Haste Makes Waste: Accelerating Persistent HTM with Lazy Logging

Abstract—**This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

100 words;

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Transactional Memory (TM) adopts in-memory transaction processing for concurrency control in multicore systems [9]. Multicore systems can perform tasks concurrently and thus provide better performance than single-core systems. The performance improvement relies on the parallelism of thread execution. This requires concurrency control to synchronize across threads such that they generate the same results as if they had executed on a single core. The major intricacy arises from orchestrating threads that access the same data. It is therefore modeled by TM after database transactions, which should satisfy the ACID properties.

However, traditional TM solutions based on volatile DRAM support only atomicity, consistency, and isolation without durability. Atomicity requires that a transaction appear to be having fully executed if it is successfully committed or having not executed at all otherwise. In the latter case, even if a noncommitted transaction has made modifications to certain data, its modifications cannot take effect. Consistency requires that a committed transaction cannot lead to a state of data values that can hardly be produced by any serial execution of transactions [9]. Isolation requires that the result of a transaction not be tampered with by concurrent transactions. It does not necessarily prohibit transactions from accessing the same data. Instead, transactions may be allowed to share data as long as they do not violate consistency. This requirement is also defined as serializability [1], [10], [21], [22], following which the result of concurrent transactions conforms to one obtained by their serial execution.

Emerging non-volatile memory (NVM) augments TM with durability and has brought ACID into practise for in-memory transaction processing. This is mainly attributed to its disk-like storage durability and DRAM-like access latency. Supporting durable storage, NVM enables transactional memory solutions with the durability property. The CPU no longer has to use slow I/O processes to persist data into disks. Unlike disks, NVM offers fast byte-addressability [14], achieving ACID for in-memory transactions fairly fast. The state-of-the-art marries NVM with efficient Hardware TM (HTM) and is usually referred to as Persistent HTM [12]. In comparison with Software TM (STM) that requires programmers to manually instrument memory references in programs with extra instructions [4], [13], [15], HTM reserves only the instructions for delineating transactions while pushing all other instructions for orchestrating memory references to hardware [2], [7], [12], [13], [18]–[20].

Toward efficiently fulfilling the ACID properties, Persistent HTM needs to craft two orthogonal design policies—version management and conflict management [17]. First, version management guarantees crash consistency on an intra-transaction scale. If a crash occurs, the recovered data of a transaction should be consistent in that they entirely follow either the modified version or the original version. Second, conflict management detects and resolves inter-transaction conflicts to guarantee the isolation property. A potential conflict occurs when transactions concurrently access the same data and at least one of them tries to modify it. Version management is fundamental for guaranteeing transaction correctness while conflict management matters for exploring transaction parallelism.

In this paper, we focus on version management and aim to enforce it in hardware with high efficiency. Specifically, we identify easy logging as a major efficiency bottleneck in write-ahead logging (WAL), the building block for version management in current persistent HTM. WAL maintains both old and new values of data in the memory hierarchy using dedicated logs that record either old or new data. It requires that logs be persisted into NVM prior to transaction commits. Then if the system crashes during a commit, we can use the persisted logs to recover either old or new data. WAL is widely adopted because of simplicity (e.g., in-place updates) in comparison with other alternatives. For example, shadow paging [3], [13] uses the log area directly for storing updates. Without in-place updates, subsequent accesses to the new data values should be redirected to the log area. This necessitates a redirection table that maps the address storing the old data to the address storing its new value. Further updates of the new value makes address redirection a costly, recursive operation. Albeit WAL may outperform other alternatives, we identify eager logging as its major efficiency bottleneck. It enforces an inevitable delay between when the write request arrives and when the corresponding in-place cache update takes place. All the extra operations for logging (e.g., collecting the meta-log information, fetching the old value if undo logging or the new value if redo logging from caches and writing it to the log entry) cost additional clock cycles.

To remedy persistent HTM from the efficiency bottleneck due to eager logging, we propose exploring lazy logging as a fundamental technique for acceleration.

intro...;

==We ... FlyHTM ..., lazy logging...;

performance

In summary, we make the following major contributions to accelerating persistent HTM with FlyHTM.

- asdf;
- asdf;

The rest of the paper is organized as follows. Section II reviews logging schemes for persistent HTM and identifies eager logging as their efficiency bottleneck. Section III proposes the lazy logging technique for acceleration and implements it through our faster persistent HTM called FlyHTM. Section IV demonstrates the performance of FlyHTM. Finally, Section V concludes the paper.

