Transaction Cache: A Persistent Memory Acceleration Approach

Jishen Zhao

July 3, 2017

MPSoC'17

Typical memory and storage hierarchy:

Memory Fast access to working data

Storage Data persistence

Persistent memory:

Fast memory interface + persistence

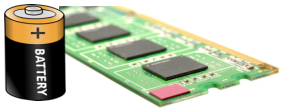
Dry it

Persistent memory is ~~coming!~~ here!

Hardware – Nonvolatile random-access memories (NVRAMs)

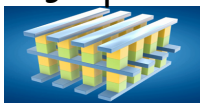
Not flash memory…

Battery-backed DRAM



NV-DIMM

3D Xpoint



DDR3 Compatible MRAM

DRAM w/ Ultra-capacitor

Software – Persistent-memory-aware system software

Persistent Memory Support in OS

Using DAX in Windows

DAX Volume Creation

```
→ Format n: /dax /q
→ Format-Volume -DriveLetter n -
```

DAX Volume Identification

Is it a DAX volume?

```
→ call GetVolumeInformation("C:\", ...)
→ check lpFileSystemFlags for FILE_DAX_VOLUME (0x20000000
```

Microsoft

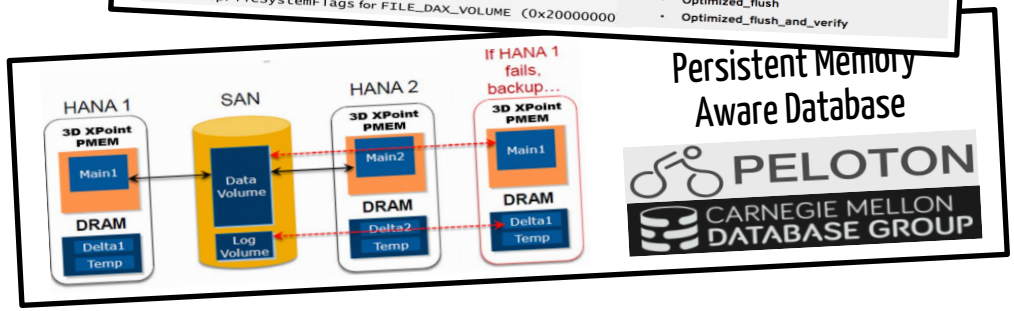
NVM.PM.FILE ACTIONS

- Implemented by kernel
  - Optimized_flush
  - Optimized_flush_and_verify

redhat

Persistent Memory File Systems

File system	Metadata atomicity	Data atomicity	Mmap Atomicity [1]
BPFS	Yes	Yes [2]	No
PMFS	Yes	No	No
Ext4-DAX	Yes	No	No
SCMFS	No	No	No
Aerie	Yes	No	No
NOVA	Yes	Yes	Yes

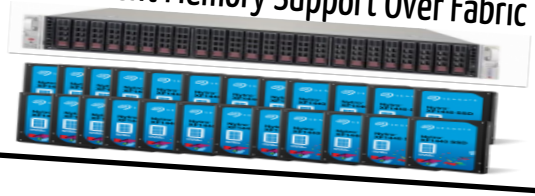


Persistent Memory Aware Database

PELOTON

CARNEGIE MELLON DATABASE GROUP

Persistent Memory Support Over Fabric



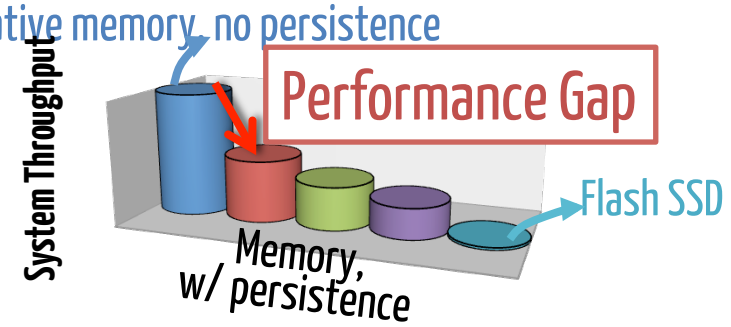
Persistent memory is ~~coming!~~ here!

...but unlocking its full potential isn't easy

Dry it with "Logs"

And copy-on-write, checkpointing, etc.

