Empirical Study of Redo and Undo Logging in Persistent Memory

Hu Wan[†], Youyou Lu[‡], Yuanchao Xu[†*], and Jiwu Shu[‡]
[†]College of Information Engineering, Capital Normal University, Beijing, China
[‡]Department of Computer Science and Technology, Tsinghua University, Beijing, China
[*]State Key Laboratory of Computer Architecture, ICT, CAS, Beijing, China
[†]{wanhu, xuyuanchao}@cnu.edu.cn, [‡]{luyouyou, shujw}@tsinghua.edu.cn

***Abstract*—Atomic and durable transactions are widely used to ensure the crash consistency in *persistent memory* (PM). However, whether to use redo or undo logging is still a hotly debated topic in persistent memory systems. In this paper, we empirically study the performance of both redo and undo logging using NVML, a persistent memory transactional object store framework. Our results on an NVDIMM server show that redo logging significantly outperforms undo logging for workloads in which a transaction updates large number of different objects, while it underperforms undo logging for workloads with intensive read operations. Furthermore, undo logging is more sensitive to the read-to-write ratios, compared to redo logging. Finally, our experiments also demonstrate that asynchronous log truncation is much helpful in redo logging for log-heavy transactions.**

I. Introduction

The up-and-coming *non-volatile memory* (NVM) technologies, such as phase change memory, memristors, and recently announced 3D XPoint technology [1], offer an intriguing blend of storage and memory. This has implications on system software (e.g., file system), libraries (e.g., persistent object store), and applications (e.g., database). For instance, state-of-the-art persistent memory file systems exploit byte-addressability of persistent memory to avoid overheads of block-oriented storage [2], [3]. Persistent memory also enables aggressive programming models that directly leverage persistence of the memory, since persistent memory allows applications to access the data structures through a fast load/store interface, without first performing block I/O and then transferring data formats into memory based structures [4], [5].

However, ensuring that application data can be correctly recovered in the event of sudden power failure or system crash is non-trivial in NVM-based persistent memory systems. To achieve crash consistency [6], it is required to serialize multiple write operations when manipulating complex data structures with fine-grained accesses. But most modern processors reorder writes to memory at cache hierarchy and even memory controller level to optimize performance [7], [8]. Because of this reordering, if a failure occurs between two reordered writes, the data structures in persistent memory could be in an inconsistent state.

Most prior persistent memory designs provide atomic and durable transactions to move the data from a consistent state to another one [7], [9]–[13]. A transaction is defined as a series of operations on persistent memory that either all occur, or nothing occurs with respect to failures. If the execution of a transaction is interrupted, it is guaranteed, after system restart, to restore the consistent state from the moment when the transaction was started. To this end, it is required to write data to a log area before updating data in-place.

Transaction log is an essential part of any transaction, and it performs several important functions that are vital for maintaining consistency. Transaction log comes in two flavors: the data before the modification (undo logging [2], [10], [11]), and the data after the modification (redo logging [9]). With redo logging, transaction appends all data updates to log area and makes them durable before writing the data to persistent memory. When the transaction commits successfully or when the log buffer runs out, the log contents are checkpointed to the original persistent locations. With undo logging, transaction works by first copying the old data to the log area and making them durable, then committing the updates in-place in the original data location. In the event the transaction fails, any modifications to persistent memory are rolled back using the old data in the undo log area. Given these two logging methods, it is essential to understand their relative performance in persistent memory.

However, whether to use redo or undo logging in persistent memory is still in hot debate. Existing research works have not yet comprehensively elaborated the performance gap between undo logging and redo logging with practical evaluations. We are hence motivated to investigate this issue.

To evaluate the two logging methods, we use a lightweight persistent memory transaction object store, which is based on the open-source *NVM library* (NVML) [14]. It saves the current value of given persistent memory range in the undo log. We extend it to support redo logging. To gain insights into application performance with undo logging and redo logging, we conduct experiments with two workloads and a typical transactional KV-store application. We make several specific observations.