II. Problem

In this section, we review the basics of write-ahead logging (WAL), the building block for version management in persistent HTM. It enforces eager logging that triggers an immediate log at the L1 cache upon every write and has to persist the log before data. We identify such logging eagerness as an efficiency bottleneck and investigate how it hinders performance.

A. WAL: Write-Ahead Logging

Version management usually uses WAL to guarantee crash consistency. Logging, as the key principle of WAL, maintains both original (old) and modified (new) versions of data in the memory hierarchy. It allocates a dedicated log area in NVM for recording certain versions of data. WAL ensures that logs are persisted into NVM prior to committing a transaction. Then new data can be updated in place upon commits. Once the system crashes during a commit, we can use the persisted logs to recover either the old or the new version of the entire data set of the crashed transaction. Such data recovery guarantees data consistency in that the recovered data set is not mixed with old and new values.

According to whether the log area persists old, new, or both versions of modified data, WAL can be classified into undo logging, redo logging, and undo+redo logging, respectively.
Undo logging logs the old data and supports in-place updates prior to transaction commits. Log and data writes resemble a pipeline [20]. Once after an undo log is persisted, the corresponding write can take place over the data. The modified data can be directly written to NVM upon cache eviction. When the transaction is about to commit, some of its modified data might still be in caches. Such data need to be written to NVM as well before the transaction commits, using forced cache line write-back commands such as `clwb`. Prior to and during a transaction commit, the system may crash. If this happens, it is likely that some modified data have been written to NVM while some others are still in caches. We can use undo logs to revert modified data in NVM to their old values and guarantee consistency [13], [18], [19], [23].

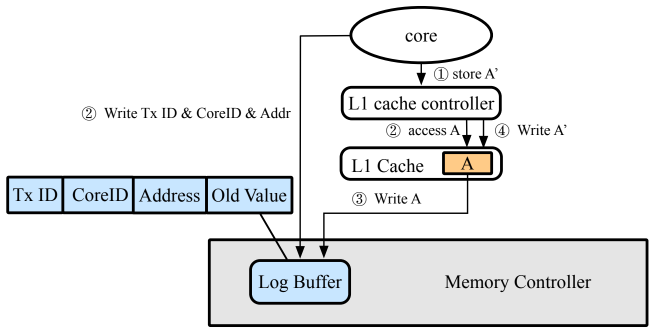

Fig. 1. Previous Hardware Undo Logging Procedure

Redo logging logs the new data and modifies data only after it persists all the corresponding redo logs. All the redo logs recording new values should be persisted prior to data writes. In other words, redo logging persists logs and writes sequentially [6], [11], [12], [15], [16], [24]. Otherwise, data consistency might be violated upon system crashes. Redo logging imposes more complexities on reads than undo logging does. Because data are not updated in caches or NVM during log creation, read accesses need to go through the log area first in case cached or in-place data are already obsolete [8].
Undo+redo logging reaps the benefits of both worlds. It logs both the old and new versions of modified data [20]. First, undo logs enable modified data to be updated in place before the transaction commits. Second, redo logs help remove the forced cache line write-back prior to commits. However, as an intrinsic limitation of redo logging, undo+redo logging still requires the forced cache line write-back upon log area overflow. Furthermore, recording both the old and new versions requires more logging time and a larger log area [11].

B. Limitation: Logging Eagerness

Albeit various WAL improvements, we identify eager logging as its efficiency bottleneck. Specifically, eager logging triggers an immediate log at the L1 cache upon every `store` instruction [11]–[13], [20]. The log needs to be persisted before data according to the WAL principle. Therefore, previous solutions often generate logs before in-place updates in caches and write logs to the memory controller by bypassing the cache hierarchy to ensure that logs run faster than data do. Figure 1 showcases the logging eagerness through a typical undo logging scheme.

- Step ①: The L1 cache controller receives a write request from the CPU core to store A'. The `store` instruction triggers the creation of an undo log including the meta-log information and old value of A.
- Step ②: The core collects the meta-log information (e.g., the transaction ID and address) and writes it to the log buffer. Meanwhile, it accesses the L1 cache to fetch the old value of A.
- Step ③: The old value of A is written directly to the log buffer. Together with the meta-log information in Step ②, it completes the log entry.

- Step ④: After the log is ready, the new value of A' is updated to the cache line.

Obviously, eager logging introduces an inevitable delay between when the write request arrives and when the corresponding in-place cache update takes place. All the extra operations for logging (e.g., collecting the meta-log information, fetching the old value if undo logging or the new value if redo logging from caches and writing it to the log entry) cost additional clock cycles.

