Computer Architecture
Lecture 14a: Emerging Memory Technologies II

Prof. Onur Mutlu
ETH Zürich
Fall 2018
1 November 2018

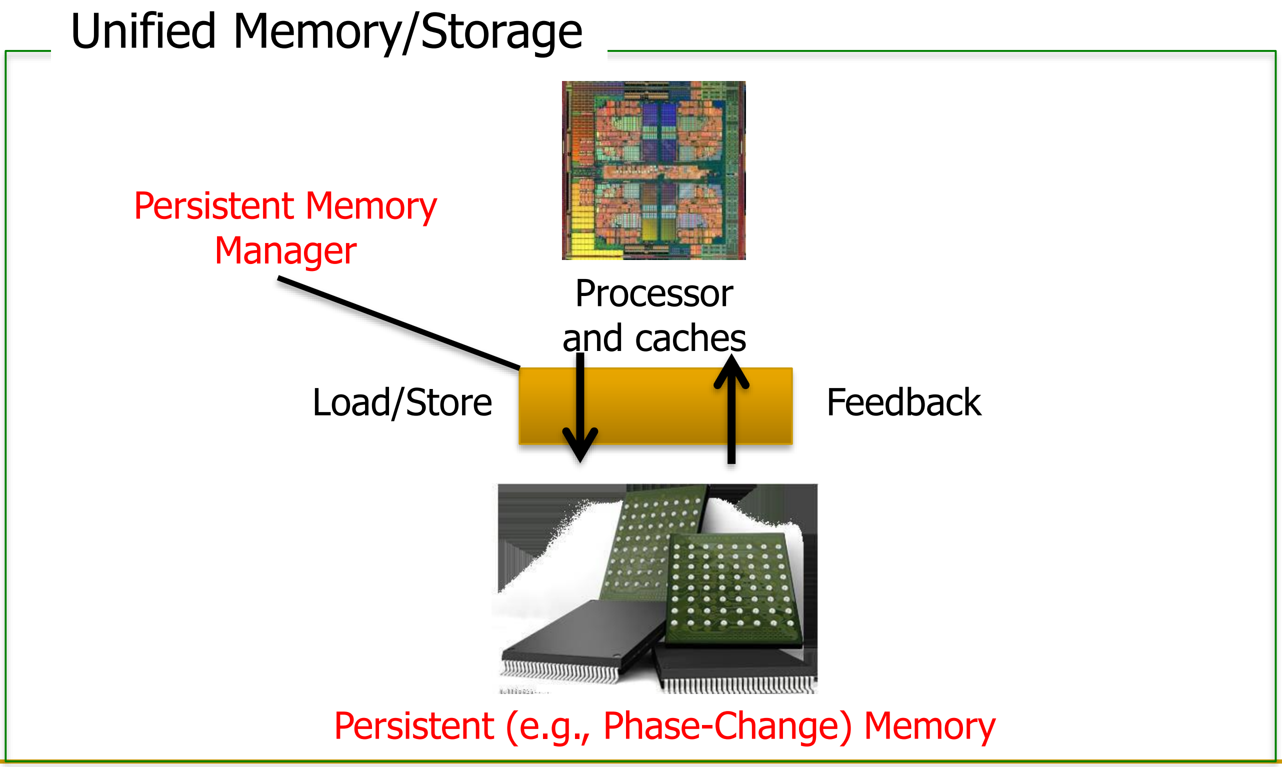
Emerging Memory Technologies

Other Opportunities with Emerging Technologies

- Merging of memory and storage
  - e.g., a single interface to manage all data
- New applications
  - e.g., ultra-fast checkpoint and restore
- More robust system design
  - e.g., reducing data loss
- Processing tightly-coupled with memory
  - e.g., enabling efficient search and filtering

Unified Memory and Storage with NVM

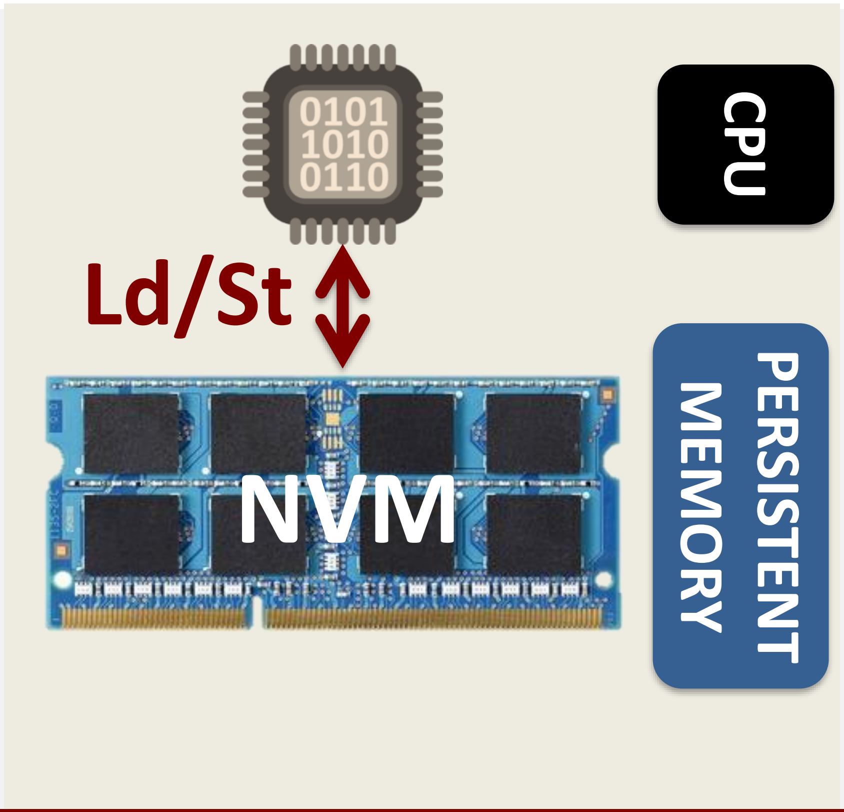
- Goal: Unify memory and storage management in a single unit to eliminate wasted work to locate, transfer, and translate data
  - Improves both energy and performance
  - Simplifies programming model as well



Unified Memory/Storage
Persistent Memory Manager
Processor and caches
Load/Store
Feedback
Persistent (e.g., Phase-Change) Memory

Meza+, "A Case for Efficient Hardware-Software Cooperative Management of Storage and Memory," WEED 2013.

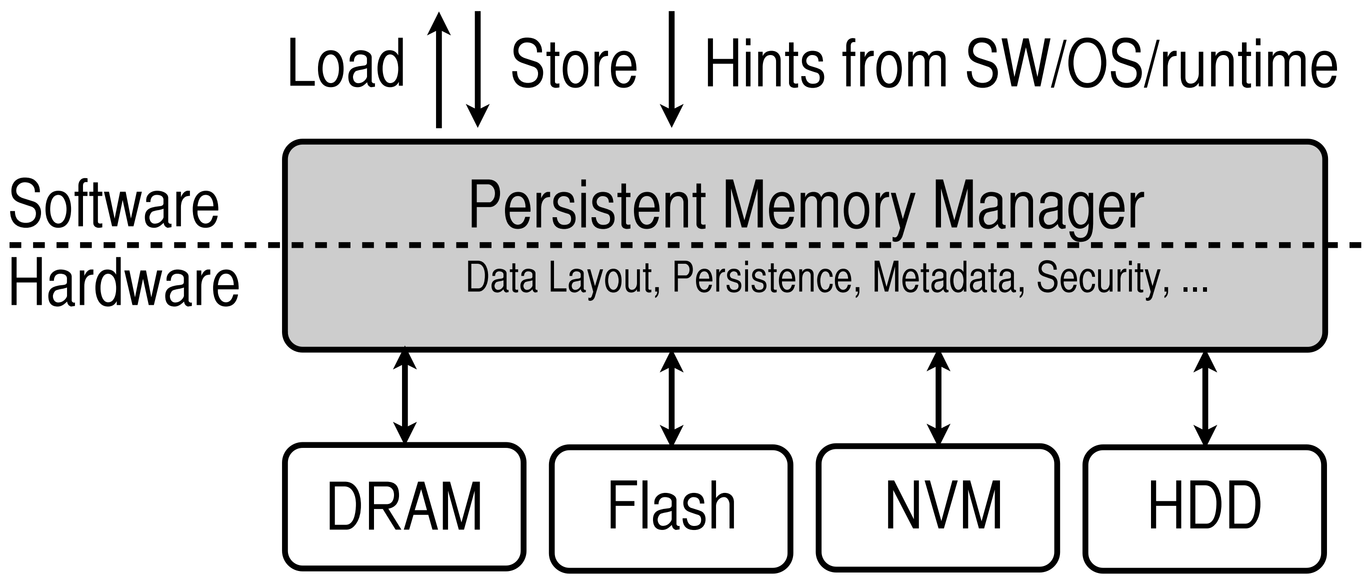
PERSISTENT MEMORY



Provides an opportunity to manipulate persistent data directly

The Persistent Memory Manager (PMM)

