Durability Through NVM Checkpointing

David T. Aksun
EPFL
Lausanne, Switzerland
david.aksun@epfl.ch

James R. Larus
EPFL
Lausanne, Switzerland
james.larus@epfl.ch

1 INTRODUCTION

After many years of anticipation, NVM became commercially available in 2019 with Intel's Optane DC Persistent Memory [11]. NVM allows building data structures that retain their contents to provide fast restart. One major issue is that processor caches are volatile and program data which reside in these caches are lost during power failure. In addition, the underlying hardware can evict caches out of order independent from the program order unless explicitly specified by the programmer. Both of these conditions can corrupt program data in NVM leading to a state that is not correct after restart from a power failure. The research community has focused its attention on building durable data structures [1, 2] with strict consistency guarantees where the data resides in NVM, in essence, using it as durable RAM.

However, to make the data structures crash-consistent, applications require ordering of the persistence of writes to make the changes durable to NVM in the correct order. Crash-consistency mechanisms rely on cache line flush (x86 clflushopt or clwb) and fence (x86 sfence) instructions to order the persistence of writes and ensure that the modifications are propagated in the right order. Explicitly ordering the persistence of writes can introduce overhead to the application [8] which can reduce the runtime performance.

Previous work [3, 9], which relied on periodic persistence optimized out the persistence of writes out of the critical path of the application by moving the persistence of writes to a batched checkpointing stage. Compared to traditional checkpointing, periodically persistent data structures provide fine-grained checkpointing intervals in the order of milliseconds and instead of dealing with page granularity, effectively benefits from the byte-addressability of NVM. In periodic persistence design, the data structure resides in DRAM. Periodically all the modifications propagate to the NVM by evicting the entire cache hierarchy. The periodic flushing of the cache line is accompanied by a crash-consistency mechanism such as multiversioning in Dali [9] or an optimized undo log in InCLL [3] to provide correct recovery after restart.

While periodic persistence approach is beneficial for removing the expensive cache line flush and fence instructions from the critical path of the application, the proposed methods bound the data to NVM and require careful design of the data structure. Firstly, the methods rely on a priviliged instruction to flush the entire cache hierarchy (x86 wbinvd) which not only creates challenges for multi-tenant systems but also can effect runtime performance by invalidating the entire cache hierarchy. Secondly the design is carefully tailored for each data structure which limits the general applicability.

Finally, DRAM is still faster than Optane. Yang [11] reported NVM sequential read latency is $2x$ DRAM and random read latency is $3x$. Write latency is similar to DRAM, except that memory writes occasionally execute $100x$ slower. However, peak NVM read bandwidth is approximately 6% of DRAM and peak write bandwidth is approximately 3% of DRAM. Using NVM as the primary memory reduces throughput by 39 – 56% on Masstree [7], a cache-efficient B+tree/Trie index (Figure 1). Adding a crash-consistency mechanism such as periodic persitence will lead to even lower performance leading to a slowdown of 50 – 64% as seen in the case of InCLL.

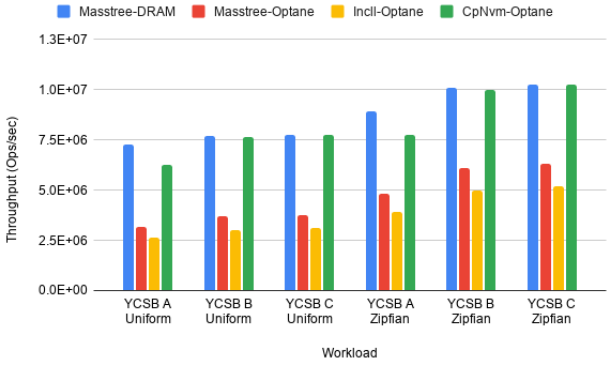

Figure 1: Throughput of Masstree data structure in DRAM and Optane compared to InCLL and *CpNvm* running on Optane.

We consider Optane as a critical optimization point and tailor the design specifically for the current hardware architectures. We propose to use processor's DRAM and caches to efficiently act as a cache (either write-through or write-back) for values persisted in NVM. Heavily read or written data should reside in DRAM, where it can be accessed at a fraction of the cost. Only modified values should be persisted in NVM. Most prior systems [1, 2, 5] used a write-through mechanism to NVM to implement a strong consistency model such as atomic transactions. As a consequence, these systems only use a processor's cache to reduce memory latency, not its DRAM. Ignoring this large memory severely limits the fraction of a data structure that can be quickly accessed and decreases overall performance.

