Accelerate Hardware Logging for Efficient Crash Consistency in Persistent Memory

Zhiyuan Lu*, Jianhui Yue*, Yifu Deng*, Yifeng Zhu†
*Computer Science, Michigan Technological University, Houghton, Michigan, USA
†Electrical & Computer Engineering, University of Maine, Orono, Maine, USA
{zhlu,jyue, yifud}@mtu.edu, yifeng.zhu@maine.edu

***Abstract*—While logging has been adopted in persistent memory (PM) to support crash consistency, logging incurs severe performance overhead. This paper discovers two common factors that contribute to the inefficiency of logging: (1) load imbalance among memory banks, and (2) constraints of intra-record ordering. Overloaded memory banks may significantly prolong the waiting time of log requests targeting these banks. To address this issue, we propose a novel log entry allocation scheme (LALEA) that reshapes the traffic distribution over PM banks. In addition, the intra-record ordering between a header and its log entries decreases the degree of parallelism in log operations. We design a log metadata buffering scheme (BLOM) that eliminates the intra-record ordering constraints. These two proposed log optimizations are general and can be applied to many existing designs. We evaluate our designs using both micro-benchmarks and real PM applications. Our experimental results show that LALEA and BLOM can achieve 54.04% and 17.16% higher transaction throughput on average, compared to two state-of-the-art designs, respectively.**

***Index Terms*—Persistent Memory, Crash Consistency, Logging, ADR**

I. Introduction

Persistent memory (PM) successfully bridges the gap between memory and storage with large capacity, fast speed, non-volatility, and byte-address-ability. However, one important challenge in adopting PM into real systems is the efficiency of crash consistency, which is to guarantee the atomicity of transaction updates in the event of system crash or power loss.

Crash consistency requires that all updates within a transaction are always committed to PM in a nothing-or-all manner, even upon a system crash. Traditional systems adopt undo logging, redo logging, or a combination of both to guarantee crash consistency. With logging, a transaction update cannot be applied to in-place data until the log of its modifications has been stored in PM. However, logging methods often suffer from inferior performance due to the order constraints between logging and in-place update, which places the execution of logging in the I/O critical path. Memory barriers are often introduced to ensure the ordering constraint that a memory update cannot be performed until its log has been written to PM. Both flushing operations of dirty cache lines and memory barriers degrade the overall system performance significantly. To reduce the logging overhead, prior work [1]–[8] have been proposed to move logging operations out of the I/O critical path. However, they do not eliminate the logging overhead.

This research is supported by the NSF grant SHF-1745748 and SHF-1618536. Corresponding author is Jianhui Yue.

Our research is motivated by the observation that the inefficiency of logging is mainly caused by two factors: (1) imbalanced workload over underlying PM banks, and (2) intra-log record persistence ordering. The imbalanced workload over banks leads to the high latency of log request persistence if a log request is served by the bank with a large number of pending memory requests. However, it is challenging to solve the bank workload imbalance. Reference locality tends to make requests cluster around a few banks, leading to workload imbalance over banks. Conventional log entry allocation schemes blindly allocate a physical address to a log request, further deteriorating this issue. To address this, we propose a load-aware log entry allocation (LALEA) scheme that allocates a log request to an address whose bank has the smallest amount of workload. The intra-log record persistence ordering requires a log record's metadata can be persisted only after all log entries of this record have been persisted to PM. The persisting serialization between a log record metadata and its entries aggravates transaction throughput. To remove this ordering constraint, we propose a new schedule, called buffer log metadata (BLOM), which stores metadata in a non-volatile ADR buffer until its log can be removed. BLOM not only eliminates the constraint of intra-record ordering but also avoids writing metadata to PM. Both LALEA and BLOM are generic and can be applied to many prior designs. Our evaluation shows that LALEA can achieve 54.04% higher transaction throughput on average, compared to the state-of-the-art design REDU. BLOM can achieve 17.16% throughput improvement over REDU on average. LALEA and BLOM together can boot throughput by 56.62%.

