Fine-Grain Checkpointing with In Cache Line Logging[*]

Nachshon Cohen[†]
Amazon
Haifa, Israel
nachshonc@gmail.com

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

Hillel Avni
Huawei
Tel Aviv, Israel
hillel.avni@huawei.com

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

1 INTRODUCTION

Non-Volatile Memory (NVM) is fast, byte-addressable memory that preserves its contents after a power failure or a system crash. New technologies, such as 3D-XPoint [5], PCM [6, 12], STT-RAM [4] and ReRAM [1, 13], promise to provide NVM at a low cost, thus blurring the lines between storage and memory. In recent years, there has been a significant interest in constructing data structures and algorithms that exploit NVM's durability and speed.

NVM can enable a fast restart of a failed machine. On existing machines, restarting after a power failure requires a significant amount of time due to the need to load application information from durable media such as disk or SSD, parse it, and rebuild internal data structures. Storing a structure in NVM can avoid these costly steps. Since NVM is byte-addressable, it is possible to store and directly access efficient data structures, such as B+ trees or hash maps. After a crash, the structure is immediately available in NVM, enabling the system to resume quickly after rebooting.

The main challenge in utilizing NVM is that processor caches are transient. At a power failure, all structure modifications that have not been propagated from cache to NVM will be lost. An additional challenge is that cache lines are not written back to memory (NVM) in the order in which the application wrote them, but rather according to the low-level and undocumented cache replacement policy of the memory system. Thus, it is entirely possible that if a single object spans two cache lines, half of the object may be up-to-date in NVM while the other half's value is stale. This creates a well-studied challenge: how to ensure that a durable copy of a data structure is well-formed (consistent) after a crash, even though NVM may contain a mixture of old and new cache lines?

To solve this challenge, most NVM systems use programmer-specified *fine-grained transactions*, which guarantee that all memory writes in a transaction reach NVM or none of them do. A modification to a data structure is first logged (using either a redo or undo log) to NVM and then applied to the data structure. The log provides sufficient information for the recovery process, regardless of whether all of the data structure modifications reach NVM. However, the system must ensure that the log is completely updated in NVM before the data structure is modified, by using a cache write flush instruction. These instructions, e.g., the Intel x64 `clflushopt` or `clwb`, transfer dirty cache lines from the (transient) cache to the (durable) NVM. These instructions incur a high overhead cost since — by definition — they require a full trip to NVM. This overhead is significant since it occurs on the application's critical path before a data structure modification completes. Some NVM systems reduce the write-back overhead by modifying the data structure in transient memory and lazily propagating modifications to NVM [7, 10]. However, these solutions come at the expense of restart time since a transient copy must be created, by copying from NVM, before the first modification to the data structure can take place. For large data structures, e.g. terabytes, copying can take many minutes since memory bandwidth is limited to tens of GB/sec.

An alternative to fine-grain transactions and logging is checkpointing, which is another widely used technique for ensuring persistence. At periodic intervals, an application's entire state is saved on durable media. After a failure, the last recorded checkpoint is restored into memory and the computation resumes from this point, requiring the re-execution of the work done between the checkpoint and failure. The interval between checkpoints is a tradeoff between the overhead of recording a checkpoint and the cost of re-executing the lost computation. Since a checkpoint typically copies the entire state of an application to a slow durable media, most systems take checkpoint at long intervals (minutes to hours) to reduce this cost.

Although it is possible to halt all program activity while a checkpoint is being taken, this can significantly reduce an application's performance. Therefore, many systems implement lazy checkpoints, allowing the application to progress while the checkpoint is recorded. To avoid corrupting the memory state, an undo log is used. The first modification to a page (or another unit of memory) is preceded by storing a copy of the original data into the checkpoint. Thus, the checkpoint state never contains any modifications that occurred after the checkpoint started.