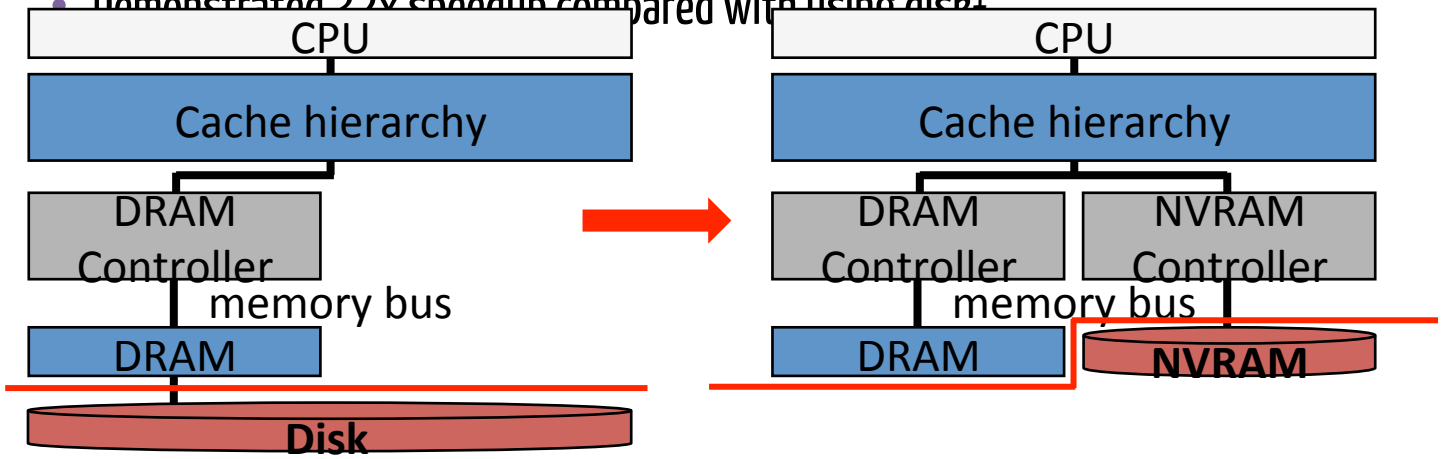
- Persistence
  - Used to be a property of storage systems
  - Now needs to be maintained in the memory system



[Zhao +, MICRO'13]

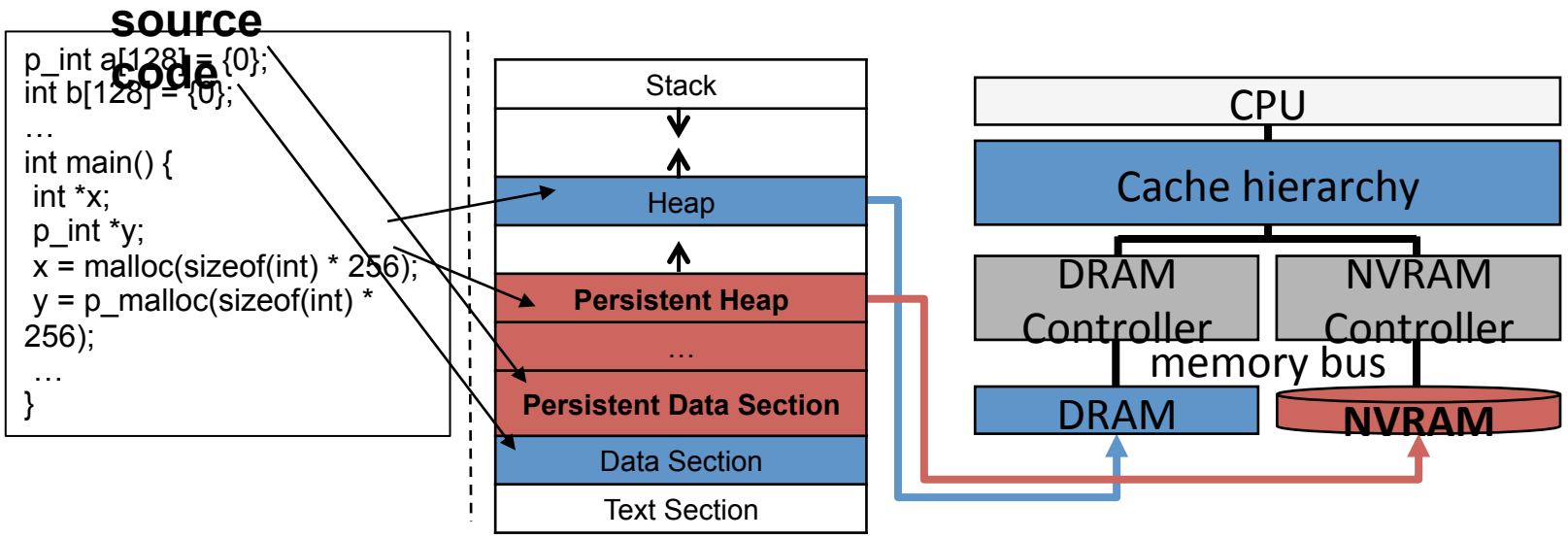
Persistent memory (PM)

- The volatile/persistence **boundary** is moved from **memory/disk** to **cache/memory**
  - Advantages over traditional disk
    - Overall access latency is faster
      - Demonstrated 22x speedup compared with using disk[1]



Persistent memory (PM)

