Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems

Matheus Almeida Ogleari*, Ethan L. Miller*,†, Jishen Zhao*,‡

*University of California, Santa Cruz †Pure Storage ‡University of California, San Diego
*{mogleari,elm,jishen.zhao}@ucsc.edu ‡jzhao@ucsd.edu

***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 of storage with the fast load/store interface of memory. 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 through write coalescing and reordering. We identify that such write-order control can be relaxed by employing undo+redo logging for data in persistent memory systems. However, traditional software logging mechanisms are expensive to adopt in persistent memory due to performance and energy overheads. Previously proposed hardware logging schemes are inefficient and do not fully address the issues in software.**

To address these challenges, we propose a hardware undo+redo logging scheme which maintains data persistence by leveraging the write-back, write-allocate policies used in commodity caches. Furthermore, we develop a cache force-write-back mechanism in hardware to significantly reduce the performance and energy overheads from forcing data into persistent memory. Our evaluation across persistent memory microbenchmarks and real workloads demonstrates that our design significantly improves system throughput and reduces both dynamic energy and memory traffic. It also provides strong consistency guarantees compared to software approaches.

I. Introduction

Persistent memory presents a new tier of data storage components for future computer systems. By attaching Non-Volatile Random-Access Memories (NVRAMs) [1], [2], [3], [4] to the memory bus, persistent memory unifies memory and storage systems. NVRAM offers the fast load/store access of memory with the data recoverability of storage in a single device. Consequently, hardware and software vendors recently began adopting persistent memory techniques in their next-generation designs. Examples include Intel's ISA and programming library support for persistent memory [5], ARM's new cache write-back instruction [6], Microsoft's storage class memory support in Windows OS and in-memory databases [7], [8], Red Hat's persistent memory support in the Linux kernel [9], and Mellanox's persistent memory support over fabric [10].

Though promising, persistent memory fundamentally changes current memory and storage system design assumptions. Reaping its full potential is challenging. Previous persistent memory designs introduce large performance and energy overheads compared to native memory systems, without enforcing consistency [11], [12], [13]. A key reason is the write-order control used to enforce data persistence. Typical processors delay, combine, and reorder writes in caches and memory controllers to optimize system performance [14], [15], [16], [13]. However, most previous persistent memory designs employ memory barriers and forced cache write-backs (or cache flushes) to enforce the order of persistent data arriving at NVRAM. This write-order control is sub-optimal for performance and do not consider natural caching and memory scheduling mechanisms.

Several recent studies strive to relax write-order control in persistent memory systems [15], [16], [13]. However, these studies either impose substantial hardware overhead by adding NVRAM caches in the processor [13] or fall back to low-performance modes once certain bookkeeping resources in the processor are saturated [15].

Our **goal** in this paper is to design a high-performance persistent memory system without (i) an NVRAM cache or 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 combined undo+redo logging scheme in hardware.

Undo+redo logging stores both old (undo) and new (redo) values in the log during a persistent data update. It offers a key benefit: relaxing the write-order constraints on caching persistent data in the processor. In our paper, we show that undo+redo logging can ensure data persistence without needing strict write-order control. As a result, the caches and memory controllers can reorder the writes like in traditional non-persistent memory systems (discussed in Section II-B).

Previous persistent memory systems typically implement *either* undo or redo logging in software. However, high-performance software undo+redo logging in persistent memory is unfeasible due to inefficiencies. First, software logging generates extra instructions in software, competing for limited hardware resources in the pipeline with other critical workload operations. Undo+redo logging can double the number of extra instructions over undo or redo logging

alone. Second, logging introduces extra memory traffic in addition to working data access [13]. Undo+redo logging would impose more than double extra memory traffic in software. Third, the hardware states of caches are invisible to software. As a result, software undo+redo logging, an idea borrowed from database mechanisms designed to coordinate with software-managed caches, can only conservatively coordinate with hardware caches. Finally, with multithreaded workloads, context switches by the operating system (OS) can interrupt the logging and persistent data updates. This can risk the data consistency guarantee in multithreaded environment (Section II-C discusses this further).

Several prior works investigated hardware undo or redo logging separately [17], [15] (Section VII). These designs have similar challenges such as hardware and energy overheads [17], and slowdown due to saturated hardware bookkeeping resources in the processor [15]. Supporting both undo and redo logging can further exacerbate the issues. Additionally, hardware logging mechanisms can eliminate the logging instructions in the pipeline, but the extra memory traffic generated from the log still exists.

To address these challenges, we propose a combined undo+redo logging scheme in hardware that allows persistent memory systems to relax the write-order control by leveraging existing caching policies. Our design consists of two mechanisms. First, a **Hardware Logging (HWL)** mechanism performs undo+redo logging by leveraging write-back write-allocate caching policies [14] commonly used in processors. Our HWL design causes a persistent data update to automatically trigger logging for that data. 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 utilizes the cache block writes to update the log with word-size values. Second, we propose a cache **Force Write-Back (FWB)** mechanism to force write-backs of cached persistent working data in a much lower, yet more efficient frequency than in software models. This frequency depends only on the allocated log size and NVRAM write bandwidth, thus decoupling cache force write-backs from transaction execution. We summarize the contributions of this paper as following:

- This is the first paper to exploit the combination of undo+redo logging to relax ordering constraints on caches and memory controllers in persistent memory systems. Our design relaxes the ordering constraints in a way that undo logging, redo logging, or copy-on-write alone cannot.
- We enable efficient undo+redo logging for persistent memory systems in hardware, which imposes substantially more challenges than implementing either undo- or redo- logging alone.
- We develop a hardware-controlled cache force write-back mechanism, which significantly reduces the performance overhead of force write-backs by efficiently tuning the write-back frequency.
- We implement our design through lightweight software support and processor modifications.

II. Background and Motivation

Persistent memory is fundamentally different from traditional DRAM main memory or their NVRAM replacement, due to its persistence (i.e., crash consistency) property inherited from storage systems. Persistent memory needs to ensure the integrity of in-memory data despite system crashes and power loss [18], [19], [20], [21], [16], [22], [23], [24], [25], [26], [27]. The persistence property is not guaranteed by memory consistency in traditional memory systems. Memory consistency ensures a consistent global view of processor caches and main memory, while persistent memory needs to ensure that the data in the NVRAM main memory is standalone consistent [16], [19], [22].

A. Persistent Memory Write-order Control

To maintain data persistence, most persistent memory designs employ transactions to update persistent data and carefully control the order of writes arriving in NVRAM [16], [19], [28]. A transaction (e.g., the code example in 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 force cache write-backs (e.g., `clflush`, `clwb`, and `dccvap`) and use memory barrier instructions (e.g., `mfence` and `sfence`) throughout transactions to enforce write-order control [28], [15], [13], [19], [29].

Recent works strived to improve persistent memory performance towards a native non-persistent system [15], [16], [13]. In general, whether employing logging in persistent memory or not, most face similar problems. (i) They introduce nontrivial hardware overhead (e.g., by integrating NVRAM cache/buffers or substantial extra bookkeeping components in the processor) [13], [30]. (ii) They fall back to low-performance modes once the bookkeeping components or the NVRAM cache/buffer are saturated [13], [15]. (iii) They inhibit caches from coalescing and reordering persistent data writes [13] (details discussed in Section VII).

