Relaxing Persistent Memory Constraints with Hardware-Driven Undo+Redo Logging

Matheus A. Ogleari
mogleari@ucsc.edu

Ethan L. Miller
elm@ucsc.edu

Jishen Zhao
jishen.zhao@ucsc.edu

University of California, Santa Cruz

November 19, 2016

Abstract

Persistent memory is a new tier of memory that functions as a hybrid of traditional storage systems and main memory. It combines the benefits of both: the data persistence property of storage with the fast load/store interface of memory. Yet efficiently supporting data persistence in memory requires non-trivial effort. In particular, most previous persistent memory designs place careful control over the order of writes arriving at persistent memory. This can prevent caches and memory controllers from optimizing system performance by coalescing and reordering the writes. We identify that such write-order control can be relaxed by employing undo+redo logging of data in persistent memory systems. However, traditional software logging mechanisms are expensive to adopt in persistent memory, due to their overhead on performance, energy, and implementation. Alternate proposed hardware-based logging schemes don't fully address the issues with software.

To address the challenges, we propose a hardware-driven undo+redo logging scheme, which maintains data persistence by leveraging the write-back write-allocate caching policy available in commodity processors. Furthermore, we develop a hardware-based cache force-write-back mechanism, which significantly reduces the performance and energy overheads in forcing persistent data into the persistent memory. Our evaluation across persistent memory microbenchmarks and real workloads demonstrates that our design leads to significant system throughput improvement and reduction in both dynamic energy and memory traffic.

1 Introduction

Persistent memory is envisioned as a new tier of data storage components to be adopted in future computer systems. By attaching nonvolatile random-access memories (NVRAMs) [46, 7, 53, 22] to the memory bus, persistent memory unifies memory and storage systems, offering the fast load/store access of memory and data recoverability of storage in a single device. Seeing the great value, various hardware and software vendors have recently begun to adopt the persistent memory technique in their next-generation designs – e.g., Intel's ISA and programming library support for persistent memory [21, 20], ARM's new cache write-back instruction [1], Microsoft's storage class memory support in Windows OS and in-memory databases [9, 13], Red Hat's persistent memory support in the Linux kernel [38], and Mellanox's persistent memory support over fabric [12].

Though promising, persistent memory fundamentally changes current memory and storage system design assumptions – reaping its full potential is challenging. In fact, most previous persistent memory designs introduce much performance and energy overheads compared with a native memory system without persistence support [39, 18, 51]. One of the key reasons is the write-order control used to enforce data persistence. Commodity processors delay, combine, and reorder writes in caches and memory controllers to optimize system performance [19, 26, 41, 51]. Yet, most previous persistent memory designs employ memory barriers and cache force-write-backs (or cache flushes) to enforce the order of persistent data arriving at NVRAM. Such write-order control prevents the performance optimization offered by the natural caching and memory scheduling mechanisms.

Several recent studies strive to relax such write-order control in persistent memory systems [26, 41, 51]. However, these studies either impose substantial hardware overhead by adding NVRAM caches in the processor [51] or fall back to low-performance modes once certain bookkeeping resources in the processor are saturated [26].

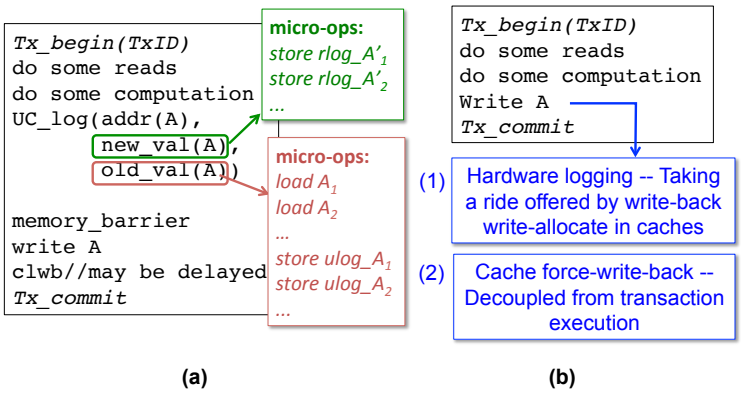

Figure 1: Example code of a transaction by employing (a) traditional software-based undo+redo logging (the log is uncacheable (UC), new value of A consists of stores of A'_1, A'_2, etc.) and (b) our design incorporated with two design principles.

Our **goal** in this paper is to design a high-performance persistent memory system without (i) NVRAM cache/buffer in the processor, (ii) falling back to a low-performance mode, or (iii) interfering with the write reordering by caches and memory controllers. Our key idea is to maintain data persistence with a hardware-driven undo+redo logging scheme.

Undo+redo logging, which stores both old values (undo) and new values (redo) in the log during a persistent data update, offers a key benefit – relaxing the write-order constraints on the caching of persistent data in the processor. In our paper, we identify that undo+redo logging can ensure data persistence without the need for the strict write-order control. As a result, the caches and memory controllers can reorder the writes in the same way as traditional non-persistent memory systems (details discussed in Section 2.2).

However, enabling undo+redo logging in persistent memory is nontrivial. Most previous persistent memory systems implement either undo or redo logging in software. However, several inefficiencies hamper feasible, high-performance software-based undo+redo logging in persistent memory. First, the software logging function introduces extra instructions, competing for the precious hardware resources in the pipeline with other critical workload operations; undo+redo logging can even double the number of extra instructions than either undo or redo logging alone. Second, logging introduces extra memory traffic in addition to working data access [51]; undo+redo logging imposes more than double extra memory traffic. Third, CPU caches are hardware components invisible to the software. As a result, software-based undo+redo logging mechanisms, which are borrowed from database mechanisms designed to coordinate with software-managed caches, can only conservatively coordinate with CPU caches. Finally, with multithreaded workloads, context switches by the operating system (OS) can interrupt the logging and persistent data updates in the memory. This can potentially risk the data persistence guarantee in multithreaded environment. (Section 2.3 discusses further details).

How about implementing it in hardware? In fact, several prior works investigated hardware-based undo or redo logging [36, 26] (Section 8). However, most previous hardware-based logging imposes similar challenges found in most previous hardware-based persistent memory designs, such as hardware and energy overheads [36], and slowdown due to saturated hardware bookkeeping resources in the processor [26]. Supporting both undo and redo logging can even exacerbate the issues. Furthermore, hardware-based mechanisms can eliminate the logging instructions in the pipeline, but the extra memory traffic of log updates still exists.

To address these challenges, we propose a hardware-driven undo+redo logging scheme that allows persistent memory systems to relax the write-order control in persistent memory systems by leveraging the caching mechanism available in most commodity processors. Our design consists of two principles as shown in Figure 1(b). First, we propose a **Hardware Logging** mechanism that performs undo+redo logging by leveraging the write-back write-allocate cache policy [19] available in most commodity processors. Persistent working data update `write A` will trigger the our logging process to be performed in parallel with the stores of this data update. Whether a store generates an L1 cache hit or miss, its address, old value, and new value are all available in the cache hierarchy. As such, our design takes the ride offered by the cache block writes to update the log with word-size values. Second, we propose a **Progressive Cache Force-write-back (FWB)** mechanism to force-write-back the cached working data in a much lower frequency than software-based mechanisms. FWB forces cache write-backs at a frequency that only depends on the allocated log size and NVRAM write bandwidth, decoupling cache force-write-backs with transaction execution. We summarize

the contributions of this paper as following:

- We identify the opportunities of relaxing the write-order constrains on caches by exploiting undo+redo logging. With undo+redo logging, we effectively replenish the natural caching behavior in persistent memory systems.
- We enable high-performance, low-implementation-cost undo+redo logging in persistent memory, by leveraging the natural caching procedure available in commodity processors.
- We develop a hardware-controlled cache force write-back, which significantly reduces the performance overhead of force write-backs by tuning the force write-back frequency.
- We implement our design by devising a set of lightweight software support and processor modifications.

