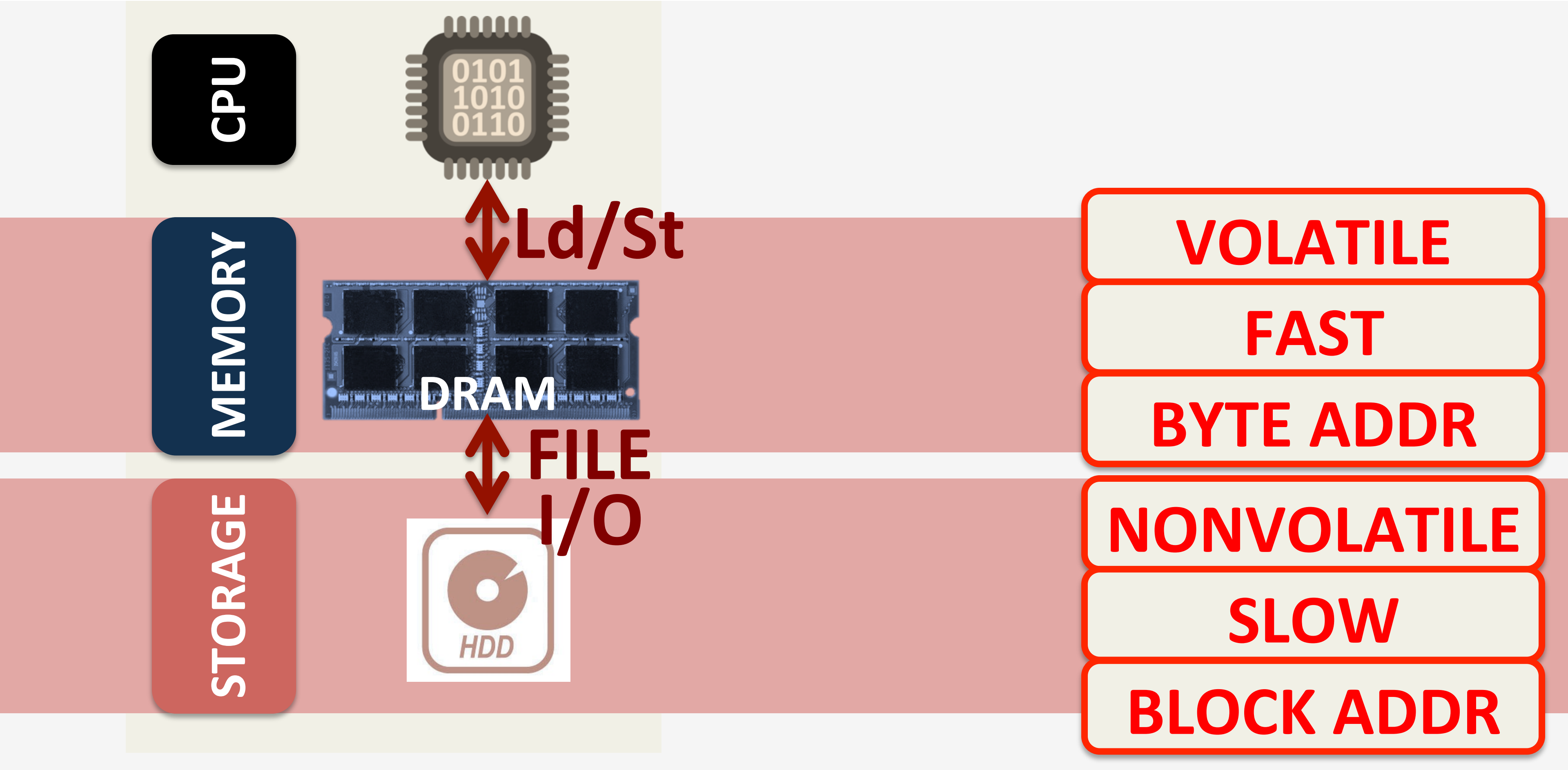
ThyNVM

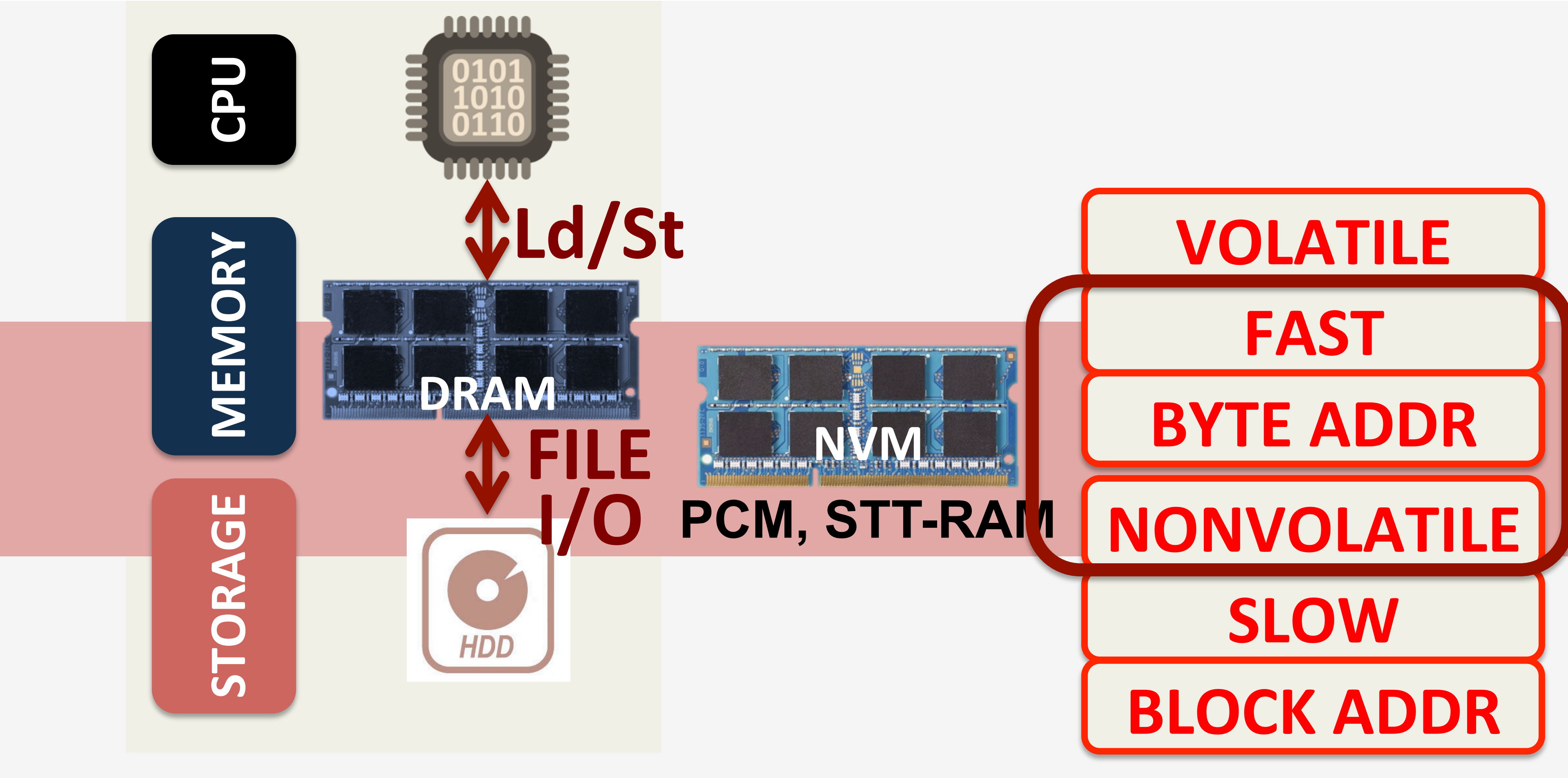
Enabling Software-Transparent Crash Consistency In Persistent Memory Systems

Jinglei Ren, Jishen Zhao, **Samira Khan**, Jongmoo Choi, Yongwei Wu, and Onur Mutlu

TWO-LEVEL STORAGE MODEL

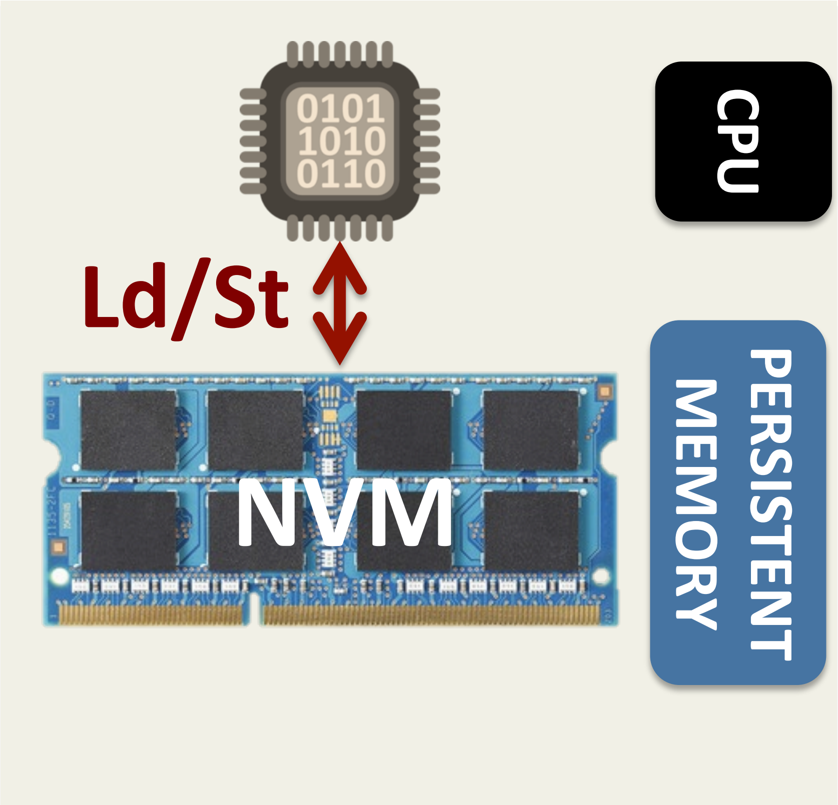


TWO-LEVEL STORAGE MODEL



Non-volatile memories combine characteristics of memory and storage

PERSISTENT MEMORY



Provides an opportunity to manipulate persistent data directly

CHALLENGE: CRASH CONSISTENCY

Persistent Memory System

System crash can result in permanent data corruption in NVM

CURRENT SOLUTIONS

Explicit interfaces to manage consistency

- NV-Heaps [ASPLOS'11], BPFS [SOSP'09], Mnemosyne [ASPLOS'11]

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

Limits adoption of NVM

Have to rewrite code with clear partition between volatile and non-volatile data

Burden on the programmers

OUR APPROACH: ThyNVM

Goal:
Software transparent consistency in persistent memory systems

ThyNVM: Summary

A new hardware-based checkpointing mechanism

- **Checkpoints** at *multiple granularities* to reduce both checkpointing latency and metadata overhead
- **Overlaps** *checkpointing* and *execution to* reduce checkpointing latency
- **Adapts** to *DRAM and NVM* characteristics

