國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

高效保證資料持續性之非揮發交易快取

Nonvolatile Transaction Cache for Efficient Persistence Guarantee

賴君濠

Chun-Hao Lai

指導教授：楊佳玲 博士

Advisor: Chia-Lin Yang, Ph.D.

中華民國 105 年 8 月

August, 2016

doi:10.6342/NTU201602526

致謝

在此感謝這些年研究生涯，一路遇到、陪伴與教導我的每一個人。

感謝指導教授－楊佳玲老師，一路以來耐心的教導，指導我研究的方法與理論，讓我了解邏輯的重要性，以及如何運用邏輯正確地分析研究過程中的任何問題，花了許多時間討論與學習，辛苦老師了。亦感謝趙戟珅老師對我研究上的教導，從百忙之中安排時間與我討論，讓我研究進度不會踟躕不前。感謝評審委員陳坤志、陳依蓉以及呂仁碩教授給予我論文的修改方向與評論，讓整個論文更完整。另外，也謝謝仁碩教授在實驗室時對我的指導與關心，也同時感謝柏翰學長回答我研究上的各種問題，不管是大學時還研究時期都幫助我許多。

感謝實驗室同學立維、王立及立展，互相照應，共同面對研究上或生活中的任何問題，雖然研究的主題與你們不相同，討論不多，但你們使得我研究的過程中增添許多愉快。也謝謝實驗室學弟益昌、學俊分擔了許多網管的負擔，還有尚軒和少甫學長網管方面的指導，以及感謝嘉麗與佩銀們的交接，還有建誼、依柔和王綸，讓實驗室增添了藝術氣息以及其他有趣的話題。

感謝古亭宿舍的高中同學，我們各自分工租借宿舍、打掃廁所、丟垃圾、吃飯和玩樂，一起渡過了許多日子，雖然大部分的時間都待在實驗室，但很晚回去的時候，你們都在，讓人倍感溫馨。也謝謝大學時的同學[1]，一起分享了不少美好時光。

感謝家人對我的照顧與陪伴，有著你們的支持和打理家中的一切，讓我每次回家時都十分安心，減少了許多負擔，能夠專注在研究上。

如此，十分感激。

[1]湧達、勁宇、聖文、弘宇、識淵、宗霖和柏軒

doi:10.6342/NTU201602526

摘要

持續記憶體架構是為一融合傳統主記憶體以及儲存單元的的新技術。藉由可接於記憶體匯流排的非揮發隨機存取記憶體 (NVRAM) 可位元組定址能力 (byte-addressability) 以及存儲單元所需要的非揮發特性 (non-volatility)，持續記憶體架構將是未來相當重要的新架構。有了持續記憶體架構的出現，將可直接將資料儲存於記憶體之中，而不用再複製到傳統硬碟或者快閃儲存系統。然而為了在記憶體階層保證資料持續性，持續記憶體必須維持資料從中央處理器快取寫到非揮發隨機存取記憶體的寫入順序。直接利用軟體指令來保證寫入順序將會造成許多中央處理器執行的限制而使得整體效能大幅下降。此外，為保證將資料寫入至記憶體的單元性，傳統軟體將會需要額外寫入一些輔助以及日誌資料。因此相關研究提出藉由硬體支援來解決上述效能下降的問題。然而，之前研究所提出的辦法並沒有完全解決此問題，而僅是將效能的負擔傳遞到快取及記憶體階層。因此，我們提出一個高效的硬體方法，提供一個新保證資料持續性的路徑，消除原本硬體架構要維持住寫入順序的負擔以及額外的日誌資料寫入，將整體效能提升至接近於原有沒有保證持續性軟體的效能。

關鍵字 — 持續記憶體, 非揮發記憶體, 資料一致性, 單元性, 耐用性, 持續性

doi:10.6342/NTU201602526

Abstract

Persistent memory is a new technique that merges main memory (DRAM) and storage (disk/flash) into one component. It can be implemented by memory-bus mounted nonvolatile memory (NVRAM), which incorporates the byte-addressability of memory and the non-volatility of storage devices. Persistent memory directly benefits computer system performance by allowing in-memory data to persist immediately without the need for accessing secondary storage, such as flash and disks. However, to ensure data persistence in memory, persistent memory systems need to preserve the ordering of writes from CPU caches toward NVRAM main memory. Directly utilizing the software instructions to ensure the write ordering will push much execution burden on CPUs and result to much performance degradation. Besides, to ensure the atomicity of storing operations to persist data, additional meta-data and logging operations are added into traditional programs without persistence guarantee. And thus, multiple hardware supported solution is proposed. However, previous hardware solutions do not solve the problem at all, which propagates the performance overhead toward cache or memory hierarchy. Therefore, we propose an efficient hardware solution and provide a new persistent path, which frees existing architecture from handling the write ordering, eliminates logging overhead and increases the overall performance near the program without persistence guarantee.

Keywords — Persistent memory, nonvolatile memory, data consistency, atomicity, durability, persistence

doi:10.6342/NTU201602526

Contents

doi:10.6342/NTU201602526

doi:10.6342/NTU201602526

List of Figures

doi:10.6342/NTU201602526

doi:10.6342/NTU201602526

List of Tables

doi:10.6342/NTU201602526

Chapter 1

Introduction

Emerging nonvolatile memory (NVRAM) technologies, such as phase-change memory (PCM) [15, 23], spin-transfer torque magnetic random access memory (STT-RAM) [9, 14] and resistive random access memory (RRAM) [27], enable a new memory hierarchy design. NVRAMs have hybrid properties of memory hierarchy (DRAM) and persistent storage (disk/flash). Like DRAM, NVRAM is byte-addressable and thus can be connected through the memory-bus and put at memory layer. Simultaneously, like disk/flash, NVRAM is nonvolatile, where different from DRAM, data saved in NVRAM will not disappear after a shutdown. With these hybrid properties, persistent memory is proposed. Persistent memory utilizes NVRAMs as memory layer storage. With better access speed and closer layer toward CPU compared to disk/flash, persistent memory directly brings great potential for storage performance. Not complicated block-based protocol toward disk/flash [3], programmers now can save data into memory via load/store interface (byte granularity). However, to utilize NVRAMs as memory layer storage and achieve better performance, carefully modifications of software or hardware are unavoidable, which incurs new design challenges.

Persistent memory changes the access behaviors of programs. Traditionally, programs only utilize memory (DRAM) to save temporary data, which will disappear after a power disconnection, but now, programs will save important data into memory (NVRAM), where saved data is persistent for future use. Therefore, under persistent memory system, programmers will care about how to persist data into NVRAM and if data is saved as their

doi:10.6342/NTU201602526

expectation. Programmers expect that the saved data of a program is consistent after a normal program exit or even a sudden power failure. Consistent data means that data is correctly saved in NVRAM as programmers want. Programmers will not expect that operated data in NVRAM is crashed (inconsistent), which wastes storage space and cannot be further operated because crashed data may not have the same format as programmers think. Formally, the support of persistent memory to ensure that data saved in NVRAM to be consistent as program behavior is called - data persistence guarantee.

To support data persistence, data writes toward NVRAM need to occur in the same order as program behaviors. For example, to append a new node to a link list, program will allocate a new node first and then make the last pointer of the link list point toward the newly allocated node. If the write ordering is reversed. Then, after a power failure during these two operations, it will result to an inconsistent condition that the last pointer points toward a position that is not allocated. Besides, even the write ordering is guaranteed, operations of a program may also be half completed and cause unexpected results. For example, previous example may result to memory leakage, where new node is allocated but not pointed by any pointer [2]. And thus, to guarantee data persistence, we have to ensure the write ordering and atomicity of operation.

However, traditionally, the ordering of data requests issued from CPU through cache hierarchy toward main memory layer will be not the same as program behaviors. For cache line eviction, cache controller will have its own replacement policy, such as least-recently used (LRU). For memory requests processing, main memory controller also will do different scheduling policy to provide better performance, such as first ready, first-come-first-service (FR-FCFS) [25]. To avoid this problem, previous works [32, 5] use software instructions, such as memory fence (sfence and mfence) and cache flushing (clflushopt, clwb and pcommit) instructions, to flush data from cache hierarchy toward NVRAM. Only after flushing data into NVRAM, memory write operations after these instructions can be executed and thus, it ensures the write ordering. Although these instructions enforce the write ordering, this method blocks CPU from executing later memory requests, couples ordering with durability and results to performance degradation. To solve this problem, Condit

doi:10.6342/NTU201602526

et al. [3, 12, 28] propose epoch barrier to let software communicate ordering requirement to cache and main memory. Cache replacement policy and main memory scheduling way will follow the software declared ordering requirement without directly flushing data from cache toward main memory and blocking other memory requests to be issued. Doshi et al. [4] propose a software/hardware library and modify memory controller to support updating data in parallel with program execution on CPU at background, which decouples persisting operations from CPU execution and delegates them to memory controller.

As described above, in addition to the write ordering, the atomicity of data write operations is the another one of the important factors to ensure data persistence. To guarantee atomicity, programmers will add additional meta-data or logging operations to track the write progresses and thus be able to recover half completed data after a sudden power failure. Most of previous works applies write-ahead logging [18, 32, 5, 28, 2] or copy-on-write [8, 3] to do redundant backup for the atomicity of write operations. After sudden system crash, the redundant backup can be utilized to recover the crashed data. However, to support write-ahead logging or copy-on-write, additional NVRAM space is occupied and write traffic is also increased because of redundant writes. Seeing this problem, Zhao et al. [36] propose to utilize a nonvolatile LLC to naturally backup data during data flow from cache toward NVRAM and eliminate the logging operations or redundant data writes.

However, even though previous works put much effort into ensuring data persistence, problem is still not solved. Previous works directly change the components in the existing architecture (cache hierarchy or memory layer) and extend the cache/memory controller to ensure the write ordering. In this way, even though Condit et al. [3] or Doshi et al. [4] relax CPU from blocking later memory requests, the cache hierarchy or memory layer are still constrained by the ordering overhead. The same, to ensure the write ordering, Zhao et al. [36] still need to flush data from L1 toward LLC, and the replacement policy of the LLC will be modified and constrained to meet the ordering requirement. Directly combining the new policy into existing cache replacement is not naive, which makes cache design logic complex [3, 36]. And the modification of memory controller from [4] is complicated, which needs to read out logging data and then updates target data. Also, previous works

doi:10.6342/NTU201602526

only solve one of the problem mentioned above, neither of them solve both the write ordering and additional logging overhead at once.