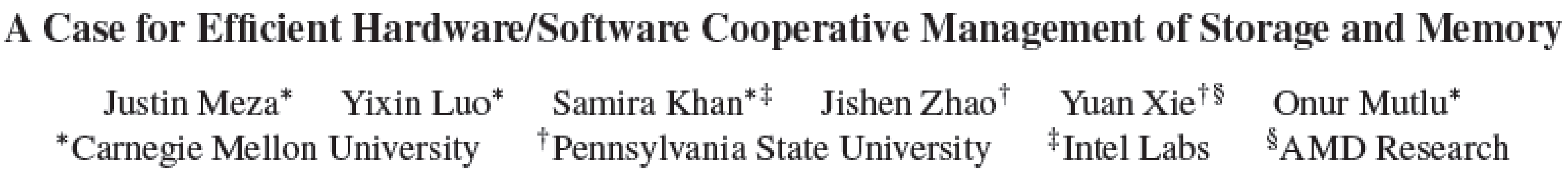
```
int main(void) {
  // data in file.dat is persistent
  FILE myData = "file.dat";
  myData = new int[64];
}
void updateValue(int n, int value) {
  FILE myData = "file.dat";
  myData[n] = value; // value is persistent
}
```



PMM uses access and hint information to allocate, locate, migrate and access data in the heterogeneous array of devices

On Persistent Memory Benefits & Challenges

- Justin Meza, Yixin Luo, Samira Khan, Jishen Zhao, Yuan Xie, and Onur Mutlu,
  "A Case for Efficient Hardware-Software Cooperative Management of Storage and Memory"
  Proceedings of the 5th Workshop on Energy-Efficient Design (***WEED***), Tel-Aviv, Israel, June 2013. Slides (pptx) Slides (pdf)


A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory

Justin Meza* Yixin Luo* Samira Khan*‡ Jishen Zhao† Yuan Xie†§ Onur Mutlu*
*Carnegie Mellon University †Pennsylvania State University ‡Intel Labs §AMD Research

Challenge and Opportunity

Combined

Memory & Storage

Challenge and Opportunity

A Unified Interface to
All Data

One Key Challenge in Persistent Memory

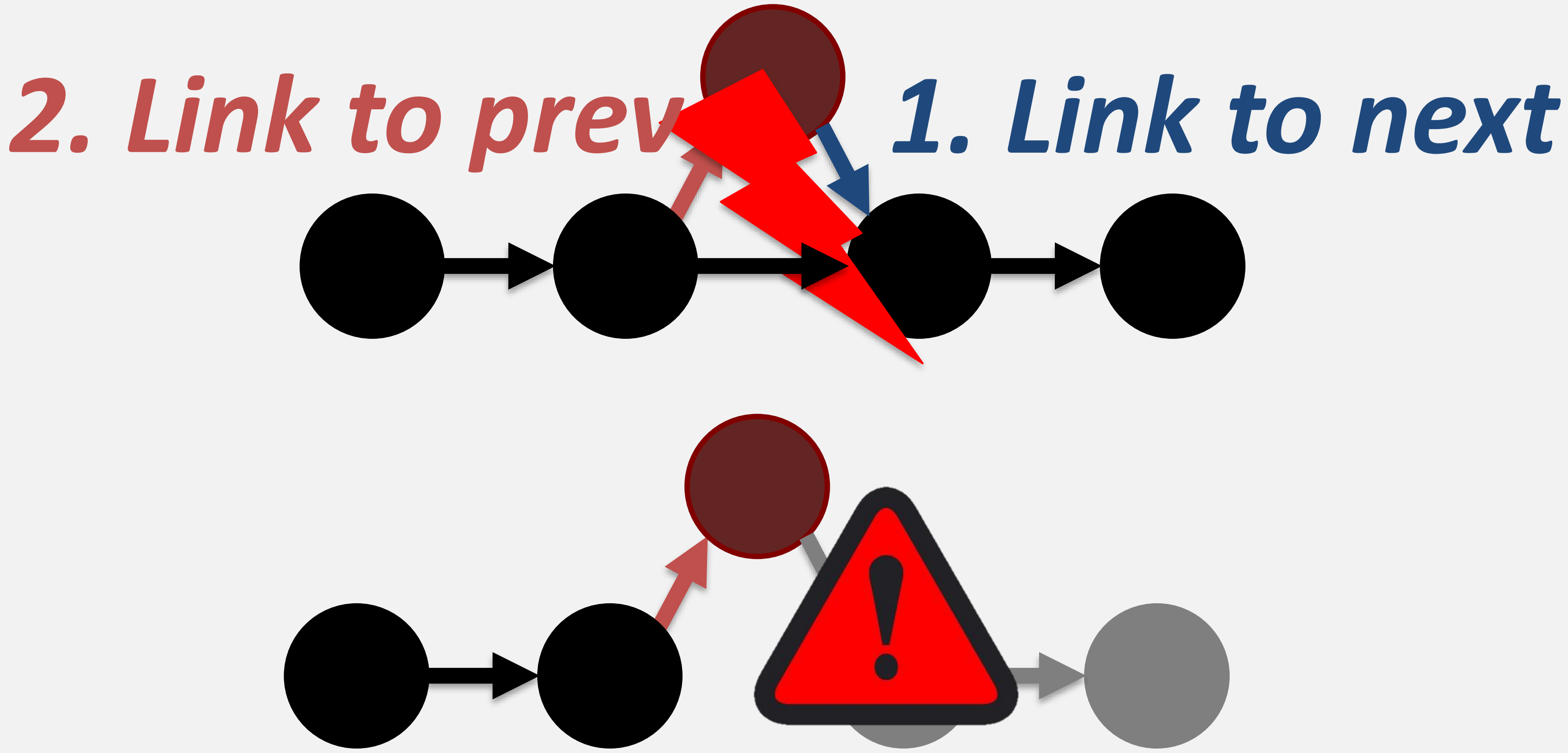
- How to ensure consistency of system/data if all memory is persistent?

- Two extremes
  - Programmer transparent: Let the system handle it
  - Programmer only: Let the programmer handle it

- Many alternatives in-between…

CRASH CONSISTENCY PROBLEM

Add a node to a linked list



System crash can result in inconsistent memory state

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

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

Third party code can be inconsistent

Prohibited Operation

Burden on the programmers

OUR APPROACH: ThyNVM

Goal:
Software transparent consistency in persistent memory systems

Key Idea:
Periodically checkpoint state;
recover to previous checkpt on crash

ThyNVM: Summary

A new hardware-based checkpointing mechanism

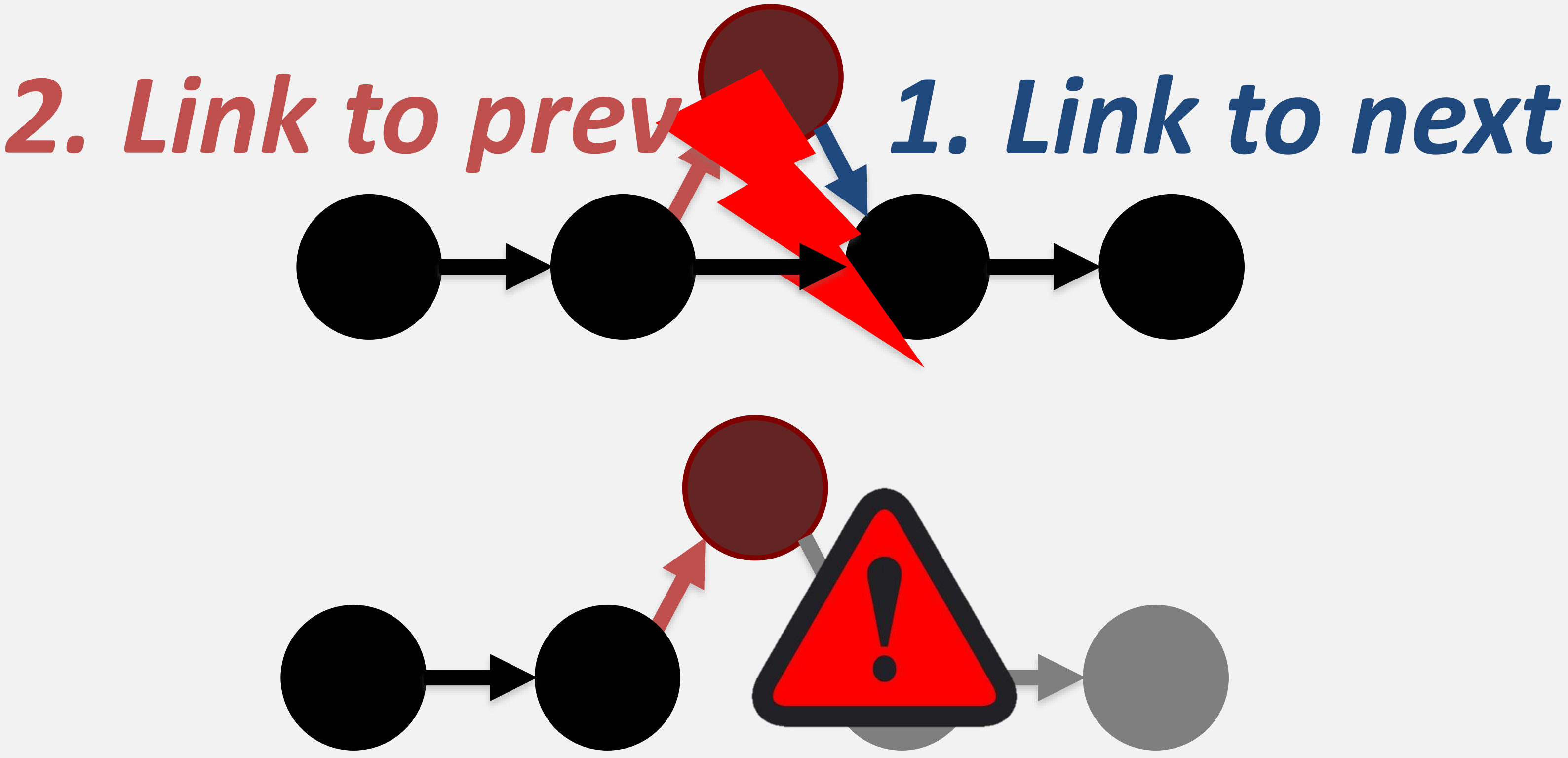
- **Checkpoints** at *multiple granularities* to reduce both checkpointing latency and metadata overhead
- **Overlaps** *checkpointing* and *execution to* reduce checkpointing latency
- **Adapts** to *DRAM and NVM* characteristics

