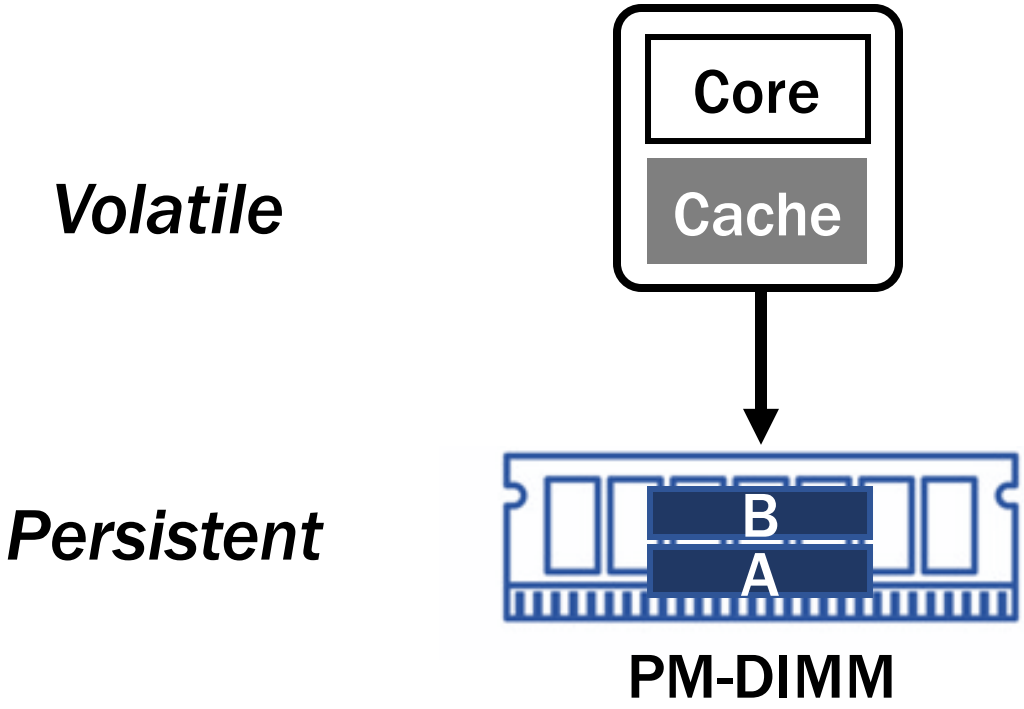
- Support for crash consistency have two fundamental guarantees
  - **Durability**: writes become persistent in PM
  - **Ordering**: one write becomes persistent in PM before another



Durability Guarantee:
writeback data from cache
Flush A

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



Ordering Guarantee:

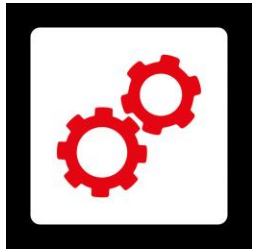
Write A before B

Writeback A

Barrier

Writeback B

PERSISTENT MEMORY PROGRAMMING



PM Programming

Expert

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

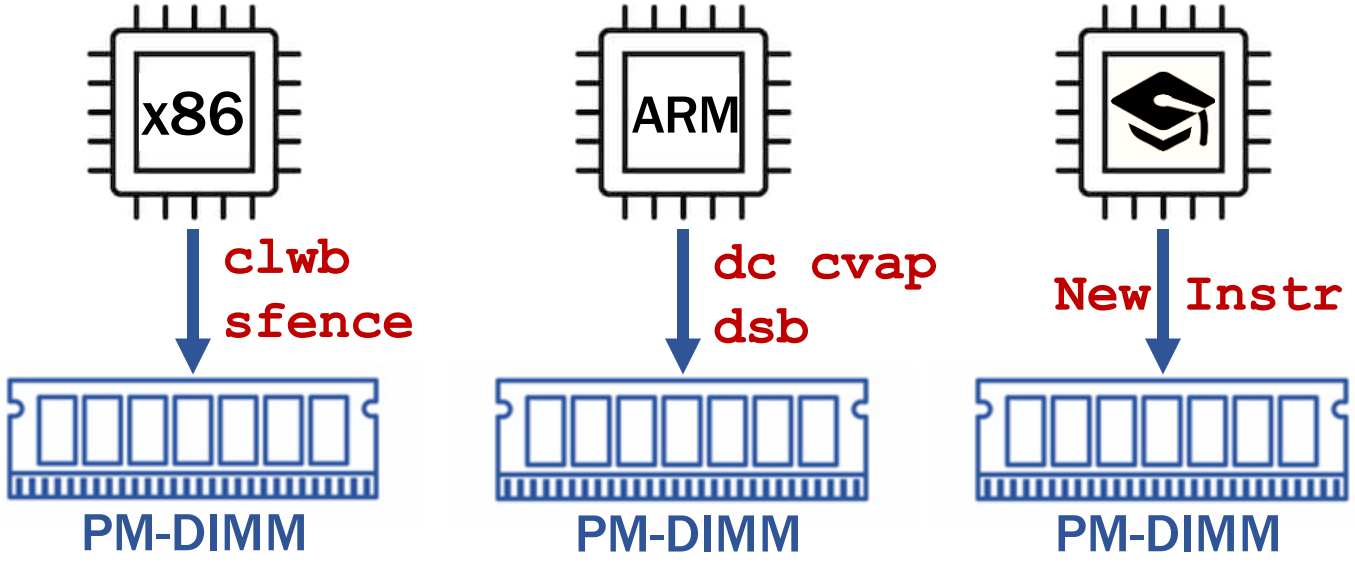
Normal

- 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

```
void listAppend(item_t new_val) {
  node_t* new_node = new node_t(new_val);
  new_node->next = head;
  head = new_node;
  persist_barrier();
}
```

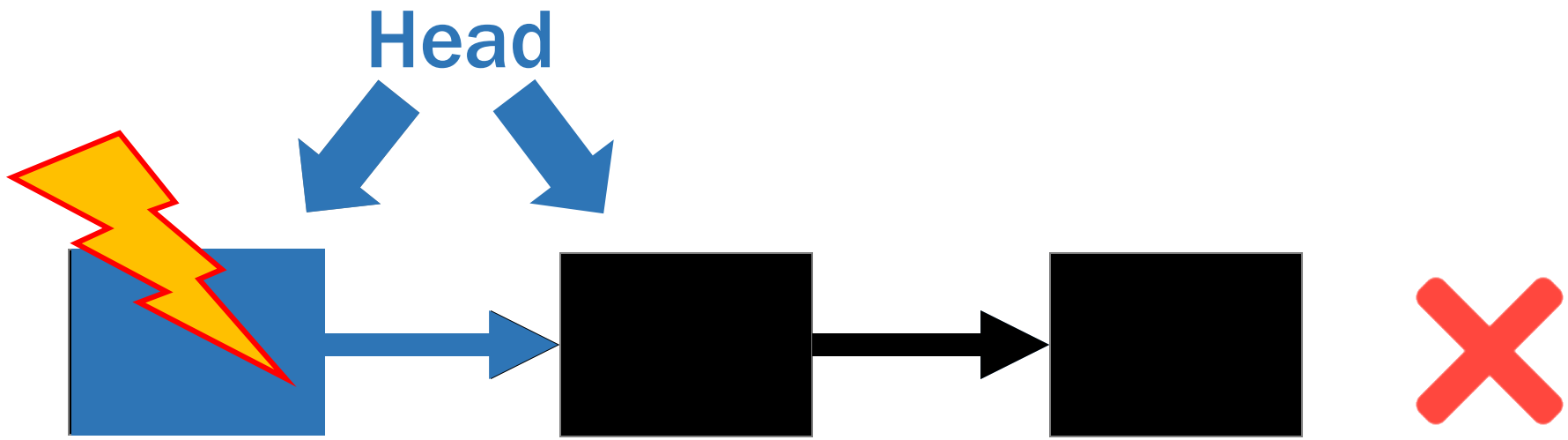
Create **new_node**

Update **new_node**

Update **head** pointer

Writeback updates

Writes to PM can reorder



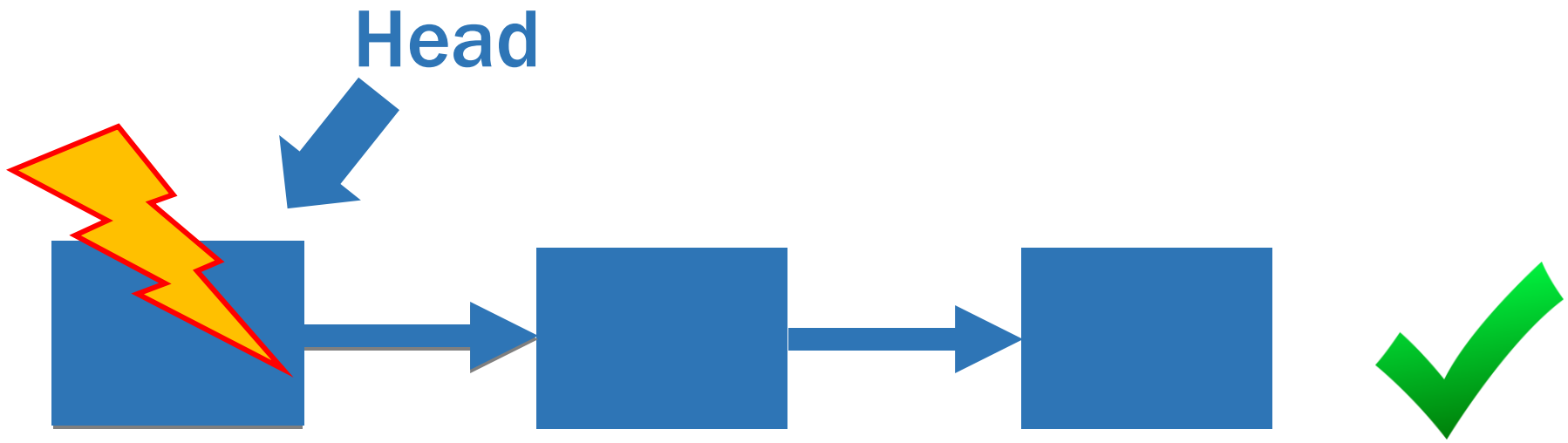
new_node is lost after failure

Inconsistent linked list

PROGRAMMING USING LOW-LEVEL PRIMITIVES

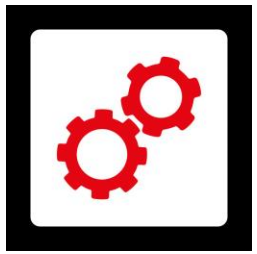
```
void listAppend(item_t new_val) {
  node_t* new_node = new node_t(new_val);
  new_node->next = head;
  persist_barrier();
  head = new_node;
  persist_barrier();
}
```

Enforce writeback before changing `head`



Ensuring crash consistency with low-level primitives is HARD!