2 Background and Motivation

2.1 Persistent Memory Write-order Control

Persistent memory is fundamentally different from traditional DRAM-based main memory or their NVRAM replacement, due to its persistence (aka. crash consistency) property inherited from storage systems. Persistent memory systems need to ensure the integrity of in-memory data despite system failures, such as system crashes and power outages [2, 48, 42, 37, 41, 25, 31, 32, 24, 30, 8]. The persistence property is not guaranteed by memory consistency of traditional memory systems – memory consistency ensures a consistent global view of processor caches and the main memory, while persistent memory needs to ensure that the data in the NVRAM main memory standing alone is consistent [41, 48, 25].

In order to maintain data persistence, most persistent memory systems employ transactions to update the persistent data and place careful control over the order of writes arriving at NVRAM [41, 48, 4]. Each transaction (Figure 1) consists of a group of persistent memory updates performed in the manner of "all or nothing" in the face of system failures. Persistent memory systems also insert cache force-write-backs (e.g., `clflush`, `clwb`, and `dccvap`) and memory barriers (e.g., `mfence` and `sfence`) inside and across the transactions to enforce the write-order control [4, 26, 51, 48, 10]: Cache flushes/force-write-backs ensure that the cached data updates made by completed (i.e., committed) transactions are forced into the NVRAM, so that the NVRAM is in a consistent state with the latest data changes; Memory barriers stall subsequent data updates until the previous updates made by the transaction, including cache force-write-backs, are complete.

Such write-order control prevents caches and memory controllers from optimizing system performance by coalescing and reordering writes. The cache force-write-backs and memory barriers can also block or interference with subsequent read and write requests that share the memory bus, even these requests are independent from the persistent data access [30, 52].

2.2 Undo+Redo Logging and Its Benefits

Undo+Redo Logging. Undo+redo logging is widely used in database management systems (DBMS's) to support data persistence [35]. In addition to the working data updates, persistent memory systems can maintain alternative copies of the updates in the log. The undo log stores old versions of data before a transaction makes any changes to the data. As such, in case system fails in the middle of the active transaction, the system roll back to the state before the transaction starts by replaying the undo log. The redo log stores new versions of data. In this case, a transaction needs to first write the data updates into the redo log, before it makes changes the working data in-place. After system failures, replaying the redo log can recover the persistent data with the latest changes being tracked in the redo log. In persistent memory systems, logs are typically uncacheable, because they are only meant to be accessed during the recovery – not reusable during application execution (Figure 1(a)).

Write-order Control with Logging. Redo and undo logging require different granularities in write-order control. With redo logging, we need to ensure that log updates in a transaction arrive at NVRAM, before we can start updating the working data in-place (Figure 2(a)). Otherwise, working data updates can be evicted out of the last-level cache in the middle of redo logging. Both working data structure and the redo log can be corrupted. Undo logging allows for finer-granularity write-order control (Figure 2(b)). As long as the old values of a small piece of working data is copied into the undo log, we can go ahead to overwrite that piece of working data. Persistent memory systems can

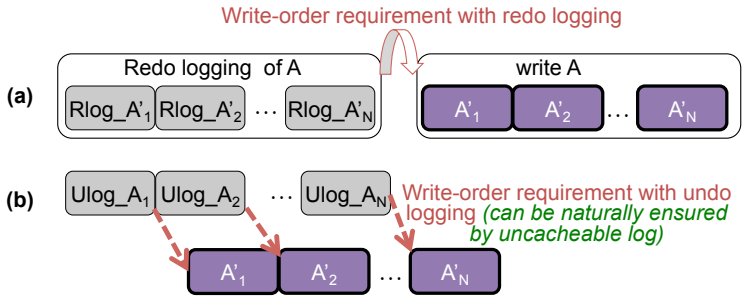

Figure 2: Different write-order requirements with (a) redo and (b) undo logging, respectively.

undo each piece of changes using the logged old values. Note that undo-logging-based persistent memory systems can eliminate the explicit write-order control by adopting an uncacheable log with word-size entries – the cached working data stores will take longer time to traverse through the cache hierarchy.

Relaxing Write-order Control with Undo+Redo Logging. Undo and redo logging can offer complementary benefits on write-order control in persistent memory systems. With uncacheable, word-granularity undo logging, we can potentially eliminate the memory barrier (shown in Figure 1(a)) between logging and working data update. This is often referred to as the "steal" attribute in DBMS, i.e, cached working data updates can steal the way into persistent storage before transaction commits. However, undo-logging-based systems need to force cache write-backs in each transaction (e.g., using `clwb` illustrated in Figure 1(a)), in order to ensure that the latest changes are persisted in the NVRAM after the transaction commits. Redo-logging-based systems, in contrast, require the memory barrier but do not need to force cache write-backs in the transaction. This attribute is referred to as "no-force" in DBMS, i.e., no need to force the working data updates out of the caches at the end of transactions. As a result, undo+redo logging enables "steal/no-force", relaxing the write-order control in persistent memory systems by eliminating both memory barriers and cache force-write-backs inside the transactions.

2.3 Inefficiencies of Software-based Logging

Undo+Redo logging has been used in many traditional database systems stored in disk/flash [35, 17]. But none of previous persistent memory systems appear to use it to maintain data persistence, because it appears costly to be adopted in memory systems that provides much higher access bandwidth than disk/flash. The performance cost stems from several reasons.

First of all, software-based logging employs a logging function. As shown in Figure 1(a), the both undo and redo logging will be translated into a large number of instructions executed in the CPU pipelines. As we demonstrate in our experimental results (Section 6), only performing undo logging can lead to more than doubled instructions compared with memory systems without persistence support; undo+redo logging can introduce prohibitively large number of instruction overhead. Worse yet, these instructions can block the subsequent instruction execution in the pipeline due to the memory barrier instruction between logging and original data updates.

Second, most of the instructions in the logging function are load/store instructions. As such, the logging function can introduce substantial increase in memory traffic. In particular, undo logging not only needs to perform stores to update the log, but also needs to first perform loads to read the old values of the working data from the memory hierarchy. That increases memory traffic even more.

Third, processor caches and memory controllers are invisible to the software. Therefore, software can only conservatively manipulate the the caches in write-order control. As a result, such conservative control can introduce unnecessary memory barriers and cache force-write-backs.

Finally, concurrent workload execution can further complicate software-based logging in persistent memory. Even if persistent memory issues `clflush` or `clwb` instructions in each transaction, a context switch by the OS can occur before a `clwb` instruction is executed. Such context switches will interrupt the control flow of transactions/epochs and divert the program to other threads, which reintroduces the risk of prematurely overwriting the log records in NVRAM. Implementing per-thread log circular buffers can mitigate such risk. But doing so can introduce new persistent memory API and complicate recovery. Such risk and software implementation complexities further increase the cost of software-based logging.

