Reducing Performance Overhead of Direct Access NVM Storage Redundancy

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Department of Electrical and Computer Engineering

Rajat Kateja

B.Tech., Mathematics and Computing, Indian Institute of Technology Guwahati

Carnegie Mellon University
Pittsburgh, PA

May, 2020

Dedicated to my mother, Alpana Kateja.

Acknowledgments

It is always about the people. Reflecting on my graduate school experience, I value the people that made it all possible much more than the knowledge and skills I acquired.

In one of our first meetings, Greg asked me to think about how NVM devices are different from decades old battery-backed DRAM systems, a question that haunts me to this day. Over the years, Greg asked many such challenging questions and helped me develop a taste for novel research problems. In addition to research, he has also shaped my writing and presentation styles. Early on in my PhD, Greg's overnight (magic-like) transformations to paper intros left me wondering if I actually did the awesome work that the intros described. Over time, I picked up some writing skills, and although I am still working on improving my exposition, I am happy that my graphs now have Greg-approved thick-enough lines and big-enough fonts. Unfortunately, even with impeccable fonts, I faced one too many paper rejections, but his support helped me look beyond them. Through all the ups and downs, Greg has let me be in the driver's seat, be it in choosing paper deadlines or in choosing a career path. He helped me think through my decisions by going over the pros and cons of various options (sometimes spread across multiple meetings), but he never forced a particular decision. Although being the final decision maker was sometimes overwhelming, I am glad to have had that freedom. Thanks, Greg, for a wonderful graduate school experience.

The process of getting Andy and Nathan involved in my projects contributed significantly towards strengthening the motivation for this dissertation. Brainstorming sessions with them, along with Greg, were critical in finalizing the designs of Vilamb and Tvarak. Andy kept me on track by setting long term as well as weekly goals; this helped me make consistent progress and develop a work routine with a healthy work life balance that I still follow. Andy has also been incredibly helpful with his candid feedback on my writing, research, and job search. I thank Nathan for his patience as I learnt the ropes of computer architecture research, trade secrets of writing an architecture-y paper, and sometimes even basic computer architecture concepts. Nathan has also been generous with his time to discuss my career trajectory and helped me think through the decision of an industry vs academic job.

I thank Kim and Vijay for their time and effort as part of my thesis committee. The seeds of this dissertation were sown across multiple conversations, one of which was with Kim at FAST 2017. Thanks, Kim, for the many insightful discussions over the years. Although I haven't had the fortune of meeting Vijay in person yet, I have benefitted from his advice on Twitter, and thoroughly enjoyed the storage research from his group and interactions with his students. I look forward to meeting and learning more from him.

In addition to my committee members, I have received valuable guidance from my internship mentors and other faculty at CMU. Anirudh, Sriram, and

Bikash helped me get started with NVM research at Microsoft, and Anirudh has continued to be a mentor throughout my PhD. Working with Niket and Mustafa at Google helped me understand the many practical challenges with NVM deployments, one of which this dissertation tries to address. I thank Phil for the amazing courses and for ever so often making time to guide me in my research, Michael Kozuch for his regular check-ins and feedback on my research, Saugata for his kind presence and mentorship throughout my PhD, Vyas for his career advice, Rashmi for the technical discussions, Ram for his technical and career guidance, and Garth, Majd, and George for the rewarding conversations.

The PDL, ECE, and CyLab staff—Karen, Chad, Mitch, Jason, Charlene, Joan, Bill, Nathan Snizaski, Brittany, Brigette, and Ivan—made my life easier and enabled me to focus on my work. Karen has helped me with almost every administrative aspect in the last five years, be it conference travels, expense reimbursements, reserving conference rooms, finding the right shirt size, or the awesome food in PDL group meetings. Chad, Mitch, and Jason have debugged numerous operational issues to help set up and start my experiments, and waived max-duration time-outs to ensure that they finish as well. Charlene's passion for life is something I hope to emulate. Joan's help and patience with posters makes me wonder how I would ever make a poster after leaving CMU. Of course, thanks to all the PDL staff for their herculean efforts towards the PDL retreats and visit days. These have been defining events year-after-year for me, and I'll dearly miss the PDL retreats.

I thank VMware and the companies of the PDL consortium—Alibaba, Amazon, Datrium, Facebook, Google, Hewlett Packard Enterprise, Hitachi, IBM Research, Intel, Micron, Microsoft Research, NetApp, Oracle, Salesforce, Samsung, Seagate, and Two Sigma—for funding this research. I thank my thesis committee—Greg Ganger (chair), Andy Pavlo, Nathan Beckmann, Kimberly Keeton, and Vijay Chidambaram—for supporting me through the PhD milestones.

My family has been pivotal in helping me reach this stage of my career. Thank you for the innumerable ways in which you have made this possible, right from teaching me physics and scolding me for my math to taking care of all my needs so that I can focus on my studies and trusting my career choices. In particular, I am indebted to my mother for her unshakable belief in me. For always putting me (and the family) before herself, I dedicate this dissertation to her.

I have had the fortune of having an amazing group of friends. Sahu, Joshi, Sachin, Garg, and Gupta have continued to spark joy in my life well beyond our undergrad. Meeting Joshi in Seattle was the highlight of my first year in graduate school, and trips with Sahu, Joshi, and Sachin provided much needed break over the years. Abhilasha was a constant presence and support during the bulk of my PhD; thank you for exploring Pittsburgh with me, for often being the only person I talked to in a day, and for being my best friend (at CMU). Ankur, Samarth, and Mansi made the last phase of my graduate school much more enjoyable with all the shared meals and movie nights. Thanks Ankur for the existential memes and never-ending topics, Samarth for the music, uno, and squash, and Mansi for the banana cakes and for bringing a positive balance to our cynical group discussions. I am glad that I met and got to spend time with Sruti, Vignesh, Amit, Ankush Desai, Sandeep, Sanghamitra, Raj, Rajshekhar, Dhivya, and Prashasti. Thank you Sruti for entertaining my drop-by visits to your desk (sometimes multiple in a day), Vignesh for the last minute help with my papers and for the regular check-ins about our research progress (or the lack thereof), Amit for easing my transition to CMU with our daily lunches, and Ankush for sharing your experiences and insights into graduate school during and after our Microsoft internship.

A special shout out to Madhuri for improving my overall well-being. Thank you for entertaining my stories (and giving me many new ones), for the silly puns and jokes, for breaking my over-thought life algorithms with unexpected inputs, for putting me back in touch with my priorities, and for the most happening summer of my life so far.

I thank my awesome PDL and CyLab colleagues—Aaron, Abutalib, Akarsh, Alexey, Andrew, Angela, Ankush Jain, Anuj, Cui, Dana, Hongyi, Jack, Janos, Jason, Jinliang, Kevin, Lianghong, Lin, Mahmood, Michael Kuchnik, Nandita, Niranjini, Pardis, Pratik, Rahul, Sara, Saurabh, Sindhoora, Soo-Jin, Thomas, Tian, Vivek, and Yixin—for their friendly presence. I have thoroughly enjoyed Aaron's dry humor that made CIC 2220 more lively and the deadline night-outs more bearable, learnings from Soo-Jin's approach to life and career, Saurabh's contagious passion for storage research, Jinliang's kind and supporting words during my job search, Niranjini's research and grocery/cooking advice, and exchanging notes with Angela about myriad things as we stepped through graduate school in lockstep.

Lastly, I would like to thank a whole bunch of content creators and characters for providing me on-demand entertainment. Thank you, Biswa Kalyan Rath, Kenny Sebastian, Rahul Dua, Rahul Subramanian, Anshu Mor, Aakash Mehta, Abhishek Upamanyu, Sahil Khattar, Sucharita Tyagi, Anupama Chopra, Aaron Marino, Jeff Cavaliere, Michael Scott, Walter White, Jesse Pinkman, and many others that I am currently forgetting.

Abstract

Non-volatile memory (NVM) based storage is poised for mainstream deployment. DIMM form-factor NVM devices reside on the memory bus and offer DRAM-like access granularities and latencies along with non-volatility. NVM's Direct Access (DAX) interface enables applications to map persistent data into their address space and access it with load and store instructions, eliminating system software overheads.

Production deployment of DAX NVM storage would require that the storage system offer resilience against firmware-bug-induced data corruption, akin to conventional storage systems. Protection against firmware-bug-induced data corruptions requires the storage system to maintain system-level redundancy, which we refer to as system-redundancy. With DAX interfacing, the lack of interposed system software makes it challenging to identify data reads and writes that should trigger system-redundancy verification and updates, respectively. Further, the DAX granularities (e.g., 64-byte cache-lines) are incongruent with typical system-redundancy granularities (e.g., 4K pages), leading to high performance overhead in maintaining system-redundancy.

This dissertation demonstrates that DAX NVM storage systems can efficiently maintain system-redundancy by relaxing the data coverage guarantees or by leveraging a hardware offload. We support the thesis with two case studies: Vilamb and Tvarak. The Vilamb library maintains system-redundancy asynchronously, avoiding critical path interpositioning and amortizes the overhead of system-redundancy updates across multiple writes to a page. As a result, Vilamb provides 3–5× the throughput of the state-of-the-art software solution at high operation rates. For applications that need system-redundancy with high performance, and can tolerate some delaying of data redundancy, Vilamb provides a tunable knob between performance and time-to-coverage. Even with the delayed coverage, Vilamb increases the mean time to data loss due to firmware-induced corruptions by up to two orders of magnitude in comparison to maintaining no system-redundancy.

Tvarak is a software-managed hardware offload to efficiently maintain system-redundancy for direct-access (DAX) NVM storage. Tvarak reconciles the mismatch between DAX granularities and typical system-redundancy granularities by introducing cache-line granular checksums (only) for DAX-mapped data. Tvarak also uses caching to reduce the number of extra NVM accesses for maintaining and verifying system-redundancy. Applications' data access locality leads to reuse of system-redundancy that Tvarak leverages with a small dedicated on-controller cache and configurable LLC partitions. Simulation-based evaluation demonstrates Tvarak's efficiency. For example, Tvarak reduces Redis set-only performance by only 3%.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Non-volatile memory (NVM) changes the way performance-sensitive applications interact with persistent data. NVM storage combines DRAM-like access latencies and granularities with disk-like durability [1, 15, 19, 56, 79]. Direct access (DAX) to NVM data exposes raw NVM performance to applications. Applications using DAX map NVM files into their address spaces and access data with load and store instructions, eliminating system software overheads associated with conventional storage interfaces.

Production storage demands more than just non-volatility and performance. A number of features that empower and simplify storage management efforts are also expected. Whereas some features extend to DAX NVM storage trivially, e.g., background scrubbing and defragmentation, others with data access inter-dependencies do not. Notably, redundancy mechanisms for fault tolerance fit poorly.

Production storage systems need to protect data from various failures. In addition to fail-stop failures like machine or device crashes, storage systems also need to protect data from silent corruption due to firmware bugs. Storage device firmware is prone to bugs because of its complexity, and these bugs can cause data corruption. Such corruption-inducing firmware bugs fall into two broad categories: lost write bugs and misdirected read or write bugs [11, 12, 43, 74, 90]. Lost write bugs cause the firmware to acknowledge a write without ever updating the data on the device media. Misdirected read or write bugs cause the firmware to read or write the data from the wrong location on the device media. Firmware-bug-induced corruptions go unnoticed even in the presence of device-level ECC, because the device-level ECC is read/written as an atom with its data during each media access performed by the firmware.

Protection against firmware-bug-induced corruption commonly relies on system-level redundancy, i.e., system-checksums for detection and parity for recovery. *System-checksums* are data checksums that the storage system computes and verifies at a layer “above” the device firmware (e.g., the file system), using separate I/O requests than for the corresponding data [11, 74, 105]. Using separate I/O requests for the data and the block containing its system-checksum (together with system-checksums for other data) reduces the likelihood of an undetected firmware-bug-induced corruption. This is because a bug is unlikely to affect both in a consistent manner. Thus, the storage system can detect a firmware-bug-induced corruption because of a mismatch between the data and its system-checksum. It can then trigger recovery using the parity [51, 63, 68, 114]. In this paper, we use the term *system-redundancy* to refer to the combination of system-checksums and parity.

Production NVM-based storage systems will need system-redundancy mechanisms for the same reasons as conventional storage. NVM device firmware involves increasingly complex functionality, akin to that of other storage devices, making it susceptible to both lost write and misdirected read/write bugs. However, maintaining system-redundancy for DAX NVM storage, without forfeiting its performance benefits, is challenging for two reasons. First, accesses via load and store instructions bypass system software, removing the straightforward ability to detect and act on data changes (e.g., to update system-redundancy). Second, NVM's cache-line granular writes increase the overhead of updating system-redundancy (e.g., system-checksums) that is usually computed over sizeable data regions (e.g., pages) for effectiveness and space efficiency.

The state-of-the-art solutions for DAX NVM system-redundancy are transactional libraries that maintain system-checksums and parity [89, 109, 112]. To address the challenge of system software bypass, these libraries require applications to inform the library about all their data accesses. Although this enables the libraries to mediate and act on all application data accesses, it imposes a restrictive programming model. Furthermore, these solutions incur a high overhead even with optimized designs and implementations [109]. This is because library-based solutions struggle to synchronously update system-redundancy at high application write throughput rates.

1.1 Thesis statement

This dissertation describes our work on efficient implementation of system-redundancy mechanisms to protect DAX NVM storage from firmware-bug-induced data corruptions. In particular, we make the following thesis statement:

Direct access NVM storage can be fortified with conventional redundancy mechanisms for protection against firmware-bug-induced data corruptions at low performance overheads by relaxing the data coverage guarantees or by leveraging a hardware offload for system-redundancy computation and verification.

To support this thesis, we detail two case studies for improving the performance of DAX NVM storage redundancy:

- **Vilamb: Low overhead asynchronous redundancy for direct-access NVM storage (Chapter 3).** Vilamb [50] provides efficient asynchronous system-redundancy for DAX NVM storage. The Vilamb user-space library maintains system-redundancy with low overhead by delaying and amortizing the system-redundancy update overhead across multiple data writes. To do so, Vilamb repurposes page table dirty bits to identify pages with outdated system-redundancy. As a result, Vilamb provides 3–5× the throughput of the state-of-the-art software solution, Pangolin [109], at high operation rates. For applications that need system-redundancy with high performance, and can tolerate some delaying of data redundancy, Vilamb provides a tunable knob between performance and quicker system-redundancy. Even with the delayed coverage, Vilamb increases the mean time to data loss due to firmware-induced corruptions by up to two orders of magnitude in comparison to maintaining no system-redundancy.

- **Tvarak: Software-managed hardware offload for redundancy in direct-access NVM storage (Chapter 4).** For DAX NVM applications with strong coverage requirements, Tvarak [49] efficiently implements synchronous system-redundancy with synchronous system-checksums and parity updates and system-checksum-verified reads. Software-only implementations of system-redundancy force a choice between reduced or delayed data protection and significant performance penalties. Offloading the update and verification of system-redundancy to Tvarak, a hardware controller co-located with the last-level cache, enables efficient protection of data from bugs in the memory controller and in NVM DIMM firmware. Simulation-based evaluation with seven data-intensive applications shows that Tvarak is efficient. For example, Tvarak reduces Redis set-only performance by only 3%.

1.2 Contributions

This dissertation makes the following key contributions.

Vilamb:

- It identifies asynchronous system-redundancy as an important addition to the toolbox of DAX NVM system-redundancy solutions.
- It describes Vilamb's efficient asynchronous system-redundancy design that improves performance for applications that can tolerate delayed coverage.
- It quantifies Vilamb's efficacy, cost, and reliability via extensive evaluation with eight macro- and micro-benchmarks.

Tvarak:

- It motivates the need for architectural support for DAX NVM storage system-redundancy, highlighting the limitations of software-only approaches.
- It proposes Tvarak, a low-overhead, software-managed hardware offload for DAX NVM storage redundancy. It describes the challenges for efficient hardware DAX NVM system-redundancy and how Tvarak overcomes these challenges with a straightforward, effective design.
- It reports on extensive evaluation of Tvarak's runtime, energy, and memory access overheads for seven applications, each under multiple workloads, showing its efficiency.

1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes the background and motivation for this dissertation. It details the two broad classes of firmware bugs, presents system-redundancy as the protection mechanism, and describes the trade-offs across various DAX NVM system-redundancy solutions. Chapter 3 describes the design, implementation, and evaluation of Vilamb. Chapter 4 describes the design of Tvarak and its evaluation using the ZSim simulator [86]. Chapter 5 concludes this dissertation and presents some future research directions.

Chapter 2

Background

This section describes NVM storage and direct access interfacing, management features expected of production storage and implications of direct access on these features, redundancy mechanisms for protection against firmware-bug-induced corruptions, and the related work on supporting those features for DAX NVM storage.

2.1 NVM Storage

Non-volatile memory (NVM) refers to a class of memory technologies that have DRAM-like access latency and granularity but are also durable like disks. Various NVM technologies are already available or expected to be available soon [1, 19, 38, 56, 79]. NVM devices have orders of magnitude lower latency and higher bandwidth than conventional storage devices. This improved device performance benefits applications even with an unmodified storage stack, but more so with NVM-optimized storage stacks [8, 9, 22, 27, 44, 52, 103, 104].

The easiest way to incorporate NVM into systems is to export it to applications via existing file system interfaces. Doing so allows legacy applications to use NVM without modification. The downside is the performance overhead of system calls, data copying, and inefficient general-purpose system software code.

The DAX interface to NVM eliminates system software overheads, enabling applications to leverage raw NVM performance [27, 44, 49, 50, 57, 59, 62, 73, 81, 89, 96, 105, 109, 112]. With DAX, applications map NVM pages into their address spaces and access persistent data via load and store instructions. File systems that map a NVM file into the application address space (bypassing the page cache) on a `mmap` system call are referred to as DAX file systems and said to support DAX-mmap [27, 59, 105]. DAX is widely used for adding persistence to conventionally volatile in-memory DBMSs [57, 62, 73, 81, 112] and is poised as a “killer use-case” for NVM.

DAX-mmap helps applications realize NVM performance benefits, but requires careful reasoning to ensure data consistency. Volatile processor caches can write-back data in arbitrary order, forcing applications to use cache-line flushes and memory fences for durability and ordering. Transactional NVM access libraries ease this burden by exposing simple transactional APIs to applications and ensuring consistency on their behalf [14, 21, 36, 39, 97]. Alternatively, the system can be equipped with enough battery to allow flushing of cached writes to NVM before a power failure [66, 70, 113]; our work assumes this option.

2.2 DAX NVM Storage Management

Administrators rely on and expect a variety of storage management features to avoid data loss [31, 74, 87, 111], data theft [13, 46, 84, 102], reduce cost [23, 55, 75, 92, 115], and handle failures [31, 41, 51, 68, 69]. As NVM storage matures for use in production environments, it will be expected to provide these features as well.

We categorize storage management features into four groups based on their inter-relationships with foreground data accesses (application reads and/or writes):

- Background scan/reorganization features, like scrubbing and defragmentation, occur independently of reads and writes.
- Space efficiency features, like compression, deduplication, and tiering, track data accesses to estimate data temperature; they can potentially impact read operations (e.g., when reading compressed data), but are unlikely to impact write operations because they operate on cold data.
- Data redundancy features like block checksums and cross-component redundancy are inter-dependent on writes; they may impact reads depending on how they are used (e.g., if reads require a checksum verification).
- Security features like encryption are directly involved in servicing of reads and writes.