Observation 1. Redo logging significantly outperforms undo logging for workloads in which a transaction updates large number of different objects, while it underperforms undo logging for workloads with intensive read operations.

Observation 2. Undo logging is more sensitive to the read-

Corresponding authors: Yuanchao Xu and Jiwu Shu.

to-write ratios, while redo logging is less sensitive.

Observation 3. Enabling asynchronous log truncation significantly boosts transaction performance, especially for log-heavy transactions.

The remainder of this paper is organized as follows. We begin in Section II with a discussion of redo logging and undo logging and why it necessitates a re-evaluation. Next, in Section III we describe experimental methodology. We then present in Section IV the evaluations and analysis. Finally, we conclude in Section V.

II. Background and Motivation

Persistent memory transactions are widely used in file systems and databases to make atomic, consistent, and durable modifications to the storage system. In such a system, constraining the order that writes persist (referred to as *persist ordering constraints* [7]) is essential to ensure consistent recovery. However, persist ordering constraints should be enforced either by using a write-through cache [15] or by explicitly flushing individual cache lines (e.g., using the clflush instruction on X86). Moreover, these flush operations should be carefully annotated with fences to prevent hardware and compiler from reordering. These mechanisms are quite slow, and thus minimizing these constraints is a key challenge to achieve high performance [7], [16]–[19].

As shown in Fig. 1, we illustrate the timeline for sequence of updating three data blocks in a transaction with undo logging and redo logging separately. We use block A_0 to represent the data block of an old valid version and use A'_0 to represent its copy stored in the log area. Accordingly, we use block A_1 to represent the new version being updated and use A'_1 to represent its copy stored in the log area. Fig. 1(a) depicts the sequence of an undo logging transaction. An in-flight persistent memory transaction allocates log space and copies the prior state of all data that will be modified (A_0, B_0, and C_0) to the log (A'_0, B'_0, and C'_0). Each log write is followed by flush and memory fence operations. After that, the application is able to modify the data structure in place (A_1, B_1, and C_1) immediately. Cache flush and memory fence operations ensure that the log data reach the persistent memory immediately after they are issued. Otherwise, a failure may cause the system to a state in which some of the original states are lost. Finally, by marking the undo log entries invalid, the transaction becomes committed. Fig. 1(b) shows the sequence of a redo logging transaction. An in-flight persistent memory transaction keeps adding new data and their addresses to the log (A'_1, B'_1, and C'_1). After all log records for that transaction have been written, transaction sets the commit record followed by cache flush and memory fence operations. Then, the system can overwrite the real data structures in the persistent memory (A_1, B_1, and C_1).

Redo logging makes it possible to reduce the ordering constraints on writes to persistent memory. The only requirement is that the log is written completely before any data within this transaction are updated. In contrast, undo logging requires ordering a log write before each memory update. On the other hand, redo logging incurs an additional overhead for all the read operations in a transaction, since every load/store to same data must be redirected to the log area instead of its original location. Undo logging, by contrast, always reads the most recent data directly from the in-place data structure so as to leverage the byte-addressability of persistent memory.

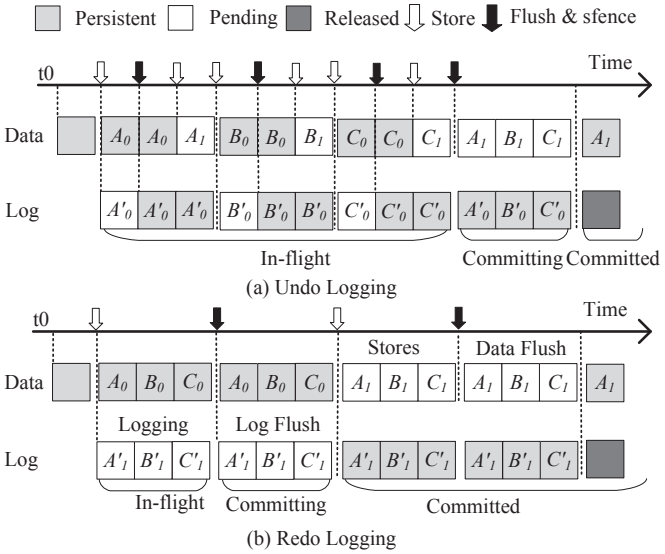

Fig. 1. Comparison of the timeline of undo logging and redo logging.