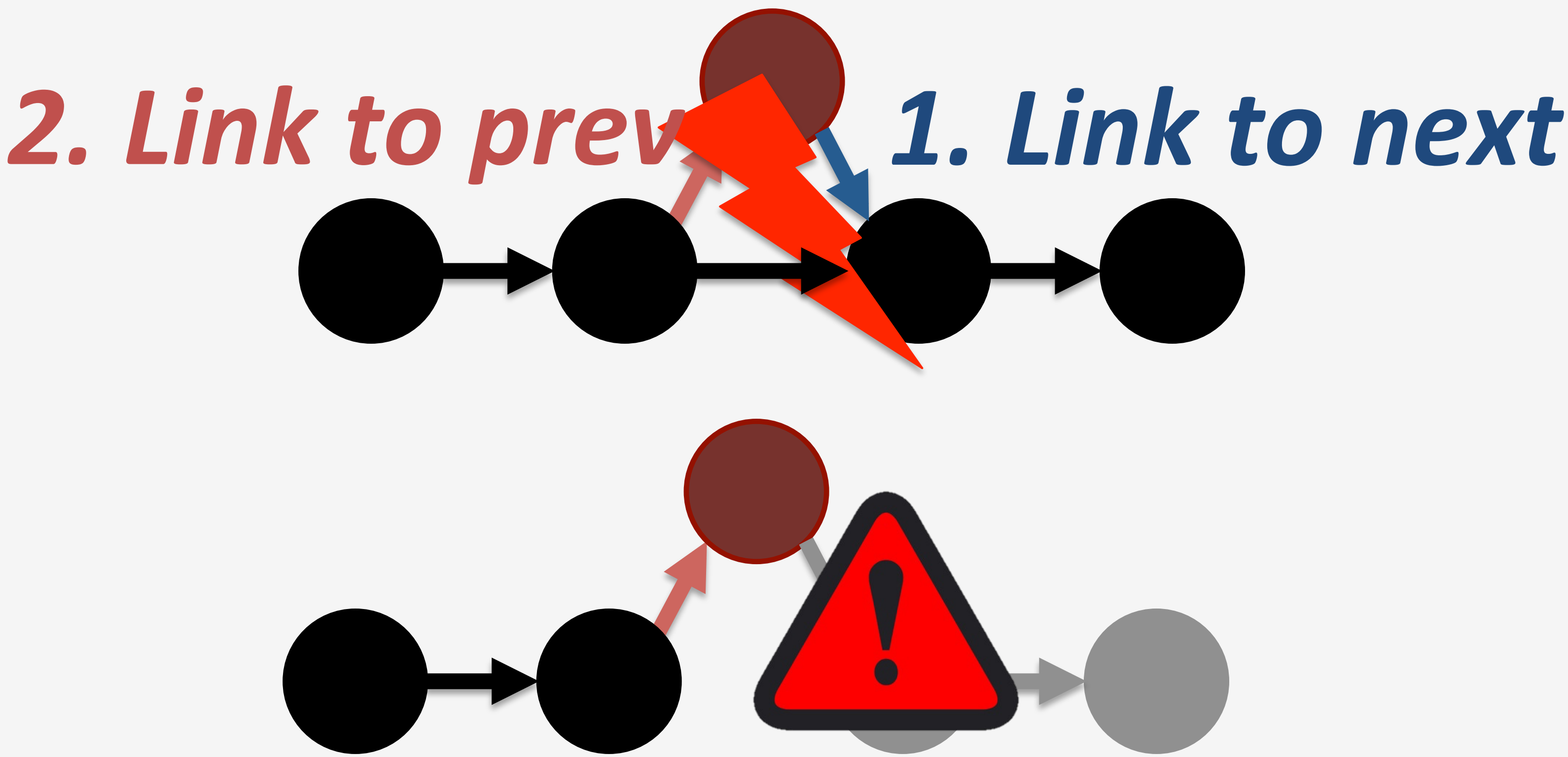
Performs within 4.9% of an *idealized DRAM* with zero cost consistency

OUTLINE

- Crash Consistency Problem
- Current Solutions
- ThyNVM
- Evaluation
- Conclusion

CRASH CONSISTENCY PROBLEM

Add a node to a linked list



System crash can result in inconsistent memory state

OUTLINE

- Crash Consistency Problem
- **Current Solutions**
- ThyNVM
- Evaluation
- Conclusion

CURRENT SOLUTIONS

Explicit interfaces to manage consistency

- **NV-Heaps** [ASPLOS'11], **BPFS** [SOSP'09], **Mnemosyne** [ASPLOS'11]

Example Code

update a node in a persistent hash table

```
void hashtable_update(hashtable_t* ht,
                      void *key, void *data)
{
   list_t* chain = get_chain(ht, key);
   pair_t* pair;
   pair_t updatePair;
   updatePair.first = key;
   pair = (pair_t*) list_find(chain,
                         &updatePair);
   pair->second = data;
}
```

CURRENT SOLUTIONS

```
void TMhashtable_update(TMARCGDECL
hashtable_t* ht, void *key, void*data)
{
  list_t* chain = get_chain(ht, key);
  pair_t* pair;
  pair_t updatePair;
  updatePair.first = key;
  pair = (pair_t*) TMLIST_FIND(chain,
                         &updatePair);
  pair->second = data;
}
```

CURRENT SOLUTIONS

Manual declaration of persistent components

```
void TMhashtable_update(TMARCGDECL
hashtable_t* ht, void *key, void*data)
{
   list_t* chain = get_chain(ht, key);
   pair_t* pair;
   pair_t updatePair;
   updatePair.first = key;
   pair = (pair_t*) TMLIST_FIND(chain,
                        &updatePair);
   pair->second = data;
}
```

CURRENT SOLUTIONS

Manual declaration of persistent components

```
void TMhashtable_update(TMARCGDECL
hashtable_t* ht, void *key, void*data)
{
   list_t* chain = get_chain(ht, key);
   pair_t* pair;
   pair_t updatePair;
   updatePair.first = key;
   pair = (pair_t*) TMLIST_FIND(chain,
                          &updatePair);
   pair->second = data;
}
```

Need a new implementation

CURRENT SOLUTIONS

Manual declaration of persistent components

```
void TMhashtable_update(TMARCGDECL
hashtable_t* ht, void *key, void*data)
{
   list_t* chain = get_chain(ht, key);
   pair_t* pair;
   pair_t updatePair;
   updatePair.first = key;
   pair = (pair_t*) TMLIST_FIND(chain,
                          &updatePair);
   pair->second = data;
}
```

