Efficient Hardware-assisted Logging with Asynchronous and Direct-Update for Persistent Memory

Jungi Jeong
School of Computing
KAIST
jgjeong@calab.kaist.ac.kr

Chang Hyun Park
School of Computing
KAIST
changhyunpark@calab.kaist.ac.kr

Jaehyuk Huh
School of Computing
KAIST
jhhuh@kaist.ac.kr

Seungryoul Maeng
School of Computing
KAIST
maeng@kaist.ac.kr

***Abstract*—Supporting atomic durability in emerging persistent memory requires data consistency across potential system failures. For atomic durability support in the non-volatile memory, the traditional write-ahead log (WAL) technique has been employed to guarantee the persistency of logs before actual data updates. Based on the WAL mechanism, recent studies proposed HW-assisted logging techniques with undo, redo, or undo+redo principles. The HW log manager allows the overlapping of log writing and transaction execution, as long as the atomicity invariant can be satisfied. Although the efficiency of both log and data writes must be optimized, the prior work exhibit trade-offs in performance under various access patterns. The undo approach experiences performance degradation due to synchronous in-place data updates since the log contains only the old values. On the other hand, the undo+redo approach stores both old and new values, and does not require synchronous in-place data updates. However, the larger log size increases the amount of log writes. The prior redo approach demands extra NVM read bandwidth for indirectly updating in-place data from the new values in logs. To overcome the limitations of the previous approaches, this paper proposes a novel redo-based logging (`ReDU`), which performs direct and asynchronous in-place data update to NVM. `ReDU` exploits a small region of DRAM as a write-cache to remove NVM writes from the critical path. The experimental results show that the proposed logging mechanism provides the best performance under a variety of write patterns, showing 8.6%, 14.2%, and 23.6% better performance compared to the previous undo, redo, and undo+redo approaches, respectively.**

I. Introduction

Non-volatile memory (NVM) technologies such as PCM, STT-MRAM, and ReRAM have emerged to provide persistency as well as much higher storage density than DRAM. Such NVM can provide access latencies closer to DRAM compared to the prior generation flash technologies [1], [14], [18]–[20]. The latency, density, and persistency of NVMs lead to a new storage class memory (SCM), providing persistent storage through memory interfaces. However, as a persistent storage medium, SCM is required to support atomic durability of transactional execution under potential system failures.

The atomic durability provides failure atomicity of data updates in persistent memory. All changes made in a transaction must be made persistent as a whole, and such atomicity must be held even on a system failure. Traditional mechanisms to provide such atomic durability are the write-ahead logging (WAL) techniques [7], [8], [12], [16], [17], [21], [28], [29], [31]. Before the updated data are written to NVM, the logs must be made persistent first. To support such logging, new memory interface instructions such as `clwb` have been added to delineate the completion of writing to the persistent memory [13]. Using the interface, applications can write logs before data updates, and make sure the log entries are made persistent before the data are written to NVM.

However, a critical disadvantage of the SW-only logging techniques is the performance overhead. Logging must be done with additional instructions within a transaction, increasing the execution cycles and only supporting coarse-grained ordering [7], [24], [28], [31]. Unlike the SW-based logging techniques, recent studies investigated the potential benefits of HW-based logging mechanisms, which reduce the logging overheads by hiding them from the critical path of the transaction execution [10], [15], [24], [28]. With the HW-based logging, log writing can be overlapped with the instruction execution in a transaction, as long as the log writing can be completed before the modified data are written to NVM.

For such HW-based logging techniques, the prior approaches have adopted the classical *undo* [15], [28], *redo* [10], and *undo+redo* [24] principles, which differ in their log contents. Under these logging principles, this paper categorizes the design space of HW logging mechanism in terms of the efficiency of log write and data update. For the efficiency of log writing, a critical aspect is to reduce the amount of log writes and the number of logging requests as much as possible. This paper identifies two key techniques for the logging efficiency; Log *coalescing* combines multiple log entries for contiguous addresses to reduce the size of log entry, and log *packing* assembles multiple log requests into a single burst request to reduce the number of log requests.

The second aspect of the design space of HW logging is the data update mechanism. Even with WAL, committed data must be eventually reflected in NVM through data updates. First, data updates can be conducted either synchronously or asynchronously with the transaction commit. The synchronous updates place cache-flush operations on the critical path,

updating the data in NVM before the commit. On the other hand, the asynchronous updates can postpone data updates of committed transactions, which provides shorter transaction commit latencies. Second, data written to NVM can be read directly from the cache hierarchy (direct updates) or indirectly from the logs (indirect updates) stored in NVM. The direct update is more bandwidth efficient since the indirect update requires extra NVM reads for reading logs to update the NVM in-place data. For the two aspects, this paper advocates *asynchronous* and *direct* updates to NVM to reduce the critical path latency of transactions and to curtail unnecessary log reads from NVM.

Although both of the log write and data update efficiency factors must be improved, none of the prior work provides all the necessary properties for the efficiency. The undo approach is limited to the synchronous update model since it stores only old data in the logs [15], [28]. The undo+redo approach allows asynchronous direct updates, but increases the amount of log writes for storing both old and new values [24]. While the redo approach can support asynchronous updates, the previous redo approach suffers from the performance degradation due to the indirect updates [15].

To overcome the limitation, we propose a **Re**do-based logging with **D**irect-**U**pdate (`ReDU`), which pursues asynchronous and direct data updates to NVM, combined with efficient log write supports of coalescing and packing. When the transaction commits, `ReDU` flushes updated cachelines from caches *synchronously* only to the DRAM cache, a small region reserved in DRAM that we exploit as a write-cache for NVM. Data stored in the DRAM cache are *asynchronously* flushed to NVM. The use of the DRAM cache decouples slow NVM data updates from the critical path of the transaction, while eliminating NVM reads for data updates required in the prior redo approach.

The experimental results show that `ReDU` provides better performance under a variety of write patterns, compared to the previous undo, redo, and undo+redo approaches. For large and sequential write patterns, `ReDU` provides 21.2% and 35.4% higher throughput than the redo and undo+redo logging, respectively, and for small and random patterns, 16.7% and 21.7% better performance than the undo and undo+redo logging, respectively. Each of the prior approaches provide good performance for only one of two scenarios, exhibiting performance trade-offs depending on write patterns. In both types of scenarios, `ReDU` outperforms all three prior approaches.

The main contributions of this paper are as follows:

- This paper explores the design space of hardware-assisted logging in two aspects: log writes and data updates. We also identify key optimizations on log writes and desired design properties in data updates.
- This paper investigates the previous approaches under the considered design properties for efficient log-write and data-update. We found that the prior approaches exhibit performance trade-offs with different write patterns.

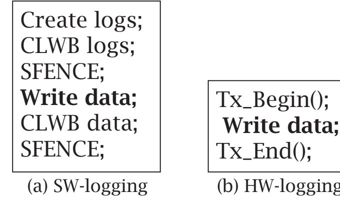


Fig. 1: Example codes of SW and HW logging in persistent memory.

- This paper proposes the redo-based logging that performs in-place data updates directly from the DRAM cache. It provides the performance benefit of direct data update, while allowing asynchronous updates to NVM.

The rest of the paper is organized as follows. Section II discusses the hardware-based logging mechanism for persistent memory, and Section III discusses the design space of hardware-assisted logging and compares the trade-offs in previous approaches. We provide the details of `ReDU` architecture in Section IV. Section V presents the performance evaluation. We summarize the related work in Section VI, followed by the conclusion of this paper.

II. Background

A. Atomic Durability through Logging

In designing systems with NVM-based persistent memory, it is a challenge to provide the consistency of in-memory data in the presence of system failures. In-memory data structures need to be consistently updated to recover the program state and data from hardware and software crashes. For example, if the system crashes after only partial data are written to NVM, this leaves the system in the inconsistent state. To avoid the inconsistency, atomic durability is required on the transactional updates of data structures in persistent memory.