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

OUTLINE

CRASH CONSISTENCY PROBLEM

Add a node to a linked list



System crash can result in inconsistent memory state

OUTLINE

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

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…

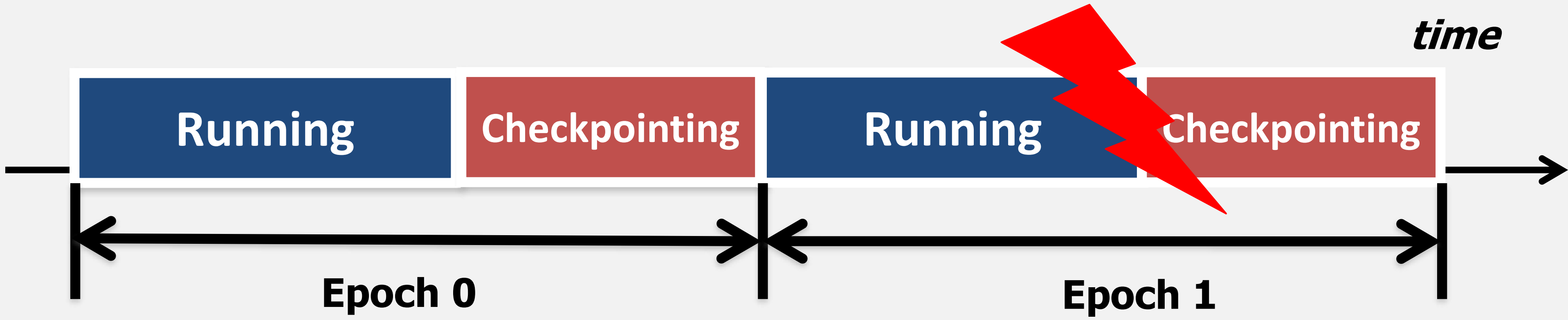
Persistent Memory System

```
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

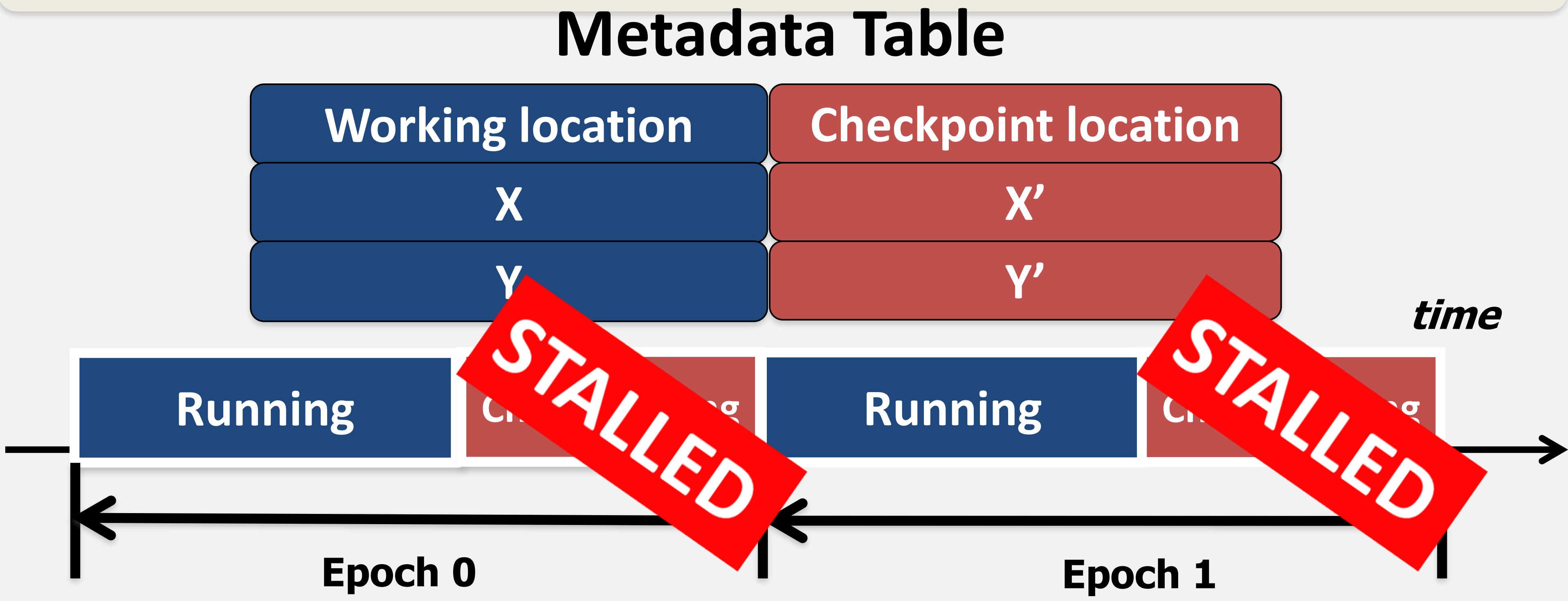
Periodic checkpointing of data managed by hardware



Transparent to application and system

CHECKPOINTING OVERHEAD

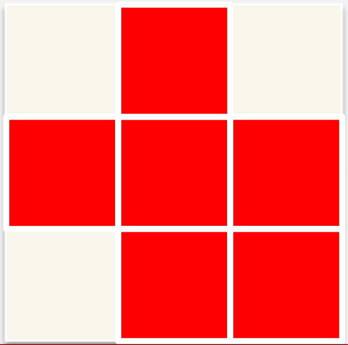
1. Metadata overhead



2. Checkpointing latency

1. METADATA AND CHECKPOINTING GRANULARITY

Working location	Checkpoint location
X	X’
Y	Y’



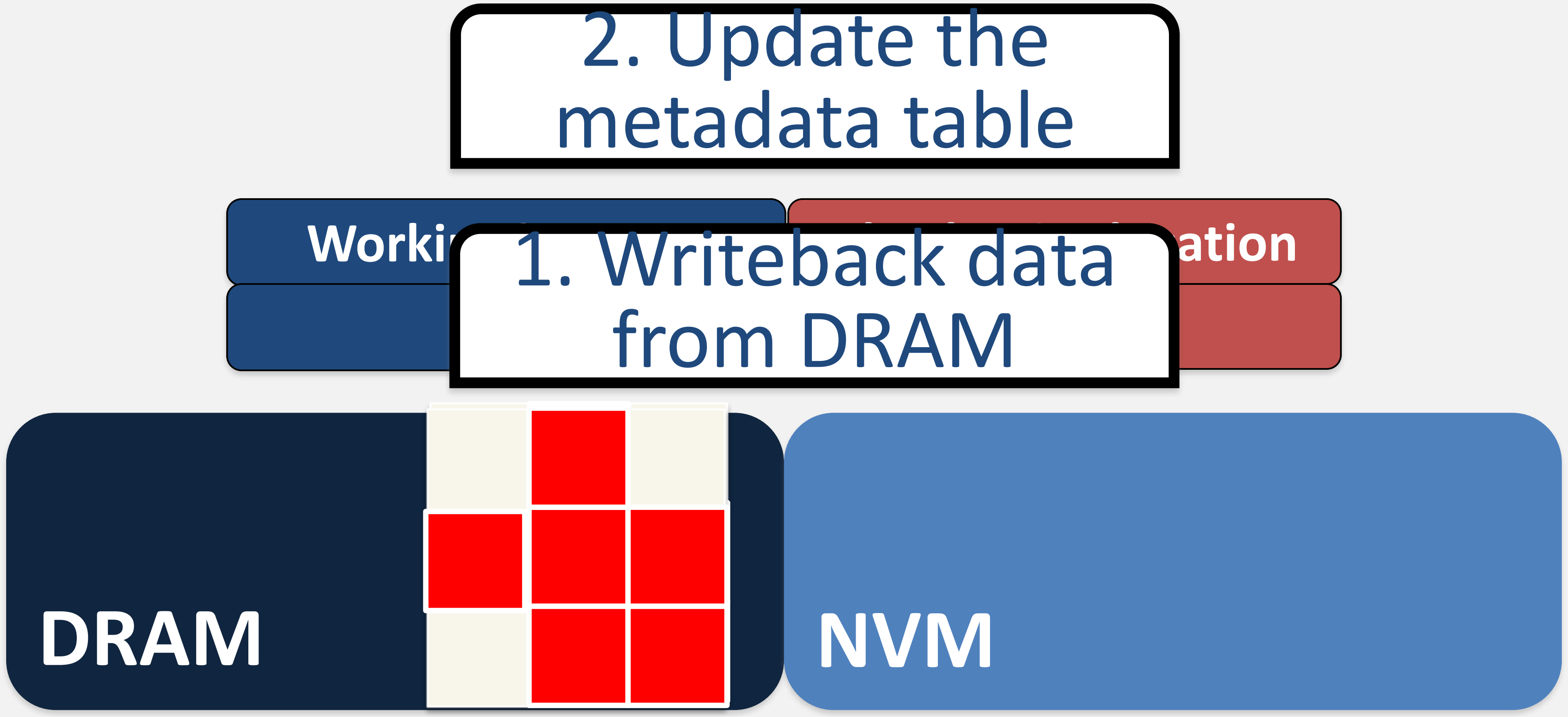
PAGE

CACHE BLOCK

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

2. LATENCY AND LOCATION

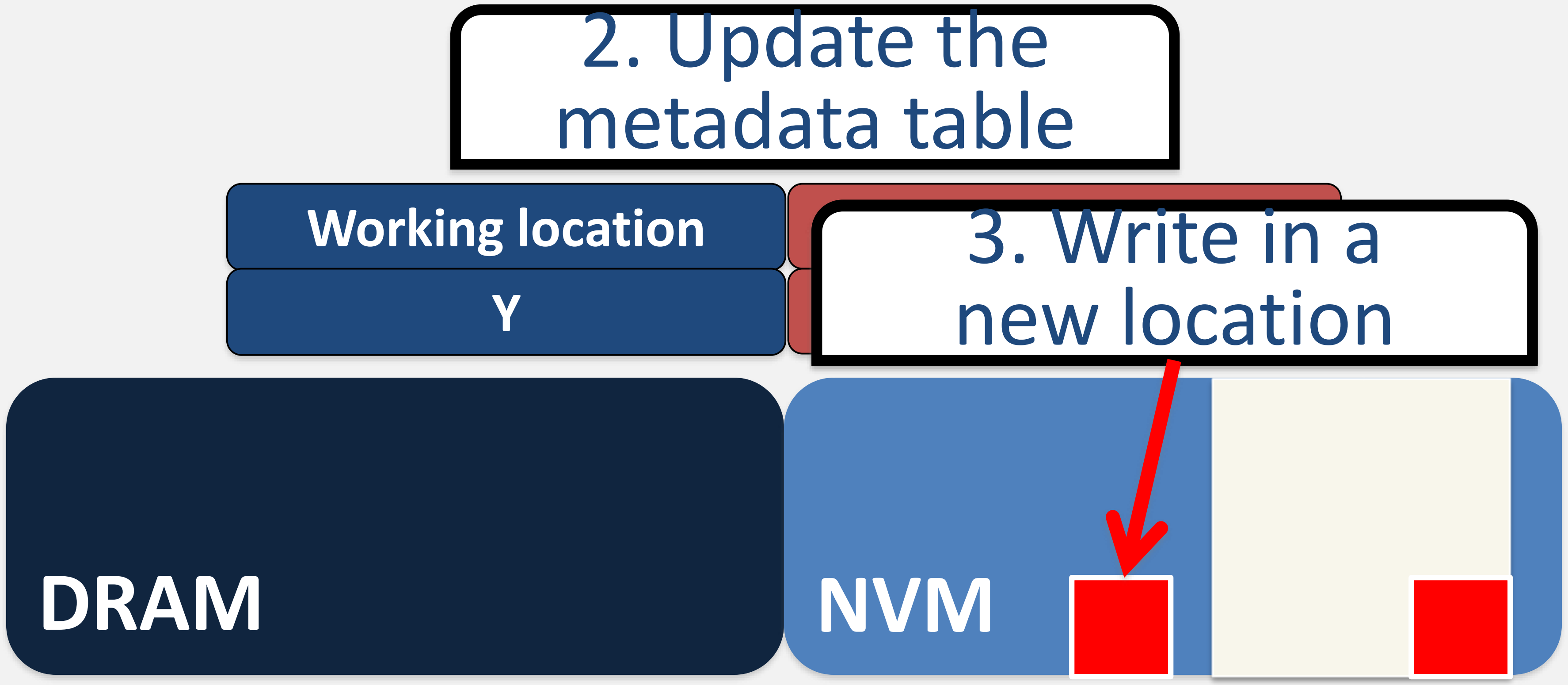
DRAM-BASED WRITEBACK



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*

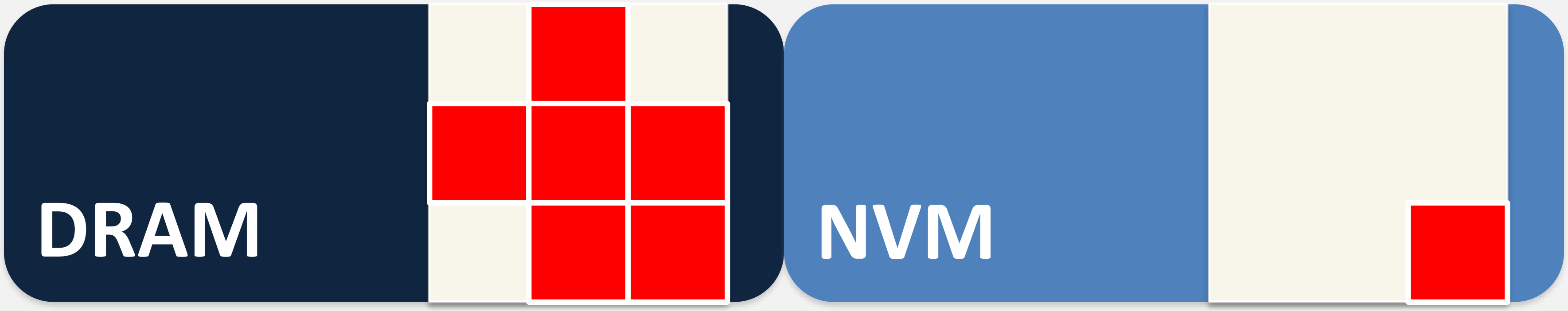
Based on these, we propose two key mechanisms