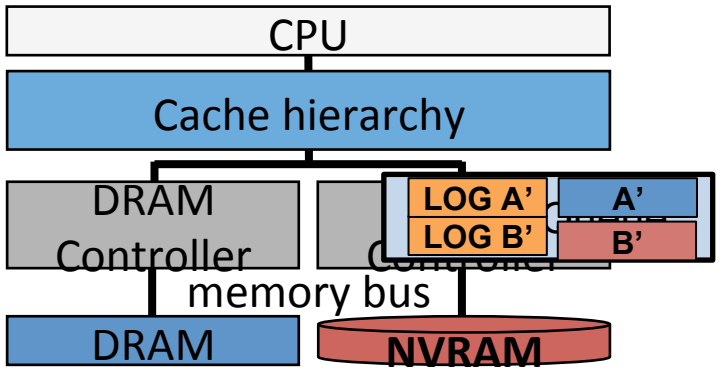
- Hybrid memory (DRAM + NVRAM)
  - Can adapt to different workload requirement
    - In this work, we concern the path from cache toward NVRAM



Related work design

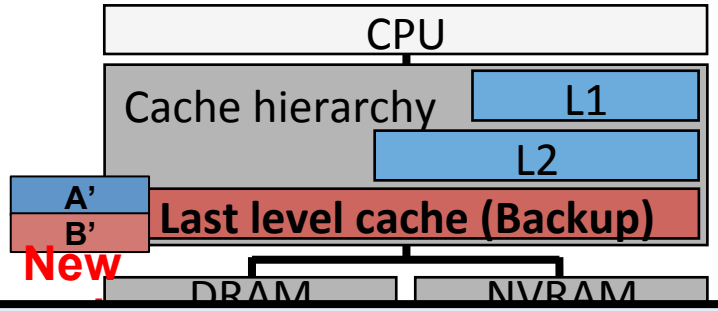
- **Epoch barrier**
  - Apply new software/hardware primitive (epoch barrier) to let cache controller and memory know the ordering rule
    - Cache and memory controller follow the ordering rule in parallel with CPU execution without CPU stall
- **Problem**
  - Still constrain cache eviction

  and memory scheduling method



Related work design

- **Kiln**
  - Utilize nonvolatile last level cache as backup to eliminate logging operations
  - To avoid CPU stall, cache controller will flush data toward LLC and ensure ordering in parallel with CPU execution
- **Problem**
  - Need to flush data toward LLC to ensure ordering
    - Result to more requests in cache hierarchy
  - Maintain backup at LLC will affect



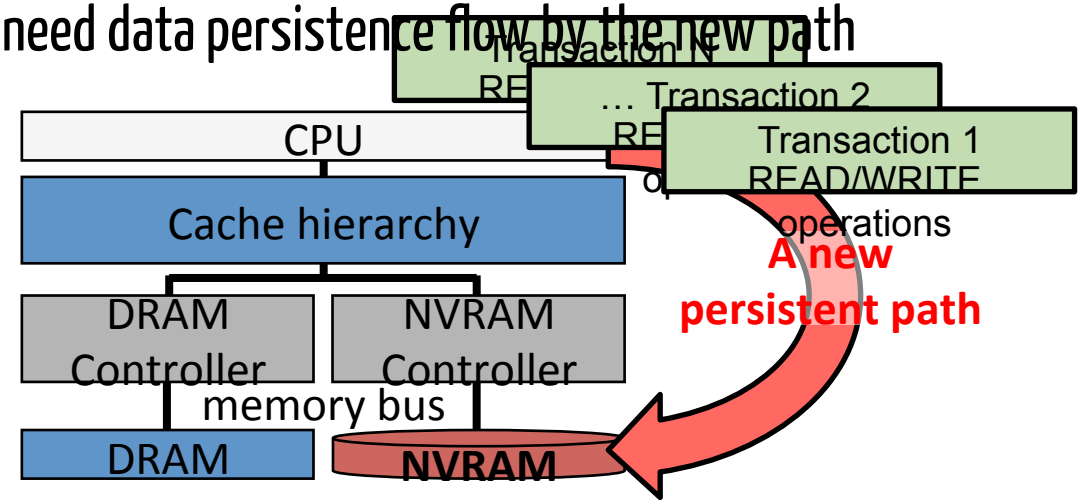
Related work doesn't totally solve CPU stall problem but just propagate overhead toward cache hierarchy or memory hierarchy

[Ki

rsior

In this work

- A new path is provided to eliminate logging overhead and free the original architecture from ordering constraint
  - Doesn't constrain the original architecture
    - Data writes that need data persistence flow by the new path



