



iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory

Qingrui Liu*, Joseph Izraelevitz[†], Se Kwon Lee[§], Michael L. Scott[‡], Sam H. Noh[§], and Changhee Jung*

*Virginia Tech
Blacksburg, VA, USA
{lqingrui,chjung}@vt.edu
 <https://orcid.org/0000-0002-6422-6549>

[†]UC San Diego
San Diego, CA, USA
jizraelevitz@eng.ucsd.edu

[§]UNIST
Ulsan, Korea
{sekwonlee,samhnoh}@unist.ac.kr

[‡]University of Rochester
Rochester, NY, USA
scott@cs.rochester.edu
 <https://orcid.org/0000-0001-8652-7644>

Abstract—This paper presents *iDO*, a compiler-directed approach to failure atomicity with nonvolatile memory. Unlike most prior work, which instrument each store of persistent data for redo or undo logging, the *iDO* compiler identifies *idempotent* instruction sequences, whose re-execution is guaranteed to be side-effect-free, thereby eliminating the need to log every persistent store. Using an extension of our prior work on JUSTDO logging, the compiler then arranges, during recovery from failure, to back up each thread to the beginning of the current idempotent region and re-execute to the end of the current failure-atomic section. This extension transforms JUSTDO logging from a technique of value only on hypothetical future machines with nonvolatile caches into a technique that also significantly outperforms state-of-the-art lock-based persistence mechanisms on current hardware during normal execution, while preserving very fast recovery times.

I. INTRODUCTION

With the emergence of fast, byte-addressable nonvolatile memory, we can now conceive of systems in which main memory, accessed with ordinary loads and stores, is “always available,” and need not be flushed to the file system to survive a crash. The obvious use case of such a technology is to allow programmers to store heap objects persistently in memory, bypassing the expensive serialization of those objects onto traditional storage devices. Unfortunately, from the perspective of crash recovery, nonvolatile main memory is compromised by the fact that caches can write data back to memory in arbitrary order, leading to inconsistent values after a crash.

In order to avoid such errors and ensure post-crash consistency of persistent data, researchers have developed failure-atomicity systems that allow programmers to delineate failure-atomic operations on the persistent data—typically in the form of transactions or *failure-atomic sections* (FASEs) protected by outermost locks [1], [2] (our own work is based on FASE-based locking). Given knowledge of where operations start and end, the failure-atomicity system can ensure, via logging or some other approach, that all operations within the code region happen atomically with respect to failure and maintain the consistency of the persistent data.

At Virginia Tech, this work was supported by NSF grants 1750503 (CAREER), 1527463, and 1814430, and by Google/AMD Faculty Research awards. At the University of Rochester, this work was supported by NSF grants 1319417, 1337224, 1422649, and 1717712, and by a Google Faculty Research award. At UNIST, this work was supported by the Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1402-52.

To simplify the management of logs for FASE-based persistence, Izraelevitz et al. introduced the notion of JUSTDO logging [2]. The JUSTDO system logs enough information to *resume* a FASE during recovery and execute it to completion (“recovery via resumption”). Immediately prior to each store instruction in a FASE, the JUSTDO system logs (in persistent memory) the program counter, the to-be-updated address, and the value to be written. During recovery, the system uses the code of the crashed program to complete the remainder of each interrupted FASE, beginning with the most recent log entry.

The problem with JUSTDO logging is its requirement that the log be written *and made persistent* before the related store—an expensive requirement to fulfill on machines with volatile caches. The key contribution of our work is to demonstrate that recovery via resumption can be made efficient on machines with volatile caches and expensive persist fences. The key is to arrange for each log operation (and in particular each persist fence) to cover multiple store instructions of the original application. We achieve this coverage via compiler-based identification of *idempotent* instruction sequences. More precisely, the compiler divides each FASE into a series of idempotent regions, so that each instruction of the FASE belongs to exactly one region. Because an idempotent region of code can safely be re-executed an arbitrary number of times without changing its output, the recovery procedure in the wake of a crash can resume execution at the beginning of the current region, eliminating the need to log each individual store instruction of the original program.

This extended abstract introduces *iDO*, a practical compiler-directed failure-atomicity system. Like JUSTDO logging, *iDO* supports fine-grained concurrency through lock-based FASEs, and avoids the need to track dependences by executing forward to the end of each FASE during post-crash recovery. Unlike JUSTDO, *iDO* allows the use of registers in FASEs, and persists its stores at coarser granularity.

Instead of logging information at every store instruction, *iDO* logs (and persists) a slightly larger amount of program state (registers, live stack variables, and the program counter) at the beginning of every idempotent code region within the overall FASE. In practice, idempotent sequences tend to be significantly longer than the span between consecutive stores—tens of instructions in our benchmarks; hundreds or even thousands of instructions in larger applications [3]. As *iDO* is implemented in the LLVM tool chain [4], our implementation

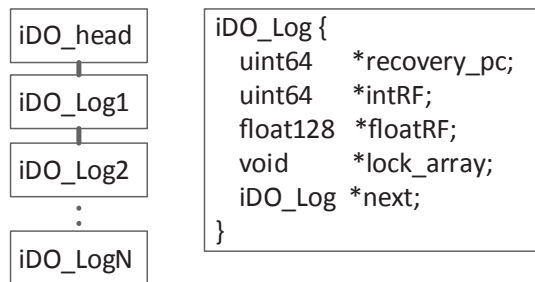


Fig. 1: iDO log structure and management: the number of iDO logs matches the number of threads created.

is also able to implement a variety of important optimizations, logging significantly less information—and packing it into fewer cache lines—than one might naively expect.