To support atomic updates in persistent memory systems, logging mechanisms such as write-ahead logging (WAL) have been proposed in prior studies [7], [8], [17], [21], [31]. The basic principle of write-ahead logging is to perform stores in two phases: `log-write` and `data-update`. These two writes must be serialized. To enforce the ordering, software-based logging schemes use cacheline flush (e.g., `clflushopt` and `clwb`) and store fence (e.g., `sfence`) instructions [8], [12], [16], [17], [21], [28], [29], [31]. The programmer is responsible for manually adding these instructions between log creation and data updates. Figure 1a illustrates an example source code of logging. Unfortunately, software logging approaches incur not only heavy burdens on programmers, but also a significant performance degradation due to the coarse-grained ordering [7], [24], [28], [31]. We observed between 1.57x and 2.7x throughput degradation of SW-logged versions of benchmarks compared to HW-assisted logging, which we will elaborate in this paper. Considering the advantages of the HW-assisted logging, this paper is focused on improving the HW-assisted logging mechanism.

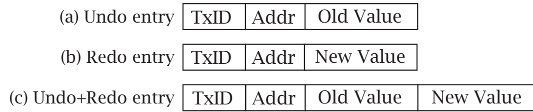


Fig. 2: Log entries of different log types.

B. Persistency Model and Assumptions

Similar to the previous studies [15], [17], [24], [28], [31], we assume the atomic durability in persistent memory is based on durable *transactions*. Within a durable transaction, all stores are committed in an all or nothing manner. In this paper, we assume a fully hardware-controlled logging that both creates and writes the log entry and persists data after logs are written. Programmers only annotate transaction boundaries using `TxBegin` and `TxEnd` interfaces, but do not need to explicitly create log entries and add cache flushing and ordering instructions. Figure 1b shows an example code of the HW-logging approaches.

In this paper, the transaction only guarantees the *atomic durability* of data updates. We assume that isolation between transactions, where two parallel transactions have conflicting accesses to the shared data, is supported by software mechanisms such as fine-grained locking [6]. In addition, nested transactions are not supported, as they are orthogonal to atomic durability, and is left for future work.

III. Hardware-assisted Logging

To overcome the overhead of software-based logging, recent studies have proposed HW-assisted logging techniques [10], [15], [24], [28]. In the hardware-assisted logging, logs are uncacheable to make them arrive in the store-order to the log area in NVM. In addition, a hardware component tracks dependency between log and data writes, which enables the fine-grained ordering without memory fence instructions, providing significantly better performance.

In this section, we first explore the design space of log and data writes in hardware-assisted logging, in Section III-A and Section III-B, respectively. Then, we discuss the advantages and disadvantages of previous approaches in Section III-C

A. Taxonomy of Log-Write

We first generalize the various design directions on `log writes` in two aspects. The first one is about the content of logs, and the second one is about optimizations on storing log entries.

1) What to log: Logging can be categorized to `undo`, `redo`, and `undo+redo` based on what is logged. Figure 2 shows a log entry of each log type. First, the undo approach preserves a copy of *old* data. On system failures, it reverts to the system state as if the transaction has not been executed. On the other hand, the redo mechanism creates log entries with the *new* versions of data. It recovers from crashes by either discarding uncompleted logs or committing new versions from logs to the NVM in-place data. Lastly, the undo+redo technique is more flexible than the others since it stores *both* the old and new versions in the log.

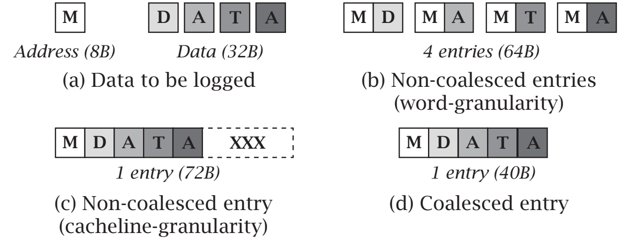


Fig. 3: Number of log entries are reduced with coalescing. Log entries are cacheblock-aligned.

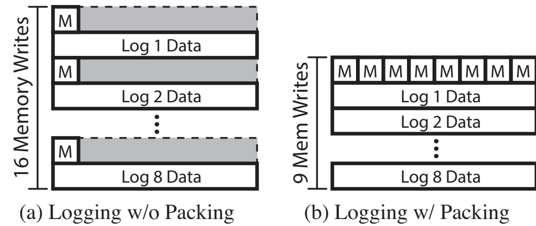


Fig. 4: Number of writes generated by logging requests with and without packing. The `M` blocks represent `Meta Data`, the address. The shaded part is wasted due to unnecessary writes.

2) Granularity and Optimizations: In addition to log types, the granularity of the log entry has a great impact on logging performance. A small granularity, i.e., *word*, might not be optimal when large data are updated in transactions. It suffers from the inefficiency storing a large number of small log entries, which is not space-efficient since about a half of log space is used for the address and meta data. On the other hand, logging with the *cacheline*-granularity may potentially waste the NVM log area and write bandwidth for storing unmodified data, if only a small portion of a cacheline is modified. Therefore, the fixed granularity does not always provide the optimal performance.

Instead of the fixed granularity, the first optimization, *coalescing*, supports *variable* granularities, providing much higher efficiency under different application behaviors. The optimization *coalesces* log entries that contains data of *contiguous* addresses and the same transaction ID into a single entry. It reduces the relative portion of meta data (e.g., address) in the log space and the number of log entries. For example, Figure 3 compares the size of log entry when logging 32B of data. The fixed granularity causes duplicate addresses for the word-sized log (b) and unnecessary log writes for unmodified data portion with the cacheline-sized log (c). On the other hand, the coalesced entry contains the exact data updated in the log, while the address portion is minimized. This optimization can be realized by placing a write-combining buffer in the processor [24], [31], [34]. Note that the log entry is coalesced up to the cacheline size (64B).

The second optimization is to reduce the number of NVM writes required for storing log entries [15]. If the size of a log entry is greater than the burst size of the memory bus, two NVM writes are required to store the log entry. Instead

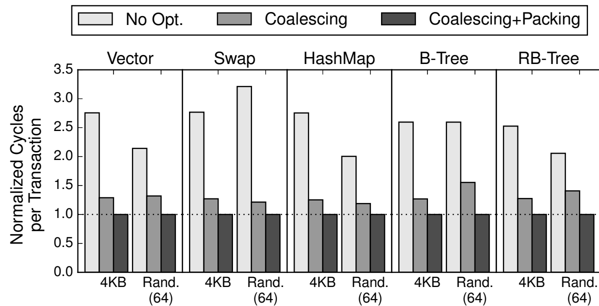


Fig. 5: Cycles per transaction (CPT) reduction by optimizing log writes. Values are normalized to when both coalescing and packing are used.

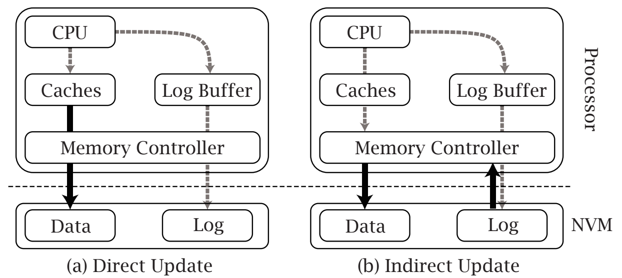


Fig. 6: Comparison of direct / indirect properties of data-update in hardware-assisted logging.

of issuing an individual log entry, the *packing* optimization co-locates multiple log entries in one large log record and reduces the number of NVM writes. Figure 4 describes the concept of the packing optimization. The log record consist of multiple log entries, for example, eight entries in the figure. Once the log entry arrives at the NVM controller, it sends data immediately to NVM while keeping the meta data (addresses) in a separate buffer. When the address buffer is filled up (eight entries) or the transaction commits, the NVM controller flushes the meta data buffer in a single burst.