In this paper, we propose the transaction cache. By adding an additional nonvolatile hardware of small size aside of the existing cache hierarchy, we free the existing architecture from explicitly handling the writing ordering and eliminate the logging operations. By this way, besides of freeing CPU from blocking later memory requests, the cache controller and memory layer also will not be constrained by persistence overhead and differently, a new persistent path via the transaction cache is provided. By removing persistence overhead, we can achieve the performance near the native case without persistence support.

doi:10.6342/NTU201602526

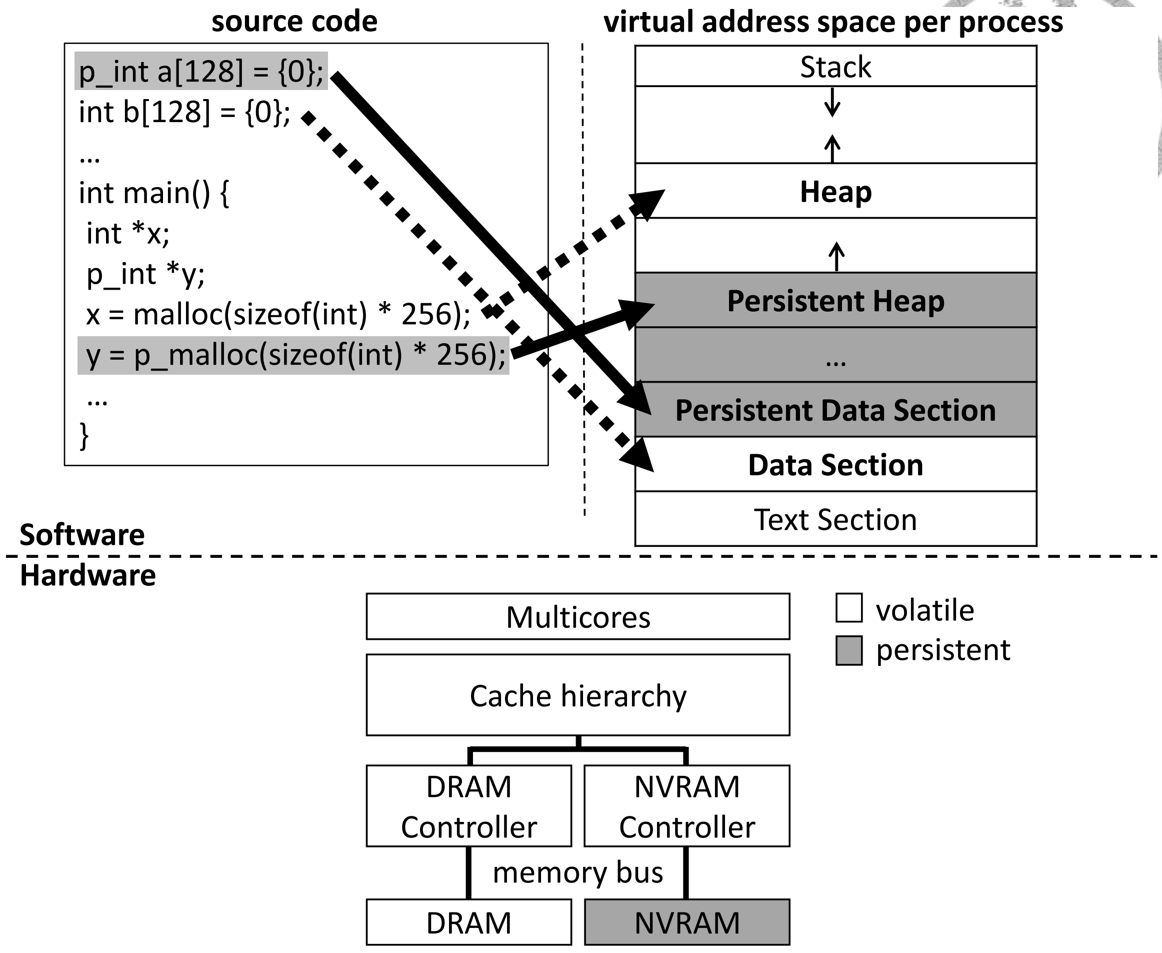
Chapter 2

Background and Motivation

To utilize NVRAM as persistent storage, without compromising the benefits of fast access speed of nonvolatile memory, persistent memory will directly connect NVRAM through memory-bus but slower SATA or PCI interfaces for disk or flash. Besides, even though NVRAM has performance near DRAM, it is not totally the same as DRAM as specially for the write latency. Thus, to obtain best performance, it is better to use a combination of DRAM and NVRAM for a system running programs with or without data persistence demand concurrently. Therefore, as the bottom hardware part of Figure 2.1 shows, we target the system that NVRAM will be put aside with DRAM and both are connected through memory-bus toward memory controller and CPU cache hierarchy. From the top software part of Figure 2.1, programs can directly access NVRAM the same as DRAM through load/store instructions or persistent memory library like Mnemosyne [32] or NV-Heaps [2] and declare, allocate or save data into NVRAM or DRAM. Programs that need data persistence will save and write data into NVRAM and utilize software or hardware mechanism to ensure that saved data is consistent. In the following, we focus on the properties to ensure data persistence.

To ensure data persistence, transaction concepts are borrowed from database. A transaction among a program is a group of memory read/write operations that modifies the content of a persistent storage (NVRAM). Transaction concepts contain atomicity, consistency, isolation and durability (ACID) [36]. Modification done by a transaction shall be atomic and must not partially complete, which is **atomicity**. Without atomicity guarantee,

doi:10.6342/NTU201602526



```
p_int a[128] = {0};
int b[128] = {0};
…
int main() {
 int *x;
 p_int *y;
 x = malloc(sizeof(int) * 256);
 y = p_malloc(sizeof(int) * 256);
 …
}
```

Figure 2.1: Persistent memory architecture

data may be half modified and inconsistent. **Consistency** means that the total operations of a transaction defined by programmers have to bring the data in the persistent storage from one consistent state to the another one. **Durability** guarantees that data modified by a transaction after it commits should be preserved in the persistent storage even after a power failure or sudden crash. And only after a transaction commits, data modified by a transaction can be viewed by other transactions, which ensures **isolation**. After a transaction **commits**, it means that the transaction can finish atomically and data modified by the transaction would be durable and readable. Among these four properties, to make NVRAM serve as a persistent storage, we have to ensure that data in the NVRAM modified by transactions of a program have to be consistent and persistent or it wastes storage, which covers ACD and guarantees data persistence.

doi:10.6342/NTU201602526

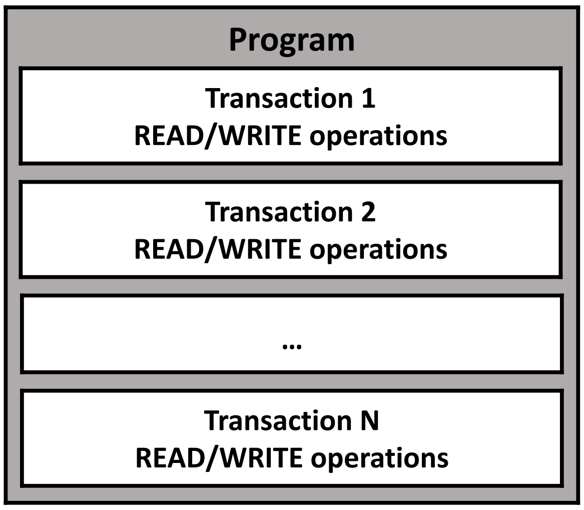


Figure 2.2: Program with transaction concept

2.1 Match Program Behavior

Based on transaction concept, a program is composed of multiple sequential transactions from 1 toward N as Figure 2.2 shows. To guarantee data persistence, we have to ensure data in the persistent storage is consistent with program behaviors. If data of transaction N is preserved in the persistent storage, data of transaction 1 till N-1 have to be also preserved in the persistent storage. This is an obvious and important rule. Data saved in the persistent storage have to match program behavior and be identical to the program results. If data of transaction 4 is durable in the persistent storage but data of transaction 3 is not durable in the persistent storage, it is an inconsistent result and some execution results of the program are missed. To match program behavior, the ordering of writes toward the persistent storage have to be the same as the execution order of different transactions order. Data of different transactions have to be persistent and durable in order.

2.2 Transaction Atomicity Guarantee

Besides of different transactions execution order, the atomicity of each transaction is another important property. To guarantee this, most previous works [32, 5, 3, 28] employ write-ahead logging [18] to make the data modified by a transaction stay consistent even after a power failure or an abrupt system crash. Figure 2.3 illustrates the transaction with

doi:10.6342/NTU201602526

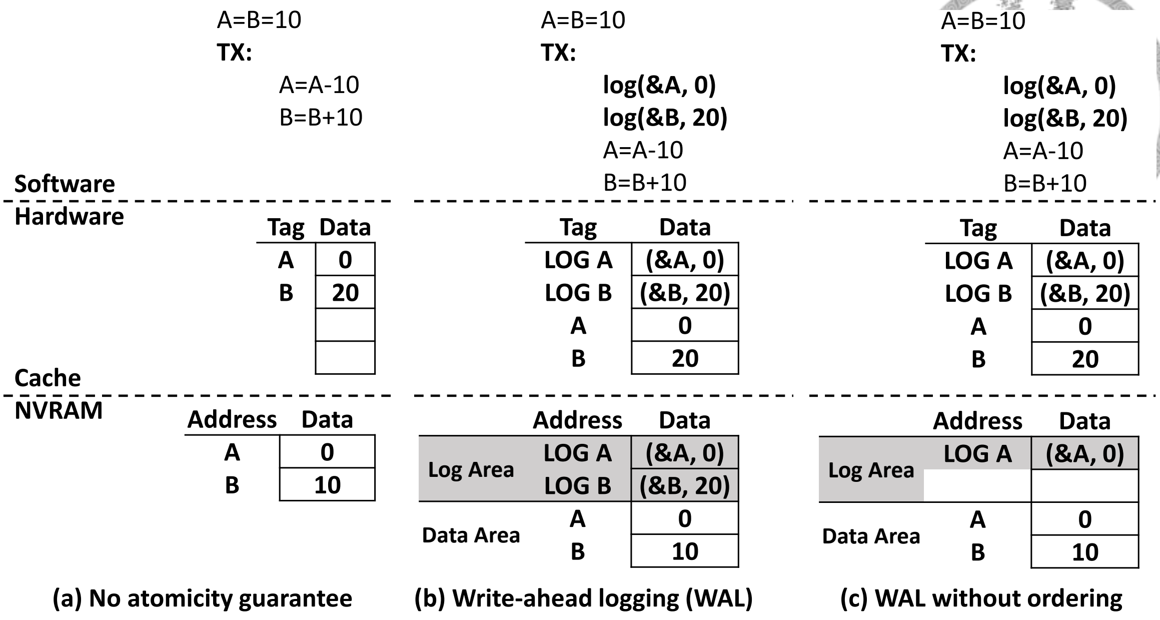


Figure 2.3: Transaction with and without atomicity guarantee by write-ahead logging

and without atomicity guarantee by write-ahead logging. The example in Figure 2.3 is to transfer 10 from variable A to variable B. Initially, both variable A and B are 10. Without losing consistency, after this transaction, the sum of variable A and B should still be 20 (10+10). But, in Figure 2.3 (a), if a system crash happens after only the variable A is written back into the NVRAM, the updated value of variable B that is still in the cache hierarchy would be lost. And the final results in the NVRAM is inconsistent. The sum of variable A and B would be 10, but not 20. Besides, from the results, the transaction only completes its operation on data A and its operation on data B is lost, which is half completed and violates atomicity. And thus, to ensure atomicity, before updating variable A and B, a transaction in Figure 2.3 (b) will do logging first. The logging data contains the target data address and to be updated data (ex, address of variable A and 0). Only after logging operations are finished, the updating operations of target data will be processed. With these guarantee, even if a system crash happens after only the variable A is written back into the NVRAM, we will have the logging data for variable B and can do recovery to update the variable B from old value 10 to target value 20, which ensures the atomicity of the transaction.

doi:10.6342/NTU201602526

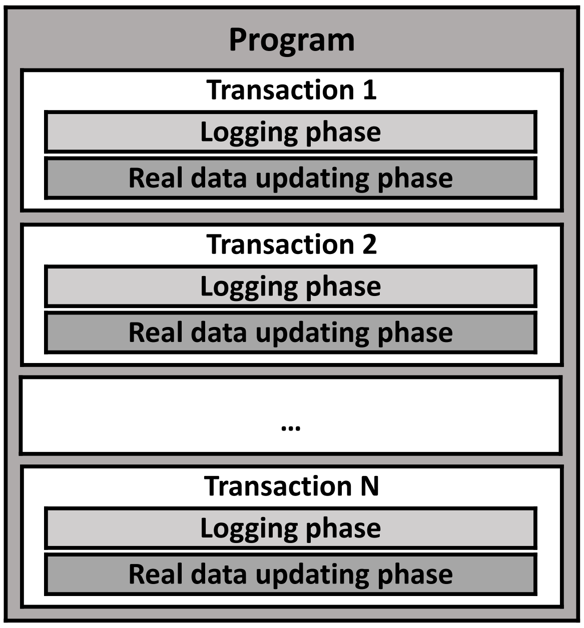


Figure 2.4: Program with write-ahead logging transaction concept

2.3 Program with WAL concept (Persistent Program)

Considering program behavior and transaction atomicity, a persistent program is composed of multiple transactions and each transaction will do its WAL for atomicity guarantee as Figure 2.4 shows. As above clarified, we can know that the logging operations and target data updating writes have to be processed in order. Also, to match program behavior, data of different transactions also have to be persistent in order. However, the write ordering from cache hierarchy toward the NVRAM is controlled by hardware and could be reordered. Cache eviction and memory scheduling policy will make the write ordering from cache hierarchy toward the NVRAM different from program behaviors, such as least-recent policy (LRU) and first ready, first come first service (FR-FCFS) [25]. Under this condition, even though a transaction applies write-ahead logging, the final data saved in the NVRAM would not be consistent and could not be recovered to a consistent state. For example, in Figure 2.3 (c), there will be a result that only the logging data for the variable A and the target variable A updating are completed and written back into the NVRAM after a sudden power failure because of hardware reordering mechanisms. For this condition, the same, after writing back the variable A into the NVRAM, we don't have the logging data for variable B to recover data, which results to inconsistent data.

doi:10.6342/NTU201602526

On the other side, there may be a result that data of the transaction 3 and 1 is saved in the NVRAM but data of the transaction 2 is lost because of reordering, which doesn't match the program behavior. That is, the ability to guarantee the write ordering is an important factor for data persistence.