II. iDO FAILURE ATOMICITY SYSTEM

Unlike UNDO or REDO logging, iDO logging provides failure atomicity via *resumption* and requires no log for individual memory stores. Once a thread enters a FASE, iDO must ensure that it completes the FASE, even in the presence of failures. At the beginning of each idempotent code region in the body of a FASE, all inputs to the region are known to have been logged in persistent memory. Since the region is idempotent, the thread never overwrites the region’s inputs before the next log event. Consequently, if a crash interrupts the execution of the idempotent region, iDO can re-execute the idempotent region from the beginning using the persistent inputs.

For each thread, the iDO runtime creates a structure called the `iDO_Log`, held in a linked list pointed to by the (persistent) `iDO_head`. Each `iDO_log` structure comprises four key fields. The `recovery_pc` field points to the initial instruction of the current idempotent region. The `intRF` and `floatRF` fields hold live-out register values. Finally, the `lock_array` field holds *indirect lock addresses* for the mutexes owned by the thread.

Here then is the series of steps required, within a FASE, to complete the execution of idempotent region r and begin the execution of region s :

- 1) Issue write-back instructions for all output registers of r (saving them to `intRF` and `floatRF`) and for all output values in the stack. Together, these comprise $OutputSet_r$. Note that live-out values that were not written in r are already sure to have persisted; no additional action is required.
- 2) Update `recovery_pc` to point to the beginning of s . Once this step is finished, s can be re-executed to recover from failures that occur during its execution.
- 3) Execute the code of s , generating the values in $OutputSet_s$. These values will be persisted at the end of s —i.e., at the boundary between s and its own successor t , as described in step 1. Note that by definition an idempotent region will never overwrite its own input.

Recovery for iDO comprises the following general steps:

- 1) On process restart, iDO detects the crash and retrieves the `iDO_Log` linked list.

- 2) iDO creates a recovery thread for each entry in the log list.
- 3) Each recovery thread reacquires the locks in its `lock_array`, then synchronizes at a barrier.
- 4) Each recovery thread restores its registers (including the stack pointer) from its iDO log, and jumps to the beginning of its interrupted idempotent region.
- 5) Each thread executes to the end of its current FASE, at which point no thread holds a lock, recovery is complete, and the recovery process can terminate.

III. EVALUATION

For evaluation, we compared iDO against several other failure atomicity runtimes [1], [2], [5] on the **Redis** [6], [7] key-value store (for further results, see the full paper [8]). As shown in Figure 2, iDO outperforms existing persistence systems by significant margins for all key ranges, with overhead of 30–50% relative to the crash-vulnerable code. As Redis has long FASEs with relatively few persistent writes, iDO can take significant advantage of idempotent regions.

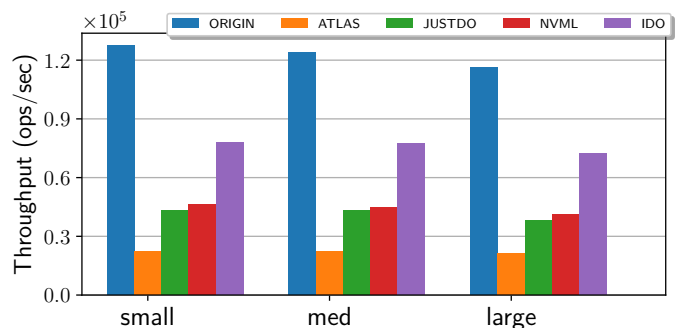


Fig. 2: Redis throughput for databases with 10K, 100K, and 1M-element key ranges using Redis’s “lru” test (80/20 read/write mix with power-law key distribution).

REFERENCES

- [1] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, “Atlas: Leveraging locks for non-volatile memory consistency,” in *Intl. Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, Portland, OR, 2014, pp. 433–452.
- [2] J. Izraelevitz, T. Kelly, and A. Kolli, “Failure-atomic persistent memory updates via JUSTDO logging,” in *21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, 2016, pp. 427–442.
- [3] M. A. de Kruijf, K. Sankaralingam, and S. Jha, “Static analysis and compiler design for idempotent processing,” in *33rd ACM Conf. on Programming Language Design and Implementation (PLDI)*, Beijing, China, 2012, pp. 475–486.
- [4] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Intl. Symp. on Code Generation and Optimization (CGO)*, San Jose, GA, 2004, pp. 75–88.
- [5] Intel, “Intel NVM Library,” <https://github.com/pmem/nvml/>.
- [6] RedisLabs, “Redis,” 2015, <http://redis.io>.
- [7] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, “An analysis of persistent memory use with WHISPER,” in *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi’an, China, 2017, pp. 135–148.
- [8] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, “iDO: Compiler-directed failure atomicity for nonvolatile memory,” in *51st IEEE/ACM Intl. Symp. on Microarchitecture*, ser. MICRO ’18, Fukuoka, Japan, 2018.