Forced cache write-backs ensure that cached data updates made by completed (i.e., committed) transactions are written to NVRAM. This ensures NVRAM is in a persistent state with the latest data updates. Memory barriers stall subsequent data updates until the previous updates by the transaction complete. However, this write-order control prevents caches from optimizing system performance via coalescing and reordering writes. The forced cache write-backs and memory barriers can also block or interfere with subsequent read and write requests that share the memory bus. This happens regardless of whether these requests are independent from the persistent data access or not [26], [31].

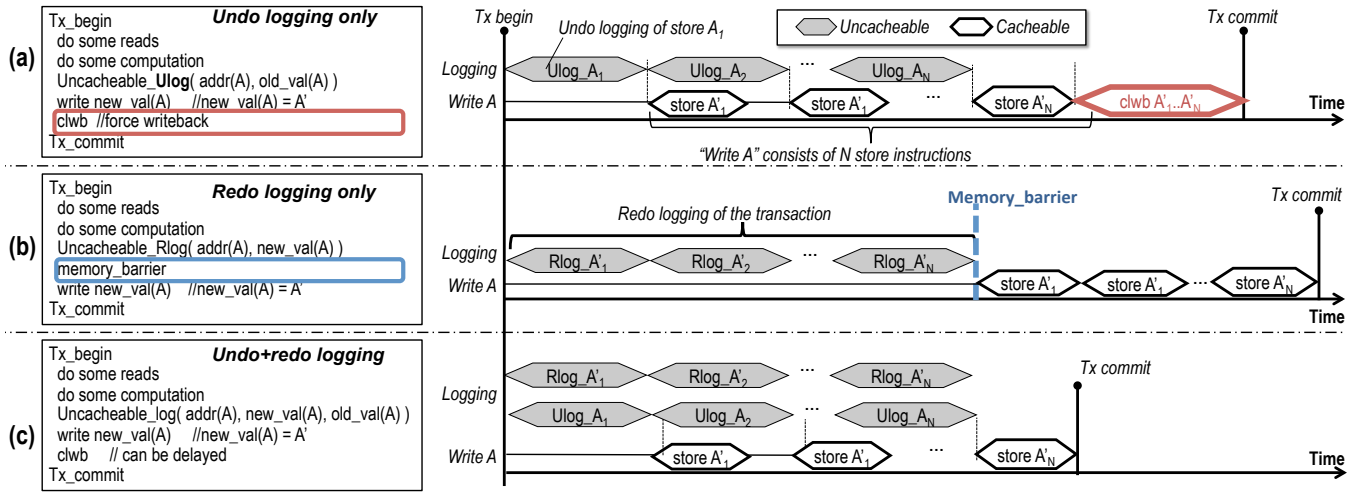


Figure 1. Comparison of executing a transaction in persistent memory with (a) undo logging, (b) redo logging, and (c) both undo and redo logging.

B. Why Undo+Redo Logging

While prior persistent memory designs only employ either undo or redo logging to maintain data persistence, we observe that using both can substantially relax the aforementioned write-order control placed on caches.

Logging in persistent memory. Logging is widely used in persistent memory designs [19], [29], [22], [15]. In addition to working data updates, persistent memory systems can maintain copies of the changes in the log. Previous designs typically employ either undo or redo logging. Figure 1(a) shows that an undo log records old versions of data before the transaction changes the value. If the system fails during an active transaction, the system can roll back to the state before the transaction by replaying the undo log. Figure 1(b) illustrates an example of a persistent transaction that uses redo logging. The redo log records new versions of data. After system failures, replaying the redo log recovers the persistent data with the latest changes tracked by the redo log. In persistent memory systems, logs are typically *uncacheable* because they are meant to be accessed only during the recovery. Thus, they are not reused during application execution. They must also arrive in NVRAM in order, which is guaranteed through bypassing the caches.

Benefits of undo+redo logging. Combining undo and redo logging (undo+redo) is widely used in disk-based database management systems (DBMSs) [32]. Yet, we find that we can leverage this concept in persistent memory design to relax the write-order constraints on the caches.

Figure 1(a) shows that uncacheable, store-granular undo logging can eliminate the memory barrier between the log and working data writes. As long as the log entry ($Ulog_A_1$) is written into NVRAM before its corresponding store to the working data ($store\ A'_1$), we can undo the partially completed store after a system failure. Furthermore, $store\ A'_1$ must traverse the cache hierarchy. The uncacheable $Ulog_A_1$ may be buffered (e.g., in a four to six cache-line sized entry write-combining buffer in x86 processors). However, it still requires much less time to get out of the processor than cached stores. This naturally maintains the write ordering without explicit memory barrier instructions between the log and the persistent data writes. That is, logging and working data writes are performed in a pipeline-like manner (like in the timeline in Figure 1(a)). is similar to the "steal" attribute in DBMS [32], i.e, cached working data updates can steal the way into persistent storage before transaction commits. However, a downside is that undo logging requires a forced cache write-back before the transaction commits. This is necessary if we want to recover the latest transaction state after system failures. Otherwise, the data changes made by the transaction will not be committed to memory.

Instead, redo logging allows transactions to commit without explicit cache write-backs because the redo log, once updates complete, already has the latest version of the transactions (Figure 1(b)). This is similar to the "no-force" attribute in DBMS [32], i.e., no need to force the working data updates out of the caches at the end of transactions. However, we must use memory barriers to complete the redo log of A before any stores of A reach NVRAM. We illustrate this ordering constraint by the dashed blue line in the timeline. Otherwise, a system crash when the redo logging is incomplete, while working data A is partially overwritten in NVRAM (by $store\ A'_k$), causes data corruption.

Figure 1(c) shows that undo+redo logging combines the benefits of both "steal" and "no-force". As a result, we can eliminate the memory barrier between the log and persistent writes. A forced cache write-back (e.g., `clwb`) is unnecessary for an unlimited sized log. However, it can be postponed until after the transaction commits for a limited sized log (Section II-C).

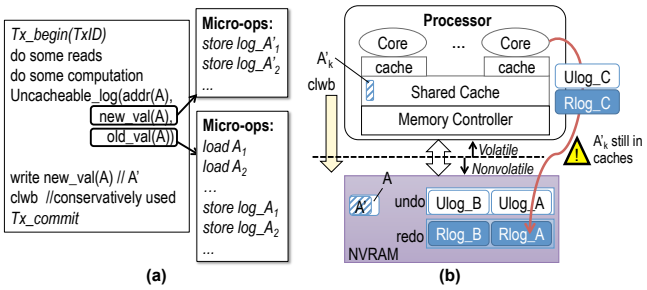


Figure 2. Inefficiency of logging in software.

C. Why Undo+Redo Logging in Hardware

Though promising, undo+redo logging is not used in persistent memory system designs because previous software logging schemes are inefficient (Figure 2).

Extra instructions in the CPU pipeline. Logging in software uses logging functions in transactions. Figure 2(a) shows that both undo and redo logging can introduce a large number of instructions into the CPU pipeline. As we demonstrate in our experimental results (Section VI), using only undo logging can lead to more than doubled instructions compared to memory systems without persistent memory. Undo+redo logging can introduce a prohibitively large number of instructions to the CPU pipeline, occupying compute resources needed for data movement.

