# Leave the Cache Hierarchy Operation as It Is: A New Persistent Memory Accelerating Approach[*]

Chun-Hao Lai[¶] Jishen Zhao[‡] Chia-Lin Yang[¶ †]
[¶]National Taiwan University, [‡]University of California, Santa Cruz, [†]Academia Sinica
{r03922024, yangc}@csie.ntu.edu.tw, jishen.zhao@ucsc.edu

## ABSTRACT

Persistent memory places NVRAM on the memory bus, offering fast access to persistent data. Yet maintaining NVRAM data persistence raises a host of challenges. Most proposed schemes either incur much performance overhead or require substantial modifications to existing architectures.

We propose a persistent memory accelerator design, which guarantees NVRAM data persistence by hardware yet leaving cache hierarchy and memory controller operations unaltered. A nonvolatile transaction cache keeps an alternative version of data updates side-by-side with the cache hierarchy and paves a new persistent path without affecting original processor execution path. As a result, our design achieves the performance close to the one without persistence guarantee.

## Keywords

Persistent memory; Nonvolatile memory; Data consistency; Atomicity; Durability; Persistence

## 1. INTRODUCTION

Emerging nonvolatile memory (NVRAM) technologies such as phase-change memory (PCM), spin-transfer torque RAM (STTRAM), resistive RAM (RRAM), and Intel and Micron's 3D XPoint [7] technology promise to revolutionize I/O performance. NVRAMs can be integrated into computer systems in various manners. One of the most exciting proposals deploys NVRAMs on the processor-memory bus, producing a hybrid of main memory and storage systems – namely persistent memory – which offers fast memory access and data persistence in a single device [5, 23].

Supporting persistence in memory-bus-attached NVRAM presents a host of opportunities and challenges for computer and system architects. System failures (crashes and inopportune power failures) can interrupt data updates across the memory bus, leaving NVRAM data structures in an partially updated, inconsistent state mixed with old and new values. Moreover, modern CPU caches and memory controllers can reorder stores to memory to improve performance, which further intricates the persistence support. Taking an example where a program inserts a node in a linked list, software can issue the node value update followed by the corresponding pointers updates. However, after being reordered, stores to the pointer can arrive at the NVRAM before those to the nodes. If system crashes in the middle, the linked list will be corrupted with dangling pointers [23]. As such, persistent memory systems need to place careful control of NVRAM data versions and ensure that the order of writes arriving at NVRAM.

A large body of recent studies focus on ensuring persistence in memory. But most works either incur substantial performance overhead to the memory system or require substantial modifications to existing processor hardware. For example, most software-based persistence mechanisms – e.g., NVRAM file systems and libraries – explicitly maintain multiple versions of data by logging or copy-on-write [2, 3, 20]. In addition, these schemes typically enforce write ordering by cache flushes and memory barriers (e.g., using `clflush`, `clwb`, and `mfence` instructions available in Intel x86 ISA) [8, 20]. As NVRAMs offer vastly improved bandwidth and latency, over disks and flash, such software-based schemes add significant performance overhead to memory systems and squander the improved performance that NVRAM offers [5, 20]. Most hardware-based mechanisms require non-trivial modifications to the processor, e.g., the cache hierarchy and memory controllers [11, 4, 9, 15, 23]. These modifications can reduce the flexibility of modern processor design and substantially increase the hardware implementation cost.

Our goal in this paper is to provide persistence guarantee in NVRAM, while minimizing the modification to the existing processor hardware implementation. To achieve our goal, we propose a persistent memory accelerator design, where the major component is a nonvolatile transaction cache that buffers the stores of in-flight transactions issued by CPU cores. The transaction cache also maintains the order of these stores to be written into NVRAM. By maintaining data persistence in the transaction cache, we create a side path in the processor to maintain data persistence. As such, we eliminate the persistence functions in software (e.g., logging and copy-on-write) and leave the cache hierarchy and memory controller operation as they are. Furthermore, we implement the persistent memory accelerator as a stand-alone module in the processor, enabling flexible physical implementation of the processor. As a result, our design achieves 98.5% the performance of the optimal case that does not provide persistence support.

## 2. CHALLENGES OF MAINTAINING PERSISTENCE IN MEMORY

Figure 1 shows an example of a system configuration integrated with persistent memory. Adding a persistent memory in the system

[*]This work is supported in part by research grants from the Ministry of Science and Technology of Taiwan (MOST- 105-2221-E-002-156-MY2, MOST-105-2622-8-002-002),and sponsored by Macronix Inc., Hsin-chu, Taiwan (105-S-C05). Zhao was supported by NSF 1652328.