Table I summarizes the high-level pros and cons of redo logging and undo logging. We found that some of the previous works even draw absolutely opposite conclusions based on respective experiments. For instance, Mnemosyne [9] and NV-Heaps [10], two state-of-the-art persistent object store systems, are divided. The former prefers to use redo logging, and the latter opts to use undo logging.

Therefore, we are motivated to conduct empirical performance study and revisit the performance of redo and undo logging methods in the context of persistent memory. By doing so, we intend to better weigh the costs and benefits in designing the persistent memory transactions.

III. Experimental Methodology

A. Experimental Setup

All of our experiments are conducted on a Linux server (kernel version 4.3.5-300) with two Intel Xeon(R) E5-2637 v3 processors and 32 GB DRAM. Each processor has four physical cores clocked at 3.5 GHz. This server is also equipped with 32 GB SMART^{TM}DDR4 NVDIMM, which has practically the same access latency as DRAM [1].

Most file systems, databases, and persistent object store systems rely on strong consistency guarantees provided by some form of transaction for persistent data structures. We focus our analysis on transactional object store in persistent memory, such as Mnemosyne [9], NV-Heaps [10] and NVML [14]. These libraries build atop memory-mapped files to provide fine-grained persistent memory management and transaction support. To this end, we use a persistent memory

[1]Due to the unavailability of emerging NVMs (e.g., PCM, STT-RAM), we leave the evaluation on them to our future work.

TABLE I
COMPARISON OF REDO LOGGING AND UNDO LOGGING IN PERSISTENT MEMORY.

Logging Method(s)	Redo Logging	Undo Logging
Ordering constraints on writes	• requires only two sync barriers per transaction, irrespective of the number of log entries [2], [9], [11], [20]	• requires a sync barrier for every log entry within a transaction [2], [9], [11]
Overhead for read operations in a transaction	• reads of uncommitted data must be intercepted and redirected to the redo log [2], [11], [19], [21], [22] • overhead of searching log is inversely proportional to logging granularity [2]	• allows to read the most recent data directly from the in-place data structure [19], [21], [22] • allows fine-grained logging [2]
Commit/abort cost	• transaction commits are slower because the new values have to be written out to memory [9]	• transaction aborts are slower since values that will be restored to memory when transaction is rolled back

aware file system to expose *direct access* (DAX) feature. More specifically, we use DAX-enabled EXT4 file system to allow applications to perform direct access to persistent memory with `mmap()`. The architecture of the testbed running on the NVDIMM-based persistent memory is depicted in Fig. 2.

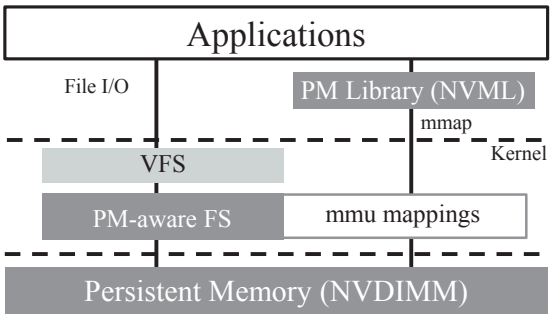

Fig. 2. Architecture of persistent memory system.

B. Redo and Undo Logging Implementation

The transactional object store system that we use in our study is based on the open-source libpmem and libpmeobj library in NVML [14]. Libpmem provides the low-level persistent memory support for libpmemobj. Libpmemobj library handles the transactional updates, flushes changes to persistent memory, and recovers data from the log. We extend this library to safely update memory block with both redo logging and undo logging.