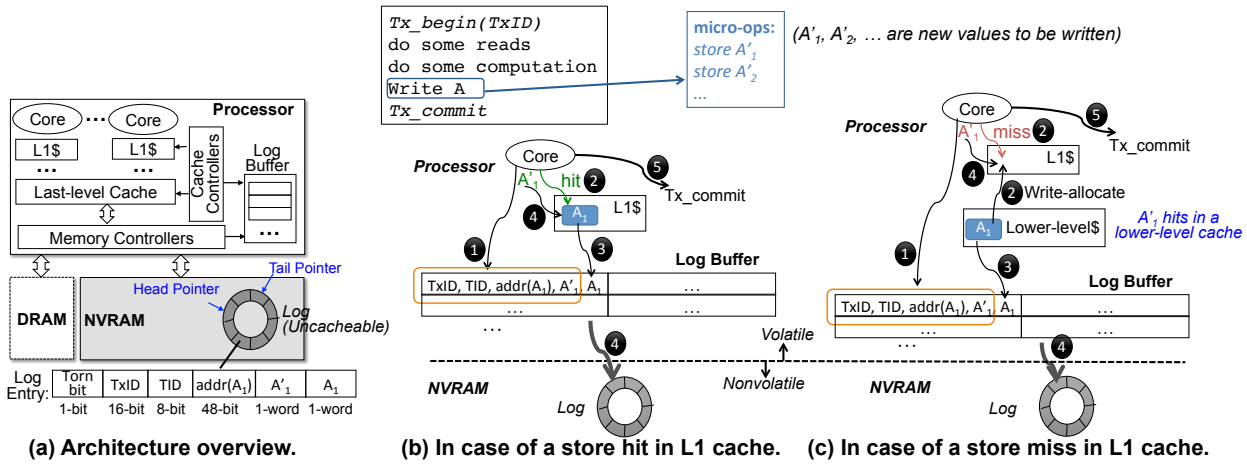

Figure 3: Overview of the proposed hardware-driven logging in persistent memory.

2.4 Challenges of Hardware-based persistent memory design

Many recent works strive to stretch persistent memory performance towards the native memory system [26, 41, 51]. Yet most of the works can (i) introduce nontrivial hardware overhead (e.g., by integrating NVRAM cache/buffers or substantial extra bookkeeping components in the processor) [51, 45], (ii) fall back to low-performance modes once the bookkeeping components or the NVRAM cache/buffer are saturated [51, 26], or (iii) prevent caches and memory controllers from coalescing and reordering persistent data writes [51] (further details discussed in Section 8).

3 Our Design

In order to address the aforementioned challenges, we propose a hardware-driven undo+redo logging scheme, which consists of a Hardware Logging and a progressive cache force-write-back (FWB) mechanisms. The Hardware Logging mechanism enables practical undo+redo logging in persistent memory systems, by leveraging the write-back write-allocate policy used in commodity processors. The FWB mechanism secures data persistence in addition to the Hardware logging. FWB significantly improves system performance by invoking force-write-backs of individual cache lines only when necessary, decoupling cache force-write-backs and transaction execution. With Hardware Logging and FWB, we also ensure data persistence in multithreading scenario. This section describes our design principles. Detailed implementation methods and the required software support will be described in Section 4.

3.1 Assumptions and Architecture Overview

Figure 3(a) depicts an overview of our processor and memory architecture. The figure also shows the log structure that is stored in NVRAM. We do not adopt any NVRAM components in the processor – the processor is pure volatile. We employ write-back caches that are available in commodity processors. The main memory can consist of DRAM and NVRAM, which are both deployed on the processor-memory bus with separate memory controllers or memory controller queues [48, 51]. However, this paper focuses on persistent data updates in the NVRAM.

Failure Model. The data in DRAM or caches is lost across system reboots, while data in NVRAM remains. Our design focuses on maintaining persistence of user-defined critical data stored in NVRAM. After failures, the system can recover critical data structures by replaying the log in NVRAM. The DRAM can be used to store data without persistence requirement, such as stacks and data transfer buffers [48, 51].

Persistent Memory Transactions. Similar to most prior persistent memory designs [48, 25], we employ persistent memory transactions as a software abstraction to enforce persistence of critical data. The writes that are included in persistent memory transactions are considered as those requiring persistence guarantee. Figure 1 illustrates simple code examples of a transaction implemented with traditional logging-based persistent memory transactions (Figure 1(a)) and our design (Figure 1(b), detailed software interface design is discussed in Section 4), respectively. The

transaction defines *object A* as a piece of critical data that needs persistence guarantee. Compared with traditional logging-based persistent memory transactions, our transactions eliminate logging functions, cache force-write-back instructions, and memory barrier instructions.

An Uncacheable Log in the NVRAM. We employ a single-consumer/single-producer Lamport circular buffer [27] as the log. It can be allocated and truncated by system software (§ 4). Our hardware mechanisms perform log appends. We adopt a circular buffer because it allows simultaneous appends and truncates without locking [27, 48]. As illustrated in Figure 3(a), each log record maintains undo and redo information of a single store (e.g., *store* A_1). In addition to the undo (A_1) and redo (A_1') values, each log record also adopts the following regions: a 16-bit transaction ID, an 8-bit thread ID, a 48-bit physical address of the data, and a *torn bit*. We also adopt a *torn bit* per log record to indicate the completion of each log update [48]. Torn bits have the same value for all writes in one pass over the log, and reverses when a log entry is overwritten. Thus, completely-written log records have the same torn bit value, while incomplete entries have mixed values [48]. The log is shared by all threads of an application to simplify system recovery. We also adopt software support (Section 4) to allocate and truncate the log. The log needs to accommodate all the write requests of undo+redo logging. But lifetime of this NVRAM region is not an issue as we will discuss in Section 7.

Log is typically used during system recovery, hence less likely reused during application execution. In addition, it is imperative to force log updates from caches to NVRAM in store-order by cache flushes and memory barriers. Therefore, we make the log *uncacheable*. This is inline with most prior works, which directly write log updates into write-combine buffer (WCB) available in commodity processors [48, 52] that coalesce multiple stores to the same cache line.

3.2 Hardware Logging

The goal of our Hardware Logging mechanism is to enable feasible undo+redo logging of persistent data, which can relax the order constraints on caching to the point where neither one of undo and redo logging can achieve. To this end, our logging mechanism leverages the information naturally available in commodity cache hierarchy, without the performance overhead of unnecessary data movement or executing logging, cache flush, and memory barrier instructions in CPU pipeline.

Commodity processors employ write-back write-allocate policy in the cache hierarchy [40]. On a write hit, the cache hierarchy will only update the cache line in the hitting level with old values remaining in lower levels; a dirty bit in the cache tag indicates such inconsistency across different levels of caches. On a write miss, the write-allocate (also called fetch-on-write) policy enforces that the cache hierarchy first loads the missing cache line from a lower-level, followed by a write hit operation.

In order to enable efficient steal/no-force support in persistent memory, we propose a hardware-driven logging mechanism by leveraging the write-back write-allocate scheme. In our design, a write in a transaction will trigger hardware-based log appends that record both redo and undo information in NVRAM (as shown in Figure 1(b)). We acquire the redo information from the write operation itself, which is currently in-flight. We acquire the undo information from the cache lines, where the current write will be placed. If the write request hits in L1 cache, we read the old value before overwriting the cache line and use that for the undo logging. If the write request misses in L1 cache, we acquire the undo information (the old value) from the cache line that will be write-allocated by the cache hierarchy. The log records, consisting of transaction ID, thread, the address of the write, and undo and redo cache line values, will be written out to the log location in the NVRAM based the head and tail pointers. These pointers are maintained in special registers as further described in Section 4.

Natural Ordering Guarantee on Data and Log Updates. Our design does not require write-order control mechanisms (e.g., cache flushes and memory barriers) to enforce the order between original data and corresponding log updates. Such order is naturally enforced by our architecture design: the uncacheable writing of the log can guarantee that log updates will arrive at NVRAM earlier than corresponding working data updates (which need to traverse through the cache hierarchy); no memory barrier is needed. We also provides an optional log buffer, which is similar to write-combining buffers in commodity processors, to coalesce the log updates. Yet, we configure the log buffer size such that we still enforce the guarantee without memory barriers. Section 4.3 and Section 6 discusses and evaluates the bound of the log buffer size, which depends on the size of the log in NVRAM and NVRAM bandwidth.

3.3 Progressive Cache Force-write-back

It appears that writes are persistent once their logs are written into NVRAM. In fact, our design allows us to commit a transaction once our hardware-driven logging of that transaction is completed. Yet this does not secure data persistence due to the circular buffering nature of the log in NVRAM (Section 2.1). Inserting force-write-back instructions (such as `clflush` and `clwb`) in transactions can impose substantial performance overhead (Section 2.1) and complex data persistence support in the multithreading scenario (Section 2.3).