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

1. DUAL GRANULARITY CHECKPOINTING

Page Writeback in DRAM

Block Remapping in NVM

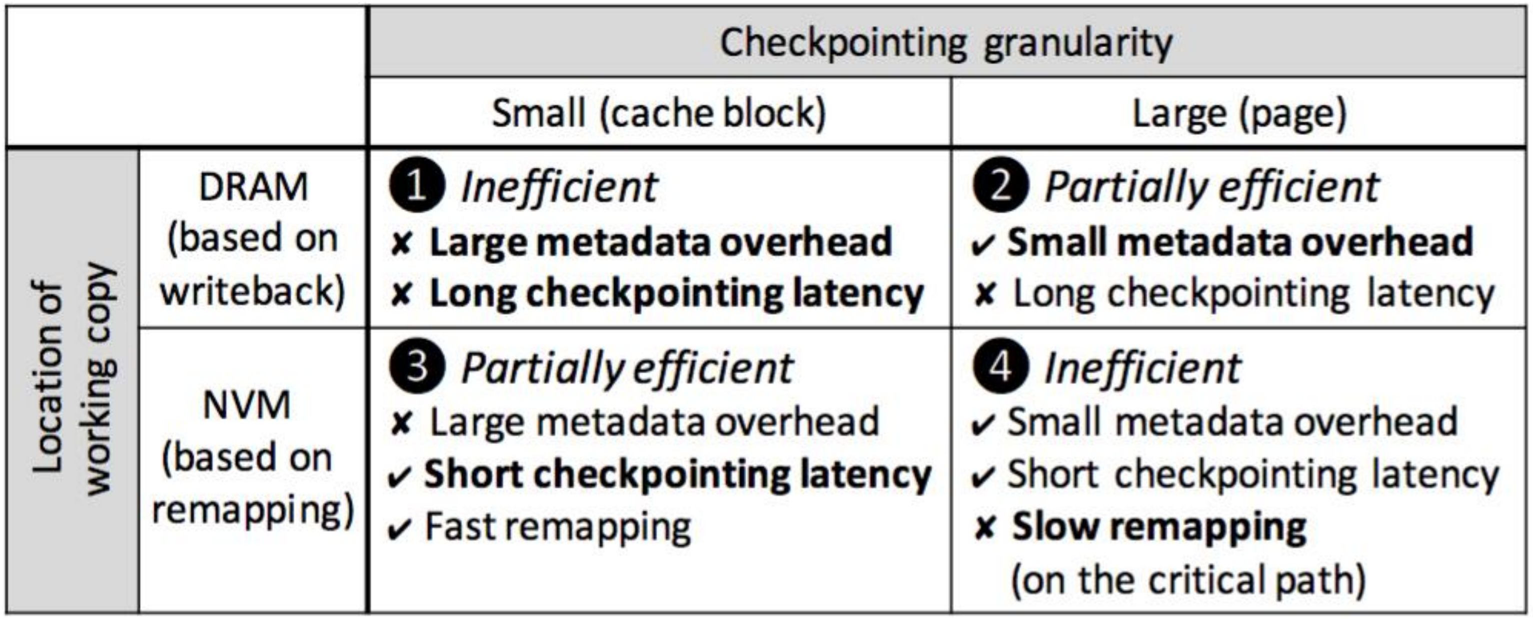


GOOD FOR STREAMING WRITES

GOOD FOR RANDOM WRITES

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

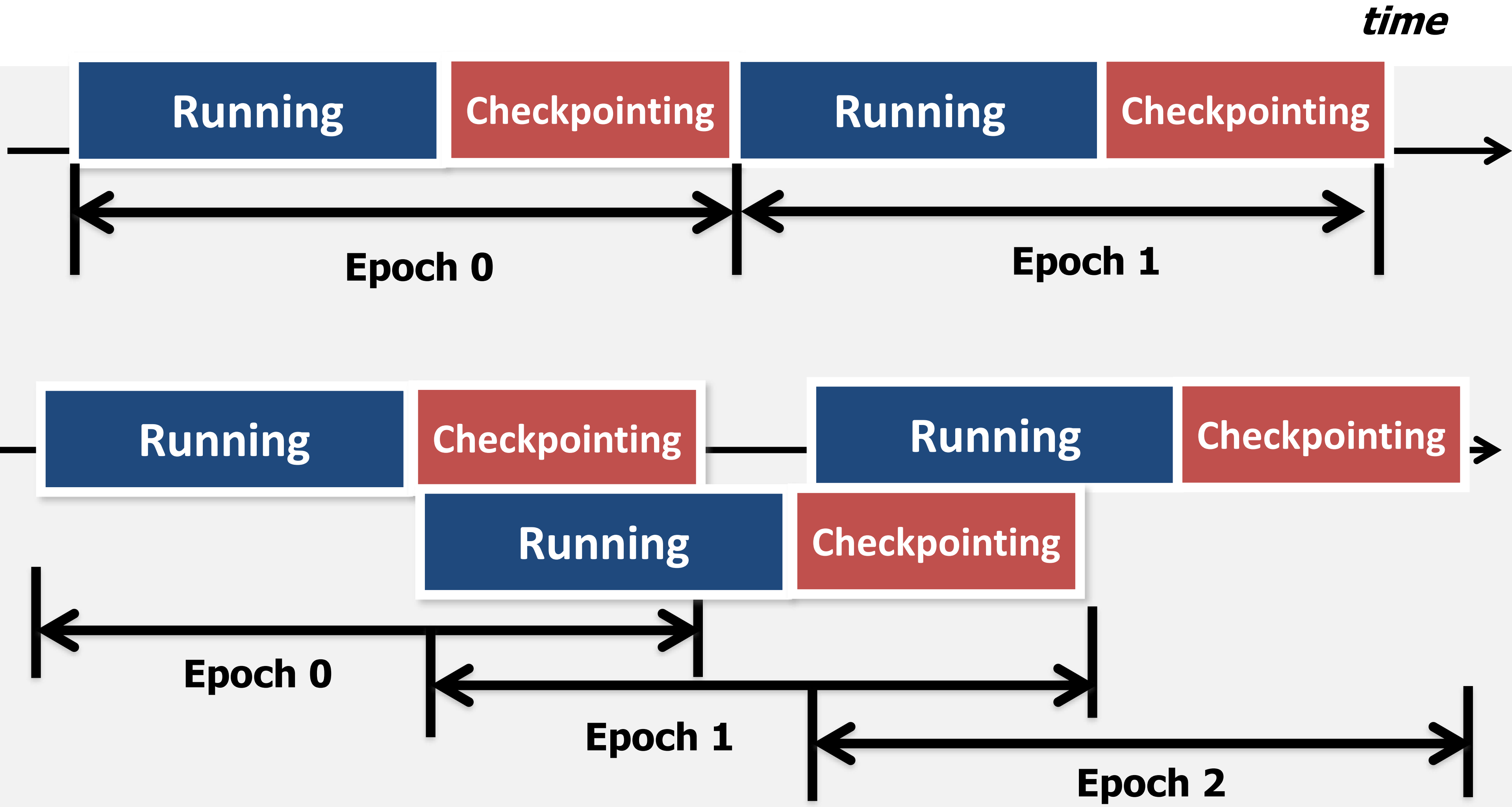
TRADEOFF SPACE



		Checkpointing granularity	
		Small (cache block)	Large (page)
Location of working copy	DRAM (based on writeback)	❶ *Inefficient* ✘ **Large metadata overhead** ✘ **Long checkpointing latency**	❷ *Partially efficient* ✔ **Small metadata overhead** ✘ Long checkpointing latency
	NVM (based on remapping)	❸ *Partially efficient* ✘ Large metadata overhead ✔ **Short checkpointing latency** ✔ Fast remapping	❹ *Inefficient* ✔ Small metadata overhead ✔ Short checkpointing latency ✘ **Slow remapping** (on the critical path)

Table 1: Tradeoff space of options combining checkpointing granularity choice and location choice of the working copy of data. The table shows four options and their pros and cons. Boldfaced text indicates the most critical pro or con that determines the efficiency of an option.