Furthermore, eager logging may induce wasted efforts when a transaction enforces multiple writes to the same data block. For undo logging, the log needs to track the old value. The first log of each modified data block suffices for recovery. Solutions such as ATOM [13] and PiCL [19] simply add indicator bits per cache line to track whether the data therein have been logged. However, for redo logging, the last update of the modified data block dominates. Since eager logging creates a log right after a write request arrives, a log can easily become obsolete when the logged data are modified again. The more frequent a data block is modified, the more overhead is induced by such obsolete logs. In other words, eager logging leads to more potential inefficiency for write-intensive transactions.

Finally, another inefficiency arising from eager logging is due to locality unfriendliness. As shown in Figure 1, writing logs bypasses the cache hierarchy and thus ignores the temporal and spatial locality therein. To regain the locality for efficiency, some solutions temporarily buffer logs between the cache and memory controller [12], [19], [20]. Buffered logs can be coalesced into larger chunks to improve bandwidth efficiency. However, this necessitates additional buffer space and complexities for processing logs therein.

III. FlyHTM

In this section, we present lazy logging to address the inefficiency caused by eager logging in existing persistent HTM designs. It directly leverages LLC-cached data as undo logs and writes logs to the memory controller upon cache eviction. This way, we can immediately update L1-cached data without stalling the write until the log is created and written to the memory controller as in eager logging. We implement lazy logging through our faster persistent HTM called FlyHTM.

A. Motivation

We observe that a log is not immediately needed when a cache line is modified. Instead, it suffices if the log can become available in the persistent domain before data are written back. Modified data usually stay in caches and may not be written back for a while. This offers a relaxed time bound between the arrival of a write request and the writing of the log it triggers. If we revive the relaxed time bound by filling it with in-cache updates, we expect to speed up transactions. The later we delay the log creation and writing process, the faster we can complete write requests of transactions. This motivates us to explore lazy logging as a fundamental acceleration technique for persistent HTM.

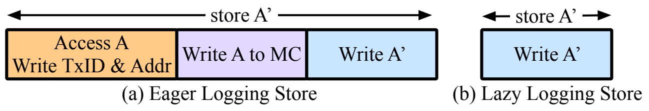

Fig. 2. Comparison the `store` instruction of eager logging and lazy logging

B. Methodology: Lazy Logging

Lazy logging accelerates persistent HTM by decoupling log creation from the arrival of write requests. It directly leverages LLC-cached data for undo logging. In commonly used inclusive caches, the LLC-cached data are a superset of the L1-cached data. When a write request tends to modify a L1-cached data block, we do not have to write the L1-cached old value back to the memory controller for undo logging. This is because the old value is also in the LLC. Even if we immediately modify it in the L1 cache, we would not miss the old value. In a nutshell, lazy logging can perform in-cache updates upon receiving write requests and prepare undo logs nearly at the same time, without interrupting consecutive writes by log creation. The old values are written back to the memory controller as undo logs when they have to be evicted from the LLC. This mitigates the overhead of eager logging to the maximal extent. Furthermore, the delayed log creation also promises more chances of log coalescing.

We now investigate more about how lazy logging accelerates `store` instructions and optimizes log coalescing.

Shortened execution of `store` instructions. Figure 2 compares the execution timeline of a `store` instruction instance under traditional eager logging and our lazy logging. Let A denote the old value and A' denote the new value to be written. Under eager logging (Figure 2(a)), the `store` instruction is tightly coupled with log creation. Log creation enforces extra operations such as writing the meta-log information (i.e., transaction ID and address of A) to the log buffer, accessing caches to fetch the old value of A, writing A to the log buffer, and merging A with the meta-log information into a log entry. Only after all these extra operations can we modify the cache line with the new value of A'. In contrast, our lazy logging eschews all these extra operations prior to in-cache updates. The `store` instruction immediately modifies A to A' in caches as if it were a non-transactional access. Meanwhile, we track the meta-log information via the cache coherence directory (Section **??**), which should be updated upon write requests anyway. We piggyback the meta-log information into the directory without much additional overhead.

Inherent support of log coalescing. Since lazy logging leverages the LLC-cached data as logs, it directly benefits from the inherent locality therein. In eager logging, previous solutions create a log immediately after a write request arrives. To coalesce logs, they usually introduce an extra buffer to temporarily hold logs such that they can be coalesced into larger blocks before being written to the memory controller or memory. In contrast, our lazy logging uses the LLC to hold logs and thus does not require an additional buffer.