Increased NVRAM traffic. Most instructions for logging are loads and stores. As a result, logging substantially increases memory traffic. In particular, undo logging must not only store to the log, but it must also first read the old values of the working data from the cache and memory hierarchy. This further increases memory traffic.

Conservative cache forced write-back. Logs can have a limited size[1]. Suppose that, without losing generality, a log can hold undo+redo records of two transactions (Figure 2(b)). To log a third transaction ($Ulog_C$ and $Rlog_C$), we must overwrite an existing log record, say $Ulog_A$ and $Rlog_A$ (transaction A). If any updates of transaction A (e.g., A'_k) are still in caches, we must force these updates into the NVRAM before we overwrite their log entry. The problem is that caches are invisible to software. Therefore, software does not know whether or which particular updates to A are still in the caches. Thus, once a log becomes full (after garbage collection), software may conservatively force cache write-backs before committing the transaction. This unfortunately negates the benefit of redo logging.

Risks of data persistence in multithreading. In addition to the above challenges, multithreading further complicates software logging in persistent memory, when a log is shared by multiple threads. Even if a persistent memory system issues `clwb` instructions in each transaction, a context switch by the OS can occur before the `clwb` instruction executes. This context switch interrupts the control flow of transactions and diverts the program to other threads. This reintroduces the aforementioned issue of prematurely overwriting the records in a filled log. Implementing per-thread logs can mitigate this risk. However, doing so can introduce new persistent memory API and complicates recovery.

These inefficiencies expose the drawbacks of undo+redo logging in software and warrants a hardware solution.

[1]Although we can grow the log size on demand, this introduces extra system overhead on managing variable size logs [19]. Therefore, we study fixed size logs in this paper.

III. Our Design

To address the challenges, we propose a hardware undo+redo logging design, consisting of Hardware Logging (HWL) and cache Force Write-Back (FWB) mechanisms. This section describes our design principles. We describe detailed implementation methods and the required software support in Section IV.

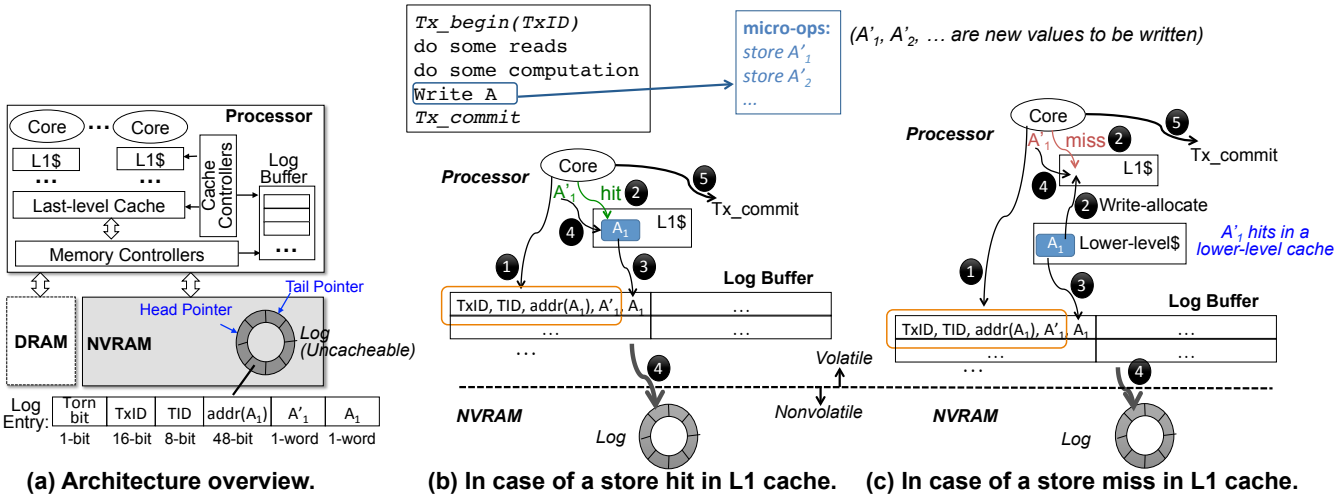
A. Assumptions and Architecture Overview

Figure 3(a) depicts an overview of our processor and memory architecture. The figure also shows the circular log structure in NVRAM. All processor components are completely volatile. We use write-back, write-allocate caches common to processors. We support hybrid DRAM+NVRAM for main memory, deployed on the processor-memory bus with separate memory controllers [19], [13]. However, this paper focuses on persistent data updates to NVRAM.

Failure Model. Data in DRAM and caches, but not in NVRAM, are lost across system reboots. Our design focuses on maintaining persistence of user-defined critical data stored in NVRAM. After failures, the system can recover this data by replaying the log in NVRAM. DRAM is used to store data without persistence [19], [13].

Persistent Memory Transactions. Like prior work in persistent memory [19], [22], we use persistent memory "transactions" as a software abstraction to indicate regions of memory that are persistent. Persistent memory writes require a persistence guarantee. Figure 2 illustrates a simple code example of a persistent memory transaction implemented with logging (Figure 2(a)), and our with design (Figure 2(b)). The transaction defines *object A* as critical data that needs persistence guarantee. Unlike most logging-based persistent memory transactions, our transactions eliminate explicit logging functions, cache forced write-back instructions, and memory barrier instructions. We discuss our software interface design in Section IV.

Uncacheable Logs in the NVRAM. We use single-consumer, single-producer Lamport circular structure [33] for the log. Our system software can allocate and truncate the log (Section IV). Our hardware mechanisms append the log. We chose a circular log structure because it allows simultaneous appends and truncates without locking [33],



(a) Architecture overview. **(b) In case of a store hit in L1 cache.** **(c) In case of a store miss in L1 cache.**

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

[19]. Figure 3(a) shows that log records maintain undo and redo information of a single update (e.g., *store* A_1). In addition to the undo (A_1) and redo (A_1') values, log records also contain the following fields: a 16-bit transaction ID, an 8-bit thread ID, a 48-bit physical address of the data, and a *torn bit*. We use a *torn bit* per log entry to indicate the update is complete [19]. Torn bits have the same value for all entries in one pass over the log, but reverses when a log entry is overwritten. Thus, completely-written log records all have the same torn bit value, while incomplete entries have mixed values [19]. The log must accommodate all write requests of undo+redo.

The log is typically used during system recovery, and rarely reused during application execution. Additionally, log updates must arrive in NVRAM in store-order. Therefore, we make the log *uncacheable*. This is in line with most prior works, in which log updates are written directly into a write-combine buffer (WCB) [19], [31] that coalesces multiple stores to the same cache line.

B. Hardware Logging (HWL)

The goal of our Hardware Logging (HWL) mechanism is to enable feasible undo+redo logging of persistent data in our microarchitecture. HWL also relaxes ordering constraints on caching in a manner that neither undo nor redo logging can. Furthermore, our HWL design leverages information naturally available in the cache hierarchy but not to the programmer or software. It does so without the performance overhead of unnecessary data movement or executing logging, cache force-write-back, or memory barrier instructions in pipeline.

