Clobber-NVM: Log Less, Re-execute More

Yi Xu[1], Joseph Izraelevitz[2], and Steven Swanson[1]

[1]UC San Diego
[2]University of Colorado, Boulder

Abstract

Non-volatile memory allows direct access to persistent storage via a load/store interface. However, because the cache is volatile, cached updates to persistent state will be dropped after a power loss. Failure-atomicity NVM libraries provide the means to apply sets of writes to persistent state atomically. Unfortunately, most of these libraries impose significant overhead.

This work proposes Clobber-NVM, a failure-atomicity library that ensures data consistency by reexecution. Clobber-NVM's novel logging strategy, clobber logging, records only those transaction inputs that are overwritten during transaction execution. Then, after a failure, it recovers to a consistent state by restoring overwritten inputs and reexecuting any interrupted transactions. Clobber-NVM utilizes a clobber logging compiler pass for identifying the minimal set of writes that need to be logged. Based on our experiments, classical undo logging logs up to 41.5× more bytes than Clobber-NVM, and requires 2× to 6.7× more expensive ordering instructions. Less logging leads to better performance: Relative to prior art, Clobber-NVM provides up to 2.2× performance improvement over Mnemosyne, 2.7× over Intel's PMDK, and up to 6.2× over HP's Atlas.

1. Motivation

Non-volatile memory exposes persistent storage via a byte-addressable load/store interface. However, because the cache is volatile, cached updates to persistent state will be dropped after a power loss.

Failure-atomicity NVM libraries provide the means to apply sets of writes to persistent state atomically. These libraries provide failure-atomicity for specified code regions: for writes within a specified code region, all writes will survive a power loss and be written to NVM, or none will. These transaction-like, "all-or-nothing," semantics make programming on NVM easier and hide architectural and caching details from programmers. Unfortunately, most current libraries impose significant overhead.

2. Limitations of the State of the Art

Most industrial failure atomicity systems use undo logging [21, 3]. In undo logging systems, where incomplete transactions are undone by applying undo log entries, the log entry must be persistent before the corresponding data update, which results in excessive persistence ordering constraints. In contrast, redo logging systems only have to persist the logs at transaction commit, but they need to intercept and redirect reads [24, 12, 9]. Prior attempts to reduce the logging cost either rely on special hardware [15, 14, 1, 10, 7, 22], target specific data structures [5, 20, 8, 26, 4, 23], double memory consumption by maintaining an additional working set shadow copy [19, 6, 17, 2, 25], impose limitations on programming model [13, 18] or relax the isolation model [11, 16].

Among these systems, JUSTDO logging [13] proposed *recovery-via-resumption*. In contrast to undo logging, JUSTDO logging tracks enough program state to resume a failure-atomic operation at recovery, resuming execution from the interrupted instruction. Subsequent work in iDO logging [18] dramatically reduces program state logging frequency by exploiting idempotent code regions (a segment of code that does not overwrite its input). However, their runtime overhead remains quite high, and they strictly limit volatile data usage during a failure-atomic operation.

3. Key Insight

Clobber-NVM ensures failure-atomicity by reexecuting any interrupted transactions at recovery and relies on a new logging method — clobber logging. Clobber logging's key insight is that logging **only** overwritten inputs is sufficient to reexecute a transaction **with the exact same results**. We call a transaction input a *clobbered input* if it may be overwritten within the transaction, and term this write a *clobber write*. Clobbered inputs are a problem for reexecution: If an input is clobbered during transaction execution, reexecuting the code will use a new value for the input.

The observation behind clobber logging is that clobber writes break reexecution — but undo logging them in a *clobber log* can preserve the clobbered inputs. **Values at other addresses never need to be logged** in clobber logging, since they will be overwritten upon reexecution. Because inputs which are not clobbered are expected to be available after a failure, and clobbered inputs are preserved in the undo log, clobber logging is sufficient for recovery.

Clobber logging drastically reduces the cost of failure atomicity. Clobber logging only logs a small set of selected values — the clobbered inputs, whereas undo logging must log before every writes in a transaction. Relative to prior recovery-via-resumption systems, clobber logging reduces log frequency. It only logs before a clobber write happens, while JUSTDO logs at every store and iDO logs at every idempotent region boundary and those regions tend to be small [18].

4. Main Artifacts

Our main artifact is Clobber-NVM, a combined compiler/runtime library that enforces failure atomicity using clobber logging.

The Clobber-NVM compiler extension, built on top of LLVM, is used to identify clobber writes within transactions using dependency analysis. It first identifies all possible input reads and all possible clobber writes as candidates for clobber logging. However, due to ambiguity in the analysis, it may conservatively over-identify clobber writes leading to excessive logging. Our compiler analysis subsequently refines the result through novel analysis propagation. The propagation reduces Clobber-NVM's logging cost by avoiding logging at writes that will never actually be clobber writes at runtime.

To support volatile data usage inside transactions, Clobber-NVM uses a separate log (the *vlog*) to store minimal volatile data needed to re-construct a transaction's volatile input.

The Clobber-NVM runtime manages the clobber log and the vlog. Our clobber log is built on PMDK's undo log API. This design choice leaves Clobber-NVM's clobber log very simple. Clobber-NVM manages the vlog directly.

The Clobber-NVM runtime also manages recovery of interrupted transactions after a crash. Figure 1(top row) shows a transaction progressing through normal execution. A transaction interrupted by a crash follows a different path. As with normal execution, the transaction starts with initialized inputs and untouched outputs (Figure 1, top left), then progresses through execution by writing to some output addresses(Figure 1, top center), including clobber writes.