Software bypass for direct access NVM storage has different implications for different categories of management features. For example, software bypass has no effect on purely background operations like scrubbing and defragmentation, because they do not depend on or impact any data accesses.

Space efficiency features track data access recency, but the loss of this information does not affect their correctness—only performance (e.g., if a hot page is compressed). Consequently, supporting them for direct access NVM is not overly complex. NVM storage systems can use page table accessed bits to track data accesses and not be concerned with occasionally losing this information (e.g., due to a power failure).

Data redundancy and security features have a strong dependency on write accesses—not knowing about data updates impacts their correctness. Data security features are also tightly coupled with read accesses (e.g., decryption) whereas data redundancy features may or may not be coupled with data reads (e.g., whether reads are verified). Given the tight coupling between security features and data accesses (both reads and writes), data security features demand hardware support. Introducing software encryption/decryption in the data path would annihilate the performance benefits of DAX NVM storage. Recent works have proposed memory-controller/-module support for NVM encryption [16, 60, 61, 106].

Data redundancy features will also benefit from hardware support, but can also be efficiently implemented in software if redundancy updates and verification can be moved out of the critical path. For example, if redundancy is verified only via background scrubbing, it would not longer impact application reads. However, state-of-the-art software-based approaches for redundancy rely on interposing on writes [89, 109, 112] in the critical path and introduce significant performance and programming limitations.

2.3 Redundancy for Firmware Bug Resilience

Production storage systems employ multiple redundancy mechanisms to address a variety of faults [31, 41, 51, 54, 65, 68, 69, 105, 110, 111]. In this work, we focus on redundancy mechanisms used to detect and recover from firmware-bug-induced data corruption (specifically, system-checksums and parity).

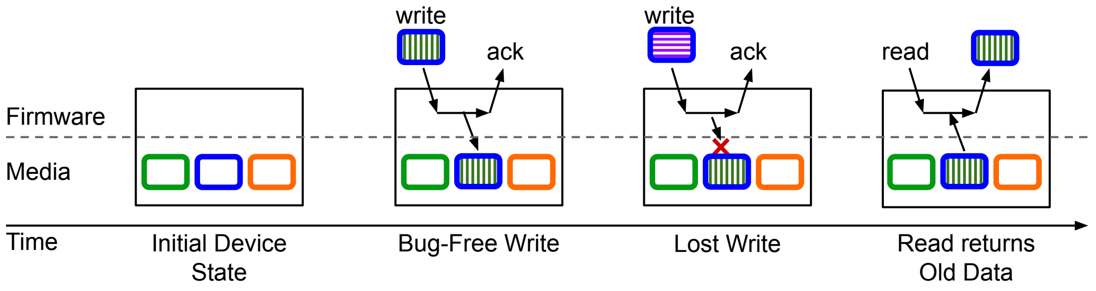
2.3.1 Firmware-bug-induced data corruption

Large-scale studies of deployed storage systems show that device firmware bugs sometimes lead to data loss or corruption [11, 12, 43, 74, 90]. Device firmware, like any software, is prone to bugs because of its complex responsibilities (e.g., address translation, dynamic re-mapping, wear leveling, block caching, request scheduling) that have increased both in number and complexity over time. Research has even proposed embedding the entire file system functionality [47] and application-specific functionalities [2, 18, 82, 83, 100] in device firmware. Increasing firmware complexity increases the propensity for bugs, some of which can trigger data loss or corruption.

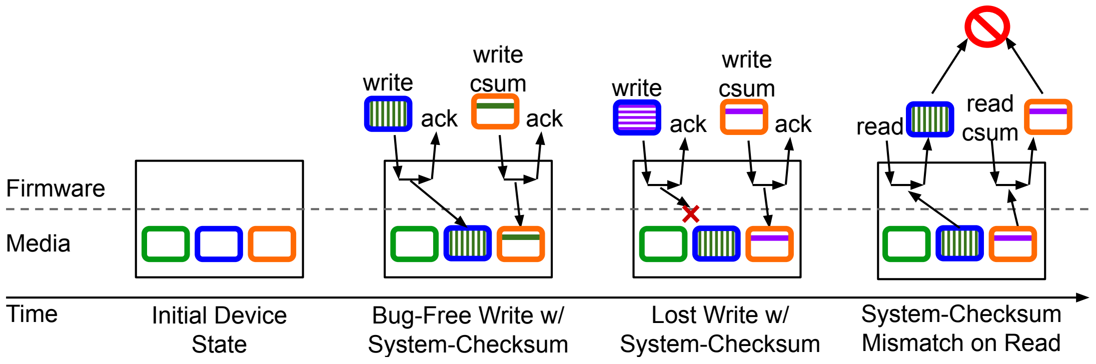
Corruption-inducing firmware-bugs can be categorized into two broad categories: lost write bugs and misdirected read/write bugs. A lost write bug causes the firmware to acknowledge a write without ever updating the media with the write request's content. An example scenario that can lead to a lost write is if a write-back firmware cache "forgets" that a cached block is dirty. Fig. 2.1(a) illustrates a lost write bug. It first shows (second stage in the time-line) a correct bug-free write to the block stored in the blue media location. It then shows a second write to the same block, but this one suffers from a lost write bug—the firmware acknowledges the write but never updates the blue media location. The subsequent read of the blue block returns the old data to the application.

A misdirected write or misdirected read bug causes the firmware to store data at or read data from an incorrect media location, respectively. Fig. 2.2(a) illustrates a misdirected write bug. As before, the first write to the block stored in the blue location is performed correctly by the firmware. For this example, the second write request shown is for the block stored in the green location. The second write request encounters a misdirected write bug wherein the data is incorrectly written to the blue media location. Notice that a misdirected write bug not only fails to update the intended block, but also corrupts (incorrectly replaces) the data of the block it incorrectly updates. In the example, the subsequent read to the block mapped to the blue location returns this corrupted data.

Although almost all storage devices maintain error-correcting codes (ECCs) to detect corruption due to random bit flips [24, 42, 94, 108], these ECCs cannot detect firmware-bug-induced corruption [11, 74]. Device-level ECCs are stored together with the data and computed and verified inline by the same firmware during the corresponding data access. So, in the case of a lost write, the firmware loses the ECC update along with the corresponding



(a) The problem: device responds to read of block that experienced the lost write with incorrect (old) data.


write
ack
write csum
ack
write
ack
write csum
ack
read
read csum
Firmware
Media
Time
Initial Device State
Bug-Free Write w/ System-Checksum
Lost Write w/ System-Checksum
System-Checksum Mismatch on Read

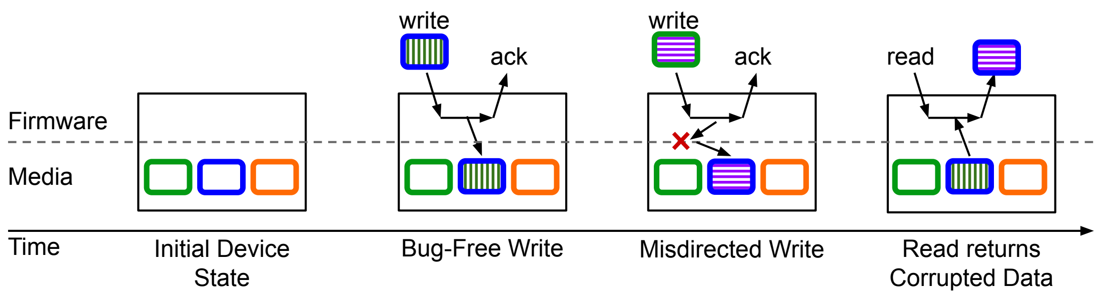
(b) The fix: having the higher-level system update and verify system-checksums when writing or reading data, in separate requests to the device, enables detection of a lost write because of mismatch between the data and the system-checksum.

Figure 2.1: Lost write bug example: Both sub-figures show a time-line for a storage device with three media locations. The device is shown in an initial state, and then upon completion of higher-level system's write or read to data (first, a successful write, then a "lost write", then a read) mapped to the same media location. Fig. 2.1(a) shows how the higher-level system can consume incorrect (old) data if it trusts the device to never lose an acknowledged write. Fig. 2.1(b) shows how the higher-level system can detect a lost write with system-checksums.

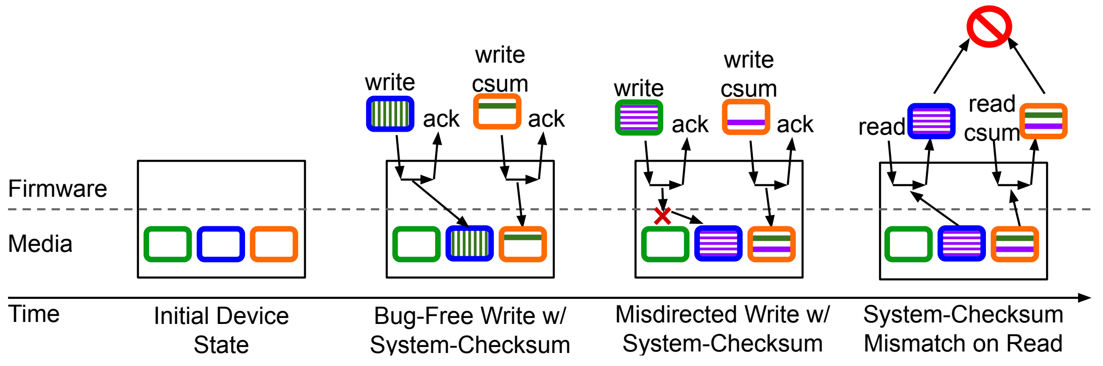
data update, because the data and ECC are written together on the media as one operation. Similarly, misdirected writes modify the ECC to match the incorrectly updated data and misdirected reads retrieve the ECC corresponding to the incorrectly read data.

2.3.2 System-checksums for detection

Production storage systems maintain per-page *system-checksums* to detect firmware-bug-induced data corruption. System-checksums are updated and verified at a layer above the firmware, such as the file system, stored in checksum pages (each containing checksums for multiple pages) separate from the data, and read and written using I/O requests separate from the corresponding data I/O requests [31, 54, 65, 105, 110, 111]. Separating the storage and accesses for data from corresponding system-checksums enables detection of firmware-bug-induced corruption, because such bugs are unlikely to affect both the data



(a) The problem: device responds to read with the incorrectly updated data from the blue location. Notice that the green location also has incorrect (old) after the misdirected write.


write
ack
write csum
ack
write
ack
write csum
ack
read
read csum
Firmware
Media
Time
Initial Device State
Bug-Free Write w/ System-Checksum
Misdirected Write w/ System-Checksum
System-Checksum Mismatch on Read

(b) The fix: having the higher-level system update and verify system-checksums when writing or reading data, in separate requests to the device, enables enables detection of a misdirected write because of mismatch between the data and the system-checksum.

Figure 2.2: Misdirected write bug example: Similar construction to Fig. 2.1, but with the second operation being a write intended for the green location that is misdirected by the firmware to the blue location.

and its system-checksum access. Even in an unlikely scenario that a bug that affect both the accesses, it is even more unlikely that the bug affects both in a consistent fashion (e.g., losing both or misdirecting both to another corresponding data and system-checksum pair) is even lower.

Fig. 2.1(b) demonstrates how system-checksums enable detection of lost writes. Although the second write to the blue block is lost, the write to the checksum block (stored in the orange location) is not. Thus, upon the data read in the example, which is paired with a corresponding system-checksum read and verification, the lost write is detected.

Fig. 2.2(b) illustrates how system-checksums enable detection of misdirected writes. A misdirected write firmware bug is extremely unlikely to affect both the data write to the green block and the corresponding system-checksum write to the orange block in a consistent manner. To do so, the firmware would have to incorrectly write the system-checksum to a location (block *and* the offset within the block) that stores the checksum for the exact block to which it misdirected the data write. In the illustration, the read of the blue block data, followed by its system-checksum read, results in a verification failure. Similarly, system-checksums also trigger a verification failure in case of a misdirected read bug, because a bug is unlikely to affect the both the data its system-checksum read.

2.3.3 Parity for recovery

To recover from a detected page corruption, storage systems store parity pages [41, 51, 63, 68, 69, 114]. Although parity across arbitrarily selected pages suffices for recovery from firmware-bug-induced corruption, storage systems often implement cross-device parity that enables recovery from device failures as well.

2.3.4 System-Redundancy for DAX NVM

NVM storage systems will be prone to firmware-bug-induced data corruption and require corresponding redundancy mechanisms, akin to conventional storage systems. NVM firmware is susceptible to corruption-inducing bugs, because it is non-trivial and its complexity can only be expected to increase over time. NVM firmware already provides for address translation, bad block management, wear leveling, request scheduling, and other conventional firmware responsibilities [77, 78, 79, 88]. Looking forward, its complexity will only increase as more NVM-specific functionality is embedded into the firmware (e.g., reducing NVM writes and wear [17, 29, 61, 106, 116]) and as the push towards near-data computation [2, 4, 5, 18, 28, 35, 47, 82, 83, 100, 107] continues. Prior experience with firmwares of such complexity demonstrate that they inevitably suffer from bugs that lead to lost or misdirected reads and writes. Consequently, production NVM storage systems will need firmware-bug resiliency mechanisms.

Maintaining system-redundancy for DAX NVM is challenging for two reasons: (i) hardware controlled data movement, and (ii) cache-line granular writes.

Hardware Controlled Data Movement: Applications' data writes to DAX NVM bypass system software. This lack of software control makes it challenging for the storage software to identify updated NVM pages for which it needs to update system-redundancy.

Cache-line Granular Writes: Incongruence in the size of DAX writes and the size of pages over which system-redundancy is usually maintained increases the overhead of maintaining system-redundancy. Most storage systems maintain system-redundancy over sizeable blocks (e.g., 4K page checksums) for space efficiency. Cache-line granular writes require reading (at least) an entire page to update the system-redundancy. Whereas RAID systems solve a similar "small write" problem by reading the data before updating it [68], a DAX NVM storage system software cannot use this solution. As discussed above, direct access to NVM bypasses system software, prohibiting the use of pre-write values for incremental system-redundancy updates.

2.4 Solutions for DAX NVM System-Redundancy

Existing proposals for maintaining system-checksums and parity in NVM storage systems compromise on performance, coverage, and/or programming flexibility for DAX-mapped data. Table 2.1 presents the design choices of prior proposals as well as Vilamb and Tvarak.

NVM Storage Redundancy Design	**Checksum Granularity**	**Redundancy Update for DAX data**	**Redundancy Verification for DAX data**	**Performance Overhead**	**Programming Model**	**Specialized Hardware Requirement**
Nova-Fortis [105], Plexistore [71]	Page	No updates	No verification	None	FS Interface	None
Mojim [112], HotPot [89]	Page[1]	On application data flush	Scrubbing	High	Transactional Library Interface	None
Pangolin [109]	Object	On application data flush	On NVM to DRAM copy	Moderate/High	Transactional Library Interface	None
Vilamb	Page	Periodically	Scrubbing	Configurable	No Restriction	None
Tvarak	Page	On LLC to NVM write	On NVM to LLC read	Low	No Restriction	Specialized Controller

Table 2.1: Solutions for DAX NVM storage system-redundancy and their trade-offs.

Two recent fault-tolerant NVM file systems, Nova-Fortis [105] and Plexistore [71], update and check system-redundancy during explicit file system calls but do not update or verify redundancy while data is DAX mapped.

Pangolin [109] is a user-space library that maintains DAX NVM system-redundancy synchronously by requiring applications to explicitly inform it about their data updates; applications piggyback these notifications on Pangolin's transactional interface. Pangolin offers strong coverage (immediate system-redundancy updates and verification) and does not require any specialized hardware resources (because it is a software-based solution). Pangolin addresses the mismatch of fine-grained DAX updates with large checksum ranges by requiring explicit object definitions and maintaining per-object checksums instead of per-page checksums. Pangolin also introduces micro-buffering, i.e., buffering objects that an application writes to in DRAM and updating the NVM only on transaction commits. This buffering also enables Pangolin to use data diffs to make system-redundancy updates more efficient.

Pangolin is well-tuned, including several overhead-reducing mechanisms, making it the state-of-the-art for an in-line software-only solution. Yet, Pangolin incurs significant performance overhead (up to 80%) in many cases. Fundamentally, Pangolin's synchronous system-redundancy update design requires updating system-redundancy at the same rate at which an object is being modified; this becomes costly for the high update rates enabled by NVM. Furthermore, Pangolin only works for applications that can be and are modified to use its object-based transactional interface. Applications that manage NVM data themselves using other data models, such as NVM-optimized databases [8], may not easily fit to Pangolin's interface. Pangolin's per-object checksums also incur higher space overhead for small data objects.

Mojim [112] and HotPot [89] are also library based solutions similar to Pangolin. They were developed for remote replication in DAX NVM but their design can be extended to include system-redundancy. Their interposing based design imposes the same programming restrictions as Pangolin because applications need to inform them about all application writes. Furthermore, their page granular system-checksums incur a even higher overhead

[1] The original Mojim and HotPot designs do not include checksums, only replication. Here we extrapolate their design to include checksums.

because of the fine-grained applications writes (Section 2.3.4).

Vilamb [50] is a software library that embraces an asynchronous approach to updating system-redundancy for updated data. Like other asynchronous redundancy-update approaches, it identifies and completes required system-redundancy updates in the background. Vilamb does not impose any programming model restrictions and does not require any specialized hardware resources. However, Vilamb reduces the data coverage guarantees by delaying system-redundancy updates. Specifically, recently modified pages may not be covered when a firmware bug affects them. Vilamb is a good option for applications that desire high performance and/or are not a good fit for Pangolin-like API and view partial system-redundancy coverage is as better than none.

Tvarak [49] is a hardware controller co-located with the last-level cache (LLC) that the file system can offload system-redundancy maintenance work onto. Tvarak is able to identify data updates by the virtue of being interposed in the data path. Tvarak offers synchronous system-redundancy updates and verification, does not restrict applications to any specific library/API, and is low-overhead.

Chapter 3

Vilamb: Low overhead asynchronous redundancy for direct-access NVM storage

This chapter describes **Vilamb**[1] [50], a user-space library for efficient asynchronous DAX NVM system-redundancy. Vilamb moves system-redundancy updates out of the critical path and delays them to amortize the overhead over multiple data updates. Delaying the system-redundancy updates creates a configurable trade-off between performance and the delay before updated data is covered.

Vilamb repurposes page table dirty bits to efficiently identify of data updates. Vilamb marks pages with updated system-redundancy as clean and identifies pages with invalidate system-redundancy by checking their dirty bit. We implement a kernel module that Vilamb uses for batched fetching and clearing for dirty bits. Vilamb ensures atomic and consistent system-redundancy updates for all dirty pages by using shadow copies of dirty bits and leveraging batteries that are common in production environments [26, 32, 45, 48, 53, 67, 98, 99].

For applications that can tolerate delayed data coverage, Vilamb offers an efficient alternative to synchronous software based redundancy without requiring specialized hardware. Extensive evaluation with eight macro- and micro-benchmarks demonstrate Vilamb's efficacy. Vilamb with a 1 second delay between system-redundancy updates reduces single-threaded Redis' YCSB throughput by only 1.6–17%, compared to 13–18% for Pangolin. Increasing the delay to 10 seconds further reduces Vilamb's overhead to 0.1–6%. Vilamb offers 3–5× higher throughput than Pangolin at high insert rates for all five of Intel's persistent memory development kit (PMDK) key-value stores. By protecting data that belongs to clean stripes from firmware-bug-induced corruptions, Vilamb increases the mean time to data loss (MTTDL) over maintaining no system-redundancy. For example, Vilamb with a 1 second system-redundancy update period increases Redis' MTTDL by 15× and 74× over No-Redundancy for a write-heavy and ready-heavy YCSB workload, respectively. Detailed timing breakdowns with fio microbenchmarks and battery cost analysis confirm

[1]Vilamb means delay in Hindi.