2. OVERLAPPING CHECKPOINTING AND EXECUTION



Hides the long latency of Page Writeback

OUTLINE

Crash Consistency Problem

Current Solutions

ThyNVM

Evaluation

Conclusion

SYSTEM ARCHITECTURE

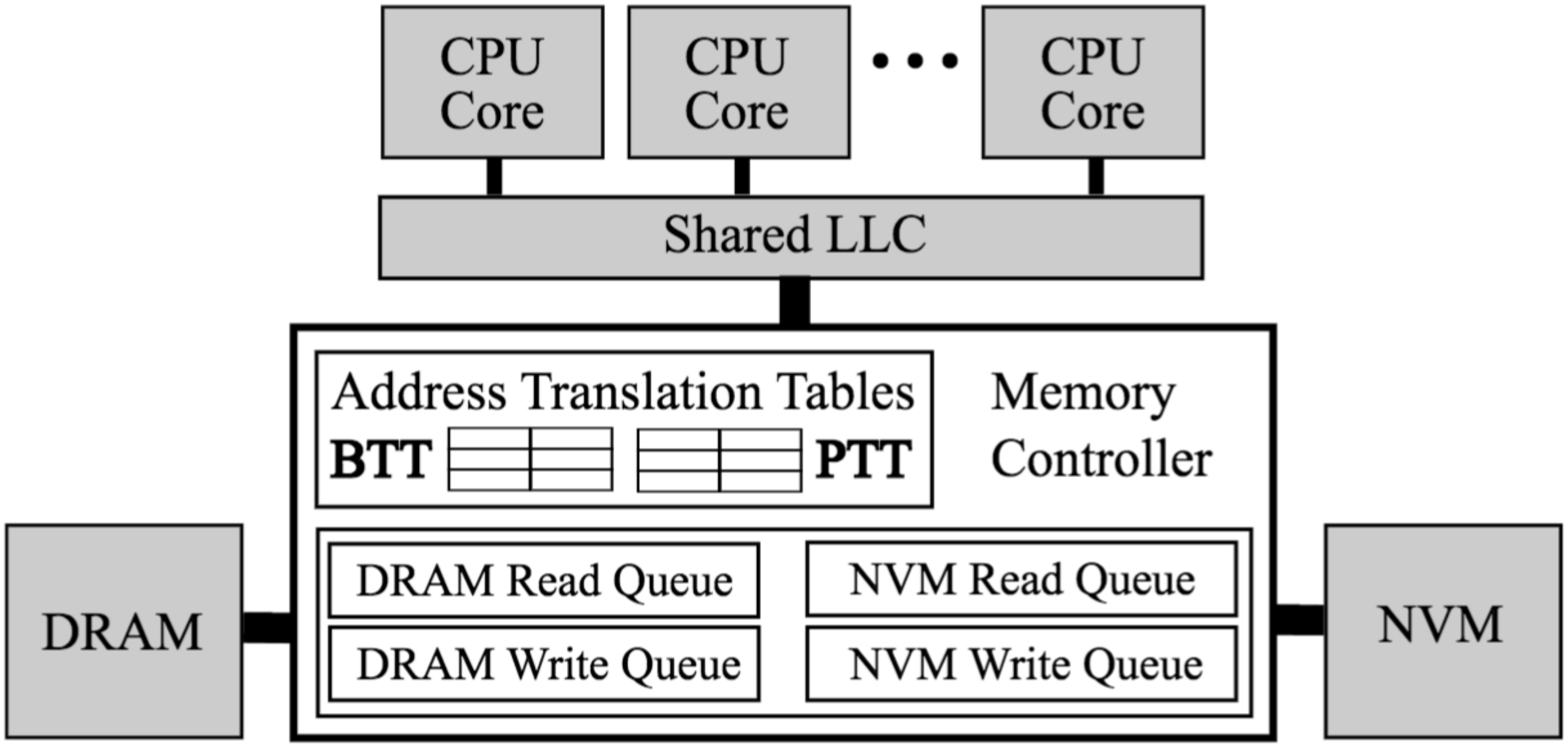
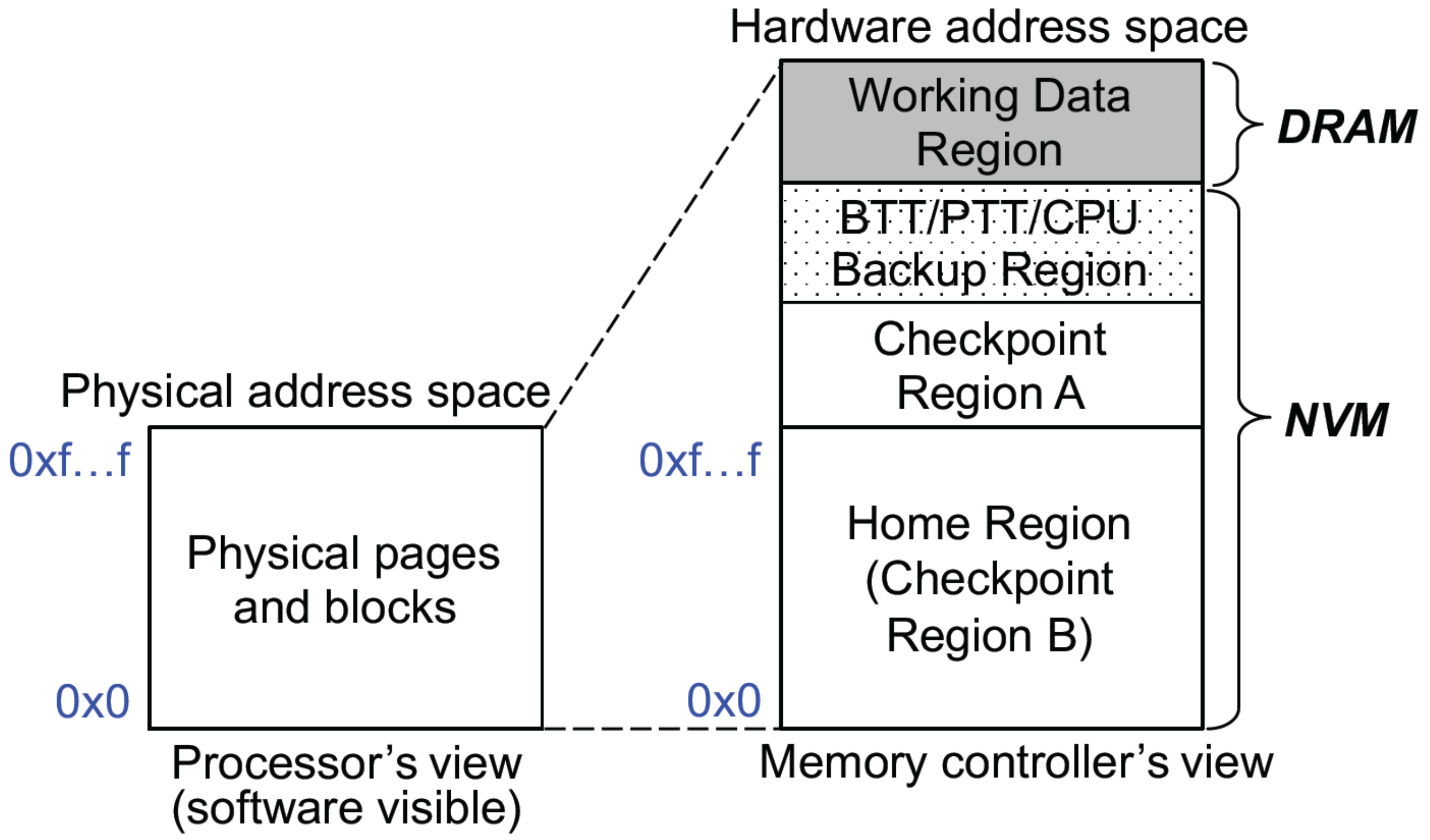


Figure 2: Architecture overview of ThyNVM.

MEMORY ADDRESS SPACE



Working Data Region: W_{active}^{page}, W_{active}^{block} (when creating C_{last})

Ckpt Regions A and B: C_{last}, C_{penult}, W_{active}^{block}

Figure 4: ThyNVM address space layout.

METHODOLOGY

Cycle accurate x86 simulator Gem5

Comparison Points:

Ideal DRAM: DRAM-based, no cost for consistency
- Lowest latency system

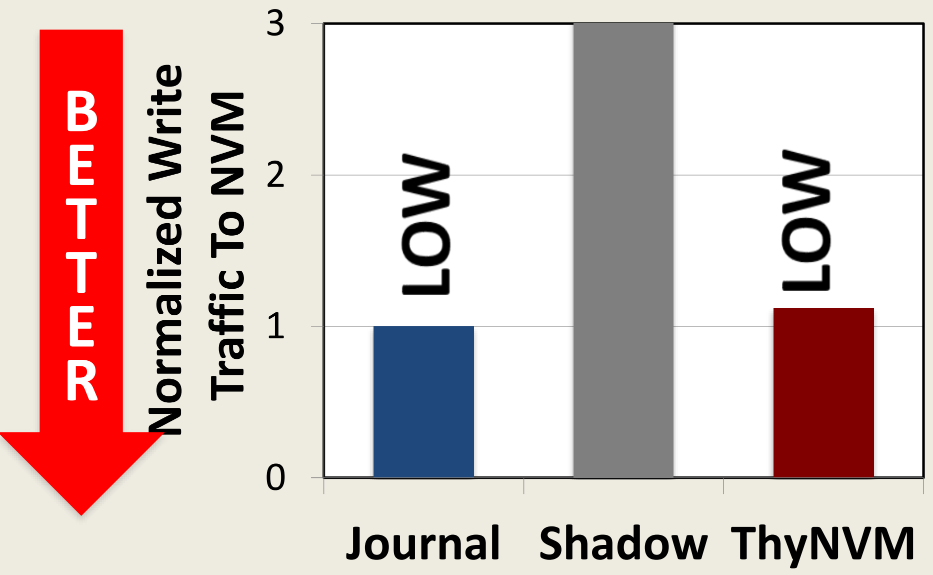
Ideal NVM: NVM-based, no cost for consistency
- NVM has higher latency than DRAM

Journaling: Hybrid, commit dirty cache blocks
- Leverages DRAM to buffer dirty blocks

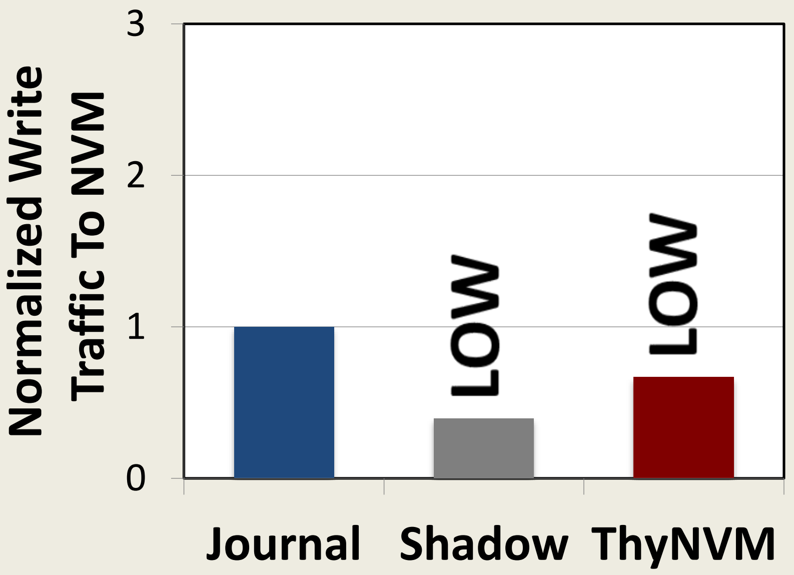
Shadow Paging: Hybrid, copy-on-write pages
- Leverages DRAM to buffer dirty pages

ADAPTIVITY TO ACCESS PATTERN

RANDOM



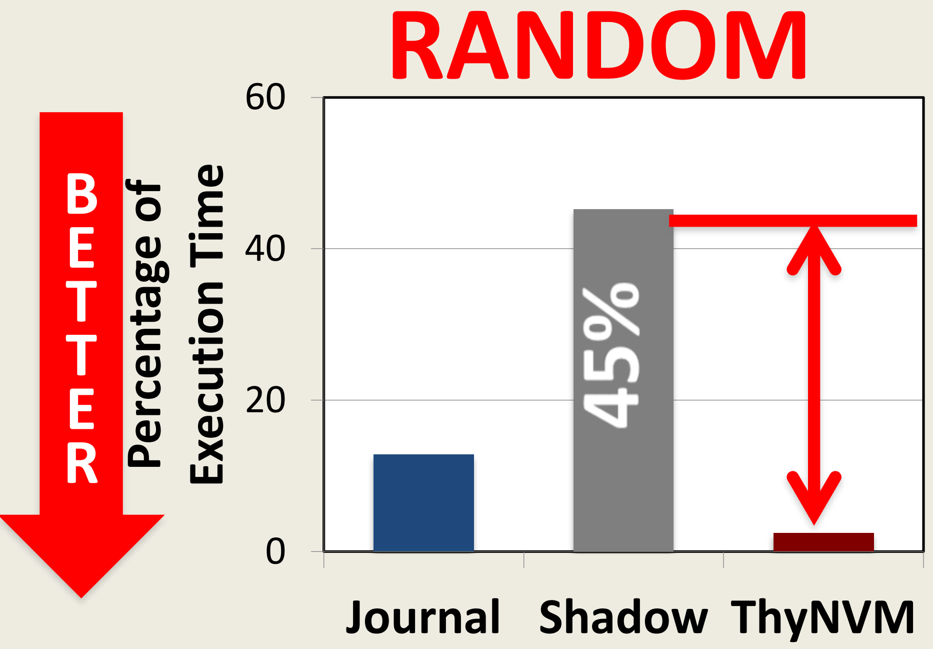
SEQUENTIAL

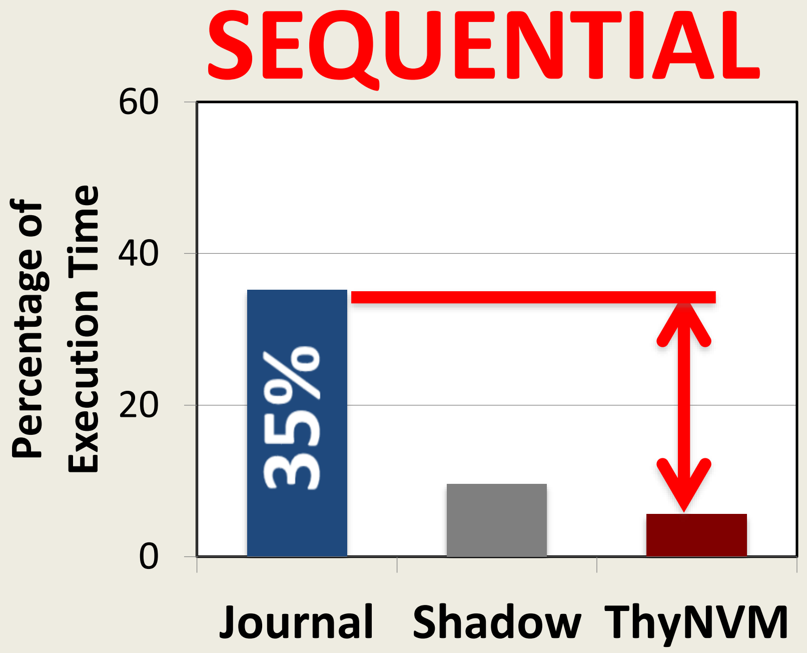