2.4 Software-supported Persistent Program

To provide software the ability to enforce the write ordering, Intel announces new memory instructions, such as **clflushopt, clwb and pcommit** [11]. As Figure 2.5 shows, software programmers can utilize these new instructions with memory fence instructions, such as **sfence and mfence**, to ensure the write ordering toward the NVRAM the same as program behaviors. Each instruction covers different guarantees:

- **clflushopt** and **clwb** (cache flushing instructions)

  These instructions flush cache lines from cache hierarchy toward main memory layer. The difference is that **clflushopt** will invalid the flushed cache line entry and **clwb** will not, which provides programmers more options for better performance under different data locality behavior of programs. To flush cache lines that will not be used in the future, we can utilize clflushopt and conversely, we can use clwb.

- **sfence** and **mfence**

  These instructions ensure that the data flushed by **clflushopt and clwb** will be flushed into the NVRAM write queue and accepted by the NVRAM before executing **pcommit**.

- **pcommit**

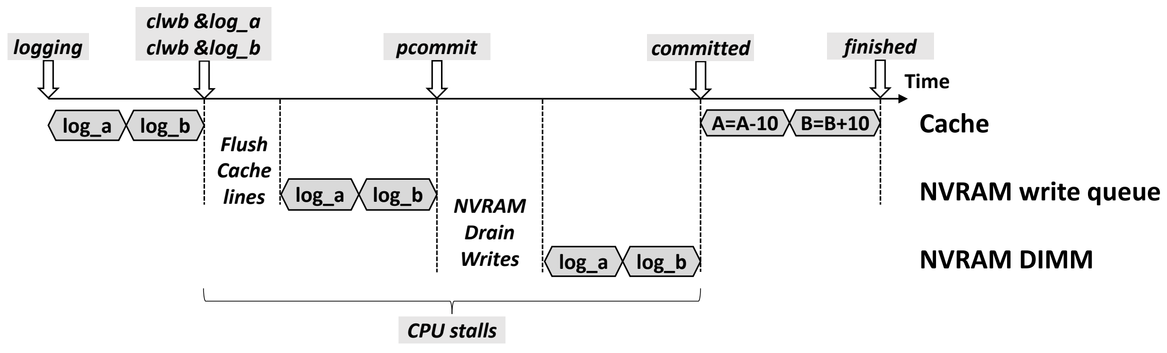
  This instruction drains the write request in the NVRAM write queue accepted by the NVRAM into the NVRAM DIMM and makes them persistent.

With these instructions, software can persist data at certain points of a program. Normally, data writes will be persistent after the positions of pcommit instructions. From Figure 2.5 (a), the same as Figure 2.3, a simple transaction to transfer 10 from variable A

doi:10.6342/NTU201602526

```
A=B=10
TX:
        log_a=log(&A, 0)
        log_b=log(&B, 20)
        clwb &log_a
        clwb &log_b
        sfence
        pcommit
        sfence
        A=A-10
        B=B+10
```

(a) Program with Intel instructions



(b) Execution timeline

Figure 2.5: Software with Intel instructions and execution timeline

to variable B, but the write ordering of logging operations and target data updating is guaranteed. By utilizing the **clwb**, we can flush the logging data of log_a and log_b from cache hierachy toward the NVRAM write queue. And after the **clwb**, the **sfence** and **pcommit** are applied. **Sfence** makes sure that **pcommit** happens right after **clwb** instructions. And thus, we can exactly use **pcommit** to drain the data, log_a and log_b, in the NVRAM write queue into NVRAM DIMM. After the draining, CPU continues to do data updating of variable A and B.

2.4.1 Inefficiency in Software-supported Persistent Program

As we can see, even though utilizing these software instructions can ensure the write ordering, the performance is sacrificed. Software-supported persistent program has two deficiency. First, from Figure 2.5, during the flushing and draining operations, CPU is stalled and does nothing but waits for logging data being drained into the NVRAM DIMM. That's, the software-instruction-supported write ordering is not efficient. By utilizing software instructions, CPU has to explicitly control and handle the write ordering

doi:10.6342/NTU201602526

from the cache hierarchy toward the NVRAM, which wastes a lot of time for CPU to do other computations. Second, to support write-ahead logging and ensure atomicity, a single write will need an another logging operation. This results to double write latency for a single write and for write intensive workloads, most of the CPU execution time are doing redundant backups without real progress for target data.

2.5 Related Works to Mitigate Persistence Overhead

With above these two problems, previous works follow these two directions to solve them. To mitigate the ordering problem, Condit et al. [3] propose epoch barrier, an instruction to deliver the software ordering requirements toward cache and memory hierarchy. With this instruction, instead of blocking following memory requests, CPU will continue to execute; cache controller and memory controller will follow the software ordering requirements asynchronously. To evict a target cache line, cache controller will check if there are cache lines that have ordering dependency before the target cache line and evict them before the target cache line first. The same, memory controller will write data into memory DIMM in the same order. Differently, Doshi et al. [4] propose to make NVRAM controller responsible for ensuring the ordering and let CPU execute in parallel with background persisting operations processed by NVRAM controller. Logging data are combined and committed into NVRAM via a new software/hardware interface, and NVRAM controller will update the target data from the logging data at background. On the other side, to eliminate logging writes, Zhao et al. [36] propose to apply nonvolatile last level cache to serve as another backup like logging data. The difference between [36] and WAL is that utilizing nonvolatile LLC and following architectural data flow, data writes from cache toward main memory will naturally back up data in the LLC, which doesn't need additional logging writes.

However, even though previous works try to solve the ordering constraint and eliminate the logging operations, the problem is just relieved and not solved. The persistence overhead is propagated toward cache hierarchy or memory layer. As above described, Condit et al. [3] need to modify the cache controller and memory controller to follow the

doi:10.6342/NTU201602526

ordering requirement and actually, the cache replacement policy and memory scheduling way will be constrained. Doshi et al. [4] have to do much modification on the memory controller to support persisting operations and the memory controller is still constrained by the logging and updating ordering. Zhao et al. [36] have to flush cache lines of a transactions toward LLC to follow the transaction ordering requirement. Besides, the replacement policy of LLC is also constrained and the in-progressing data of a transaction are not replaceable, which results to fewer cache space for other data. In addition, none of previous works solve these two problems at once.

Seeing these problems, in this work, we propose the transaction cache, which removes overall persistence overhead from the existing architecture and make them execute natively without any constraint. In the following, we describe our mechanism to solve persistence overhead in chapter 3. And then, we show the experimental results in chapter 4. And for future study, we discuss the prior works related to persistent memory in chapter 5. Finally, we conclude this work in chapter 6.

doi:10.6342/NTU201602526

Chapter 3

Transaction Cache Design

3.1 Overview

The goal of our design is two-fold. First, allow a program to progress as if there is no persistent requirement. Second, the design needs to be non-intrusive to the existing cache or memory hierarchy. Figure 3.1 shows the high-level view of the proposed transaction cache (TC) design. Flushing and memory fence instructions are eliminated and program executes as usual. Hardware will guarantee the persistence that program needs. A new persistent path via the transaction cache is added to handle persistent writes asynchronously with program execution. This new path is put side by side with the existing cache hierarchy, guaranteeing the persistence without bothering the L1, L2 or LLC cache controllers. The transaction cache is a nonvolatile memory buffer, which inserts and evicts data as a first-in/first-out memory (FIFO). A persistent write is written to both the transaction cache and the L1 cache. TC buffers and persists the persistent writes at background in parallel with program execution. Also, as a nonvolatile buffer, TC serve as a logging area. After writing all the data of a transaction, TC will write back the data of the transaction to the NVRAM. And only after writing back data into the NVRAM, TC can remove the data of the transaction.

Since TC serves as the logging area, we eliminate writing logs at the NVRAM layer. All the persistent writes toward the NVRAM have been logged in TC. Without writing logs at the NVRAM layer, the ordering requirement among a transaction to ensure atomicity

doi:10.6342/NTU201602526

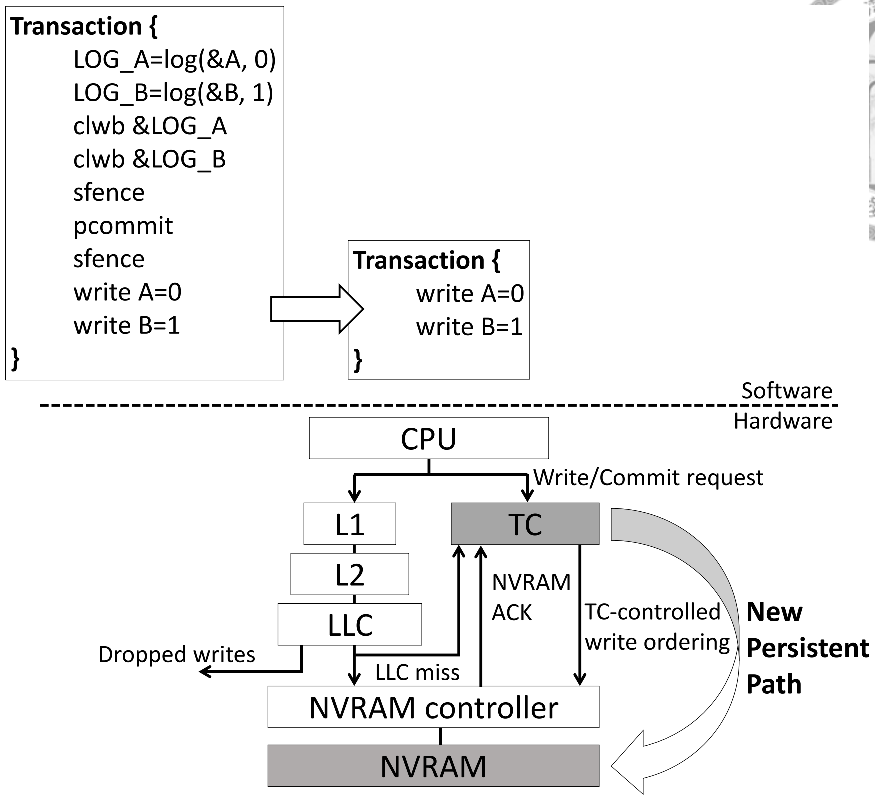


Figure 3.1: Transaction Cache Overview

is removed. The ordering requirement now is only the sequential ordering of different transactions. But, with backup data in TC, even the NVRAM controller schedules the persistent writes out of the sequential ordering of different transactions, such as writes data of later transactions first, after a sudden crash, we can still utilize the backup data from TC to do recovery. That is, the NVRAM controller can schedule the persistent writes without ordering constraint.

Besides, when a cache line containing persistent data is replaced from the last level cache, it is dropped and not written back to the NVRAM so the state of persistent data is not affected. All the persistent data toward the NVRAM will flow via the transaction cache. In this way, a program can execute speculatively on the original data path without extra writes for logging and stalls for ensuring correct write order. Besides, existing cache hierarchy or memory layer is not constrained by the persistence overhead.

In the proposed design, without compromising performance, two important properties (write ordering and transaction atomicity) to ensure data persistence are guaranteed.