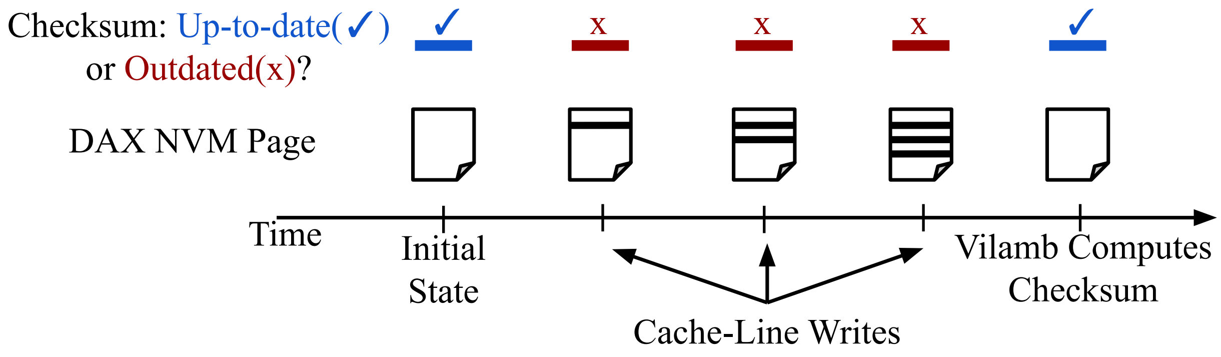


Figure 3.1: Delayed Checksum Computation Example: By computing per-page system-checksums asynchronously, Vilamb amortizes the computation overhead over multiple cache-line writes to the same NVM page.

Vilamb's design decisions.

3.1 Vilamb Design and Implementation

This section begins by describing Vilamb's design elements: delayed system-redundancy updates and repurposing of dirty bits. It then describes the effect of Vilamb's design on resilience against different failures and ends with Vilamb's implementation details.

3.1.1 Asynchronous System-Redundancy

Vilamb asynchronously maintains per-page system-checksums and cross-page parity for DAX NVM storage. A background thread periodically updates system-redundancy for pages which have been written to since Vilamb last updated their system-redundancy. By delaying system-redundancy updates, Vilamb amortizes the overhead over multiple cache-line writes to the same DAX NVM page.

Fig. 3.1 illustrates how Vilamb reduces work for per-page system-checksums (cross-page parity is not shown in the example, but is updated at the same time as the system-checksum). The figure shows a DAX NVM page and its system-checksum; the system-checksum can either be up-to-date (✓) or outdated (x). In the initial state, the system-checksum is up-to-date with the data. The first write to the page makes the system-checksum stale. Instead of updating the checksum immediately, Vilamb delays the update until after two more writes. By delaying the update Vilamb performs a single checksum (and parity, not shown in the example) computation, instead of three.

Vilamb scrubs the data using a separate background thread to detect data corruption. Upon mismatch between the page data and system-checksum for a clean page, Vilamb raises an error and halts the program. The OS can recover corrupted pages using the parity pages, with potential re-mapping to different physical pages [105, 109].

3.1.2 Repurposing Dirty Bits

The conventional use-case of dirty bits is irrelevant for DAX NVM pages, making them available for repurposing. The dirty bit is conventionally used to identify updated, or "dirtied", in-memory pages that the storage system needs to write back to persistent storage. In case of DAX NVM storage, the file system maps NVM-resident files into application address spaces using the virtual memory system [27, 59]. Consequently, even though each mapped page has a corresponding dirty bit, the conventional semantic of these dirty bits is irrelevant because the pages already reside in persistent NVM storage.

Vilamb repurposes dirty bits to identify pages that have been written to since Vilamb last updated their system-redundancy. When a file is first DAX mapped, its pages' dirty bits are clear and system-redundancy is up-to-date (potentially updated during initialization for newly created files). A page write, which causes its system-redundancy to become stale, sets the page's dirty bit. In each successive invocation, Vilamb's background thread updates the system-redundancy for pages with their dirty bit set and then clears the corresponding dirty bits again.

Shadow Dirty Bits: Vilamb carefully orchestrates the non-atomic two-step process of updating a page's system-redundancy and clearing its dirty bit; performing these steps without any safeguard is incorrect. Clearing the dirty bit after updating the system-redundancy is incorrect because an interleaved application access can invalidate the system-redundancy. Reversing the order is not safe either. A system-checksum verification (e.g., in a scrubbing thread) after the dirty bit is cleared, but before the system-checksum is updated, would cause a spurious mismatch between the data and its system-checksum. Vilamb makes a persistent shadow copy of the dirty bit before clearing it, and clears this shadow copy only after completing the redundancy update. If either of the dirty bit or its shadow copy is set for a page, Vilamb knows that the page's redundancy is outdated.

Impact of Huge Pages: Huge pages are detrimental to Vilamb's performance. For each dirty page, Vilamb needs to read the entire page to compute its system-checksum and read other pages in its stripe to compute their cross-page parity. With huge pages, the amount of data that Vilamb has to read increases, increasing Vilamb's overhead for maintaining system-redundancy. If and when huge pages become more prevalent, Vilamb would need alternative mechanisms to track updated data. Potential options include architectural modifications to maintain multiple finer-granularity dirty bits per-huge-page, or tracking data writes using a Pangolin-like scheme that requires application modifications [109]. We leave this exploration for future work.

3.1.3 Failure Coverage

Vilamb's asynchronous approach to system-redundancy introduces a tunable window of vulnerability. Pages that an application writes to remain susceptible to corruption until Vilamb updates their system-redundancy. We describe the implication of this window of vulnerability for different kinds of failures below.

Page Corruption: System-redundancy's primary goal is to protect data from firmware-bug-induced corruption. Additionally, system-redundancy also protects from random bit flip induced corruptions, though on-device ECC is already expected to address those. Vilamb's delayed system-checksums would detect corruption to all but recently written (dirty) pages. We illustrate this with an example lost write bug triggered in three different scenarios.

Consider a firmware that uses an on-device write-back cache and that suffers from a bug wherein the firmware (infrequently) "forgets" to destage some data from the cache to the device media.

- For the first scenario, consider an application write that is evicted from the CPU caches to the NVM device, is stored in the on-device write-back cache, and then lost by the firmware before Vilamb updates the corresponding page's system-checksum. This would lead to a silent corruption because Vilamb would use the incorrect (old) data to compute the system-checksum.
- For the second scenario, consider that Vilamb updates the page's system-checksum before the firmware bug is triggered (i.e., while the data is in the CPU caches or in the on-device cache). Vilamb would update the system-checksum correctly in this scenario and detect the subsequent corruption because of a data-system-checksum mismatch at a later point.
- For the third scenario, imagine the bug affects a clean page while the firmware is performing wear leveling. Vilamb would be able to detect this data loss in its scrubbing thread.

Among the pages that Vilamb detects as corrupted, Vilamb can recover those that belong to stripes with all clean pages (and hence, an up-to-date parity). Any dirty page in a stripe invalidates the parity. Thus, even if the corrupted page is itself clean, Vilamb can recover it only if all other pages in its stripe are also clean.

Power Failures: Vilamb avoids any inconsistencies between data and its system-redundancy by ensuring that the system-redundancy is made up-to-date if there is a power failure. To that end, Vilamb leverages battery backups that are common in production environments [26, 32, 33, 45, 48, 53, 98]. Conventional storage systems use batteries to flush DRAM to a persistent medium upon a power failure [26, 33, 45, 48]. NVM does not need batteries to make its contents persistent because they are already persistent. Vilamb instead leverages the battery backup to update system-redundancy upon a power failure, ensuring that no pages are left uncovered. Given that batteries are and will continue to be used to address other issues as well, e.g., brief power losses and spikes [67], Vilamb can exploit them for updating system-redundancy.

NVM DIMM Failures or Machine Failures: Vilamb's system-redundancy is not intended for protection against DIMM or machine failures; the storage system can protect against these using remote replication [89, 112]. Being a machine-local fault-tolerance mechanism, system-redundancy, independent of its implementation, is ineffective against machine failures. For DIMM failures, Vilamb's asynchronous system-redundancy design makes it unable to reconstruct the fraction of the pages in the failed DIMM that belonged to a stripe with outdated system-redundancy. Although the storage system could still recover a large fraction of the data (Section 3.2.8), it would need other redundancy to recover the remaining data.

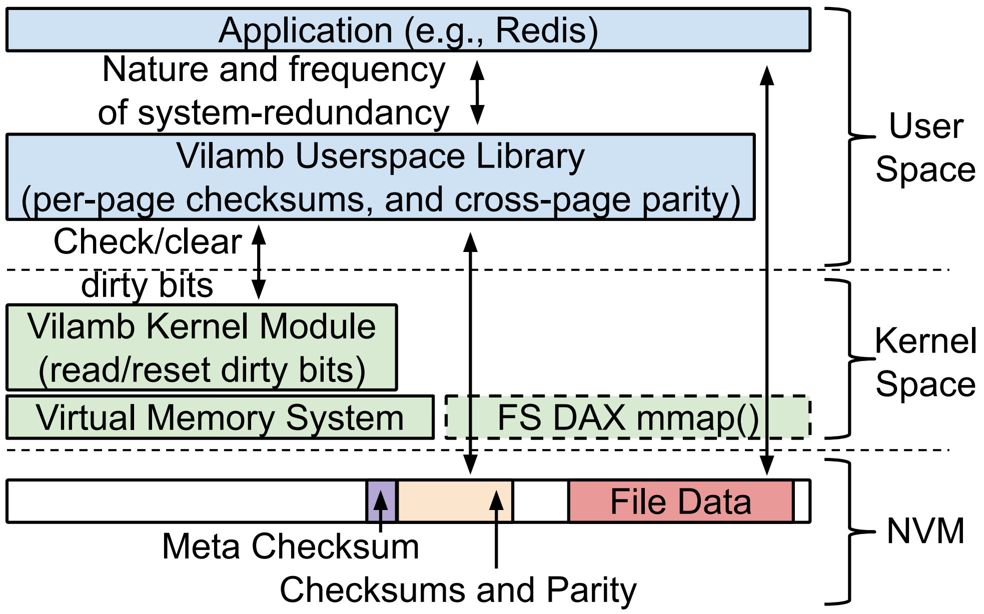

Application (e.g., Redis)
Nature and frequency of system-redundancy
Vilamb Userspace Library (per-page checksums, and cross-page parity)
Check/clear dirty bits
Vilamb Kernel Module (read/reset dirty bits)
Virtual Memory System
FS DAX mmap()
File Data
Meta Checksum
Checksums and Parity
User Space
Kernel Space
NVM

Figure 3.2: Vilamb's Implementation: The user space library performs the system-checksum and parity computations with a period that is set by the application. The kernel module checks and clears the dirty bits when requested by the user space library.

3.1.4 Implementation

We implement Vilamb as a user-space library. The library exposes an API that applications can use to configure the nature of system-redundancy (e.g., type of checksum and number of pages in a stripe) and its update frequency. The library uses a periodic background thread that checks and clears the dirty bits using new system calls that we implement, and performs the system-redundancy updates for the dirty pages. Our implementation uses a stripe size of five pages by default, with four consecutive data pages and one parity page. The stripes are statically determined at the time of initialization. Fig. 3.2 shows the components of our implementation.

New System Calls: We implement two new system calls, `getDirtyBits` and `clearDirtyBits`, to check and clear the dirty bits for pages in a memory range, respectively. `getDirtyBits` returns a bitvector that has the dirty bits for pages in the input memory range. `clearDirtyBits` accepts a dirty bitvector as its parameter in addition to a memory range. It clears the dirty bit for a page in the memory range only if the corresponding bit is set in the input dirty bitvector. Since Vilamb is unaware of pages dirtied in between the checking and clearing and will not update their system-redundancy, it uses this input dirty bitvector for `clearDirtyBits` to clear the dirty bits only for pages that were dirty when initially checked.

Batched Checking and Clearing: Vilamb checks and clears dirty bits for multiple NVM pages (e.g., 512 in our experiments) as a batch for efficiency. Both checking and clearing of dirty bits require a system call and traversing the hierarchical page table; clearing

Algorithm 1: System-Redundancy Update Thread

```
   Parameter : Batch Size, B pages
   Parameter : Number of Pages in File, N
   Parameter : Number of Pages in a Parity Group, P
1  for $i \leftarrow 0$ to $N$ increment by $B$ do
2      dirtyBitvector ← checkDirtyBits($i$, $i + B$);
3      dirtyBitvectorCopy ← dirtyBitvector;
4      currentBatchStartingPage ← $i$;
5      memoryFence;
6      clearDirtyBits($i$, $i + B$, dirtyBitvector);
7      for $j \leftarrow i$ to $i + B$ increment by $P$ do
8          for $k \leftarrow j$ to $j + P$ increment by $1$ do
9              updateParity ← False;
10             if bitIsSet(dirtyBitvector, $k - i$) then
11                 updateParity ← True;
12                 computePageChecksum($k$);
13             end
14         end
15         if updateParity then
16             computeParity($j$, $j + P$);
17         end
18     end
19     memoryFence;
20     dirtyBitvectorCopy ← 0;
21 end
22 computeMetaChecksum();
```

dirty bits further requires invalidating the corresponding TLB entries. Each of these is a costly operation, as evinced by prior research [6], and demonstrated by our experiments (Section 3.2.6). Batching allows pages to share the system call, fractions of the page table walk, and the TLB invalidation. We found that batching reduced the amount of time spent in checking and clearing dirty bits by up to two orders of magnitude.

Algorithm: Algorithm 1 details the steps that Vilamb's background thread performs on each invocation. Vilamb loops over all the N pages in a given DAX NVM file in increments of B pages; B being the batch size for which Vilamb checks the dirty bits using a single system call (Line 2). Vilamb stores a persistent shadow copy of the dirty bits (Line 3) and then clears them (Line 6). Vilamb updates the system-checksum of each dirty page (Line 12), and the parity of a group of P pages if either of them is dirty (Line 16). Vilamb stores the system-checksums and parity separately from the data (Fig. 3.2) and then clears the shadow copy of the dirty bits (Line 20). Vilamb then updates a meta-checksum (checksum of the page system-checksums) after every iteration (Line 22 and Fig. 3.2).

As a performance optimization, instead of storing a shadow copy of the dirty bit for each page, we use a single dirty bitvector of size B along with the current batch's starting page number (Line 3 and Line 4). Together, the starting page number and the dirty bitvector copy suffice to store shadow copies of the dirty bits for pages in the current batch; pages not in the current batch do not need a shadow copy of their dirty bits because their dirty bits are not being cleared. Having a single dirty bitvector improves performance by reducing cache pollution.

Vilamb's system-redundancy verification thread (i.e., the scrubbing thread) computes and verifies the checksum only for pages that are clean, i.e., they have neither their dirty bit nor their shadown dirty bit set. If the system-checksum verification succeeds, the thread moves to the next page. In case of a system-checksum mismatch, the scrubbing thread re-checks whether the page is clean. This second check is to ensure that the page was not modified after the first check but before the system-checksum verification. If the second check also indicates that the page is clean, the scrubbing thread raises a signal to halt the application. The file system can then recover the page, if it belongs to a clean stripe (we have not implemented recovery).

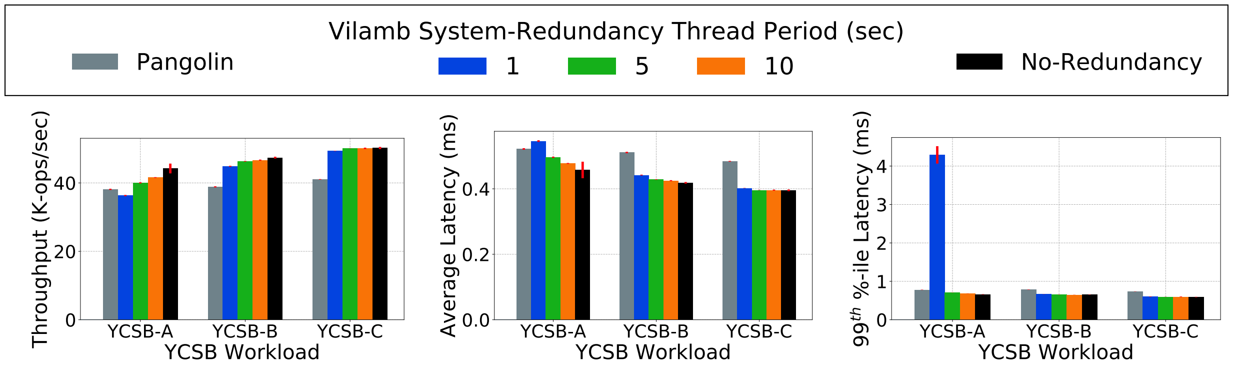
Leveraging Hardware Support: Our implementation of Vilamb leverages hardware-support whenever possible. We use CRC-32C checksums and employ the `crc32q` instruction when available. Similarly, we use SIMD instructions for computing the parity whenever possible (e.g., by operating on 256-byte words in our experiments). We never flush cache-lines for persistence because we assume battery-backed servers. We do, however, use fences to ensure ordering between updates. For example, the fence at Line 5 ensures that the shadow copy of the dirty bits and current batch's starting page number writes are completed before the dirty bits are cleared. Similarly, the fence at Line 19 ensures that system-redundancy is written before the dirty bits' shadow copy is cleared. We extend the same performance benefits (e.g., no cache-line flushes and SIMD parity computations) to the alternatives that we compare Vilamb with in our evaluation.

3.2 Evaluation

This section evaluates Vilamb and compares it to No-Redundancy and Pangolin, using eight macro- and micro-benchmarks. No-Redundancy serves as the baseline, providing the best performance but not implementing any system-redundancy. Pangolin is a state-of-the-art userspace library that updates system-redundancy when applications commit their data writes to NVM.

We obtained Pangolin's code from the authors and run it with system-checksum and parity updates enabled but system-checksum verification disabled (referred to as Pangolin-MLPC in the Pangolin paper [109]). We run Vilamb also with system-checksum and parity updates enabled and scrubbing (for background system-checksum verification) disabled. As shown in the evaluation of Pangolin [109], and confirmed by our experiments, system-checksum verification via scrubbing at reasonable frequencies incurs negligible overhead. Pangolin can also verify system-checksums on object reads, which Vilamb cannot, but doing so reduces throughput by up to 50% for large objects [109].

Unless mentioned otherwise, Vilamb uses a 512-page batch size for checking/clearing dirty bits. To accurately quantify Vilamb's overheads, we pin it to the same core(s) as the application. For single threaded applications such as Redis, this means that the application and Vilamb run on the same logical core (i.e., same hyper-thread). Each data point in our results is an average of three runs with root mean square error bars. We use a dual-socket Intel Xeon Silver 4114 machine with Linux 4.4.0 kernel with 4K pages for our experiments. The system has 192 GB DRAM, from which we use 64 GB as emulated NVM [72].