The main contributions of this paper are as follows:

- We identify two fundamental and common reasons that lead to the inefficiency of logging. Log requests are likely to be sent to banks with heavy workloads, causing high persistence latency. The intra-record ordering serializes the persistence of data entries and metadata, increasing transaction commit latency.
- To reduce the log request persistence time, we propose a load-aware log entry allocation (LALEA) scheme that allocates log request to the address whose bank has the lightest workload.
- To address the intra-record ordering issue, we propose to buffer log metadata (BLOM) in a non-volatile ADR buffer until its log can be removed.
- Both LALEA and BLOM are generic. We incorporated

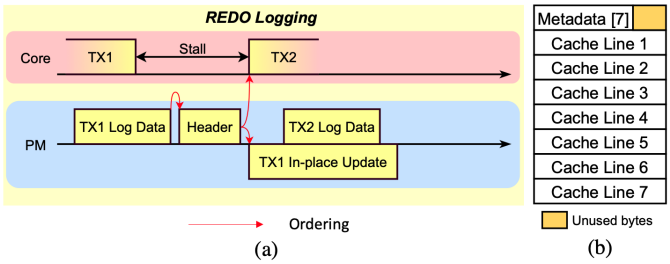

Fig. 1: (a) Ordering constraints of transaction execution in redo log. (b) A log record.

them into the state-of-art designs including REDU and LAD, and evaluated them using both micro-benchmark and real PM application workloads. Our experimental results show our designs significantly reduce log entry persistence latency and transaction commit latency, and hence boost transaction throughput considerably.

The rest of this paper is organized as follows. Section II discusses logging for crash consistency and ADR buffer. We present our design and evaluation in Section III and IV. Section V summarizes related work, and Section VI concludes this paper.

II. Background

A. Logging for Crash Consistency

Persistent memory systems need to guarantee crash consistency to be able to recover from system crash or power failure. Prior studies [1]–[3], [5], [9] have proposed various software-based logging schemes to achieve crash consistency. Based on log content, these logging schemes can be classified into two categories: *undo log* and *redo log*. For undo log, the target stale data to be updated are copied as log before new data are written in-place. After a crash, logs are used to undo all changes made to recover data back to a consistent state. For the redo log, new data are written to the log before in-place stale data are updated. During recovery, logs are used to redo all modifications. Both undo and redo log can degrade the overall performance significantly.

Ordering constraints are the root cause of the slow execution of transactions. Figure 1(a) shows the ordering in redo log. The execution of transaction $TX1$ is stalled when the CPU core meets $TxEnd$, which is $sfence$ in the X86 architecture. $TX1$ is stalled until all log requests of $TX1$ have been persisted to PM, marked as time instance $t2$. After $t2$, the CPU core proceeds to execute transaction $TX2$ as the memory controller performs in-place updates for $TX1$ in background. The $sfence$ puts part of log operations on the critical path, speeding up their execution can reduce CPU stall time and hence improve the system performance, which motivates the research presented in this paper.

B. Conventional Log Organization

A transaction log includes a 64-byte data content and a 8-byte home address for each write request occurred in this transaction. The address information in the log is referred to as log metadata. To reduce the log metadata write traffic, ATOM [1] proposes log entry collation (LEC) that co-locates seven log blocks' metadata in a 64-byte block referred to as a header. These seven log entries and the header constitute a log record, shown in Figure 1(b). A log record header can be persisted to PM only after its log data blocks are written to PM. This is the ordering constraint for persisting a log record, and it is referred to as the intra-record ordering. The intra-record ordering is on the critical execution path for both the redo and undo logging scheme. Figure 1(a) shows the ordering under redo logging.

However, LEC is sub-optimal in terms of write traffic reduction when the number of write requests in a transaction is smaller than seven. For example, if a transaction writes one cache line, its 64B log record header stores an 8B address, wasting PM write bandwidth. Whisper [10] shows most transactions have less than two write requests in the PM workloads. In addition, the log metadata traffic accounts for 12.5% of log traffic for transactions with a large number of write requests, degrading performance, and PM endurance.