In order to guarantee persistence in multithreaded applications yet eliminate the need for force-write-back instructions, we design a hardware-based, progressive force-write-back mechanism that only force certain original data blocks out of the caches when necessary. To this end, we introduce a *force-write-back* bit (*fwb*) into the cache tag of each cache line. With *fwb* bit and the *dirty* bit existing in traditional cache tags, we maintain a finite state machine of each cache block (§ 4.4). The *dirty* bit is maintained by traditional caching schemes: a cache line update will set the bit and a cache eviction (write-back) will reset it. The *fwb* is maintained by cache controllers that periodically scan all cache tags. Per every two scans, the cache controllers will set the *fwb* bit of the dirty cache lines in one scan, and force-write-back all the cache lines with $\{fwb, dirty\} = \{1, 1\}$ in the other scan. If the *dirty* bit value of any cache lines changes between the two scans, the cache line will not be force-written-back later on.

Our mechanism maintains force-write-backs in a hardware-controlled manner decoupled from software multithreading mechanisms. As such, our mechanism is robust to the interruption of software context switches. The frequency of this force-write-backs can vary. But force-write-backs need to be faster than the rate at which the log updates wrap around and overwrite the first entry. In fact, we can determine force-write-back frequency (associated with the scanning frequency) based on the log size and the NVRAM write bandwidth as we discuss in § 4.4. Our evaluation shows that the frequency can be every three million cycles (§ 6).

3.4 A Free Ride for Transaction Commits

Most previous designs require software or hardware memory barriers (and/or cache flushes) at transaction commits to enforce the order of writing log updates (or persistent data) into NVRAM across consecutive transactions [51, 30]. Instead, our design gives transaction commits a "free ride" – no actual instructions need to be executed in the CPU pipeline. In the log, we identify the commit of a transaction by the presence of the log record header of its immediately subsequent transaction. Our mechanisms also naturally enforce the order of intra- and inter-transaction log updates: we issue log updates in the order of writes to corresponding original data; we also write the log updates into NVRAM in the order as they are issued (the log buffer is a FIFO so does not affect such order). As such, log updates of subsequent transactions can only be written into NVRAM after the current log updates are written.

3.5 Putting It All Together

Figure 3(b) and (c) illustrate a high-level summary of how our hardware-driven logging works. Hardware treats all writes encompassed in transactions – e.g., `write A` in the transaction delimited by `tx_begin` and `tx_commit` in Figure 1(b) – as persistent writes. Those writes automatically trigger the logging mechanisms described in the above sections. The mechanisms work together as follows. Note that the log updates will be directly sent to the WCB or NVRAM, if the system does not adopt the log buffer.

Processor core sends the writes of data object A (a variable or other data structures), which consists of the new values of one or multiple cache lines $\{A'_1, A'_2, ...\}$ to L1 cache. Upon updating an L1 cache line (e.g., from old value A_1 to a new value A'_1):

1. Update the redo information of the cache line (❶).
   (a) If the update is the first cache line update of the data object A, the CPU core – which has the information of transaction ID and the address of A – will write a log record header into the log buffer.
   (b) Otherwise, the L1 cache controller writes the new value of the cache line (e.g., A'_1) into the log buffer.
2. Obtain the undo information of the cache line (❷). This step can be parallelized with Step-1.
   (a) If the cache line write request hits in L1 cache (Figure 3(b)), the L1 cache controller will extract the old value (e.g., A_1) directly from the cache line before writing. Rather than using an additional read instruction, the L1 cache controller reads the old value from the hitting line out of the L1 cache read port and write it into the log buffer in the Step-3. Note that the read port is typically not used when servicing write requests, so it would otherwise be idle.

```
void persistent_update( int threadid )
{
  tx_begin( threadid );
  // Persistent data updates
  write A[threadid];
  tx_commit();
}
// ...
int main()
{
  // Executes one persistent
  // transaction per thread
  for ( int i = 0; i < nthreads; i++  )
    thread t( persistent_update, i );
}
```

Figure 4: Pseudocode showing an example use for the `tx_begin` and `tx_commit` functions in which the thread ID is assigned to the transaction ID to perform one persistent transaction per thread.

(b) If the write request misses in L1 cache (Figure 3(c)), the cache hierarchy will naturally perform write-allocate on that cache line. The cache controller at a lower-level cache that owns that cache line will extract the old value (e.g., A_1). The cache controller will send the extracted old value to the log buffer in Step-3.

3. Update the undo information of the cache line: the cache controller writes the old value of the cache line (e.g., A_1) to the log buffer (❸).
4. The L1 cache controller can update the cache line in the L1 cache (❹). The cache line can be evicted by natural cache eviction policy without being subjected to data persistence constraints. In addition, our log buffer is small enough to guarantee that the log updates traverse through the log buffer faster than the cache line traversing through the cache hierarchy (§ 4.4). Therefore, this step can be performed without waiting for the corresponding log entries to arrive at NVRAM.
5. Memory controller evicts the log buffer entries to NVRAM in a first-in-first-out manner (❹). This step is independent from other steps.
6. Repeat Step-1-(b) through 5, if the data object A consists of multiple cache line writes. The log buffer will also coalesce the log updates of any writes to the same cache line.
7. After log updates of all the writes in the transaction are issued, the transaction can commit (❺).
8. Original data object updates can remain in caches until they are written back to NVRAM by a natural eviction or our progressive cache force-write-backs.

4 Implementation

In this section, we describe implementation details of our design and hardware overhead (the impact of the hardware overhead and NVRAM space consumption and lifetime/endurance will be discussed in Section 7).

4.1 Software Support

Our design employs software support for defining persistent memory transactions, allocating and truncating the circular log in NVRAM, and reserving a special character as the log header indicator.

Transaction Interface. We employ a pair of transaction functions, `tx_begin( txid )` and `tx_commit()`, define transactions which incorporate the writes in the program that need persistence support. We use the `txid` to provide the transaction ID information used by our hardware-based logging scheme. This ID is used to group writes from the same transaction together. Such transaction interface has been used by many previous persistent memory designs [51, 10]. Figure 4 shows a multithreaded example in pseudocode with these transaction functions in use.

System Library Functions that Maintain the Log. Our hardware mechanisms perform the log updates, whereas the log structure is maintained by system software. In particular, we employ a pair of system library functions, `log_create()` and `log_truncate()` (similar to the ones used in prior work [48]), to allocate and truncate log, respectively. The memory controller obtains such log maintenance information by reading special registers (§ 4.2) that

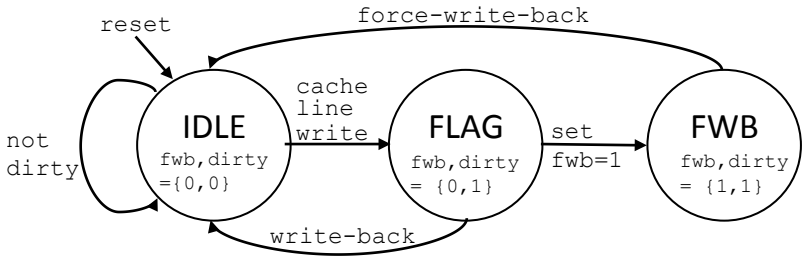

Figure 5: The finite-state machine in the cache controller for implementing the progressive cache write-back.

indicate the head and tail pointers of the log. Furthermore, a single transaction that exceeds the originally allocated log size can corrupt persistent data. We provide two options to prevent such overflows: 1) The `log_create()` function allocate a large-enough log by taking an hint of the maximum transaction size from the program interface (e.g., `#define MAX_TX_SIZE N`); 2) An additional library function `log_grow()` allocates additional log regions when the log is filled by an uncommitted transaction.