Need a new implementation

Third party code can be inconsistent

CURRENT SOLUTIONS

Manual declaration of persistent components

```
void TMhashtable_update(TMARCGDECL
hashtable_t* ht, void *key, void*data)
{
   list_t* chain = get_chain(ht, key);
   pair_t* pair;
   pair_t updatePair;
   updatePair.first = key;
   pair = (pair_t*) TMLIST_FIND(chain,
                         &updatePair);
   pair->second = data;
}
```

Need a new implementation

Prohibited Operation (=)

Third party code can be inconsistent

Burden on the programmers

OUTLINE

Crash Consistency Problem

Current Solutions

ThyNVM

Evaluation

Conclusion

OUR GOAL

Software transparent consistency in persistent memory systems

- **Execute** *legacy applications*
- **Reduce burden** *on programmers*
- **Enable** *easier integration of NVM*

NO MODIFICATION IN THE CODE

```
void hashtable_update(hashtable_t* ht,
                      void *key, void *data)
{
list_t* chain = get_chain(ht, key);
pair_t* pair;
pair_t updatePair;
updatePair.first = key;
pair = (pair_t*) list_find(chain,
                           &updatePair);
pair->second = data;
}
```

RUN THE EXACT SAME CODE…

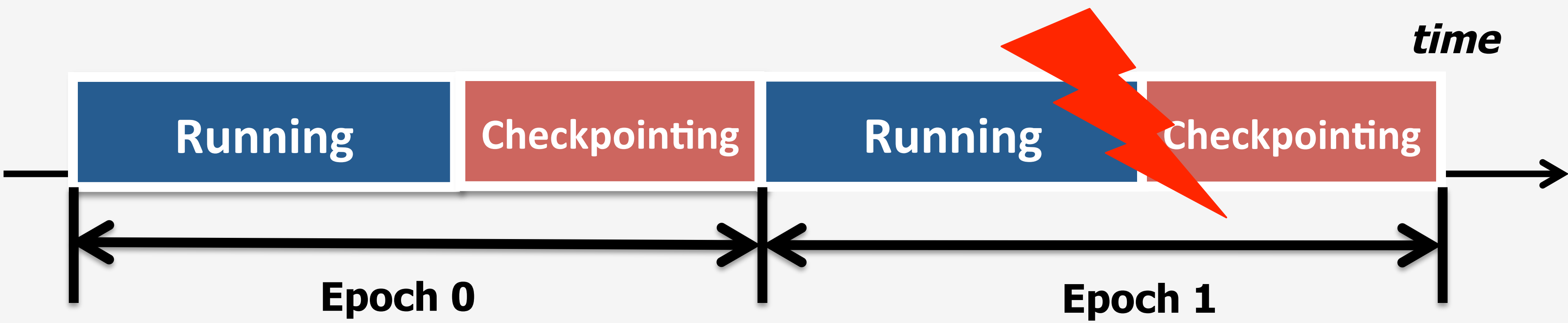
Persistent Memory System

```
void hashtable_update(hashtable_t* ht,
                      void *key, void *data){
  list_t* chain = get_chain(ht, key);
  pair_t* pair;
  pair_t updatePair;
  updatePair.first = key;
  pair = (pair_t*) list_find(chain,
                             &updatePair);
  pair->second = data;
}
```

Software transparent memory crash consistency

ThyNVM APPROACH

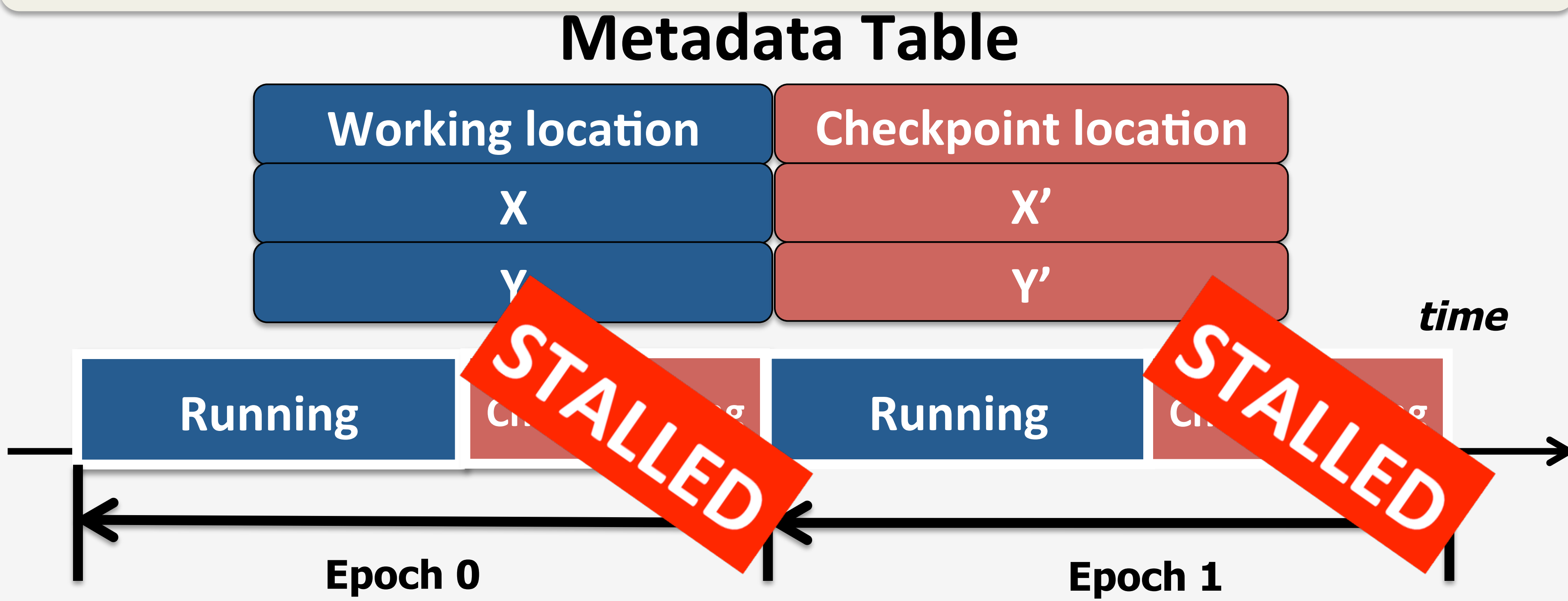
Periodic checkpointing of data managed by hardware