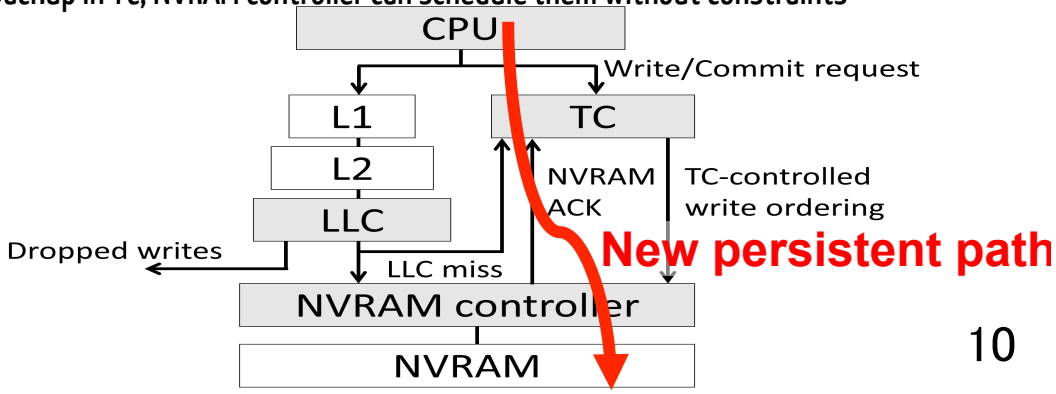
Transaction cache design

- Provide a new persistent path via **nonvolatile hardware buffer, called transaction cache (TC)**
  - With non-volatility
    - **Eliminate logging operations overhead**
      - Data that flows into this persistent path will naturally back up in TC
  - With buffer capability
    - **Free CPU from ordering overhead**
      - After issuing write toward cache hierarchy (L1 + TC), CPU can continue to execute without stall (the same as normal write)
  - Add aside of original cache hierarchy (New path)
    - **Free cache and memory controller from ordering overhead**
      - ① Original cache controller doesn't need to handle the write ordering but drops write back request, TC will control the write ordering
      - ② Data issued toward NVRAM will have backup in TC, NVRAM controller can schedule them without constraints

```
Transaction {
        LOG_A=log(&A, 0)
        LOG_B=log(&B, 1)
        sfence
        clflushopt &LOG_A
        clflushopt &LOG_B
        sfence
        pcommit
        sfence
        write A=0
        write B=1
}
```

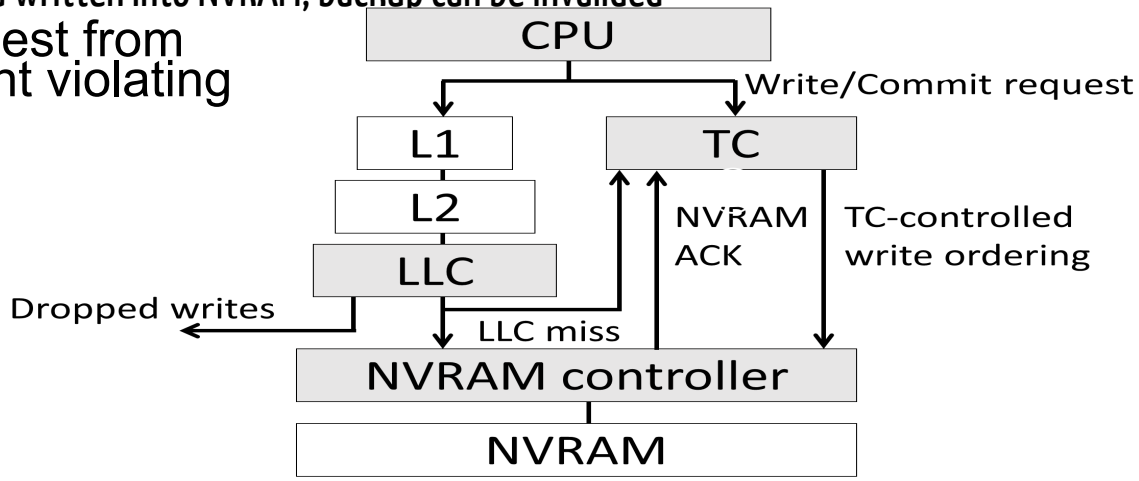
⇒

```
Transaction {
        write A=0
        write B=1
}
```