4.2 Special Registers

The `txid` argument provided through the `tx_begin()` function will be translated into an 8-bit unsigned integer – a *physical* transaction ID – that is stored in a special register in the processor. Because the transaction IDs are used to group writes of the same transactions, we can simply pick a not-in-use physical transaction ID to represent a newly received `txid`. An 8-bit length can accommodate 256 unique active persistent memory transactions at a time. A physical transaction ID can be reused after the transaction commits.

We also employs two 64-bit special registers to store the head and tail pointers of the log. The system library initializes the pointer values when allocating the log by `log_create()`. During log updates, the pointers can be updated by the memory controller and `log_truncate()` function. If `log_grow()` is used, we will employ additional registers to store the head and tail pointers of newly allocated log regions and an indicator of the active log region.

4.3 An Optional Volatile Log Buffer in the Processor

To improve the performance of log updates written to NVRAM, we provide an optional log buffer design (a volatile FIFO, similar to WCB) in the memory controller to buffer and coalesce log updates. Note that the log buffer is not required for ensuring data persistence but only for performance optimization. Section 6 evaluates system performance across various log buffer sizes.

Data persistence requires that a log record arrives at NVRAM before the corresponding cache line (original data) update. To ensure such ordering, we bound the log buffer size such that the time for an undo+redo log record to traverse through the log buffer is shorter than the minimum time that a cache line traverses through the cache hierarchy – the time that an L1 cache line directly gets evicted all the way until out of the last-level cache. As such, the maximum allowable log buffer size is 15 cache-line-size entries based on our processor configuration (Table 2); each log entry also employs two additional bits: one indicates whether the entry stores a record header; the other is a torn bit [48]. Further increasing the number of log buffer entries can improve the performance of log updates until we hit the write bandwidth provided by NVRAM as we demonstrated in Section 6, but will risk losing data persistence.

4.4 Cache Modifications

To implement our progressive cache write-back scheme, we add one `fwb` to the tag of each cache line, along with the *dirty* bit existing in conventional cache implementations. We also maintain three states – IDLE, FLAG, and FWB – for each cache block using these tag bits.

Cache Block State Transition. Figure 5 shows a finite-state machine that we implement in the cache controller of each level. When an application starts to execute, cache controllers initialize (reset) each cache line to IDLE state by setting `fwb` bit to be 0 (conventional cache implementations will also initialize *dirty* and *valid* bits to be 0s). During application execution, existing cache controllers implementations will set the *dirty* and *valid* bits to 1 whenever a cache line is written and set the *dirty* bit back to 0 after the cache line is written back to the lower level. To implement our state machine, the cache controllers will also periodically scan the tag of each cache line and act as following.

Mechanism	Logic Type	Size
Transaction ID register	flip-flops	1 Byte
Log head pointer register	flip-flops	8 Bytes
Log tail pointer register	flip-flops	8 Bytes
Log buffer (optional)	SRAM	964 Bytes
Fwb tag bit	SRAM	768 Bytes

Table 1: Summary of major hardware overhead.

- A cache line with $\{fwb, dirty\} = \{0,0\}$ is in IDLE state; the cache controller will do nothing to those cache lines;
- A cache line with $\{fwb, dirty\} = \{0,1\}$ is in FLAG state; the cache controller will set the `fwb` to 1. This indicates that the cache line needs to be force-write-back during the next scanning iteration, in case it still remains in the cache.
- A cache line with $\{fwb, dirty\} = \{1,1\}$ is in FWB state; the cache controller will force-write-back this line. After the force-write-back, the cache controller changes the line back to IDLE state by resetting $\{fwb, dirty\} = \{0,0\}$.
- If a cache line is evicted out of the cache at any time point, the cache controller will reset that cache line to IDLE.

Determining the Cache Force-Write-Back Frequency. The tag scanning frequency determines the frequency of our cache force write-back operations. The force write-back (i.e., the scanning) needs to be performed as frequent as to ensure that the original data is written back to NVRAM before the corresponding log records are overwritten by newer updates. As a result, the more frequent the write requests are serviced, the more frequent the log will be overwritten; the larger the log, the less frequent the log will be overwritten. As such, the scanning frequency can be determined by the maximum log update frequency (bounded by NVRAM write bandwidth because applications cannot write to the NVRAM faster than its bandwidth) and log size (see the sensitivity study in § 6).

4.5 Summary of Hardware Overhead

Table 1 amalgamates the hardware overhead of our implementation mechanisms in the processor. Note that these values may vary depending on the native processor and ISA. Our implementation assumes a 64-bit machine which is why the circular log head and tail pointers are 8 bytes. Only half of these bytes are required in a 32-bit machine. The size of the log buffer varies based on the size of the cache line. The size of the overhead needed for the `fwb` state varies on the total number of cache lines at all levels of cache. Our value in the table was computed based on the specifications of all our system caches described in § 5.

Also note that these are the major logic components that make up storage on-chip. Additional gates for logic operations are also necessary to implement the design mechanisms described in previous section. However, the simplicity of the design means the gates used will be primarily small and medium-sized gates, on the same complexity level as a multiplexer or decoder.

4.6 Recovery

We outline the steps of recovering the critical in-memory data in systems that adopt our design.

Step 1: Following a power failure, the first step is to obtain the head and tail pointers of the log in NVRAM. These pointers are part of the log structure. They allow systems to correctly order the log entries. We employ only one circular buffer as the log of all transactions of all threads. Therefore, we use the reference to look up these pointers.

Step 2: System recovery handler fetches log entries from NVRAM and use the address, old value, and new value fields to generate writes to NVRAM to the addresses specified. We identify which writes did not commit by going backwards from the tail pointer. Each log entry that does not match its value in NVRAM is considered a non-committed entry. The address stored with each entry corresponds to the address of the data member in the original persistent data structure. Aside from the head and tail pointers, we use the *torn bit* to correctly order these writes [48].

Step 3: The generated writes go directly to NVRAM and do not use the cache hierarchy. We use volatile caches so their state is reset and all these generated writes on recovery are persistent, so they can bypass the caches without issue.

Step 4: With each persistent write generated from the circular log, update the head and tail pointers accordingly. After all the updates from the log are redone (or undone), the head and tail pointers of the log should point to the same location and the entries need to be invalidated.

Processor	Similar to Intel Core i7 / 22 nm
Cores	4 cores, 2.5GHz, 2 threads/core
IL1 Cache	32KB, 8-way set-associative, 64B cache lines, 1.6ns latency,
DL1 Cache	32KB, 8-way set-associative, 64B cache lines, 1.6ns latency,
L2 Cache	8MB, 16-way set-associative, 64B cache lines, 4.4ns latency
Memory Controller	64-/64-entry read/write queues
NVRAM DIMM	8GB, 8 banks, 2KB row 36ns row-buffer hit, 100/300ns read/write row-buffer conflict [28].
Power and Energy	Processor: 149W (peak) NVRAM: row buffer read (write): 0.93 (1.02) pJ/bit, array read (write): 2.47 (16.82) pJ/bit [28]

Table 2: Processor and memory configurations.

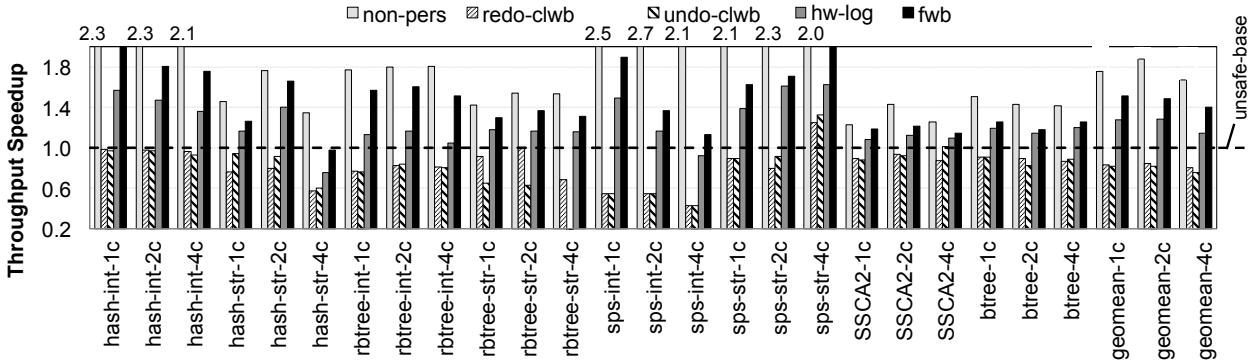

Figure 6: Transaction throughput speedup (higher is better)), normalized to *unsafe-base* (the dashed line)).