In this work, we use *Fine-Grained Checkpointing* (Nawab calls a similar approach "periodic persistence" [11]) that uses frequent checkpoints to NVM to ensure persistence at low cost. Instead of ensuring that every memory write propagates to NVM as rapidly as possible, we partition an application's execution into epochs and ensure that after a crash, data structures can be restored to their state at the end of the most recently completed epoch. Our system flushes the processors' caches to NVM at the end of an epoch, thus ensuring that at this point NVM contains *all* modified data.

This approach offers many advantages. The number of cache lines that must be flushed is bounded by the cache size, and modified cache lines could have been written back during the execution of the epoch, so the cost of flushing the cache (recording the checkpoint) is low. Furthermore, the modified lines are flushed in a batch by hardware, further reducing the cost. Our approach's low cost allows short checkpoint epochs, e.g., tens of milliseconds (we use 64 milliseconds), thereby reducing both the potential data loss and the recovery time. In addition, a software developer does not have to annotate the application to identify fine-grain transactions, rather he or she only needs to ensure that the application is at a consistent state at the end of an epoch.

Our approach also differs from a traditional checkpoint in that there is no distinct in-memory copy of a data structure or memory image. The in-NVM data structure essentially serves as its own

[*]Paper appeared in ASPLOS 2019, April 2019. https://doi.org/10.1145/3297858.3304046
[†]Work done while the first author was a postdoc at EPFL.

durable checkpoint. The challenge is to keep the checkpoint recoverable during an epoch, as execution modifies this data structure. After a crash, its NVM state will consist of a mixture of the checkpointed state from the previous epoch, which must be kept, and modifications from the failed epoch, which should be discarded.

The system must be able to distinguish between these intermixed states and recover a consistent image from the previous epoch. A solution to this intermixing problem is to log the old value before each modification. The log can be applied, in reversed application order, to roll back the modifications and to revert to the state at the end of the previous epoch. But, as we saw with fine-grain transactions, logging requires that the writes to the log reach NVM before the data structure is modified. Therefore, it again introduces the cost of flushing the cache on the application's critical path.

To solve the problem of fine-grained modifications in NVM, we introduce the novel concept of an *In Cache Line Log* (*InCLL*). InCLL is similar to an undo log. But instead of using an external log, the InCLL is placed *in the same cache line* as the modified data structure field. InCLL relies on the Persistent Cache Store Order (PCSO) memory-ordering model of NVM (two writes to the same cache line reach NVM in program order) to ensure the ordering requirements of the log, without introducing cache flushes and delays on the critical path.

The main limitation of the InCLL is capacity. Since it resides in the same cache line as the data, an InCLL should be small and cannot handle all modifications. If an object is modified multiple times during an epoch, InCLL may be insufficient to ensure crash recoverability. In this case, the approach falls back on object-level logging. After the entire object is logged, subsequent modifications to it do not require additional actions. Overall, the combination of InCLL and object-level logging drastically reduces the number of synchronous writes to NVM on the critical path of the application.

We incorporated our Fine-Grained Checkpointing algorithm into Masstree [9], a high-performance combination of a B+ tree and Trie. Measurements show that the overhead of our scheme is minimal (generally less than 10%) and restart time is dramatically reduced.

1.1 Persistent Memory Ordering Model

In this paper, we use the Persistent Cache Store Order (PCSO) memory-ordering model for NVM [3]. Cache lines are written back to NVM according to a computer's (unspecified) cache replacement policy. Cache lines can be written at any order, and we do not assume any specific behavior of the cache. The application may also explicitly force specific cache lines to be written NVM, by using special cache-line write-back instruction, such as x64's `clflushopt` or `clwb`. These instructions are asynchronous: due to the high and variable cost of reaching memory, they only initiate the memory transfer but do not wait until the data actually reaches NVM. To ensure they complete, an application must issue a fence instruction, such as `sfence`, which delays CPU execution until the write-back instructions finish. Since the combination of write back and fence instructions wait until the data reaches NVM, they are expensive to execute on the critical path of an application.