(a) Throughput (b) Average Latency (c) Tail Latency

Figure 3.3: YCSB with Redis: Throughput and read latency of YCSB workloads with Redis.

3.2.1 Key Evaluation Takeaways

Key takeaways from our evaluation include:

- Vilamb is low-overhead. For example, Vilamb with a 10 second system-redundancy update period reduces Redis' YCSB throughput by only 0.1–6% in comparison to No-Redundancy.
- Vilamb significantly outperforms Pangolin. For example, Vilamb has 3–5× higher insert throughput than Pangolin for five PMDK key-value stores. Even for low throughput applications like single threaded Redis serving YCSB, Vilamb has up to 18% higher throughput than Pangolin.
- Vilamb significantly increases the MTTDL. For example, Vilamb increases the MTTDL for PMDK key-value stores by up to two orders of magnitude.
- Vilamb offers a tradeoff between performance and time-to-coverage. For example, decreasing the delay between system-redundancy updates from 5 second to 1 second increases Redis' YCSB-A MTTDL by 3× but decreases the throughput by 10%.
- Vilamb's battery requirements are low. Across all of our workloads, the cost of batteries that Vilamb requires never exceeds \$10.

3.2.2 YCSB with Redis

Redis [80] is a widely used open-source NoSQL DBMS. We modify it to use a DAX NVM file for its data heap. Our implementation uses the `libpmemobj` library [37] from the Intel persistent memory development kit (PMDK) [39] for No-Redundancy.

Modifying Redis to use Vilamb and Pangolin: For Vilamb, we added 10 lines of initialization and cleanup code in one file. The initialization code registers Redis' NVM heap with Vilamb and sets the system-redundancy update delay. To use Pangolin's transactional API (which is similar to but different than `libpmemobj`), we changed 346 lines of code across 10 files in Redis. Whereas most of these changes were to the transactional interface (e.g., using `pgl_tx_begin`), we also had to modify Redis to invoke Pangolin before reading data from an object (using `pgl_get`). Doing so enables Pangolin to determine whether the object is in NVM or in DRAM and provide Redis with the correct pointer.

Experimental Setup: We use three core YCSB workloads: YCSB-A (50:50 reads:updates), YCSB-B (95:5 reads:updates), and YCSB-C (read-only). We initialize the DBMS with 1M (1×2^{20}) key-value pairs for a NVM footprint of 10 GB and run the workloads for five minutes. The YCSB workload generator uses 20 threads and runs on a different socket than Redis.

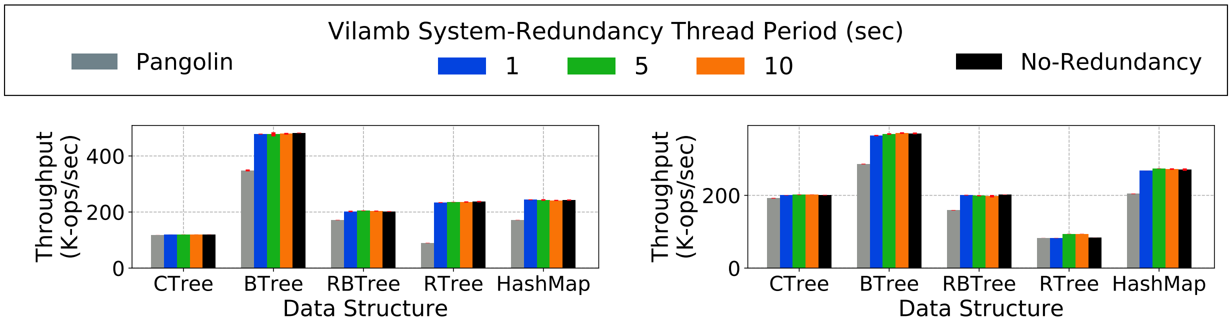
Results: Fig. 3.3 presents throughput and read latencies. Vilamb reduces the throughput, in comparison to No-Redundancy, by 0.1–6% for a system-redundancy update period of 10 second and by 1.6–17% for a period of 1 second. Increasing the delay for system-redundancy updates improves Vilamb's performance because it performs fewer system-redundancy updates and hogs less CPU. With aggressive system-redundancy updates every second, Vilamb increases the tail latency for YCSB-A because it stalls Redis while updating system-redundancy on the same core. This effect can be mitigated if Vilamb and Redis were to run on separate cores.

Pangolin's throughput is 13–18% lower than No-Redundancy, with a higher overhead for more read-heavy workloads. In addition to the overhead of updating system-redundancy, Pangolin incurs overhead because of two other factors, both related to its micro-buffering design. First, on every object read, Pangolin probes a cuckoo hash table to check whether the latest copy of the object is in a DRAM micro-buffer or in NVM. Second, when Redis adds an object to a transaction, Pangolin copies the entire object to DRAM for micro-buffering, rather than just the modified data ranges.

For the write-heavy workload YCSB-A, Pangolin outperforms Vilamb with a system-redundancy update period of 1 second. This is because Pangolin's micro-buffering design enables it to perform system-checksum and parity updates using the diff of the updated data. Pangolin uses the new data in the DRAM micro-buffer and the old data in the NVM to compute the data diff. In contrast, Vilamb has to read the entire page to update the system-checksum, and also read other pages in the stripe to update the parity. With 5 and 10 second system-redundancy update periods, Vilamb outperforms Pangolin by 5–7%.

For read-heavy workloads YCSB-B and YCSB-C, Vilamb reduces the throughput marginally (e.g., less than 2% for YCSB-C) whereas Pangolin reduces the throughput by 18%. This is because even though the number of system-redundancy updates reduce, Pangolin continues to incur the additional overheads described above. For example, Pangolin has to check whether the data is in DRAM or NVM for object reads.

Pangolin's moderate overhead (up to 18%) compared to No-Redundancy and Vilamb is an artifact of Redis' inefficiencies. In particular, Redis' single-threaded design causes it to have low performance (tens of thousands of operations per second) that does not fully expose the system-redundancy update overheads. In the next section, we show that multi-threaded key-value stores that perform millions of operations per second benefit significantly from Vilamb's asynchronous approach.



(a) Insert Throughput

(b) Remove Throughput

Figure 3.4: PMDK Key-Value Stores: Throughput for insert-only, remove-only benchmarks with different PMDK key-value stores.

3.2.3 PMDK Key-Value Stores

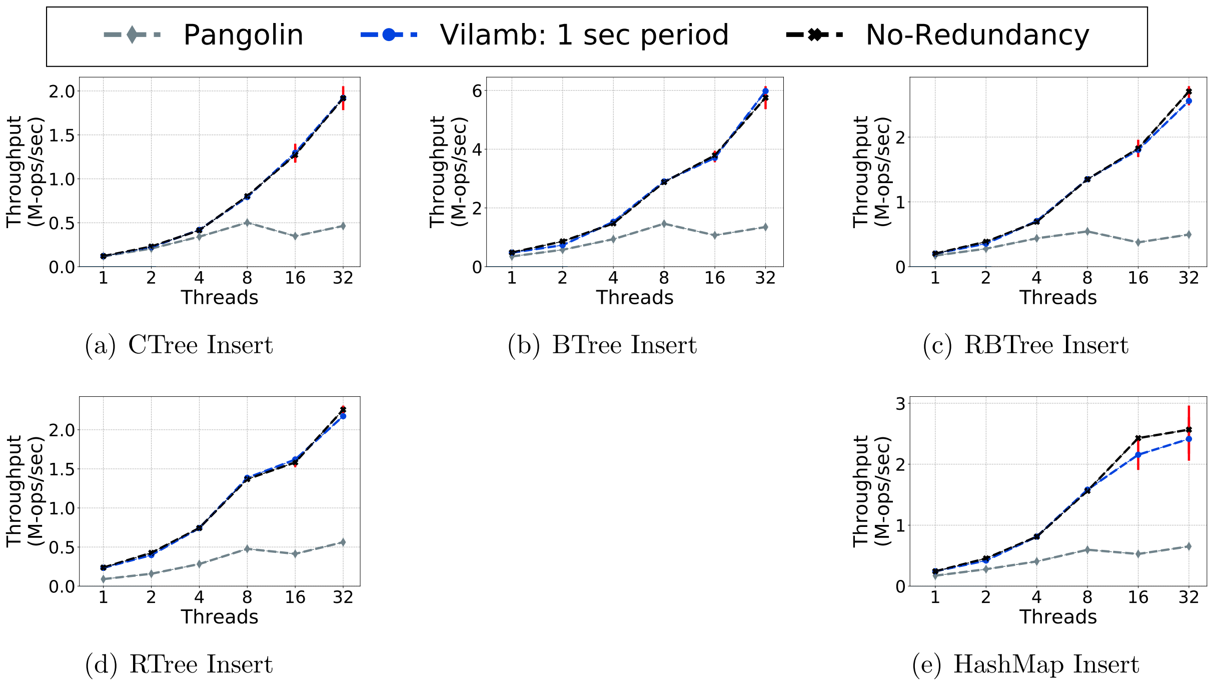
Intel persistent memory development kit (PMDK) [39] implements NVM-optimized key-value stores and includes performance benchmarks.

Experimental Setup: Similar to Pangolin [109], we use insert-only, and remove-only benchmarks for five key-value stores: Crit-Bit Tree (CTree), BTree, Red-Black Tree (RBTree), Range Tree (RTree) and chaining hashmap (HashMap). We first re-create the experiment and results from Pangolin [109] with a single-thread that performs 5 million operations. We then use multiple threads (1 to 32) with 100,000 operations per thread.

We modify the PMDK benchmark for multi-threaded benchmarking. In the original implementation, the threads synchronize using a coarse-grained lock; each thread holds a lock over the entire data structure for the entire duration of its transaction. Not surprisingly, the coarse-grained lock leads to poor scaling. We modified the implementation such that each thread maintains and operates on its own instance of the data structure. All the threads share the same NVM pool, but do not synchronize their changes because they operate on different data. Our modifications enabled close to linear scaling for the baseline case of No-Redundancy.

Results: Figs. 3.4(a) and 3.4(b) show the throughput for the insert-only and remove-only workloads when using a single thread for the key-value store. Pangolin's overheads are similar to those reported in their paper [109]. Vilamb's performance improves with increasing delay in system-redundancy updates. Of the five key-value stores, both Pangolin and Vilamb have the highest overhead in comparison to No-Redundancy for RTree because RTree's insertion touches the largest amount of data. For the remove-only workload, Pangolin outperforms Vilamb with 1 second system-redundancy update period because removing objects touches only a small amount of data and Pangolin can efficiently update system-redundancy using the diffs for small data.

Figs. 3.5(a) to 3.5(e) show the insert-only throughput for the five key-value stores with increasing number of threads. Increasing the number of threads updates NVM data more aggressively and generates more system-redundancy updates. This causes Pangolin to have up to 80% lower throughput than No-Redundancy. Across the the five key-value store, Vilamb has 3–5× higher throughput than Pangolin when using 32 threads.



(a) CTree Insert (b) BTree Insert (c) RBTree Insert

(d) RTree Insert (e) HashMap Insert

Figure 3.5: PMDK Key-Value Stores: Throughput for insert-only, remove-only benchmarks with different PMDK key-value stores.

3.2.4 NVM Transaction Microbenchmarks

Pangolin [109] used micro-benchmarks to measure the latency of transactional operations (allocation, overwrite, and deallocation), and to measure the scalability of overwriting NVM regions with multiple threads.

Experimental Setup: We perform each transactional operation (allocation, overwrite, deallocation) 1 million times for different sized objects in a single thread and report the average latency. We use an NVM file of 10 GB for this. For scalability, we increase the number of threads with each thread overwriting 64-byte and 4 KB regions 200,000 times.

Results: Fig. 3.6 shows the latency for performing the transactional operations using a single thread. For 64-byte objects, Pangolin incurs 23%, 44%, and 30% higher latency than No-Redundancy for allocation, overwrite, and deallocation, respectively. In contrast, Vilamb with a system-redundancy update period of 1 second increases the corresponding latencies by only 9%, 5%, and 3%; increasing the system-redundancy update period further reduces Vilamb's latencies. Increasing the object sizes increases the latency for all configurations, because more data is touched (except for deallocation, in which only metadata is updated). However, even for 4 KB objects, Vilamb with a system-redundancy update period of 1 second has 13%–31% lower latencies than than Pangolin.

Fig. 3.7 shows the throughput for overwriting 64-byte and 4 KB regions with an increasing number of threads. Vilamb scales close to No-Redundancy, with only up to 25% lower throughput. In contrast, Pangolin has up to 77% lower throughput. Pangolin's experiments with real NVM (in contrast to our DRAM-based emulation) showed that No-Redundancy performance does not scale well beyond 8 threads because of NVM's

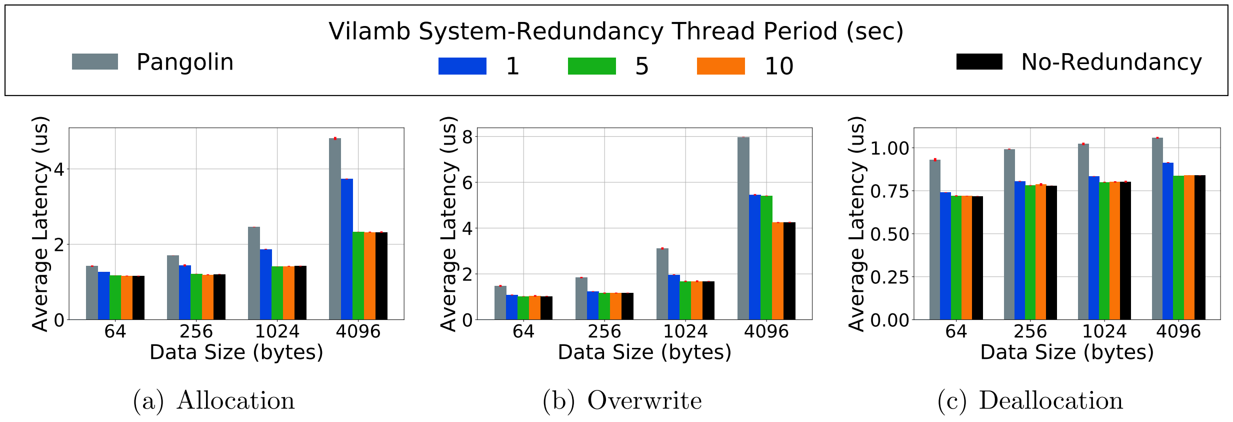


Figure 3.6: NVM Transaction Latencies: Latencies for transactional allocation, overwriting, and deallocation.

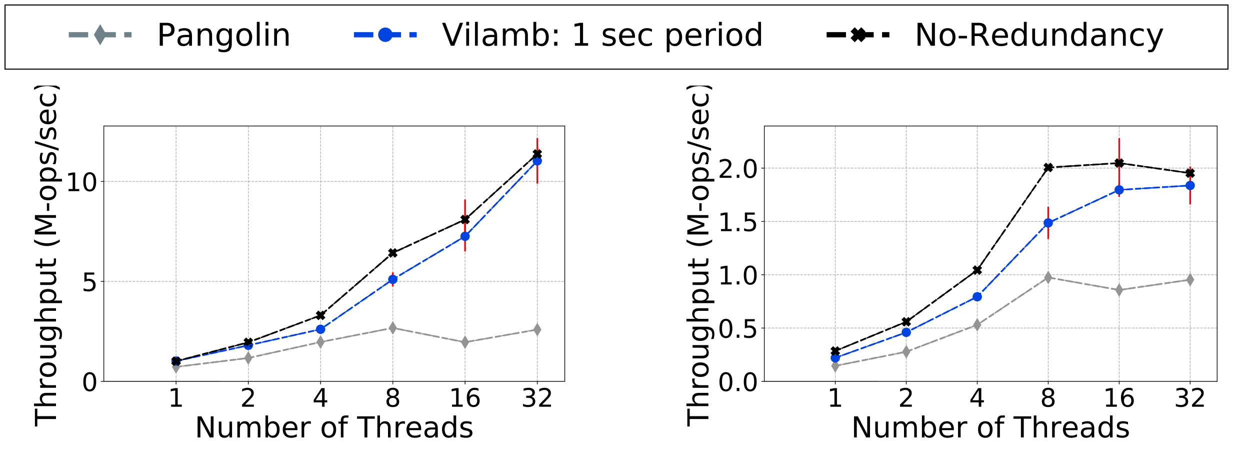
limited bandwidth [109]. However, even with 8 threads Vilamb's throughput is double of Pangolin's. As NVM performance improves and gets closer to DRAM performance, the benefits of Vilamb's asynchronous system-redundancy maintenance will become more pronounced. We also evaluated overwriting with other intermediate data sizes (256 and 1024 bytes) and obtained similar trends.

3.2.5 Fio Microbenchmarks

This section evaluates Vilamb's performance using fio [10] microbenchmarks. We cannot evaluate Pangolin using fio because fio's NVM engine [30] does not use object based transactions. Rather, fio treats the entire DAX-mapped file as a raw sequence of bytes. This illustrates Pangolin's programming model restriction. Applications that manage DAX-mapped data themselves, either as raw data as in fio microbenchmarks or in more complex fashion like NVM databases [8], can benefit from Pangolin only if they can be and are modified to use its APIs.

Experimental Setup: Fio's libpmem engine reads/writes DAX NVM files at a cache-line granularity. We use write-only and read-only workloads with a 16 GB file and three access patterns: uniform random, sequential, and Zipf. The workloads perform reads/writes equal to the file size. The random and sequential workloads choose previously unread/unwritten cache-lines, consequently reading/writing each cache-line in the entire file exactly once. We use a single thread and pin it to a logical core along with Vilamb. Experiments with the read-only workloads quantify the overhead of Vilamb's dirty bit checking, because that is the only work Vilamb has to perform for read-only workloads.

Results: Fig. 3.8 shows the throughput for the two workloads with three access patterns each. For write-only workloads, Vilamb reduces throughput by 0.5–56% with higher overheads for more frequent system-redundancy updates. Vilamb's overheads are highest for the random workload and lowest for the sequential workload; sequential workloads offer the best opportunity to reduce computations, because successive cache-line writes belong to the same page. Even for random workloads, the overhead is only 10% with a system-redundancy update delay of 60 seconds. Vilamb reduces the throughput by only



(a) 64 Byte Writes (b) 4096 Byte Writes

Figure 3.7: NVM Overwrite Throughput

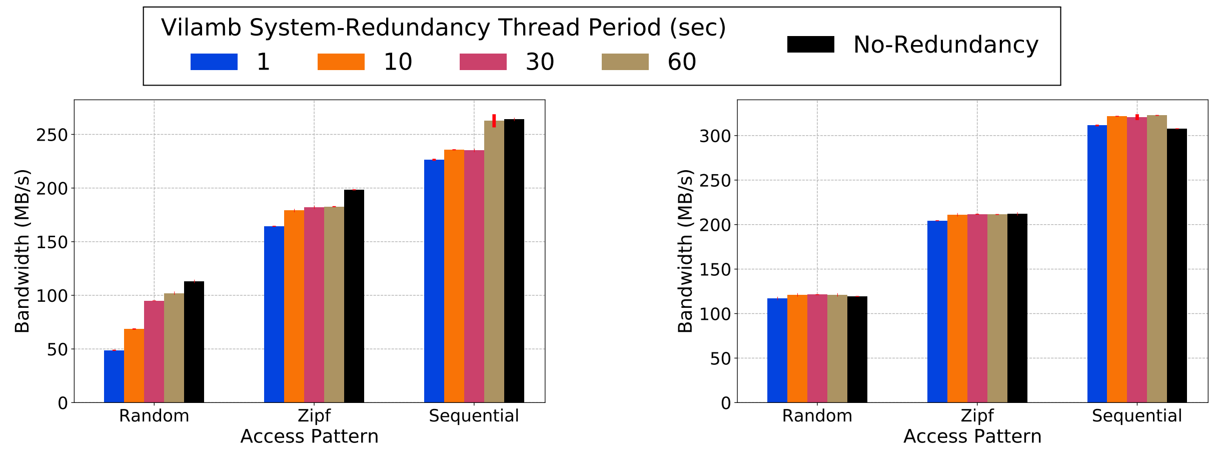
up to 3% for read-only workloads, demonstrating the efficacy of its checking of dirty bits. Vilamb's througput is higher than No-Redundancy for the read-only sequential workload with an update period of more than 10 seconds; this is an artifact of the experimental setup. While checking for dirty bits, Vilamb populates the page table entries and reduces the number of soft page faults. The performance benefit of reduced soft page faults outweigh the overhead of checking the dirty bits infrequently (i.e., with a period of more than 10 seconds). This anamoulous inversion of performance can be resolved by pre-populating the page table entries for No-Redundancy as well.