*DAC '17, June 18-22, 2017, Austin, TX, USA*
© 2017 ACM. ISBN 978-1-4503-4927-7/17/06. . . 15.00
DOI: http://dx.doi.org/10.1145/3061639.3062272

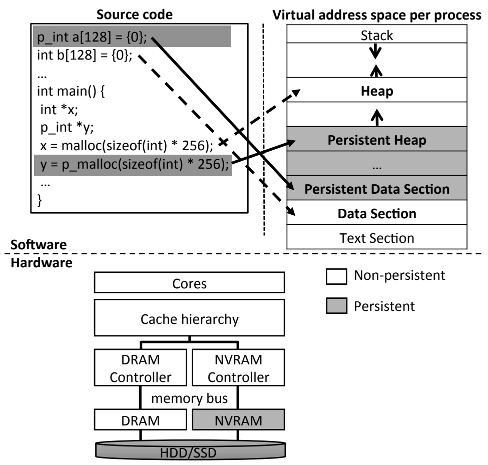


**Figure 1: Overview of memory systems integrated with persistent memory.**

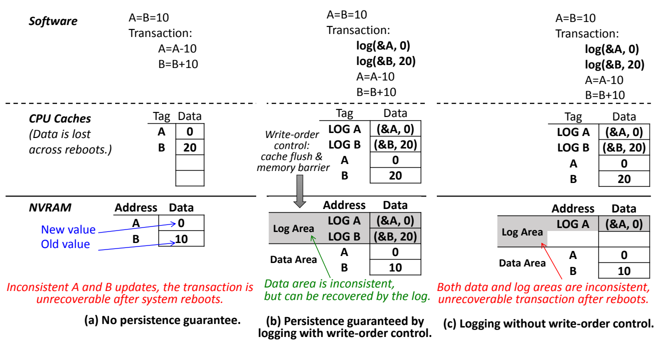

Software
A=B=10
Transaction:
A=A-10
B=B+10
log(&A, 0)
log(&B, 20)
CPU Caches
(Data is lost across reboots.)
Tag
Data
LOG A
LOG B
(&A, 0)
(&B, 20)
Write-order control: cache flush & memory barrier
NVRAM
Address
New value
Old value
Log Area
Data Area
Inconsistent A and B updates, the transaction is unrecoverable after system reboots.
Data area is inconsistent, but can be recovered by the log.
Both data and log areas are inconsistent, unrecoverable transaction after reboots.
(a) No persistence guarantee.
(b) Persistence guaranteed by logging with write-order control.
(c) Logging without write-order control.

**Figure 2: The performance overhead of log-based persistent memory.**

may seem as simple as implementing another memory region, but several challenges can prevent the immediate wide-spread adoption of this emerging technique.

Current memory systems have no knowledge about data persistence; they assume that the data stored in memory is lost – and therefore unusable – across system reboots. Therefore, conventional systems only maintain memory consistency that guarantees a consistent view of data stored across the cache hierarchy and main memory during application execution.

Rather, persistent memory systems need to guarantee the consistency of data stored in NVRAM alone across system reboots. To this end, system software, hardware or their combination needs to maintain versioning of critical data updates by logging [20] or copy-on-write [3], along with write-order control (e.g., cache flushes and memory barriers) that prevents caches and memory controllers to reorder the stores issued by applications [23]. As a result, providing persistence guarantee in memory can impose substantial inefficiencies in current memory systems, in terms of performance overhead, implementation cost and flexibility as we discuss below. To the best of our knowledge, our design is the first that tackles all of the challenges.

## 2.1 Performance Overhead of Persistence Support

One key challenge of persistent memory design is the performance overhead of persistence support introduced to existing memory systems. Many persistent memory systems maintain a log of NVRAM updates in addition to the stores to the original data. For example, differently from traditional program without persistence guarantee (Figure 2(a)), to ensure persistence of data structures A and B, persistent memory systems need to execute the logging instructions `log(address, new value)`, generating extra instructions that read and write the addresses and values of A and B (Figure 2(b)). As such, even if system failures leave the original data or the log partially updated, the data structures can be recovered by the other. Furthermore, write-order control mechanisms (e.g., cache flushes and memory barriers) need to enforce that the log updates arrive at NVRAM before corresponding original data updates. Otherwise, upon system failures, the original data in NVRAM can be partially updated while the corresponding log updates remaining in caches is lost (Figure 2(c)) – the NVRAM data structures will be unrecoverable after system reboots. Copy-on-write-based persistent memory systems maintains multi-versioning and write-order control in a similar manner [3]

Multi-versioning and write-order control are major sources of the performance overhead. Multi-versioning requires duplication of data and therefore can significantly increase NVRAM write traffic [23]. Write-order control will cancel out the performance optimization provided by the reordering of caches and memory controllers. As a result, the performance of persistent memory systems can be much lower than optimal memory systems that does not support persistence.