Observation: While the undo and redo approaches create the same size of logs, the undo+redo approach doubles the log data sizes. As the size of logs increases, more time is required to store logs, which is on the critical path of transaction execution. In terms of log write overheads, the undo+redo log has a drawback over the other two approaches, although it has advantages in data updates as will be discussed in the next section.

Optimizations such as coalescing and packing are equally applicable to all logging approaches. The differences in transaction throughput of each approach with and without such optimizations are significant. As an example, Figure 5 shows a breakdown of throughput in the undo+redo logging with and without the two log optimizations. The label, `Rand.(64)`, indicates that the transaction modifies a word from 64 random items. Refer to Section V-A for the details of the evaluation setup. For both sequential and random write patterns, coalescing and packing result in a substantial reduction of execution cycles. The undo and redo mechanisms also exhibit similar performance improvements with the two optimizations.

B. Taxonomy of Data-Update

Following log writes, the data are updated in-place in NVM. Based on how to write the modified data of a transaction to NVM, we classify the possible approaches into three aspects: 1) synchronous or asynchronous, 2) direct or indirect, and 3) SW- or HW-initiated cacheline flush.

1) Synchronous / Asynchronous: The first property is related to when to in-place update modified data on NVM. The *synchronous* update writes the modified data before the transaction commit. On the other hand, the *asynchronous* update can defer the in-place writing of data to after the transaction commit. Note that the undo log approach only allows the synchronous update model. The undo log entry only contains the old data, and thus system failures cause data loss if the modified data are not written to NVM before the transaction commit. Without synchronous updates, there is no guarantee when new versions of data will persist in NVM.

There is a trade-off between synchronous and asynchronous updates. The synchronous update places cache flush operations on the critical path. The transaction must wait until the entire flushing operations complete, resulting in a longer critical path in transaction execution. In contrast, the asynchronous update allows transactions to finish without updating data in-place. The asynchronous update can make better use of the NVM bandwidth by parallelizing data updates of the committed transaction and log writes of the next transaction. For improving transaction throughputs, the asynchronous update has more opportunities to better utilize NVM bandwidth and to reduce the critical path length. However, the asynchronous update increases the complexity of cache modifications since caches need to track the write-sets of previous transactions.

2) Direct / Indirect: This property determines which component provides the updated data during in-place updates in NVM. Figure 6 illustrates the direct and indirect data updates. In the *direct* update, new versions of data are directly read from the caches and written-back to NVM. To flush only modified cachelines, it requires to track the write-set of each transaction, which we will discuss in the section below.

On the other hand, in the *indirect* update, new values in logs are read to update in-place data in NVM. It does not require a modification on the cache hierarchy since all information to update in-place data are stored in log entries. However, the indirect update has a fundamental limitation. Since data are fetched from logs, the indirect update requires additional NVM read operations. Therefore, it consumes more NVM bandwidth than the direct update, which does not waste bandwidth for reading logs. Note that only redo or undo+redo log can be combined with the indirect data updates since new versions are stored in logs.

3) SW flush / HW flush: To update in-place data, cache-flush operations are required to flush new values in the cache to NVM. There are two ways of flushing cachelines. The first one is to get aids from software by instrumenting cache-flush

Previous Work	Log-Write			Data-Update			Limitation
	Type	Granularity	Optimization	From where	When	Flush	
ATOM [15]	undo	cacheline	packing	direct	synchronous to NVM	SW flush	Long critical path
Proteus [28]	undo	32B	none	direct	synchronous to NVM	SW flush	Long critical path
Wrap [10]	redo	word	none	indirect	asynchronous to NVM	N/A	Requires extra NVM reads
FWB [24]	undo+redo	word	coalescing	direct	asynchronous to NVM	HW flush	Demands more log writes
Our Design	redo	word	coalescing +packing	direct	synchronous to DRAM asynchronous to NVM	HW flush	Consumes DRAM capacity

TABLE I: Summary and comparison of previous approaches on hardware-assisted logging with our design.

instructions such as clwb before the transaction ends. Another option is to let the hardware track cachelines modified in a transaction and flush them when the transaction commits. The software-aided cache-flush not only imposes the responsibility of recording modified addresses and instrumenting flushing instructions on software, but also causes a coarse-grained ordering by sfence. However, while the hardware-assisted flushing requires the hardware to record cachelines updated in the transaction, it removes unnecessary stalls caused by memory barrier instructions.

Observation: Based on the logging type, the possible design spaces are narrowed. For example, in the undo approach, the data updates must have the direct and synchronous properties. However, the synchronous update forces a transaction to wait until all the updated data are written to NVM before the commit. On the other hand, the redo and undo+redo approaches can work with any combination of (in)direct and (a)synchronous properties.

The direct and asynchronous properties with HW-initiated cache-flush achieves the best performance in transaction throughput. Nevertheless, none of prior studies are designed with these properties except the undo+redo scheme [24]. However, it has a shortcoming of larger logs. We discuss advantages and disadvantages of previous approaches in the next section.

C. *Prior Approaches*

In this section, we revisit the approaches taken in the previous studies and discuss trade-offs in their design choices. Table I summarizes the prior work with their characteristics.

1) Undo Approaches: The prior hardware undo approaches commonly use the fixed granularity of logging with cacheline or 32B data size [15], [28]. The fixed granularity can incur inefficiency in log writes, since it either stores unmodified data as well in logs or generates a large number of log entries, as discussed in Section III-A2. Note that in undo logging approaches, data-update must be done *directly* and *synchronously*. The two prior studies, ATOM and Proteus, also employ the direct and synchronous update. In addition, the two approaches flush by programmer-inserted [15] or compiler-inserted [28] cache-flush instructions.

Another design optimization of undo approaches, which has yet to be explored, can be to use hardware-initiated cache flushing. However, undo approaches have an inherent limitation of the longer critical path due to the synchronous update model.

2) Redo Approaches: The interesting design option used in the previous redo approach is the way it performs in-place data updates [10]. It updates data in NVM in-place, in the background, by reading log entries containing new versions of data. This process is called *log retire*. Although the NVM controller operates the log retire in background non-intrusively, it consumes an additional NVM bandwidth for re-fetching log entries from the NVM log space. The bandwidth consumption of the log retires is reported as a bottleneck on performing write-intensive transactions [15]. The indirect commit can potentially limit the throughput of transaction execution.

3) Undo+Redo Approaches: The recent work has presented the undo+redo approach on hardware-assisted logging [24]. The novelty of the undo+redo design resides in how to perform in-place data update directly and asynchronously. They added status bits on each cacheline to track whether it is updated or not by transactions. They designed a state machine that periodically scans and flushes dirty cachelines to NVM, called forced write-back (FWB). In addition, the undo+redo approach provides higher flexibility than the other two approaches.

While the previous undo+redo approach provides an excellent design option for data updates and flexibility, it has a limitation in what they log. Since both old and new versions of data are logged, it increases the amount of log writes to NVM.

IV. ARCHITECTURE

A. *Design*

Our logging scheme is designed to fulfill the following design goals. First, it should include optimizations on log writes such as coalescing and packing to enable efficient variable-granularity logging and to log only the *exact* data updated. Second, for efficient in-place data updates in NVM, data updates to NVM must be performed *asynchronously* and *directly*. The last design goal is that cache flush operations are initiated by hardware instead of cache-flush instructions followed by a memory barrier.