3.2.6 Cost of Checking/Clearing Dirty Bits

To better understand the cost of checking and clearing dirty bits, we break down the cost into its constituent components: (i) system call, (ii) page table walk to desired page table entries, (iii) reading/resetting the dirty bits, and (iv) TLB invalidation after clearing dirty bits. We also demonstrate the benefits of batching multiple pages when checking and clearing the dirty bits.

Experimental Setup: We use the write-only fio workload with 64-byte writes and a uniform random access pattern. We configure Vilamb to check/clear the dirty bits every second. We measure the average amount of time spent in each of the components for a single invocation of Vilamb's background thread. We vary the batch size to demonstrate the impact of batching.

Results: Fig. 3.9(a) presents the time spent in various components of checking and clearing dirty bits. The batch size is set to 512 pages for this experiment. Doubling the file size, and consequently the total number of pages, roughly doubles the amount of time spent in each of the components. This is because the number of system calls, page walks, and reads of the dirty bits are all directly proportional to the total number of pages. The number of pages for which the dirty bit is cleared and the number of TLB invalidations depend on the workload's access pattern. For the uniform random access workload, these are also directly proportional to the total number of pages.



(a) Write Only Workload

(b) Read Only Workload

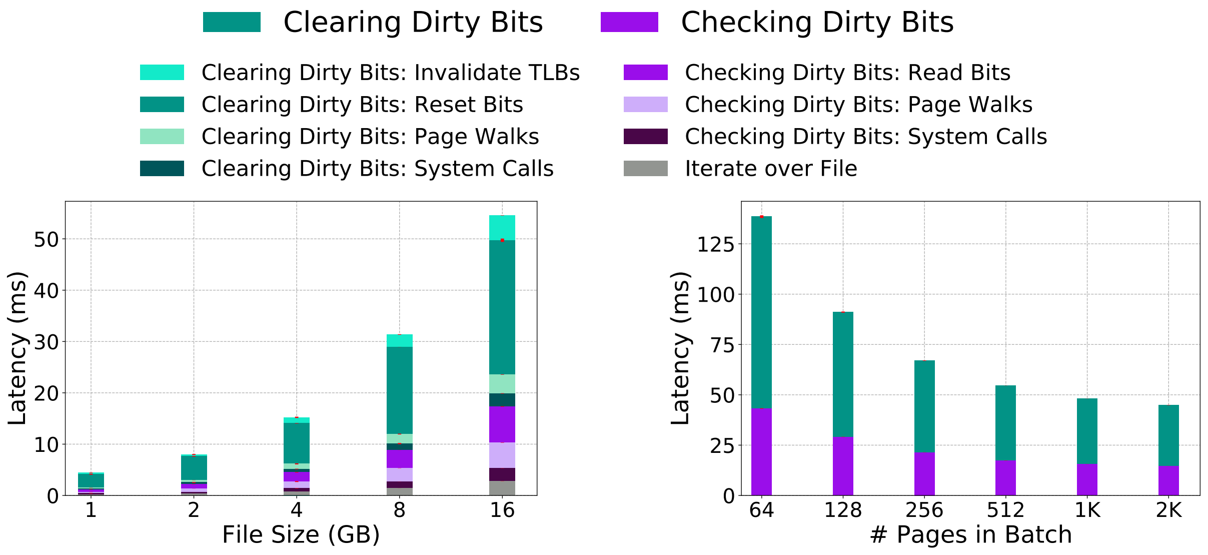
Figure 3.8: Fio Microbenchmarks: Throughputs for write-only and read-only workloads with different access patterns.

Fig. 3.9(b) presents the impact of batch size for a 16 GB file. As the batch size increases, the time spent in checking/clearing dirty bits decreases with diminishing marginal returns. This decrease is because the number of system calls reduce and larger fractions of the page table walks are shared between the pages in the same batch. The benefits are diminishing with increasing batch size, because of the fixed cost of reading all the dirty bits and resetting the ones that are found to be set.

3.2.7 Battery Capacity Requirements

This section analyzes the cost of batteries required for Vilamb to update the system-redundancy after a power failure for various workloads. We consider two kinds of batteries: ultra-capacitors that cost \$2.85/KJ [66, 99], and lithium-ion batteries that cost \$0.02/KJ [67, 99]. Conventionally, datacenters use lithium-ion batteries; modern datacenters additionally use ultra-capacitors because of their higher energy efficiency and density [99]. We consider servers with 500W [99] power usage.

For Redis with the write-heavy workload YCSB-A, one iteration of Vilamb's system-redundancy updates takes 143 ms when performed every second and 562 ms when performed every 10 seconds. These correspond to less than 1 KJ of energy required, i.e. the cost would be less than \$2.85 when using ultra-capacitors and less than \$0.02 when using the conventional lithium-ion batteries. This is the case for all PMDK key-value stores except RTree as well. For RTree, because of its sparse and large writes, Vilamb can require up to 5 seconds to update the system-redundancy upon a power failure, requiring 2.5 KJ of energy. This corresponds to \$7.2 in ultra-capacitor cost or \$0.05 lithium-ion battery cost. For fio, even with the adversarial random write workload with a system-redundancy update period of every 60 seconds, Vilamb requires only 4.5 seconds after a power failure. This translates to 2.25 KJ of required energy and \$6.4 in ultra-capacitor cost or \$0.04 in lithium-ion battery cost. The battery requirement, and the associated cost, can be further reduced by



(a) Breakdown of Time Spent (b) Impact of Batch Size

Figure 3.9: Cost of Checking/Clearing Dirty Bits: 3.9(a) shows the time spent in each component of checking/clearing dirty bits for a batch size of 512 pages and increasing file sizes. 3.9(b) shows that increasing the batch size reduces the time spent in checking/clearing dirty bits with diminishing returns.

limiting the number of pages that can be dirty (i.e., with outdated system-redundancy) using Viyojit's [48] design.

3.2.8 Reliability Analysis

We now evaluate the increase in mean time to data loss (MTTDL) over No-Redundancy when using Vilamb. For No-Redundancy, a single page corruption causes data loss. $MTTDL_{No-Redundancy} = \frac{MTTF_{\text{PAGE}}}{P}$, where P is the number of pages in the system.

A page corruption affects data protected with Vilamb in different ways. If the corruption affects a page that is dirty, Vilamb would checksum the corruption, leading to a silent data corruption. If the corruption affects a page that is itself clean but belongs to a stripe with a dirty page (hence, an outdated parity), Vilamb cannot recover the page, causing a data loss. For a corruption that affects a page that is itself clean and belongs to a stripe with all clean pages, Vilamb can recover the page. In summary, if the corruption affects a page in a *vulnerable stripe*, i.e., a stripe with even one dirty page, it would lead to data loss. $MTTDL_{Vilamb} = \frac{MTTF_{\text{PAGE}}}{V \times N}$, where V is the number of vulnerable stripes, and N is the number of pages in a stripe. Vilamb increases the $MTTDL$ by $\frac{P}{V \times N}$ in comparison to No-Redundancy.

We use the above to compute the increase in the MTTDL with Vilamb over No-Redundancy for the various applications and workloads described in Section 3.2. Workload access patterns, i.e., the rate and locality of their data updates determine the number of vulnerable stripes. We emperically measure the average number of vulnerable stripes for the various workloads and use that to compute the increase in MTTDL. For Redis, Vilamb with a system-redundancy update period of 1 second increases the MTTDL by 15× for the write-heavy workload YCSB-A and 74× for the ready-heavy workload YCSB-B. Increasing the delay reduces the MTTDL, because a larger fraction of data remains dirty (e.g., 21× and 13× for YCSB-B with 5 second and 10 second period, respectively). For PMDK's key-value stores, Vilamb increases the MTTDL by up to two orders of magnitude (e.g., 112× for RBTree insert-only workload with 32 threads).

3.3 Conclusion

Vilamb provides low-overhead system-redundancy for DAX NVM data by embracing an asynchronous approach. In doing so, Vilamb creates a tunable trade-off between performance and time-to-coverage. For example, decreasing the system-redundancy update delay from 5 seconds to 1 second reduces Vilamb's throughput for Redis with YCSB-A workload by 10% but also increases the MTTDL by 3×. Vilamb's asynchronous approach amortizes the performance overhead of updating system-redundancy over multiple data writes. As a result, Vilamb outperforms the state-of-the-art synchronous system-redundancy solution, Pangolin, by up to 5×. Although Vilamb's delayed data coverage design is not suited for all applications, it adds a high throughput option to the suite of DAX NVM system-redundancy options available to applications.

Chapter 4

Tvarak: Software-managed hardware offload for redundancy in direct-access NVM storage

This chapter proposes **Tvarak** [49][1], a software-managed hardware offload that efficiently maintains system-redundancy for direct access (DAX) NVM data. Tvarak resides with the last-level cache (LLC) controllers and coordinates with the file system to provide DAX data coverage without application involvement. The file system informs Tvarak when it DAX-maps a file. Tvarak verifies each DAX NVM cache-line read and updates the system-redundancy for each DAX NVM cache-line write-back.

Tvarak's design relies on two key elements to achieve efficient system-redundancy verification and updates. First, Tvarak reconciles the mismatch between DAX granularities (typically 64-byte cache-lines) and typical 4KB system-checksum block sizes by introducing cache-line granular system-checksums (only) while data is DAX-mapped. Tvarak accesses these cache-line granular system-checksums, which are themselves packed into cache-line-sized units, via separate NVM accesses. Maintaining these checksums only for DAX-mapped data limits the resulting space overhead. Second, Tvarak uses caching to reduce the number of extra NVM accesses for system-redundancy information. Applications' data access locality leads to reuse of system-checksum and parity cache-lines; Tvarak leverages this reuse with a small dedicated on-controller cache and configurable LLC partitions for system-redundancy information.

Simulation-based evaluation with seven applications, each with multiple workloads, demonstrates Tvarak's promise of efficient synchronous DAX NVM storage system-redundancy. For Redis, Tvarak incurs only a 3% slowdown for maintaining system-redundancy with a set-only workload, in contrast to 50% slowdown with Pangolin-like software approach, without compromising on coverage or checks. For other applications and workloads, the results consistently show that Tvarak efficiently updates and verifies system-checksums and parity, especially in comparison to software-based synchronous alternatives. The efficiency benefits are seen in both application runtimes and energy.

[1]Tvarak means accelerator in Hindi.

4.1 Tvarak Design

Tvarak is a hardware controller that is co-located with the last-level cache (LLC) bank controllers. It coordinates with the file system to protect DAX-mapped data from firmware-bug-induced corruptions. We first outline the goals of Tvarak. We then start by describing a naive system-redundancy controller, and improve its design to reduce its overheads, leading to Tvarak. We end with Tvarak's architecture, area overheads, and walk through examples.

4.1.1 Tvarak's Goals and Non-Goals

Tvarak intends to enable the following for DAX-mapped NVM data: (i) detection of firmware-bug induced data corruption, and (ii) recovery from such corruptions. To this end, the file system and Tvarak maintain per-page system-checksums and cross-DIMM parity with page striping, as shown in Fig. 4.1. Note that a RAID-4 or RAID-5 like geometry (as shown in Fig. 4.1) precludes a cache-line granular interleaving across the NVM DIMMs. This is because a cache-line granular interleaving would require certain cache-lines in pages (e.g., every fourth cache-line in case of 4 NVM DIMMs) to be reserved for parity, breaking the contiguity or application's virtual address space. Instead, a page-granular interleaving enables the OS to map contiguous virtual pages to data pages (bypassing the parity pages) and retain the contiguity of application's virtual address space. This restriction is not unique to Tvarak and required for any scheme that implements a RAID-4 or RAID-5 like geometry.

Tvarak's system-redundancy mechanisms co-exist with other complementary file system redundancy mechanisms that each serve a different purpose. These complementary mechanisms do not protect against firmware-bug-induced corruptions, and Tvarak does not intend to protect against the failures that these mechanisms cover. Examples of such complementary redundancy mechanisms include remote replication for machine failures [31, 41, 51, 69], snapshots for user errors [33, 85, 105, 111], and inline sanity checks for file system bugs [54].

Although not Tvarak's primary intent, Tvarak also aids in protecting data from random bit flips and in recovery from DIMM failures. Tvarak can detect random bit flips because it maintains a checksum over the data. This coverage is in concert with device-level cache-line granular ECCs [24, 42, 94] that are designed for detecting and recovering from random bit flips. Tvarak's cross-DIMM parity also enables recovery from DIMM failures. The file system and Tvarak ensure that recovery from a DIMM failure does not use corrupted data/parity from other DIMMs. To that end, the system-checksum for a page is stored in the same DIMM as the page, and the file system verifies a page's data with its system-checksum before using it.

4.1.2 Naive System-Redundancy Controller Design

Fig. 4.2 illustrates a basic system-redundancy controller design that satisfies the requirements for detecting firmware-bug-induced corruptions, as described in Section 2.3. We will improve this naive design to build up to Tvarak. The naive controller resides above the device firmware in the data path (with the LLC bank controllers). The file system informs the

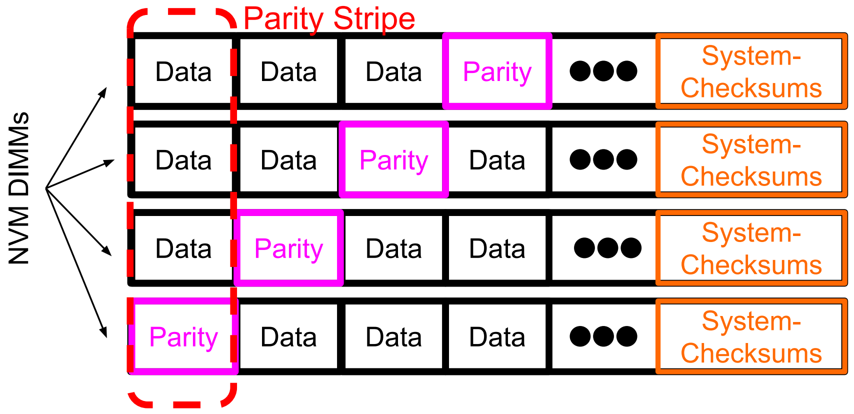


Figure 4.1: Tvarak coordinates with the file system to maintain per-page system-checksums and cross-DIMM parity akin to RAID-5 with page striping.

controller about physical page ranges of a file when it DAX-maps the file, along with the corresponding system-checksum pages and the parity scheme. For each cache-line write-back from the LLC and cache-line read into the LLC, the controller performs an address range matching. The controller does not do anything for cache-lines that do not belong to a DAX-mapped region, as illustrated in the leftmost access in Fig. 4.2. The file system continues to maintain the system-redundancy for such data [71, 105].

For DAX-mapped cache-lines, the controller updates and verifies system-redundancy using separate accesses from the corresponding data. The request in the center of Fig. 4.2 shows a DAX cache-line read. To verify the read, the controller reads the entire page (shown with black arrows), computes the page's checksum, reads the page's system-checksum (shown in olive) and verifies that the two match. The rightmost request in Fig. 4.2 shows a cache-line write. The controller reads the old data in the cache-line, the old system-checksum, and the old parity (illustrated using black, olive and pink, respectively). It then computes the data diff using the old and the new data and uses that to compute the new system-checksum and parity values[2]. It then writes the new data, new system-checksum, and the new parity to NVM. The cross-DIMM parity design and the use of data diffs to update parity is similar to recently proposed RAIM-5b [114].

Note that we assume that the NVM servers are equipped with backup power to flush CPU caches in case of a power failure. This guarantees that the controller can complete the system-checksum and parity writes for a data write in case of a power failure, even if the controller caches this information, as we will describe later. The backup power could either be from top-of-the-rack batteries with OS/BIOS support to flush caches, or ADR-like support for caches with cache-controller managed flushing. Both of these designs are common in production systems [3, 26, 33, 48, 66, 70, 113]. Backup power also eliminates the need for durability-induced cache-line flushes and improves performance [66, 113] at a low cost (e.g., $2 for a 16 MB LLC [66]). We extend this assumption, and its performance benefits, to the all the designs we compare Tvarak to in Section 4.2.

[2]We assume that the controller implements incremental system-checksums that can be updated using the data diffs, e.g., CRC.

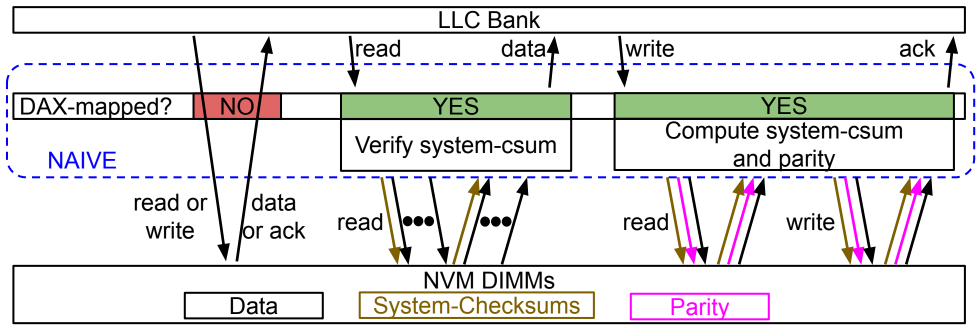


Figure 4.2: Naive System-Redundancy Controller Design: The system-redundancy controller operates only on DAX-mapped data. For DAX-mapped cache-line reads, the controller reads the entire page to compute the checksum, reads the system-checksum, and verifies that the two match. For cache-line writes, it reads the old data, system-checksum, and parity, computes the data diff, uses that to compute the new system-checksum and parity, and writes them back.

4.1.3 Efficient Checksum Verification

Verifying system-checksums in naive design incurs a high overhead because it has to read the entire page, as shown in Fig. 4.2. For typical granularities of 4KB checksum pages and and 64B cache-lines, the naive design reads 65$\times$ more cache-lines (64 cache-line in a page and one for the checksum). The read amplification would be even higher with huge pages. Although a smaller checksum granularity would reduce the checksum verification overhead, doing so would require dedicating more of the expensive NVM storage for redundant data. For example, per-cache-line checksums would require 64$\times$ more space than per-page checksums for 4KB pages with 64 byte cache-lines. Instead, the trend in storage system designs is to move towards larger, rather than smaller, checksum granularities [58, 93, 101].