- **Write Ordering:** Different from the existing cache hierarchy, all the persistent

doi:10.6342/NTU201602526

writes flows the new persistent path will not be reordered. The transaction cache serves writes as FIFO. The persistent writes from CPU are inserted into and evicted from TC in FIFO order, which simply conforms to the write ordering. The ordering of different transactions is guaranteed.

- **Transaction Atomicity:** All the persistent writes of a transaction toward the NVRAM will have completed backup in the transaction cache. Only after totally backing up a transaction, the transaction cache will issue the data of the transaction toward the NVRAM. And only after updating the NVRAM, the corresponding backup in the TC can be dropped. If a crash happens during the time for the NVRAM controller to process the data of a transaction, we can recover data in the NVRAM from a half completed one to a consistent one by the backup in the TC. If a crash happens before the transaction cache issues the data of the transaction to the NVRAM, data in the NVRAM is intact and consistent. With the completed backup in the TC, transaction atomicity is guaranteed. To accomplish above behavior, after finishing a write request, the NVRAM controller have to send back a acknowledgment message back toward the TC to let the TC to drop written back backup data.

Since the persistent data to be replaced from the LLC is not written back to the NVRAM, these new version persistent data is discarded and the persistent memory may only contain the stale value. This scenario appears when the persistent data replacement happens before TC evicts the corresponding data toward the NVRAM. 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.

doi:10.6342/NTU201602526

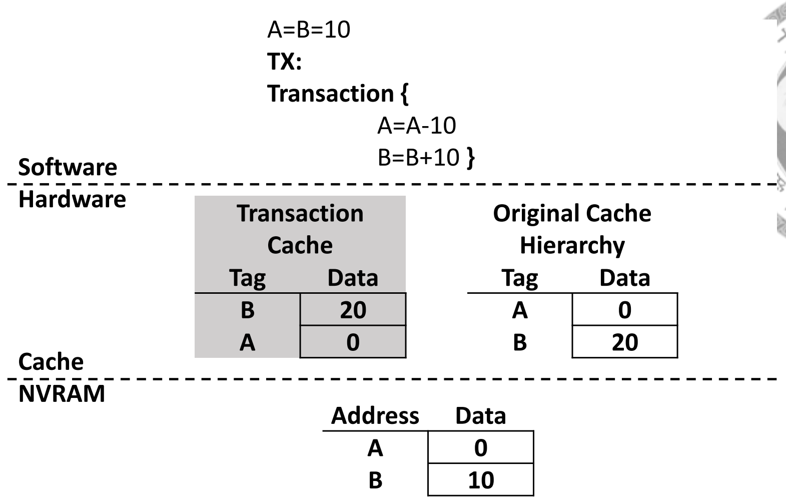


Figure 3.2: Transaction cache example

3.2 Transaction Cache Example

Figure 3.2 shows the proposed transaction cache execution results with the same transaction example as Figure 2.3. To execute a transaction, software now only needs to do target data updating operations and provide the transaction boundary information. The flushing and logging operations for ordering and transaction atomicity guarantee are removed. Hardware will guarantee the persistence with the transaction information provided from software. We can see the program in Figure 3.2 is almost the same as the program without persistence guarantee in Figure 2.3 (a).The only differences are the transaction mark and big parentheses. With the proposed transaction cache mechanism, data flows via the transaction cache toward NVRAM in FIFO order as program behaviors. Transaction cache will issue the write back request of the variable A first and then the variable B. But, the memory controller can schedule these two requests concurrently without ordering constraint because the transaction cache has backed up the data. Figure 3.2 shows the condition after NVRAM updates the variable A. The variable A is updated from old value 10 to new value 0 and the variable B is still the old value 10. If there is a power failure happens after NVRAM updates the variable A, we can recover the variable B in NVRAM from old value 10 to the new value 20 by the backup in the transaction cache.

Figure 3.3 shows the execution time line of the proposed transaction cache. With the transaction cache, after writing data A and B and finishing a transaction, CPU now can

doi:10.6342/NTU201602526

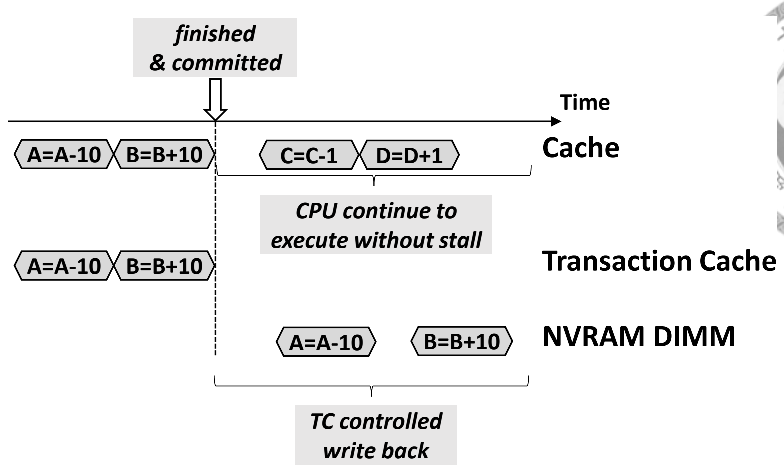


Figure 3.3: Transaction cache execution time line

continue to execute to do computation on other data C and D without stalling to control the write ordering. And the existing cache hierarchy will serve these requests constantly without flushing operations to ensure the write ordering. Besides, we can notice that because logging operations are eliminated, after inserting data into the transaction cache, one transaction can directly commit and finish. After that, the write ordering of data A and B would be controlled by the transaction cache.

In the following sections, we describes detailed hardware implementation of the transaction cache and modification on the existing architecture design.

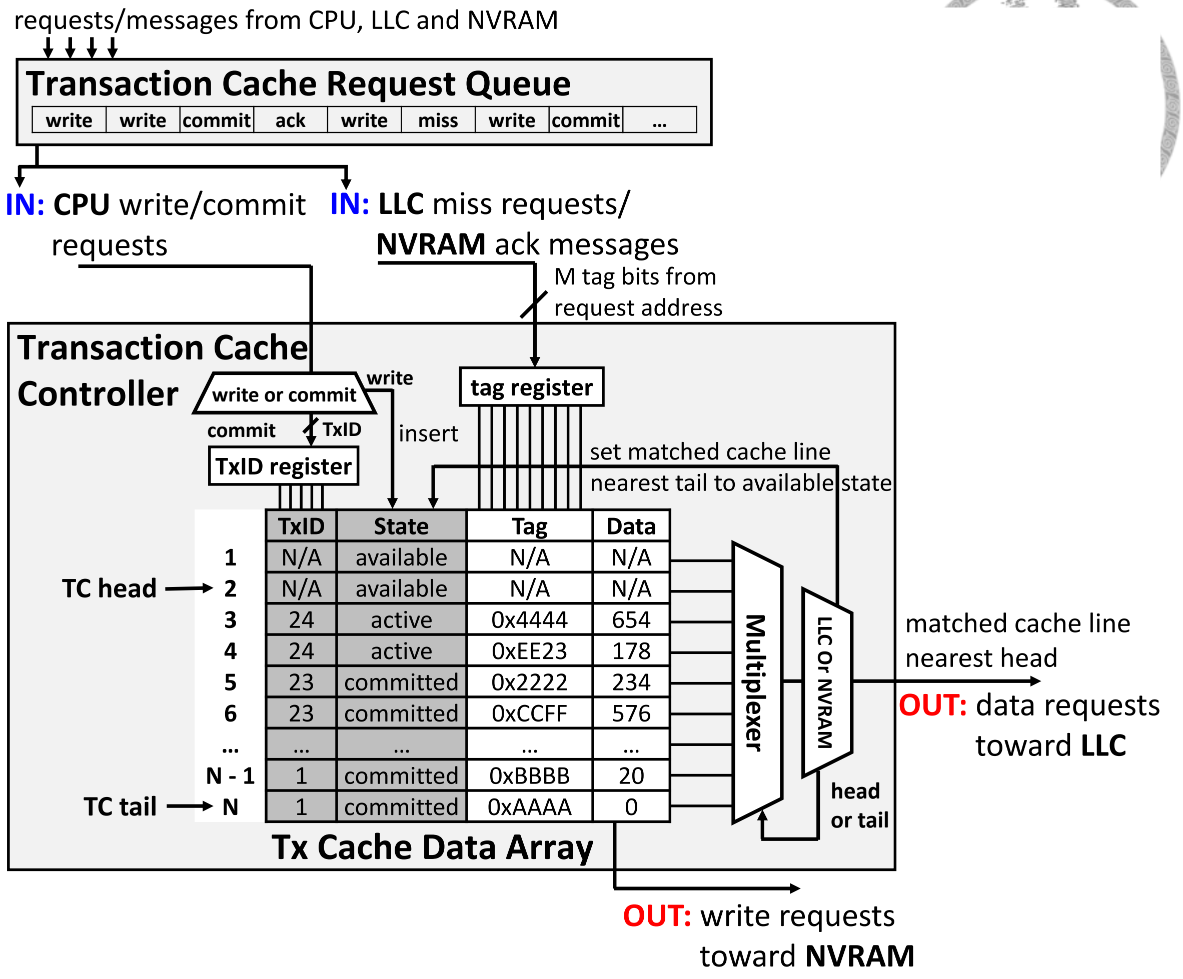
3.3 Detailed Software and Hardware Modification

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.

3.3.1 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) [21, 30]. 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

doi:10.6342/NTU201602526



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
commit
TxID
TxID register
insert
tag register
set matched cache line
nearest tail to available state
TC head
TC tail
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
Tx Cache Data Array
Multiplexer
LLC Or NVRAM
head or tail
matched cache line nearest head
OUT: data requests toward LLC
OUT: write requests toward NVRAM

Figure 3.4: Transaction Cache architecture

operation. Figure 3.4 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 3.4, besides of the tag and data value, each cache line among the TC data

doi:10.6342/NTU201602526

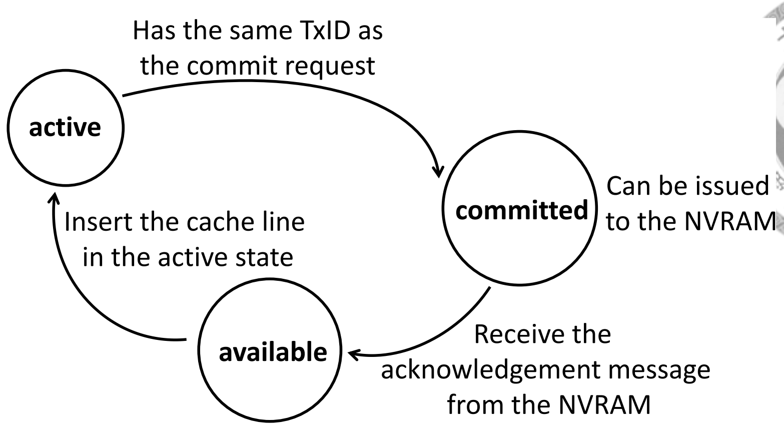

Has the same TxID as the commit request
active
committed
Can be issued to the NVRAM
Insert the cache line in the active state
available
Receive the acknowledgement message from the NVRAM

Figure 3.5: Cache line transition state of Transaction Cache

array also records the transaction information to ensure transaction atomicity, the transaction ID (TxID) and the transaction state of corresponding transaction ID (State). Each cache line entry among the TC data array transitions in three state: **available**, **active** and **committed** (Figure 3.5). 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. To prevent the problem that NVRAM serves writes too slow and lets the transaction cache easily gets full, if the transaction cache is almost full, the transaction cache will send a signal toward NVRAM to tell NVRAM controller to drain the writes.