PERSISTENT MEMORY PROGRAMMING



PM 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

```
void ListAppend(item_t new_val) {
  TX_BEGIN {
    node_t *new_node = makeNode(new_val);
    TX_ADD(list.head, sizeof(node_t*));
    List.head = new_node;
    List.length++;
  } TX_END
}
```

Create `new_node`

backup `head`

Update `head`

Update `length`

`length` **is not backed up before update!**

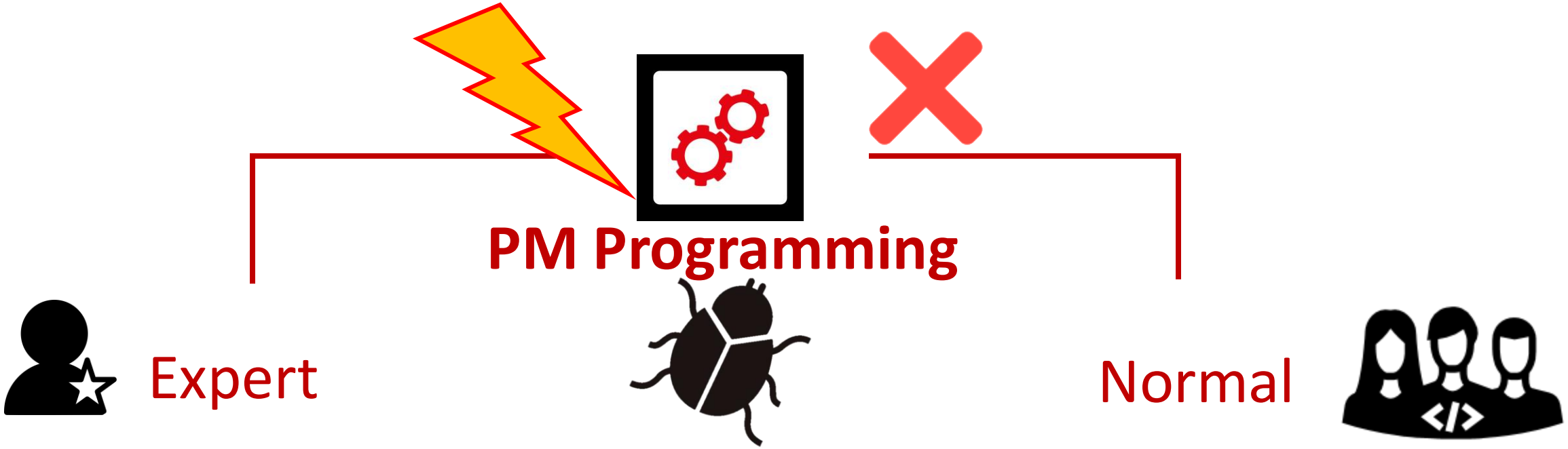
PROGRAMMING USING TRANSACTIONS

```
void ListAppend(item_t new_val) {
  TX_BEGIN {
    node_t *new_node = makeNode(new_val);
    TX_ADD(list.head, sizeof(node_t*));
    List.head = new_node;
    TX_ADD(list.length, sizeof(unsigned));
    List.length++;
  } TX_END
}
```

Backup `length` before update

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

PERSISTENCE MEMORY PROGRAMMING IS HARD



Expert

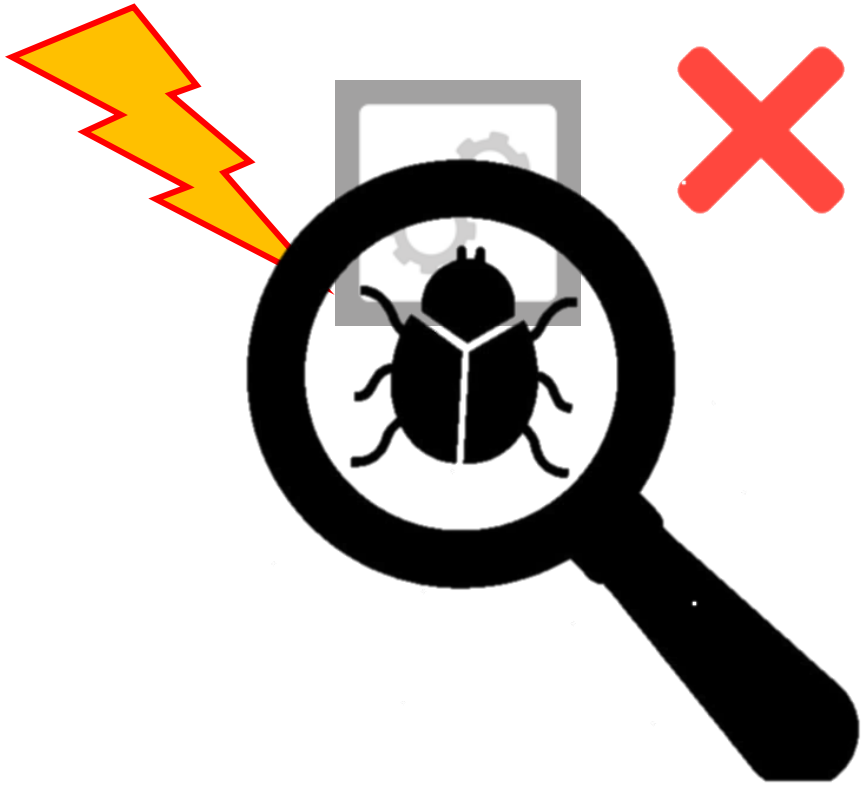
- Uses low-level primitives
- Understands the hardware
- Understands the algorithm

Normal

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

Both expert and normal programmers can make mistakes

PERSISTENT MEMORY PROGRAMMING IS HARD

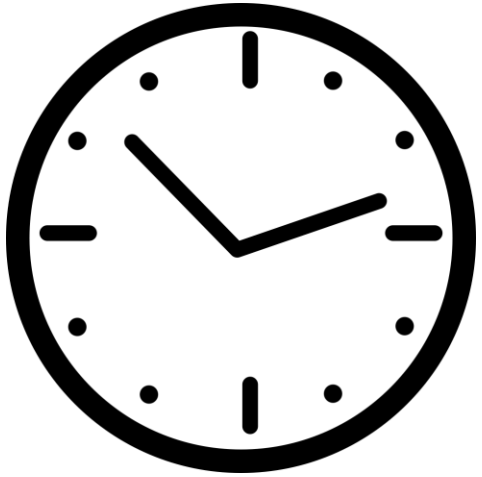


Detect crash consistency bugs

We need a tool to detect crash consistency bugs!

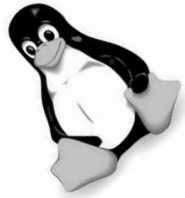
REQUIREMENTS OF THE TOOL

Fast



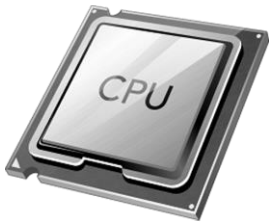
Flexible

PM Libraries



Kernel Modules

Custom Programs



Existing HW

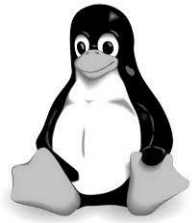
Future HW and Models

[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, Strata'17, SCM-FS'15, DAX, etc.] [LevelDB, etc.] [Ext4, etc.]

Our work: PMTest

Flexible


Kernel Modules ✓


PM Libraries ✓


Custom Programs ✓

Existing HW ✓

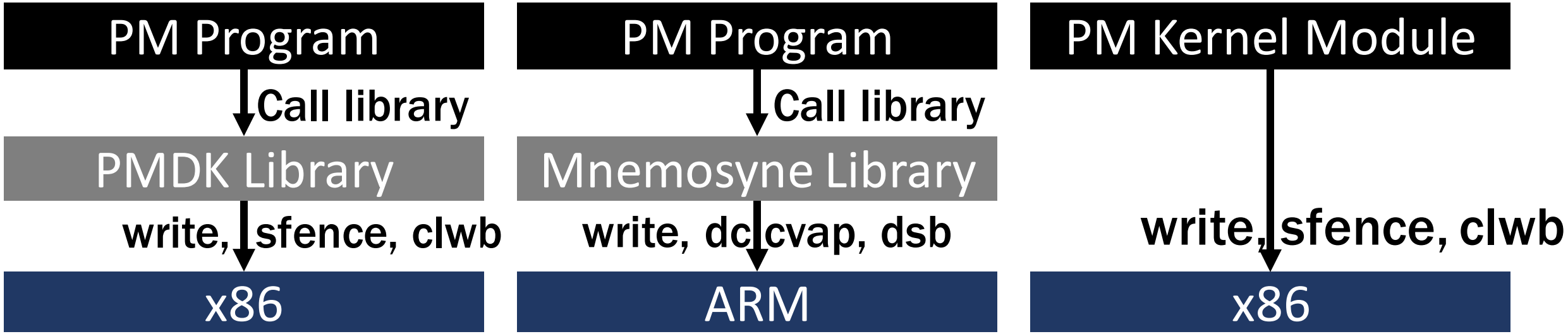

Academic Proposals ✓

Fast Less than 2X overhead in real workloads

PMTest has detected new bugs in PMFS and PMDK applications
Artifact available at pmtest.persistentmemory.org

PMTest KEY IDEAS: FLEXIBLE

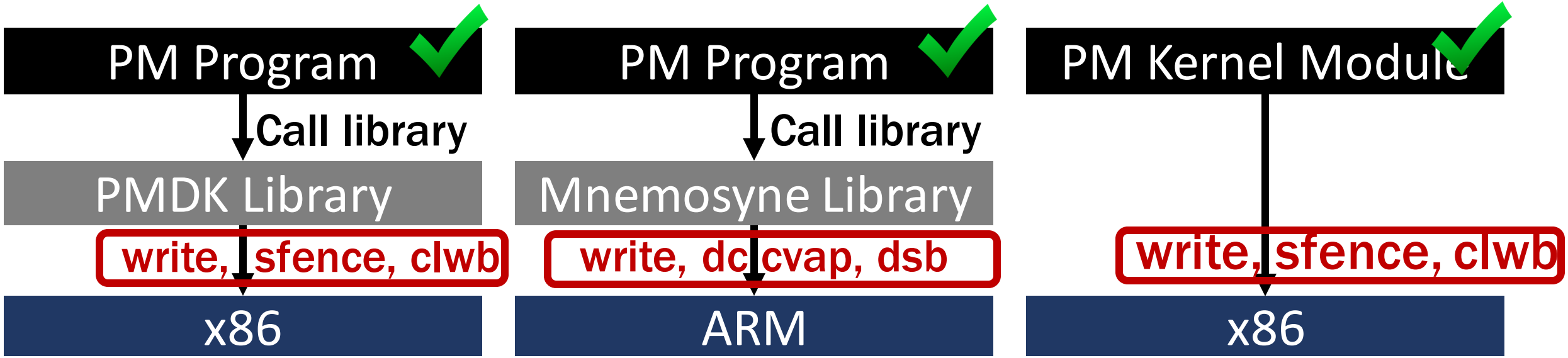
- Many different programming models and hardware primitives available



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

PMTest KEY IDEAS: FLEXIBLE

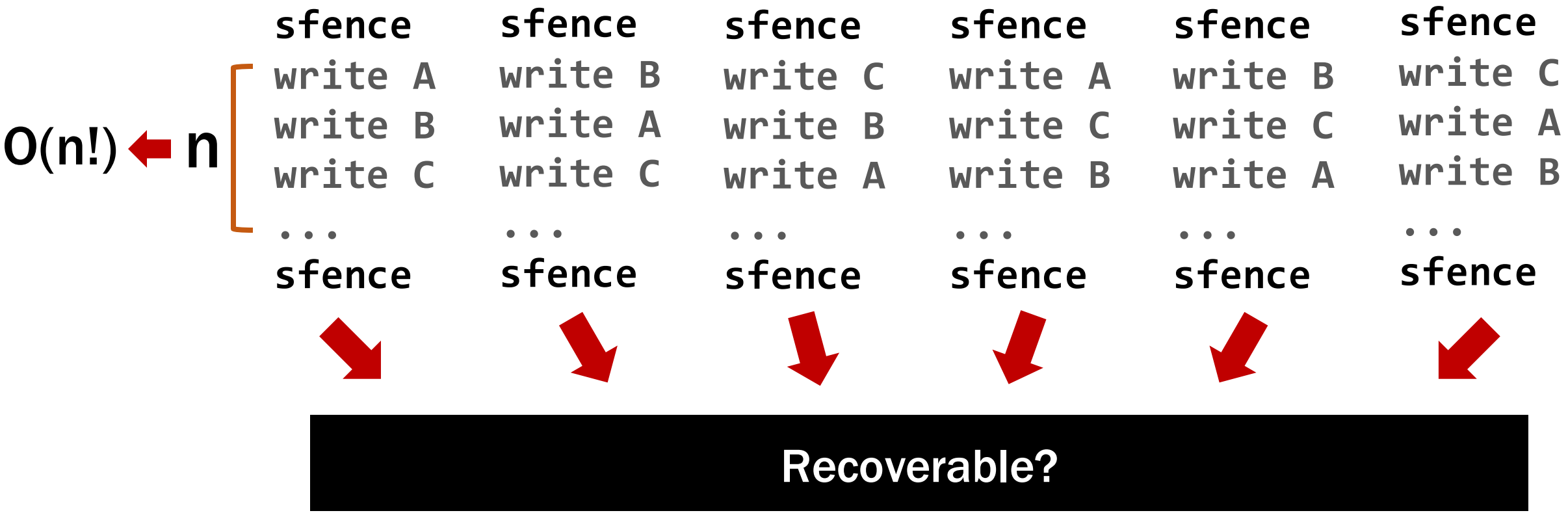
Operations that maintain crash consistency are similar:
ordering and durability guarantees



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

PMTest KEY IDEAS: FAST

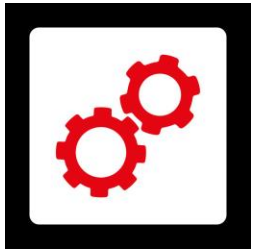
- Prior work [Yat'14] uses exhaustive testing



