Quantifying the Performance Overheads of PMDK

William Wang, Stephan Diestelhorst
Arm Research
{william.wang,stephan.diestelhorst}@arm.com

1 INTRODUCTION TO PERSISTENT MEMORY PROGRAMMING

For systems with non-volatile main memories, *i.e.*, *NVDIMMs*, *failure atomicity* is required to guarantee that systems can always recover to consistent states following power or system failures. Such failure atomicity can be achieved with logging and flushing as with filesystems. Similarly, with non-volatile main memories, failure atomicity can be achieved with user space applications using write logging, cacheline flushing, and barriers that order such operations. *Write logging*, either undo or redo logging, ensures atomicity when a failure interrupts the last atomic operation from completion. Undo logging helps systems recover to the last consistent state immediately before the failed atomic operation, and redo logging helps systems restore to the consistent state right after the failed atomic operation. *Cacheline flushing* ensures volatile caches do not hold persistent data from reaching the point of persistence, so persistent data won't be lost when a sudden power or system failure occurs. *Barriers* help prevent potential reordering in the memory hierarchy, as caches and memory controllers may reorder memory operations. For example, a barrier ensures the undo log copy of the data gets persisted onto the persistent memory before the data is mutated in-place, so it's guaranteed that the last atomic operation can be rewound should a failure happens. However, it's non-trivial to add such failure atomicity in user applications with low-level operations such as write logging, cacheline flushing, and barriers [5].

PMDK [4] is a user space library that abstracts such low-level operations away from application developers and wraps such operations into transactional APIs in *libpmemobj* that user space applications can call for ensuring failure atomicity. *libpmemobj* transactional APIs do not guarantee multithread atomicity though [2], developers need to take care of multithread atomicity as with traditional volatile memories. **ATLAS** [3] adds support of failure atomicity with traditional lock-based concurrent applications.

2 EVALUATION

Given PMDK is supported on Linux, Windows and FreeBSD platforms, it has been widely adopted to port legacy applications, such as in-memory databases, to persistent memory [6]. The convenience of calling into transactional APIs in PMDK doesn't come for free though. In this work, we quantify the performance overheads of PMDK transactions, to identify potential areas for hardware acceleration.

2.1 Workloads

We use the small kernel workloads implemented with PMDK *libpmemobj* transactional APIs that come with PMDK, as well as a more realistic workload, the PMDK-enabled Redis-server, driven by TPCC payload as well as Redis-benchmark.

For small kernel workloads, we have trees and hashmaps implemented with *libpmemobj* APIs, including *ctree, btree, rtree, rbtree* and *hashmap*. The throughputs of *insert, delete* and *lookup* operations are measured.

In addition, we have PMDK-enabled Redis-server driven with TPCC payload and Redis-benchmark. The TPCC transactional operations [1] consist of 1) new order 2) payment 3) order status 4) stock level and 5) delivery, in which 3) and 4) are read-only operations, and performance is measured by throughputs of TPCC transactions. Redis-benchmark, a command line tool in Ubuntu distributions, is a synthetic payload generator for evaluating the Redis-server under different load conditions. We use Redis-benchmark to generate SET requests as payload to the Redis-server.

2.2 Methodology

The small kernel workloads are run with logging and flushing functions turned on and off. For PM-Redis, the workload is run with flushing turned on and off to measure the overhead with cacheline flushing. All experiments are run on a commodity Xeon E5-1660v4 machine that features CLFLUSH only [1] and an i7-6600U machine that features CLFLUSHOPT [2]. CLFLUSHOPT optimizes away the implied fence with CLFLUSH that over-serializes cacheline flush operations.

We modify the code in *pmem.c* to turn cacheline flushing off, and measure the performance difference as compared to the default with cacheline flushing on, to isolate the cost of flushing. And, we isolate the cost of ordering as well on the i7-6600U machine that supports CLFLUSHOPT by turning fencing off in *pmem.c*. Similarly, we modify the code in *tx.c* to turn PMDK undo logging off, and measure the performance difference as compared to the default with undo logging on, to isolate the performance overhead incurred with logging.

3 RESULTS

In this section, we discuss the transactional overheads with PMDK *libpmemobj*, which includes the overheads of logging, persisting, and ordering in transactions. In addition, we also discuss the translational overhead to dereference persistent pointers for direct load and store accesses.

MEMSYS, October 1–4, 2018, Old Town Alexandria, VA, USA

ACM ISBN 978-1-4503-6475-1/18/10.
https://doi.org/10.1145/3240302.3240423

[1] CLWB or CLFLUSHOPT is not yet featured in latest Xeon E5
[2] CLWB is not yet featured in latest i7

3.1 Transactional Overheads

For trees and hashmaps, persisting and logging together incur *63%–72%* performance overhead for insertions and *61%–68%* overhead for deletions. Figure 1 shows the overhead for each operation when logging, persisting or/and ordering is turned on, the measured performance is normalized to the baseline transaction throughput with PMDK logging, flushing and ordering all turned off. Lookup operations are read-only, reads do not incur overhead, as logging and persisting are for writes only.