In this paper, we propose a redo approach that pursues the asynchronous and direct updates to NVM, called ReDU. We summarize our design approaches and enabling techniques that realize each approach:

- We exploit a small region of DRAM as a write cache for NVM writes. Cachelines modified by a transaction are flushed from the cache hierarchy only to DRAM Cache when the transaction commits. In-place data are updated by new values stored in the DRAM cache, unlike the prior

redo, which reads new data from the logs. The use of DRAM cache enables the *direct* update property, and the redo principle allows *asynchronous* in-place data updates to NVM.
- To enable *hardware-initiated cacheline flush*, it is required to track the `write-set by hardware`. The design adds one bit for each cacheline *only* in the private L1 cache and flushes modified cachelines.
- To achieve the *variable-granularity* in logging, the new logging scheme employs optimizations such as `coalescing` and `packing`.

Figure 7 shows the architecture of `ReDU`, with the shaded components added for `ReDU`.

B. The use of DRAM as Write-Cache

A DRAM buffer has been proposed to address the *early-update* problem in the prior redo approach [10]. The early-update problem occurs only in redo logging, where logs only contain the new data. In undo or undo-redo logs, cachelines that contains new values can be safely evicted to NVM, directly updating the in-place data, as long as the log is persisted before the data. Note that such cache evictions can occur in any order regardless of the transaction commit order. In the prior redo logging, the in-place data are updated from the new values in the log, and the evicted cachelines do not directly update the NVM data. Without any buffering, the new values of the evicted cachelines exist only in the logs. The critical overhead caused by the early update problem is that any subsequent reads must search the redo logs as new values may exist in the logs. To mitigate the overhead, the evicted cachelines are temporarily stored in a DRAM buffer until the corresponding log entry retires, and subsequent cache misses search the DRAM buffer, instead of searching the logs.

However, the volatile buffer proposed in the previous work has following limitations. First, it discards the data stored in the buffer. New versions of data must be fetched from logs for in-place data updates. We extend the idea of using a volatile victim buffer as a write-back cache to update in-place NVM data directly with new values in the DRAM buffer. This improvement eliminates the extra NVM reads for the log entries during the log retire process. The second limitation of the prior redo approach is its effect on the NVM read latency. Since new versions may reside in the victim buffer, each cache miss must lookup the DRAM buffer first, before accessing the NVM, increasing read latencies. This indirection incurs at least two DRAM reads for checking the DRAM buffer, in each NVM access. However, the actual chance to find the updated values in the DRAM cache can be very low. Exploiting the low probability of hits on the DRAM buffer, we propose a new filtering-based approach to mitigate the overheads.

C. Direct-Update with the DRAM Cache

Supporting direct and asynchronous in-place data updates is non-trivial. The hardware needs to maintain whole write-sets of both committed and currently in progress transactions. Otherwise, it needs to check the log space whether the

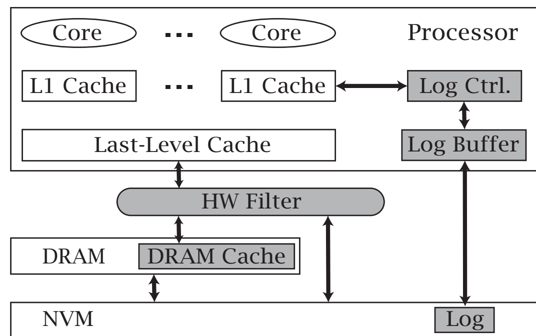


Fig. 7: `ReDU` architecture.

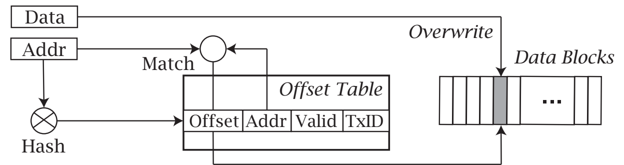


Fig. 8: New data overwrites existing one if the address matches in the offset table.

address of evicted cacheline matches one in the logs whenever cachelines are evicted. If the address is found in the log, the log entry is marked to be removed. However, both impose significant modifications on existing hardware and require non negligible performance overheads.

The previous redo approach avoids this challenge by performing *indirect* in-place data update [10]. The indirect redo logging scans the log area and is able to identify whether new values are permitted to be updated to in-place data based on log entries. In addition, the previous undo+redo approach addresses this challenge through substantial hardware extensions on the cache hierarchy [24].

Instead of either inefficient indirect updates or complex hardware extensions, when a transaction commits, `ReDU` *synchronously* flushes the modified cachelines only to the fast DRAM cache. Data updates to slow NVM are done *asynchronously* by write-back from the DRAM cache (explained Section IV-E). Figure 7 shows the architecture of `ReDU`. The synchronous updates to the DRAM cache efficiently reduce the overheads that would have been imposed with the asynchronous updates. The use of DRAM replaces slow NVM writes with fast DRAM writes.

To record the write-set of the transaction, we add a `transaction` bit for each cacheline in the *private* L1 cache. If the cacheline with the transaction bit is evicted to lower level caches (e.g., L2 and L3), it is also flushed to the DRAM cache. Note that this early update to the DRAM cache does not affect the correctness. When the transaction commits, the cache controller scans all cachelines in the private caches and flushes them if the transaction bit is set.

D. The DRAM Cache Organization

1) Transaction Management: The DRAM cache stores cachelines flushed from the cache hierarchy. There are two

cases when the DRAM cache accepts new data. First, cachelines are explicitly flushed when the transaction commits. They can be written to NVM in an arbitrary order since their logs have already become persistent. Second, cachelines are evicted by the replacement policy before the transaction commits. These cachelines must not be flushed to NVM until the corresponding transaction commits. To distinguish these two cases, cachelines stored in the DRAM cache need to be associated with its transaction semantic, indicating whether the transaction is committed or not.

To manage incoming and outgoing data from the DRAM cache as well as logs in NVM, two tables are maintained: the offset table and transaction table.

Offset table translates the physical address of incoming data to the relative offset within the DRAM cache where they are written. Each entry in the offset table consists of the original physical address (48-bit), the transaction ID (15-bit), and one valid-bit. The DRAM cache offset is fixed for each offset table entry. When a new cacheline is flushed from the cache, the hash function returns an index of the offset bucket using its address. Each bucket (64B) contains up to eight addresses. If the incoming address matches one in the table, meaning that the old copy exists in the DRAM cache, the incoming cacheline overwrites the existing data, as shown in Figure 8. If the address does not match any entries, a new entry is allocated and replaces an existing one. The victim entry is written to NVM. The replacement policy is explained in Section IV-E.

Transaction table (TT) maintains the information related to transactions. The TT entry includes the transaction ID (TxID), the start and last log addresses, and the block count that indicates the number of cachelines associated with the transaction ID stored in the DRAM cache. On the *TxBegin*, a new table entry with the TxID and current global log address is created. When a new cacheline arrives at the DRAM cache, the block count of the TT entry with the transaction ID is incremented by one. On flushing from the DRAM cache, the count will be decremented. On the transaction commit, notified by the *TxEnd*, the last log offset field of the TT entry is set with the current log offset, marking the end of the transaction. A transaction is completely retired from the DRAM cache, if its updated data are fully reflected to NVM. The TT entry of the committed and retired transaction has the last log offset value as a mark for commit, and the block count must be zero, which indicates all cachelines belonging to the transaction are already updated to NVM.

2) The Size of the DRAM Cache: A small region of DRAM is reserved for the DRAM cache. It does not require a dedicated DIMM or stacked memory. The size of the DRAM cache is configurable during the system booting time. For storing two tables described above, it requires one fourth of the data block size. For example, if the DRAM cache size of 32MB, the system requires 8MB more DRAM capacity for the offset table and transaction table. In our evaluation, we set the size of data blocks in the DRAM cache to 32MB.

In case of overflow, which can happen when the working set of the outstanding transactions exceeds the size of the CPU caches and the DRAM cache, the transaction has to abort and fall back to the software path, provided by the programmers. It is worthy of noting that the overflow is extremely rare since most transactions in in-memory key-value stores and other persistent data structures have less than KBs of footprints.