Leveraging Existing Undo+Redo Information in Caches. Most processors caches use write-back, write-allocate caching policies [34]. On a write hit, a cache only updates the cache line in the hitting level with the new values. A dirty bit in the cache tag indicates cache values are modified but not committed to memory. On a write miss, the write-allocate (also called fetch-on-write) policy requires the cache to first load (i.e., allocate) the entire missing cache line before writing new values to it. HWL leverages the write-back, write-allocate caching policies to feasibly enable undo+redo logging in persistent memory. HWL automatically triggers a log update on a persistent write in hardware. HWL records both redo and undo information in the log entry in NVRAM (shown in Figure 2(b)). We get the redo data from the currently in-flight write operation itself. We get the undo data from the write request's corresponding write-allocated cache line. If the write request hits in the L1 cache, we read the old value before overwriting the cache line and use that for the undo log. If the write request misses in L1 cache, that cache line must first be allocated anyway, at which point we get the undo data in a similar manner. The log entry, consisting of a transaction ID, thread, the address of the write, and undo and redo values, is written out to the circular log in NVRAM using the head and tail pointers. These pointers are maintained in special registers described in Section IV.

Inherent Ordering Guarantee Between the Log and Data. Our design does not require explicit memory barriers to enforce that undo log updates arrive at NVRAM before its corresponding working data. The ordering is naturally ensured by how HWL performs the undo logging and working data updates. This includes i) the uncached log updates and cached working data updates, and ii) store-granular undo logging. The working data writes must traverse the cache hierarchy, but the uncacheable undo log updates do not. Furthermore, our HWL also provides an optional volatile log buffer in the processor, similar to the write-combining buffers in commodity processor design, that coalesces the log updates. We configure the number of log buffer entries based on cache access latency. Specifically, we ensure that the log updates write out of the log buffer before a cached

store writes out of the cache hierarchy. Section IV-C and Section VI further discuss and evaluate this log buffer.

C. Decoupling Cache FWBs and Transaction Execution

Writes are seemingly persistent once their logs are written to NVRAM. In fact, we can commit a transaction once logging of that transaction is completed. However, this does not guarantee data persistence because of the circular structure of the log in NVRAM (Section II-A). However, inserting cache write-back instructions (such as `clflush` and `clwb`) in software can impose substantial performance overhead (Section II-A). This further complicates data persistence support in multithreading (Section II-C).

We eliminate the need for forced write-back instructions and guarantee persistence in multithreaded applications by designing a cache Force-Write-Back (FWB) mechanism in hardware. FWB is decoupled from the execution of each transaction. Hardware uses FWB to force certain cache blocks to write-back when necessary. FWB introduces a *force write-back* bit (*fwb*) alongside the tag and dirty bit of each cache line. We maintain a finite state machine in each cache block (Section IV-D) using the *fwb* and *dirty* bits. Caches already maintain the *dirty* bit: a cache line update sets the bit and a cache eviction (write-back) resets it. A cache controller maintains our *fwb* bit by scanning cache lines periodically. On the first scan, it sets the *fwb* bit in dirty cache blocks if unset. On the second scan, it forces write-backs in all cache lines with $\{fwb, dirty\} = \{1, 1\}$. If the *dirty* bit ever gets reset for any reason, the *fwb* bit also resets and no forced write-back occurs.

Our FWB design is also decoupled from software multithreading mechanisms. As such, our mechanism is impervious to software context switch interruptions. That is, when the OS requires the CPU to context switch, hardware waits until ongoing cache write-backs complete. The frequency of the forced write-backs can vary. However, forced write-backs must be faster than the rate at which log entries with uncommitted persistent updates are overwritten in the circular log. In fact, we can determine force write-back frequency (associated with the scanning frequency) based on the log size and the NVRAM write bandwidth (discussed in Section IV-D). Our evaluation shows the frequency determination (Section VI).

D. Instant Transaction Commits

Previous designs require software or hardware memory barriers (and/or cache force-write-backs) at transaction commits to enforce write ordering of log updates (or persistent data) into NVRAM across consecutive transactions [13], [26]. Instead, our design gives transaction commits a "free ride". That is, no explicit instructions are needed. 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 working data. We also write the log updates into NVRAM in the order they are issued (the log buffer is a FIFO). Therefore, log updates of subsequent transactions can only be written into NVRAM after current log updates are written and committed.

E. Putting It All Together

Figure 3(b) and (c) illustrate how our hardware logging works. Hardware treats all writes encompassed in persistent transactions (e.g., `write A` in the transaction delimited by `tx_begin` and `tx_commit` in Figure 2(b)) as persistent writes. Those writes invoke our HWL and FWB mechanisms. They work together as follows. Note that log updates go directly to the WCB or NVRAM if the system does not adopt the log buffer.

The processor sends writes of data object A (a variable or other data structure), consisting of new values of one or more cache lines $\{A'_1, A'_2, ...\}$, to the L1 cache. Upon updating an L1 cache line (e.g., from old value A_1 to a new value A'_1):

1) Write the new value (redo) into the cache line (❶).
   a) If the update is the first cache line update of data object A, the HWL mechanism (which has the transaction ID and the address of A from the CPU) writes a log record header into the log buffer.
   b) Otherwise, the HWL mechanism writes the new value (e.g., A'_1) into the log buffer.
2) Obtain the undo data from the old value in the cache line (❷). This step runs parallel to Step-1.
   a) If the cache line write request hits in L1 (Figure 3(b)), the L1 cache controller immediately extracts the old value (e.g., A_1) from the cache line before writing the new value. The cache controller reads the old value from the hitting line out of the cache read port and writes it into the log buffer in the Step-3. No additional read instruction is necessary.
   b) If the write request misses in the L1 cache (Figure 3(c)), the cache hierarchy must write-allocate that cache block as is standard. The cache controller at a lower-level cache that owns that cache line extracts the old value (e.g., A_1). The cache controller sends 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 updates the cache line in the L1 cache (❹). The cache line can be evicted via standard cache eviction policies without being subjected to data persistence constraints. Additionally, our log buffer is small enough to guarantee that log updates traverse through the log buffer faster than the cache line traverses the cache hierarchy (Section IV-D).

Therefore, this step occurs without waiting for the corresponding log entries to arrive in NVRAM.

5) The memory controller evicts the log buffer entries to NVRAM in a FIFO 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 coalesces the log updates of any writes to the same cache line.
7) After log entries of all the writes in the transaction are issued, the transaction can commit (❺).
8) Persistent working data updates remain cached until they are written back to NVRAM by either normal eviction or our cache FWB.

F. Discussion

Types of Logging. Systems with non-volatile memory can adopt centralized [35] or distributed (e.g., per-thread) logs [36], [37]. Distributed logs can be more scalable than centralized logs in large systems from software's perspective. Our design works with either type of logs. With centralized logging, each log record needs to maintain a thread ID, while distributed logs do not need to maintain this information in log records. With centralized log, our hardware design effectively reduces the software overhead and can substantially improve system performance with real persistent memory workloads as we show in our experiments. In addition, our design also allows systems to adopt alternative formats of distributed logs. For example, we can partition the physical address space into multiple regions and maintain a log per memory region. We leave the evaluation of such log implementations to our future work.

NVRAM Capacity Utilization. Storing undo+redo log can consume more NVRAM space than either undo or redo alone. Our log uses a fixed-size circular buffer rather than doubling any previous undo or redo log implementation. The log size can trade off with the frequency of our cache FWB (Section IV). The software support discussed in Section IV-A allow users to determine the size of the log. Our FWB mechanism will adjust the frequency accordingly to ensure data persistence.