5 Experimental Setup

We evaluate our design by implementing our mechanisms in McSimA+ [3], a Pin-based [33] cycle-level multi-core simulator. The simulator is configured to model a multi-core out-of-order processor and NVRAM DIMM described in Table 2. We feed the performance simulation results into McPAT [29] to estimate processor dynamic energy consumption. We also modified McPAT processor configuration to model our hardware modifications. We adopt phase-change memory parameters in the NVRAM DIMM [28]. Yet, because all of our performance numbers shown in § 6 are relative, the same observations should be valid for various NVRAM latencies and access energy. Our work focuses on improving persistent memory access so we do not evaluate DRAM access in our experiments.

We evaluate both microbenchmarks and real workloads in our experiments. The microbenchmarks repeatedly update persistent memory storing various data structures, including hash table, red-black tree, array, B+tree, and graph. These are data structures widely used in storage systems [10]. Table 3 describes the details these benchmarks. Our experiments use several versions of each benchmark and vary the data type between integers and strings within them. Data structures with integer elements pack less data (smaller than a cache line) per element, whereas those with strings have multiple cache lines per element and allows us to explore more complex structures used in real-world applications. In our microbenchmarks, each transaction performs an insert, delete, or swap operation. The number of transactions is proportional to the data structure size, listed as "memory footprint" in Table 3. We compile these benchmarks in native x86 and run on the McSimA+ simulator. We evaluate both single-threaded and multi-threaded behavior of each benchmark. In addition, we evaluate a persistent version of memcached [34], which is a real-world workload studied in prior works [10].

Name	Memory Footprint	Description
Hash [10]	256 MB	Searches for a value in an open-chain hash table. Insert if absent, remove if found.
RBTree [51]	256 MB	Searches for a value in a red-black tree. Insert if absent, remove if found
SPS [51]	1 GB	Random swaps between entries in a 1 GB vector of values.
BTree [6]	256 MB	Searches for a value in a B+ tree. Insert if absent, remove if found
SSCA2 [5]	16 MB	A transactional implementation of SSCA 2.2, performing several analyses of large, scale-free graph.

Table 3: A list of evaluated microbenchmarks.

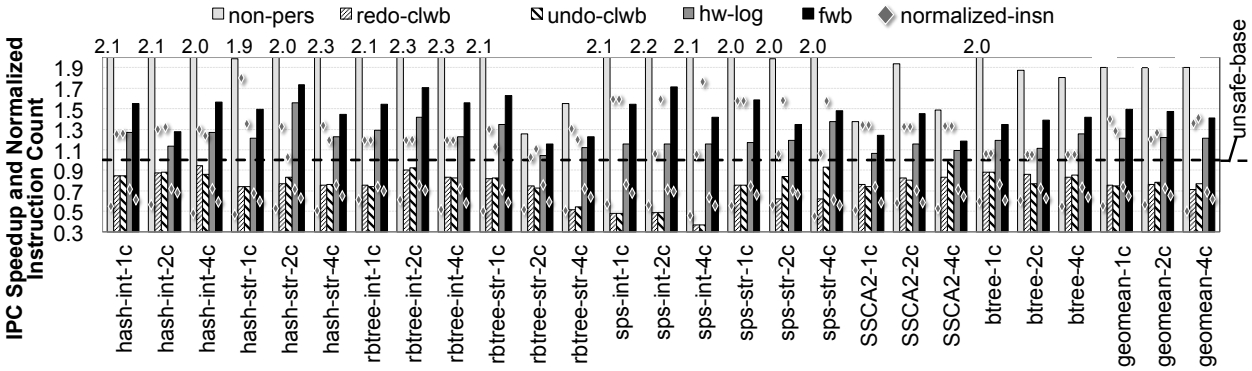

Figure 7: IPC speedup (higher is better) and instruction count (*normalized-insn*, lower is better), normalized to *unsafe-base* (the dashed line).

6 Results

We evaluate our design in terms of transaction throughput, instruction per cycle (IPC), instruction count, NVRAM traffic, and system dynamic energy. Our experiments compare among the following cases.

- **No-pers** – This employs NVRAM as a working memory without data persistence guarantee or logging. This configuration can yield an ideal yet unachievable performance for persistent memory systems [51].
- **Unsafe-base** – Software-based logging that does not perform forced cache flushes and write-back. As such, it does not guarantee data persistence. Note that the dashed lines in our results figures only show the best case achieved by employing either redo or undo logging for that benchmark.
- **Redo-clwb** and **Undo-clwb** – Software-based redo and undo logging, respectively. These invoke the `clwb` instruction to force cache write-backs.
- **HW-log** – This performs undo+redo logging with our hardware logging design, but uses the `clwb` instruction to force cache write-backs.
- **FWB** – Our hardware-driven undo+redo logging design, including both our hardware logging and FWB mechanisms.

6.1 Microbenchmark Results

We make the following major observations of our microbenchmark experiments and analyze the results. Although we evaluated benchmark configurations from single to eight threads, we only show results with two (*-1c*), four (*-2c*), and eight (*-4c*) threads due to space limit.

System Performance and Energy Consumption. Figure 6 and Figure 8 compare the transaction throughput and

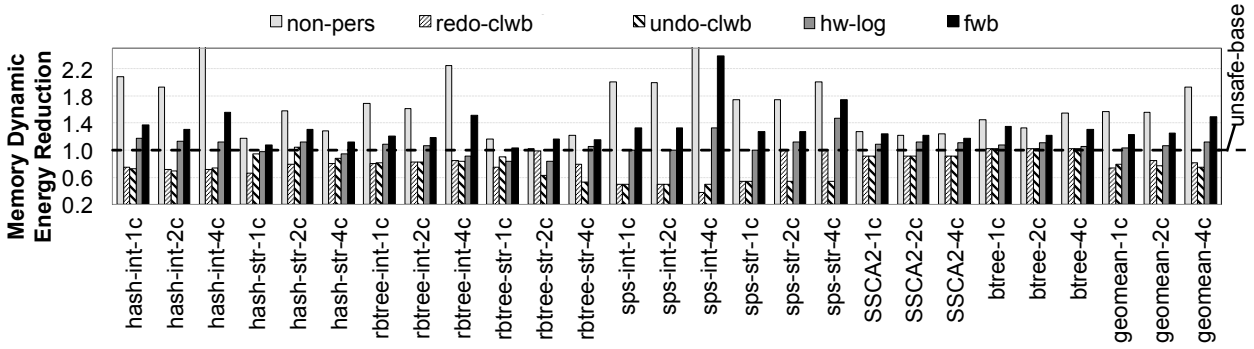

Figure 8: Dynamic energy reduction (higher is better), normalized to *unsafe-base* (the dashed line).