Exhaustive testing is time consuming and not practical

PMTest KEY IDEAS: FAST

- Reduce test time by using only one dynamic trace



Persistent Memory Application

Runtime Trace

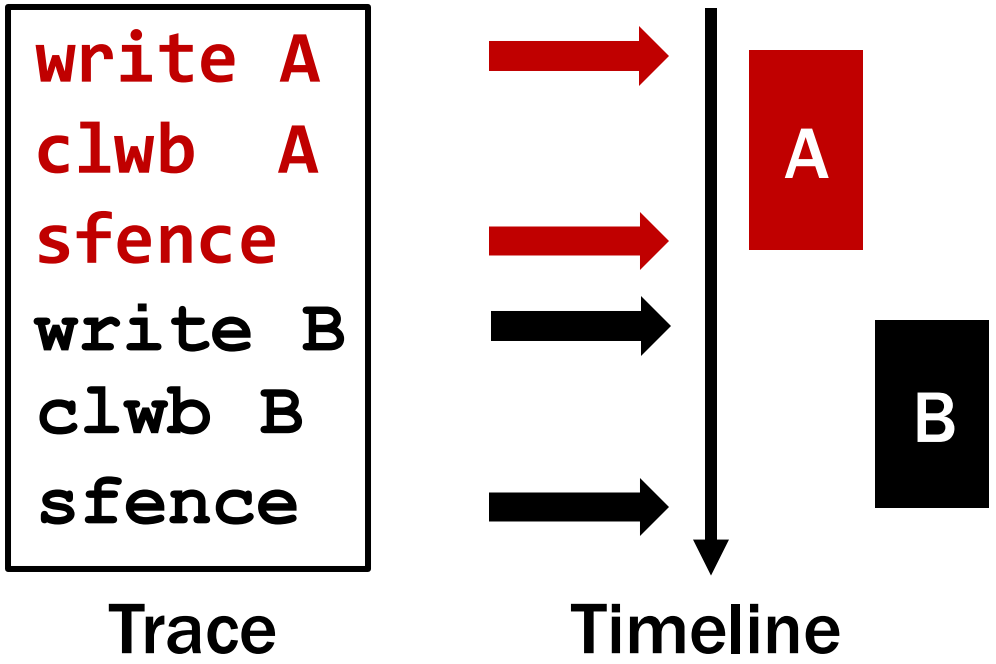
```
sfence
write C
write B
write A
...
sfence
```

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



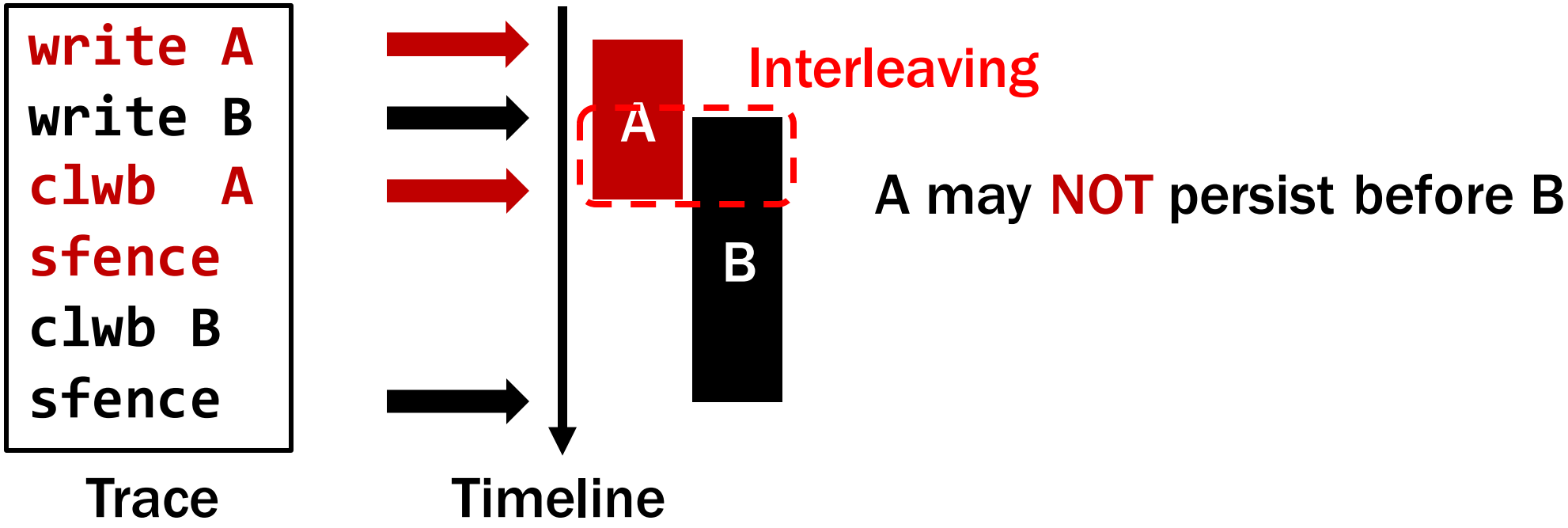
A persists before B

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

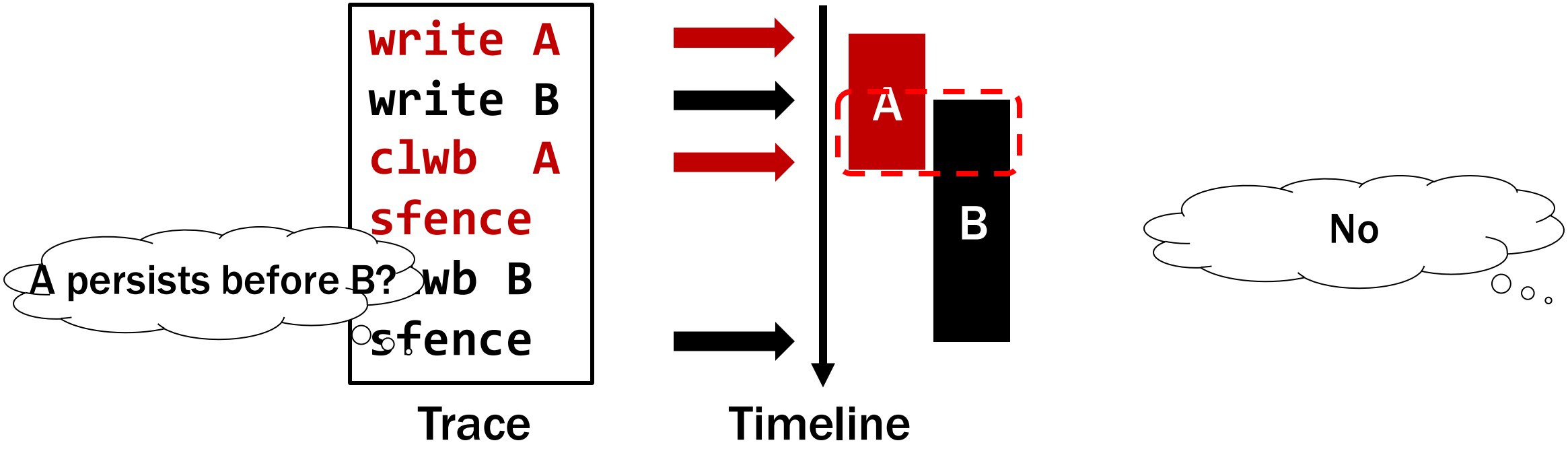


An overlapping interval indicates that there is a case where A does not persist before B

PMTest KEY IDEAS: FAST

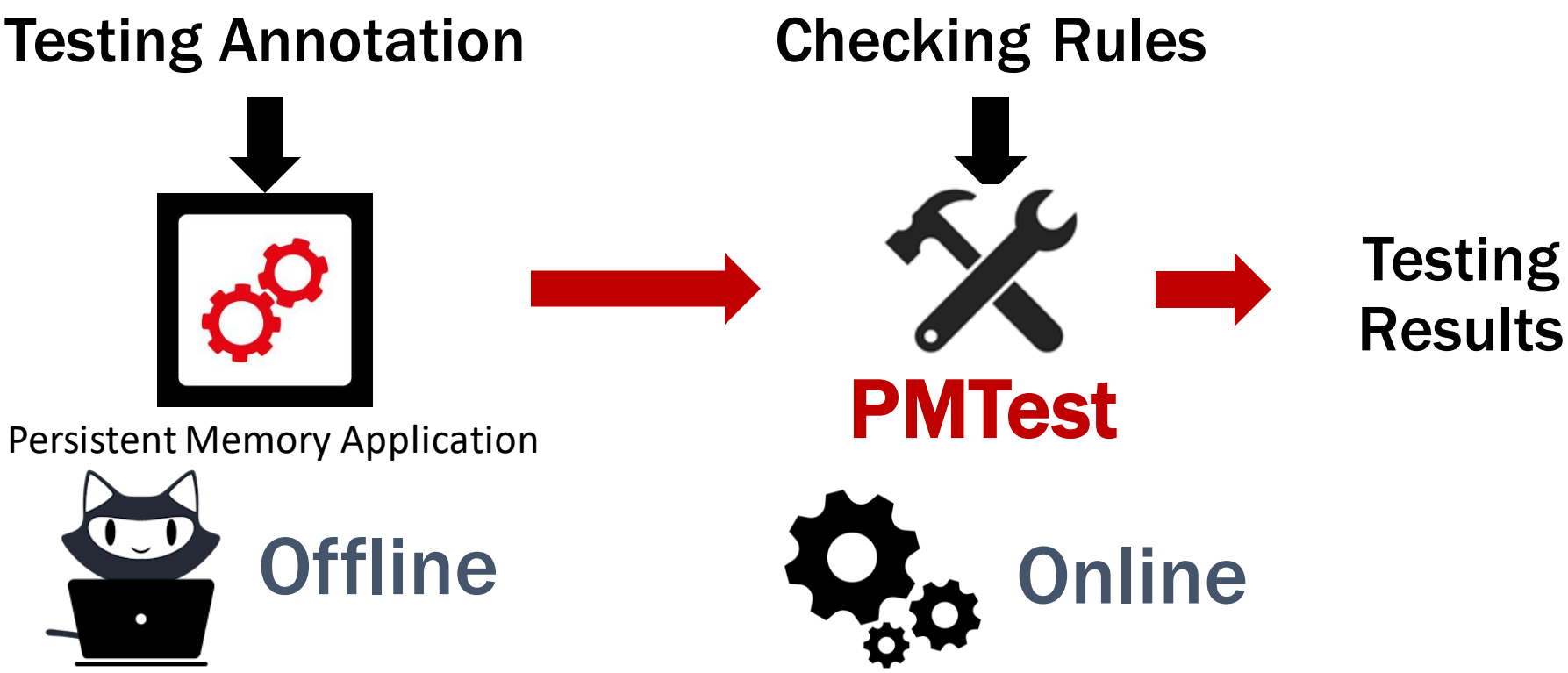
- PMTest infers the **persistence interval** from PM operation trace

