Zero-Overhead NVM Crash Resilience

Faisal Nawab[*,†] Dhruva Chakrabarti[†]
Terence Kelly[†] Charles B. Morrey III[†]
[*]CS Dept., UC Santa Barbara [†]HP Labs, Palo Alto, CA

Abstract

Byte-addressable non-volatile memory (NVM) allows fine-grained in-place update of durable data. NVM transaction mechanisms prevent failures during updates from corrupting data, but such mechanisms carry substantial performance overheads. Our new alternative for high-performance multi-threaded software guarantees consistent recovery of application data following failure and has *zero* overhead during failure-free operation. Our approach preserves application data integrity in crash-injection experiments and its performance rivals state-of-the-art NVM transactions.

Introduction

Density scaling limitations cast doubt on the long-term viability of DRAM [10]. The most likely replacement technologies retain data without power, i.e., they provide byte-addressable non-volatile memory (NVM) [2]. HP Labs is developing memristor memories [18] within a larger effort to build a next-generation computer [9].

Familiar durability interfaces such as file systems and databases will survive into the NVM era [5, 15]. However, NVM on the memory bus invites a fundamentally new way to manipulate durable data: via LOAD and STORE instructions. In contrast to indirect block-granularity modifications mediated by complex software stacks, this new "NVM style of programming" involves direct, fine-grained, in-place updates. Today's applications use separate data formats for memory and storage, requiring cumbersome translation between formats. NVM-style programs use only an in-memory format, simplifying software and improving performance.

Of course, failures such as process crashes, OS kernel panics, and power outages during NVM updates can corrupt application data. Realizing the full value of NVM-style programming requires mechanisms to preserve application data integrity across NVM updates in the presence of failures. Below we review existing approaches to this problem, outline our new approach, and evaluate its effectiveness. Our tech report provides more detail [12].

Related Work

Whole-system persistence (WSP) tolerates power outages by using residual energy in the system power supply to flush CPU state into NVM, thus transforming a power failure into a suspend/resume [11]. One drawback is that peripheral devices can be reset by a power outage unbeknownst to their drivers, which may cause errors when computation resumes. Another limitation is that WSP does not tolerate software bug crashes (kernel panics and process crashes), which are common in practice.

Transactional file update tolerates process, kernel, and machine crashes [13, 16]. It furthermore supports higher abstractions such as persistent heaps, which facilitate NVM-style programming and which can slide beneath complex legacy software with remarkable ease. For example, we crash-proofed HP Indigo printers by retrofitting transactional file update beneath their control software [1]. Despite its attractions, transactional file update targets block storage rather than byte-addressed NVM.

NVM transaction mechanisms include Mnemosyne, which adds durability semantics to a software transactional memory [17]. NV-Heaps leverages hardware support to implement atomic NVM updates and provides type-safe pointers and garbage collection [4]. Our Atlas system automatically infers consistent states of a persistent heap from applications' use of synchronization primitives (e.g., mutexes) and uses UNDO logs to ensure that recovery can restore a consistent heap [3]. These NVM transaction mechanisms provide the full benefits of NVM-style programming, but they incur performance overheads from transaction logging.

NVM Transaction Overheads

NVM transaction overhead largely stems from forcing data from volatile CPU caches to NVM (e.g., via cache line flushes). For example, before allowing an in-place update of NVM within a transaction, Atlas must first ensure that an UNDO log entry has reached NVM; before deleting an UNDO log Atlas must force the in-place updates into NVM [3]. We can eliminate the need to force data into NVM by borrowing an insight from whole-system persistence: *flush-on-failure* can replace flush-as-you-go. We need not insist that data *has reached* NVM during failure-free operation if instead we are assured that the data *will reach* NVM in the event of failure [12].

Different failures require different flush-on-failure mechanisms. Tolerating power outages requires sufficient standby power for orderly system shutdown. Fortunately, NVM dramatically reduces the cost compared to DRAM: The time and energy required to flush CPU caches to NVM is orders of magnitude smaller than would be required to dump volatile DRAM to block storage. Narayanan & Hodson report that a typical computer's power supply contains sufficient residual energy for this task [11]. Tolerating OS kernel panics requires the kernel panic handler to flush all CPU caches to NVM before halting the system. An HP team has modified

the Linux kernel to do this on x86 systems, which required a small amount of fairly simple code.

Tolerating process crashes is surprisingly easy: Even on conventional hardware and on OSes such as Linux, cached modifications to shared file-backed memory mappings—which outlive the crashed process—are immediately visible to the file's readers and will eventually find their way down through the CPU cache and page cache to the backing file. The details are somewhat involved (see Appendix A of [12]), but the upshot for NVM transaction systems is that flush-on-failure support for tolerating process crashes is already available and will remain so on NVM-based hardware. Extensive experiments confirm that Atlas tolerates process crashes *without* cache flushing, greatly reducing Atlas transaction overhead [12]. Unfortunately, even if flush-on-failure allows us to eliminate CPU cache flushing from NVM transaction systems, transaction logging overhead remains.