Transparent to application and system

CHECKPOINTING OVERHEAD

1. Metadata overhead



2. Checkpointing latency

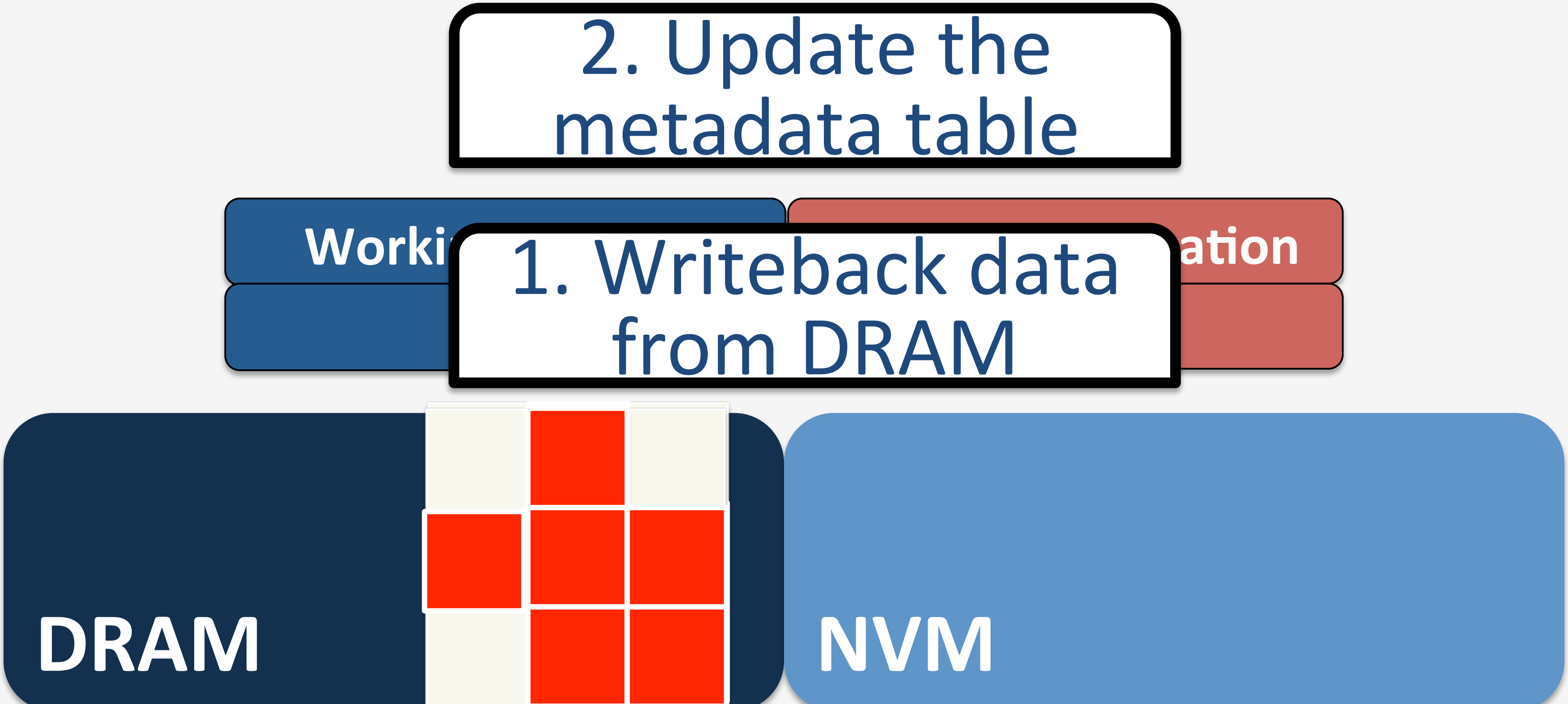
1. METADATA AND CHECKPOINTING GRANULARITY

Working location	Checkpoint location
X	X’
Y	Y’

PAGE GRANULARITY	BLOCK GRANULARITY
One Entry Per Page Small Metadata	One Entry Per Block Huge Metadata