  The interval in which a write can possibly become persistent



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

PMTest OVERVIEW



SUMMARY SO FAR

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

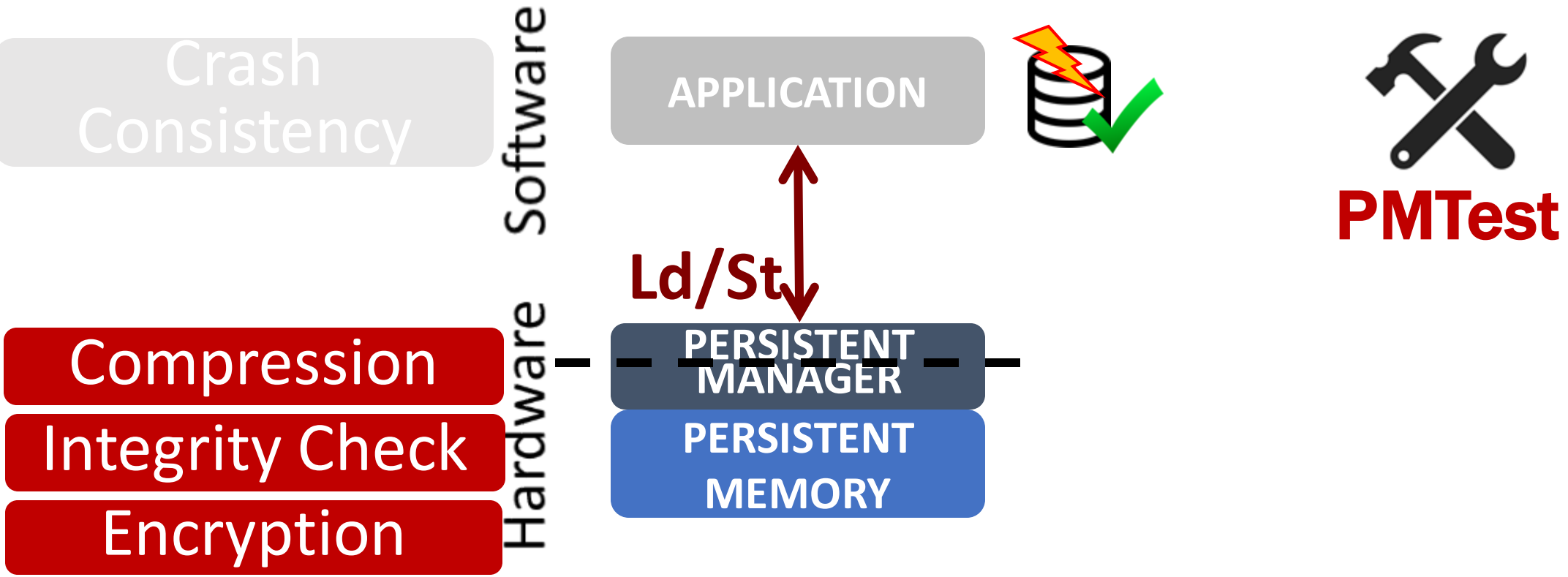
PMTest

pmtest.persistentmemory.org

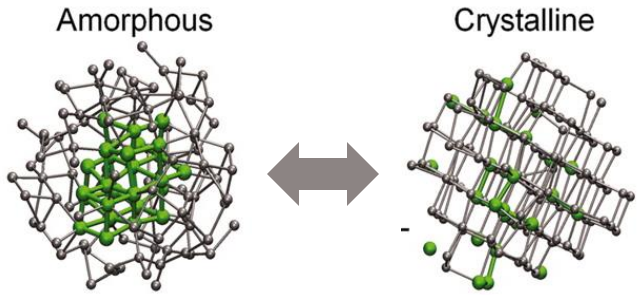
- Our tool PMTest is **fast** and **flexible**
  - Flexible: Supports kernel modules, custom PM programs, transaction-based programs
  - Fast: Incurs < 2X overhead in real-workload applications

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

CHALLENGE: MEMORY & STORAGE SYSTEM SUPPORT



Not the operating system,
hardware is responsible for many system support in PM



NON-VOLATILE MEMORY

PERSISTENT MEMORY

Unified Memory and Storage

Rethinking System Support

PMTEST: Testing for Correctness

ASPLOS'19

JANUS: Optimizing for Efficiency

ISCA'19

Conclusion

MEMORY AND STORAGE SUPPORT

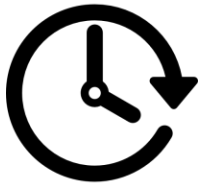
The memory and storage support is designed for

Security

Prevent attackers from stealing or tampering data
Encryption, integrity verification, etc.

Bandwidth

Improve NVM's limited bandwidth
Deduplication, compression, etc.

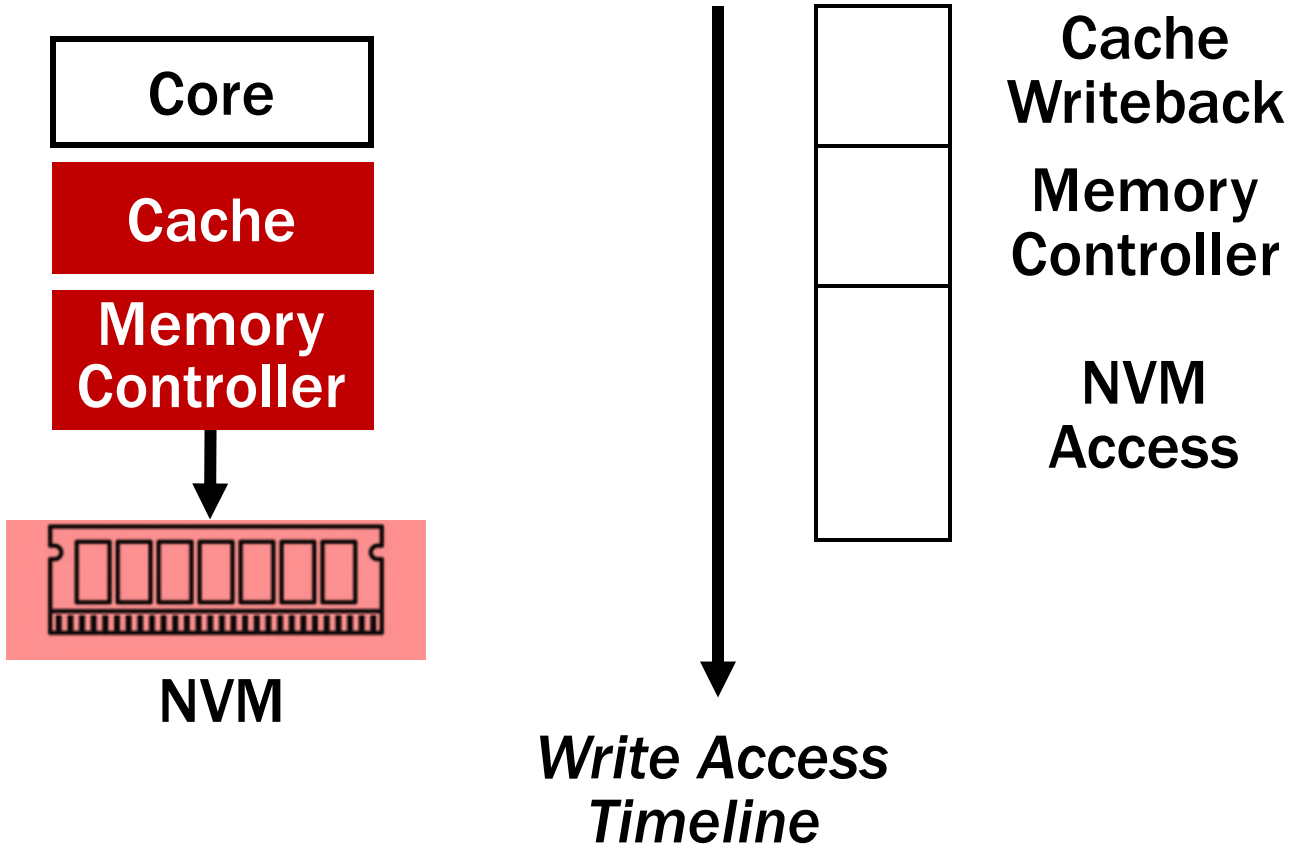


Endurance

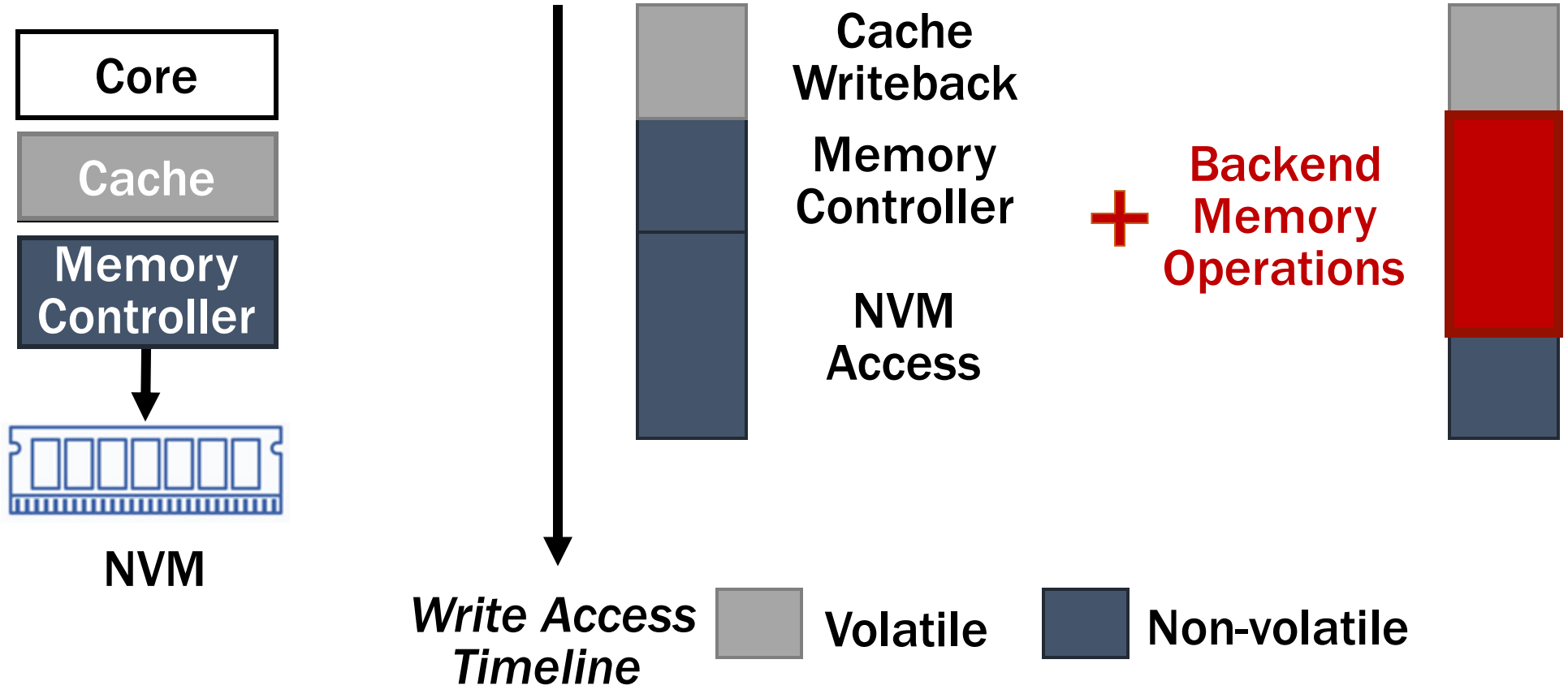
Extend NVM's limited lifetime
Wear-leveling, error correction, etc.