Journaling is better for Random and Shadow paging is better for Sequential

ThyNVM adapts to both access patterns

OVERLAPPING CHECKPOINTING AND EXECUTION

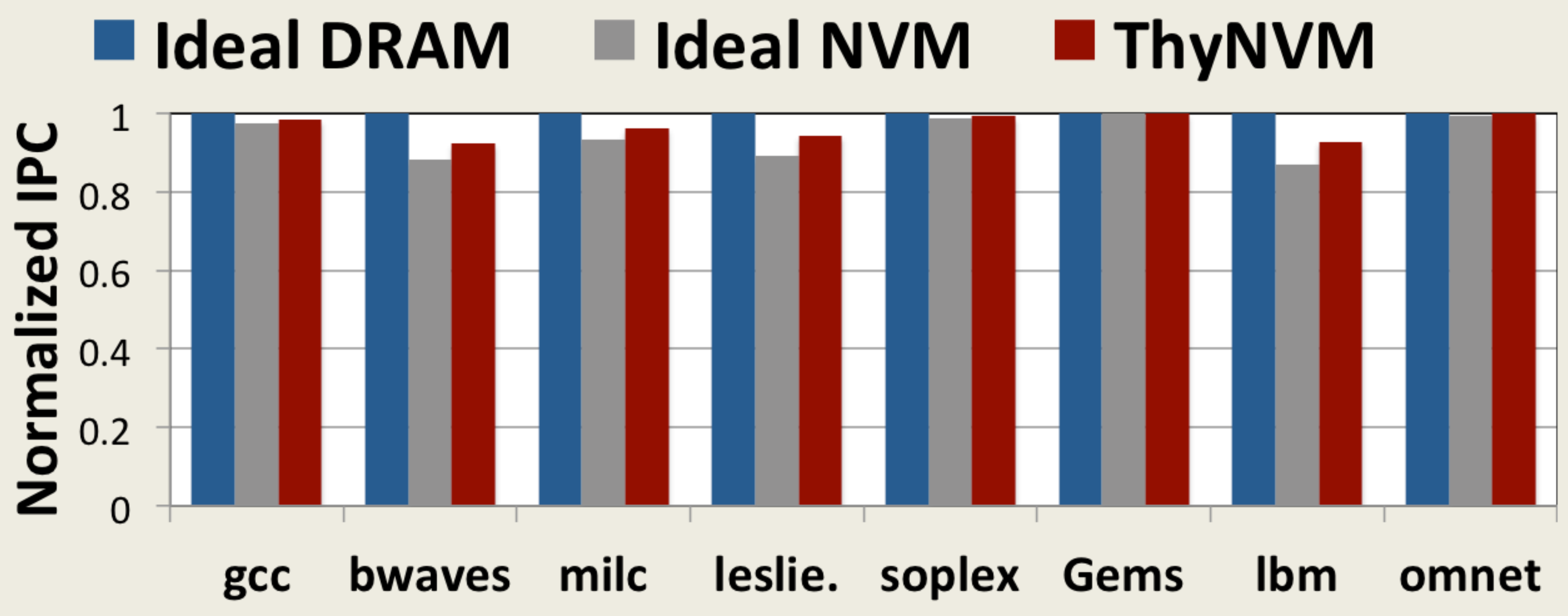




Can spend 35-45% of the execution on checkpointing

Stalls the application for a negligible time

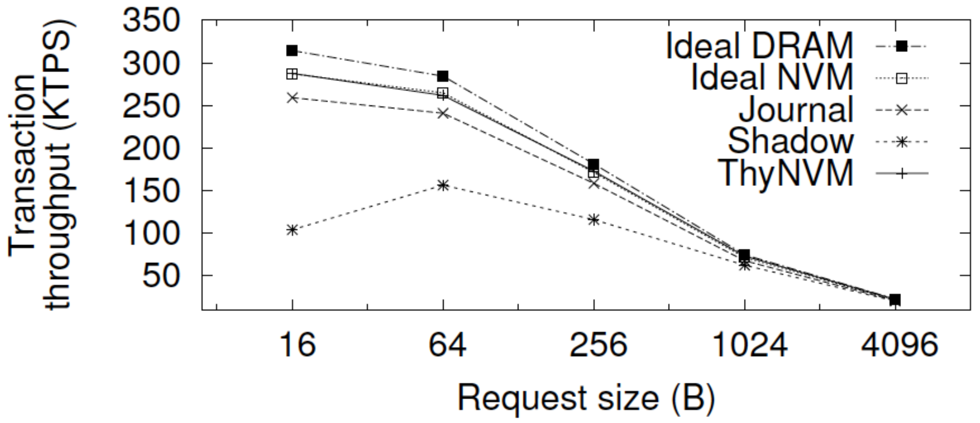
PERFORMANCE OF LEGACY CODE



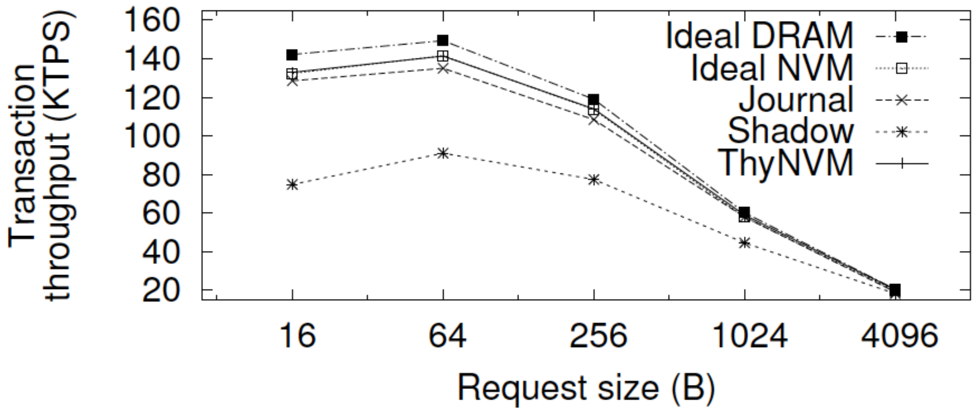
Within -4.9%/+2.7% of an idealized DRAM/NVM system

Provides consistency without significant performance overhead

KEY-VALUE STORE TX THROUGHPUT



(a) Hash table based key-value store



(b) Red-black tree based key-value store

Figure 9: Transaction throughput for two key-value stores: (a) hash table based, (b) red-black tree based.

Storage throughput close to Ideal DRAM

OUTLINE

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

Source Code and More Available at

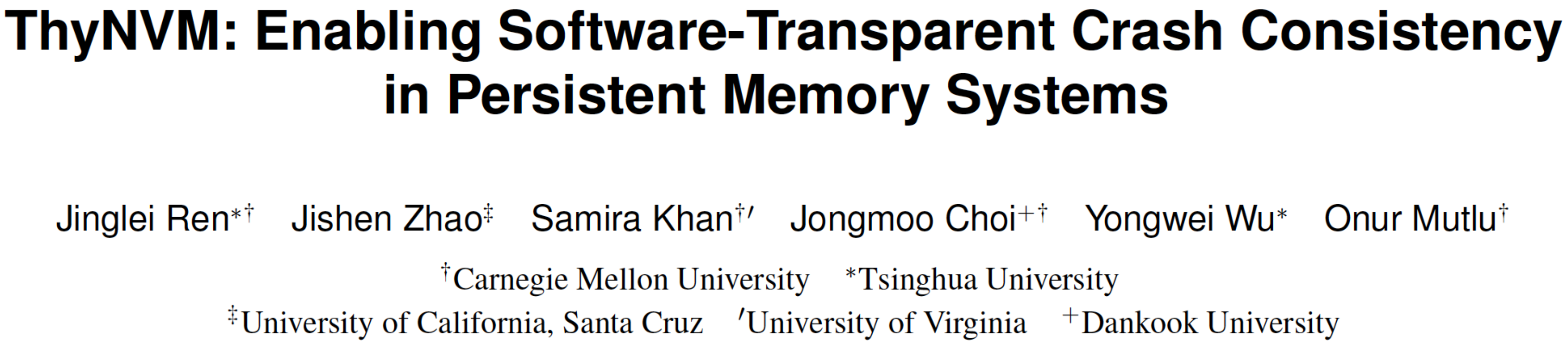
http://persper.com/thynvm

ThyNVM

Enabling Software-transparent Crash Consistency In Persistent Memory Systems

More About ThyNVM

- Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu,