C. ADR Buffer

Recently, Intel introduced *Asynchronous DRAM Refresh* (ADR) [11], which ensures that some pending write requests received by the memory controller will be persisted to NVM even upon the power failure, by utilizing backup power held in capacitors. With the support of ADR, the memory controller sends an acknowledgment immediately to the CPU for each $clwb$ instruction accepted by the ADR buffer. ADR allows the memory controller to become part of the persistence domain. The ADR buffer can accommodate 16 cache lines in the state-of-art Intel Optane [12]. ADR provides an exciting opportunity to design mechanisms with less log to achieve the atomicity of transaction updates. LAD [8] takes advantage of the non-volatility of the ADR buffer to eliminate log operations for a transaction if the write set of the transaction is smaller than ADR. Otherwise, LAD falls back to undo log to ensure crash consistency. However, concurrent transaction execution frequently makes LAD to incur ADR buffer overflow and suffer from log operations, leading to inefficient utilization of the ADR buffer.

III. Design

In this section, we present two optimizations to speed up logging process. The first one is the load-aware log entry allocation (*LALEA*), which accelerates the persistence of log requests. The second one is an efficient log metadata block buffering (*BLOM*) that further reduces the overhead caused by intra-record ordering.

A. Load-Aware Log Entry Allocation (LALEA)

The log operation execution of a transaction could be on the critical path. For example, redo logging can commit a pending transaction only after its all log requests are persisted. The slow execution of these log operations dictates the transaction commit latency. Log requests in a bank with a larger number of pending requests suffer from a higher persistence latency. Therefore, the slow log operation execution could be caused

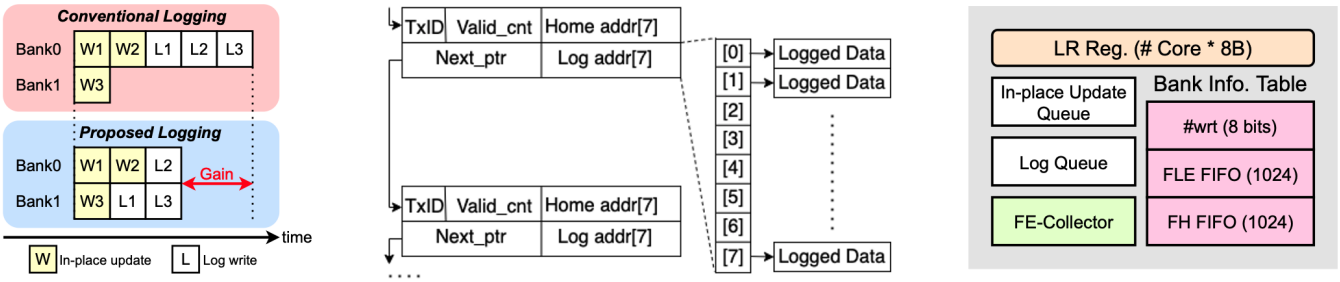

Fig. 2: (a) Compare LALEA with conventional logging. (b) LALEA log record organization. (c) LALEA controller.

by the imbalanced workloads over memory banks. The uneven workload distribution over banks could be caused by the inherent locality of workloads [13], [14], which makes in-place updates occur in one or a few banks. Furthermore, all prior designs prefer to allocate a well-known contiguous PM space for log entries, to facilitate log management and recovery. This workload-agnostic log allocation scheme deteriorates the issue of bank workload imbalance.

As discussed above, balanced workload among banks can lead to low latency for log requests. To balance banks workload, we propose a novel strategy that performs *load-aware log entry allocation* (LALEA) and effectively balances log write requests over PM banks. The main idea of LALEA is to allocate a log entry address according to the underlying banks' current pending operations. Specifically, LALEA allocates an incoming log write operation a free block in the log region whose bank has the smallest number of pending requests. Such adaptive log entry allocation can mitigate the workload imbalance of memory banks caused by transactions' in-place update operations. The balanced workload brings two performance benefits: (1) reduced log request persistence latency and (2) reduced memory latency for non-log requests. The evenly distributed bank traffic can decrease memory latency for both memory read requests and memory write requests issued in the transaction execution stage.