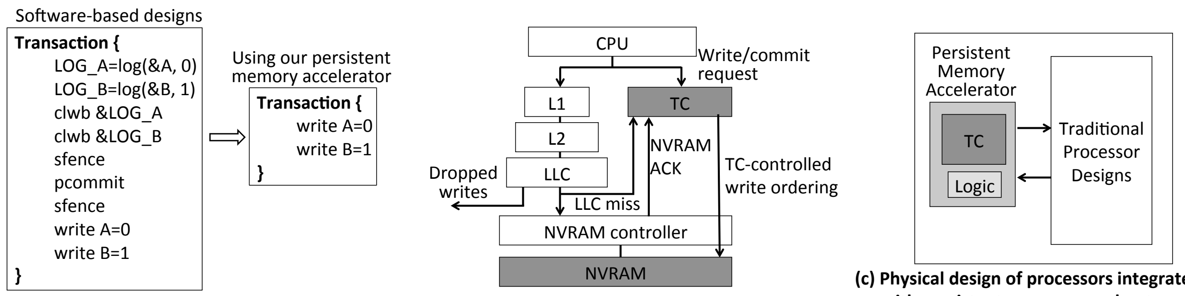
## 2.2 Implementation Cost and Flexibility of Persistence Support

The performance overhead of maintaining persistence in memory has been noted by other researchers. Common approaches for avoiding this overhead is to modify the processor and memory hardware to exploit hardware's support for data persistence [4, 23]. However, this approach can impose substantial implementation overhead on top of existing designs. Without careful consideration, the processor and memory modifications can significantly increase the cost and reduce the flexibility of hardware implementation. For instance, most prior works require to modify cache hierarchy (e.g., replacing the last-level cache with NVRAM technologies, interfering cache control flows, the tag arrays, and cache coherence mechanisms) [3, 10, 12, 23] and memory controllers (e.g., modifying memory scheduling policy) [4, 6, 15]. Such hardware reimplementation requires new processor architecture or memory controller design and increases manufacturing hardware cost. Furthermore, mingling persistence support with processor or memory operation can increase hardware implementation complexity. For example, doing so can increase the number of states in the cache operation state machine [23] and need additional function supports in memory controller for persistence ordering requirement [4], intricating the caching and memory scheduling schemes.

## 3. PERSISTENT MEMORY ACCELERATOR

**Architecture Design Overview.** The goal of our design is to provide a high-performance, easy to implement, and easy to use persistent memory design[1]. Yet instead of mingling the persistence support with caching or memory controller operations, we provide a side data path for accelerating persistent memory updates. Figure 3 shows an overview of our architecture design. The key component of our persistent memory accelerator is a nonvolatile transaction cache (TC), which is deployed side-by-side with the cache hierarchy. The TC serves as a FIFO of writes in transactions; whenever a transaction commits, the TC will issue the corresponding stores to the NVRAM. CPU caches can operate as it is without maintaining data persistence; in order to prevent the reordered CPU cache blocks to contaminate the persistent data in NVRAM, we drop the last-level

[1]Memory systems may consist of DRAM and NVRAM regions, where the NVRAM stores critical data structures and the DRAM stores temporary data that does not require persistence [23]. Our work focuses on efficiently accelerating data persistence in NVRAM rather than the DRAM access.


Software-based designs
```
Transaction {
      LOG_A=log(&A, 0)
      LOG_B=log(&B, 1)
      clwb &LOG_A
      clwb &LOG_B
      sfence
      pcommit
      sfence
      write A=0
      write B=1
}
```
Using our persistent memory accelerator
```
Transaction {
      write A=0
      write B=1
}
```

**(a) Example code.** **(b) Architecture design.** **(c) Physical design of processors integrated with persistent memory accelerator.**

**Figure 3: Transaction cache overview.**

cache write-backs – these blocks are simply discarded after being evicted out of the last-level cache. By maintaining persistence in the TC, we also allow the NVRAM controller to operate as it is; the NVRAM controller only needs to send an acknowledgment message back to the TC, after the persistent writes are written into NVRAM.

**Benefits.** Our design offers several promising benefits. First, *performance benefits* – we allow CPU caches and memory controllers to operate as it is, without being interfered by persistence support. Doing so can maximize system performance, while reducing hardware implementation complexity. Second, *flexible hardware implementation* – by providing persistence support at the side data path, our design allows flexible and low-cost processor implementation. For example, the capacity of the transaction cache can be flexibly configured based on the transaction sizes of the processor's target applications. Third, *enabling modular hardware design and implementation* – the transaction cache and the rest of the processor can be manufactured on two separate dies and later integrated on a silicon interposer [18]. As such, new processors with persistence support can adopt legacy processor designs. As interposer-based integration is adopted by increasingly large population of processor designs, such modular hardware implementation can be an attractive solution for persistent memory systems. Finally, *ease of use* – our software interface only involves transaction definitions similar to commodity ISAs [8].

**The Persistence Support provided by Our Design.** Without compromising performance, our persistent memory accelerator provides multi-versioning of data and write-order control to ensure data persistence.