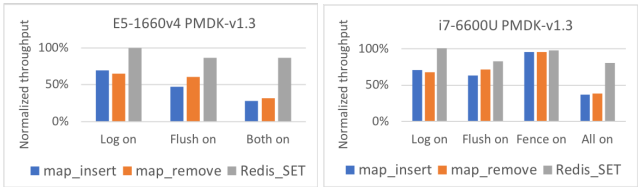


Figure 1: Overheads of logging, persisting and ordering in PMDK transactions

3.1.1 Logging Overhead. PMDK undo logging incurs an overhead of *29%–31%* for insertions and *32%–35%* for deletions with trees and hash maps. Lookups are reads only, reads do not incur logging overhead. Redis-SET doesn't incur logging overhead due to no logging featured in the PM-Redis implementation.

3.1.2 Persisting Overhead. Persisting incurs an overhead of *37%–53%* for insertions and *28%–39%* for deletions with trees and hash maps. Lookups are reads only, reads do not incur persisting overhead. Redis-SET incurs an overhead of *14%–17%*.

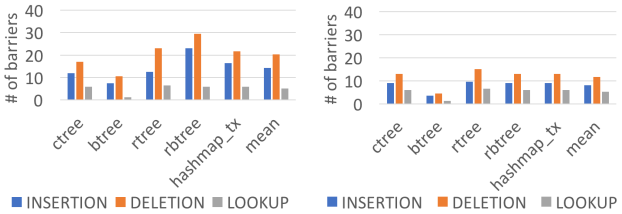
3.1.3 Ordering Overhead. Ordering incurs an overhead of *4%* for insertions and *5%* for deletions with trees and hash maps, and an overhead of *2%* with Redis-SET.

Each operation, i.e., insertion, deletion or lookup, has on average *15.7* synchronization barriers. Each operation is typically comprised of one PMDK transaction, sometimes the transaction triggers another transaction, for example, insertion or deletion may trigger re-balancing of balanced trees and resizing of hashmaps. Each PMDK transaction has on average *14.1* synchronization barriers.

For TPCC driven PMDK-enabled Redis, each TPCC transaction has on average *12* barriers, while each PMDK transaction has on average *7* barriers. The difference with trees and hashmaps is due to different read and write ratios.

Figure 2 shows the number of barriers with and without the undo logging in PMDK. Without undo logging in PMDK, each operation has on average *11* synchronization barriers, while each PMDK transaction has on average *10* synchronization barriers. Undo logging increases the number of barriers by *40%*.

As an example, for a *ctree* insert operation, the transaction begins with copying old data to the undo log with two barriers, and is followed by the commit phase with four barriers. The four barriers in the commit stage are for 1) allocating a new node 2) initializing the node with zeros 3) assigning values to the new node and inserting the new node to the tree and 4) changing commit status and log cleanup. Barriers are not only in transactional API implementations,



(a) With undo logging **(b) Without undo logging**

Figure 2: Number of barriers per operation

they are embedded in library calls within transactions, for example, *pmalloc* and *pmemset* contain barriers too as in 1) and 2).

3.2 Translation Overheads

Figure 3 shows how certain functions in the PMDK implementation contribute to the total number of executed instructions in persistent applications. The function *pmemobj_tx_add_range* adds the data to be mutated into the undo log, and the function *pmemobj_tx_commit* commits the transaction. The function *pmemobj_direct* converts a persistent pointer that is represented by a pool ID and an offset, where the pool ID is looked up in a cuckoo hashmap to get mapped to the start address and then added by the offset, to a naked pointer that can be dereferenced for direct load and store accesses. The persistent pointer to naked pointer conversation and vice versa constitute as the *translation overheads*. The translation overheads are significant for search operations at *25%* and *44%* of the total number of executed instructions, which is a significant cost for read-heavy persistent workloads. For insert and delete operations, the translation overhead is at *7%–9%* of the total executed instructions.

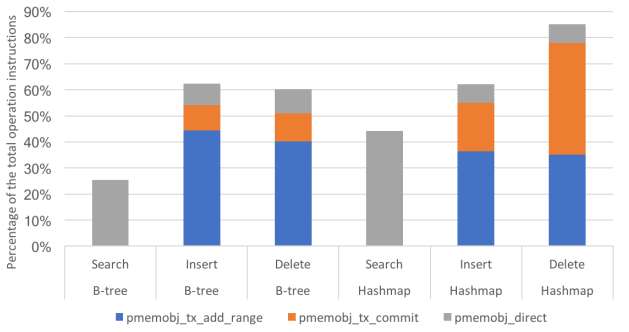


Figure 3: Translation overheads

4 CONCLUSIONS

The significant performance overheads of the PMDK implementation on current commodity hardware, as discussed in this paper, point to areas for potential hardware acceleration, i.e., logging, persisting, ordering and translation. Several efforts have already gone in this direction. Wang *et.al.*[8] accelerates undo logging in hardware. WHISPER [6] accelerates persisting with persistent buffers and introduces a *dfence* to reduce ordering overhead. Proteus [7] accelerates translation with special load and store instructions.

REFERENCES

[1] 2010. TPC Benchmark Standard Specification Revision 5.11. (February 2010). http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[2] 2017. Persistent Memory Atomics and Transactions. (January 2017). https://www.snia.org/sites/default/files/technical_work/Whitepapers/PM_Atomics_and_Transactions.pdf.

[3] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 433–452.

[4] Intel. [n. d.]. Non-Volatile Memory Library. ([n. d.]). Retrieved December 5, 2017 from https://github.com/pmem/nvml/

[5] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, Santa Clara, CA.

[6] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. 2017. 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*. ACM, 135–148.

[7] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. 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-50 '17)*. ACM, New York, NY, USA, 178–190.

[8] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. 2017. Hardware Supported Persistent Object Address Translation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 800–812.