Fig. 2(a) compares our LALEA with the conventional log management by using a simple example. For simplicity, assume that the system has only two banks, $Bank0$ and $Bank1$, and they have two pending in-place update requests and one in-place update request, respectively. Conventional log management schemes sequentially allocate free memory blocks in $Bank0$ to log requests L1, L2, and L3, ignoring the current workload on these banks. This leads to a longer service time for the log request L3. However, our proposed method can allocate a log entry to the bank with the minimal number of pending operations, and these three log entries are allocated to two banks. LALEA can balance memory requests over banks and reduce the log requests completion time, especially for L3, decreasing the transaction commit latency.

The log organization scheme, such as LEC, requires that all log entries of a log record are sequentially stored in the log region so that they can be accessed without storing their addresses. However, the lack of the addresses of log entries makes it impossible for LELEA to access non-sequentially stored log entries of a log record. To address this issue, we enhance a log recorder header that includes an extra 64B metadata block to store addresses for seven log entries stored in different banks, as shown in Fig2(b). The field $next\_ptr$ in the second metadata block points to the log header of the next log record belonged to a transaction. The $next\_ptr$ is NULL if its log record is the last one in a transaction. The $valid\_cnt$ in the header indicates the count of valid log entries in its log record. The two-64B header is allocated to the same bank that has the minimal number of pending requests. Although LALEA writes extra log metadata to PM, the introduced performance overhead is shadowed by the performance improvement brought by LALEA, which is confirmed by our experimental results.

LALEA requires that each bank has a managed log region to store log entries. Two contiguous 64B data blocks of a LALEA header must be stored in the same bank. However, two contiguous free log entries cannot be guaranteed to be found in the free block list quickly. To solve this issue, we divide a bank's log region into the log entry region and the header region, and their allocation entries are 64B and 128B, respectively. The free log entry (FLE) FIFO and the free header (FH) FIFO indicate the free blocks to be allocated. When an entry is released, its address is appended to its related FLE FIFO. These two FIFOs only maintain addresses of free blocks. They are not required to be persisted in that they can be rebuilt during the crash recovery process.

Fig. 2(c) shows the LALEA controller. It contains a bank information table, log record registers (LR), a free entries collector (FE), a log queue, and an in-place update queue. Each entry of the bank information table includes a write operation counter, an FLE FIFO, and an FH FIFO. An 8-bit operation counter indicates the number of pending memory operations in the bank. Upon a log entry or a header allocation request, the controller first identifies the bank with the minimal operation counter and then grabs a free block address from the corresponding FIFO in the bank. An FLE FIFO and an FH FIFO contain 1024 and 256 free block addresses, respectively. A free block address only includes a row address and a partial column address. Assuming a row address and a column address have 16 and 10 bits respectively, a log entry block address and a header address have 20 bits and 19 bits. Therefore, an FLE FIFO and an FH FIFO require 2.5 KB and 2.375 KB,

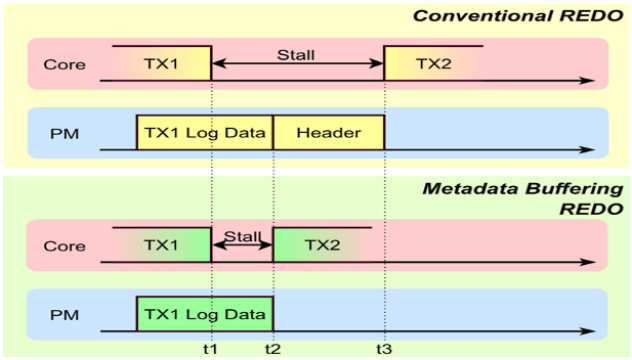

Fig. 3: Compare the redo logging and our proposed log metadata buffering.

respectively. A CPU core has an 8B LR register that points to the header of the first log record whose transaction's log records have been persisted to PM. The LR register is non-volatile in that persisted log entries of a CPU core are accessed through its LR register.