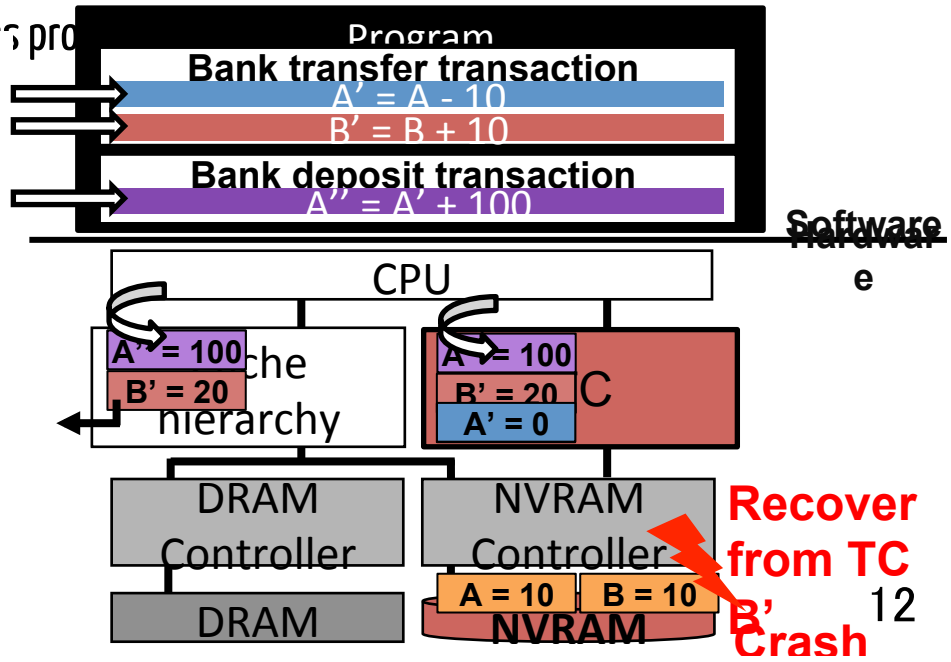
Transaction cache design – data flow

- Transaction cache serves as backup store to eliminate logging operations and control the write ordering
  - ① Data writes from CPU will be issued toward both original cache hierarchy and TC
  - ② TC serves as FIFO, insert and write backs data as program order in parallel with CPU execution
    - TC write backs the data of a transaction after the transaction commits (backs up all its data into TC for atomicity)
      - Only after data written into NVRAM, backup can be invalided
- Nonvolatile eviction request from LLC is dropped to prevent violating consistency



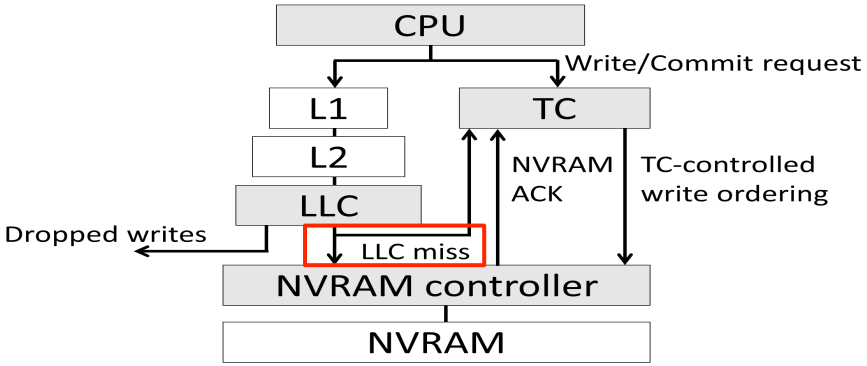
Transaction cache design – data flow

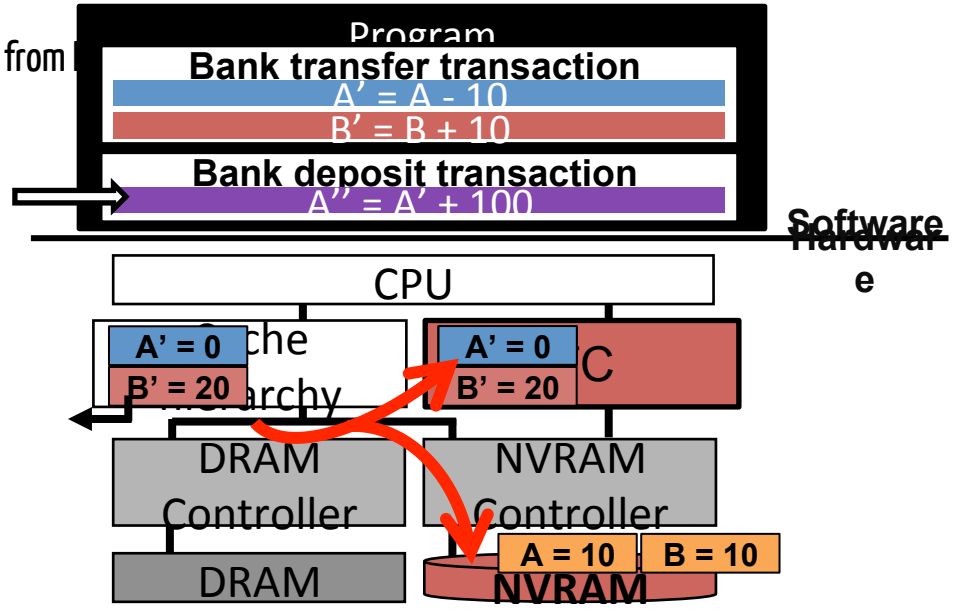
- Transaction cache serves as backup store to eliminate logging operations and control the write ordering
  - ① Data writes from CPU will be issued toward both original cache hierarchy and TC
  - ② TC serves as FIFO, insert and write backs data as pro
    - TC write backs the data of a transaction after the transaction commits (backs up all its data into TC for atomicity)
      - Only after data written into NVRAM, backup can be invalided
- Nonvolatile eviction request from LLC is dropped to prevent violating consistency
  - **Ex. A = 10, B = 10**
    - A program with a transfer transaction from A to B + a deposit transaction for A