2. LATENCY AND LOCATION

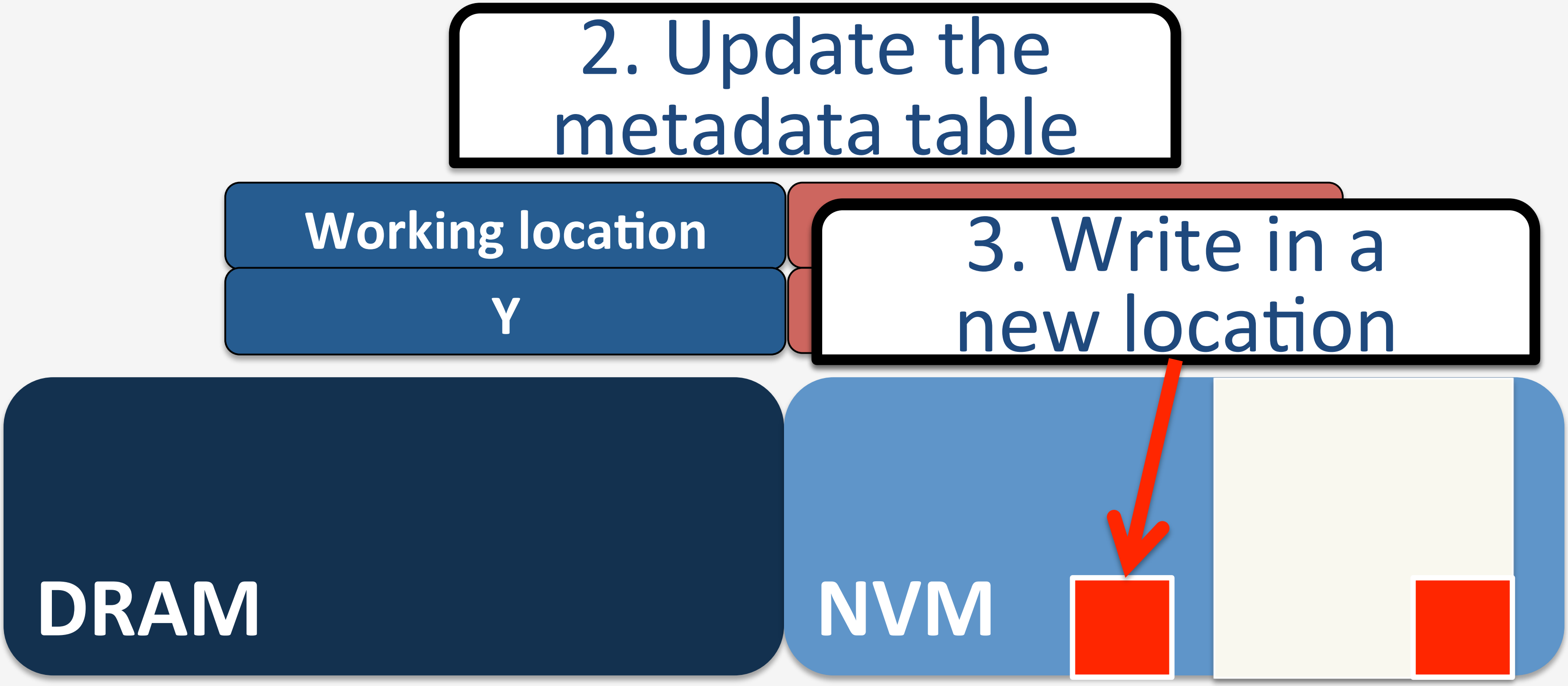
DRAM-BASED WRITEBACK



Long latency of writing back data to NVM

2. LATENCY AND LOCATION

NVM-BASED REMAPPING



Short latency in NVM-based remapping

ThyNVM KEY MECHANISMS

Checkpointing granularity

- *Small granularity: large metadata*
- *Large granularity: small metadata*

Latency and location

- *Writeback from DRAM: long latency*
- *Remap in NVM: short latency*

Based on these we propose two key mechanisms

1. Dual granularity checkpointing
2. Overlap of execution and checkpointing

1. DUAL GRANULARITY CHECKPOINTING

Page Writeback in DRAM

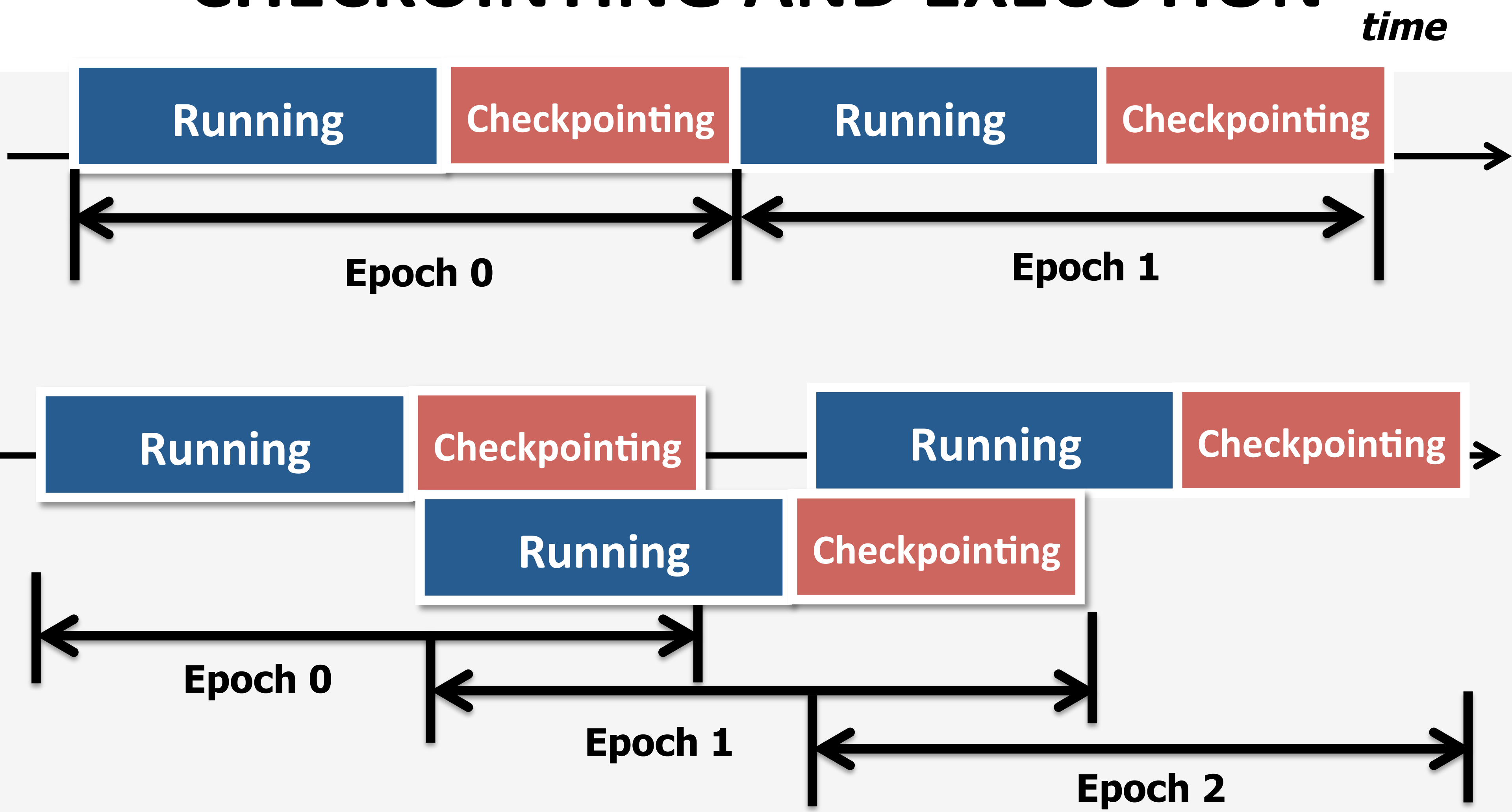