Lifetime of NVRAM Main Memory. The lifetime of the log region is not an issue. Suppose a log has 64K entries (4MB) and NVRAM (assuming phase-change memory) has a 200 ns write latency. Each entry will be overwritten once every 64K $\times$ 200 ns. If NVRAM endurance is 10^8 writes, a cell, even statically allocated to the log, will take 15 days to wear out, which is plenty of time for conventional NVRAM wear-leveling schemes to trigger [38], [39], [40]. In addition, our scheme has two impacts on overall NVRAM lifetime: logging normally leads to write amplification, but we improve NVRAM lifetime because our caches coalesce writes. The overall impact is likely slightly negative. However, wear-leveling will trigger before any damage occurs.

```
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 example for `tx_begin` and `tx_commit`, where thread ID is transaction ID to perform one persistent transaction per thread.

IV. Implementation

In this section, we describe the implementation details of our design and hardware overhead. We covered the impact of NVRAM space consumption, lifetime, and endurance in Section III-F.

A. Software Support

Our design has 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 use a pair of transaction functions, `tx_begin( txid )` and `tx_commit()`, that define transactions which do persistent writes in the program. We use `txid` to provide the transaction ID information used by our HWL mechanism. This ID is groups writes from the same transaction. This transaction interface has been used by numerous previous persistent memory designs [13], [29]. Figure 4 shows an example of multithreaded pseudocode with our transaction functions.

System Library Functions Maintain the Log. Our HWL mechanism performs log updates, while the system software maintains the log structure. In particular, we use system library functions, `log_create()` and `log_truncate()` (similar to functions used in prior work [19]), to allocate and truncate the log, respectively. The system software sets the log size. The memory controller obtains log maintenance information by reading special registers (Section IV-B), indicating 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 overflows: 1) The `log_create()` function allocates a large-enough log by reading 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.

B. Special Registers

The `txid` argument from `tx_begin()` translates into an 8-bit unsigned integer (a *physical* transaction ID) stored in a special register in the processor. Because the transaction IDs 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 use 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 using `log_create()`. During log updates, the memory controller and `log_truncate()` function update the pointers. If `log_grow()` is used, we employ additional registers to store the head and tail pointers of newly allocated log regions and an indicator of the active log region.

C. An Optional Volatile Log Buffer

To improve performance of log updates to NVRAM, we provide an optional log buffer (a volatile FIFO, similar to WCB) in the memory controller to buffer and coalesce log updates. This log buffer is not required for ensuring data persistence, but only for performance optimization.

Data persistence requires that log records arrive at NVRAM before the corresponding cache line with the working data. Without the log buffer, log updates are directly forced to the NVRAM bus without buffering in the processor. If we choose to adopt a log buffer with N entries, a log entry will take N cycles to reach the NVRAM bus. A data store sent to the L1 cache takes at least the latency (cycles) of all levels of cache access and memory controller queues before reaching the NVRAM bus. The the minimum value of this latency is known at design time. Therefore, we can ensure that log updates arrive at the NVRAM bus before the corresponding data stores by designing N to be smaller than the minimum number of cycles for a data store to traverse through the cache hierarchy. Section VI evaluates the bound of N and system performance across various log buffer sizes based on our system configurations.

D. Cache Modifications

To implement our cache force write-back scheme, we add one `fwb` bit to the tag of each cache line, alongside the *dirty* bit as in conventional cache implementations. FWB maintain three states (IDLE, FLAG, and FWB) for each cache block using these state bits.

Cache Block State Transition. Figure 5 shows the finite-state machine for FWB, implemented in the cache controller of each level. When an application begins executing, cache controllers initialize (reset) each cache line to the IDLE state by setting `fwb` bit to 0. Standard cache implementation also initializes *dirty* and *valid* bits to 0. During application execution, baseline cache controllers naturally set the *dirty* and *valid* bits to 1 whenever a cache line is written and reset the *dirty* bit back to 0 after the cache line is written back to a lower level (typically on eviction). To implement our state machine, the cache controllers periodically scan the valid, dirty, and `fwb` bits of each cache line and performs the following.

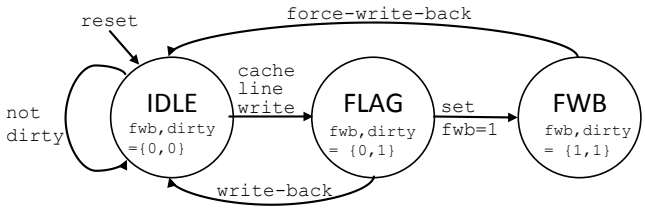


Figure 5. State machine in cache controller for FWB.

- A cache line with $\{fwb, dirty\} = \{0, 0\}$ is in IDLE state; the cache controller does nothing to those cache lines;
- A cache line with $\{fwb, dirty\} = \{0, 1\}$ is in the FLAG state; the cache controller sets the `fwb` bit to 1. This indicates that the cache line needs a write-back during the next scanning iteration if it is still in the cache.
- A cache line with $\{fwb, dirty\} = \{1, 1\}$ is in FWB state; the cache controller force writes-back this line. After the forced write-back, the cache controller changes the line back to IDLE state by resetting $\{fwb, dirty\} = \{0, 0\}$.
- If a cache line is evicted from the cache at any point, the cache controller resets its state to IDLE.

Determining the Cache FWB Frequency. The tag scanning frequency determines the frequency of our cache force write-back operations. The FWB must occur as frequently as to ensure that the working data is written back to NVRAM before its log records are overwritten by newer updates. As a result, the more frequent the write requests, the more frequent the log will be overwritten. The larger the log, the less frequent the log will be overwritten. Therefore, the scanning frequency is determined by the maximum log update frequency (bounded by NVRAM write bandwidth since applications cannot write to the NVRAM faster than its bandwidth) and log size (see the sensitivity study in Section VI). To accommodate large cache sizes with low scanning performance overhead, we also grow the size of the log to reduce the scanning frequency accordingly.