In case of a crash, the recovery module can revert the system to a consistent state, by accessing log entries persisted to PM. The non-volatile LR register maintains the address to the chain of headers for these persisted log data blocks. The $next\_ptr$ in a log record header links log records stored in PM, forming the chain. Following this chain, we can walk through these persisted log record headers and apply each logged data to the home region, to make the system state back to a consistent state.

B. Buffering Log Metadata in ADR

Conventional log organization is inefficient in terms of log writing throughput. The intra-record ordering constraints the degree of parallelism in log writes. Additionally, the low utilization of the recorder header wastes precious PM write bandwidth when the corresponding transaction has a few write requests, which is common in PM workloads [10].

To address these limitations of conventional log organization, we propose to buffer log metadata in ADR buffer, which is referred to as *BLOM*. The main idea is that log metadata blocks are buffered in the ADR buffer until they are not needed. Log metadata blocks can be discarded when all in-place updates of their corresponding transaction are completed. If there is no crash, the buffering log metadata blocks are not written to PM, improving the system performance and enhancing PM lifetime. Upon a crash, ADR persists the buffered log metadata blocks to PM. When the system reboots, the recovery module can recover the system to a consistent state, following the conventional recovering procedure. When the ADR buffer is used up, the memory controller writes the buffered metadata block of a selected transaction to PM.

Fig. 3 compares our proposed metadata buffering and the conventional redo logging. The conventional redo logging initiates the persisting of transaction $TX1$'s header at the time point $t2$ when the log entries are persisted. The header write operation is finished at time $t3$. The intra-recorder ordering requires the serialization of the persisting log data blocks and the header, and the stall of the CPU core execution until time $t3$. Our proposed log metadata buffering eliminates the ordering constraint by buffering the metadata block in the ADR buffer. Furthermore, this metadata buffering avoids writing the metadata block to PM. Therefore, the metadata buffering allows transaction $TX2$ to execute at time $t2$, reducing the CPU core stalling time.

Our proposed metadata buffering is inspired by LAD [8]. However, ours differs from LAD in the following aspects. First, while LAD buffers all update requests of a transaction, our design only buffers log metadata blocks. LAD degrades to the conventional logging scheme when the ADR buffer overflows. This is a common case when multiple transactions run concurrently and compete for the limited ADR buffer. ADR buffer overflow happens infrequently in our design as it only buffers metadata blocks. In addition, our design writes a smaller number of blocks than LAD upon an ADR buffer overflow. An overflow forces LAD to write log blocks for all update requests in a transaction, while our design only writes the minimal number of buffered metadata blocks.

IV. Evaluation

A. Experiment Setup

Cores	4 OoO core @2GHz,192 ROB entries, 48 STQ entries
TLB	L1: 6 sets, 4 ways; L2: 128 sets, 12 ways
L1 I/D Cache	private, 32KB, 2 cycles, 8 ways 8 log buffer entries
L2D Cache	private, 256KB, 8 cycles, 8 ways
LLC	8MB, 25 cycles, 16 ways
Memory Controller	1 channel, 1 rank, 8 banks, 8GB NVM 16 ADR Buffer entries [12], 32 Log Queue entries
NVM Access Latency	300(48) ns write(read) [15], [16]

TABLE I: System Parameters

The proposed design is implemented and evaluated by using ChampSim [17] with DRAMSim2 [18]. ChampSim is an Intel PIN [19] based simulator that models out-of-order micro-architecture at cycle level with detailed memory access behaviors, including TLB, LSQ memory dependence, and MSHR. To accurately model PM accesses, the cycle-level memory simulator DRAMSim2 is integrated with ChampSim. We enhance ChampSim to support *tx_begin*, and *tx_end*. The configurations of the processor and memory system used in our experiments are listed in Table I. The default memory address mapping is the page-level interleaving. We use both micro-benchmarks and real workloads in our experiments. The micro-benchmarks include B tree (B-Tree), chain queue (ChainQueue), hash table (HashTable), and red-block tree (RB-Tree), the real workloads include TPCC [20] and TATP [21]. Workloads are simulated under a 4-core processor and each core executes the same workload.