Transaction cache design – serve read

- Because nonvolatile eviction request from LLC is dropped
  - Miss request on these dropped data cannot be served from NVRAM if data are still in TC and not written back
    - To serve the request, nonvolatile miss request from
      - Use data from TC first





TC software/hardware modification

With eliminated logging operations and hardware ensured ordering, software provide

Compiler will transfo… them into CPU …

Last level cache(LLC)

- Issue miss request toward NVRAM and TC
- Simply drop the nonvolatile eviction requests, which has persistent flag = 1

Memory controller

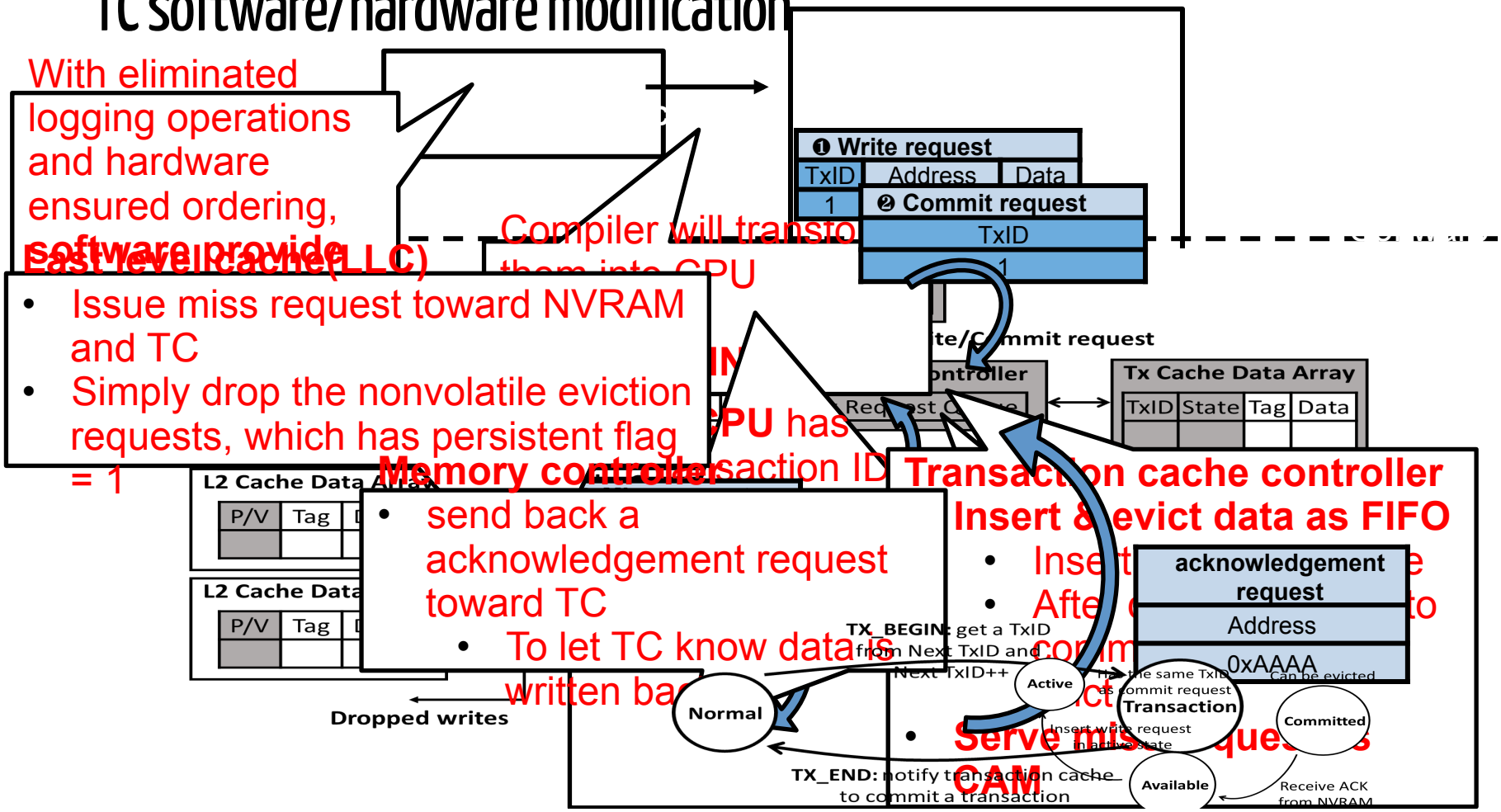
- send back a acknowledgement request toward TC
  - To let TC know data is … written ba…