Block Remapping in NVM

GOOD FOR STREAMING WRITES

GOOD FOR RANDOM WRITES

High write locality pages in DRAM, low write locality pages in NVM

2. OVERLAPPING CHECKOINTING AND EXECUTION



Hides the long latency of Page Writeback

OUTLINE

- Crash Consistency Problem
- Current Solutions
- ThyNVM
- **Evaluation**
- Conclusion

METHODOLOGY

Cycle accurate x86 simulator Gem5

Comparison Points:

Ideal DRAM: DRAM-based, no cost for consistency

– Lowest latency system

Ideal NVM: NVM-based, no cost for consistency

– NVM has higher latency than DRAM

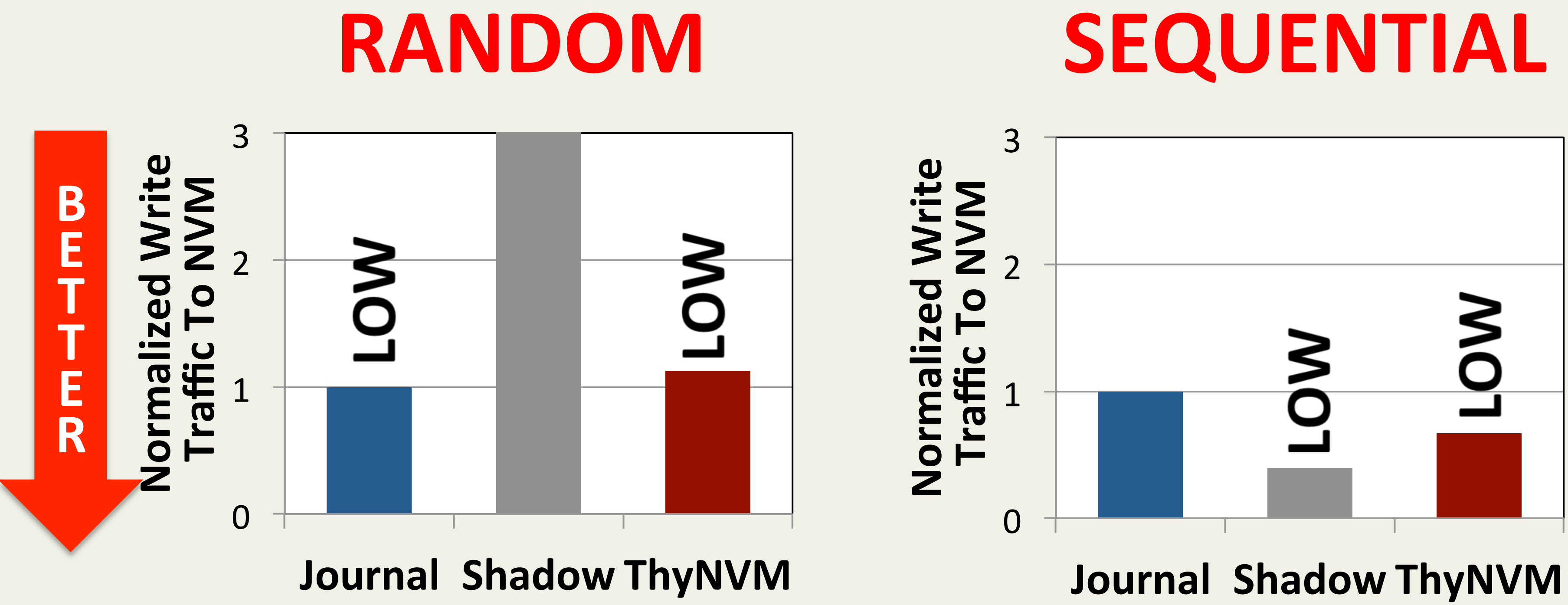
Journaling: Hybrid, commit dirty cache blocks