However, a power failure during execution drops the transaction to the recovery path. After the power loss, the transaction loses all volatile memory and some NVM outputs that still resided in the machine's (volatile) caches (Figure 1, bottom left). At restart, Clobber-NVM restores both the volatile and clobbered inputs using the transaction's logs, though the outputs may still be inconsistent (Figure 1, bottom center). With its inputs restored, the transaction is reexecuted from the beginning (Figure 1, bottom right).

Once the transaction executes past the point when the failure happened, the transaction has overwritten any incomplete outputs, erasing any inconsistencies caused by the power loss (Figure 1, top center). The transaction will continue to progress to completion and commit (Figure 1, top right).

5. Key Results

Figure 2 shows memcached performance on Clobber-NVM. We make the following observations. First, Clobber-NVM always outperforms Intel's PMDK [21] and Mnemosyne [24], two prior failure atomicity systems. It provides, on average, 1.6× and 1.4× of PMDK and Mnemosyne performance, respectively. Second, Clobber-NVM outperforms PMDK and Mnemosyne more on write intensive workloads, because they involve more logging operations. Third, on single thread workloads, Clobber-NVM outperforms PMDK and Mnemosyne by up to 1.7× and 2.2×, respectively. Because this version of memcached uses a global lock to provide concurrency, Clobber-NVM and PMDK, as lock-based systems, scales poorly compared to Mnemosyne. We see their scalability and performance improve on data structures with a finer-grain locking scheme (see Figure 6 in the full paper).

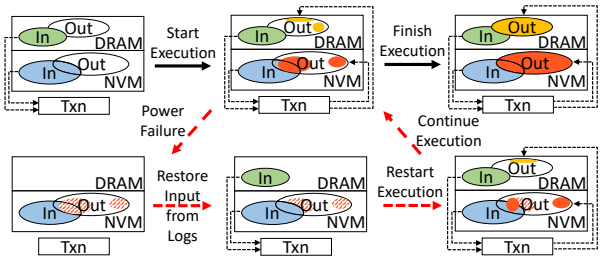

Figure 1: **Recovery Process of One Transaction.** The *In* and *Out* indicate addresses, instead of values — an output address may be updated several times during the transaction.

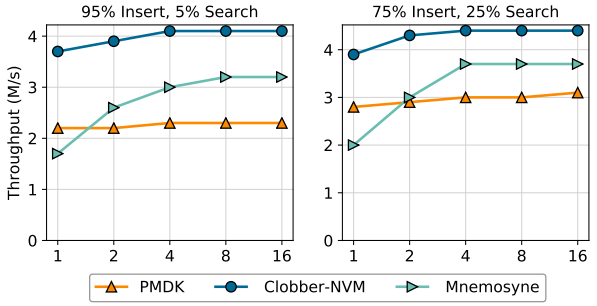

Figure 2: **Memcached Performance with 16-byte Keys and 64-byte Values**

6. Contributions

This paper makes the following contributions:

- It presents *clobber logging*, a novel, recovery-via-resumption strategy that reduces log size by only recording overwritten transaction inputs.
- It presents a clobber logging compiler pass for identifying those transaction inputs that need to be logged.
- It introduces Clobber-NVM, a compiler-based failure-atomicity solution based on recovery-via-resumption.
- It demonstrates that Clobber-NVM's performance compares favorably with the existing state-of-the-art systems, showing up to 2.2× improvement over Mnemosyne and 2.7× over Intel's PMDK, and up to 6.2× improvement over HP's Atlas. It shows Clobber-NVM's logging strategy reduces log size by 1.02× to 41.5× and required expensive ordering fences by 2× to 6.67× relative to Intel's PMDK.

References

[1] Miao Cai, Chance C Coats, and Jian Huang. Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 584–596. IEEE, 2020.

[2] Daniel Castro, Paolo Romano, and João Barreto. Hardware transactional memory meets memory persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 368–377, 2018.

[3] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, 2014.

[4] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[5] Nachshon Cohen, David T Aksun, Hillel Avni, and James R Larus. 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*, pages 441–454, 2019.

[6] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 271–282, 2018.

[7] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. Lazy release persistency. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1173–1186, 2020.

[8] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. *ACM SIGPLAN Notices*, 53(1):28–40, 2018.

[9] Ellis R Giles, Kshitij Doshi, and Peter Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.

[10] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. Relaxed persist ordering using strand persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 652–665. IEEE, 2020.

[11] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 913–928, USA, 2019. USENIX Association.

[12] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 468–482, 2017.

[13] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, New York, NY, USA, 2016. ACM.

[14] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 399–411, 2016.

[15] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M Chen, and Thomas F Wenisch. Delegated persist ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[16] R Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 335–349, 2020.

[17] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. *ACM SIGPLAN Notices*, 52(4):329–343, 2017.

[18] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270. IEEE, 2018.

[19] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 499–512, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B Morrey III, Dhruva R Chakrabarti, and Michael L Scott. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[21] pmem.io. Persistent Memory Development Kit, 2017. http://pmem.io/pmdk.

[22] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 175–186, 2017.

[23] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, USA, 2011. USENIX Association.

[24] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS '11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2011. ACM.

[25] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. Pmthreads: persistent memory threads harnessing versioned shadow copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 623–637, 2020.

[26] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 167–181, 2015.