We introduce *DAX-CL-checksums* to reconcile the performance overhead of page-granular checksum verification with the space overhead of cache-line-granular checksums. Adding DAX-CL-checksums to the naive controller results in the design shown in Fig. 4.3. As the name suggests, DAX-CL-checksums are cache-line-granular checksums that the controller maintains only when data is DAX-mapped. The read request in the middle of Fig. 4.3 illustrates that using DAX-CL-checksums reduces the read amplification to only 2$\times$ from 65$\times$ for the naive design—the controller only needs to read the DAX-CL-checksum in addition to the data to verify the read. The additional space required for DAX-CL-checksums is a small frction (e.g., $\frac{1}{64}^{\text{th}}$ of the DAX-mapped data size, assuming 64 byte cache-lines) and temporary in nature—it is required only for the fraction of NVM that an application has DAX-mapped and is freed after the application unmaps the NVM data. This is in contrast to the dedicated space overhead of maintaining fine-grained object-granular checksums for all NVM data at all times [109]. DAX-CL-checksums make the controller performance independent of the architecture's page size. This makes the controller suitable to use with huge pages, unlike Vilamb which incurs higher overhead with huge pages (Section 3.1.2).

The controller accesses DAX-CL-checksums separately from the corresponding data to ensure that the it continues to provide protection from firmware-bug-induced corruptions. For DAX-mapped cache-line writes, the controller updates the corresponding DAX-CL-

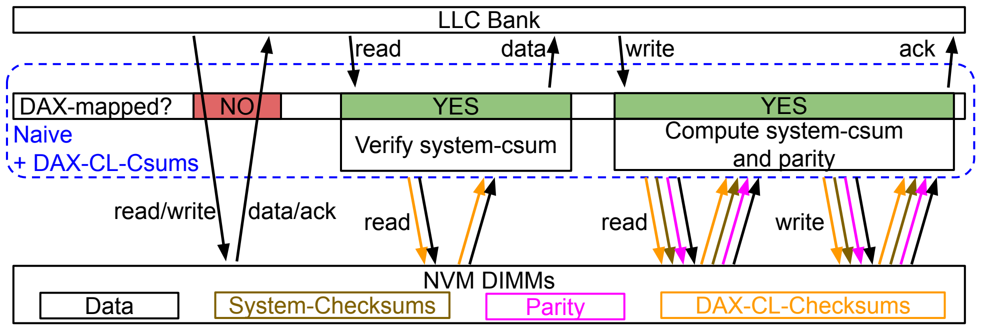


Figure 4.3: Efficient Checksum Verification: DAX-CL-checksums eliminate the need to read the entire page for DAX cache-line read verification. Instead, the controller only reads the cache-line and its corresponding DAX-CL-checksum.

checksum as well, using a similar process as that for system-checksums and parity (rightmost request in Fig. 4.3).

Managing DAX-CL-checksums is simple because the controller uses them only while data is DAX-mapped. In particular, when recovering from any failure or crash, the file system verifies data integrity using system-checksums rather than DAX-CL-checksums. Thus the controller need not maintain the DAX-CL-checksums persistently. When the file system DAX-maps a file, the controller requests a buffer space for DAX-CL-checksums. The file system can allocate this buffer space in either NVM or in DRAM; our implementation stores DAX-CL-checksums in NVM. The file system reclaims this space when it unmaps a file. Unlike page system-checksums, DAX-CL-checksums need not be stored on the same DIMM as its corresponding data because they are not used to verify data during recovery from a DIMM failure.

4.1.4 Efficient Checksum and Parity Updates

The rightmost request in Fig. 4.3 shows that the controller incurs 4 NVM reads and writes for each cache-line write to update the system-redundancy. To reduce these NVM accesses, we note that system-redundancy information is cache-friendly. Checksums are typically small and multiple checksums fit in one cache-line. In our implementation of 4 byte CRC-32C checksums, one 64 byte cache-line holds 16 checksums. DAX-CL-checksums for consecutive cache-lines and system-checksums for consecutive physical pages in a DIMM belong to the same cache-line. Access locality in data leads to reuse of DAX-CL-checksum and system-checksum cache-lines. Similarly, accesses to logically consecutive pages lead to reuse of parity cache-lines because they belong to the same RAID stripe.

Fig. 4.4 shows a design that caches the system-redundancy data, i.e., system-checksums, DAX-CL-checksums, and parity, in a small on-controller cache. The controller does not cache the corresponding NVM data because the LLC already does that. The controller also uses a partition of the LLC to increase its cache space for system-redundancy information (not shown in the figure). Using a reserved LLC partition for caching system-redundancy information limits the interference with application data. The controller can insert up to 3 system-redundancy cache-lines (checksum, DAX-CL-checksum, and parity) per data cache-

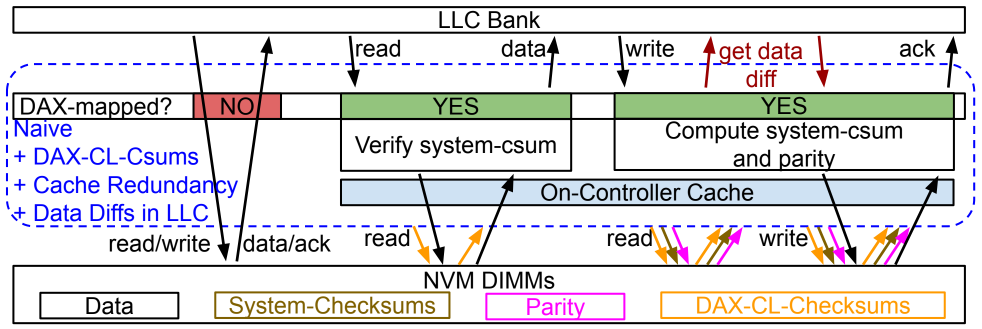


Figure 4.4: Efficient Checksum and Parity Updates: The controller caches system-redundancy cache-lines in an on-controller cache and a LLC partition (not shown). It also uses a LLC partition to store data diffs, eliminating the need to read the old data from NVM upon cache-line write-backs.

line write-back. If the controller were to share the entire LLC for caching system-redundancy information, each of these system-redundancy cache-lines could force out application data. Reserving a partition for system-redundancy information eliminates this possibility because the system-redundancy controller can only evict a system-redundancy cache-line when inserting a new one.

We further eliminate the need for the controller to fetch the old data from NVM to compute the data diff. Cache-lines in the LLC become dirty when they are evicted from the L2. Since the LLC already contains the soon-to-be-old data value, the controller uses it to compute the data diff and stores this diff in a LLC partition. This enables the controller to directly use this data diff upon a LLC cache-line write-back (shown as maroon arrows from the controller to the LLC in the rightmost request in Fig. 4.4). Upon an eviction from the LLC data diff partition (e.g., to insert a new data diff), the controller writes back the corresponding data without evicting it from the LLC, and marks the data cache-line as clean in the LLC. This ensures that the future eviction of the data cache-line would not require the controller to read the old data either, while allowing for reuse of the data in the LLC.

The LLC partitions (for caching system-redundancy and storing data diffs) are completely decoupled from the application data partitions. The LLC bank controllers do not lookup application data in system-redundancy and data diff partitions, and the system-redundancy controller does not look up system-redundancy or data diff cache-lines in application data partitions. Our design of storing the data diff in the LLC assumes inclusive caches. We evaluate the impact of exclusive caching in Section 4.2.8.

4.1.5 Putting it all together with Tvarak

Fig. 4.5 shows Tvarak's components that implement the functionality of a system-redundancy controller with all the optimizations described above (DAX-CL-checksums, system-redundancy caching and storing data diffs in LLC). One Tvarak controller resides with each LLC cache bank. Each Tvarak controller consists of comparators for address range matching and adders for checksum and parity computations. Tvarak includes a small on-controller cache

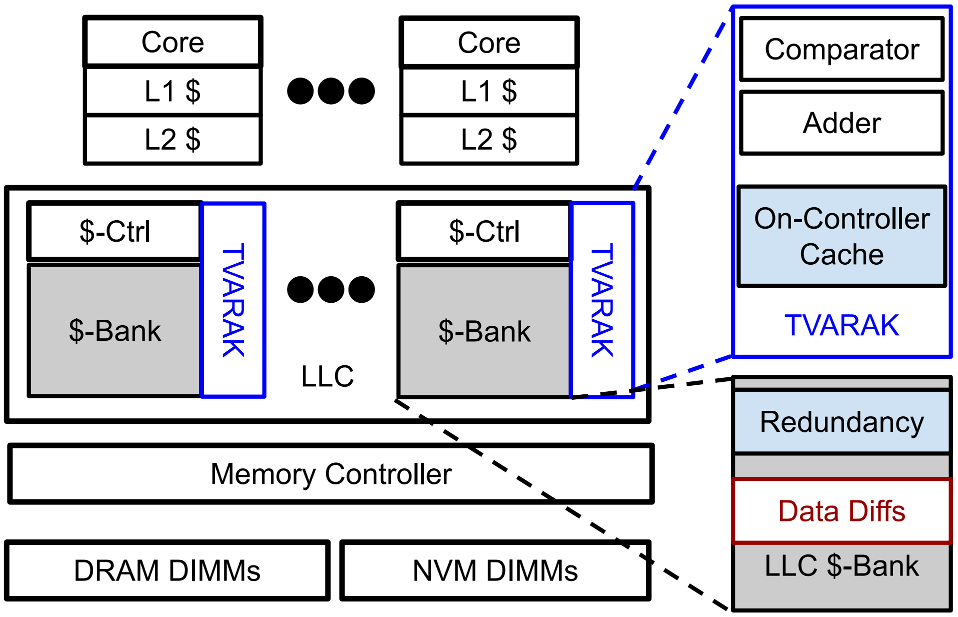


Figure 4.5: Tvarak resides with the LLC bank controllers. It includes comparators to identify cache-lines that belong to DAX-mapped pages and adders to compute checksums and parity. It includes a small on-controller system-redundancy cache that is backed by a LLC partition. Tvarak also stores the data diffs to compute checksums and parity.

for system-redundancy data and uses a LLC way-partitions for caching system-redundancy data. The on-controller cache and LLC way-partition form an inclusive cache hierarchy. The controllers use MESI coherence protocol for sharing the system-redundancy cache-lines between their private caches. Tvarak also uses a separate LLC way-partition of storing data diffs.

Area Overhead: The on-controller cache dominates Tvarak's area overhead because its other components (comparators and adders) only require small logic units. In our evaluation with 2MB LLC cache banks, each Tvarak controller consists of a 4KB cache. This implies that Tvarak's dedicated area is only 0.2% of the LLC. Tvarak's design of caching system-redundancy in an LLC partition instead of using its own large cache keeps Tvarak's dedicated area overheads low, without compromising on performance (Section 4.2).

DAX-mapped Cache-Line Accesses with Tvarak: For a DAX-mapped cache-line read, Tvarak computes the corresponding DAX-CL-checksum address and looks it up in the on-controller cache. Upon a miss, it looks up the DAX-CL-checksum in the LLC system-redundancy partition. If it misses in the LLC partition as well, Tvarak reads the DAX-CL-checksum from NVM and caches it. Tvarak reads the data cache-line from NVM, computes its checksum, and verifies it with the DAX-CL-checksum. If the checksum verification succeeds, Tvarak hands over the data to the bank controller. In case of an error, Tvarak raises an interrupt that traps into the OS; the file system then initiates a recovery using the cross-DIMM parity.

On a DAX-mapped cache-line write, Tvarak computes the corresponding system-checksum, DAX-CL-checksum, and parity addresses and reads them following the same

Redis	Set-only and get-only with 1–6 parallel instances
N-Store	Read heavy, balanced, and write-heavy YCSB workloads
C-Tree	Insert-only, and 100:0, 50:50, & 0:100 updates:reads with 12 parallel instances
B-Tree	Insert-only, and 100:0, 50:50, & 0:100 updates:reads with 12 parallel instances
RB-Tree	Insert-only, and 100:0, 50:50, & 0:100 updates:reads with 12 parallel instances
Fio	Sequential and random reads and writes with 12 threads
Stream	4 memory bandwidth intensive kernels with 12 threads

Table 4.1: Tvarak Evaluation: applications and their workloads.

process as above. Tvarak retrieves the data diff from the LLC bank partition and uses that to compute the new system-checksum, DAX-CL-checksum, and parity. Tvarak stores the updated system-redundancy information in the on-controller cache, and writes-back the data cache-line to NVM. Tvarak can safely cache the updated system-redundancy information because it assumes that servers are equipped with backup power to flush caches to persistence in case of a power failure (Section 4.1.2).

Tvarak fills an important gap in NVM storage system-redundancy with simple architectural changes. Integrating NVM devices into servers already requires changing the on-chip memory controller to support the new DDR-T protocol [40]. Hence, we believe that Tvarak can be easily integrated in storage server chips, especially given its low-overhead and energy-efficient design, as we show next.

4.2 Evaluation

We evaluate Tvarak with 7 applications and with multiple workloads for each application. Table 4.1 describes our applications and their workloads. Redis [81], Intel PMDK's [39] tree-based key-value stores (C-Tree, B-Tree, and RB-Tree), and N-Store [8] are NVM applications with complex access patterns. We also use fio [10] to generate synthetic sequential and random access patterns, and stream [91] for sequential access memory-bandwidth intensive microbenchmarks.

We compare Tvarak with three alternatives: *No-Redundancy*, *TxB-Object-Csums*, and *TxB-Page-Csums*. No-Redundancy implements no system-redundancy mechanisms. TxB-Object-Csums and TxB-Page-Csums are software-only system-redundancy approaches; TxB-Object-Csums is based on Pangolin [109] and TxB-Page-Csums is based on Mojim [112] and HotPot [89][3]. Both TxB-Object-Csums and TxB-Page-Csums update system-checksums and parity when applications inform the interposing library after completing a write, which is typically at a transaction boundary (TxB). TxB-Object-Csums maintains system-checksums at an object granularity, whereas TxB-Page-Csums maintains system-checksums at a page granularity. TxB-Object-Csums does not need to read the entire page to compute the system-checksum after a write, however, TxB-Object-Csums has higher space overhead because of the object-granular checksums. Unlike Pangolin, TxB-Object-Csums does not

[3] The original Mojim and HotPot designs do not include checksums, but can be extended to include them.

Cores	12 cores, x86-64 ISA, 2.27 GHz, Westmere-like OOO [86]
L1-D caches	32KB, 8-way set-associative, 4 cycle latency, LRU replacement, 15/33 pJ per hit/miss [64]
L1-I caches	32KB, 4-way set-associative, 3 cycle latency, LRU replacement, 15/33 pJ per hit/miss [64]
L2 caches	256KB, 8-way set-associative, 7 cycle latency, LRU replacement, 46/94 pJ per hit/miss [64]
L3 cache	24MB (12 2MB banks), 16-way set-associative, 27 cycle latency, shared and inclusive, MESI coherence, 64B lines LRU replacement, 240/500 pJ per hit/miss [64]
DRAM	6 DDR DIMMs, 15ns reads/writes
NVM	4 DDR DIMMs, 60ns reads, 150ns writes [56] 1.6/9 nJ per read/write [56]
Tvarak	4KB on-controller cache with 1 cycle latency, 15/33 pJ per hit/miss 2 cycle latency for address range matching 1 cycle per checksum/parity computation and verification, 2 ways (out of 16) reserved for caching redundancy information, 1 way (out of 16) for storing data diffs.

Table 4.2: Tvarak Evaluation: Simulation Parameters

copy data between NVM and DRAM. Consequently, TxB-Object-Csums avoids the data copying overhead of Pangolin (demonstrated in Section 3.2) and does not verify data reads (which can also incur up to 50% overhead [109]). However, because data is updated in place, TxB-Object-Csums also loses the ability to update parity using a data diff. TxB-Page-Csums also does not verify data reads and updates parity by recomputing it as opposed to using a data diff.

Tvarak updates system-checksums and parity upon every write-back from the LLC to the NVM, and verifies system-checksums upon every read from the NVM to the LLC. As mentioned in Section 4.1.2, we assume battery-backed CPU caches, so none of the designs flush cache-lines for durability.

Methodology: We use ZSim [86] to simulate a system similar to Intel Westmere processors [86]. Table 4.2 details our simulation parameters. We simulate 12 OOO cores, each with 32KB private L1 and 256KB private L2 caches. The cores share a 24MB last-level cache (LLC) with 12 banks of 2MB each. The simulated system consists of 6 DRAM DIMMs and 4 NVM DIMMs. For NVM DIMMs, we use the latency and energy parameters derived by Lee et al. [56] (60/150 ns read/write latency, 1.6/9 nJ per read/write). We use a fixed-work methodology and perform the same amount of application work for each design: No-Redundancy, Tvarak, TxB-Object-Csums, and TxB-Page-Csums. Unless stated otherwise, we present the average of three runs for each data point with root mean square error bars.

4.2.1 Key Evaluation Takeaways

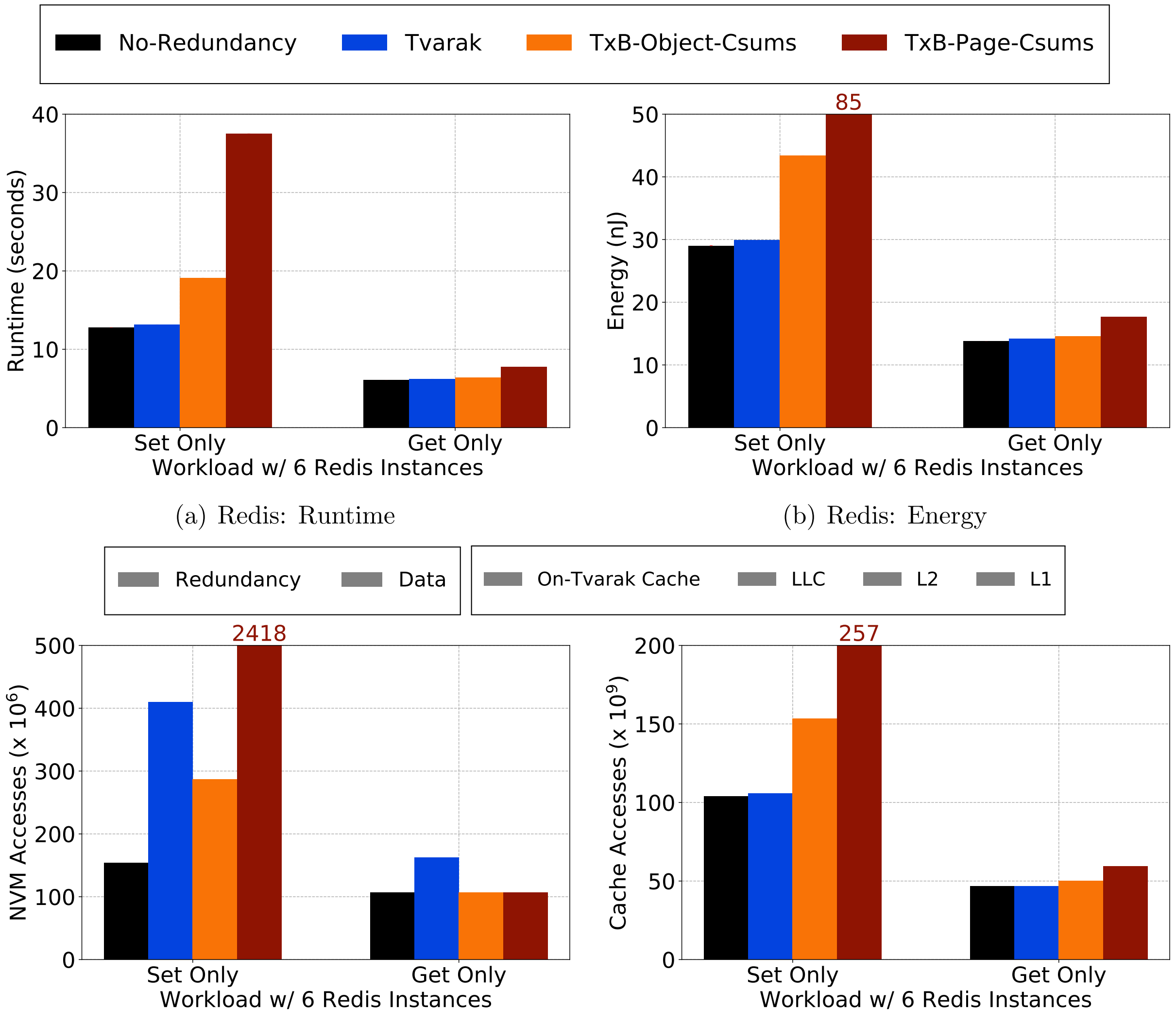
We highlight the key takeaways from our results before describing each application's results in detail.