- **Multiversioning.** We make copies of all persistent writes of each transaction in the TC. Only after buffering all the persistent writes of a transaction, the TC will issue these writes to the NVRAM. In addition, only after updating the NVRAM, the buffered writes in the TC can be dropped. If system failures happen before all the writes of a committed transaction are written into NVRAM, we can recover the data using the buffered writes in the TC; If system failures happen before the TC even issues any buffered writes to the NVRAM or after all the buffered writes of a committed transaction are written into the NVRAM, data in the NVRAM is intact and consistent. Note that after completing all the writes, the NVRAM controller needs to send an acknowledgment message back to the TC to indicate that the TC can drop the buffered data.
- **Write-order Control.** Write-order control is naturally supported by the TC. Different from the writes that go through CPU caches, those go through the TC – which is a FIFO – will not be reordered. The persistent writes from CPU are inserted into and evicted from the TC in FIFO order, which simply conforms to the write ordering requirement.

**Persistent Memory Accelerator Working Flow.** With our persistent memory accelerator, persistent writes will go through the following flow. CPU sends the data and commit information of each transaction to the TC in a non-blocking manner (without stalls). Instead of explicitly performing logging or copy-on-write, the data temporarily buffered in TC serves as an alternative version of the original data; this buffered data can be used to recover the original data when system failures happen. Instead of employing cache flushes and memory barriers, the write ordering requirement of persistent memory is also met by the TC. Corresponding persistent data to be evicted from the last-level cache will not written back to the NVRAM. Instead, they will be dropped to ensure that the persistent memory only contain the consistent data sent by the TC. After the discarding, if there are miss requests on the discarded cache lines, last level cache will grab the old version data from the NVRAM but the new one in the transaction cache and results to inconsistency execution results. Therefore, to serve a miss request, last level cache will issue miss requests toward not only the NVRAM but also the transaction cache to get the newest value. The TC will also ensure that different write requests of conflicted addresses (the same cache line address and the same row in a NVRAM bank) are issued to the NVRAM in program order.

## 4. PERSISTENT MEMORY ACCELERATOR IMPLEMENTATION

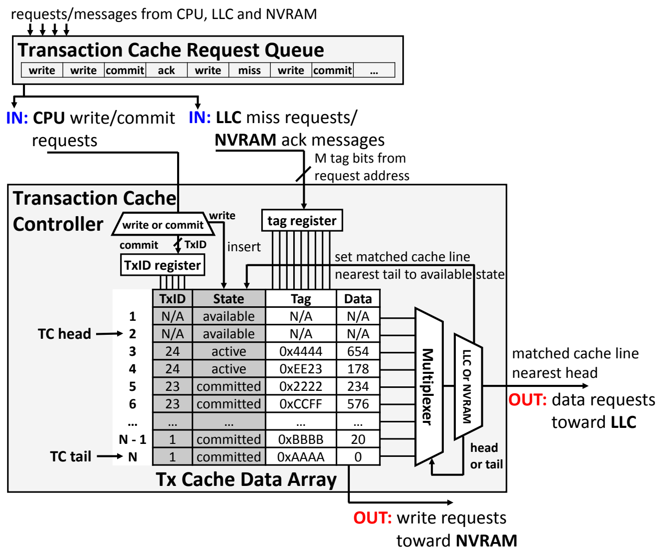
In this section, we present our implementation details on transaction cache architecture and its associated logic, software interface, other modifications in the processor, and a summary of our hardware cost.

### 4.1 Transaction Cache Implementation

In-order to maintain the transaction information of each cached entry, we implement the TC as a content-addressable, first-in/first-out (CAM FIFO) [14, 19]. The FIFO architecture is to match program ordering and the CAM architecture is to serve the miss requests or acknowledgment messages fast to get matched cache line entry in a single operation. Figure 4 shows the detailed hardware design of the transaction cache. The transaction cache consists of **transaction cache queue**, **transaction cache controller** and **transaction cache data array**. Transaction cache queue buffer requests from the other controllers (CPU, LLC and NVRAM). There are four types of requests:

- **Write requests of a transaction from CPU**: contain the transaction ID (TxID), data addresses and data value.
- **Commit request of a transaction from CPU**: contain only the transaction ID (TxID).
- **Miss requests from LLC**: contain the data address of the missed cache line.
- **Acknowledgment messages from NVRAM**: contain the address of the corresponding written back backup in TC.