Fig. 3 shows the differences between undo logging and redo logging in a persistent memory transaction from a programmer's perspective. A single transaction goes through a series of stages. At first, `TX_BEGIN` starts a new transaction in the current thread. Then in undo logging transaction, the existing content of the memory range is copied into log as shown in Line 2 and Line 4 in Fig. 3(a). After that, we can freely modify the key and value fields of map object. In contrast, in a redo logging transaction, the programmer needs to explicitly add the tentative values to the log as shown in Line 2 and Line 3 in Fig. 3(b), and the updates to their original locations will be delayed until the transaction commits. In case of reads of uncommitted data, they will be intercepted and redirected to the redo log area. Finally, `TX_END` ends the transaction.

As discussed in Section II, in undo logging transaction, we ensure the log data reach the persistent memory immediately by issuing flush and memory fence operations after each log

```
1: TX_BEGIN(pop) {
2:   ADD_UNDO_LOGENTRY(map, key);
3:   map->key = new_key;
4:   ADD_UNDO_LOGENTRY(map, value);
5:   map->value = new_value;
6: } TX_END
```

(a) Undo Logging

```
1: TX_BEGIN(pop) {
2:   ADD_REDO_LOGENTRY(map, key, new_key);
3:   ADD_REDO_LOGENTRY(map, value, new_value);
4:   map->key = new_key;
5:   map->value = new_value;
6: } TX_END
```

(b) Redo Logging

Fig. 3. An example of undo logging and redo logging persistent memory transaction.

write. In redo logging transaction, cache flush and memory fence operations are performed until transaction commits. Undo log is a list of data structures that contain the offset of the memory range, its size and the original data. Accordingly, redo log list entries record the offset, size and update data. In order to avoid wasting a lot of memory and time on allocating for tiny log entries, we adopt the optimization proposed in NVML [14] to create an undo/redo log cache which will be preallocated for each transaction and will store log entries whose size does not exceed the threshold (e.g., 32 B in our case). The cache is re-used across transactions. Moreover, in order to make transaction log to never contain duplicated memory ranges, we search for all overlapping ranges and add only the memory range that is not overlapping.

We also implement both synchronous and asynchronous log truncation for redo logging transaction similar to Mnemosyne [9]. Synchronous truncation forces new values written to memory during transaction commits, then truncates the entire log. Asynchronous truncation retains the log after transaction commits and truncates the log offline, which moves the latency of truncating off the critical path.

C. Workloads

Based on the benchmark framework provided in NVML [14], we evaluate the undo logging and redo logging performance with various log entry sizes under single object (referred to as single-obj) and multiple object (referred to as multi-obj) workloads.

- **single-obj**: Each transaction allocates one persistent object, and adds the same object to undo/redo log many times.
- **multi-obj**: Each transaction allocates many persistent objects, and adds all the objects to undo/redo log.

The two configurations are shown in Table II, where i::j::k denotes a set of values from i to k by a stepping factor of j.

TABLE II
CONFIGURATIONS OF WORKLOADS.

Workload	Log entry size (byte)	Number of operations
single-obj	128::2::16384	1000
multi-obj	128::2::16384	1000

We then modify the benchmark framework to vary the percentage of the read operations in each transaction. We set different ratios of read and write operations in a transaction, from 10 % reads to 30% and 50% reads, respectively.

We also evaluate the overall performance with hashmap_tx, a real-world transactional KV-store application provided in NVML [14], which uses hashmap as its backend. This workload provides transactional object store. The default map entry size in hashmap_tx is 24 B (key and value sizes are 8 B and 16 B, respectively). We extend hashmap_tx and vary the size of map entry from 128 B to 8192 B. We perform 1000 insert operations each time.