E. In-place Data Update

In-place data in NVM are updated through write-backs from the DRAM cache. The write-back policy determines when to flush data to NVM. Note that cachelines in the DRAM cache can be written-back to NVM *at anytime* and *in any order* once their corresponding transactions commit.

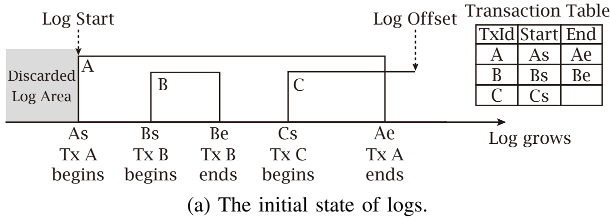
The write-back policy can be configured in different ways. For example, the DRAM cache buffers data for a certain temporal (e.g., in seconds) and spatial (e.g., in bytes) thresholds. This is similar to the page cache of Linux kernel. This paper investigates two write-back polices: eager write-back and least-recently used (LRU) policy. The eager write-back policy flushes cachelines to NVM as soon as they arrive. The eager policy minimizes the use of the DRAM cache, reducing the cost of the filtering mechanism discussed in the next section. On the other hand, the LRU policy holds the updated data in the DRAM cache until they are evicted by the LRU policy when the DRAM cache is full. The LRU policy has a higher chance for merging updates in the DRAM cache, and providing the data requested by read operations directly from the DRAM cache.

F. NVM Read Latency

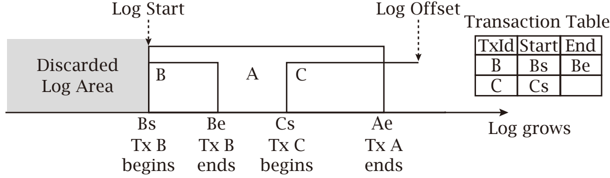
A NVM read request needs to search the DRAM cache first since it may hold a new version of the data. In case of the DRAM cache hit, the NVM read request is served from the DRAM cache without accessing NVM and by two DRAM reads, one for the offset table and another for data. If not found, the response is delayed by the DRAM cache lookup latency.

To mitigate the overheads of the DRAM lookup latency, `ReDU` uses a hardware-based filter to avoid unnecessary DRAM or NVM accesses. We design different HW filters for two write-back policies. Since the eager policy flushes cachelines immediately and the DRAM cache hits are infrequent, a small filter is suitable. Therefore, the HW filter for the eager policy is implemented as a counting bloom filter and consists of 1K entries with 4-bit for each entry. The total hardware cost is 512B of SRAM. For any data stored in the DRAM cache, the counting filter can accurately report their hits in the DRAM cache. However, it may have false-positives for non-cached data, reporting their hit status erroneously. If the filter returns a DRAM cache hit, the NVM read request checks the DRAM cache first. Otherwise, it skips the DRAM lookup and fetches data directly from NVM. With uncommon false-positives, the DRAM cache miss can be found after the DRAM cache lookup, and a subsequent NVM access is initiated sequentially. During an insertion to or eviction from the DRAM cache, the corresponding bloom filter entry is incremented or decremented.

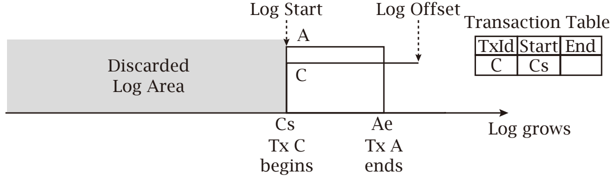
On the other hand, in the LRU policy, where the DRAM cache hits are more common than the eager policy, implement-



(a) The initial state of logs.



(b) After logs of the transaction B are removed.



(c) After logs of the transaction A are removed.

Fig. 9: Example of the log removal.

ing a counting filter requires large capacity overheads. Instead of the counting bloom filter, `ReDU` uses a non-counting bloom filter which can have both false positives and negatives. If the bloom filter returns a possible membership in the DRAM cache, read requests first access the DRAM cache. If the DRAM cache does not contain the data, NVM is accessed sequentially. If the filter reports a miss, the read request checks the DRAM cache and accesses NVM simultaneously. Such simultaneous accesses can waste DRAM or NVM bandwidth and consume more energy. The non-counting bloom filter for the LRU policy consists of 32K entries with 1-bit for each entry. The total hardware cost is 4KB of SRAM. The non-counting bloom filter resets if the false-positive rate exceeds the threshold (e.g., 50%).

G. Discussion

1) Log Management: We assume that there is a global log area in NVM shared by all processes and only accessible by the memory controller. There are two special registers used to manage log areas. All log entries are sequentially appended in the global log area, from the address where `Log Offset` points to. The `Log Start` indicates the starting address of the *valid* log space. When the logs are removed, a new address is written to the `Log Start` register.

Log entries are removable only when corresponding cachelines stored in the DRAM cache are written-back (flushed) to NVM. However, it is non-trivial to scan the log area and remove the log entry on every write-back since the log entries of different transactions are interleaved in the shared log area. Instead, we propose to remove a range of logs, which are sequentially allocated, at once.

Processor	2GHz, x86, out-or-order
L1 I/D Cache	Private 32KB, 8-way
L2 Cache	Private 256KB, 8-way
L3 Cache	Shared 8MB, 16-way
DRAM	DDR3-1600 (800MHz), 4GB Single channel
tRCD-tRP-tRAS-tWR-tWTR-tRTP-tRRD-tXAW	
13.75-13.75-35–15-7.5-7.5-30(ns)	
NVM	Single Channel Read = 50ns Write = 150ns

TABLE II: Simulation system configuration.

For example, Figure 9a illustrates the log space when two transactions, A and B, are already finished while transaction C is in progress. We omit the block count field in the transaction table for simplicity. Suppose that all the cachelines belonging to transaction B are flushed. Although the log entries of transaction B are now removable, we do not delete them immediately, but remove an entry from the transaction table, as shown in Figure 9b. When the logs of transaction A become removable, we move the *Log Start* pointer to the new start address, *Cs*, shown in Figure 9c. Even if the log entries of the already-retired transaction A remains in the log area, it does not corrupt the recovery process.

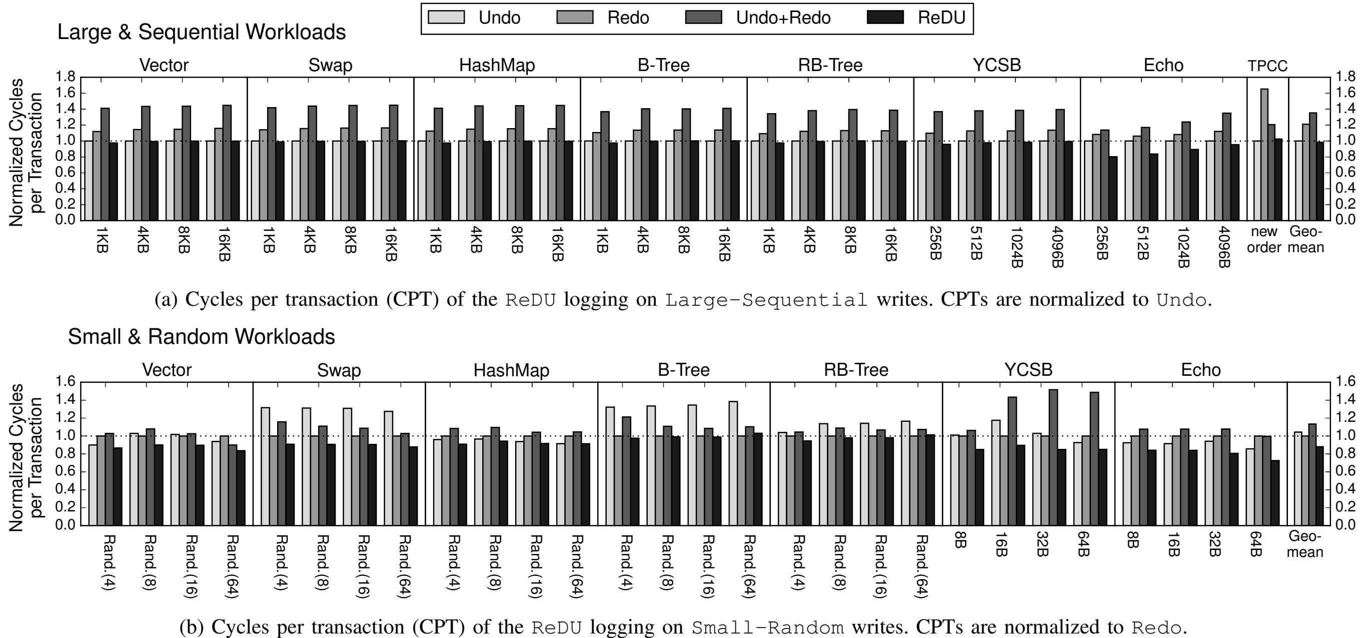
2) Recovery: The recovery mechanism after system failures is the same as the prior redo logging. The log entries of uncommitted transactions are discarded while those of committed transactions are copied back to data. We identify whether the transaction commits or not by the transaction commit record in the log area. The `Log Start` register is stored in the predefined address of NVM, and the recovery software begins by reading the base address. The recovery software scans the log area and copies or discards log entries based on whether the transaction is committed or not.