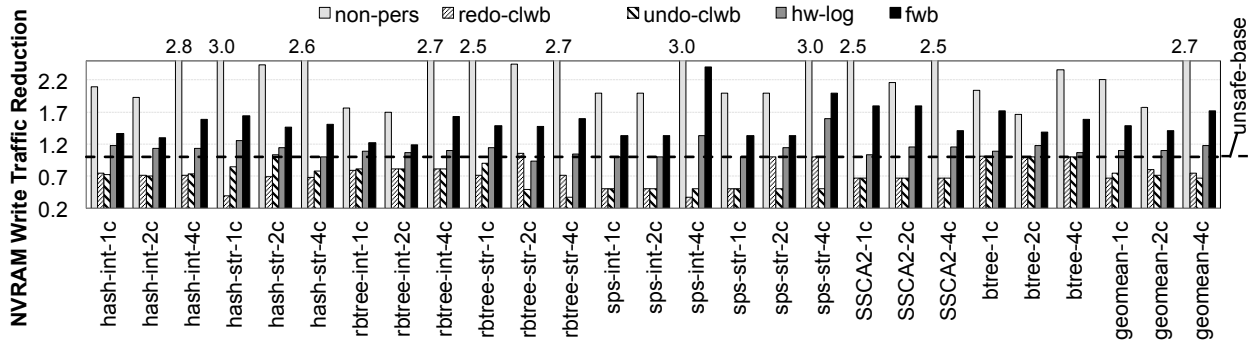

Figure 9: Memory write traffic reduction (the higher the better), normalized to *unsafe-base* (the dashed line).

memory dynamic energy of designs. We observe that processor dynamic energy is not significantly altered by various configurations. Therefore, we only show memory dynamic energy in the figure. The figures illustrate that *hw-log* alone already significantly improves system throughput and dynamic energy consumption, compared with software-based logging schemes. Note that our design supports undo+redo logging, while the evaluated software-based logging mechanisms only support either undo or redo logging, not both. *Fwb* yields higher throughput and lower energy consumption – overall, it improves throughput by 1.86× with two threads and 1.75× with eight threads, compared with the better of *redo-clwb* and *undo-clwb*. SSCA2 and BTree benchmarks generate less throughput and energy improvement over software-base logging. This is because SSCA2 and BTree employ more complex data structures, where the overhead of manipulating the data structures outweigh that of the log structures.

The figures also show that *unsafe-base*, *redo-clwb*, and *undo-clwb* significantly degrade throughput by up to 59% and impose up to 62% memory energy overhead compared with the ideal case *non-pers*. Our design brings system throughput back up. *Fwb* achieves 86% throughput, with only 6% processor-memory system and 20% memory dynamic energy overhead, respectively. Furthermore, the performance and energy benefits remain with *hw-log* and *fwb* compared with software-based logging, when we increase the number of threads.

IPC and Instruction Count. We also study the IPC and the number of executed instructions during transaction execution (benchmark initialization is not counted), shown in Figure 7. Overall, *hw-log* and *fwb* significantly improves IPC compared with software-based logging. This appears promising, given our hardware-driven logging executes much fewer number of instructions as shown in the figure. Compared with *non-pers*, software-based logging can impose up to 2.5× of instructions executed. Our design *fwb* only imposes 30% instruction overhead.

Performance Sensitivity to Log Buffer Size. As we discussed in § 4.3, the log buffer size is bounded by the data persistence requirement: the log updates need to arrive at NVRAM before the corresponding original data updates. This bound is $\leq$15 entries based on our processor configuration. Indeed, larger log buffers better improve throughput as we studied using *Hash* benchmark (Figure 10(a)): an 8-entry log buffer improves system throughput by 10%; our implementation with a 15-entry log buffer improves throughput by 18%. Further increasing log buffer size – although will no longer guarantee data persistence – can continue to improve system throughput until hitting the NVRAM write

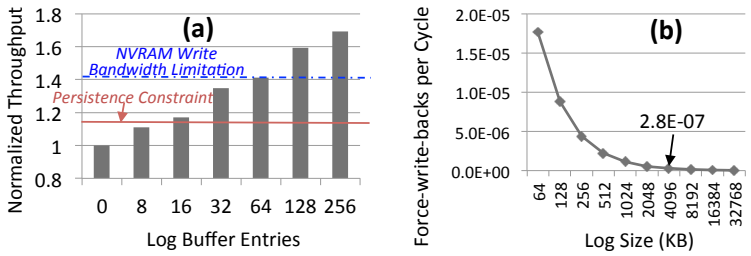

Figure 10: Sensitivity studies of (a) system throughput with various log buffer sizes and (b) cache force write-back frequency with various NVRAM log sizes.

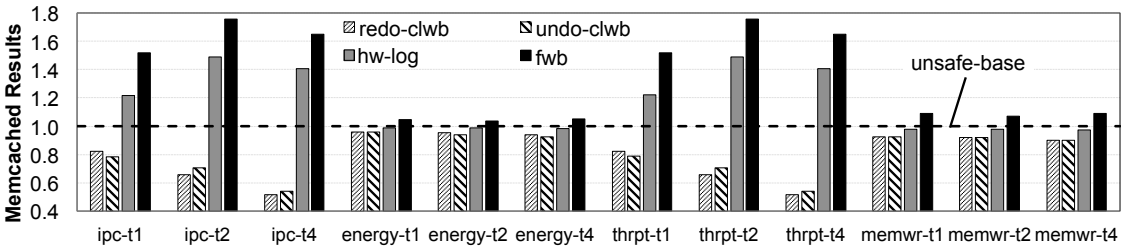

Figure 11: Memcached results, including IPC, dynamic memory energy consumption, transaction throughput, and NVRAM write traffic, normalized to *unsafe-base* (the dashed line).

bandwidth limitation (64 entries based on our NVRAM configuration). Note that the system throughput results with 128 and 256 entries are generated assuming infinite NVRAM write bandwidth.

The Relationship Between Force Write-back Frequency and Log Size. As we discussed in § 4.4, the force write-back frequency is determined by the NVRAM write bandwidth and log size. With given NVRAM write bandwidth, we studied the relationship between the required force write-back frequency and the log size. As shown in Figure 10(b), we only need to perform force-write-backs every three million cycles if we have a 4MB log.

6.2 Memcached Results

With memcached, we observe even more promising results than the microbenchmarks (Figure 11): at four threads (*-4t*), *fwb* throughput achieves 3.3× of the best case of software-based logging. This is also within 89% of *non-pers* throughput at the same number of threads. In addition, the memcached workload does not show diminishing returns of *fwb* with increasing number of threads.

7 Discussion

Impact on NVRAM Capacity Utilization. Storing Undo+redo log can consume large NVRAM space than either undo or redo logging. But we employ a fixed-size circular buffer as the log, instead of requiring to double any of previous undo or redo log implementations. The log size can trade off with the frequency of our cache force-write-backs (Section 4). With the software support discussed in Section 4.1, we also allow users to determine the size of the log. Our force-write-back mechanism will adjust the frequency accordingly to ensure data persistence.

Impact on Lifetime of the NVRAM Main Memory. Lifetime of the log region is not an issue: say a log has 64K entries (4MB) and NVRAM (assuming phase-change memory) write latency is 200ns, each entry will be overwritten once every 64K * 200 ns. If NVRAM endurance is 10^8, a cell – even statically allocated to the log – will take 15 days to wear out, which is more than enough time for conventional NVRAM wear-leveling schemes to kick in [54, 43, 44]. In addition, our scheme has two impacts on overall NVRAM lifetime: logging can result in write amplification, whereas we improve NVRAM lifetime by allowing caching to coalesce writes. The overall impact is likely slightly negative. But, as we note, wear-leveling will kick in before any damage occurs.

Impact on DRAM+NVRAM Hybrid Persistent Memory Systems. In DRAM+NVRAM hybrid memory systems, the persistent memory systems also need to flush the data cached/buffered in the DRAM into NVRAM when transactions commit, using `memcpy()` [4] or hardware-based data copy mechanisms [45]. Such memory copying can substantially slow down transaction committing process, degrading transaction throughput [4]. Furthermore, the data copying involves both reads from DRAM and writes to the NVRAM, substantially increasing memory traffic. Our undo+redo logging scheme can generally be used to relax the write-order control of various cache components in persistent memory systems, including the DRAM cache. As such, our design can also be used to improve persistent memory systems that adopt DRAM+NVRAM hybrid memory. We will explore such benefits in our future work.