IV. EXPERIMENTAL RESULTS AND ANALYSIS

Our findings are based on the experiments across workloads, log entry sizes and read-to-write ratios. Figures 4-6 depict the throughput of undo logging, redo logging with synchronous log truncation (denoted as redo-sync) and redo logging with asynchronous log truncation (denoted as redo-async) under the single-obj and multi-obj workloads. In addition, we use the term redo logging to refer to both redo-sync logging and redo-async logging. In each plot, the X-axis denotes the size of log entry (byte) and the Y-axis denotes the throughput (operations per second).

A. Observations

By comparing redo and undo logging using different configurations, we make the following three observations:

Observation 1. *Redo logging significantly outperforms undo logging for workloads in which a transaction updates large number of different objects, while it underperforms undo logging for workloads with intensive read operations.*

Fig. 4 presents the throughput performance for the single-obj and multi-obj workloads for transactions with 100% writes. From Fig. 4(a), we observe that redo logging and undo logging show a great difference in throughput as the log entry size decreases. Redo-sync logging and redo-async logging show 17% and 20% performance improvements on average over undo logging. In particular, redo-sync logging and redo-async logging outperform undo logging by 24% and 37% when the log entry size is 128 B, respectively. This is because both redo-sync logging and redo-async logging hide the long persist latency by deferring the heavy-weight cache flush and memory fence operations until transaction commits. In contrast, undo logging issues flush and memory fence operations immediately after each log write. For multi-obj workload (Fig. 4(b)), in which a transaction writes multiple objects, redo-async logging gains performance improvement by up to 63% when the log entry size is 256 B, and 32% on average when the log entry size is less than 2048 B than undo logging. The reason is that redo logging significantly reduces the persist ordering constraints. More concretely, redo logging requires only two sync barriers per transaction, irrespective of the number of log entries, while undo logging requires a sync barrier for each log entry within a transaction. We also see that redo-sync logging performs at par with and even worse than undo logging as the log entry size increases. This is mainly because the log truncation overheads gradually become relatively more significant that other parts as the transaction log size (i.e., the number of log entries×each log entry size) increases.

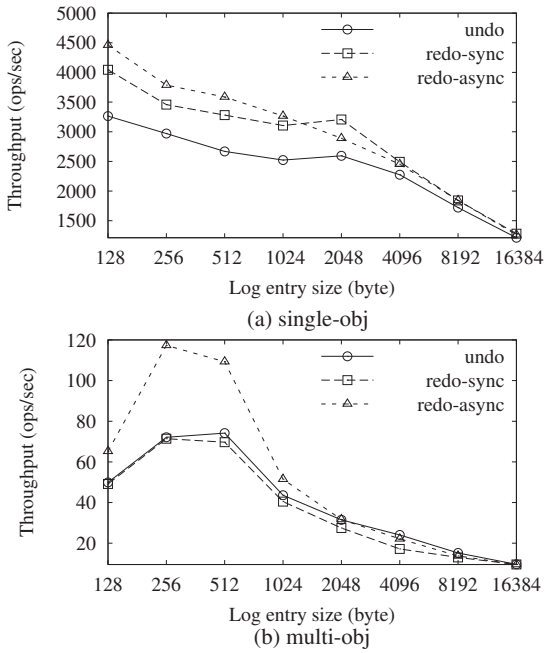

Fig. 4. Throughput of single-obj and multi-obj workloads with varied log entry sizes.

However, when taking read operations into account, as shown in Fig. 5 and Fig. 6, redo logging considerably underperform undo logging, especially for read-intensive workloads. We observe that, when the percentage of read operations is 50% for single-obj workload (Fig. 5(c)), undo logging has better performance than both redo-sync logging and redo-async logging, and it outperforms redo-sync logging and redo-async logging by 40-252% and 20-236%, respectively. The source of this enhancement mainly comes from three aspects: (1) Undo logging transaction can always read the most recent data directly from the in-place data structure, while redo logging transaction incurs substantial overheads due to the redirection of reads. (2) Undo logging transaction benefits more from the strong access locality exhibited in this workload. (3) The transaction log size is small due to the optimization for duplicate log entries discussed in Section III-C; therefore persist overhead for undo logging is less significant than software overhead for redo logging (e.g., searching log entries for uncommitted reads). Similar observation can also be found from multi-obj workload (Fig. 6(c)). Although larger transaction log size and stricter ordering constraint on writes