Transaction cache controller

Insert & evict data as FIFO

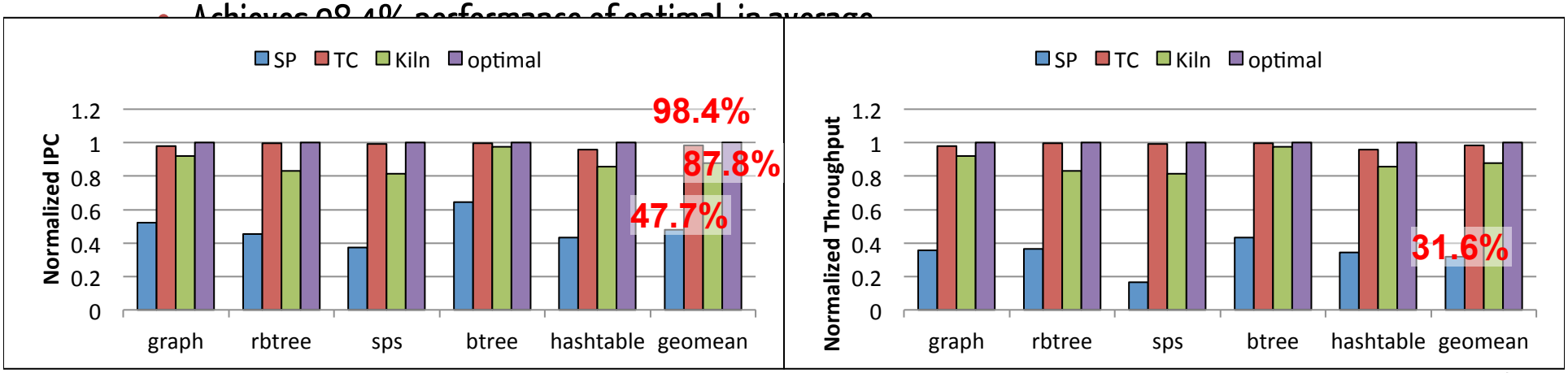
- Inse…
- Afte… …to

- **Serve mi… requ…s CAM**



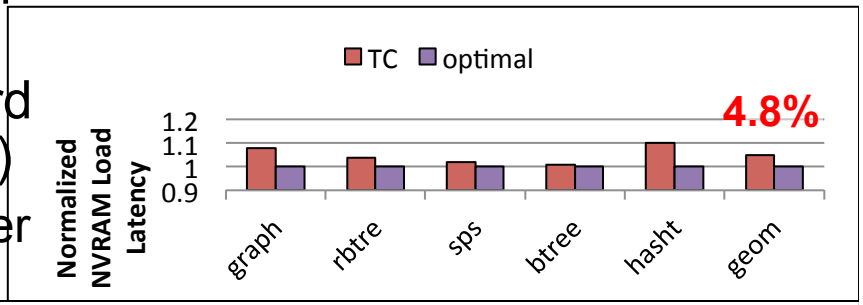
Performance results (IPC & Throughput)

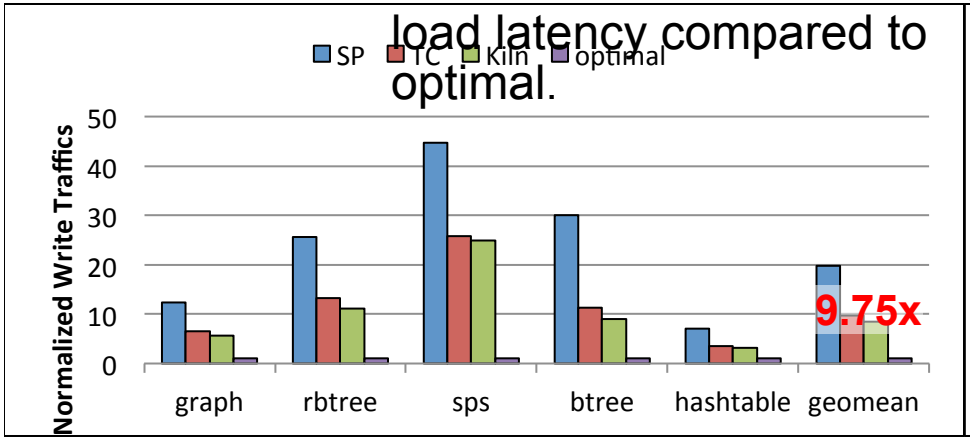
- **IPC: instructions/cycle, Throughput: transaction/sec**
- **SP**
  - Achieves 47.7%, 31.6% performance of optimal by IPC and throughput
- **TC**
  - Achieves 98.4% performance of optimal in average