  "ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems"
  Proceedings of the 48th International Symposium on Microarchitecture (***MICRO***), Waikiki, Hawaii, USA, December 2015.
  [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
  [Source Code]



ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems

Jinglei Ren[*†] Jishen Zhao[‡] Samira Khan[†′] Jongmoo Choi[+†] Yongwei Wu[*] Onur Mutlu[†]

[†]Carnegie Mellon University [*]Tsinghua University
[‡]University of California, Santa Cruz [′]University of Virginia [+]Dankook University

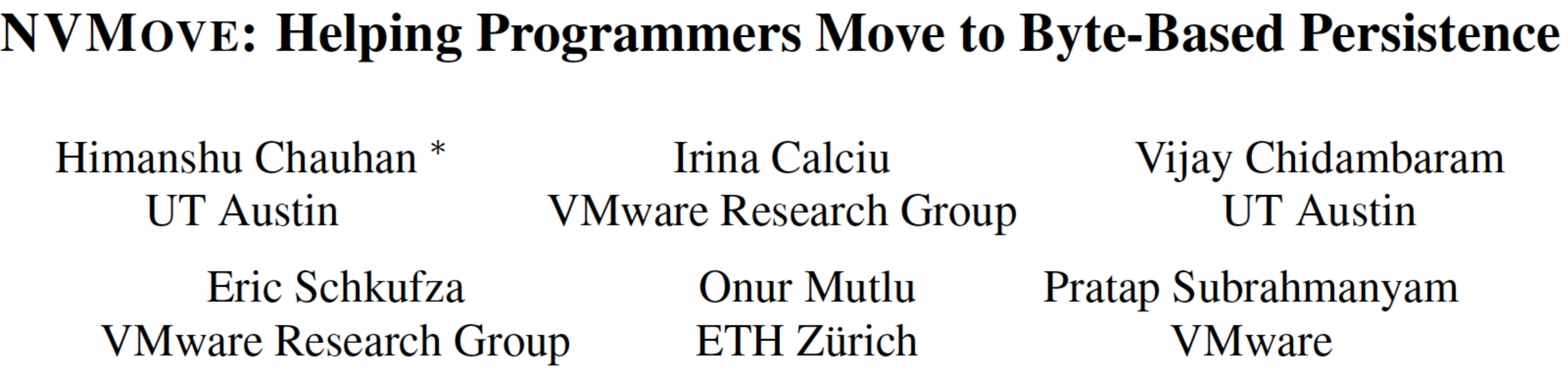
Another Key Challenge in Persistent Memory

Programming Ease

to Exploit Persistence

Tools/Libraries to Help Programmers

- Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam, **"NVMove: Helping Programmers Move to Byte-Based Persistence"**
  *Proceedings of the 4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (**INFLOW**)*, Savannah, GA, USA, November 2016.
  [Slides (pptx) (pdf)]



NVMOVE: Helping Programmers Move to Byte-Based Persistence

Himanshu Chauhan * — UT Austin
Irina Calciu — VMware Research Group
Vijay Chidambaram — UT Austin