- Tvarak provides efficient system-redundancy updates for application data writes, e.g., with only 1.5% overhead over No-Redundancy for an insert-only workload with tree-based key-value stores (C-Tree, B-Tree, RB-Tree).
- Tvarak verifies all application data reads, unlike most existing solutions, and does so efficiently. For example, in comparison to No-Redundancy, Tvarak slows down Redis get-only workload by only 3%.
- Tvarak benefits from application data access locality because it improves cache usage for system-redundancy information. For example, for synthetic fio benchmarks, Tvarak has negligible overheads with sequential accesses, but 2% overhead for random reads and 33% for random writes, compared to No-Redundancy.
- Tvarak outperforms existing software-only system-redundancy mechanisms. For example, for NStore workloads, TxB-Object-Csums is 33–53% slower than Tvarak, and TxB-Page-Csums is 180–390% slower than Tvarak.
- Tvarak's efficiency comes without an increase in (dedicated) space requirements. TxB-Object-Csums outperforms TxB-Page-Csums but at the cost of higher space overhead for per-object checksums. Tvarak instead uses DAX-CL-checksums that improve performance without demanding dedicated storage.

4.2.2 Redis

Redis is a widely used single-threaded in-memory key-value store that uses a hashtable as its primary data structure [80]. We modify Redis (v3.1) to use a persistent memory heap using Intel PMDK's `libpmemobj` library [39], building upon an open-source implementation [81]. We vary the number of Redis instances, each of which operate independently. We use the redis-benchmark utility to spawn 100 clients that together generate 1 million requests per Redis instance. We use set-only and get-only workloads. We show the results only for 6 Redis instances for ease of presentation; the trends are the same for 1–6 Redis instances that we evaluated.

Fig. 4.6(a) shows the runtime for Redis set-only and get-only workloads. In comparison to No-Redundancy, Tvarak increases the runtime by only 3% for both the workloads. In contrast, TxB-Object-Csums typically increases the runtime by 50% and TxB-Page-Csums by 200% over No-Redundancy for the set-only workload. For get-only workloads, TxB-Object-Csums and TxB-Page-Csums increase the runtime by a maximum of 5% and



(a) Redis: Runtime

(b) Redis: Energy

(c) Redis: NVM Accesses

(d) Redis: Cache Accesses

Figure 4.6: Redis: Runtime, energy, and NVM and cache accesses for Redis. We divide NVM accesses into data and system-redundancy information accesses, and cache accesses into L1, L2, LLC, and on-Tvarak cache.

28% over No-Redundancy, respectively. This increase for TxB-Object-Csums and TxB-Page-Csums, despite them not verifying any application data reads, is because Redis uses libpmemobj transactions for get requests as well and these transactions lead to persistent metadata writes (e.g., to set the transaction state as started or committed). Redis uses transactions for get requests because it uses an incremental hashing design wherein it rehashes its hashtable incrementally upon each request. The incremental rehashing can lead to writes for get requests also. We do not change Redis' behavior to eliminate these transactions to suit our get-only workload, which wouldn't actually trigger a rehashing.

Figs. 4.6(b) to 4.6(d) show the energy, NVM accesses and cache accesses. The energy results are similar to that for runtime. For the set-only workload, Tvarak performs more NVM accesses than TxB-Object-Csums because Tvarak does not cache the data or system-redundancy information in the L1 and L2 caches; TxB-Object-Csums instead performs more cache accesses. Even though TxB-Page-Csums can and does use the caches (demonstrated by TxB-Page-Csums's more than 2.5× more cache accesses than No-Redundancy), it also requires more NVM accesses because it needs to read the entire page to compute the page-granular system-checksums. For get-only workloads, Tvarak performs more NVM accesses than both TxB-Object-Csums and TxB-Page-Csums because it verifies the application data reads with DAX-CL-checksums.

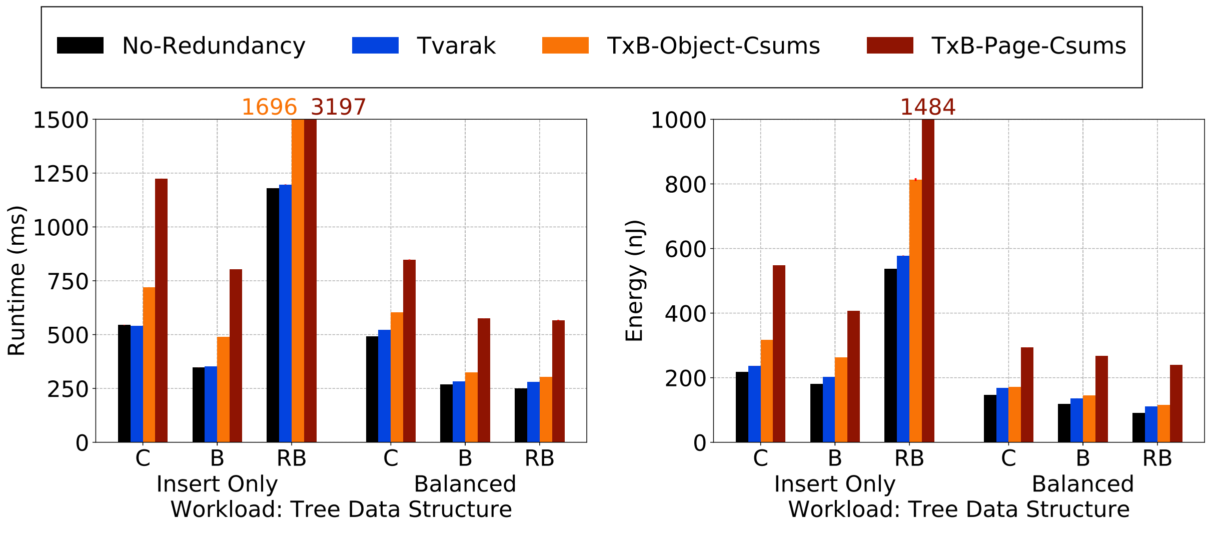
4.2.3 Key-value Data Structures

We use three persistent memory key-value data structures, namely C-Tree, B-Tree, and RB-Tree, from Intel PMDK [39]. We use PMDK's pmembench utility to generate insert-only, update-only, balanced (50:50 updates:reads), and read-only workloads. We stress the NVM usage by using 12 instances of each data-structure; each instance is driven by a single threaded workload generator. Having 12 independent instances of single-threaded workloads allows us to remove locks from the data-structures and increase the workload throughput. We show the results for insert-only and balanced workloads; the trends are the same for other workloads.

Fig. 4.7 show the runtime and energy for the different workloads and data-structures. For the insert-only workload, Tvarak increases the runtime by a maximum of 1.5% (for RB-Tree) over No-Redundancy while updating the system-redundancy for all inserted tuples. In contrast, TxB-Object-Csums and TxB-Page-Csums increase the runtime by 43% and 171% over No-Redundancy, respectively. For the balanced workload, Tvarak updates the system-redundancy for tuple updates and also verifies tuple reads with DAX-CL-checksums with only 5% increase in runtime over No-Redundancy for C-Tree and B-Tree. TxB-Object-Csums incurs a 20% increase in runtime over No-Redundancy for just updating the system-redundancy upon tuple updates; TxB-Page-Csums performs even worse.

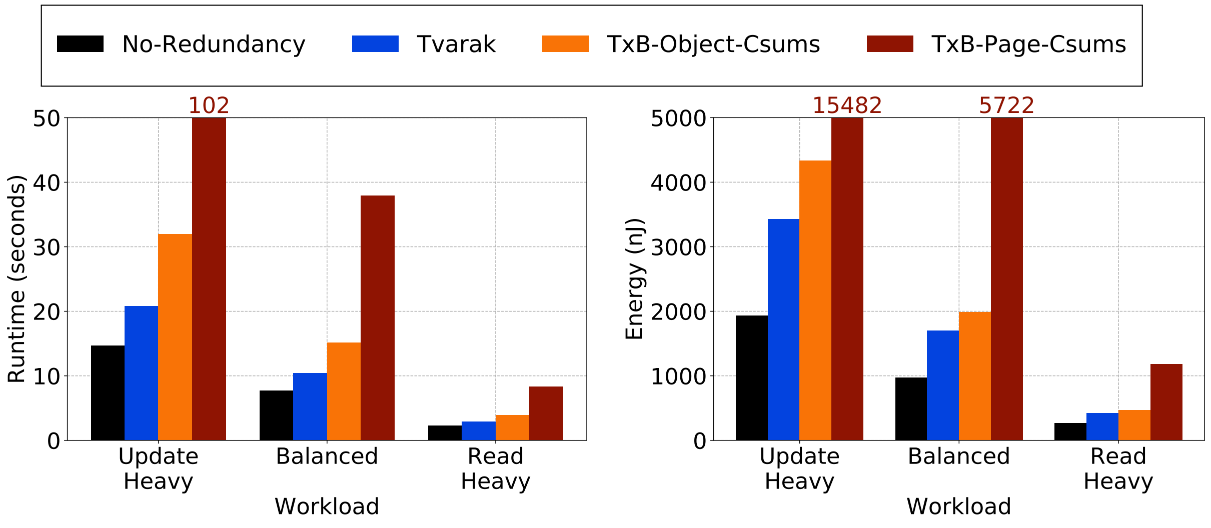
4.2.4 N-Store

N-Store is a NVM-optimized relational DBMS. We use update-heavy (90:10 updates:reads), balanced (50:50 updates:reads) and read-heavy (10:90 updates:reads) YCSB workloads with high skew (90% of transactions go to 10% of tuples) [8]. We use 4 client threads to drive



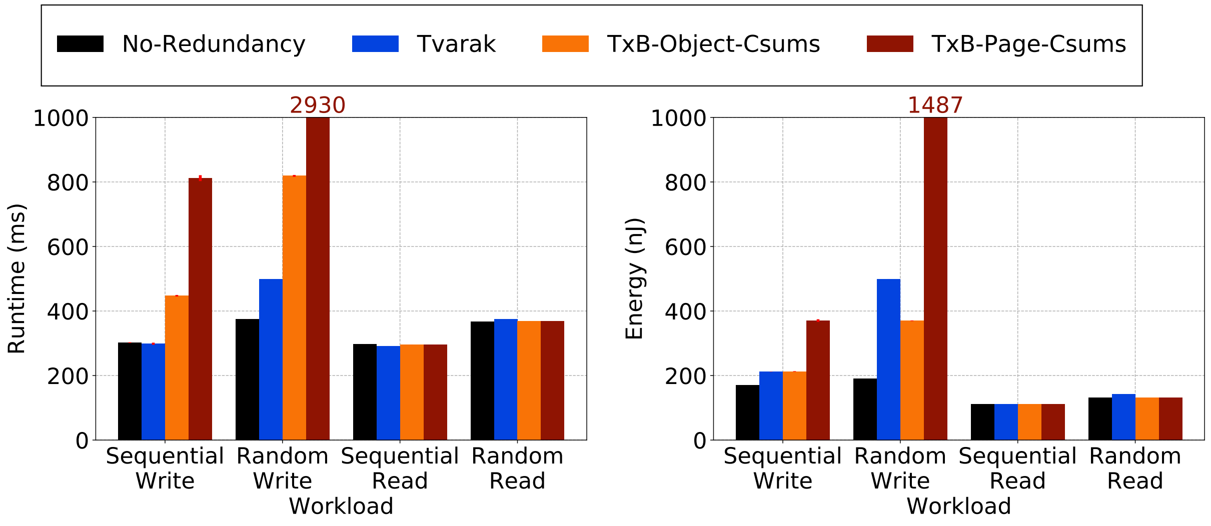
(a) KV-Structures: Runtime (b) KV-Structures: Energy

Figure 4.7: Key-Value Data Structures: Runtime and energy for various PMDK key-value stores.



(a) N-Store: Runtime (b) N-Store: Energy

Figure 4.8: N-Store: Runtime and energy for different YCSB access patterns with N-Store database.



(a) Fio: Runtime (b) Fio: Energy

Figure 4.9: Fio Microbenchmarks: Runtime and energy for sequential and random access patterns with fio microbenchmarks.

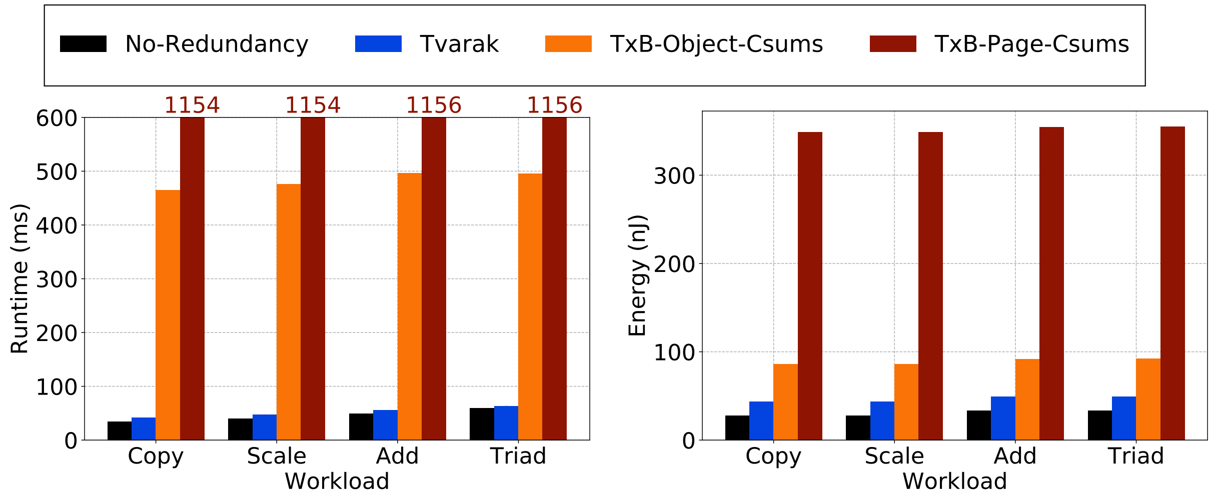
the workload and perform a total of 800,000 transactions. For N-Store, we present results from a single run with no error bars.

Fig. 4.8 shows the runtime and energy. Tvarak increases the runtime by 27% and 41% over No-Redundancy for the read-heavy and update-heavy workloads, respectively. Tvarak's overheads are higher with N-Store than with Redis or key-value structures, because N-Store uses a linked list based write-ahead log that leads to a random write access pattern for update transactions. Each update transaction allocates and writes to a linked list node. Because the linked list layout is not sequential in NVM, Tvarak incurs cache-misses for the system-redundancy information and performs more NVM accesses. The random write access pattern also affects TxB-Object-Csums and TxB-Page-Csums, with a 70%–117% and 264%–600% longer runtime than No-Redundancy, respectively. This is because TxB-Object-Csums and TxB-Page-Csums also incur misses for system-redundancy information in the L1, L2 and LLC caches and have to perform more NVM accesses for random writes.

4.2.5 Fio Microbenchmarks

Fio is a file system benchmarking tool that supports multiple access patterns [10]. We use fio's libpmem engine that accesses DAX-mapped NVM file data using load and store instructions. We use sequential and random read and write workloads with a 64B access granularity. We use 12 concurrent threads with each thread performing 32MB worth of accesses (reads or writes). Each thread accesses data from a non-overlapping 512MB region, and no cache-line is accessed twice.

Fig. 4.9 show the results for fio. As discussed above in the context of N-Store, random access patterns in the application hurt Tvarak because of poor reuse for system-redundancy cache-lines with random accesses. This trend is visible for fio as well—whereas Tvarak has



(a) Stream: Runtime

(b) Stream: Energy

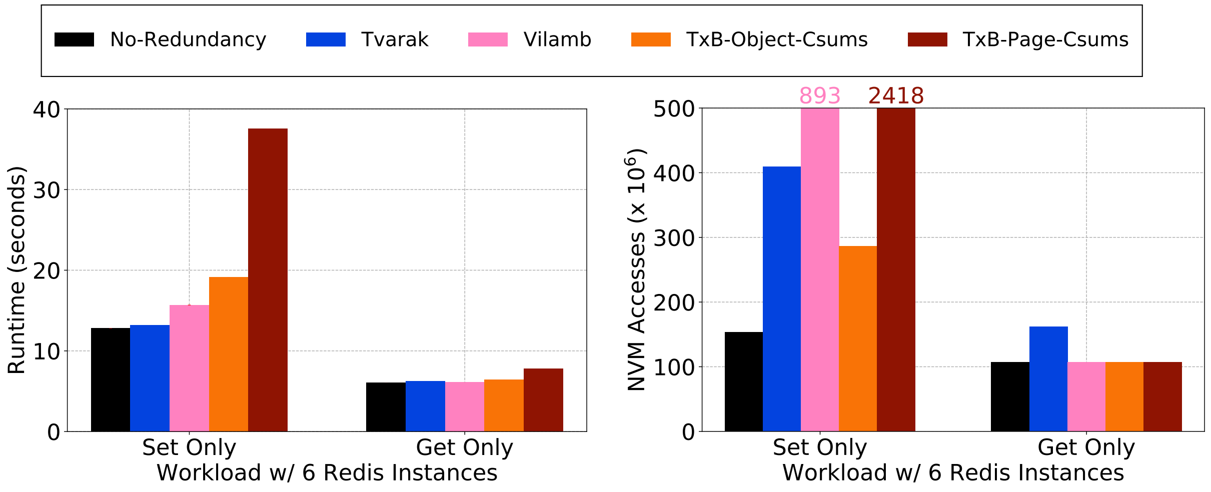
Figure 4.10: Stream Microbenchmarks: Runtime and energy for four memory intensive stream microbenchmark kernel.

essentially the same runtime as No-Redundancy for sequential accesses, Tvarak increases the runtime by 2% and 33% over No-Redundancy for random reads and writes, respectively. However, Tvarak still outperforms TxB-Object-Csums and TxB-Page-Csums for the write workloads. For read workloads, TxB-Object-Csums and TxB-Page-Csums have no impact because they do not verify application data reads. For the random write workload, Tvarak incurs a higher energy overhead than TxB-Object-Csums. This is because the energy required for additional NVM accesses that Tvarak generates exceed that required for the additional cache accesses that TxB-Object-Csums generates.