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). Because requests issued from CPU are handled in FIFO and thus after serving a commit requests, it means all the data of a transaction are backed up in the TC. In this way, writes requests issued to the NVRAM will have backup in the TC data array and thus can be scheduled out of order. Besides, because conflicted requests (same

doi:10.6342/NTU201602526

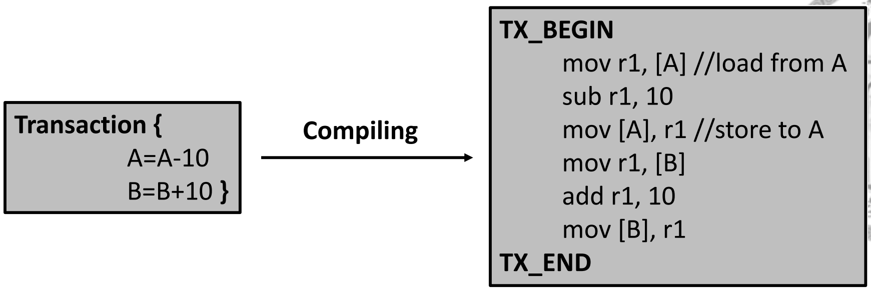


Figure 3.6: Software modification

cache line address and toward the same row of a bank) have to be handled sequentially in the NVRAM, the NVRAM controller will schedule the conflicted write requests of different transactions in the same as the FIFO issue order and is consistent with program.

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 entries 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.

doi:10.6342/NTU201602526

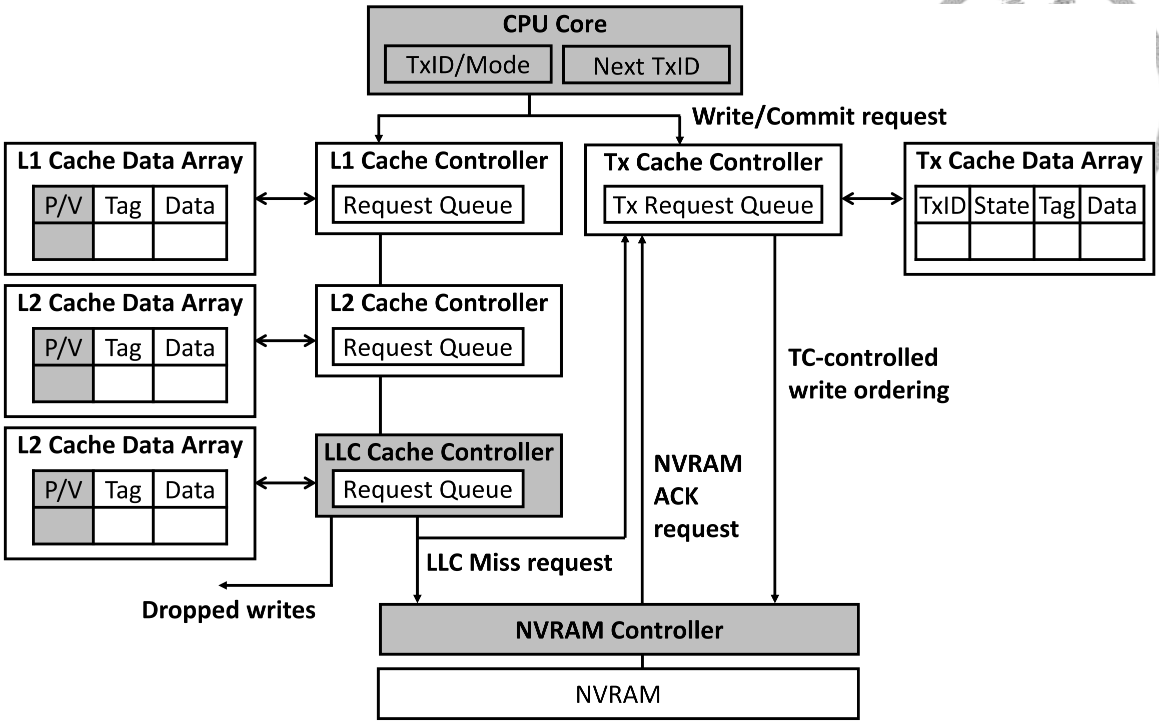


Figure 3.7: Hardware architecture modification

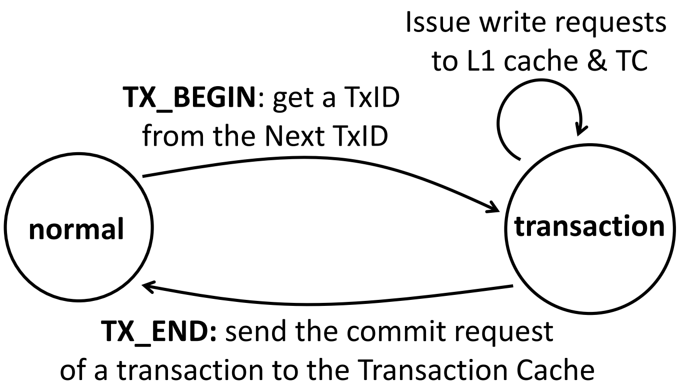

Issue write requests to L1 cache & TC
TX_BEGIN: get a TxID from the Next TxID
normal
transaction
TX_END: send the commit request of a transaction to the Transaction Cache

Figure 3.8: CPU transition state

3.3.2 Software Interface

Software now only needs to provide the transaction boundary information. As shown in Figure 3.6, transactions can be implemented as following:

Transaction {...}

. This function will be compiled into CPU primitives TX_BEGIN and TX_END. Encountering these primitives, CPU can execute in both normal mode (without persistence guarantee) and transaction mode (with persistence guarantee) (Figure 3.8). If CPU is executing the codes enclosed by TX_BEGIN and TX_END, then it is in the transaction mode.

doi:10.6342/NTU201602526

Otherwise, it is in the normal mode. As illustrated in Figure 3.7, CPU maintains a mode register that indicates whether it is in the normal or transaction modes. 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.

3.3.3 Other Modifications in the Processor

In Figure 3.7, 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 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.

doi:10.6342/NTU201602526

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 size
Other TC components (Multiplexer, tag register)	flip-flops	depends on TC data array size
TC data array	STTRAM	flexible

Table 3.1: Summary of major hardware overhead.

3.3.4 Accommodating Multicore Systems

The write requests issued to both the L1 cache and the transaction cache are not synchronous. The cache line of a write request is directly inserted into the TC, but is inserted into the L1 cache and visible by other cores after cache coherence protocol. Besides, the TC is readable. Therefore, the TC may make some cache lines visible to other cores earlier than cache coherence protocol allows so. Before the L1 cache controller completes the write coherence protocol operation of an to-be-inserted cache line, the to-be-inserted cache line may be inserted into the TC and read by the other read requests from the other cores. To address this multicore issue, we force the TC controller to wait for the L1 cache controller to complete the corresponding coherence operation of corresponding writes and obtaining the ownership of the write request. To this end, we implement a TC request queue. Persistent write requests issued from CPU are first buffered in the TC request queue. After the L1 cache controller finishes the cache coherence operation of corresponding writes, the L1 cache controller will send a finishing signal to the TC controller. Until then, the TC can start serving the corresponding persistent write requests at tail of the TC request queue[1]. The finishing signal can be implemented the same way as acknowledgment signals among traditional cache controllers.

[1]Under x86 architecture, the L1 cache controller will follow the total store order memory consistency model and handle write requests in program order. Thus, upon receiving a finishing signal, it must indicate that the write request at tail of the TC request queue finishes cache coherence protocol.

doi:10.6342/NTU201602526

3.3.5 Hardware Overhead

For the storage overhead showed in Table 3.1, a cache line in the TC data array has to record transaction ID (TxID), state and cache line data (tag and data). For a core with a 4KB transaction cache size, if each transaction only touches one cache line, 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 finishing 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.

3.3.6 Detailed Transaction Cache Execution Example

Figure 3.9 shows the detailed execution of the same transaction example as before. We divide the execution sequence with time T1, T2 and T3. At executing "TX_BEGIN" before time T1, CPU 0 will get the transaction ID from the next transaction ID initialized with 1. We can see that after executing "TX_BEGIN", CPU 0 has mode ID register with value 1, which is in the transaction mode. And in the transaction mode, all the data writes will be issued toward both the existing cache hierarchy and the transaction cache. Requests issued toward the existing cache hierarchy will set up "P" flag to know that it is persistent request. Requests issued toward transaction cache will have transaction ID information. In

doi:10.6342/NTU201602526

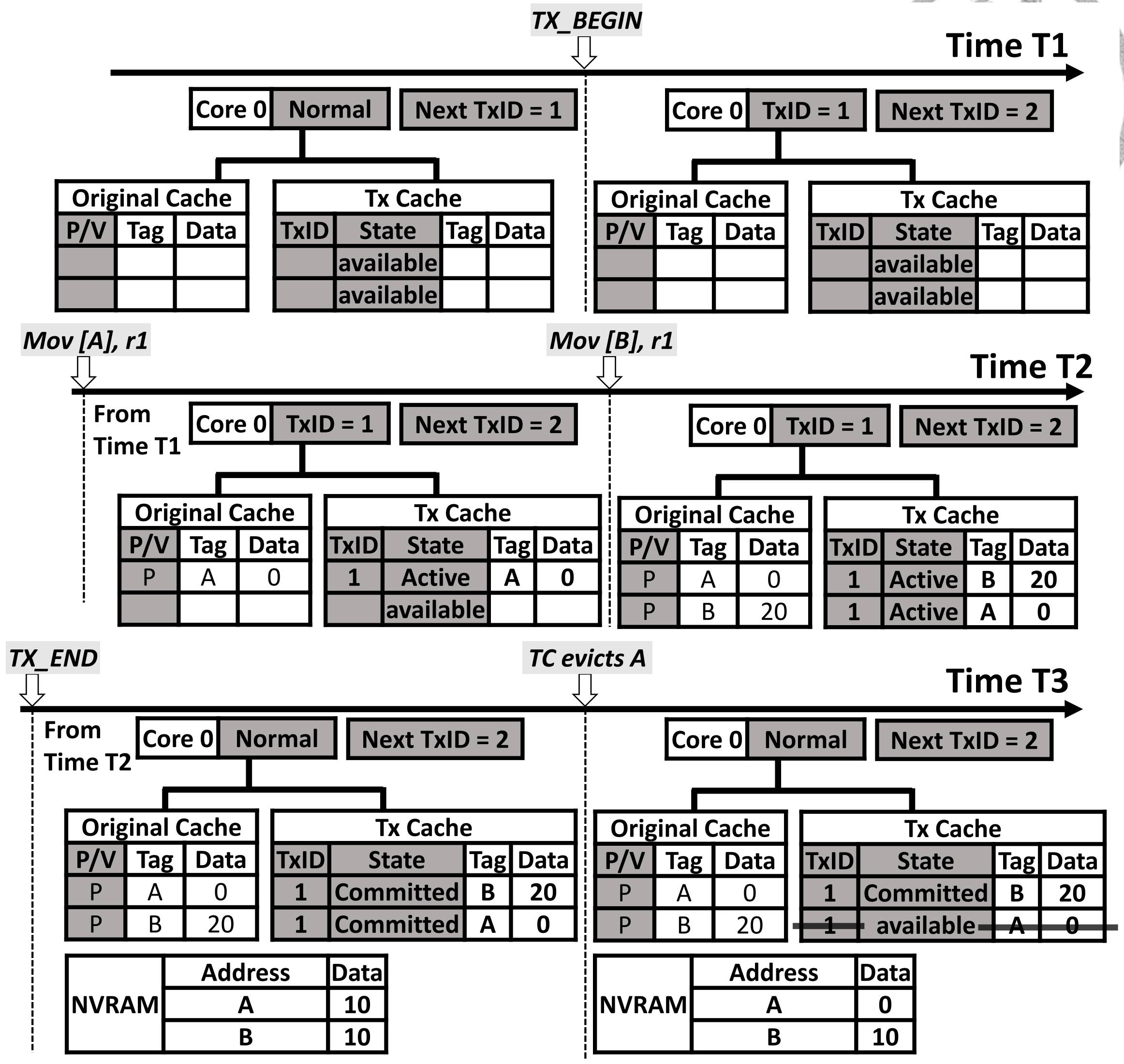


Figure 3.9: Detailed transaction cache execution example

the beginning, all the entries in the transaction cache is in available state. After receiving requests from CPU, the transaction cache will insert the data with transaction ID and set them into active state. We can see, from Figure 3.9., during time L1 to time L2, CPU 0 is handling store requests A and B. And during the same time, the transaction cache will receive corresponding store requests from CPU and insert them in the same as the issuing order from CPU. Cache line of request A is inserted first than the one of request B. After time T2, at executing ”TX_END”, CPU 0 will send a commit request with the transaction ID information to the transaction cache, set mode register as 0 and enter the normal mode. Receiving the commit request with the transaction ID, the transaction cache can use the transaction ID of the commit request to set all the state of the cache lines of