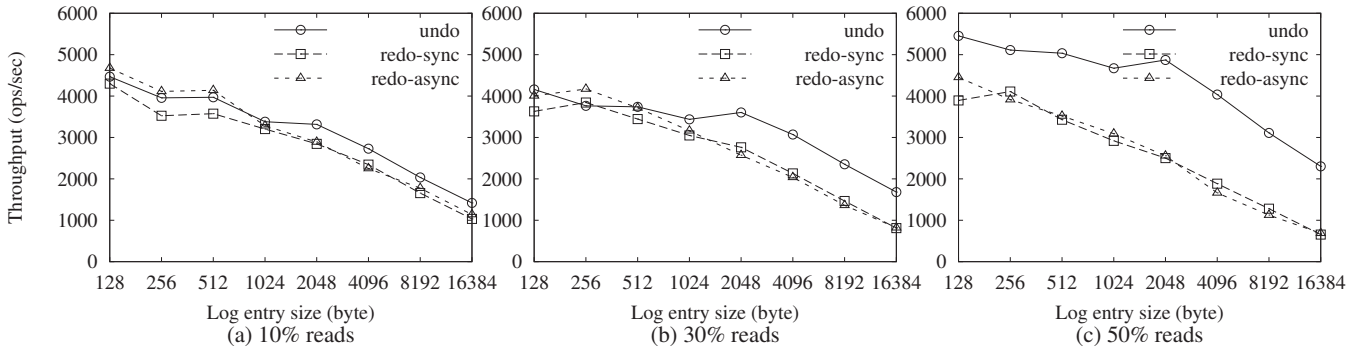

Fig. 5. Throughput of single-obj workload with varied log entry sizes and read-to-write ratios.

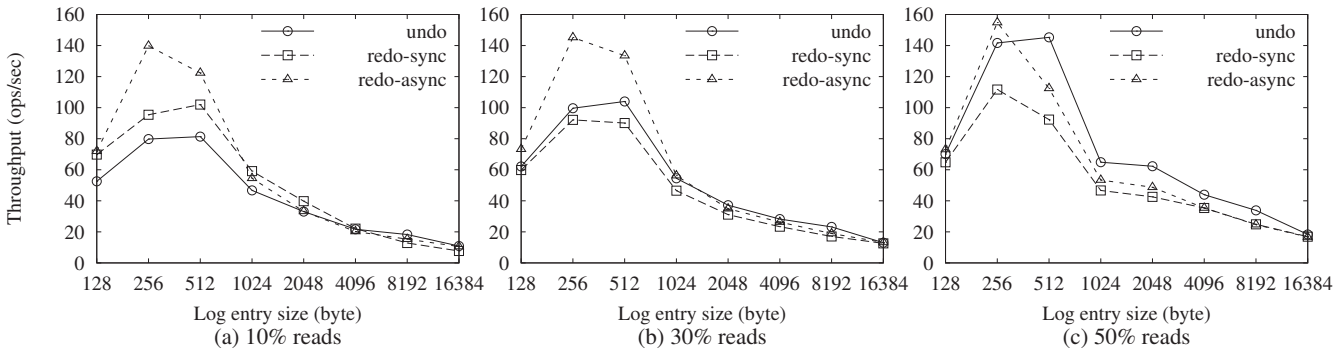

Fig. 6. Throughput of multi-obj workload with varied log entry sizes and read-to-write ratios.

penalize undo logging performance, it still yields better performance than redo logging except that the log entry size is less than 256 B. The striking advantage of undo logging is absolutely opposite to what we have obtained for workloads without read operations in transactions.