Zero-Overhead Atomic Updates

Our main contribution is a crash resilience technique that avoids all of the performance overheads of existing NVM transaction mechanisms because it performs no logging. Our technique combines flush-on-failure with a class of *multi-threaded isolation* mechanisms into a *consistent recovery* mechanism.

Non-blocking algorithms ensure orderly (e.g., race-free) multi-threaded access to shared memory and guarantee that the suspension or termination of some threads cannot prevent others from making useful progress [7]. Conventional mutual exclusion cannot support non-blocking algorithms because a thread that loops infinitely or terminates while holding a mutex can prevent all other threads from making progress. Non-blocking algorithms instead rely on atomic CPU instructions such as compare-and-swap. We employ lock-free and wait-free sub-species of non-blocking algorithms, using the latter term for brevity.

Our technique is to combine flush-on-failure with non-blocking algorithms: Consistent data recovery will always succeed following the abrupt termination of a program that manipulates NVM via non-blocking algorithms on a system with flush-on-failure support. To understand why, consider a *recovery observer* [14], a hypothetical thread that is created at, and observes the state of NVM at, the instant when failure halts a program's real threads. The definition of non-blocking algorithm ensures that the recovery observer will see a "sane" state of memory and can make useful progress. Flush-on-failure ensures that real recovery code will have precisely the same view of NVM as the hypothetical recovery observer. Therefore real recovery code has a consistent view of application data and can resume correct execution. In summary, using non-blocking algorithms to manipulate NVM on a system with flush-on-failure is sufficient to enable consistent recovery of application data. A more detailed discussion is available in our tech report [12].

Our synthesis of non-blocking algorithms and flush-on-failure avoids the overheads of generic NVM transactions because it does not maintain transaction logs. Instead it uses flush-on-failure to ensure consistent recoverability for high-performance non-blocking algorithms. The downside is that our approach lacks the generality and convenience of NVM transactions. Designing non-blocking algorithms is a subtle art and using them to meet practical requirements is not always easy.

Experiments

Our experiments use a lock-free skip list designed by Herlihy & Shavit [8] and implemented in C by Dybnis [6]. We adapted the code to store data in a file-backed memory mapping, which enjoys flush-on-failure for process crashes. We test crash resilience by injecting sudden asynchronous process crashes (`SIGKILL`) during updates crafted to make corruption both likely and obvious; our recovery code checks the integrity of the backing file. Hundreds of injected crashes left no evidence of data corruption.

We also compared the performance of the multi-threaded non-blocking skip list with a mutex-based multi-threaded hash table crash-fortified by Atlas. We ran Atlas in both flush-as-you-go and flush-on-failure modes. On a server-class machine, the non-blocking skip list is almost twice as fast as the former and over 30% faster than the latter; indeed, the non-blocking skip list approaches the performance of the non-Atlas-ized (i.e., crash-vulnerable) hash table [12].

Conclusions & Future Work

Flush-on-failure support and non-blocking algorithms together guarantee consistent recovery of application data in NVM. Our ongoing work applies this technique to a range of applications including high-throughput streaming data processing and large-scale graph analysis.

1. REFERENCES

[1] A. Blattner et al. Generic Crash-Resilience for Indigo. Technical Report HPL-2013-75, HP Labs, 2013.

[2] G. W. Burr et al. Candidate device technologies for SCM. *IBM J. of Res. & Dev.*, 52(4.5), 2008.

[3] D. Chakrabarti et al. Atlas: Leveraging Locks for NVM consistency. In *OOPSLA*, 2014.

[4] J. Coburn et al. NV-Heaps: Making persistent objects fast & safe with NVM. In *ASPLOS*, 2011.

[5] S. R. Dulloor et al. System Software for persistent memory. In *EuroSys*, 2014.

[6] J. Dybnis. Non-Blocking Data Structures Library for x86 and x86-64, Apr. 2009.

[7] K. Fraser and T. Harris. Concurrent Programming without locks. *ACM TOCS*, 25(2), May 2007.

[8] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. Pp. 339–349.

[9] HP Labs. A New Kind of Computer, Nov. 2014.

[10] Int'l Tech. Roadmap for Semiconductors, 2013 Ed.

[11] D. Narayanan and O. Hodson. Whole-System persistence. In *ASPLOS*, 2012.

[12] F. Nawab et al. Procrastination Beats Prevention. Technical Report HPL-2014-70, HP Labs, 2014. Submitted to EDBT.

[13] S. Park et al. Failure-Atomic `msync()`. In *EuroSys*, 2013.

[14] S. Pelley et al. Memory Persistency. In *ISCA*, 2014.

[15] S. Pelley et al. Storage Management in the NVRAM era. In *VLDB*, 2014.

[16] N. Talagala. Atomic Writes Accelerate MySQL (FusionIO/Sandisk blog), Oct. 2011.

[17] H. Volos et al. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.

[18] C. Xu et al. Overcoming challenges of crossbar resistive memory. In *HPCA*, 2015. (forthcoming).