doi:10.6342/NTU201602526

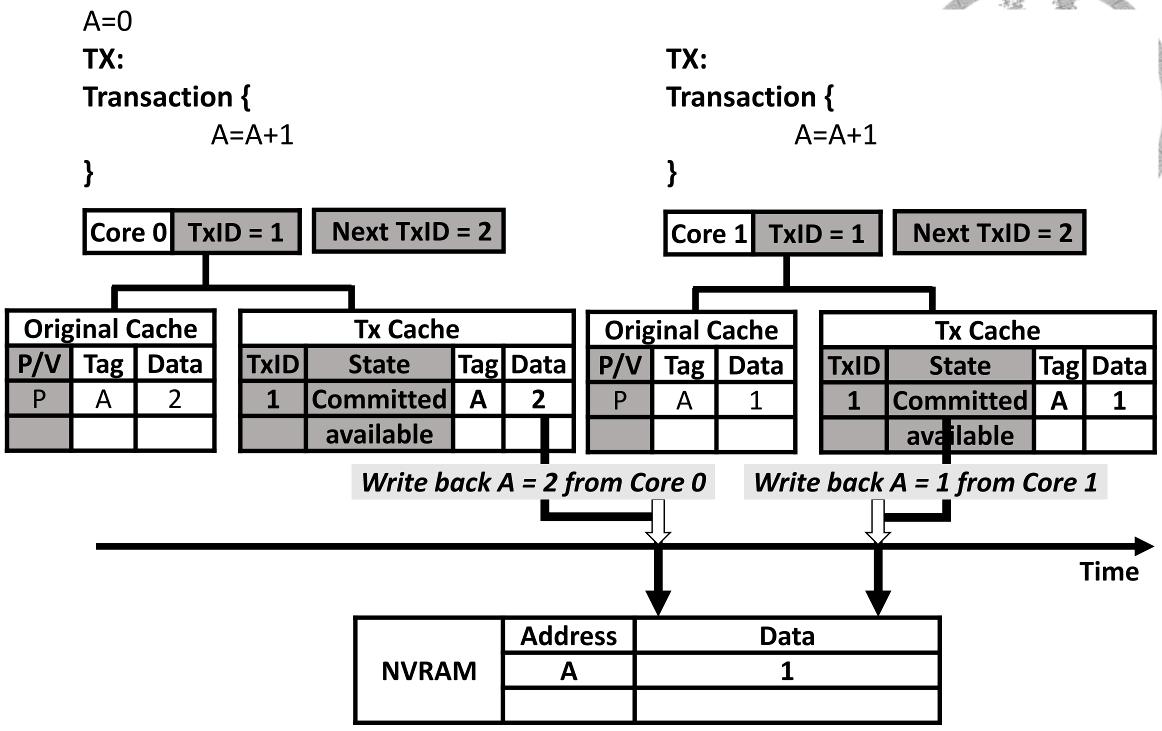


Figure 3.10: Eviction order problem on multi-cores

a transaction into committed state and commit the transaction. After this moment, all the committed cache lines of the transaction 0 can be evicted toward NVRAM. After evicting the committed cache line A, the data A in NVRAM is updated from original version 10 to new version 0. And after writing the data A, the NVRAM controller will send back the acknowledgment message toward the transaction cache and the transaction cache can remove the cache line A, set the corresponding entry into available state and make room for future writes.

3.4 Detailed Discussions of Multicore Systems

With multi-cores, one important thing to be noted is that different cores may touch the shared cache line of the same address. At this time, the execution order have to be carefully handled. Different cores have to see the shared data in the same value. That's why we need cache coherence. But, in the above design, the transaction caches of different cores are not synchronized, which may results to two problems.

Eviction order problem: The first problem is that the insertion or write-back order

doi:10.6342/NTU201602526

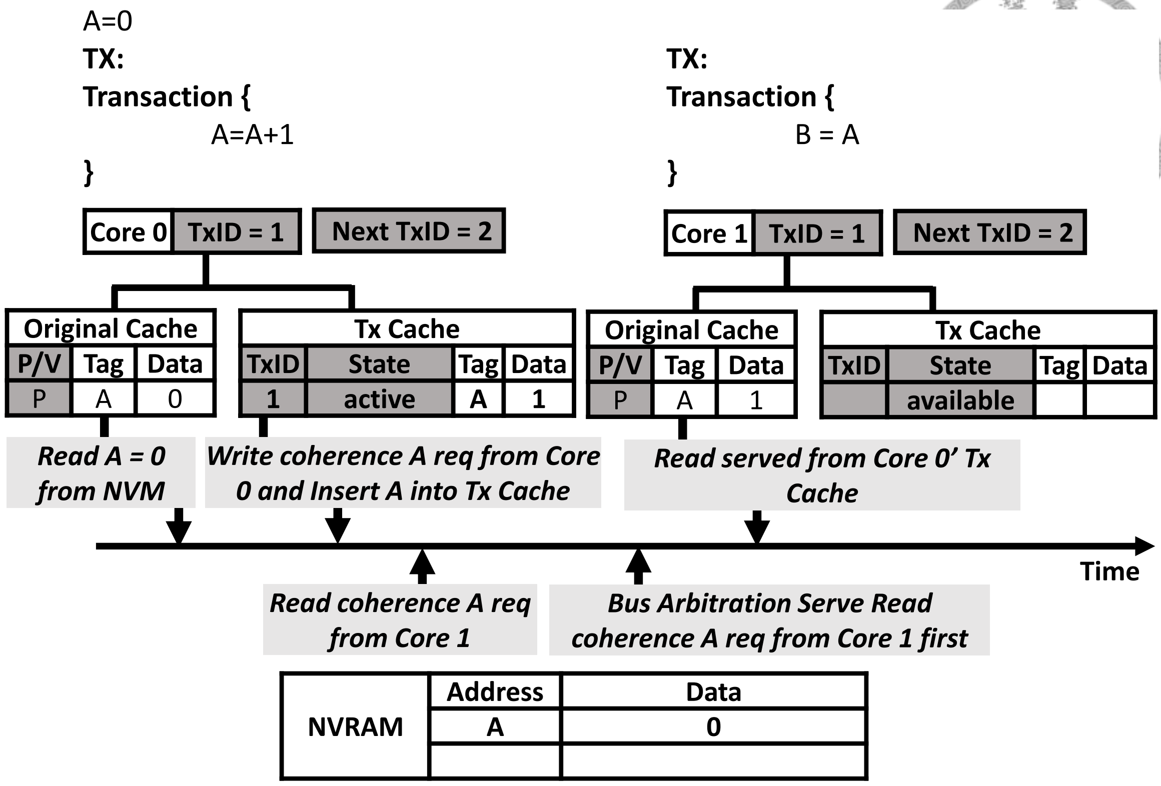


Figure 3.11: Early read problem on multi-cores

toward the transaction cache may be different from the results of cache coherence. For example, in Figure 3.10, two cores 0 and 1 execute a transaction to increase the shared variable A by 1. Core 1 has variable A = 1 and core 0 has variable A = 2, so it is obvious that the execution order and cache coherence results is that core 1 executes earlier than core 0. Both cores finish the transaction and backup data in the transaction cache. At this moment, the transaction cache will issue the write-back request of the committed variable A toward the NVRAM at background. However, the issue order between different transaction caches is not synchronous. The transaction cache doesn't check if other transaction cache has the data of the same address to be written back. Therefore, the final result may not match the execution results on a multicore system. For example, in Figure 3.10, if the transaction cache of core 1 writes back the data of variable A afterward than core 2, then the final value of variable A is 1 but 2, which is inconsistent.

Early read problem: The second problem is that if we directly insert data into the transaction cache, early read problem may happen. The problem results from that our transaction cache is readable. If we insert data and write data into the transaction cache

doi:10.6342/NTU201602526

TC Request Queue	Write Request	TxID	SeqID	Address	Data
		1	0 (Invalid)	0xAAAA	0
		TxID	SeqID	Address	Data
		1	1	0xBBBB	20

Figure 3.12: Transaction request queue

earlier than the cache coherence results, read requests from other cores may be served from the transaction cache that should be seen afterward than the read request. Fig. 3.11. is an example that read miss on variable A from core 1 is early served from the new version variable A in the transaction cache that actually will be executed and seen by other cores after core 1's read miss.

To solve above problems, we have to ensure that the write-back order matches the cache coherence and prevent making data readable earlier than the cache coherence results. To prevent early read problem, we can add a flag into transaction requests to indicate that if the cache coherence operation of this request is finished to know that the data can be inserted into the transaction cache. And to ensure eviction order, a naive way is to utilize the cache coherence mechanism to let cache transfer the dependency order between different cores after the cache coherence. However, this will make cache controller design more complicated because cache controllers have to track and transfer the dependency information in the messages during cache coherence. Besides, the transaction cache have to check if the corresponding dependent cache line on the other core has been issued to NVRAM, which is not simple.

Differently, we apply a simple method to know the write-back order. What we do is to record the global write order results after cache coherence, but not the detailed coherence relation between different cores. We add two registers (global sequence id and global write-back sequence id) into our design.

- **Global sequence id**: record the total write order among all the cores.
- **Global write-back sequence id**: record the last sequence id issued toward NVRAM to synchronize the write-back order between different transaction caches.

Each write request in the TC request queue is added a sequence id flag as Figure 3.12 shows. Sequence id records the order of each write request and indicate that if the write

doi:10.6342/NTU201602526

request finishes cache coherence. Write request with sequence id larger than zero means that the request finishes cache coherence and can be handled by the transaction cache controller and inserted into the transaction cache data array. After finishing a cache coherence, L1 cache controller will issue a finishing signal toward the TC request queue. Receiving the finishing signal, the sequence id of the write request at tail [2] in the TC request queue will be filled with the value from the the global sequence id and the global sequence id will then automatically increase by one to maintain the order of different write requests.

The cache line in the transaction cache data array also have to record the sequence id, which will be copied from the write request. With the recorded sequence id, to write back a cache, the transaction cache have to check the sequence id of the cache line. If the sequence id of the cache line matches the global write-back sequence id, then it can be issued toward NVRAM. After issuing the cache line with matched id, the global write-back sequence id will increase by one to make the next cache line with matched id issuable. By this way, write-back order of the cache lines from transaction caches of different cores will match the cache coherence results.

3.4.1 Cache Hierarchy Modification for Mulicore Systems

LLC modification to get dropped nonvolatile requests: Different transaction caches may have the data of the same addresses and only one among them is the newest. To get the new version dropped data, LLC will issue read miss request toward transaction caches and NVRAM and use data from transaction cache at first because data from transaction cache are newer than NVRAM. But, one important thing to be pointed out that different transaction caches may all have different version data of the same cache line, but only one among them is the newest one. To solve this, LLC can simply use the cache line returned from transaction caches with the biggest SeqID, which means that it is the newest inserted data.