3) Flexibility: In terms of the traditional flexibility of when to update in-place data, the undo+redo scheme is more flexible than our scheme. However, we can effectively mitigate the limitation from inflexibility by fast synchronous writes to the DRAM cache, backed by asynchronous direct updates to NVM. Uncommitted data already evicted to the DRAM cache overwrite the existing data. This allows to mimic the steal property of the undo and undo+redo schemes by overwriting uncommitted data to the DRAM cache.

V. Evaluation

A. Methodology

We implemented and evaluated `ReDU` and the other logging designs using the SE mode of gem5 simulator [5]. We configure the simulator to model a multi-core out-of-order processor with both DRAM and NVM memory systems. The detailed system configurations are listed in Table II. To model NVM performance, we set NVM read latency to 50ns and write latency to 150ns, in line with previous research [15], [20], [28], [29], [32] and the NVM simulator [27]. We also evaluate on much slower NVM latencies in Section V-E.



(a) Cycles per transaction (CPT) of the `ReDU` logging on `Large-Sequential` writes. CPTs are normalized to `Undo`.

(b) Cycles per transaction (CPT) of the `ReDU` logging on `Small-Random` writes. CPTs are normalized to `Redo`.

Fig. 10: Transaction performance of the `ReDU` logging and other logging designs.

Microbench	Vector	Insert/update entries in vector
	Swap	Swap random two entries in vector
NVML [12]	HashMap	Insert/update entries in hash table
	B-Tree	Insert/update nodes in b-tree
	RB-Tree	Insert/update nodes in red-black tree
Macrobench [23]	YCSB [2]	20%/80% of inserts/updates
	TPCC [2]	New order transactions
	ECHO [4]	Insert/update on persistent hash table

TABLE III: List of benchmarks used in our evaluation.

Table III presents the benchmarks evaluated in this paper. The `vector` benchmark appends new persistent object at the end of the array, and the `swap` benchmarks swaps random two entries in the array. For macro-benchmarks, we use implementations of `HashMap`, `B-Tree`, and `RB-Tree` provided in the NVML library [12]. In addition, `YCSB`, `TPCC`, and `ECHO` from WHISPER benchmarks [23] are modified to use pmalloc/pfree interfaces instead of a mmap interface.

We evaluate these benchmarks using various data sets that exhibit different access patterns within transactions. Data sets can be classified to two groups: `Large` and `Random`. Object sizes in the `Large` data set range from 1KB to 16KB while the `Random` sets modify multiple small objects. For example, `Rand.(n)` indicates a workload that accesses and updates a word in *n* different objects indexed by *n* different keys. For TPCC benchmark, we use the `New Order` transaction workload.

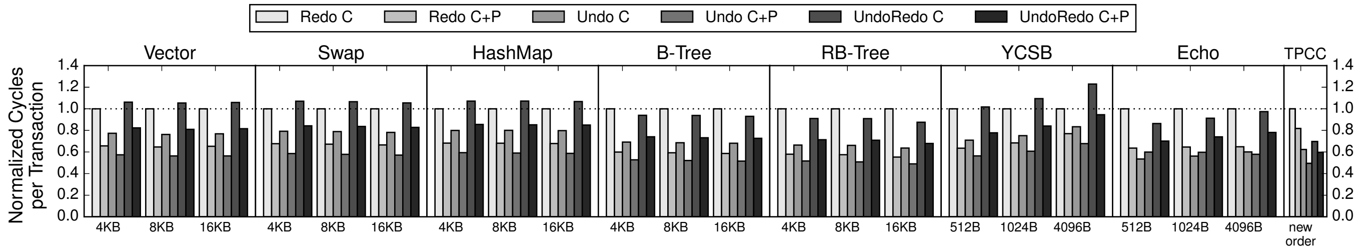
For measuring the transaction throughput, all workloads contain only a `Put` operation that allocates and writes new value in NVM. We also evaluated the `Get`-only workload in Section V-D. We ran eight copies of benchmarks when evaluating micro benchmarks while four copies for NVM and macro benchmarks.

We compare the following designs. For fair comparison, all designs equally include log optimizations such as coalescing and packing.

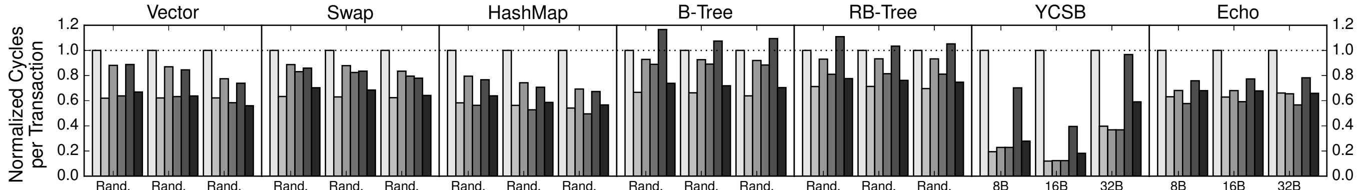
- **Undo:** the undo implementation based on the previous work [15]. Posted and source log optimizations, which are originally proposed in the previous work, are also included. This implementation represents a highly optimized *undo* approach with the *direct* and *synchronous* data-update property.
- **Redo:** the redo implementation similar to the previous study [10], which performs the in-place data update by re-fetching log entries from NVM. This design is a representative of the *redo* logging with the *indirect* and *asynchronous* update property.
- **Undo+Redo:** the undo+redo implementation proposed in the previous work [24]. The forced write-backs are triggered in every five million cycles. This method represents the *undo+redo* log with the *direct* and *asynchronous* updates.
- **`ReDU`:** the proposed scheme with the *direct* and *asynchronous* data updates on the *redo*-based logging. The eager write-back policy (Eager) is used unless otherwise mentioned.

B. Transaction Performance

Figure 10 compares the transaction throughput of each design. Throughput is represented in cycles per transaction, where lower values equate to better throughput. The new order transaction of TPCC exhibits large and sequential update patterns, so it is only shown in Figure 10a. We use the eager write-back policy in this evaluation for comparing only durability costs of



(a) Normalized cycles per transaction (CPT) of each hardware logging technique on `large and sequential` writes.



(b) Normalized cycles per transaction (CPT) of each hardware logging technique on `small and random` writes.