More recent systems [6, 10] have switched to a write-back model in which a data structure resides in DRAM, and a redo log propagates changes to a persistent copy in NVM. Their (emulated) performance overhead is too high for general use, ranging from 43 – 122% for PMThreads [10] to 7.4 – 24.6% overhead on top of TinySTM's undocumented cost in DudeTM [6]. In addition, PMThreads is limited to lock-based parallel programs.

2 CHECKPOINTING DESIGN

This paper presents *CpNvm*, a runtime system that uses periodic checkpoints to maintain a recoverable copy of a program's data, with overhead low enough for widespread use. Our primary goal is low overhead, even at the cost of relaxing the crash-recovery model. We use checkpointing, rather than transactions, to permit recovery after a crash, which both increases performance and reduces the programming burden. Modifications performed in an uncommitted epoch are lost. A copy of a data structure to be kept persistent resides in DRAM, where it can be accessed and modified at low cost. *CpNvm* tracks modifications to the structure and periodically records changes to NVM, in a persistent log that is concurrently replayed to update the persistent copy of the structure.

To use *CpNvm*, a programmer inserts a library call at statements that write to a persistent data structure and another call when the structure is in a consistent state similar to application-level checkpointing. Identifying writes to a data structure require less insight into a program's semantics than identifying the sequence of statements that transition between consistent program states and hence should form a transaction. Moreover, many services are event-driven, so recognizing when they are idle is easy, and they typically support a mechanism for quiescing the system by deferring incoming work. When idle, it is easy to take a consistent snapshot.

CpNvm snapshots have four stages. The first is *execution*. *CpNvm* runtime provides DRAM-allocated memory called regions for storing persistent data. The program runs normally but must make additional calls to the runtime to track the locations in a region that are modified. The tracking mechanism uses a globally shared bitmap where the modified memory regions are marked and thread-local address lists which are useful to search modified memory locations within the bitmap fastly. The second is *checkpointing*, where *CpNvm* does coordinated checkpointing and snapshots the regions by to a thread-local redo log in NVM. Each thread can snapshot in parallel using both thread-local address lists and redo logs and better uses the NVM bandwidth due to sequential access pattern. When the log is complete, the program resumes execution. To deal with redo log write redirection, log size growth and the cost of replaying the entire redo log after restart, we keep an NVM copy of the data structure similar to DudeTM [6]. The third is *background replay*, in which *CpNvm* asynchronously replays the log on a persistent copy of the data in NVM. The fourth is *recovery*, which can occur after a crash. In this stage, *CpNvm* finishes replaying the persistent log if necessary. The NVM copy becomes the backing store for the program data structures residing in DRAM. *CpNvm* restarts the program, which lazily pages in the data from NVM at first reference to it.

For applications that can tolerate a more relaxed consistency model, checkpointing is a more attractive approach. Our approach is software-based and does not require the use of any hardware modifications such as introducing additional batteries which can leak. *CpNvm* can efficiently support data structures that are larger than the available DRAM capacity in the system using Intel Mixed Memory Mode. *CpNvm* allows a programmer the flexibility to trade performance against consistency by decreasing the size of an epoch (which increases logging overhead).

3 EVALUATION

CpNvm performs well with minimal program changes. For YCSB workloads on Masstree, we added 26 calls to the *CpNvm* library in a 27K LoC code base. As shown in figure 1 the overhead is less than 15% across all the YCSB workloads using 64ms intervals. For read-only workload YCSB-C the overhead is close to 0%. The main reason is that, the program is running the data structure in DRAM. The only cost for persistence is the cost of the modifications which are stored to NVM periodically. We also measured Memcached [4], which incurred only $0 - 6.5\%$ overhead ($0 - 100\%$ write workload, respectively) with 29 calls on *CpNvm* in 4.5K LoC using 200ms intervals.

4 CONCLUSION

The main result of this work is that existing and new techniques can be combined in a novel way that yields a simple checkpointing programming model that can be implemented with low overhead for NVM. We show the feasability of the *CpNvm* on both Masstree and Memcached using Optane.

REFERENCES

[1] 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)*. Association for Computing Machinery, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[2] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 105–118.

[3] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 441–454. https://doi.org/10.1145/3297858.3304046

[4] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux Journal* 124 (2004).

[5] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

[6] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 329–343. https://doi.org/10.1145/3093337.3037714

[7] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[8] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. 2018. Persistent Memory Transactions. (2018). arXiv:cs.DC/1804.00701

[9] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *31st International Symposium on Distributed Computing (DISC 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Andréa W. Richa (Ed.), Vol. 91. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 37:1–37:16. https://doi.org/10.4230/LIPIcs.DISC.2017.37

[10] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. 2020. PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 623–637. https://doi.org/10.1145/3385412.3386000

[11] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST '20)*. 169–182.