While ordering different cache lines is expensive, ordering writes to the same cache line is essentially free. If two writes target the same cache line, it is sufficient to preserve the order in which they reach *the cache* to ensure they reach NVM in the same order. Preserving the order of cache writes can be done with the release memory ordering in C++11, which introduces a *happens-before* relation between the writes [2, 8]. On the x64 architecture, the release memory fence incurs *no* runtime overhead and only limits the ability of the compiler to reorder writes.

Formally, given two writes X and W, we say that $X <_p W$ if X is written to persistent memory no later than W. $X <_{hb} W$ is the standard *happens before* relationship. $c(X)$ represents the cache line address X writes to. The following holds [3]:

- $W <_{hb} \textit{writeback}(c(W)) <_{hb} \textit{sfence} <_{hb} X \Rightarrow W <_p X$ (explicit flush).
- $W <_{hb} X \wedge c(W) = c(X) \Rightarrow W <_p X$ (granularity).

Our InCLL technique relies on the second ordering guarantee. It specifies that if two writes target the same cache line, a happens-before relation is sufficient to ensure persistence ordering.

REFERENCES

[1] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proc. IEEE* 98, 12 (dec 2010), 2237–2251. https://doi.org/10.1109/JPROC.2010.2070830

[2] Hans-J. Boehm, Sarita V. Adve, Hans-J. Boehm, and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proc. 2008 ACM SIGPLAN Conf. Program. Lang. Des. Implement. - PLDI '08*, Vol. 43. ACM Press, New York, New York, USA, 68. https://doi.org/10.1145/1375581.1375591

[3] Nachshon Cohen, Michal Friedman, and James R Larus. 2017. Efficient Logging in Non-Volatile Memory by Exploiting Coherency Protocols. In *Proc. 2017 ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Lang. Appl.*

[4] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. 2005. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. In *IEEE Int. Devices Meet. 2005. IEDM Tech. Dig.* IEEE, 459–462. https://doi.org/10.1109/IEDM.2005.1609379

[5] Intel. 2015. Intel and Micron Produce Breakthrough Memory Technology. (2015). https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/#gs.ktnOlUHc

[6] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proc. 36th Annu. Int. Symp. Comput. Archit. - ISCA '09*, Vol. 37. ACM Press, New York, New York, USA, 2. https://doi.org/10.1145/1555754.1555758

[7] 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. In *Proc. Twenty-Second Int. Conf. Archit. Support Program. Lang. Oper. Syst. - ASPLOS '17*, Vol. 45. ACM Press, New York, New York, USA, 329–343. https://doi.org/10.1145/3037697.3037714

[8] Jeremy Manson, William Pugh, Sarita V. Adve, Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Proc. 32nd ACM SIGPLAN-SIGACT sysposium Princ. Program. Lang. - POPL '05*, Vol. 40. ACM Press, New York, New York, USA, 378–391. https://doi.org/10.1145/1040305.1040336

[9] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proc. 7th ACM Eur. Conf. Comput. Syst. - EuroSys '12.* ACM Press, New York, New York, USA, 183. https://doi.org/10.1145/2168836.2168855

[10] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proc. Twelfth Eur. Conf. Comput. Syst. - EuroSys '17.* ACM Press, New York, New York, USA, 499–512. https://doi.org/10.1145/3064176.3064215

[11] 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 Int. Symp. Distrib. Comput. - DISC 2017*, Vol. 91. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPICS.DISC.2017.37

[12] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Comput. Archit. News* 37, 3 (jun 2009), 24. https://doi.org/10.1145/1555815.1555760

[13] H.-S. Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T. Chen, and Ming-Jinn Tsai. 2012. Metal-Oxide RRAM. *Proc. IEEE* 100, 6 (jun 2012), 1951–1970. https://doi.org/10.1109/JPROC.2012.2190369