Fig. 11: Normalized cycles per transaction (CPT) of different hardware logging techniques.

each loggings without benefits from read-/write-caching in the DRAM cache. `ReDU` gains performance benefits from caching in the DRAM cache while the previous designs cannot. We evaluate the effect of caching in the LRU policy in Section V-F.

Figure 10a shows that `ReDU` outperforms `Redo` and `Undo+Redo` on the large and sequential update patterns while it shows similar performance compared to `Undo`. The reason of lower performance of `Redo` is because the log retire demands NVM bandwidth that would be used to storing logs or data otherwise. In `Undo+Redo`, the performance is significantly degraded because it needs to write twice the amount of logs compared to others. With the eager write-back policy, `ReDU` and `Undo` show similar throughput on the large and sequential writes since they perform the same amount of NVM reads and writes. Interestingly, there are some cases when `ReDU` shows better throughput than `Undo`, e.g., Echo benchmark. We found that this is because frequently updated data are merged in the DRAM cache, resulting in less writebacks required on data updates from the DRAM cache to NVM.

Figure 10b compares performances of each design under small and random writes. `ReDU` and `Redo` exhibit similar throughput since the overheads of indirect updates in `Redo` are hidden in these workloads. However, we observed that `ReDU` does outperform `Redo` and `Undo+Redo` in benchmarks such as YCSB and Echo. Since these benchmarks have a short-term temporal locality, updates are merged before they are flushed to NVM. On the other hand, `Undo`, which has to commit synchronously, shows the worst performance in small and random workloads.

The right most plots of Figure 10a and Figure 10b show geomean values of CPTs. For the large and sequential workloads, where `ReDU` and `Undo` show comparable performance, `Redo` and `Undo+Redo` approaches exhibit performance degradation over `ReDU` about 21.2% and 35.4%, respectively. For the small and random write patterns, `ReDU` outperforms by 16.7%, 12.1%, and 21.7% over `Undo`, `Redo`, and `Undo+Redo`, respectively. `Undo` suffers from the synchronous update property. `Redo` has performance degradation from the log retire. Lastly, `Undo+Redo` needs to write twice the amount of the log. However, `ReDU` does not have these limitations and shows better performance. Overall, `ReDU` shows 8.6%, 14.2%, and 23.6% improvements on the transaction throughput over the prior `Undo`, `Redo`, and `Undo+Redo` approaches which are intensively optimized.

		No Opt.	Coalesce	Coalesce+Packing
Large Tx	Log Writes	1	1	0.71
	Log Sizes	1	0.56	0.56
Random Tx	Log Writes	1	1	0.73
	Log Sizes	1	0.78	0.68

TABLE IV: Reduction of the number of log writes and log sizes due to log optimizations. Normalized to the one without any optimizations.

C. Log Optimizations

The transaction throughput of logging designs is substantially affected by log optimizations. `Redo C` and `Redo C+P` represent redo implementation with coalescing and coalescing+packing optimizations, respectively. We label the other approaches similarly. Figure 11a compares cycles per transaction of benchmarks on large and sequential workloads while Figure 11b shows performance on small and random workloads. CPTs are normalized to those of `Redo C` in both figures. We omit the results of each logging design without any log optimizations because they are not even comparable in the same plot.

According to the results, log optimizations are critical for all logging schemes to achieve higher performance. Therefore, the comparison may be misleading if the optimizations are

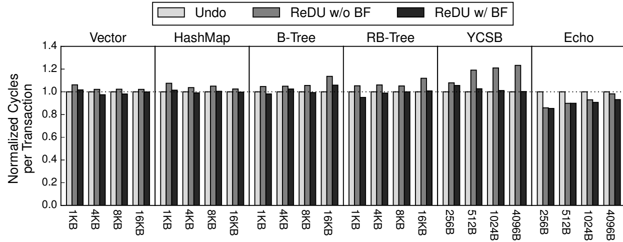


Fig. 12: Normalized cycles per transaction of `ReDU` in *Get*-only workloads.

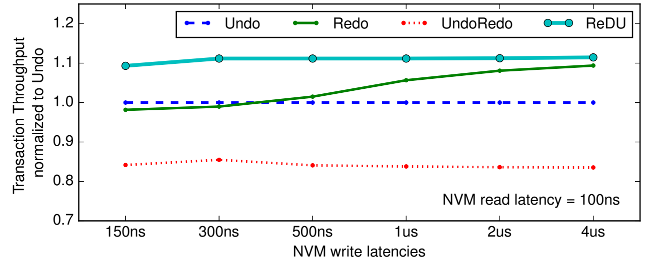
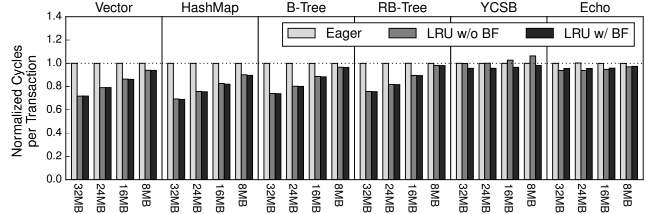
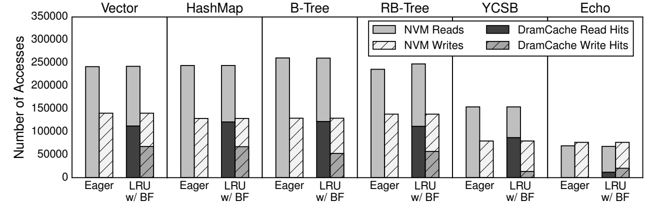


Fig. 13: Comparison of transaction throughputs of each logging schemes on different NVM latencies.



(a) Comparison of cycles per transaction of the eager and LRU policy.



(b) The ratio of the DRAM cache hits among read/write accesses to NVM.

Fig. 14: The LRU write-back policy of the DRAM Cache.

not equally attributed. For example, `Redo C+P` significantly outperforms unoptimized undo and undo+redo approaches. The overheads of the log retire are amortized by coalescing and packing optimizations that reduce the bandwidth consumption of indirect updates. Similarly, `Undo C+P` shows a far greater performance compared to the others without log optimizations.

Table IV shows the normalized number of log-write requests and log sizes of the benchmarks when each log optimization is applied. Note that the coalescing optimization reduces the number of log entries that results in small log sizes. In addition, the number of NVM log writes is reduced by 30% by the packing optimization.

In addition, no single previous logging scheme, whether it be undo or redo, performs the best for all workloads. On the large and sequential write patterns, undo series show better performance than other approaches. `Undo C+P` provides the best throughput in these workloads while redo series experience performance degradation due to the log retire and undo+redo approaches write about twice the logs compared to the others. On the other hand, for the small and random patterns, redo series perform the best. However, `ReDU`, redo-based logging with the direct and asynchronous updates, shows the best among all logging schemes in various workloads.

D. Read-Only Workloads

To measure the overheads of increased NVM read latency due to the DRAM cache lookup, we compare the transaction throughput of `ReDU` with and without the HW filter that filters unnecessary DRAM accesses. The HW filter is denoted as the Bloom Filter (BF) in the figure. `ReDU` uses the eager write-back policy in this experiment. We also show the throughput of `Undo` for quantifying the overhead to the other approaches. The results are shown in Figure 12. As expected, `ReDU` without the HW filter suffers from performance degradation up to 20% on the YCSB benchmark. Fortunately, the HW filter successfully reduces unnecessary accesses to the DRAM cache, and the transaction throughput follows similar to the one of `Undo`. `ReDU` shows better performance even without the filter for the Echo benchmark. This is because the Echo benchmark copies the value of the given key to a buffer allocated in NVM and returns the buffer to the client. Therefore, the Echo benchmark performs NVM writes as well in a get operation.