[2]Under x86 architecture, L1 cache controller will follow the total store order memory consistency model and handle write requests in program order. And thus, at receiving a finishing signal, it must indicate that the write request at tail of the TC request queue finishes cache coherence.

doi:10.6342/NTU201602526

3.4.2 Hardware Overhead for Multicore Systems

The additional storage overhead is the global sequence id, the global write-back sequence id and the sequence id recorded in the write request and the TC data array. For a core with a 4KB transaction cache size, if each transaction only touches one cache line, there will be at most 256 executed transactions (4 * 1024 / 64 * 4) on a multi-core system of 4 processors, so the additional bits of the sequence id are 8 bits.

doi:10.6342/NTU201602526

Chapter 4

Experimental Results

4.1 Experimental Setup

Table 4.1: Machine Configuration

Device	Description
CPU	4 cores, 2GHz, 4 issue, out of order
L1 I/D	Private, 32KB/core, 4-way LRU, 64B line, 3-cycle, 1.5ns
L2	Private, 256KB/core, 8-way LRU, 64B line, 9-cycle, 4.5ns
L3 (LLC)	Shared, 64MB, 16-way LRU, 64B line, 20-cycle, 10ns
Transaction Cache (STTRAM)	Private, 4KB/core Fully-Associative CAM FIFO [21, 30], 64B line, 21-cycle, 10.5ns [29]
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, 64B row size, 65-ns read, 76-ns write [36]
DRAM Memory	DDR3 8GB, 4 ranks, 8 banks/rank, 64B row size

To evaluate the transaction cache design, we conduct experiments with MARSSx86 [20] and DRAMsim2 [26]. MARSSx86 is a cycle-accurate full-system simulator that uses PTLsim [35] 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 is modified to simulate related works and the transaction cache design of this work, which serves the miss requests from LLC and controls the

Table 4.2: Workloads

Name	Description
graph	Randomly generate an edge. If it is not added, add it into the adjacency list graph.
rbtree	Randomly search a value in a red-black tree. If it doesn't exist in tree, then insert it into the red-black tree.
sps	Randomly swap two different elements in an array.
btree	Randomly search a value in a B+tree. If it doesn't exist in tree, then insert it into the B+tree.
hashtable	Randomly generate a key-value pair. If it doesn't exist, then insert it into the hashtable.

write ordering toward NVRAM. NVRAM model of DRAMSim2 is modified to transfer back an acknowledgment message after finishing a persistent write. We set up a multi-cores system with four cores, which is 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 [29, 36]. The default transaction cache has 4KB size and 10.5ns latency. Table 4.1 describes the core and memory configurations.

To analyze different 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 many of databases and file systems. The size of all manipulated key and value data in these benchmarks is 64 bits. We simulate each benchmark for 1.7 billion instructions. Table 4.2 lists the descriptions of the benchmarks.

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 [36]**

doi:10.6342/NTU201602526

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.

4.2 Performance Evaluation

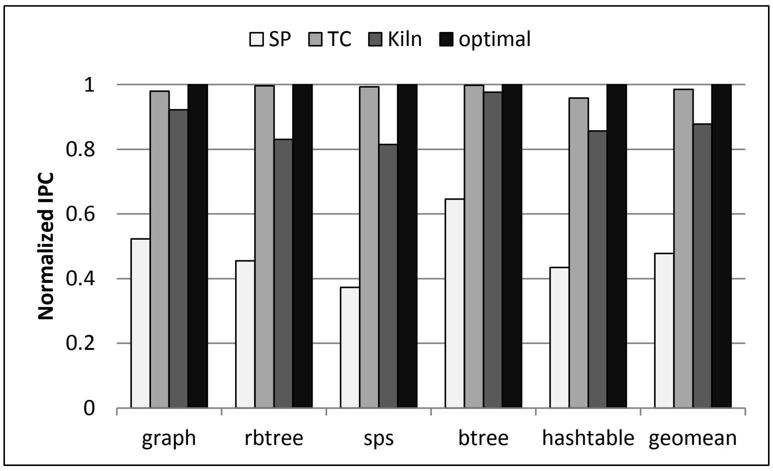


Figure 4.1: Performance improvements (IPC)

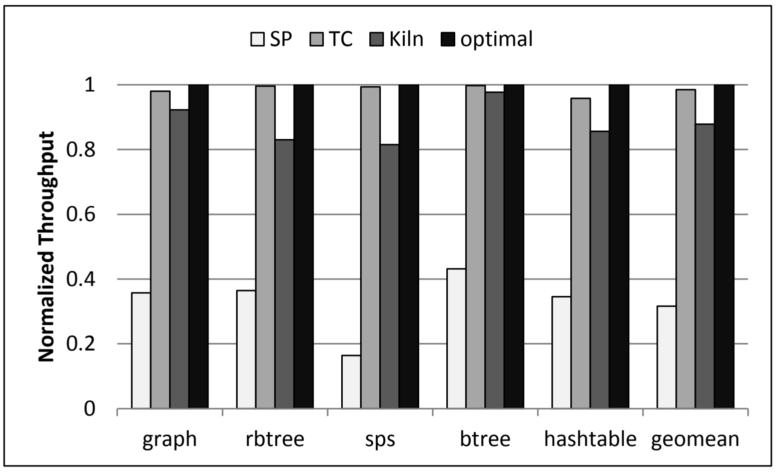


Figure 4.2: Performance improvements (Throughput)

doi:10.6342/NTU201602526

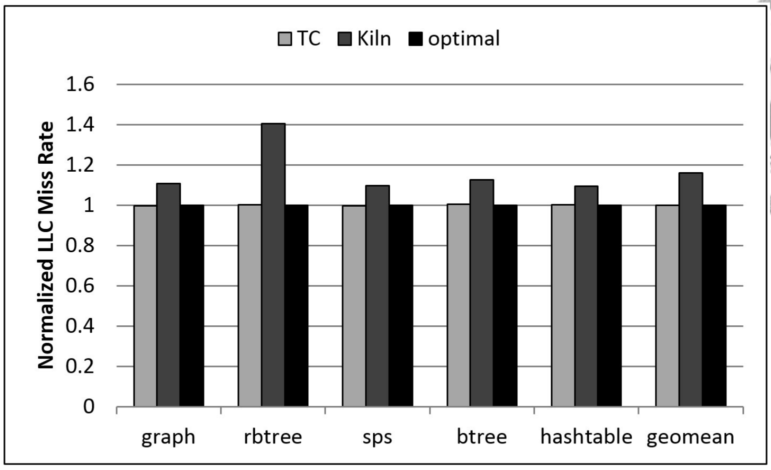


Figure 4.3: LLC miss rate

Figure 4.1 and Figure 4.2 show the IPC (instructions per cycle) and throughput (transactions per cycle) of various schemes normalized to the optimal. We can see that 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 and memory requests during transaction commits and results in bursts of traffic in the cache hierarchy. Besides, to replace a cache line in the LLC, Kiln have to wait the to-be-replaced cache line for writing back into NVRAM and results to longer load/store latency for bringing data into the LLC. Figure 4.3 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

doi:10.6342/NTU201602526

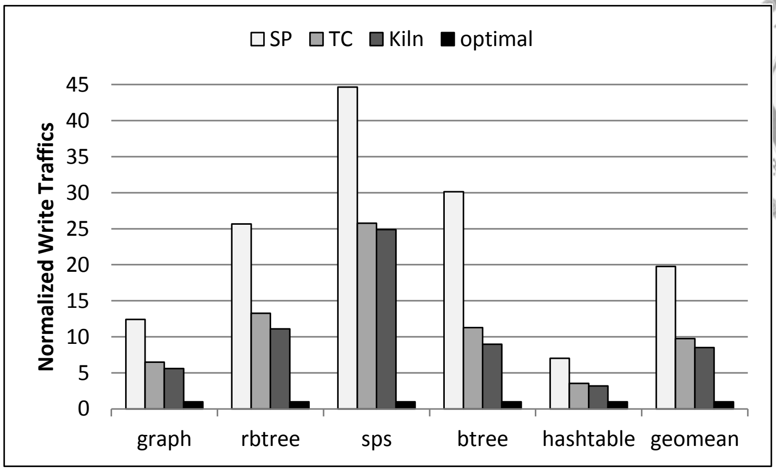


Figure 4.4: Write traffic

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 4.4 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.

The increased write traffic will affect the latency of nonvolatile read requests and results to performance degradation. Figure 4.5 shows the NVRAM read latency of various schemes normalized toward the optimal case. In average, TC and Kiln have 48.3% and 21.73% longer read latency of the optimal case respectively, which is unavoidable for persistence.

doi:10.6342/NTU201602526

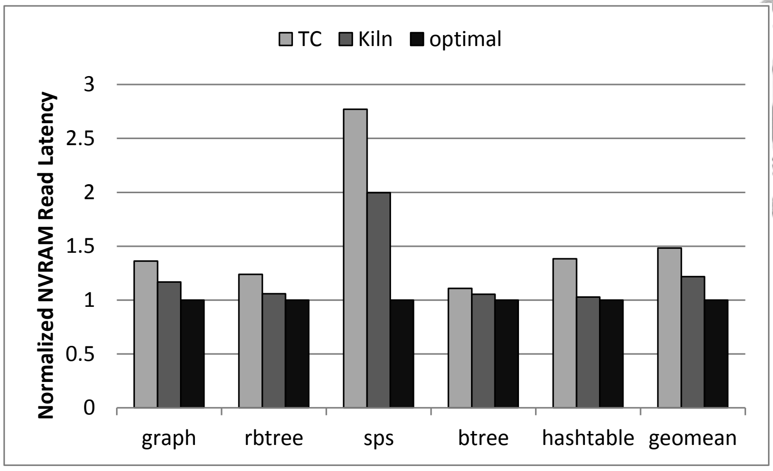


Figure 4.5: Read latency

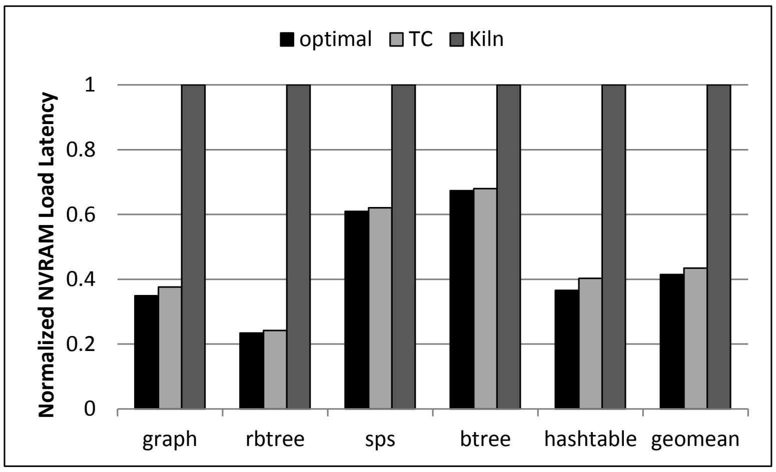


Figure 4.6: Load latency

Figure 4.6 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, which only increases a little because of write traffic.

4.3 Sensitivity Analysis

Figure 4.7 shows the throughput under different transaction cache size from 1KB to 8KB normalized to the optimal case. The transaction cache size affects how often the CPU has to stall for persistent writes. As we can see, for the benchmarks tested in this work, a 4K TC size is quite sufficient. Even with 1K and 2K TC, it has only 1.8% and 0.2%

doi:10.6342/NTU201602526

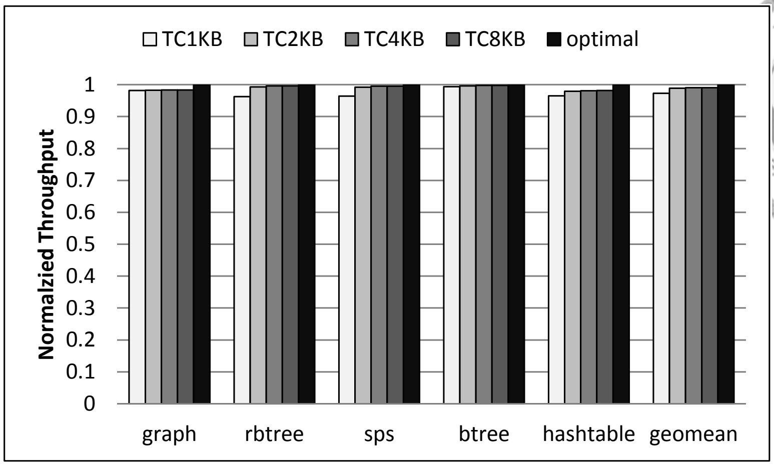


Figure 4.7: Throughput under different transaction cache size

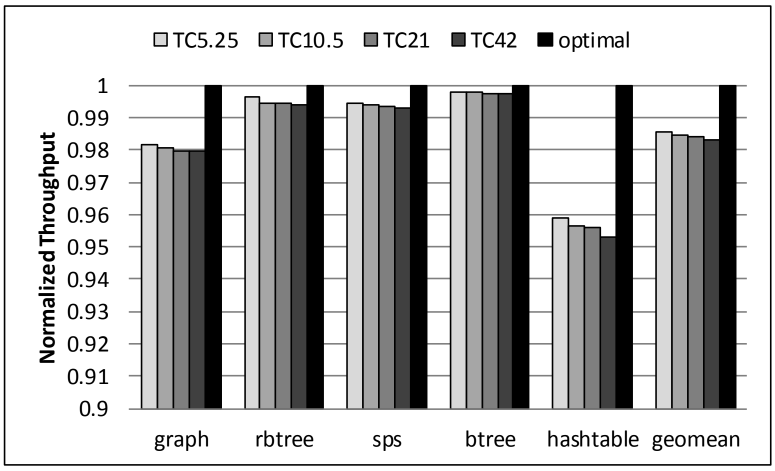


Figure 4.8: Throughput under different transaction ache latency

performance degradation compared to the 4KB transaction cache.