8 Related Work

Logging in Persistent Memory. Most previous persistent memory logging schemes are software based. Several studies [48, 10, 25] exploit software-based logging in persistent memory transactions. Most of the studies employ redo logging [48, 10], which typically offers better system performance than undo logging due to the less memory traffic and relaxed write-backs of committed data updates. Other works, such as NOVA [50] and PMFS [15], develop an entire file system to improve performance persistent memory systems. They exploit file system structures and techniques to embed logging at the OS-level to guarantee data persistence. These software approaches are viable, but increase the exposure and burden of data persistence to the programmer. Furthermore, most software-based schemes cannot exploit detailed micro-architectural information to manipulate individual data access in the processor. As such, they can impose substantial performance and energy overhead due to the under-utilization of the underlying persistent memory hardware. Our work adopts a different approach from most prior work. We demonstrate that updating the log by hardware can best utilize the microarchitecture information to substantially reduce the overhead of logging, such that even combining undo and redo logging is more than feasible.

Several recent studies proposed hardware support for log-based persistent memory design. Lu *et al.* proposed custom hardware logging mechanisms and multi-versioning caches to reduce intra- and inter-transaction dependencies [32]. Kolli *et al.* proposed a delegated persist ordering [26] that substantially relaxes persistence ordering constraints by leveraging hardware support and cache coherence. However, the design relies on snoop-based coherence and a dedicated persistent memory controller. We provide a flexible design that directly leverages the information already existing in commodity cache hierarchy.

Non-Logging Persistent Memory Designs. Due to the overhead and inefficiency of traditional logging schemes, many previous works strive to explore persistent memory designs without logging. One promising scheme is copy-on-write [47, 11]. The scheme avoids direct overwritten of original data by allocating an alternative memory location to store the updates. However, copy-on-write cannot provide the benefits of logging as discussed in § 1. Other studies exploit hardware support for data persistence [51, 45]. These mechanisms cannot provide the benefits of logging either. Moreover, most prior mechanisms require expensive hardware modifications such as on-chip nonvolatile transaction caches/buffers [51, 45, 49, 26, 16, 14]. Our design only employs lightweight hardware modifications on existing processor designs without the need for expensive nonvolatile on-chip transaction buffering components. Other related work focuses on efficient persist barriers [23], which similarly reduces cache flushing using hardware extensions in the cache controller. However, it still relies on programmer-inserted memory barriers and software-based redo or undo logging. Furthermore, these hardware approaches play a passive role, merely *assisting* the software in handling data persistence rather than *driving* it. Our design takes a bolder approach in supporting data persistence by directly handling logging and persistence requirements.

9 Conclusions

We proposed a hardware-driven logging scheme, which allows persistent memory systems to provide undo+redo logging at low performance, hardware, and software costs. The steal/no-force support enabled by undo+redo logging allows the caches to perform natural cache line updates and write-backs without employing nonvolatile buffers or caches in the processor. These mechanisms coalesce into a complexity-effective design that improves over traditional software-based logging for persistent memory. Our evaluation shows that our design makes significant improvements over the baselines in terms of performance, memory traffic, and energy. Despite our implementation being on-par with the baselines for two of benchmarks, our design secure data persistence whereas the baselines (even with `clwb`) do

not. We showed through significant improvements on various persistent memory benchmarks that the benefits of our system outweigh the hardware cost.

References

[1] ARM, ARMv8-a architecture evolution, 2016.

[2] Intel, a collection of linux persistent memory programming examples, https://github.com/pmem/linux-examples.

[3] J. H. Ahn, S. Li, O. Seongil, and N. P. Jouppi. Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *ISPASS*, pages 74–85. IEEE, 2013.

[4] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *Proc. VLDB Endow.*, 10(4):337–348, Nov. 2016.

[5] D. A. Bader and K. Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing*, pages 465–476, 2005.

[6] T. Bingmann. STX B+ Tree, Sept. 2008, http://panthema.net/2007/stx-btree.

[7] C. Cagli. Characterization and modelling of electrode impact in HfO2-based RRAM. In *Proceedings of the Memory Workshop*, 2012.

[8] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 387–400, 2012.

[9] N. Christiansen. Storage class memory support in the Windows OS. In *SNIA NVM Summit*, 2016.

[10] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–118, 2011.

[11] 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*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[12] K. Deierling. Persistent memory over fabric. In *SNIA NVM Summit*, 2016.

[13] C. Diaconu. Microsoft SQL Hekaton c towards large scale use of PM for in-memory databases. In *SNIA NVM Summit*, 2016.

[14] 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, 2016.

[15] 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, 2014.

[16] E. Giles, K. Doshi, and P. Varman. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–14, 2015.

[17] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[18] S. Haria, S. Nalli, M. M. Swift, M. D. Hill, H. Volos, and K. Keeton. Hands-off persistence system (HOPS). In *Non-volatile Memories Workshop*, 2017.

[19] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[20] Intel. Intel persistent memory programming. http://pmem.io.

[21] Intel. Intel architecture instruction set extensions programming reference, 2016. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[22] Intel and Micron. Intel and Micron produce breakthrough memory technology, 2015. http://newsroom.intel.com/community/intel_newsroom/.

[23] 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, 2015.

[24] S. Kannan, A. Gavrilovska, and K. Schwan. Reducing the cost of persistence for nonvolatile heaps in end user devices. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 1–12, 2014.

[25] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–12, 2016.

[26] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated persist ordering. In *Proceedings of the 49th International Symposium on Microarchitecture*, pages 1–13, 2016.

[27] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, Mar. 1977.

[28] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *International Symposium on Computer Architecture*, pages 2–13, 2009.

[29] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.

[30] 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. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 455–470, 2014.

[31] Y. Lu, J. Shu, and L. Sun. Blurred persistence in transactional persistent memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–13, 2015.

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

[33] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005.

[34] Memcached. Memcached, a distributed memory object caching system, 2014.

[35] 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.

[36] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: log-based transactional memory. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, pages 1–12, 2006.

[37] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, N. Binkert, and P. Ranganathan. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the ACM Conference on Timely Results in Operating Systems*, pages 1–17, 2013.

[38] J. Moyer. Persistent memory in Linux. In *SNIA NVM Summit*, 2016.

[39] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An analysis of persistent memory use with WHISPER. In *Proceedings of th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–14, 2017.

[40] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[41] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceedings of the International Symposium on Computer Architecture*, pages 1–12, 2014.

[42] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the NVRAM era. *Proceedings of the VLDB Endowment*, 7(2), 2013.

[43] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, 2009.

[44] 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.

[45] 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 48th International Symposium on Microarchitecture (MICRO-48)*, pages 1–13, 2015.

[46] V. Sousa. Phase change materials engineering for RESET current reduction. In *Proceedings of the Memory Workshop*, 2012.

[47] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[48] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[49] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014.

[50] 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.

[51] 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 International Symposium on Microarchitecture (MICRO)*, pages 421–432, 2013.

[52] J. Zhao, O. Mutlu, and Y. Xie. FIRM: Fair and high-performance memory control for persistent memory systems. In *Proceedings of the 47th International Symposium on Microarchitecture (MICRO-47)*, 2014.

[53] W. Zhao, E. Belhaire, Q. Mistral, C. Chappert, V. Javerliac, B. Dieny, and E. Nicolle. Macro-model of spin-transfer torque based magnetic tunnel junction device for hybrid magnetic-CMOS design. In *Behavioral Modeling and Simulation Workshop, Proceedings of the 2006 IEEE International*, pages 40–43, Sept 2006.

[54] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 14–23, New York, NY, USA, 2009. ACM.