E. Sensitivity Study on Different NVM Latencies

Figure 13 illustrates how transaction throughput changes with various NVM latencies. We measured cycles per transaction of micro-benchmarks with both large and random workloads while NVM read latency is fixed to 100ns and write latencies vary from 150ns to 4us. Note that `Redo` outperforms `Undo` as the latency increases. This is because `Undo` has to suffer synchronous NVM writes for both data and logs while `Redo` performs in-place data update in background. On the other hand, `Undo+Redo` shows lower throughput than other schemes since it incurs the most NVM writes. Finally, `ReDU` always shows the best throughput among all configurations. As NVM write latency increases, `Redo` performs closely with `ReDU` since the overheads of in-direct updates due to NVM log reads become less-significant.

F. LRU Write-back Policy

We also evaluate the LRU write-back policy of the DRAM cache in Figure 14. We compare the LRU policy with and without the HW filter to the eager policy. We measured cycles per transaction while reducing the size from 32MB to 8MB. The benchmark performs either an insert/update (50%) or get (50%) operation. The access patterns are generated by the YCSB suite [9] and follow the zipfian distribution where the zipfian theta is 0.8.

Figure 14a shows cycles per transaction on each benchmark. The LRU policy outperforms the eager policy on the most of benchmarks even without the HW filter. YCSB benchmark only experiences slow down when the DRAM cache size is small (e.g., 16MB and 8MB). However, the HW filter for the LRU policy efficiently eliminates the overheads of the DRAM cache lookup latency. The performance gain by the LRU policy is up to 30.4% on `HashMap`. The gain becomes greater as the size of the DRAM cache increases.

Figure 14b decomposes the number of hits on the DRAM cache among total NVM reads or writes. The figure shows the decomposition when the DRAM cache size is 32MB. We observe that the hit rates in the DRAM cache are low even in skewed access patterns. This is because the DRAM cache only stores cachelines evicted from L1 cache, while blocks evicted from LLC and fetched from NVM are not cached. On average 43.6% of total NVM reads are cached in the DRAM cache. The LRU policy allows to respond for reads without NVM reads and merge updates which reduces the number of NVM writes.

VI. Related Work

A number of research have been proposed to support atomic updates in persistent memory [3], [7], [11], [12], [17], [21], [22], [31]. They support atomic durability through write-ahead logging implemented in software. Mnemosyne [31] supplements software transactional memory (STM) to provide atomic updates in persistent memory while using write-ahead logging for durability. Therefore, to guarantee ordering constraints between data and log, softwares should employ CPU instructions such as clwb and sfence, which resides on the critical path of program execution. To overcome drawbacks of persisting logs in the critical path, bulk or group commit [22], [26] and decoupling [21] can be used, however, they have to insert new instructions in the program. The logging of DudeTM [21] is similar to hardware redo logging proposed in Wrap [10]. DudeTM commits only when logs are written in volatile memory while Wrap completes on storing logs in NVM. Meanwhile both have to read logs from NVM to update data in-place, referred to *retire* in Wrap and *reproduce* in DudeTM. Arulraj et al. proposed Write-Behind Logging (WBL) that commits in-place data before logs [3]. WBL and `ReDU` share the key idea of merging duplicate updates to NVM in a volatile buffer. However, WBL is only applicable in a multi-versioned system, where multiple versions of data co-exist. A single-versioned system, as `ReDU` is designed for, still needs to log old copies before updates. In addition, WBL requires undo+redo logging while `ReDU` is based on redo logging. All mentioned previous studies proposed to support the atomic durability using software while our approach focuses on the hardware solution.

In addition, research into persistent data structures in persistent memory have been proposed [8], [30], [33]. For example, NV-Tree [33] proposed a persistent B+ tree structure that mitigates NVM write overheads by disordering keys in leaf nodes. However, their applicabilities is limited to a certain data structure. While NV-Heaps [8] uses logging and copying, CCDS [30] provides a versioning that copies and inserts memory fence and flush instructions. In addition, its timestamp-based multi-versioning in CCDS enables a software controlled write ordering.

Pelly et al. [25] proposed the concept of memory persistency models in terms of persist ordering constraints. In [16], the authors analyzed a persistent memory ordering and defined using Intel's CPU instructions. They decoupled volatile ordering from persist ordering and proposed hardware-modification to support the delegated persist ordering. In DCT [17], the authors found dependencies that need to be satisfied to support transactions under different persistency models. These research reduces inefficiency derived from dependencies across transactions.

Kiln [34] eliminates the logging by holding two versions of data, one in its in-place and the other version in the non-volatile caches (NVCache). The direct update in `ReDU` is done by either early cache eviction or cache flush on commit, similar to those of Kiln. However, `ReDU` differs with Kiln in incorporating the conventional DRAM with HW redo logging instead of the non-volatile caches.

VII. Conclusion

Recent studies proposed hardware-assisted logging techniques to overcome the performance overheads of current SW-based logging approaches. In this study, we first identify the design spaces of hardware-assisted logging and observe that the previous studies possess limitations on their designs.

To this end, we propose a new hardware-assisted logging design, `ReDU`, that updates in-place data directly from the cache hierarchy and asynchronously after the transaction commit. By using a small region of DRAM as a write-cache for NVM writes, our logging design successfully decouples the slow data update to NVM from the critical path. In addition, our design enables the best design choice of hardware-assisted logging, which includes optimizations of log writes and *direct & asynchronous* in-place data updates. We find that our design offers 15.4% performance improvements on average over the prior approaches.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF-2016R1A2B4014109) and by the Institute for Information & communications Technology Promotion (IITP-2017-0-00466). Both grants were funded by the Ministry of Science, ICT and future Planning (MSIP), Korea.

References

[1] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, Dec 2010.

[2] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

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

[4] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, “Exploring storage class memory with key value stores,” in *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW)*, 2013.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, pp. 1–7, 2011.

[6] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, “Atlas: Leveraging locks for non-volatile memory consistency,” in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2014.

[7] 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.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, “NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking Cloud Serving Systems with YCSB,” in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[10] 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.

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

[12] Intel, “Nvm library,” https://github.com/pmem/nvml.

[13] *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, Intel, Sep. 2016.

[14] “Intel and micron produce breakthrough memory technology,” https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/, Intel and Micron, 2015.

[15] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, “ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[16] 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 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[17] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, “High-performance transactions for persistent memories,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[18] E. Kultursay, , M. Kandemir, A. Sivasubramaniam, and O. Mutlu, “Evaluating stt-ram as an energy-efficient main memory alternative,” in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[19] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, “Phase-change technology and the future of main memory,” *IEEE Micro*, vol. 30, no. 1, pp. 143–143, Jan 2010.

[20] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting phase change memory as a scalable dram alternative,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[21] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, and J. Ren, “DudeTM: Building Durable Transactions with Decoupling for Persistent Memory,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[22] Y. Lu, J. Shu, and L. Sun, “Blurred persistence in transactional persistent memory,” in *31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[23] 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 the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[24] M. A. Ogleari, E. L. Miller, and J. Zhao, “Steal but no force: Efficient hardware undo+redo logging for persistent memory systems,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[25] S. Pelley, P. M. Chen, and T. F. Wenisch, “Memory persistency,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, 2014.

[26] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, “Storage Management in the NVRAM Era,” *Proc. VLDB Endow.*, vol. 7, no. 2, pp. 121–132, Oct. 2013.

[27] M. Poremba and Y. Xie, “NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories,” in *IEEE Computer Society Annual Symposium on VLSI*, 2012, pp. 392–397.

[28] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, “Proteus: A flexible and fast software supported hardware logging approach for nvm,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[29] S. Shin, J. Tuck, and Y. Solihin, “Hiding the long latency of persist barriers using speculative execution,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[30] 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)*, 2011.

[31] 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)*, 2011.

[32] H. Wang, J. Zhang, S. Shridhar, G. Park, M. Jung, and N. S. Kim, “Duang: Fast and lightweight page migration in asymmetric memory systems,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[33] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, “NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems,” in *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[34] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, “Kiln: Closing the performance gap between systems with and without persistence support,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.