In Figure 4, besides of the tag and data value, each cache line among the TC data array also records the transaction information to ensure transaction atomicity, the transaction ID (TxID) and the


requests/messages from CPU, LLC and NVRAM
Transaction Cache Request Queue
write
write
commit
ack
write
miss
write
commit
...
IN: CPU write/commit requests
IN: LLC miss requests/ NVRAM ack messages
M tag bits from request address
Transaction Cache Controller
write or commit
write
tag register
commit
TxID
insert
TxID register
set matched cache line nearest tail to available state
TC head
TC tail
Multiplexer
LLC Or NVRAM
matched cache line nearest head
OUT: data requests toward LLC
head or tail
Tx Cache Data Array
OUT: write requests toward NVRAM

| | TxID | State | Tag | Data |
|---|---|---|---|---|
| 1 | N/A | available | N/A | N/A |
| 2 | N/A | available | N/A | N/A |
| 3 | 24 | active | 0x4444 | 654 |
| 4 | 24 | active | 0xEE23 | 178 |
| 5 | 23 | committed | 0x2222 | 234 |
| 6 | 23 | committed | 0xCCFF | 576 |
| ... | ... | ... | ... | ... |
| N - 1 | 1 | committed | 0xBBBB | 20 |
| N | 1 | committed | 0xAAAA | 0 |

**Figure 4: Transaction Cache architecture.**

transaction state of corresponding transaction (State). Each cache line entry among the TC data array transitions in three state: **available**, **active** and **committed**. Write requests from CPU are inserted from the TC head and evicted at the TC tail. To serve a write request, first we check if the cache line entry pointed by the TC head is in the available state. If it is in the available state, we copy the tag information from the data addresses, data value and transaction ID of the write request into the cache line pointed by the TC head and set it in the active state. After that, the TC head points to the next entry. If it is not in the available state, then the transaction cache is full and we have to wait for data being written back into NVRAM and then we can insert new cache lines.

And to serve a commit request, the content-addressable TC data array are compared with the transaction ID of the commit request. All the matched cache lines with the same transaction ID of the commit request are set from the available state into the committed state. Committed cache lines are written back and issued toward the NVRAM in FIFO order (also program order).

After issuing the committed cache lines toward the NVRAM, the TC tail will not move or point to the next entry. Only after receiving the acknowledgment messages from the NVRAM, the committed cache line can be changed into the available state. And because different cache line entries may complete out of order from the NVRAM, at each time receiving the acknowledgment message, we will check if the cache line entry pointed by the TC tail is changed into the available state. If it is in the available state, the TC tail will continuously change its position until it points to the first entry that is not in the available state, which makes room for future write requests.

To serve acknowledgment messages from the NVRAM, because cache lines are issued to the NVRAM in FIFO order and different write requests of the same address are handled in the same issue order by the NVRAM controller, thus the content-addressable TC data array are compared with the address of the acknowledgment message and the matched one nearest to the TC tail is set into available state, which is issued toward and handled in the NVRAM first. On the other hand, to serve the miss requests from LLC, the content-addressable TC data array are compared with the address of the miss request and the matched one nearest the TC head is returned toward the LLC, which is the newest because data is inserted from the TC head in FIFO order.

**Transaction Cache Overflow.** The TC can overflow, if a transaction exceeds the TC capacity, i.e., the TC is filled up by active

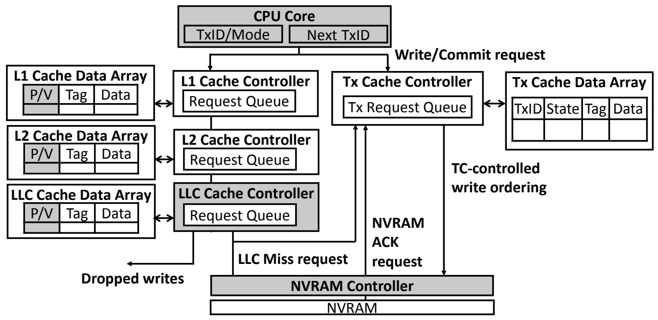


**Figure 5: Hardware architecture modification.**

updates before the transaction is completed. This can lead to CPU stalls and deadlocks. To address the issue, we adopt a fall-back path similar to prior studies [23, 21], i.e., allowing the overflowed transaction updates to be written back to the NVRAM with hardware-controlled copy-on-write. However, compared to prior studies [23], the buffering nature of the TC allows us to adopt a much simpler overflow detection mechanism. Instead of considering the possibility of overflows across various transaction states, we simply trigger the fall-back path once the TC is almost filled (e.g., 90% full). Furthermore, the TC is not susceptible for cache associativity overflows as prior studies do [23].

## 4.2 Software Interface

Software only needs to provide the transaction boundary information as following:

**Transaction {...}**

This function will be compiled into CPU primitives TX_BEGIN and TX_END. Encountering these primitives, CPU can execute in normal mode (without persistence guarantee) or transaction mode (with persistence guarantee). If CPU is executing the codes enclosed by TX_BEGIN and TX_END, then it is in the transaction mode. Otherwise, it is in the normal mode. As illustrated in Figure 5, CPU maintains a mode register that indicates whether it is in the normal mode or transaction mode. CPU also maintains a next transaction register that differentiates the execution of different transactions in programs. If the value of the mode register is non-zero, CPU is in the transaction mode. If the value of the mode register is zero, then the CPU is in normal mode. In the normal mode, CPU will only issue writes to L1 caches. In the transaction mode, CPU will issue write requests to both the L1 caches and the TC. Write requests issued to the L1 cache is tagged with persistent flag to let existing cache hierarchy differentiate persistent and volatile cache lines. Write requests issued to the TC contains the transaction ID. At encountering TX_BEGIN, CPU will copy the transaction ID from the next transaction ID into the mode register and enter the transaction mode. The next transaction register will automatically increase by one for the next transaction to be executed. And at encountering TX_END, CPU will issue commit request to the TC to let the TC know that a transaction commits, set the mode register as zero and enter the normal mode.