Eric Schkufza — VMware Research Group
Onur Mutlu — ETH Zürich
Pratap Subrahmanyam — VMware

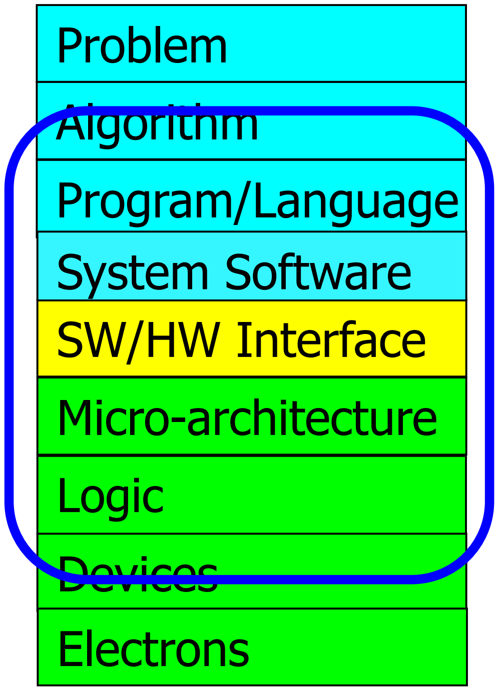
The Future of Emerging Technologies is Bright

- Regardless of challenges
  - in underlying technology and overlying problems/requirements

Can enable:

- Orders of magnitude improvements

- New applications and computing systems



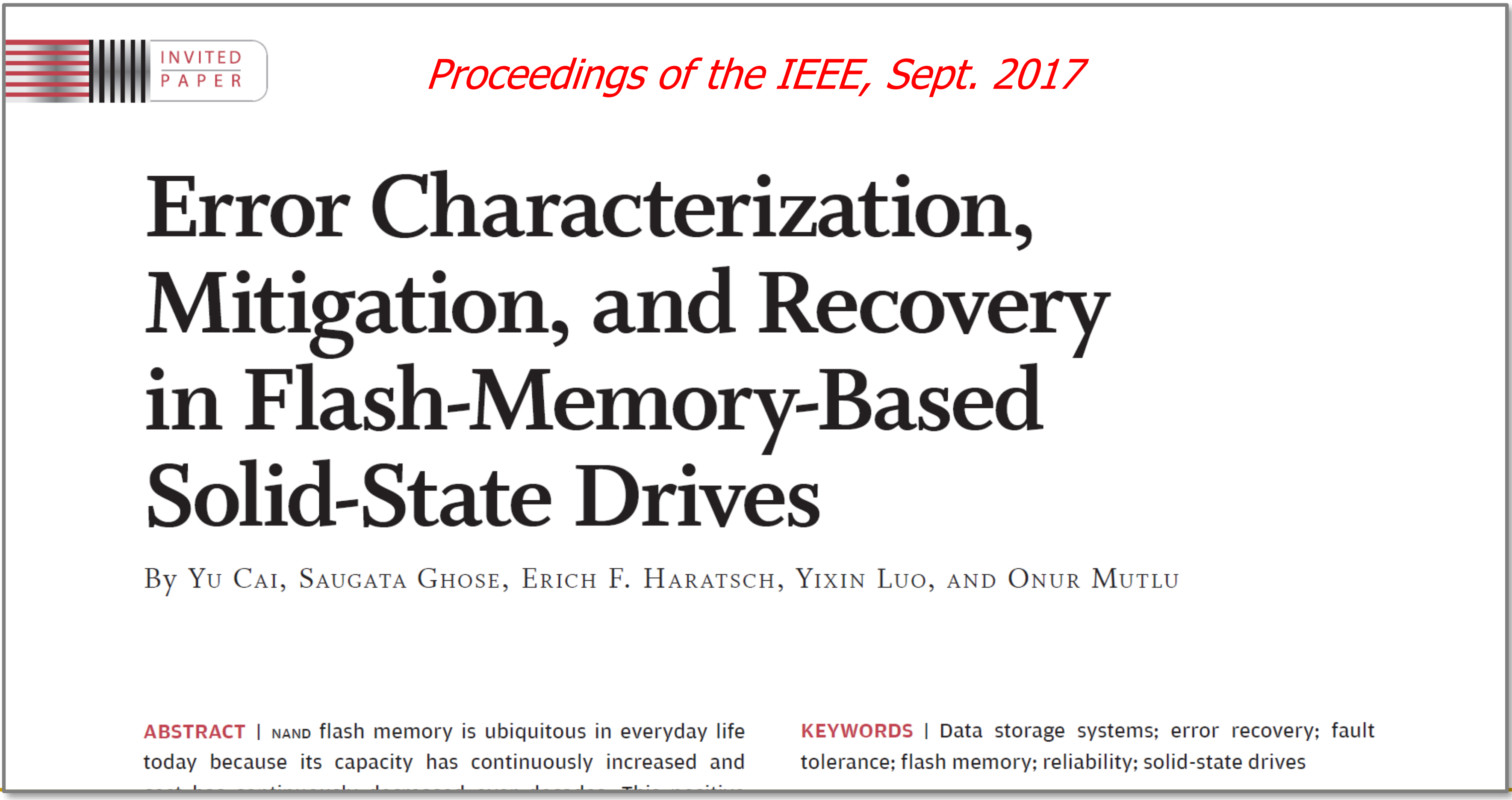
Yet, we have to

- Think across the stack

- Design enabling systems

If In Doubt, Refer to Flash Memory

- A very "doubtful" emerging technology
  - for at least two decades



INVITED PAPER

Proceedings of the IEEE, Sept. 2017

Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives

By Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu

ABSTRACT | NAND flash memory is ubiquitous in everyday life today because its capacity has continuously increased and

KEYWORDS | Data storage systems; error recovery; fault tolerance; flash memory; reliability; solid-state drives

SAFARI

https://arxiv.org/pdf/1706.08642

Computer Architecture
Lecture 14a: Emerging Memory Technologies II

Prof. Onur Mutlu
ETH Zürich
Fall 2018
1 November 2018