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

CPU

MEMORY

STORAGE

DRAM

FILE I/O

Ld/St

VOLATILE

FAST

BYTE ADDR

NONVOLATILE

SLOW

BLOCK ADDR
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

System crash can result in permanent data corruption in NVM

Persistent Memory System
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
Add a node to a linked list

1. Link to next
2. Link to prev

CRASH CONSISTENCY PROBLEM
System crash can result in inconsistent memory state
OUTLINE

- Crash Consistency Problem
- Current Solutions
- ThyNVM
- Evaluation
- Conclusion
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;
}
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;
}
void TM_hashtable_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;
}

Manual declaration of persistent components
Manual declaration of persistent components

```c++
void TM_hashtable_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

```c
void TM_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*)TMLIST_FIND(chain, &updatePair);
    pair->second = data;
}
```

Need a new implementation

Third party code can be inconsistent
CURRENT SOLUTIONS

Manual declaration of persistent components

```c
void TM_hashtable_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
OUR GOAL

Software transparent consistency in persistent memory systems

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

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;
}
ThyNVM APPROACH

Periodic checkpointing of data managed by hardware

Transparent to application and system
CHECKPOINTING OVERHEAD

1. Metadata overhead

Metadata Table

- Working location
  - X
  - Y
- Checkpoint location
  - X'
  - Y'

2. Checkpointing latency

Epoch 0

Epoch 1

STALLED

STALLED
1. METADATA AND CHECKPOINTING GRANULARITY

- **PAGE GRANULARITY**
  - One Entry Per Page
  - Small Metadata

- **BLOCK GRANULARITY**
  - One Entry Per Block
  - Huge Metadata
2. LATENCY AND LOCATION

DRAM-BASED WRITEBACK

1. Writeback data from DRAM

2. Update the metadata table

Long latency of writing back data to NVM
2. LATENCY AND LOCATION

NVM-BASED REMAPPING

2. Update the metadata table

- Working location
- \( Y \)

3. Write in a new location

DRAM

NVM

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

DRAM
NVM

GOOD FOR STREAMING WRITES
GOOD FOR RANDOM WRITES

High write locality pages in DRAM, low write locality pages in NVM
2. OVERLAPPING CHECKPOINTING 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

RANDOM

Normalized Write Traffic To NVM

Journal  Shadow  ThyNVM

LOW  LOW

SEQUENTIAL

Normalized Write Traffic To NVM

Journal  Shadow  ThyNVM

LOW  LOW

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

ThyNVM adapts to both access patterns
OVERLAPPING CHECKPOINTING AND EXECUTION

**RANDOM**

- Journal
- Shadow
- ThyNVM

**SEQUENTIAL**

- Journal
- Shadow
- ThyNVM

**Can spend 35-45% of the execution on checkpointing**

**Stalls the application for a negligible time**
PERFORMANCE OF LEGACY CODE

Provides consistency without significant performance overhead

Within -4.9%/+2.7% of an idealized DRAM/NVM system
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
ThyNVM
Enabling Software-transparent Crash Consistency In Persistent Memory Systems

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

Available at http://persperper.com/thynvm