Furthermore, logs stay in the LLC until the cache lines they reside have to be evicted. Each LLC cache line intrinsically contains address-adjacent data. This saves lazy logging from much extra overhead to coalesce logs for transactions with high spatial locality.

C. System Model

We use the lazy logging technique to develop a faster persistent HTM called FlyHTM. As with existing HTM solutions, the core of FlyHTM resides on the CPU chip. Figure 3 illustrates the architecture of FlyHTM. Before delving into the specific logging framework, we define the system model. We consider a multi-core system with a commonly used three-level cache hierarchy. Each core has a private L1 cache and a private L2 cache. All cores share the Last-Level Cache (LLC). The caches are inclusive and write-back with write allocate. To guarantee cache coherence and detect transaction conflicts, we use the MESI directory-based protocol [5] for tracking data states across private caches.

Motivated by Proteus [23] and ATOM [13], FlyHTM adopts a battery-backed persistent memory controller. The benefit is to persist logs directly in the memory controller without having to write them to the much slower NVM. Another motivation behind this design choice is that most logs persisted into the NVM are not really needed. HTM solutions keep logs persistent for recovering data to a consistent state when the system crashes. However, system crashes occur rarely and transactions can commit successfully more often. Therefore, writing unnecessary logs to the NVM wastes considerable memory bandwidth and induces high overhead. Yet a dilemma arises when we cannot make sure whether the system may crash before the ongoing transaction commits. In other words, we can hardly decide which logs can be omitted. Using a persistent memory controller helps to mitigate the dilemma because persisting logs into the memory controller is much faster than into the NVM. Only when logs overflow the memory controller need them be written to the NVM. Furthermore, once a transaction commits, its logs can be discarded to save space.

D. Logging Framework

Figure 3 illustrates how FlyHTM performs lazy logging stepwise. At a high level, the core executes a transaction and the modified data during transaction execution remain in private caches. Meanwhile, the undo logs contain the old values of the modified data; they remain in the LLC until the corresponding cache lines are evicted upon replacement or flushed upon transaction commits. We walk through the logging steps in Figure 3 as follows.

`Store` (Figure 3(a)):

- Step ①: The core issues a write request to the L1 controller. Instead of enforcing the L1 controller to create a log as in eager logging (Figure 1), FlyHTM directly uses the already cached old value in the LLC as the undo log. It can easily use the addresses maintained in the write set to track which data in the LLC are used as logs, without any extra operations on the LLC cache lines.
- Step ②: Since no extra log creation is required, the write request can immediately update data in the L1 cache.

`Flush` (Figure 3(b)):

- Step ①: FlyHTM flushes undo logs from the LLC to the memory controller in two cases. The first is when the cache line containing undo logs is evicted due to replacement. The second is when the transaction commits and flushes non-persisted logs prior to data. In both cases, the old value is evicted from the LLC to the memory controller while the core writes the transaction ID and address to the memory controller. All these items form a complete undo log entry.
- Step ②: To maintain the inclusion property of inclusive caches, evicting an LLC cache line triggers back invalidation on the L2 and L1 cache lines containing the same data block.
- Step ③: Specifically, we first invalidate both the cache lines in the L2 and L1 caches. Then we write the new value from the L1 cache to the memory controller, which further persists the data into the NVM. This ensures that logs are persisted prior to data, a requirement of WAL.

IV. Evaluation

In this section, we evaluate the performance of FlyHTM.

V. Conclusion

TABLE I
TABLE TYPE STYLES

Table Head	**Table Column Head**		
	Table column subhead	***Subhead***	***Subhead***
copy	More table copy[a]		

[a]Sample of a Table footnote.

References

[1] U. Aydonat and T. S. Abdelrahman, "Hardware support for relaxed concurrency control in transactional memory," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2010, pp. 15–26.

[2] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood, "Tokentm: Efficient execution of large transactions with hardware transactional memory," in *ISCA*, 2008, pp. 127–138.

[3] M. Cai, C. C. Coats, and J. Huang, "Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory," in *ISCA*, 2020.

[4] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: Why is it only a research toy?" *Queue*, vol. 6, no. 5, pp. 46–58, 2008.

[5] I. Coorporation, "Intel 64 and ia-32 architectures optimization reference manual," 2016.

[6] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," in *HPCA*, 2016, pp. 77–89.

[7] E. Giles, K. Doshi, and P. Varman, "Hardware transactional persistent memory," in *MEMSYS*, 2018, p. 190–205.