Our proposed LALEA and BLOM are generic logging optimization methods that can be applied to prior designs. Due to space limitation, this paper only demonstrates their capabilities by only applying them to two start-of-the-art designs: REDU [5] and LAD [8]. LALEA and BLOM represent that they are applied to REDU, while LALEA-LAD

denotes the LALEA optimization is applied to LAD. Since LALEA and BLOM are orthogonal, they can work together. LALEA+BLOM represents their combinations is applied to REDU.

B. Transaction Throughput

Figure 4 shows the transaction throughput improvement over REDU. LALEA, BLOM, and LALEA+BLOM improve the throughput of REDU by 54.04%, 17.16%, and 56.62% on average, respectively. LALEA balances the banks' workloads and reduces log operation latency, leading to throughput improvement. BLOM removes the intra-recording ordering constraints and reduces log traffics, thus also improving the throughput. LALEA is more effective than BLOM. This is because LALEA accelerates the writing of all log entries while BLOM only speeds up serving log record headers. As expected, LALEA+BLOM achieves an extra 2.58% throughput gain over LALEA.

Our evaluation shows that LAD is inferior to REDU in throughput. There are several reasons for LAD's performance degradation. First, concurrent transactions under multi-core CPU compete for the limited ADR buffer and the depletion of ADR buffer makes LAD fall back to log operations. Second, ADR buffer capacity is limited in the commodity CPUs which support PM. For example, the ADR buffer in the memory controller for Intel Optane memory only can buffer 16 cache lines [12]. Last but not the least, LAD issues log requests when LAD buffer overflows, while REDU can immediately initiate log requests as soon as they arrive. The delayed issue of log requests increases LAD transaction commit latency. In addition, REDU's logging scheme is more efficient than the logging scheme used by LAD. After being applied to LAD, LALEA can efficiently execute log requests and outperform LAD by 67.45% in terms of throughput.

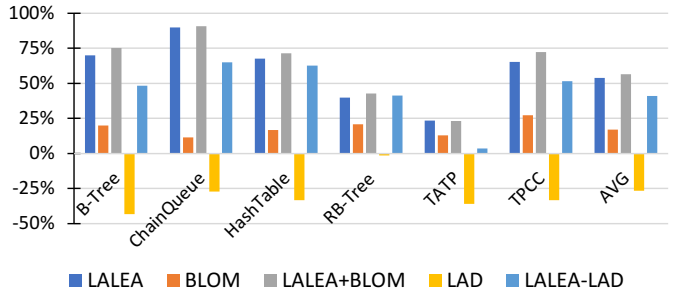

Fig. 4: Improving transaction throughput

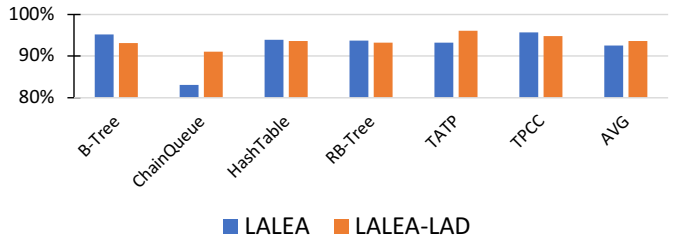

Fig. 5: Reducing log entry persistence latency

C. Log Entry Persistence Latency

Figure 5 shows incorporating LALEA into REDU and LAD can reduce the latency of log entry persistence by 92.51% and 93.66% on average over these two prior designs, respectively. LALEA effectively reduces log request queuing time by balancing PM banks workloads. The short queuing time can be translated into decreased persistence latency, leading to the higher transaction throughput.