– Leverages DRAM to buffer dirty blocks

Shadow Paging: Hybrid, copy-on-write pages

– Leverages DRAM to buffer dirty pages

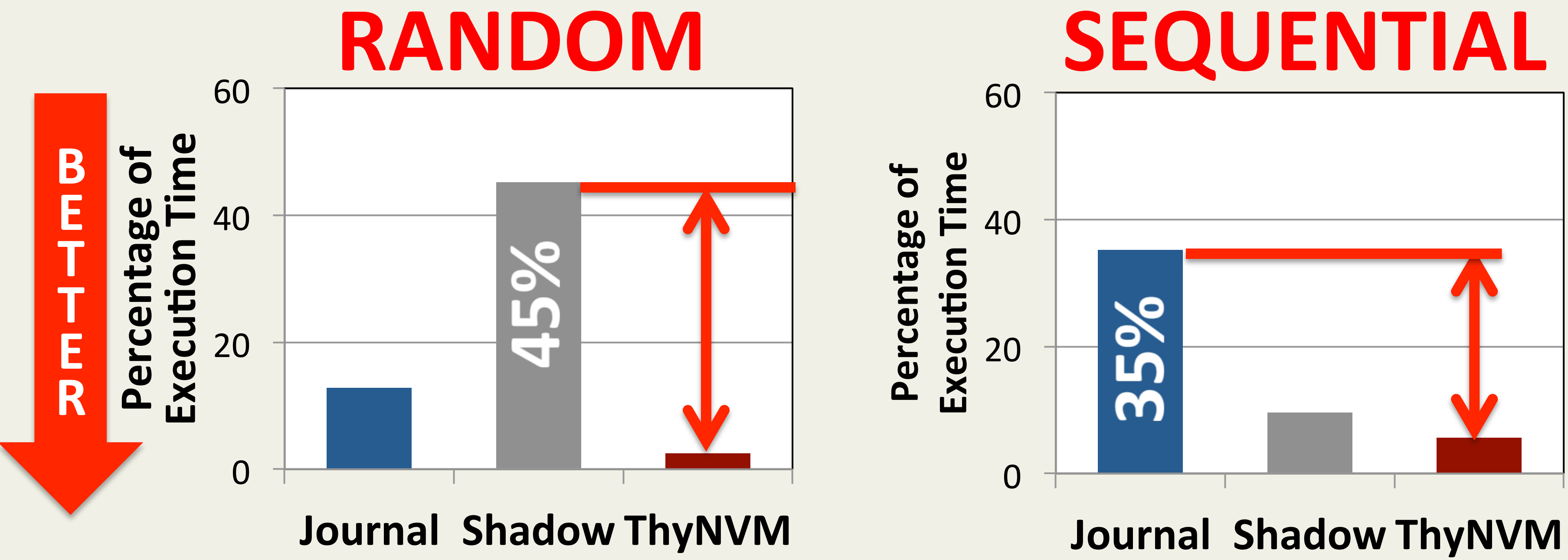
ADAPTIVITY TO ACCESS PATTERN



Journaling is better for Random and Shadow paging is better for Sequential

ThyNVM adapts to both access patterns

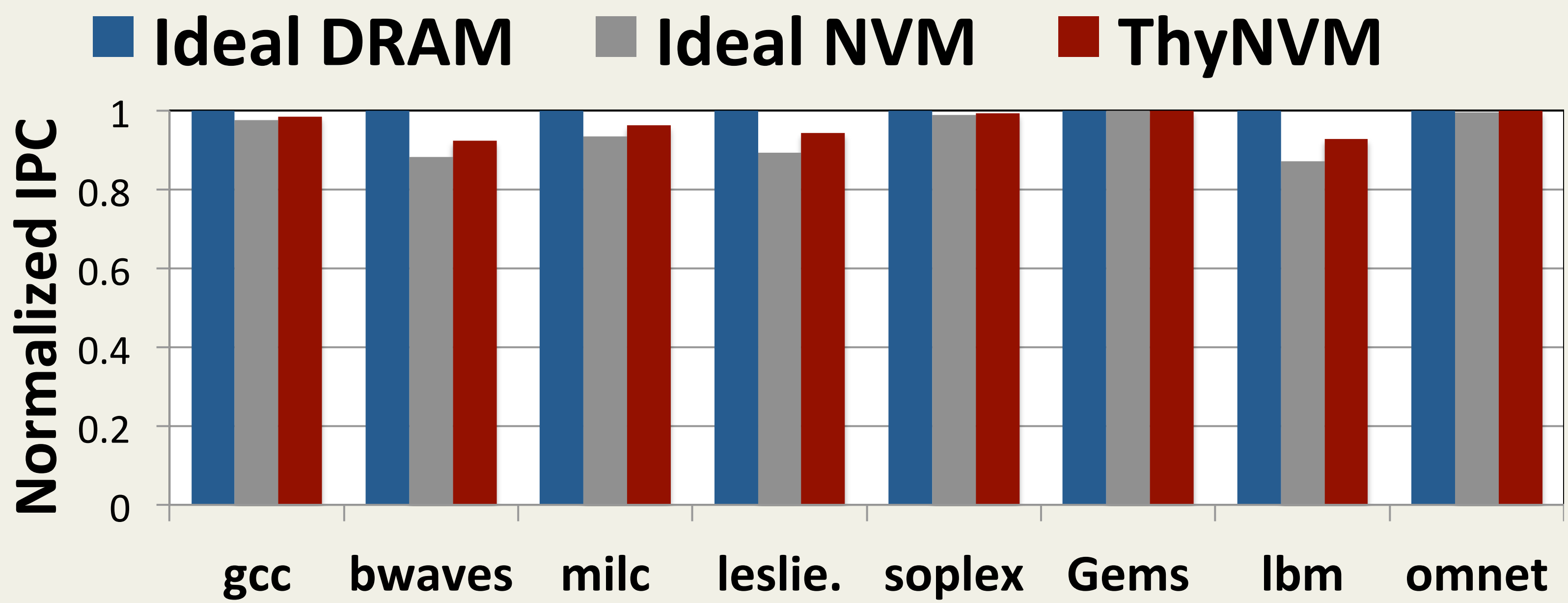
OVERLAPPING CHECKPOINTING AND EXECUTION



Can spend 35-45% of the execution on checkpointing

Stalls the application for a negligible time

PERFORMANCE OF LEGACY CODE



Within -4.9%/+2.7% of an idealized DRAM/NVM system

Provides consistency without significant performance overhead

OUTLINE

Crash Consistency Problem

Current Solutions

ThyNVM

Evaluation

Conclusion

ThyNVM

A new **hardware-based** *checkpointing mechanism*, **with no programming effort**

- **Checkpoints** at *multiple granularities* to minimize both latency and metadata
- **Overlaps** *checkpointing* and *execution*
- **Adapts** to *DRAM and NVM* characteristics

Can enable widespread *adoption* of persistent memory

Available at
http://persper.com/thynvm

ThyNVM

Enabling Software-transparent Crash Consistency In Persistent Memory Systems

Jinglei Ren, Jishen Zhao, **Samira Khan**, Jongmoo Choi, Yongwei Wu, and Onur Mutlu

Carnegie Mellon University
SAFARI