We conclude that redo logging and undo logging have completely different behaviors. Redo logging significantly outperforms undo logging for workloads in which a transaction updates large number of different objects, while it underperforms undo logging for workloads with intensive read operations.

Observation 2. *Undo logging is more sensitive to the read-to-write ratios, while redo logging is less sensitive.*

As shown in Fig. 5 and Fig. 6, we observe that read-to-write ratio have large effects on undo logging performance. As the read-to-write ratio increases, undo logging yields significant performance gains.

As shown in Fig. 5, in single-obj evaluation, undo logging has larger performance improvement when the read-to-write ratio is higher. For instance, by comparing figures 5(a), 5(b), and 5(c), we can see that the average performance of undo logging is improved by 40% when the read-to-write ratio is increased from 10% to 50%. Conversely, the overall performance of redo-sync logging and redo-async logging are decreased when the read-to-write ratio is increased from 10% to 50%. This is mainly because that redo logging suffers from redirecting reads to redo log. It is worth noting that the amplitudes of performance decrease are 26% and 17% for redo-sync logging and redo-async logging, respectively, which are lower than that of redo logging. Similar results are seen in Fig. 6. Compared to the 10% reads case, undo logging performance with 50% reads is improved by 72%. By comparison, redo-sync logging and redo-async logging performance are decreased by 33% and 30%, respectively.

This set of experiments demonstrate the notable benefits of avoiding redirecting reads of uncommitted data to the log, especially for workloads with intensive read operations.

Observation 3. *Enabling asynchronous log truncation significantly boosts transaction performance, especially for log-heavy transactions.*

As shown in Fig. 4, Fig. 5, and Fig. 6, log truncation methods have a strong impact on transaction performance. More specifically, log-heavy workloads gain more benefits from asynchronous log truncation.

For multi-obj workload with redo-sync logging and redo-async logging (as shown in Fig. 4(a), Fig. 6), both performance increase from 128 B to 256 B, followed by a curve where throughput rapidly falls off. In the increasing region, where log entry size is small, the overall throughput is improved greatly. This is attributed to the fact that persistent memory allocator used in our study can get more performance benefits as the allocation size increases. When the log entry size is larger than 256 B, transaction performance shows a remarkable drop as the log entry size increases. The reason is that larger allocations for new log entries make the persist overheads become relatively higher than the benefits it gains.

However, redo-sync logging and redo-async logging show a vast difference in throughput when the log entry size is less than some certain thresholds (e.g., 8192 B in most cases). Redo-async logging shows up to nearly 38%, 25%, 24%,

and 20% performance improvements over redo-sync logging when varying the read-to-write ratio. This is mainly because asynchronous log truncation reduces the latency of transaction commits by retaining the log after transaction commits and truncating them offline. Instead, synchronous log truncation suffers from parsing the log records and forcing new values to persistent memory during transaction commits.

B. Real-world Application Analysis

To further investigate the performance of undo logging and redo logging for a real-world application, we evaluate the hashmap_tx, a hashmap-based transactional KV-store. Fig. 7 shows the experimental results obtained when varying the map entry size from 128 B to 8192 B along with different logging methods.

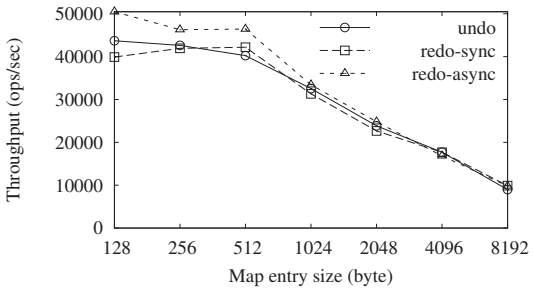

Fig. 7. Throughput of a transactional KV-store with varied map entry sizes.