[8] S. Gupta, A. Daglis, and B. Falsafi, "Distributed logless atomic durability with persistent memory," in *MICRO 2019*, 2019, pp. 466–478.

[9] M. Herlihy, J. Eliot, and B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA*, 1993, pp. 289–300.

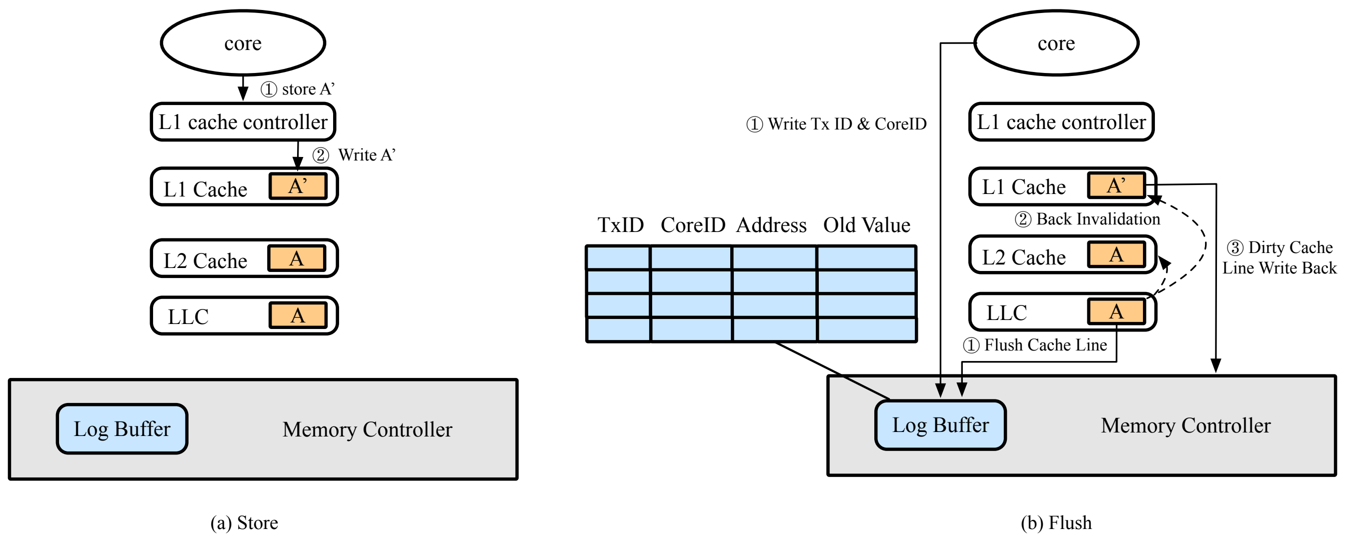

Fig. 3. FlyHTM architecture with lazy logging. related to Figure 3, need NVM unit?

Fig. 4. Example of a figure caption.

[10] S. A. R. Jafri, G. Voskuilen, and T. Vijaykumar, "Wait-n-gotm: improving htm performance by serializing cyclic dependencies," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 521–534, 2013.

[11] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *MICRO*, 2018, pp. 520–532.

[12] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: Durable hardware transactional memory," in *ISCA*, 2018, pp. 452–465.

[13] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *HPCA*, 2017, pp. 361–372.

[14] R. M. Krishnan, J. Kim, A. Mathew, X. Fu, A. Demeri, C. Min, and S. Kannan, "Durable transactional memory can scale with timestone," in *ASPLOS*, 2020, pp. 335–349.

[15] 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 *ASPLOS*, 2017.

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

[17] M. Lupon, G. Magklis, and A. González, "A dynamically adaptable hardware transactional memory," in *MICRO*, 2010, pp. 27–38.

[18] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: log-based transactional memory," in *HPCA*, 2006, pp. 254–265.

[19] T. Nguyen and D. Wentzlaff, "Picl: A software-transparent, persistent cache log for nonvolatile main memory," in *MICRO*, 2018, pp. 507–519.

[20] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+ redo logging for persistent memory systems," in *HPCA*, 2018, pp. 336–349.

[21] X. Qian, B. Sahelices, and J. Torrellas, "Omniorder: Directory-based conflict serialization of transactions," in *ISCA*, 2014, pp. 421–432.

[22] H. E. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware transactional memory for increased concurrency," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2008, pp. 246–257.

[23] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *MICRO*, 2017, pp. 178–190.

[24] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ASPLOS*, 2011.