E. Summary of Hardware Overhead

Table I presents the hardware overhead of our implemented design in the processor. Note that these values may vary depending on the native processor and ISA. Our implementation assumes a 64-bit machine, hence 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. This is much lower than previous studies that track transaction

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 I
SUMMARY OF MAJOR HARDWARE OVERHEAD.

information in cache tags [13]. The numbers in the table were computed based on the specifications of all our system caches described in Section V.

Note that these are major state logic components on-chip. Our design also also requires additional gates for logic operations. However, these gates are primarily small and medium-sized gates, on the same complexity level as a multiplexer or decoder.

F. Recovery

We outline the steps of recovering the persistent 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 use only one centralized circular log for all transactions for all threads.

Step 2: The 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. The addresses are maintained via page table in NVRAM. We identify which writes did not commit by tracing back from the tail pointer. Log entries with mismatched values in NVRAM are considered non-committed. The address stored with each entry corresponds to the address of the persistent data member. Aside from the head and tail pointers, we also use the *torn bit* to correctly order these writes [19]. Log entries with the same `txid` and torn bit are complete.

Step 3: The generated writes bypass the caches and go directly to NVRAM. We use volatile caches, so their states are reset and all generated writes on recovery are persistent. Therefore, they can bypass the caches without issue.

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

V. EXPERIMENTAL SETUP

We evaluate our design by implementing it in McSimA+ [41], a Pin-based [42] cycle-level multi-core simulator. We configure the simulator to model a multi-core out-of-order processor with NVRAM DIMM described in Table II. Our simulator also models additional memory traffic for logging and clwb instructions. We feed the performance simulation results into McPAT [43], a widely used architecture-level power and area modeling tool, to estimate processor dynamic energy consumption. We modify the McPAT processor configuration to model our hardware modifications, including the components added to support HWL and FWB. We adopt phase-change memory parameters in the NVRAM DIMM [44]. Because all of our performance numbers shown in Section VI are relative, the same observations are valid for different NVRAM latency and access energy. Our work focuses on improving persistent memory access so we do not evaluate DRAM access in our experiments.

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 [44].
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 [44]

Table II
PROCESSOR AND MEMORY CONFIGURATIONS.

Name	Memory Footprint	Description
Hash [29]	256 MB	Searches for a value in an open-chain hash table. Insert if absent, remove if found.
RBTree [13]	256 MB	Searches for a value in a red-black tree. Insert if absent, remove if found
SPS [13]	1 GB	Random swaps between entries in a 1 GB vector of values.
BTree [45]	256 MB	Searches for a value in a B+ tree. Insert if absent, remove if found
SSCA2 [46]	16 MB	A transactional implementation of SSCA 2.2, performing several analyses of large, scale-free graph.

Table III
A LIST OF EVALUATED MICROBENCHMARKS.

We evaluate both microbenchmarks and real workloads in our experiments. The microbenchmarks repeatedly update persistent memory storing to different data structures including hash table, red-black tree, array, B+tree, and graph. These are data structures widely used in storage systems [29]. Table III describes these benchmarks. Our experiments use multiple 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 require multiple cache lines per element. This allows us to

explore 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 III. We compile these benchmarks in native x86 and run them on the McSimA+ simulator. We evaluate both singlethreaded and multithreaded versions of each benchmark. In addition, we evaluate the set of real workload benchmarks from the WHISPER persistent memory benchmark suite [11]. The benchmark suite incorporates various workloads, such as key-value stores, in-memory databases, and persistent data caching, which are likely to benefit from future persistent memory techniques.

VI. Results

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

- **non-pers** – This uses NVRAM as a working memory without any data persistence or logging. This configuration yields an ideal yet unachievable performance for persistent memory systems [13].
- **unsafe-base** – This uses software logging without forced cache write-backs. As such, it does not guarantee data persistence (hence "unsafe"). Note that the dashed lines in our figures show the best case achieved between either redo or undo logging for that benchmark.
- **redo-clwb** and **undo-clwb** – Software redo and undo logging, respectively. These invoke the `clwb` instruction to force cache write-backs after persistent transactions.
- **hw-rlog** and **hw-ulog** – Hardware redo or undo logging with no persistence guarantee (like in unsafe-base). These show an extremely optimized performance of hardware undo or redo logging [13].
- **hwl** – This design includes undo+redo logging from our hardware logging (HWL) mechanism, but uses the `clwb` instruction to force cache write-backs.
- **fwb** – This is the full implementation of our hardware undo+redo logging design with both HWL and FWB.

A. Microbenchmark Results

We make the following major observations of our microbenchmark experiments and analyze the results. We evaluate benchmark configurations from single to eight threads. The prefixes of these results correspond to one (*-1t*), two (*-2t*), four (*-4t*), and eight (*-8t*) threads.