## 4.3 Other Modifications in the Processor

In Figure 5, each cache line in the existing cache hierarchy is modified to record if it is persistent cache line with one additional persistent or volatile flag (P/V). Persistent or volatile information is provided from the write requests issued from CPU. The LLC cache controller is modified to drop persistent eviction and have to issue miss requests toward both the transaction cache and the NVRAM controller and use the newer data from the transaction cache first. After handling a persistent write request, the NVRAM controller is modified to send an acknowledgment message with the same data

**Table 1: Summary of major hardware overhead.**

| Component | Type | Size |
|---|---|---|
| CPU TxID/Mode register | flip-flops | 6 bits |
| CPU Next TxID register | flip-flops | 6 bits |
| Cache P/V flag | SRAM | 1 bit |
| TxID in TC data array | STTRAM | 6 bits |
| State in TC data array | STTRAM | 1 bit |
| TC head/tail pointer | flip-flops | depends on TC data array |
| Other TC components (Multiplexer, tag register) | flip-flops | depends on TC data array |
| TC data array | STTRAM | flexible |

address back toward the transaction cache to let the TC know that a backup is written back into the NVRAM. The acknowledgment message can utilize the address bus the same as the read request to transfer the address information of the written back cache line.

## 4.4 Hardware Overhead

For the storage overhead showed in Table 1. For a core with a 4KB transaction cache size, if one cache line per transaction, there will be at most 64 executed transactions (4 * 1024 / 64) on a core, so all the CPU TxID/Mode register, next TxID register and TxID in TC data array needs 6 bits. And both P/V and state flag needs 1 bits. The total additional bits for a cache line in the TC data array are 7 bits (TxID + state) and the total additional bits for the existing cache hierarchy is 1 bit (P/V), which is much small compared to a cache line with 64 bytes and tag data. And with a multi-core system of 4 processors, the additional TCs size 16KB (4*4KB) are not much compared to the LLC size 64MB. Besides, the size of the transaction cache can be flexibly configured based on the transaction sizes of the processor's target applications.

For the logic modification overhead, the logic modification of the existing cache hierarchy is not much, which is to drop eviction requests, send finish signals and issue miss requests toward the TC. And the NVRAM controller only have to sent back the acknowledgment messages. All can be completed with simple logic. Besides, the transaction cache logic can simply adopt the logic of the CAM FIFO hardware structure to serve miss request from LLC or acknowledgment message from NVRAM with content-addressable data array and write or evict the cache line in FIFO order.

## 5. EXPERIMENTAL RESULTS

## 5.1 Experimental Setup

To evaluate the proposed transaction cache design, we conduct experiments with MARSSx86 [13] and DRAMsim2 [16]. MARSSx86 is a cycle-accurate full-system simulator that uses PTLsim [22] for CPU simulation on top of the QEMU [1] emulator. DRAMSim2 is a cycle-accurate simulator of main-memory systems. The DRAMSim2 is modified to model hybrid persistent memory system with DRAM and NVRAM through memory-bus. The cache model of MARSSx86 and NVRAM model of DRAMSim2 is modified to simulate related works and the transaction cache design of this work. We set up a multi-cores system with four cores (Intel Core i7 like). The memory system has two memory controller for NVRAM and DRAM respectively. Both the transaction cache and NVRAM apply STTRAM technology [17, 23]. The default transaction cache has 4KB size and 10.5ns latency. Table 2 describes the core and memory configurations.

To analyze the mechanisms, we implement five benchmarks similar to the benchmark suite used by NV-heaps [2]. These benchmarks have similar behaviors to the programs used in databases and file systems. The size of all manipulated key-value pairs in the benchmarks is 64 bits. We simulate each benchmark for 1.7 billion instructions. Table 3 lists the descriptions of the benchmarks.