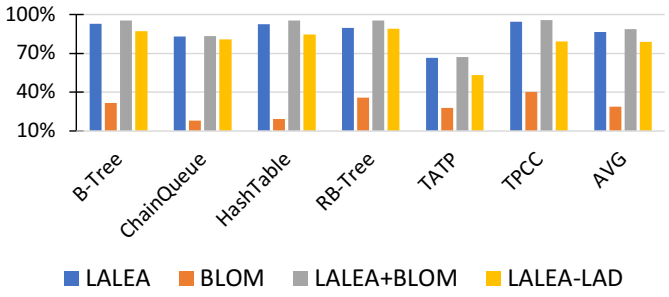

Fig. 6: Reducing transaction commit latency

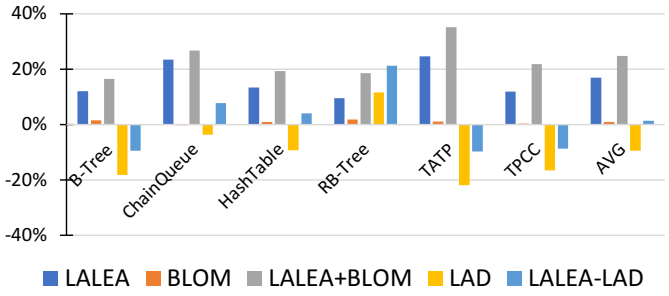

Fig. 7: Improving transaction throughput with cache line level interleaving

D. Transaction Commit Latency

Figure 6 shows transaction commit latency reduction for LALEA, BLOM, and LALEA+BLOM over REDU, and for LALEA-LAD over LAD. A transaction commit latency is defined as the period from the issue of its last instruction to the completion of its all log requests. On average, LALEA, BLOM, and LALEA+BLOM decrease the transaction commit latency by 86.55%, 28.74%, and 88.74%, respectively. In addition, LALEA reduces the commit latency by 79.02%, when it is applied to LAD. Since LALEA can reduce the log entry persistence latency, a transaction's log requests take a shorter time to complete, leading to a lower commit latency. BLOM avoids persisting log metadata block and thus also decreases the finish time of log requests. When these two optimizations work together, the commit latency is further reduced.

E. Throughput under the Alternative Address Mapping Scheme

We have demonstrated that LALEA and BLOM can improve the transaction execution performance, by decreasing the log entry persistence latency. These two optimizations work well when the imbalanced bank workload causes the longer log persistence latency. To demonstrate LALEA and BLOM capability,

we evaluate them under an aggressive address mapping scheme, cache line level interleaving, which distributes contiguous cache lines to different banks. Due to space limitations, we only present LALEA, BLOM, and LALEA+BLOM throughput improvement over REDU under this aggressive address mapping scheme. Figure 7 indicates that they improve the throughput by 17.01%, 0.99%, and 24.81% on average, respectively. In addition, LALEA improves LAD's throughput by 10.96%. As expected, the aggressive address mapping scheme can mitigate the imbalance of bank workload and achieve less performance gain for these optimizations than page-level address mapping. The more balanced bank workload makes BLOM achieve marginal performance improvement. However, LALEA still achieves significant performance gain.

V. Related Work

To efficiently reduce the overhead of logging, there are some research works on optimizing the logging operations. LAD [8] and VADR [22] were introduced to ensure the transaction atomicity and crash consistency, with logging-free support based on a non-volatile ADR buffer. In such design, all the cache lines in the ADR buffer can be flushed to NVM without logging once all updated cache lines of an in-flight transaction are collected inside the ADR buffer. Besides these in-place update approaches mentioned, the out-of-place update is also an emerging choice for crash consistency. Shadow Sub-page [23], [24] have been proposed to efficiently perform out-of-place updates based on hardware.