System Performance and Energy Consumption. Figure 6 and Figure 8 compare the transaction throughput and memory dynamic energy of each design. We observe that processor dynamic energy is not significantly altered by different configurations. Therefore, we only show memory dynamic energy in the figure. The figures illustrate that *hwl* alone improves system throughput and dynamic energy consumption, compared with software logging. Note that our design supports undo+redo logging, while the evaluated software 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 one thread 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 logging. This is because SSCA2 and BTree use more complex data structures, where the overhead of manipulating the data structures outweigh that of the log structures. Figure 9 shows that our design substantially reduces NVRAM writes.

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 1.86× throughput, with only 6% processor-memory and 20% dynamic memory energy overhead, respectively. Furthermore, our design's performance and energy benefits over software logging remain as we increase the number of threads.

IPC and Instruction Count. We also study IPC number of executed instructions, shown in Figure 7. Overall, *hwl* and *fwb* significantly improve IPC over software logging. This appears promising because the figure shows our hardware logging design executes much fewer instructions. Compared with *non-pers*, software logging imposes up to 2.5× the number of instructions executed. Our design *fwb* only imposes a 30% instruction overhead.

Performance Sensitivity to Log Buffer Size. Section IV-C discusses how the log buffer size is bounded by the data persistence requirement. The log updates must arrive at NVRAM before its corresponding working data updates. This bound is $\leq$15 entries based on our processor configuration. Indeed, larger log buffers better improve throughput as we studied using the *hash* benchmark (Figure 11(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 the log buffer size, which may no longer guarantee data persistence, additionally improves system throughput until reaching the NVRAM write 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. We also improve throughput over baseline hardware logging *hw-rlog* and *hw-ulog*.

Relation Between FWB Frequency and Log Size. Section IV-D discusses that the force write-back frequency is determined by the NVRAM write bandwidth and log size. With a given NVRAM write bandwidth, we study the relation between the required FWB frequency and log size. Figure 11(b) shows that we only need to perform forced

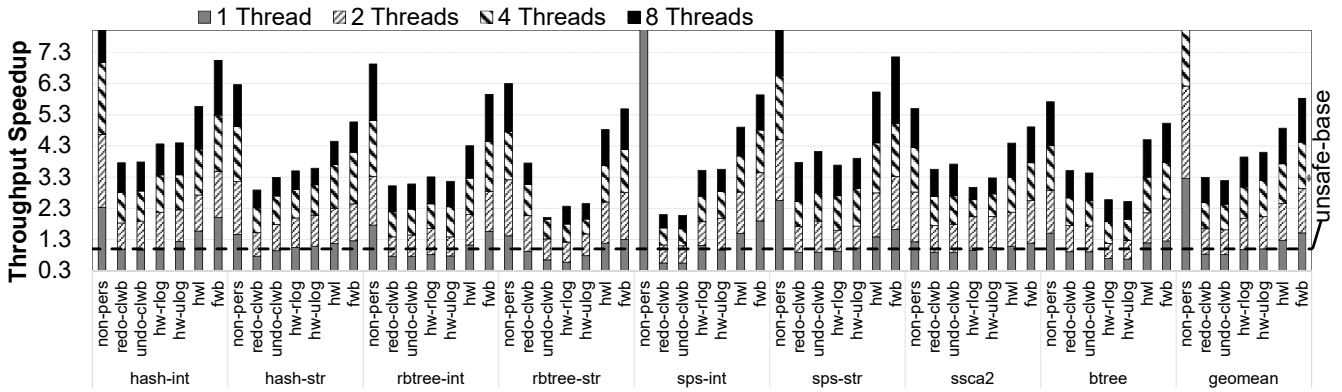


Figure 6. Transaction throughput speedup (higher is better), normalized to *unsafe-base*.

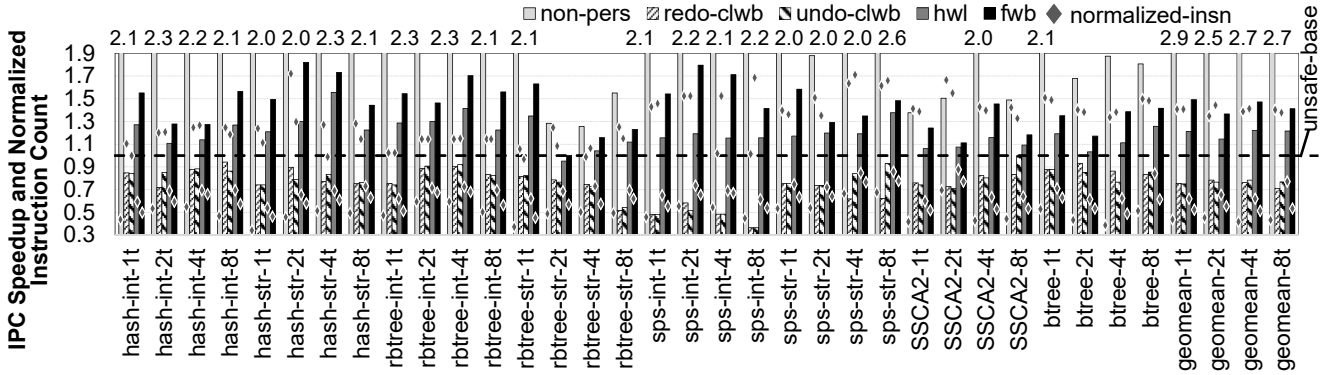


Figure 7. IPC speedup (higher is better) and instruction count (lower is better), normalized to *unsafe-base*.

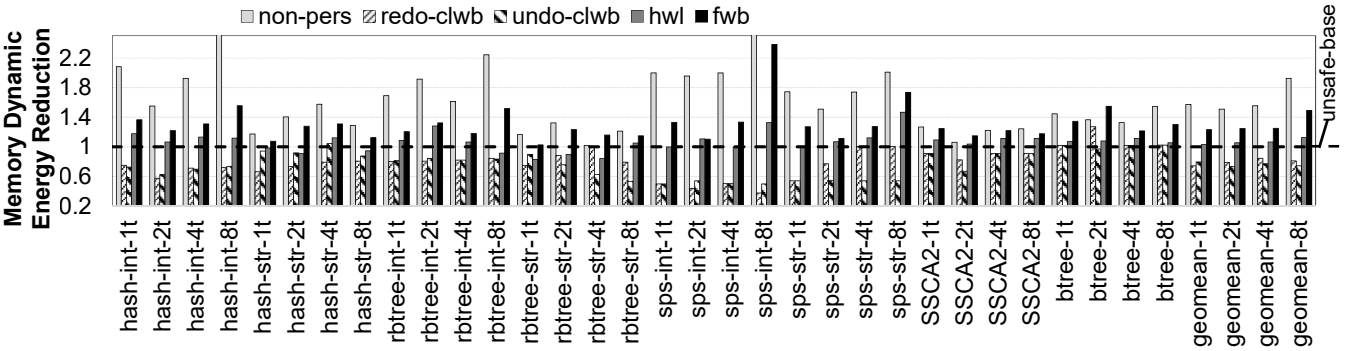


Figure 8. Dynamic energy reduction (higher is better), normalized to *unsafe-base* (dashed line).

write-backs every three million cycles if we have a 4MB log. As a result, the fwb tag scanning only introduces 3.6% performance overhead with our 8MB cache.

B. WHISPER Results

Compared with microbenchmarks, we observe even more promising performance and energy improvements in real persistent memory workloads in the WHISPER benchmark suite with large data sets (Figure 10). Among the WHISPER benchmarks, ctree and hashmap benchmarks accurately correspond to and reflect the results achieved in our microbenchmarks due to their similarities. Although the magnitude of improvement vary, our design leads to much higher performance, lower energy, and lower NVRAM traffic than our baselines. Compared with *redo-clwb* and *undo-clwb*, our design significantly reduces the dynamic memory energy consumption of tpcc and ycsb due to the high write intensity in these workloads. Overall, our design (*fwb*) achieves up to 2.7× the throughput of the best case in *redo-clwb* and *undo-clwb*. This is also within 73% of *non-pers* throughput of the same benchmarks. In addition, our design achieves up to a 2.43× reduction in dynamic memory over the baselines.

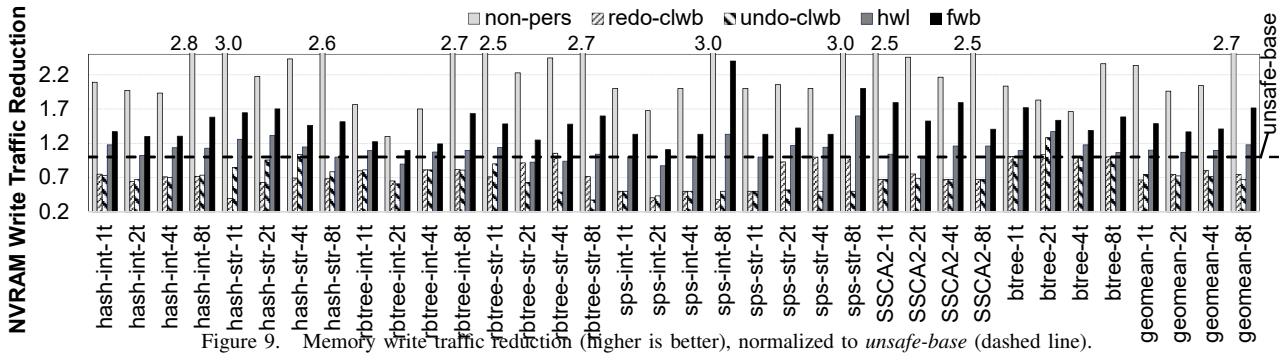


Figure 9. Memory write traffic reduction (higher is better), normalized to *unsafe-base* (dashed line).

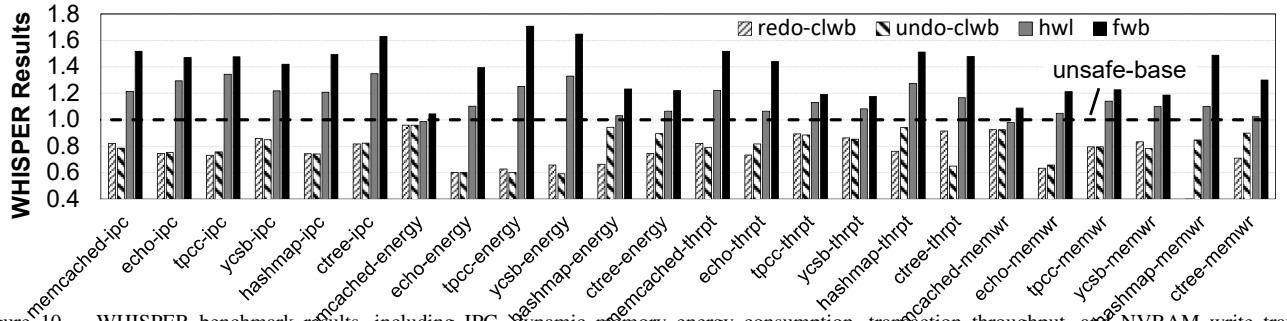


Figure 10. WHISPER benchmark results, including IPC, dynamic memory energy consumption, transaction throughput, and NVRAM write traffic, normalized to *unsafe-base* (the dashed line).

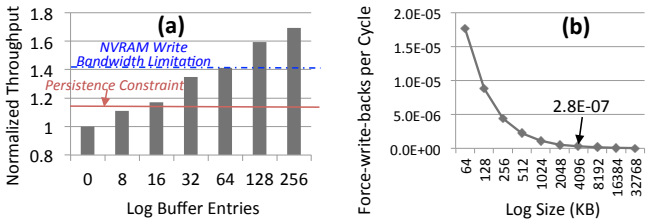


Figure 11. Sensitivity studies of (a) system throughput with varying log buffer sizes and (b) cache fwb frequency with various NVRAM log sizes.

VII. Related Work

Compared to previous architecture support for persistent memory systems, our design further relaxes ordering constraints on caches with less hardware cost.[2]

Hardware support for logging. Several recent studies proposed hardware support for log-based persistent memory design. Lu *et al.* proposes custom hardware logging mechanisms and multi-versioning caches to reduce intra- and inter-transaction dependencies [24]. However, they require both large-scale changes to the cache hierarchy and cache multi-versioning support. Kolli *et al.* proposes a delegated persist ordering [15] 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. Instead, our design is flexible because it directly leverages the information already in the baseline cache hierarchy. ATOM [35] and DudeTM [47] only implement either undo or redo logging. As a result, the studies do not provide the level of relaxed ordering offered by our design. In addition, DudeTM [47] also relies on a *shadow* memory which can incur substantial memory access cost. Doshi *et al.* uses redo logging for data recoverability with a backend controller [48]. The backend controller reads log entries from the log in memory and updates data in-place. However, this design can unnecessarily saturate the memory read bandwidth needed for critical read operations. Also, it requires a separate victim cache to protect from dirty cache blocks. Instead, our design directly uses dirty cache bits to enforce persistence.

Hardware support for persistent memory. Recent studies also propose general hardware mechanisms for persistent memory with or without logging. Recent works propose that caches may be implemented in software [49], or an additional non-volatile cache integrated in the processor [50], [13] to maintain persistence. However, doing so can double the memory footprint for persistent memory operations. Other works [31], [26], [51] optimize the memory controller to improve performance by distinguishing logging and data updates. Epoch barrier [29], [16], [52] is proposed to relax the ordering constraints of persistent memory by allowing coarse-grained transaction ordering. However, epoch barriers incur non-trivial overhead to the cache hierarchy. Furthermore, system performance can be sub-optimal with small epoch sizes, which is observed in many persistent memory workloads [11]. Our design uses lightweight hardware changes on existing processor designs without expensive non-volatile on-chip transaction buffering components.