Redo-async logging is the best in performance, as shown in Fig. 7, for almost all map entry sizes. Redo-async logging performs 8%∼15% better on average compared to undo logging. The dominating operations of inserting entries into KV-store are write operations; therefore the overhead of searching log is negligible. In contrast, redo-async logging benefits from relaxing persist ordering constraints as well as reducing the overhead of synchronous log truncation. The performance gap between redo-sync logging and redo-async logging also demonstrates the benefits of moving the latency of truncating off the critical path.

V. Conclusion

Efficient design of persistent memory transaction is still an open problem today. Redo or undo logging are widely used mechanisms in persistent memory transaction. However, whether to use redo or undo logging in persistent memory is still in hot debate. In this paper, we empirically evaluate redo and undo in the open source NVML library. Our study suggest "one does not fit all". Redo logging significantly outperforms undo logging for workloads in which a transaction updates large number of different objects, while it underperforms undo logging for workloads with intensive read operations. Also, undo logging is more sensitive to the read-to-write ratios, while redo logging is less sensitive. In addition, asynchronous log truncation is much helpful in redo logging for log-heavy transactions. These observations do help in choosing proper logging method for future consistency designs of persistent memory systems, which can behave various access patterns.

VI. Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback to this paper. This work is supported by the National Natural Science Foundation of China (Grant No. 61502266), the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), the China Postdoctoral Science Foundation, and the State Key Laboratory of Computer Architecture (Grant No. CARCH201503).

References

[1] Intel and Micron, "Intel and micron produce breakthrough memory technology," https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/, July 2015.

[2] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *EuroSys*. ACM, Apr. 2014, pp. 1–15.

[3] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *EuroSys*. ACM, 2016, pp. 1–16.

[4] R. Andy, "Programming models for emerging non-volatile memory technologies," *;login:*, vol. 38, no. 3, pp. 40–45, June 2013.

[5] H.-J. Boehm and D. R. Chakrabarti, "Persistence programming models for non-volatile memory," HP Technical Report HPL-2015-59, Tech. Rep., 2015.

[6] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Crash consistency," *Commun. ACM*, vol. 58, no. 10, pp. 46–51, 2015.

[7] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *ICCD*. IEEE, Oct. 2014, pp. 216–223.

[8] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *MICRO*. ACM, Dec. 2015, pp. 660–671.

[9] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ASPLOS*. ACM, Mar. 2011, pp. 91–104.

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *ASPLOS*. ACM, Mar. 2011, pp. 105–118.

[11] D. Narayanan and O. Hodson, "Whole-system persistence," in *ASPLOS*. ACM, Mar. 2012, pp. 401–410.

[12] 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 *MICRO*. ACM, Dec. 2013, pp. 421–432.

[13] J. Ou and J. Shu, "Fast and failure-consistent updates of application data in non-volatile main memory file system," in *MSST*. IEEE, May 2016, pp. 1–16.

[14] C. Krzysztof and R. Andy, "nvml: Linux nvm library," https://github.com/pmem/nvml, Aug. 2014.

[15] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Implications of cpu caching on byte-addressable non-volatile memory programming," HP Technical Report HPL-2012-236, Tech. Rep., 2012.

[16] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *SOSP*. ACM, Oct. 2009, pp. 133–146.

[17] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ISCA*. IEEE, June 2014, pp. 265–276.

[18] Y. Lu, J. Shu, and L. Sun, "Blurred persistence in transactional persistent memory," in *MSST*. IEEE, May 2015, pp. 1–13.

[19] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *ASPLOS*. ACM, Apr. 2016, pp. 399–411.

[20] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "NVWAL: Exploiting NVRAM in write-ahead logging," in *ASPLOS*. ACM, 2016, pp. 385–398.

[21] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen, "Persistent transactional memory," *Computer Architecture Letters*, vol. 14, no. 1, pp. 58–61, 2015.

[22] S. Kannan, M. Qureshi, A. Gavriloska, and K. Schwan, "Energy aware persistence: Reducing the energy overheads of persistent memory," *Computer Architecture Letters*, vol. PP, no. 99, pp. 1–4, 2015.