Memory scheduling algorithms can improve memory system performance. Recently, PPO [25] schedules persistent requests to improve the parallelism of memory and network, to achieve BLP-aware global barrier epoch management under the buffered epoch persistence model. Our LALEA differs from the prior bank-level-parallel (BLP) aware memory scheduling algorithm in the following aspects. First, LALEA is more aggressive than PPO in that LALEA reshapes the log requests distribution over banks to more effectively attack the issue of bank workload imbalance. In contrast, PPO only passively schedules memory requests. Second, LALEA works for more situations than PPO. For example, PPO does not work efficiently for either a non-buffered persistency model or the workload without inter-thread dependencies. Lastly, LALEA is orthogonal to PPO and can be integrated into PPO to improve performance further.

VI. Conclusions

Logging schemes are widely used to ensure crash consistency for persistent memory (PM). The ordering constraints between log operations and a transaction execution places some log operations on the I/O critical path, thus introducing severe performance overhead. We have identified two fundamental and common reasons for the inefficiency of log operations. First, the over-loaded PM banks increase the queuing time for log requests served by these banks, leading to a high persistence latency for log requests. Second, the intra-record ordering serializes the persisting of log entries and the header, increasing transaction commit latency. To address the first issue, we propose LALEA that balances banks' workload through the novel log entries allocation method. To address the second issue, we propose BLOM that removes intra-record ordering constraints by buffering headers in the ADR buffer. Our evaluation shows that LALEA can achieve 54.04% and 17.16% higher transaction throughput on average compared to prior designs, respectively.

References

[1] 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.

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

[3] 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.

[4] S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," in *ISCA*, 2017, pp. 175–186.

[5] 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.

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

[7] X. Wei, D. Feng, W. Tong, J. Liu, and L. Ye, "Morlog: Morphable hardware logging for atomic persistence in non-volatile main memory," in *ISCA*. IEEE, 2020, pp. 610–623.

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

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

[10] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper."

[11] D. Mulnixl. Intel Xeon processor D product family technical overview. https://software.intel.com/en-us/articles/intel-xeon-processor-dproduct-family-technical-overview/.

[12] "Intel Optane DC persistent memory sampling today revenue delivery 2018," 2018. [Online]. Available: https://www.servethehome.com/intel-optane-dc-persistent-memory-sampling-today-revenue-delivery-2018

[13] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ISCA*. ACM, 2009, pp. 14–23.

[14] P. Zuo, Y. Hua, and Y. Xie, "Supermem: Enabling application-transparent secure persistent memory with low overheads," in *MICRO*. ACM, 2019, pp. 479–492.

[15] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *HPCA*, 2018, pp. 310–323.

[16] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing memory and storage support for non-volatile memory systems," in *ISCA*, 2019.

[17] Champsim. https://github.com/ChampSim/.

[18] Dramsim. https://github.com/umd-memsys/DRAMSim2.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[20] Transaction processing performance council (TPC), TPC-C. http://www.tpc.org/tpcc/default.asp.

[21] A. W. Markku manner Vilho Raatikka Simo Neuvonen. TATP telecommunication application transaction processing (benchmark description). http://tatpbenchmark.sourceforge.net/TATP-Description.pdf.

[22] Z. Lu, J. Yue, Y. Deng, and Y. Zhu, "Improving the performance of nvm crash consistency under multicore," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020, pp. 561–564.

[23] Y. Ni, J. Zhao, H. Litz, D. Bittman, and E. L. Miller, "SSP: Eliminating redundant writes in failure-atomic NVRAMs via shadow sub-paging," in *MICRO*, 2019.

[24] Y. Deng, J. Yue, Z. Lu, and Y. Zhu, "Efficient hardware-assisted out-place update for persistent memory," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 507–512.

[25] X. Hu, M. Ogleari, J. Zhao, S. Li, A. Basak, and Y. Xie, "Persistence parallelism optimization: A holistic approach from memory bus to RDMA network," in *MICRO*. IEEE Computer Society, 2018, pp. 494–506.