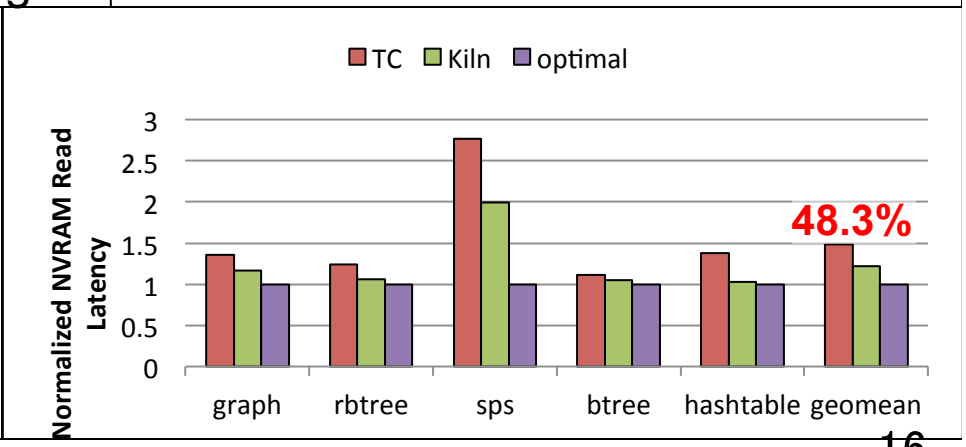


The degradation reason of TC

- More write traffic is generated to ensure persistence in NVRAM
  - In average, **TC** has 9.75x write traffic of optimal
    - More write traffic results to longer NVRAM read latency and thus load latency. (read request toward NVRAM, load request from CPU)
      - In average, **TC** has 48.3% longer NVRAM read and 4.8% longer load latency compared to optimal.

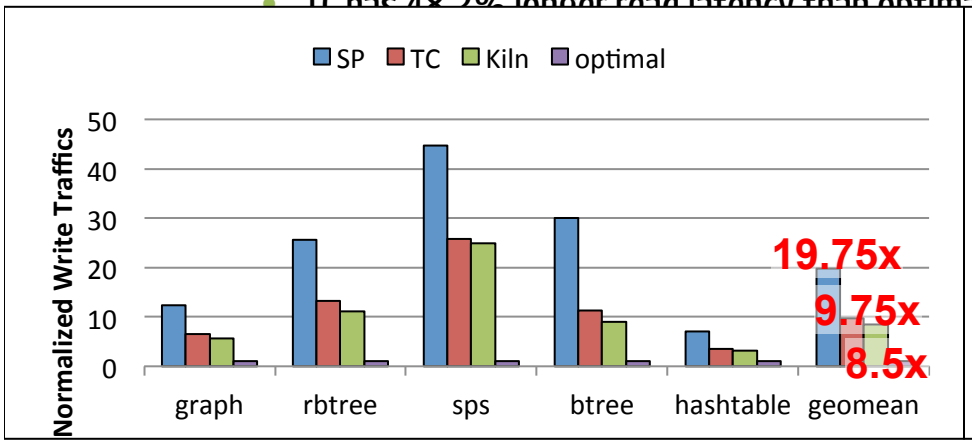


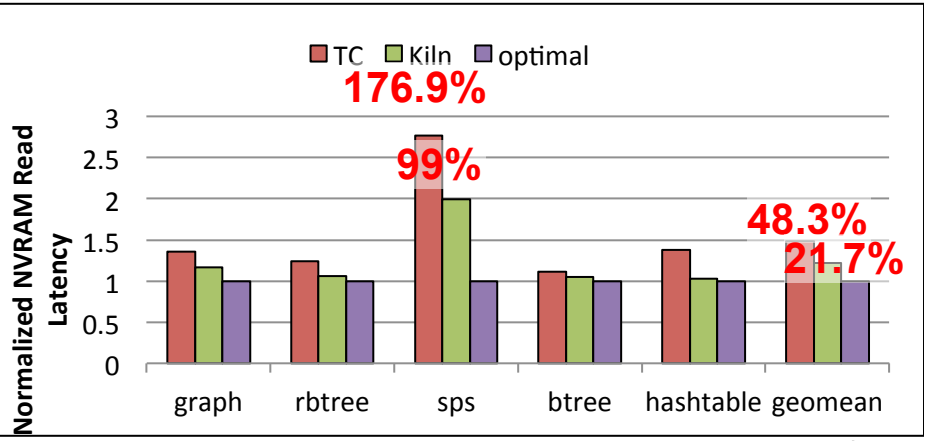




The degradation reason of TC

- **SP** has 19.75x write traffic of optimal for persistence, logging
- **Kiln** has 8.5x write traffic of optimal for persistence
- Compared to **Kiln**, **TC** has 14.8% more write traffic.
  - Because **Kiln** utilize nonvolatile LLC as backup, transaction data will buffer in LLC
    - TC has 48.3% longer read latency than optimal





Conclusion

- To solve persistence overhead, method applied by related work is to propagate persistence overhead toward cache and memory hierarchy
- In this work, an efficient hardware mechanism is proposed to provide a new persistent path
  - Utilize additional nonvolatile hardware to eliminate extra backup operations.
  - Free the original hardware architecture from ensuring the write ordering
- Experimental results show that our efficient hardware mechanism achieve the close performance of the optimal case without data persistent guarantee (98.4%).