We refer to the memory and storage support as
backend memory operations

BACKEND MEMORY OPERATION LATENCY

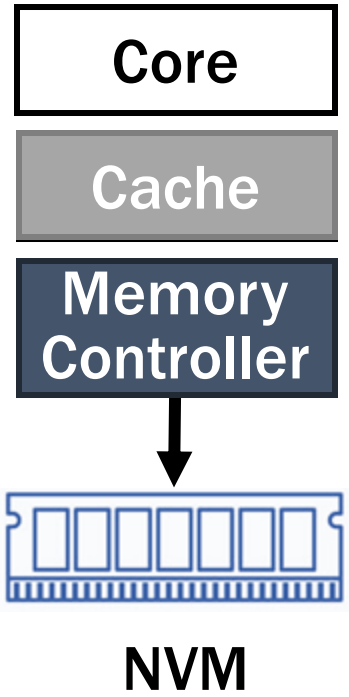


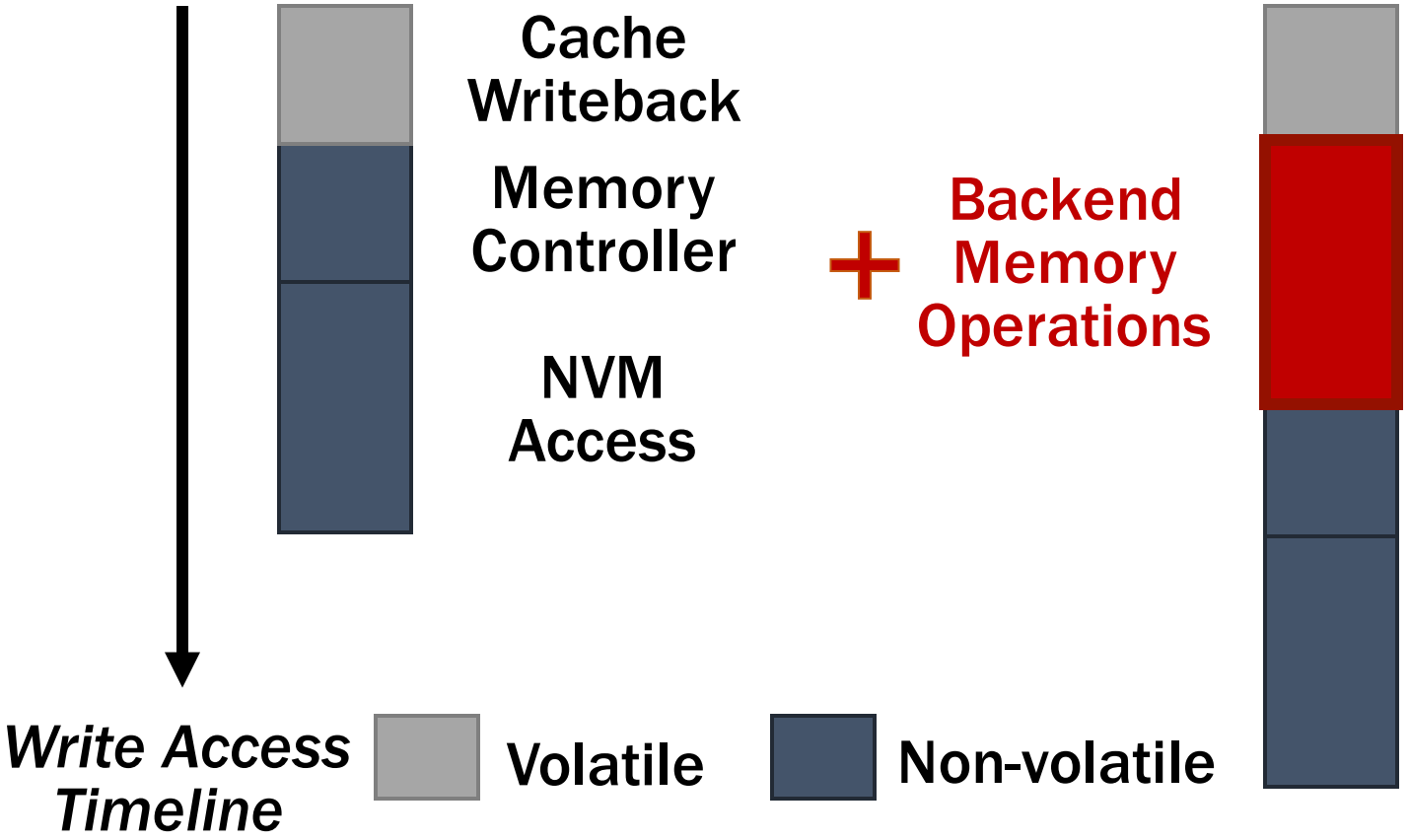
BACKEND MEMORY OPERATION LATENCY



Recent NVM support guarantees writes accepted by memory controller is non-volatile

BACKEND MEMORY OPERATION LATENCY





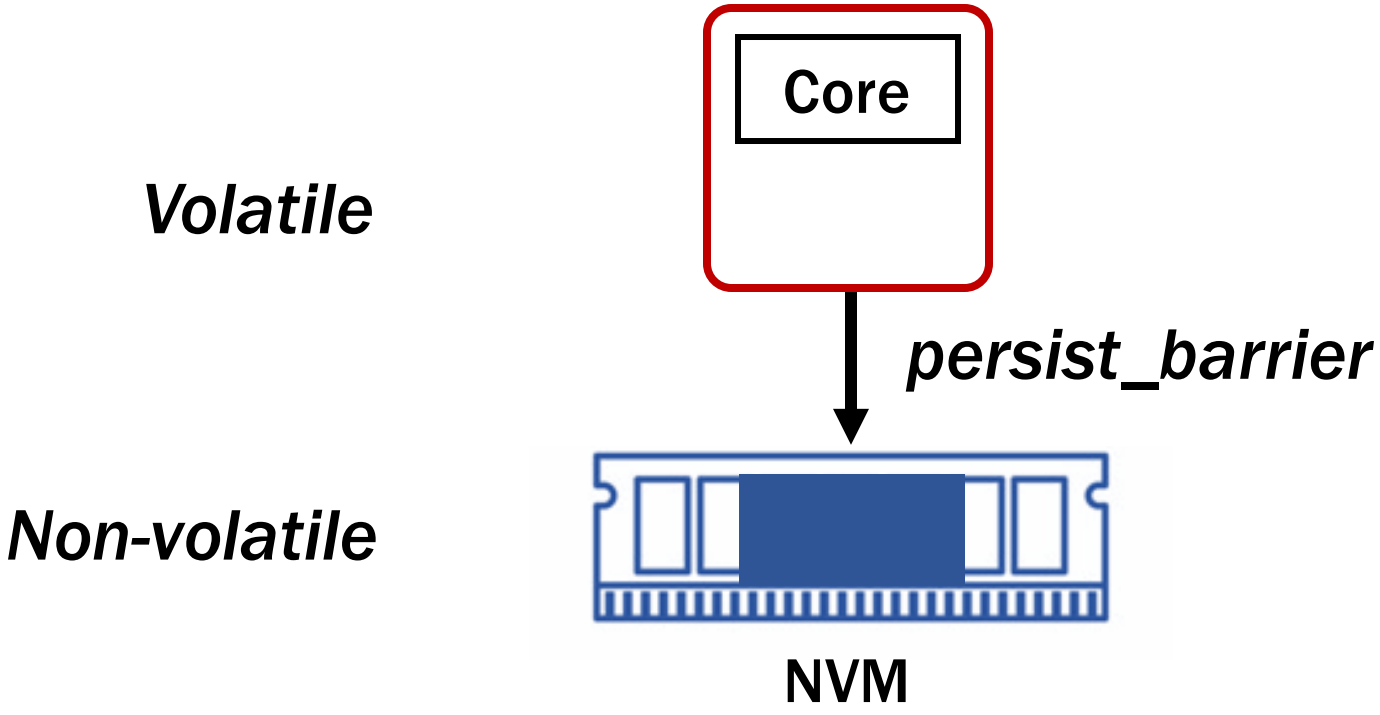
Latency to Persistence

~15 ns

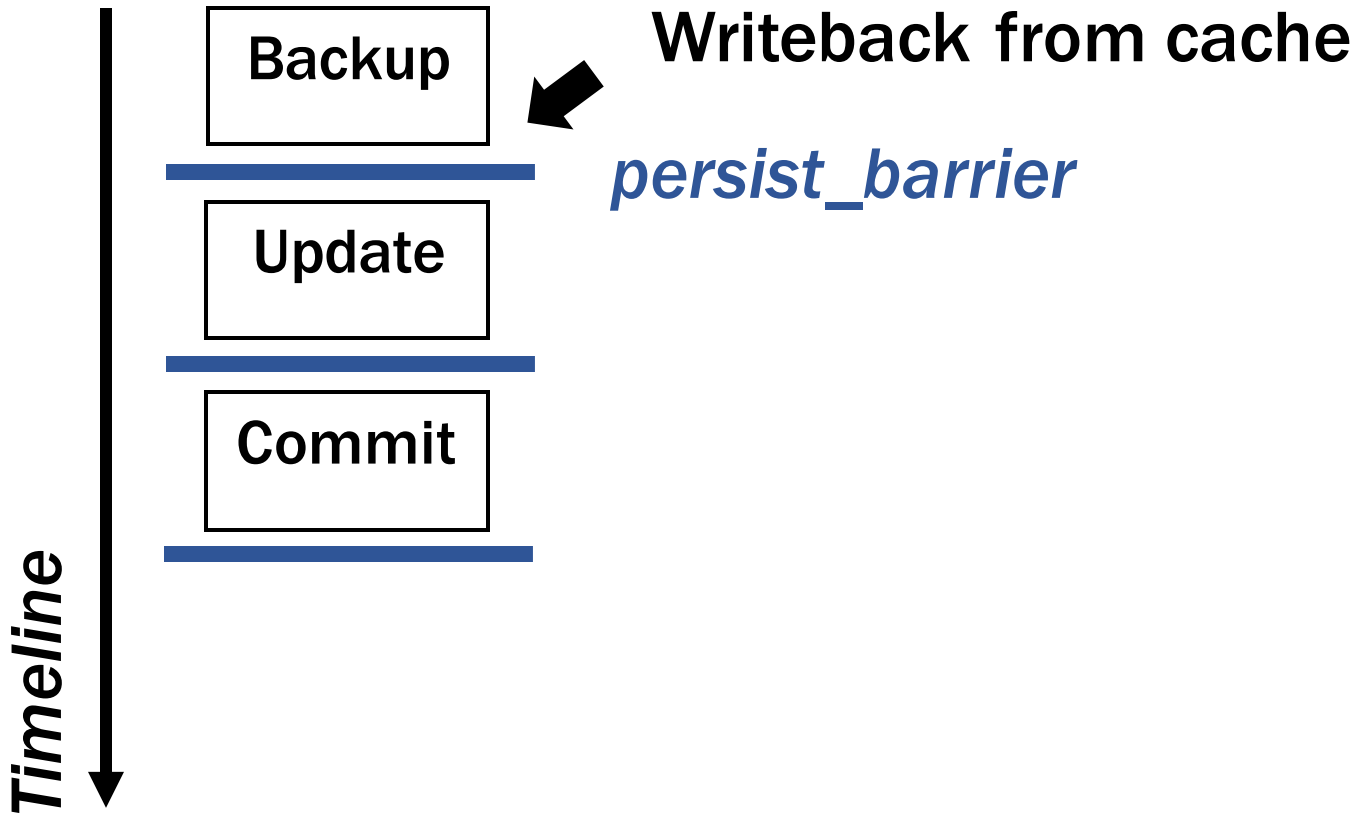
>100 ns

WHY WRITE LATENCY IS IMPORTANT?

- NVM programs need to use **crash consistency mechanisms** that enforces data writeback