Figure 4.8 shows the throughput under different transaction cache latency from 5.25ns to 42ns normalized to the optimal case (the default latency is 10.5ns). The transaction cache latency affects the time to serve miss requests from LLC and insert transaction data into TC. As we can see from the results, our design is quite insensitive to the TC latency. The reason is that with the new persistent path, operations related to maintaining atomicity or write ordering is decoupled from the program execution.

doi:10.6342/NTU201602526

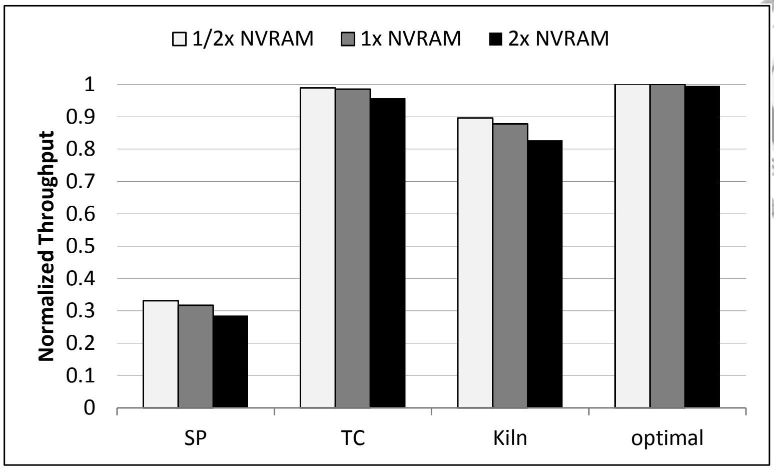


Figure 4.9: Throughput under different NVRAM latency

Figure 4.9 shows the average throughput under different NVRAM latency from 0.5x to 2x (the default 1x latency is 65ns read, 76ns write). With 2x NVRAM latency, TC can still achieve 96.2% of the optimal performance. We can also observe that Kiln is more sensitive to the NVRAM latency. With a better NVRAM technology, i.e., 0.5x latency, TC can further improve performance, 98.8% performance of the optimal case.

doi:10.6342/NTU201602526

Chapter 5

Related Works

5.1 NVRAM Access Library

Persistent memory utilizes nonvolatile memory (NVRAM) as main memory level storage. Compared to traditional disk or flash storage, persistent memory brings many performance benefits with better access latency of NVRAM and closer layer toward CPU. To exploit and use NVRAM more efficiently without compromising the performance potential of NVRAM, many works pay much effort into it. To make programmer easily operate and allocate data into nonvolatile memory, NV-heap [2], Mnemosyne [32], nvml [13], nvm-direct [19] and HEAPO [10] propose to build NVRAM library to help programmer operate data in NVRAM like traditional memory allocation and management library. With these libraries, programmer can utilize new proposed API to easily develop and build programs that target and use NVRAM.

5.2 NVRAM File System

Besides of allocating data into NVRAM via load/store interface, as a storage, it is common that we manage and operate data via file system abstraction. However, because of different access speed and operation behavior (load/store) of NVRAM compared to disk and flash, traditional file system may not suit persistent memory well. Directly utilizing traditional file systems may decrease the benefits of NVRAM and results to large soft-

doi:10.6342/NTU201602526

ware overhead. BPFS [3], PMFS [5], SCMFC [33], Aerie [31] and NOVA [34] propose to rebuild file systems that utilizes the fine-grained access behavior of NVRAM to eliminate software overhead and remove duplicate buffer cache copy in DRAM, which suits persistent memory well.

5.3 Write Ordering for NVRAM Persistence

To utilize NVRAM as persistent storage, data persistence guarantee is an important topic. To ensure data persistence, we need to ensure the data write ordering toward NVRAM. Mnemosyne [32], nvm-direct [19] and nvml [13] utilize memory fence (sfence and mfence) and cache flushing instructions (clflushopt, clwb and pcommit) to ensure the write ordering. However, utilizing these instructions will delay later requests to be handled, bound ordering with durability and result to poor performance. To solve this, epoch barrier [3] proposes to use a new software and hardware primitive to make software declare ordering constraint toward hardware. Receiving these information, cache eviction and memory scheduling way will follow the software declared rule. DP2 [28] finds out that epoch barrier will push too much constraint on memory controller and relaxes the ordering constraint between logs and target data updating operations of WAL. Strand persistence [22] points out that program may not only have one single ordering rule from the beginning till the end of a program and can have different ordering dependency among different part of a program and thus, proposes strand command to make program to start a new ordering constraint among a program. Giles and Doshi et al. [7, 4] propose a different direction and delegate the logging and real data updating operation toward the NVRAM controller. NVRAM controller persists data at background in parallel with CPU execution. Joshi et al. [12] finds out that even though epoch barrier makes cache and memory layer follow the ordering at background in parallel with CPU execution, to evict a cache line and ensure the ordering, it will result to long cache stall time, let cache eviction become critical path and thus, affect the overall performance. Therefore, Joshi et al. propose to proactively evict the cache lines earlier to remove the ordering constraint from critical path.

doi:10.6342/NTU201602526

5.4 Write Atomicity for NVRAM Persistence

In addition to the write ordering, increased write traffic of duplicated writes to protect writes atomicity and avoid half completed operations for persistence after a power failure is also an severe problem. To reduce the write traffic toward NVRAM and increase cache efficiency, Loose-ordering [17] proposes to utilize multi-version backup at cache layer and speculatively write back the data. Kiln [36] applies nonvolatile last level cache to eliminate backup data operations overhead. WSP [6] proposes to use the residual energy to save whole system state and can continue a program execution after a power failure. With the ability to resume a program after a crash, half finished operations will not be a problem and duplicated writes for atomicity are removed. Instead of whole system states including device states, ThyNVM [24] proposes that checkpointing only data state and CPU execution state can ensure data persistence. Besides, thyNVM overlaps execution with checkpointing operation to hide persistence overhead.

To improve performance under persistence constraints, all above methods follow the design way that changing the existing cache hierarchy or memory layer to avoid persistence overhead. However, it is complicated to combine the existing cache hierarchy logic with these new design and changing the existing cache hierarchy or memory just propagates the persistence overhead toward them and results to performance degradation at the critical time to ensure persistence. Our method provide a new path to extract the persistence overhead out of the overall architecture, conquer it and solve the write traffic and ordering problem at once.

5.5 Scheduling Methods for Stand-alone NVRAM Main Memory System

On the other side, NVM duet [16] and FIRM [37] improve the performance of the system that uses NVRAM as stand-alone main memory without DRAM. Both volatile and persistent write requests will go toward NVRAM, so a well-defined scheduling method of

doi:10.6342/NTU201602526

NVRAM controller is important. NVM duet makes NVRAM controller schedule volatile requests without persistent constraints and prioritizes the persistent requests which are more critical than volatile requests. FIRM analyzes the volatile and persistent workload behaviors, balances the time to process volatile or persistent requests and reduces the bus-turnaround time for the write intensive property of persistent workloads.

doi:10.6342/NTU201602526

Chapter 6

Conclusion

To ensure data persistence of persistent memory, the software solution based on Intel released new instructions incurs much performance degradation. Extra fence and backup operations cause longer write latency and the write ordering pushes much execution burden on CPUs. To solve these problems, related works propose to modify cache hierarchy or memory controller to hide persistence overhead at background in parallel with CPU execution. However, these design philosophies will propagate the persistence overhead toward the existing cache hierarchy or memory layer at some critical moments. Differently, we attempt a new direction, propose an efficient hardware mechanism to free the existing hardware architecture from ensuring the write ordering, paves a new persistent path via additional nonvolatile hardware (transaction cache) to eliminate extra backup operations and be responsible for the write ordering. The experimental results show that our efficient hardware mechanism eliminates the software overhead to ensure data persistence, extracts persistence overhead from the existing architecture and achieves the performance near the one without data persistent guarantee.

doi:10.6342/NTU201602526

Bibliography

[1] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 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. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[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. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[4] K. Doshi, E. Giles, and P. Varman. Atomic persistence for scm with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 77–89, March 2016.

[5] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

doi:10.6342/NTU201602526

[6] O. H. Dushyanth Narayanan. Whole-system persistence with non-volatile memories. In *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*. ACM, March 2012.

[7] E. Giles, K. Doshi, and P. Varman. Bridging the programming gap between persistent and volatile memory using wrap. In *Proceedings of the International Conference on Computing Frontiers*, pages 30:1–30:10, 2013.

[8] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[9] Y. Huai. Spin-transfer torque mram (stt-mram): Challenges and prospects. *AAPPS Bulletin*, 18(6):33–40, 2008.

[10] T. Hwang, J. Jung, and Y. Won. Heapo: Heap-based persistent object store. *ACM Transactions on Storage (TOS)*, 11(1):3, 2015.

[11] Intel. Instruction set extensions programming reference manual 319433-024 february. *https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf*, 2016.

[12] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 660–671, New York, NY, USA, 2015. ACM.

[13] C. Krzysztof and R. Andy. Linux nvm library. *https://github.com/pmem/nvml*.

[14] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 256–267. IEEE, 2013.

doi:10.6342/NTU201602526

[15] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. *SIGARCH Comput. Archit. News*, 37(3):2–13, June 2009.

[16] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang. Nvm duet: Unified working memory and persistent store architecture. *SIGPLAN Not.*, 49(4):455–470, Feb. 2014.

[17] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pages 216–223. IEEE, 2014.

[18] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.

[19] Oracle. Nvm direct. *https://github.com/oracle/NVM-Direct*.

[20] A. Patel, F. Afram, S. Chen, and K. Ghose. Marss: A full system simulator for multicore x86 cpus. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 1050 –1055, june 2011.

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

[22] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA ’14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.

[23] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA ’09, pages 24–33, New York, NY, USA, 2009. ACM.

[24] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the*

doi:10.6342/NTU201602526

48th International Symposium on Microarchitecture, MICRO-48, pages 672–685, New York, NY, USA, 2015. ACM.

[25] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. *SIGARCH Comput. Archit. News*, 28(2):128–138, May 2000.

[26] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16–19, 2011.

[27] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *nature*, 453(7191):80–83, 2008.

[28] L. Sun, Y. Lu, and J. Shu. Dp2: Reducing transaction overhead with differential and dual persistency in persistent memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 24:1–24:8, New York, NY, USA, 2015. ACM.

[29] 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. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 329–338, New York, NY, USA, 2011. ACM.

[30] 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.

[31] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 14. ACM, 2014.

[32] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *SIGPLAN Not.*, 47(4):91–104, Mar. 2011.

[33] X. Wu and A. L. N. Reddy. Scmfs: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing,*

doi:10.6342/NTU201602526

Networking, Storage and Analysis, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[34] J. Xu and S. Swanson. Nova: a log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

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

[36] 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. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.

[37] J. Zhao, O. Mutlu, and Y. Xie. Firm: Fair and high-performance memory control for persistent memory systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 153–165, Washington, DC, USA, 2014. IEEE Computer Society.

doi:10.6342/NTU201602526