[2]Volatile TM supports concurrency but does not guarantee persistence in memory.

Persistent memory design in software. Previous works, such as Mnemosyne [19] and REWIND [53], utilize write-ahead logging implemented in software. These rely on instructions, such as `clflush`, `clwb`, and `pcommit`, to achieve persistency and enforce log-data ordering in their critical path. Our design does not require these persistent instructions that we've shown can clog the pipeline and are often inefficient or unnecessary. JUSTDO [54] logging also relies on these instructions (for manual flushing by the programmer). Additionally, these works' programming models all falter because Intel's `pcommit` instruction is now deprecated. Our design works on legacy code and does not have strict functionality requirements on the programming model. This makes it immune to relying on instructions or software functions that become deprecated.

VIII. Conclusions

We proposed a hardware logging scheme that allows the caches to perform cache line updates and write-backs without non-volatile buffers or caches in the processor. These mechanisms make up a complexity-effective design that excels over traditional software logging for persistent memory. Our evaluation shows that our design significantly increases performance while reducing memory traffic and energy.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This paper is supported in part by NSF grants 1652328 and 1718158, and NSF I/UCRC Center for Research on Storage Systems.

References

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

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

[3] 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*, Sept 2006, pp. 40–43.

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

[5] 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.

[6] "ARM, ARMv8-a architecture evolution," 2016.

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

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

[9] J. Moyer, "Persistent memory in Linux," in *SNIA NVM Summit*, 2016.

[10] K. Deierling, "Persistent memory over fabric," in *SNIA NVM Summit*, 2016.

[11] 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*, 2017, pp. 1–14.

[12] 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.

[13] 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)*, 2013, pp. 421–432.

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

[15] 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*, 2016, pp. 1–13.

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

[17] 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*, 2006, pp. 1–12.

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

[19] 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*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 91–104.

[20] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the NVRAM era," *Proceedings of the VLDB Endowment*, vol. 7, no. 2, 2013.

[21] 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*, 2013, pp. 1–17.

[22] 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*, 2016, pp. 1–12.

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

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

[25] 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*, 2014, pp. 1–12.

[26] 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*, 2014, pp. 455–470.

[27] 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*, 2012, pp. 387–400.

[28] J. Arulraj, M. Perron, and A. Pavlo, "Write-behind logging," *Proc. VLDB Endow.*, vol. 10, no. 4, pp. 337–348, Nov. 2016.

[29] 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*, 2011, pp. 105–118.

[30] 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)*, 2015, pp. 1–13.

[31] 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.

[32] 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.*, vol. 17, no. 1, pp. 94–162, Mar. 1992.

[33] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Softw. Eng.*, vol. 3, no. 2, pp. 125–143, Mar. 1977.

[34] 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)*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[35] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 361–372.

[36] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proc. VLDB Endow.*, vol. 7, no. 10, pp. 865–876, Jun. 2014.

[37] J. Huang, K. Schwan, and M. K. Qureshi, "Nvram-aware logging in transaction systems," *Proc. VLDB Endow.*, vol. 8, no. 4, pp. 389–400, Dec. 2014.

[38] 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*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 14–23.

[39] 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*, 2009, pp. 14–23.

[40] 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*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 24–33.

[41] J. H. Ahn, S. Li, O. Seongil, and N. Jouppi, "Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, 2013, pp. 74–85.

[42] 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*, New York, NY, USA, 2005, pp. 190–200.

[43] 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*, 2009, pp. 469–480.

[44] 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*, 2009, pp. 2–13.

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

[46] 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*, 2005, pp. 465–476.

[47] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 329–343.

[48] 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)*, 2016, pp. 77–89.

[49] P. Li, D. R. Chakrabarti, C. Ding, and L. Yuan, "Adaptive software caching for efficient nvram data persistence," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 112–122.

[50] C.-H. Lai, J. Zhao, and C.-L. Yang, "Leave the cache hierarchy operation as it is: A new persistent memory accelerating approach," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: ACM, 2017, pp. 5:1–5:6.

[51] 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*, 2015, pp. 24:1–24:8.

[52] 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*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 133–146.

[53] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "REWIND: Recovery write-ahead system for in-memory non-volatile data-structures," *Proc. VLDB Endow.*, vol. 8, no. 5, pp. 497–508, Jan. 2015.

[54] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *SIGPLAN Not.*, vol. 51, no. 4, pp. 427–442, Mar. 2016.