**Table 2: Machine Configuration**

| Device | Description |
|---|---|
| CPU | 4 cores, 2GHz, 4 issue, out of order |
| L1 I/D | Private, 32KB/core, 1.5ns, 4-way |
| L2 | Private, 256KB/core, 4.5ns, 8-way |
| L3 (LLC) | Shared, 64MB, 10ns, 16-way |
| Transaction Cache (STTRAM) | Private, 4KB/core, Fully-Associative CAM FIFO [14, 19], 10.5ns [17] |
| 2 Memory Controllers | 8/64-entry read/write queue, read-first or write drain when the write queue is 80% full |
| NVRAM Memory (STTRAM) | 8GB, 4 ranks, 8 banks/rank, 65-ns read, 76-ns write [23] |
| DRAM Memory | DDR3 8GB, 4 ranks, 8 banks/rank |

**Table 3: Workloads**

| Name | Description |
|---|---|
| graph | Insert in an adjacency list graph. |
| rbtree | Search/Insert nodes in a red-black tree. |
| sps | Randomly swap elements in an array. |
| btree | Search/Insert nodes in a B+tree. |
| hashtable | Search/Insert a key-value pair in a hashtable. |

We compare four mechanisms:

- **SP (Software-supported Persistence)**:The software mechanism that supports write-ahead logging and ensures the write ordering through software instructions.
- **TC**:The transaction cache mechanism proposed in this work.
- **Kiln [23]**:A prior work that adopts a non-volatile last level cache and maintains writing orderings at the hardware level.
- **Optimal**:It represents the native execution without persistence overhead.

SP runs the transactions with logging operations and other three mechanisms run the transactions without logging operations. Both TC and Kiln applies hardware to ensure atomicity without logging operations.

## 5.2 Performance Evaluation

Figure 6 and Figure 7 show the IPC (instructions per cycle) and throughput (transactions per cycle) of various schemes normalized to the optimal. Software-supported persistence (SP) does impose significant overheads for maintaining persistence, achieving only 47.7% and 31.6% performance of the optimal case for both IPC and throughput metrics. The proposed transaction cache (TC) mechanism performs comparable to the optimal case for both IPC and throughput metrics (98.49% and 98.5%). In this experiment, we use a 4K transaction cache per core, and find that the CPU hardly stalls due to a full transaction cache. Only sps, the benchmark with the highest write intensity among the benchmarks, stalls for 0.67% of execution time. Kiln achieves 87.8% performance of the optimal case for both IPC and throughput. The reason is that when committing each transaction, the cache controllers of Kiln need to flush the writes of that transaction into the nonvolatile LLC; corresponding LLC blocks cannot be written back to NVRAM main memory before the cache flushes complete. Doing so blocks subsequent cache

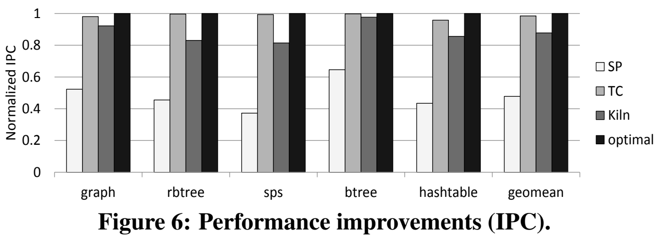


**Figure 6: Performance improvements (IPC).**

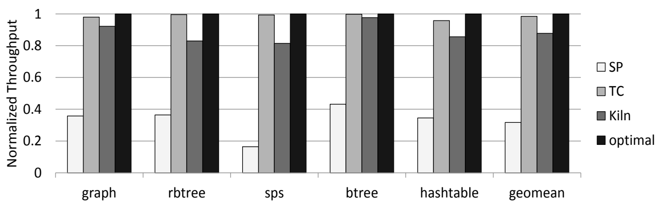


**Figure 7: Performance improvements (Throughput).**

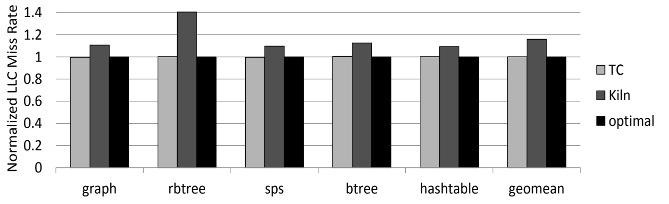


**Figure 8: LLC miss rate.**

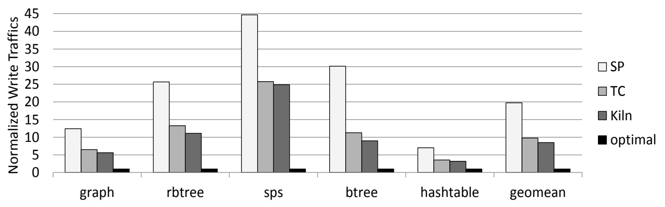


**Figure 9: Write traffic.**

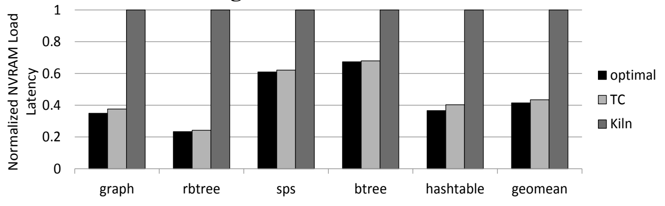


**Figure 10: Load latency.**

and memory requests during transaction commits and results in bursts of traffic in the cache hierarchy. Figure 8 shows the miss rate of different schemes normalized to the optimal case. On average, Kiln incurs 16% higher LLC miss rate compared to TC and the optimal case. The reason is that Kiln needs to keep all uncommitted cache blocks in the LLC, which can prevent other reusable data being stored in the LLC. Our design does not incur such constraints in the LLC and therefore leading to much lower LLC miss rate.