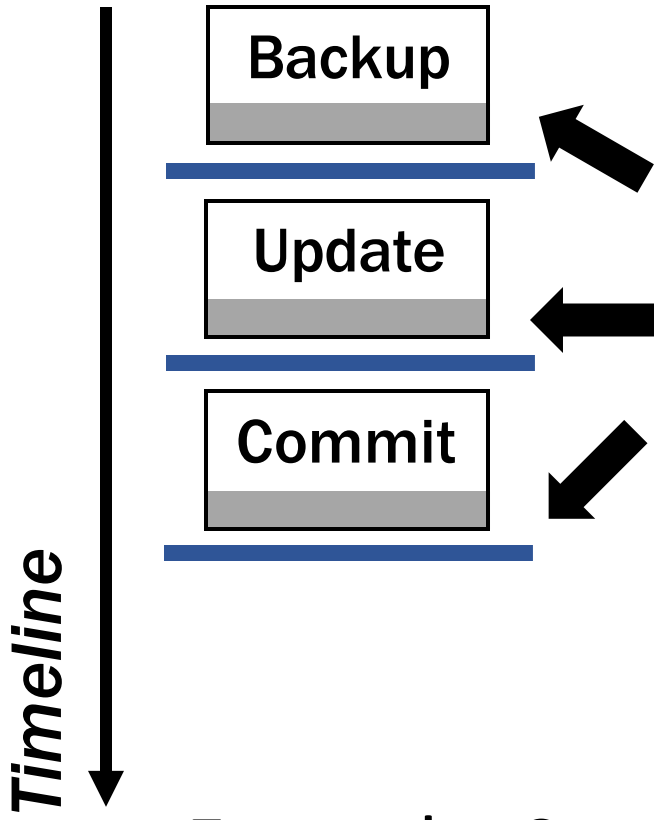
WRITE LATENCY IN NVM PROGRAMS



Example: Steps in undo logging transaction

Execution cannot continue until writeback completes

WRITE LATENCY IN NVM PROGRAMS

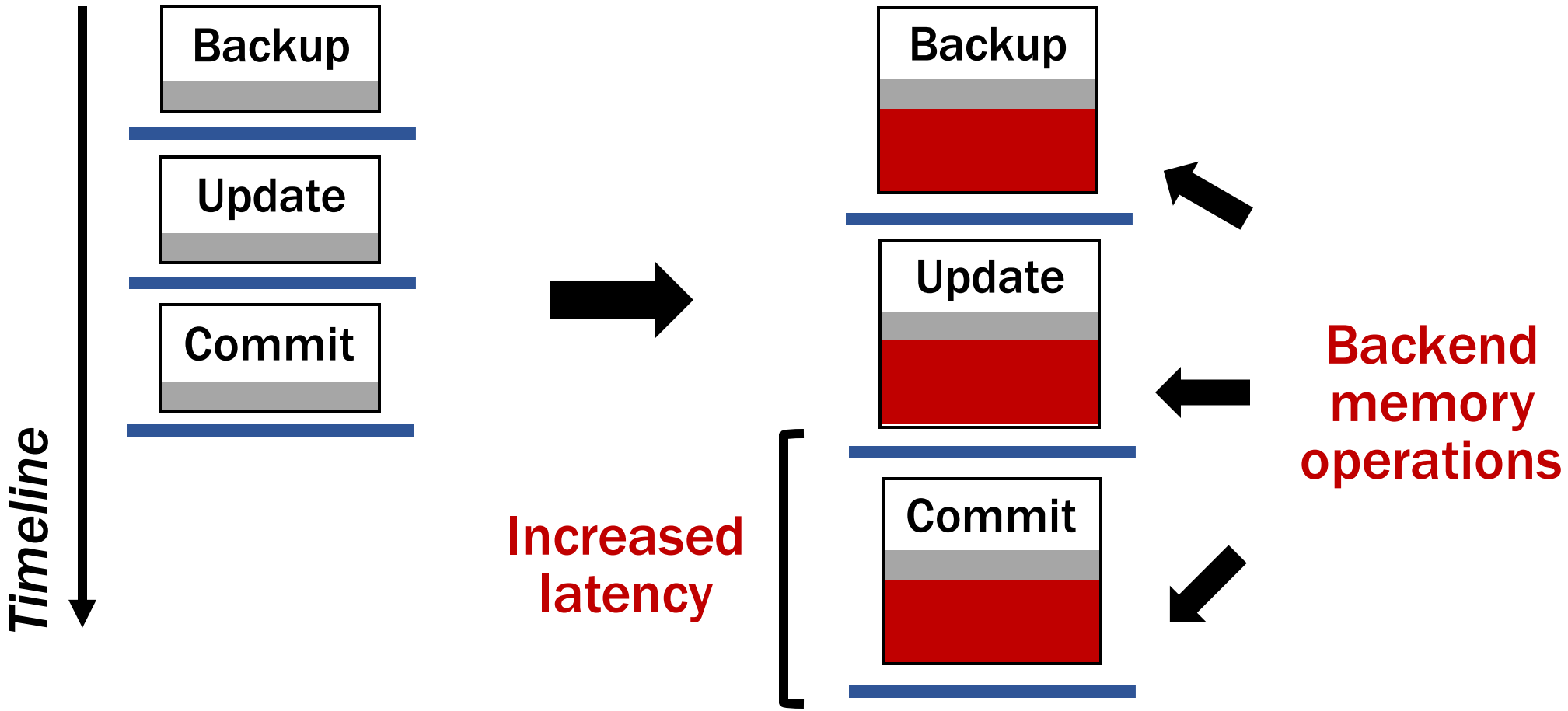


Write latency is on the critical path

Example: Steps in undo logging transaction

Crash consistency mechanism puts **write latency** on the **critical path**

WRITE LATENCY IN NVM PROGRAMS



Backend memory operations increase the writeback latency

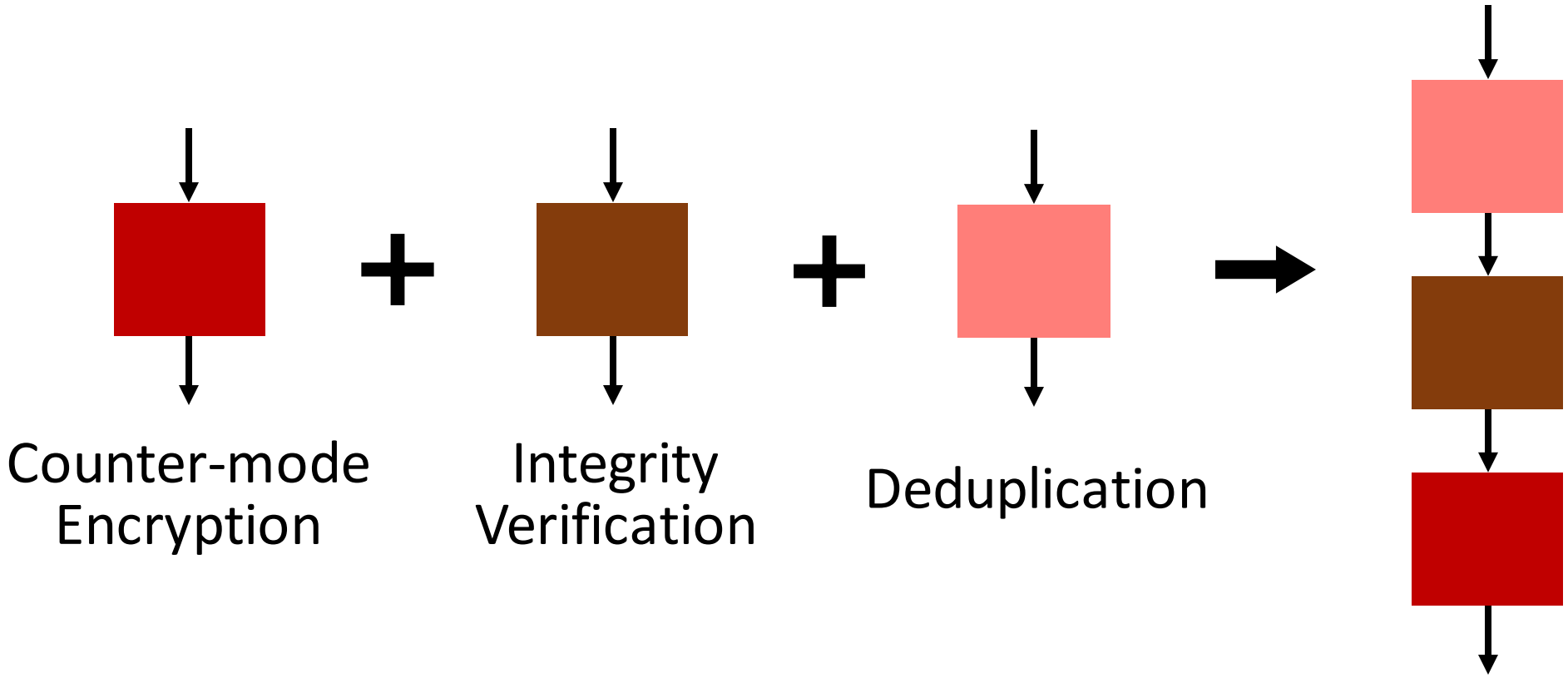
Backend memory operations are on the **critical path**

How to **reduce** the latency?

OBSERVATION

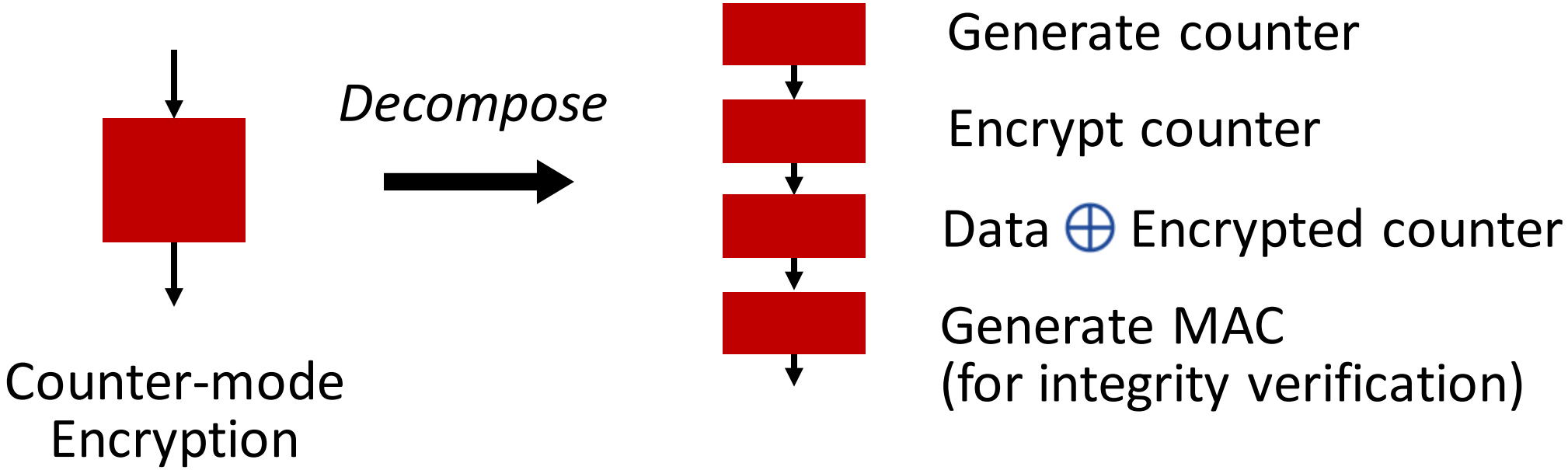
Each backend memory operation seems **indivisible**

➡ Integration leads to **serialized** operations



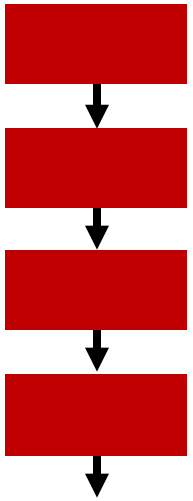
OBSERVATION

However, it is possible to decompose them into sub-operations

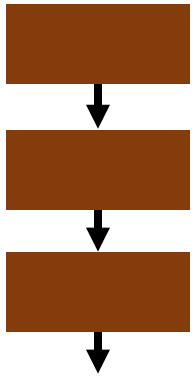


KEY IDEA I: PARALLELIZATION

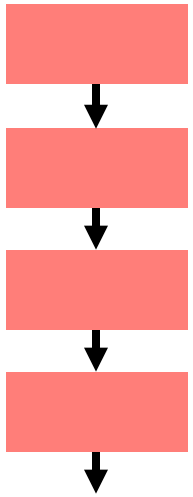
After **decomposing** the example operations:



Counter-mode Encryption



Integrity Verification



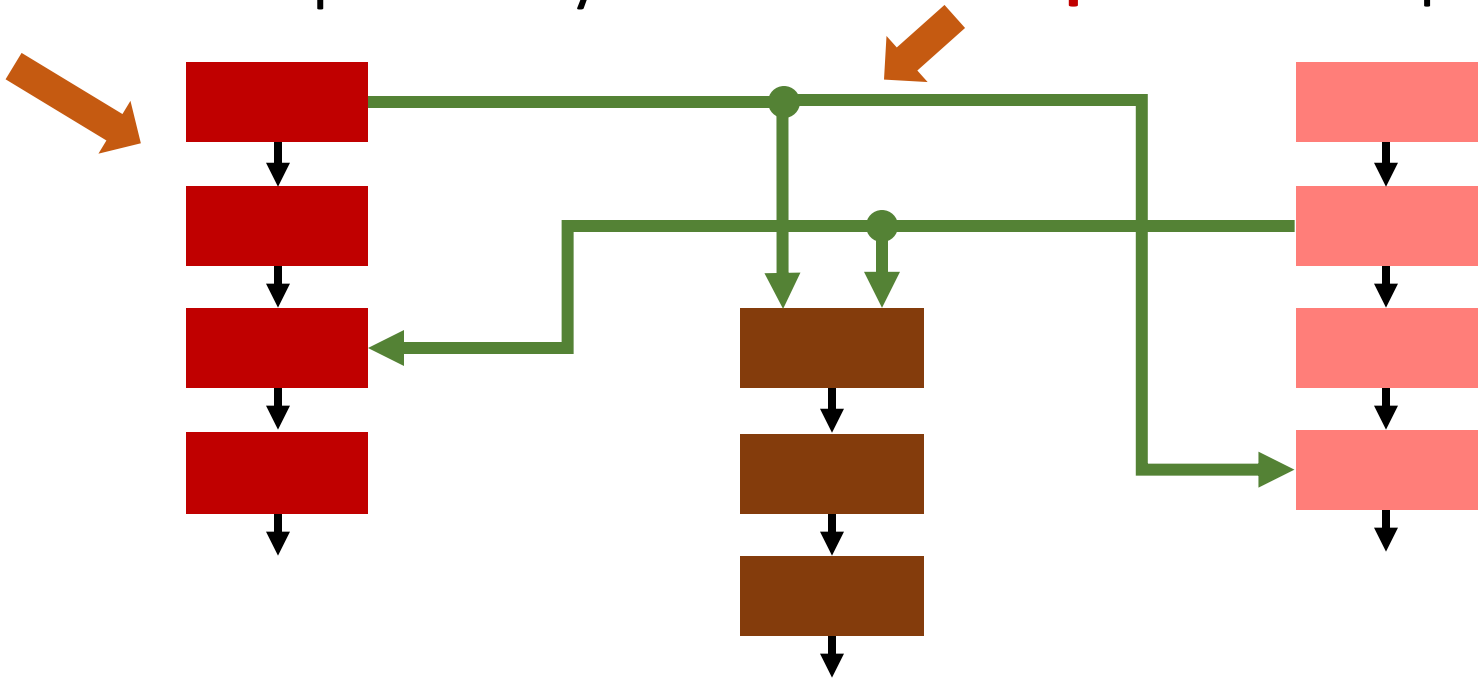
Deduplication

KEY IDEA I: PARALLELIZATION

There are two types of dependencies:

Intra-operation dependency

Inter-operation dependency



1. Dependency within each operation

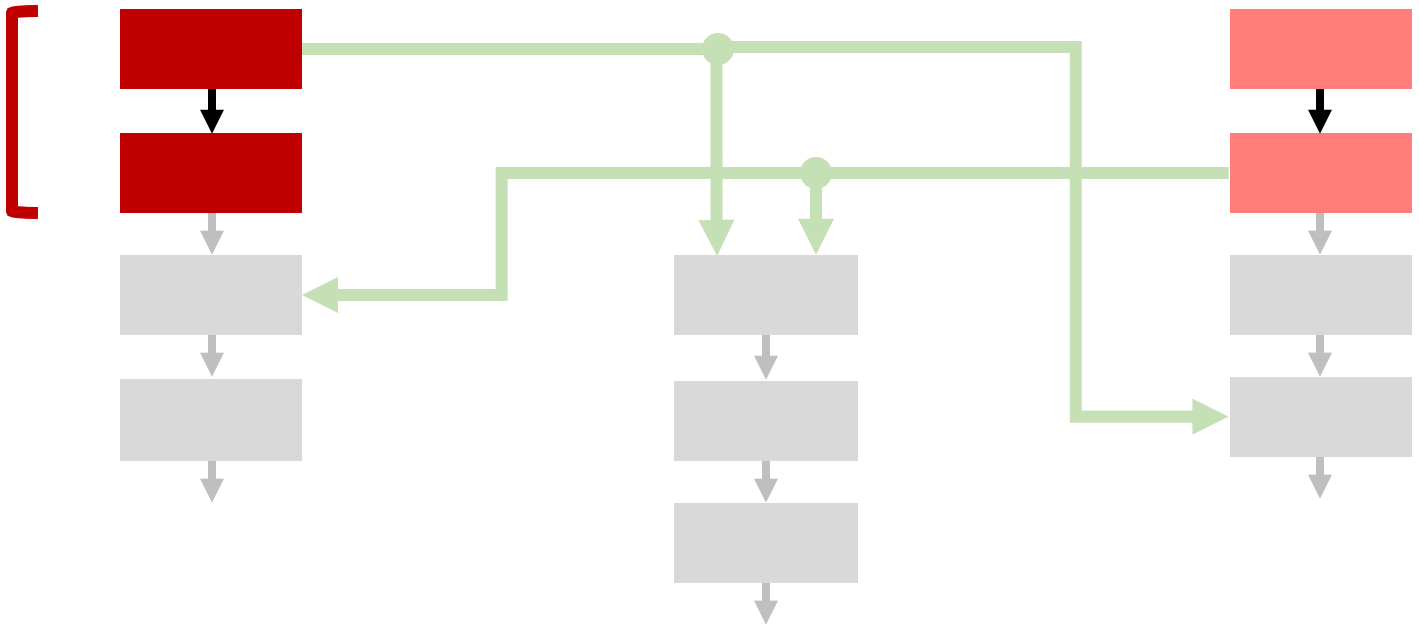
KEY IDEA I: PARALLELIZATION

There are two types of dependencies:

Intra-operation dependency

Inter-operation dependency

Parallelizable



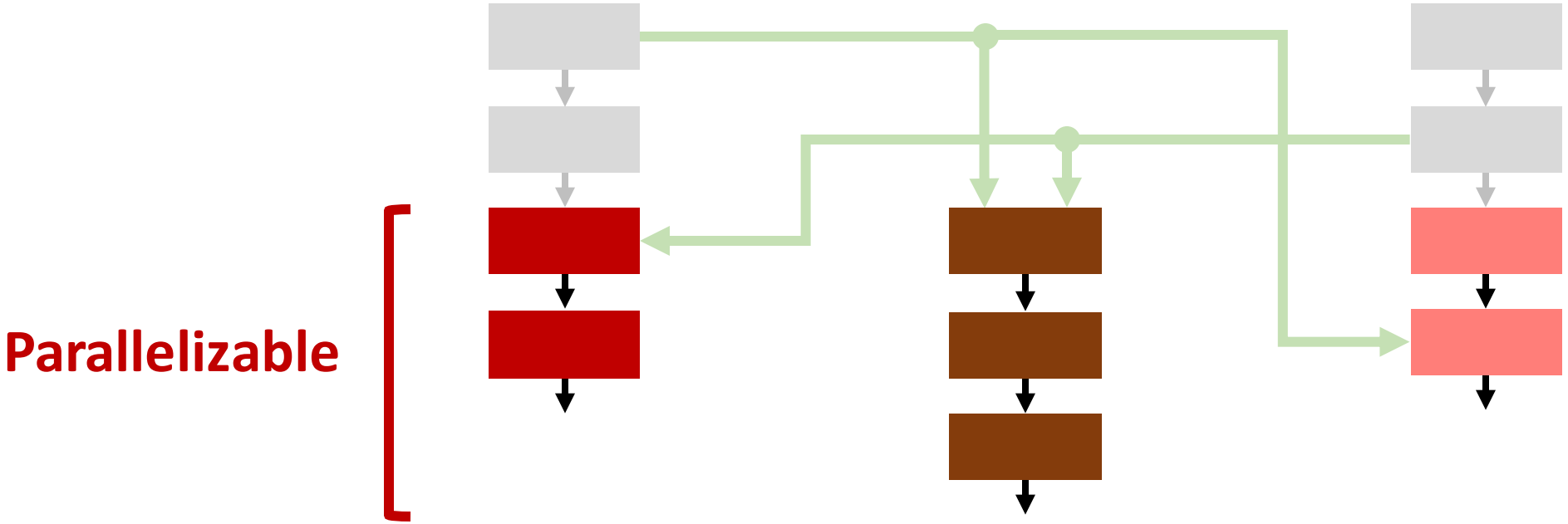
Sub-operations without dependency can execute **in parallel**

KEY IDEA I: PARALLELIZATION

There are two types of dependencies:

Intra-operation dependency

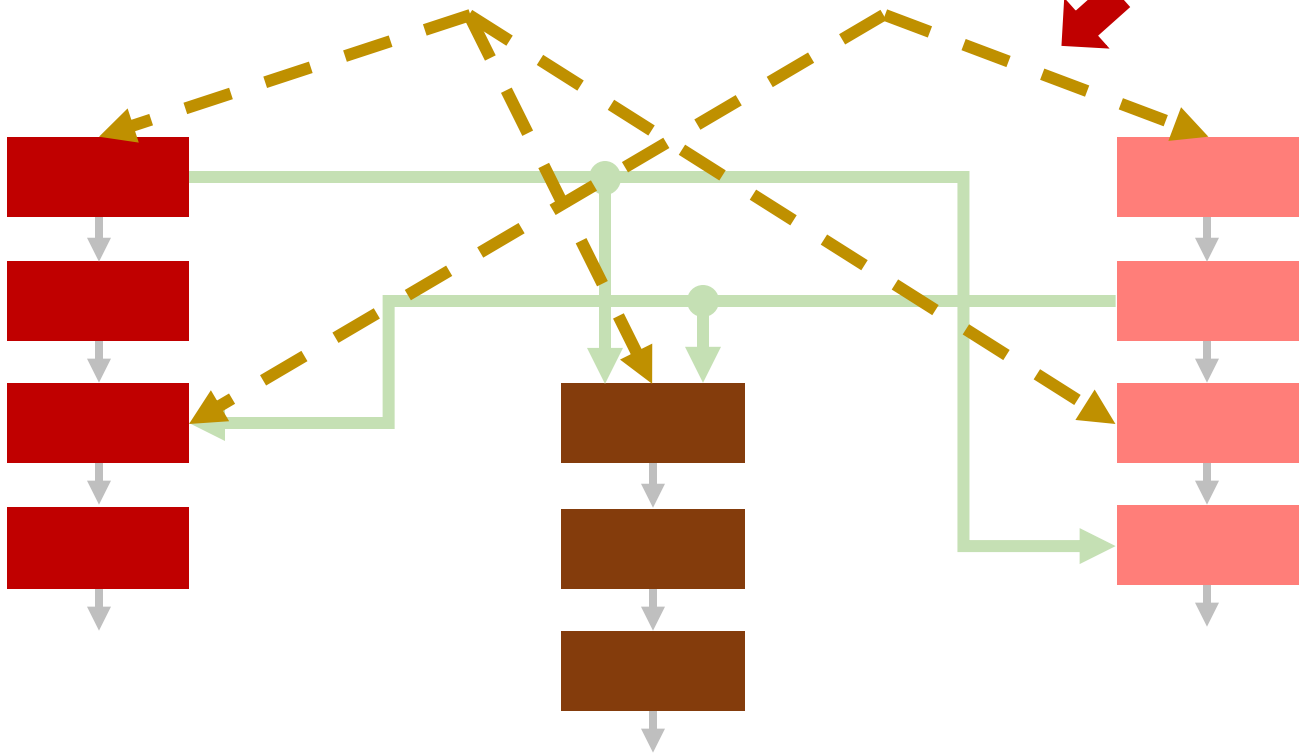
Inter-operation dependency



Sub-operations without dependency can execute **in parallel**

KEY IDEA II: PRE-EXECUTION

A write consists of: Address Data **External** dependency

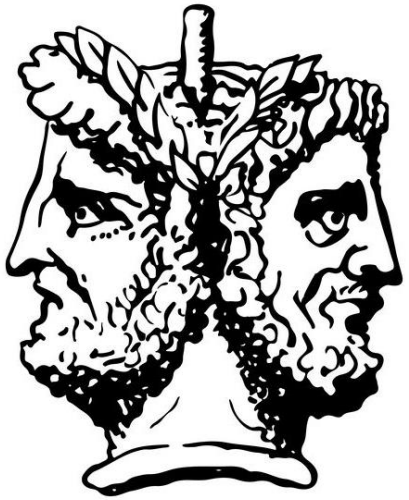


Sub-operations can **pre-execute**
as soon as their data/address dependency is resolved

OUR PROPOSAL: JANUS

Janus is a Roman god with two faces:
one looks into the **past**, and another into the **future**

When dependent data and address become available



Pre-execute operations with dependency resolved

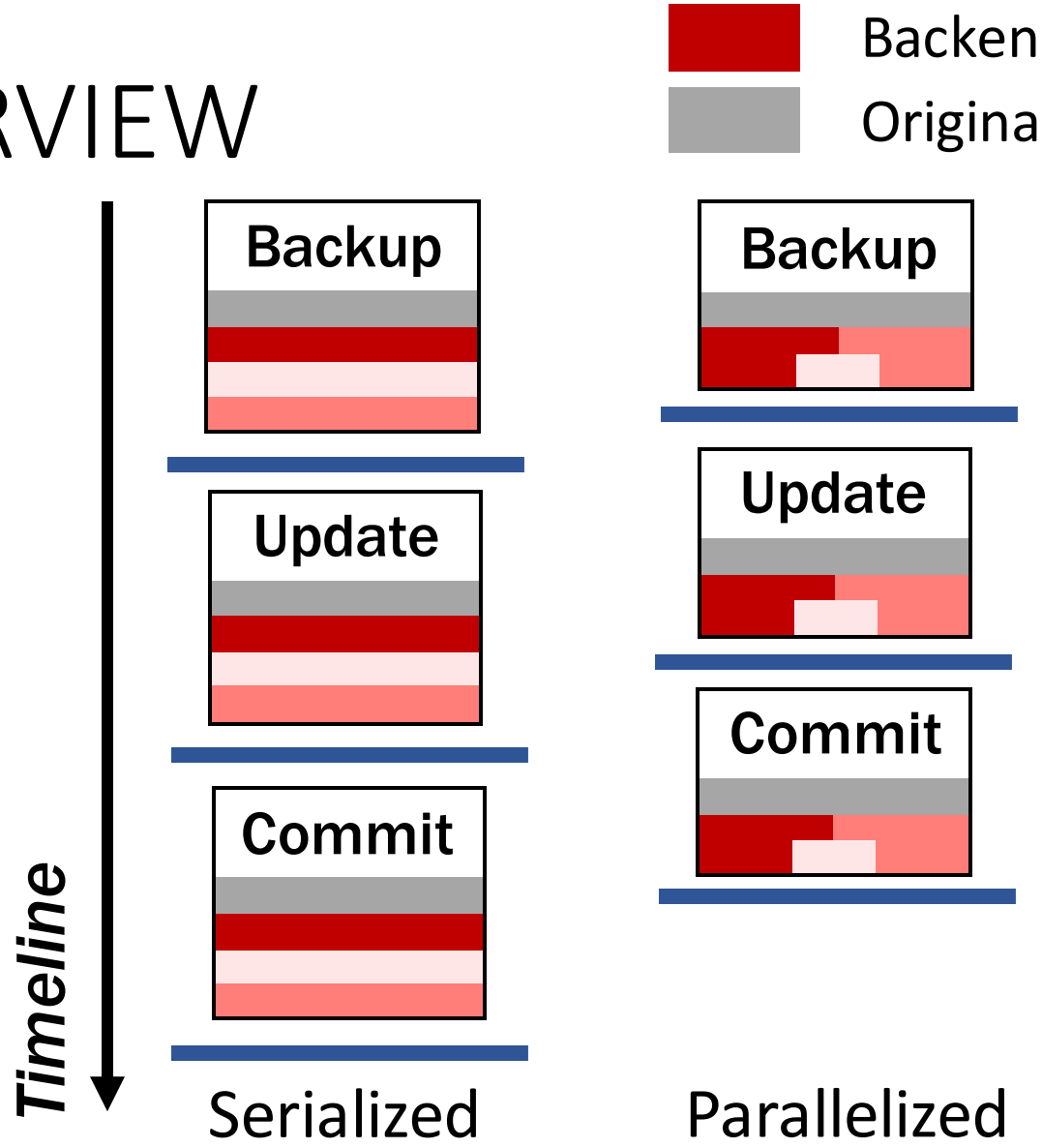
Past

Future

JANUS OVERVIEW

Janus:

- Parallelization

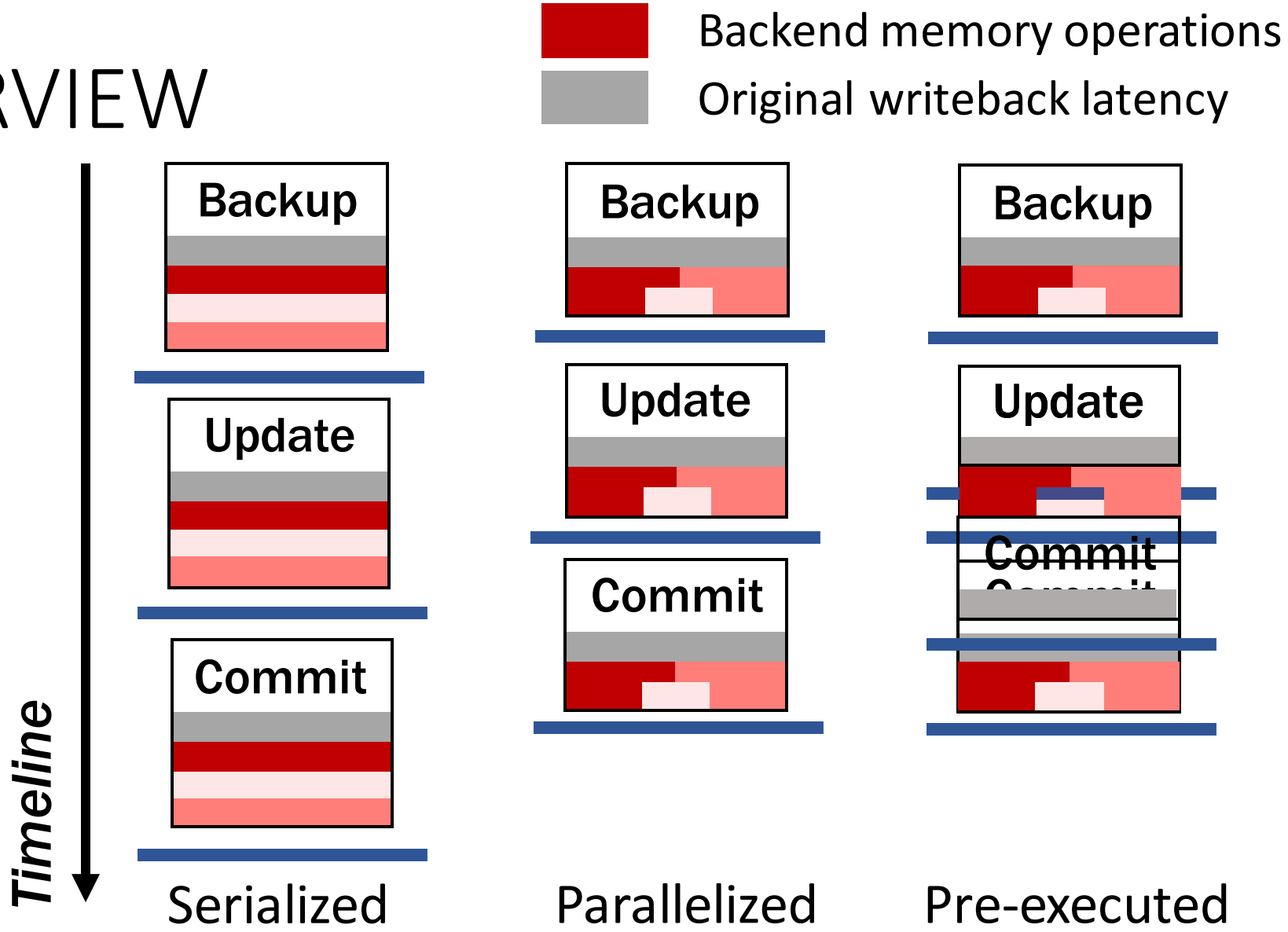


Parallelization **reduces** the latency of each operation

JANUS OVERVIEW

Janus:

- Parallelization
- Pre-execution



Pre-execution moves their latency **off the critical path**

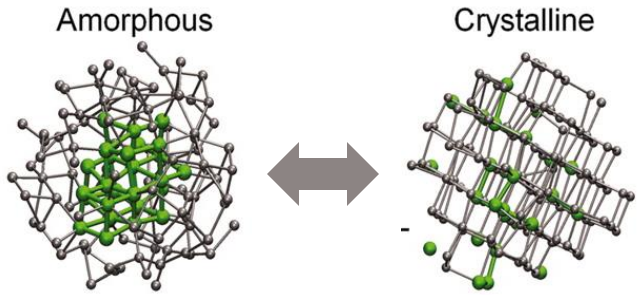
PERFORMANCE

- Janus provides a **software interface** to issue pre-execution
- Compared to baseline with **serialized** operations:

Janus

Manual: **2.35X speedup**

Automated: **2X speedup**



NON-VOLATILE MEMORY

PERSISTENT MEMORY

Unified Memory and Storage

Rethinking System Support

PMTEST: Testing for Correctness

ASPLOS'19

JANUS: Optimizing for Efficiency

ISCA'19

Conclusion

NEEDS STORAGE AND MEMORY SYSTEM SUPPORT

Crash Consistency

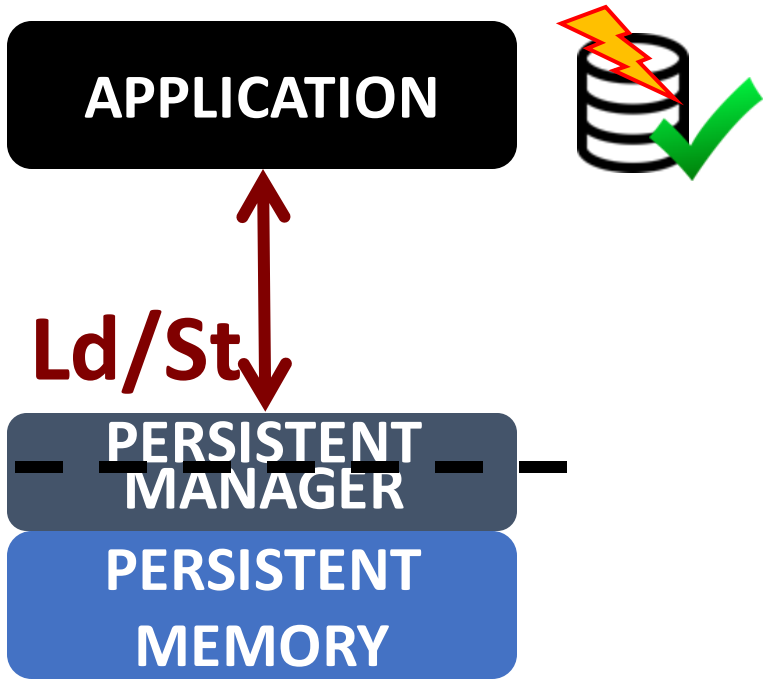
Software

Hardware

Compression

Integrity check

Encryption



PMTest

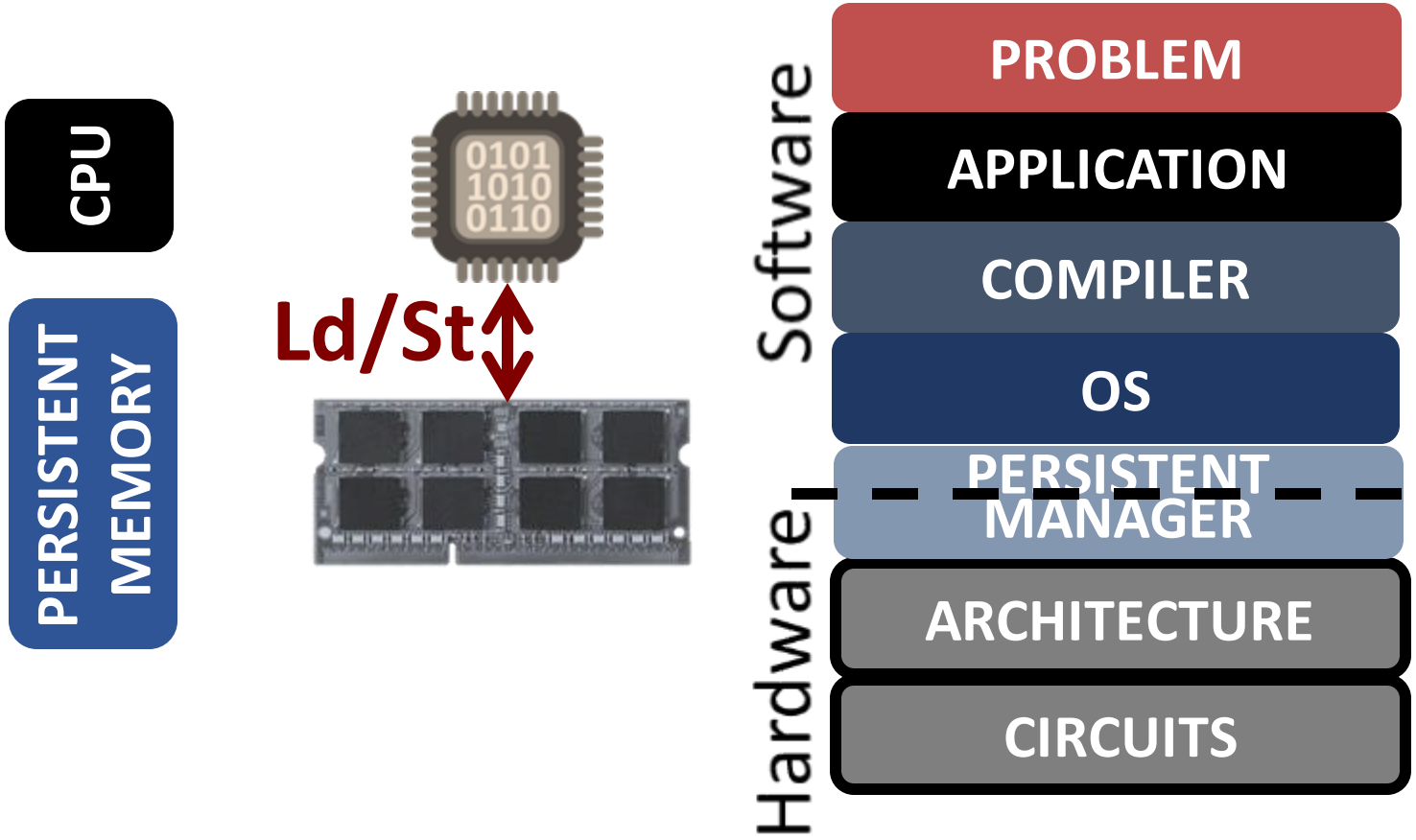
pmtest.persistentmemory.org

Janus

PMTest focuses on correctness of persistent memory applications

Janus focuses on reducing the overhead of system support

GOAL: END-TO-END SYSTEM FOR PERSISTENT MEMORY



How to write consistent code?

How to test the code is correct?

How to recover and resume application and OS?

How to provide efficient hardware support?

A full stack support for persistent memory applications
Many directions to explore!

Rethinking System Support for Persistent Memory

Samira Khan

University of Virginia