Figure 9 shows write traffic to the NVRAM memory of various schemes normalized to the optimal case. We can see that SP has close to 20 times more write traffics than the native execution (Optimal) due to logging overheads and cache flushes. Both TC and Kiln reduces the write traffics significantly, but still have more writes toward NVRAM than the optimal. The reason is that to ensure data persistence, TC and Kiln have to write back persistent data after a transaction commits or for a nonvolatile LLC replacement, but for the optimal case, these persistent data is just cached and coalesced in upper volatile cache layer. Even though TC has higher write traffics than the naive execution, it can still achieve comparable performance to the optimal case as shown above. This is because these writes are from the TC data path that are decoupled from the program execution. TC has more write traffic than Kiln because after a transaction commits, TC directly update the transaction data to NVRAM but Kiln only flush the data into the nonvolatile LLC.

Figure 10 shows the CPU persistent load latency of various schemes normalized toward Kiln. Because of cache flushes and the changes to the LLC replacement due to maintaining transaction ordering, Kiln has 2.41 times and 2.3 times load latency compared to the optimal case and TC, in average. Our mechanism TC achieves load latency close to the the optimal case.

## 6. CONCLUSION

In this work, we identify the key challenges to adopt the emerging persistent memory technique, which includes performance overhead of enabling persistence support in memory, implementation overhead of persistence support, and ease of use due to compatibility and flexibility issues. Based on the identification, we propose and implement an efficient hardware-based persistent memory accelerator design, which creates a side new persistent path and allows CPU caches and the NVRAM controller to operate as it is. The proposed design enables flexible modular processor implementation that is compatible with legacy designs, and releases software's burden on maintaining persistence. With comprehensively comparison of the performance of our proposed persistent memory accelerator to prior hardware-based persistent memory designs and the optimal case that does not provide persistence support, our results show that our design performs the best and achieves 98.5% the performance of the optimal case.

## 7. REFERENCES

[1] F. Bellard. QEMU, a fast and portable dynamic translator. ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[2] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. ASPLOS' 11, pages 105–118.

[3] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[4] K. Doshi, E. Giles, and P. Varman. Atomic persistence for SCM with a non-intrusive backend controller. HPCA '16, pages 77–89.

[5] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. EuroSys '14, pages 15:1–15:15, 2014.

[6] E. Giles, K. Doshi, and P. Varman. Bridging the programming gap between persistent and volatile memory using wrap. CF '13, pages 30:1–30:10, 2013.

[7] Intel. Revolutionizing the storage media pyramid with 3D XPoint technology. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html.

[8] Intel. Intel architecture instruction set extensions programming reference, 2015. https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf.

[9] A. K. Joseph Izraelevitz, Terence Kelly. Failure-atomic persistent memory updates via justdo logging. ASPLOS '16, pages 427–442, 2016.

[10] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Efficient persist barriers for multicores. MICRO-48, pages 660–671, 2015.

[11] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, and S. Pelley. Delegated persist ordering. MICRO-49, pages 1–13, 2016.

[12] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *ICCD*, 2014.

[13] A. Patel, F. Afram, S. Chen, and K. Ghose. Marss: A full system simulator for multicore x86 cpus. DAC '11, pages 1050 –1055, june 2011.

[14] J. Pedicone, T. Chiacchira, and A. Alvarez. Content addressable memory FIFO with and without purging, Apr. 18 2000. US Patent 6,052,757.

[15] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. MICRO-48, pages 1–13, 2015.

[16] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16–19, 2011.

[17] Z. Sun, X. Bi, H. H. Li, W.-F. Wong, Z.-L. Ong, X. Zhu, and W. Wu. Multi retention level STT-RAM cache designs with a dynamic refresh scheme. MICRO-44, pages 329–338, New York, NY, USA, 2011. ACM.

[18] M. Sunohara, T. Tokunaga, T. Kurihara, and M. Higashi. Silicon interposer with TSVs (through silicon vias) and fine multilayer wiring. In *Proc. of the Electronic Components and Technology Conference*, pages 847–852, 2008.

[19] J. Swanson and J. Wickeraad. Apparatus and method for tracking flushes of cache entries in a data processing system, July 8 2003. US Patent 6,591,332.

[20] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[21] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. ASPLOS '94, pages 86–97, 1994.

[22] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS*, pages 23–34. IEEE Computer Society, 2007.

[23] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. MICRO-46, pages 421–432, 2013.