4.2.6 Stream Microbenchmarks

Stream is a memory bandwidth stress tool [91] that is part of the HPC Challenge suite [34]. Stream comprises of four sequential access kernels: (i) *Copy* data from one array to another, (ii) *Scale* elements from one array by a constant factor and write them in a second array, (iii) *Add* elements from two arrays and write them in a third array, and (iv) *Triad* which is a combination of Add and Scale: it scales the elements from one array, adds them to the corresponding elements from the second array, and stores the values in a third array. We modify stream to store and access data in persistent memory. We use 12 concurrent threads that operate on non-overlapping regions of the arrays. Each array has a size of 128MB.

Fig. 4.10 shows the results for the four kernels. The trends are similar to the preceding results. Tvarak, TxB-Object-Csums, and TxB-Page-Csums increase the runtime by 6%–21%, 700%–1200%, and 1800%–3200% over No-Redundancy, respectively. The absolute values of the overheads are higher for all the designs because the baseline No-Redundancy already saturates the NVM bandwidth, unlike the real-world applications considered above that consume the data in more complex fashions. The impact of computation complexity is evident across the four microbenchmarks: copy is the simplest kernel, followed by scale,



(a) Redis: Runtime (b) Redis: NVM Accesses

Figure 4.11: Comparison with Vilamb: Runtime and NVM accesses for Redis workloads with Vilamb in addition to No-Redundancy, Tvarak, TxB-Object-Csums, and TxB-Page-Csums.

add, and triad. Consequently, the overheads for all the designs are highest for the copy kernel and lowest for the triad kernel.

4.2.7 Comparison with Vilamb

This section compares Tvarak and Vilamb with the Redis set-only and get-only benchmarks described in Section 4.2.2. We run Vilamb with a system-redundancy update period of 1 second and scrubbing disabled. Tvarak provides synchronous system-redundancy updates and verification.

Fig. 4.11(a) presents the runtime for Redis with Vilamb in addition to the previously reported No-Redundancy, Tvarak, TxB-Object-Csums, and TxB-Page-Csums. Vilamb with a 1 second system-redundancy update period is 22% slower than No-Redundancy, in contrast to only 3% slower Tvarak and 50% slower TxB-Object-Csums. Fig. 4.11(b) presents the number of NVM accesses performed by each of the mechanisms. As expected, Vilamb has more NVM accesses than Tvarak. However, Vilamb also has more NVM accesses than TxB-Object-Csums. This is because TxB-Object-Csums uses object granular checksums and can update checksums by just reading the object. In contrast, Vilamb has to read an entire page every time it updates a checksum. Despite the increase in NVM accesses, Vilamb performs better than TxB-Object-Csums because it computes two orders of magnitude fewer checksums than TxB-Object-Csums. For the get-only workload, Vilamb has negligible overhead.

4.2.8 Impact of Tvarak's Design Choices

We break down the impact of Tvarak's design choices, namely, using DAX-CL-checksum, caching system-redundancy information, and storing data diffs in LLC. We present the

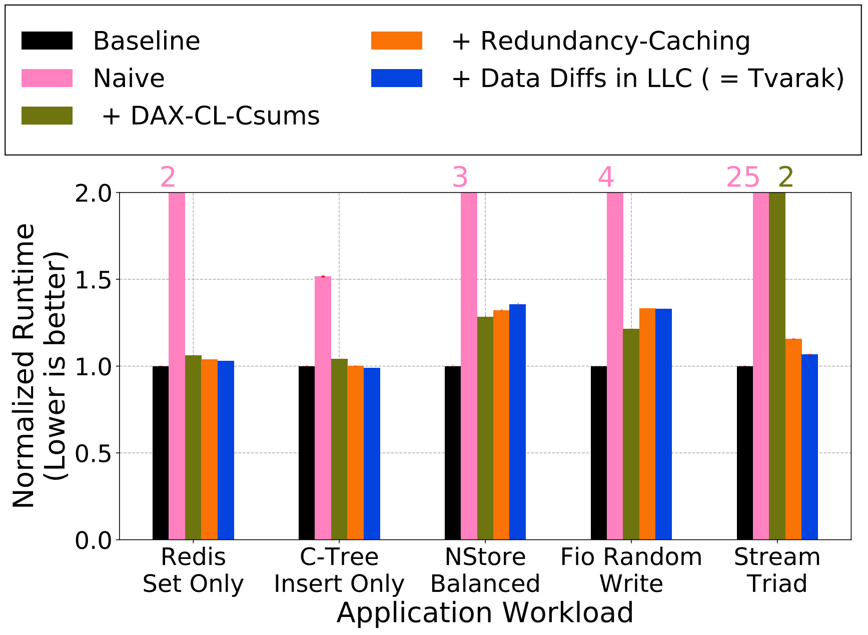


Figure 4.12: Impact of Tvarak's Design Choices: We evaluate the impact of Tvarak's design optimizations with one workload for each application. We present the results for the naive design and then add optimizations: DAX-CL-checksums, system-redundancy caching, and data diffs in LLC. With all the optimizations enabled, we get Tvarak.

results for one workload from each of the above applications: set-only workload with 6 instances for Redis, insert-only workload for C-Tree, balanced workload for N-Store, random write workload for fio, and triad kernel for stream.

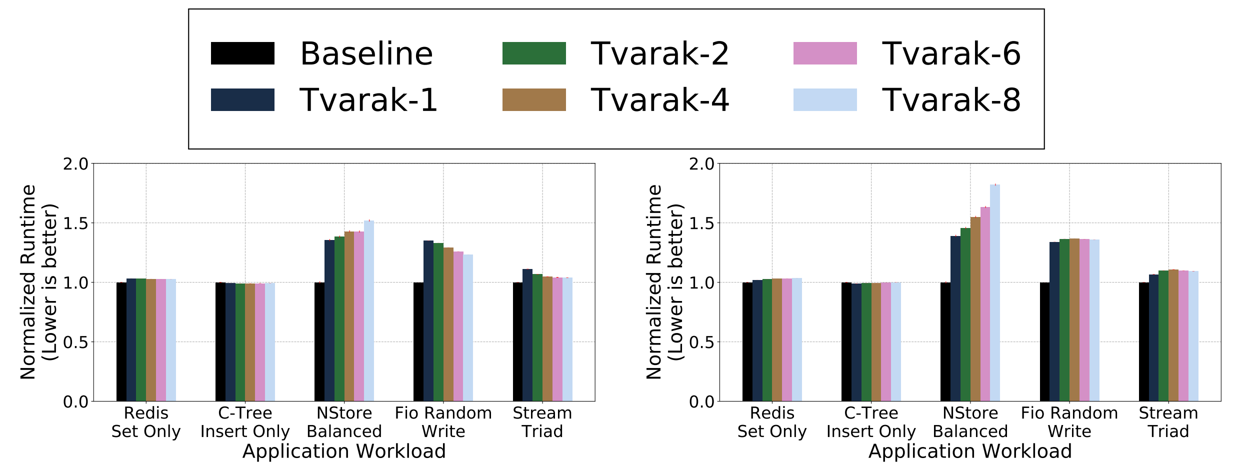
Fig. 4.12 shows the performance for the naive design, and then adds individual design elements, i.e., DAX-CL-checksums, system-redundancy caching, and storing data diffs in LLC. With all the design elements, we get the complete Tvarak design. For Redis, C-Tree and stream's triad kernel, all of Tvarak's design choices improve performance. This is the case for B-Tree, RB-Tree, other stream kernels, and fio sequential access workloads as well (not shown in the figure).

For N-Store and fio random write workload, system-redundancy caching and storing data diffs in the LLC hurt performance. This is because taking away cache space from application data creates more NVM accesses than that saved by caching the system-redundancy data and storing the data diffs in LLC for N-Store and fio random writes—their random access patterns lead to poor reuse of system-redundancy cache-lines.

This evaluation highlights the importance of right-sizing LLC partitions that Tvarak uses to cache system-redundancy information and to store data diffs. We now evaluate the application performance sensitivity to these parameters.

4.2.9 Sensitivity Analysis

We evaluate the sensitivity of Tvarak to the size of LLC partitions that it can use for caching system-redundancy information and storing data diffs. We present the results



(a) Sensitivity to number of ways for caching system-redundancy information.

(b) Sensitivity to number of ways for storing data diffs.

Figure 4.13: Impact of changing the number of LLC ways (out of 16) that Tvarak can use for caching system-redundancy data and for storing data diffs.

for one workload from each of the set of applications, namely, set-only workload with 6 instances for Redis, insert-only workload for C-Tree, balanced workload for N-Store, random write workload for fio, and triad kernel for stream.

Fig. 4.13(a) shows the impact of changing the number of LLC ways (out of 16) that Tvarak can use for caching system-redundancy information. Redis and C-Tree are largely unaffected by the system-redundancy partition size, with Redis benefitting marginally from reserving 2 ways instead of 1. Stream and fio, being synthetic memory stressing microbenchmarks, demonstrate that dedicating a larger partition for system-redundancy caching improves Tvarak's performance because of the increased cache space. N-Store is cache-sensitive and taking away the cache from application data for system-redundancy hurts its performance.

Fig. 4.13(b) shows the sensitivity of Tvarak to the number of ways reserved for storing data diffs. As with the sensitivity to system-redundancy information partition size, changing the data diff partition size has negligible impact on Redis and C-Tree. For N-Store, increasing the number of ways reserved for storing data diffs hurts performance because N-Store is cache-sensitive. Stream and fio show an interesting pattern: increasing the number of data diff ways from 1 to 4 hurts performance, but increasing it to 6 or 8 improves performance (although the performance remains worse than reserving just 1 way). This is because dedicating more ways for storing data diffs has two contradicting effects. It reduces the number of write-backs due to data diff evictions, but it also causes more write-backs because of the reduced cache space for application data. Their combined effect dictates the overall performance.

4.3 Conclusion

Tvarak efficiently maintains synchronous system-redundancy for DAX NVM storage, addressing controller and firmware imperfections expected to arise with NVM as they have with other storage technologies. As a hardware offload, managed by the storage software, Tvarak does so with minimal overhead and much more efficiently that software-only approaches.

Chapter 5

Conclusion and Future Directions

This dissertation demonstrates that direct access NVM storage can be fortitifed with redundancy mechanisms to protect against firmware-bug-induced data corruptions at low overhead by delaying the system-redundancy updates or by leveraging a hardware offload. To that end, it presents two case study systems.

First, we present the design and implementation of Vilamb, a userspace library that maintains system-redundancy asynchronously. Vilamb repurposes the dirty bits to identify pages with stale system-redundancy. Vilamb's background thread periodically updates system-redundancy for such pages. In doing so, it amortizes the overhead of system-checksum and parity computation across multiple writes to a page. Vilamb's asynchronous approach leads to 3–5× higher throughput than a state-of-the-art synchronous software-based approach, Pangolin, for multi-threaded PMDK key-value insertion workloads. Vilamb's asynchronous approach creates a trade-off between performance and time-to-coverage. For example, with a system-redundancy update period of 1 second, Redis with YCSB-A incurs a 17% throughput reduction in comparison to maintaining no system-redundancy. Increasing the delay between system-redundancy updates to 5 seconds leads to only a 9% throughput reduction. However, this increase in delay also reduces the mean time to data loss by a third. Vilamb is suitable for applications that desire performance but are willing to accept slightly reduced coverage.

Second, we present Tvarak, a software-managed hardware offload for system-redundancy maintenance. Tvarak is a hardware controller co-located with last-level cache banks. Its interpositioning in the data path allows it to identify and act upon NVM accesses and maintain system-redundancy synchronously. Tvarak implements simple techniques such as temporarily maintained cache-line granular checksums, and caching of system-redundancy to reduce NVM accesses leading to and efficient design. Tvarak is able to synchronously update and verify system-redundancy for Redis with only a 3% slowdown compared to maintaining no system-redundancy. Tvarak significantly outperforms software-based synchronous system-redundancy techniques. For example, TxB-Object-Csums, based on Pangolin, is up to 53% slower than Tvarak for the N-Store database.

5.1 Future Directions

This section describes some future research directions based on the work presented in this dissertation. All the research directions discussed below, as well as NVM research in general, would also benefit from identifying key real-world applications that would leverage direct access. Despite the excitement in the storage community, there are currently only a few commercial applications that leverage direct access to NVM (e.g., SAP HANA database [7]). As NVM storage matures and more applications emerge, future research would be able to use more realistic applications and benchmarks.

5.1.1 Automated tuning of Tvarak's LLC partition sizes

As described in Section 4.2.8, reserving even a small partition of the last-level cache for caching system-redundancy and storing data diffs can have adverse impact on some cache-sensitive applications. Automatically tuning the cache partition sizes at runtime, particularly with multiple co-running applications, is a challenging problem that warrants more research. The problem would be even more difficult with long-running applications that exhibit varying access patterns. Potential solutions could include set duelling [76], i.e., choosing between two potential choices at runtime by continuous profiling (if the search space can be a-priori reduced to a couple of choices), communicating applications' access characteristics to Tvarak via better hardware-software co-design [95], or runtime profiling and tweaking by the operating system.

5.1.2 Study of firmware-bug-induced failures in NVM storage

This dissertation is motivated by the existence of corruption inducing firmware bugs in storage devices. The presence of such bugs was uncovered and proven by large-scale studies of deployed storage systems with hard disks and solid state drives. Although there is no reason to expect that such bugs would not exist for NVM devices, a large-scale study (both in terms of number and duration) of deployed NVM storage devices would further support the motivation and can provide interesting insights. This would also enable failure injection studies to compare the recovery of various solutions (Pangolin, Vilamb, and Tvarak) based on the statistical properties of NVM device firmware bugs. It can also potentially spawn new research directions if the nature of bugs differs between NVM devices and conventional storage devices.

5.1.3 Extending Vilamb and Tvarak for cross machine DAX NVM storage replication

Mojim [112] and HotPot [89] implement cross-machine replication for DAX NVM. However, they are both interposing library based solutions and thus have the corresponding performance overhead and programming restrictions. It would be interesting to explore if Vilamb and Tvarak can provide more efficient cross-machine replication.

Appendix A

Other Related Research of the Author

Research conducted as part of this dissertation also includes **Viyojit** [48], a solution to address the challenge of battery scaling in battery-backed DRAM servers. Battery-backed DRAM servers are widely used to emulate NVM by treating the DRAM as durable storage and using the battery to flush out data from DRAM onto a truly durable medium (e.g., SSD) upon a power failure.

Provisioning battery backup for high capacity battery-backed DRAM servers is challenging. Whereas DRAM capacities have experienced excellent scaling over time, battery capacities have scaled poorly. The stunted growth of battery capacities leads to (volumetrically) large and unwieldy batteries that pose challenges in terms of their cooling cost, packaging, availability, and maintainability.

Viyojit decouples the battery capacity from the DRAM capacity by leveraging a skew common in application access patterns. Typically, applications update only a small fraction of their dataset. Our analysis of workload traces from four Microsoft datacenter applications show that for the majority of workloads, the fraction of data written in any one hour period is less than 15% of the total data.

Viyojit leverages applications' write skew to provision a smaller battery capacity that can only support writing back a fraction of the total DRAM capacity. To ensure that no data is lost upon a power failure, Viyojit bounds the number of dirty pages in DRAM based on the provisioned (smaller) battery capacity. Viyojit write-protects the battery-backed pages allocated to an application. Whenever Viyojit makes a page writable (and hence a page that it would need to write back upon a power failure), it checks the number of such pages in DRAM. If required to enforce the bound, Viyojit writes back existing dirty pages before making a new page writable. For efficiency, Viyojit tracks the update frequency of pages, keeps the frequently updated pages in a dirty state, and proactively writes back infrequently updated pages.

We implement Viyojit as a userspace library and evaluate it using Redis. We demonstrate that Viyojit can reduce the battery capacity to 11% of the original capacity with only 7–25% reduction in Redis' YCSB throughput.

Bibliography

[1] *Intel Optane/Micron 3d-XPoint Memory.* `http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html`.

[2] I. F. Adams, J. Keys, and M. P. Mesnier. Respecting the block interface – computational storage using virtual objects. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[3] *AGIGARAM Non-Volatile System.* `http://www.agigatech.com/agigaram.php`.

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 105–117, New York, NY, USA, 2015. ACM.

[5] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 336–348, New York, NY, USA, 2015. ACM.

[6] N. Amit. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 27–39, Berkeley, CA, USA, 2017. USENIX Association.

[7] M. Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M. Sharique, S. Seifert, S. Vishnoi, D. Booss, T. Peh, I. Schreter, W. Thesing, M. Wagle, and T. Willhalm. SAP HANA Adoption of Non-volatile Memory. *Proc. VLDB Endow.*, 10(12):1754–1765, Aug. 2017.

[8] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 707–722, New York, NY, USA, 2015. ACM.

[9] J. Arulraj, M. Perron, and A. Pavlo. Write-behind Logging. *Proc. VLDB Endow.*, 10(4):337–348, Nov. 2016.

[10] J. Axboe. Fio-flexible I/O tester. *URL https://github.com/axboe/fio*, 2014.

[11] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An Analysis of Data Corruption in the Storage Stack. *Trans.*

Storage, 4(3):8:1–8:28, Nov. 2008.

[12] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A Fresh Look at the Reliability of Long-term Digital Storage. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 221–234, New York, NY, USA, 2006. ACM.

[13] M. Blaze. A Cryptographic File System for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 9–16, New York, NY, USA, 1993. ACM.

[14] B. Bridge. *NVM support for C applications*, 2015. Available at `http://www.snia.org/sites/default/files/BillBridgeNVMSummit2015Slides.pdf`.

[15] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 74–83, New York, NY, USA, 1996. ACM.

[16] S. Chhabra and Y. Solihin. i-NVMM: A Secure Non-volatile Main Memory System with Incremental Encryption. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 177–188, New York, NY, USA, 2011. ACM.

[17] S. Cho and H. Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 347–357, Dec 2009.

[18] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger. Active Disk Meets Flash: A Case for Intelligent SSDs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 91–102, New York, NY, USA, 2013. ACM.

[19] L. Chua. Memristor-the missing circuit element. *Circuit Theory, IEEE Transactions on*, 18(5):507–519, Sep 1971.

[20] *Peloton Database Management Systems*. `http://pelotondb.org`.

[21] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[22] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[23] B. Debnath and and. ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory. USENIX, June 2010.

[24] T. J. Dell. A white paper on the benefits of chipkill-correct ecc for pc server main

memory. *IBM Microelectronics Division*, 11:1–23, 1997.

[25] M. Dong and H. Chen. Soft Updates Made Simple and Fast on Non-volatile Memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 719–731, Santa Clara, CA, 2017. USENIX Association.

[26] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. ACM.

[27] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[28] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 283–295, Feb 2015.

[29] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé. Increasing PCM Main Memory Lifetime. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 914–919, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[30] *Running FIO with pmem engines.* `https://pmem.io/2018/06/25/fio-tutorial.html`.

[31] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[32] S. Govindan, A. Sivasubramaniam, and B. Urgaonkar. Benefits and Limitations of Tapping into Stored Energy for Datacenters. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 341–352, New York, NY, USA, 2011. ACM.

[33] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[34] *HPC Challenge Benchmark.* `https://icl.utk.edu/hpcc/`.

[35] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 25–32, Oct 2016.

[36] Q. Hu, J. Ren, A. Badam, J. Shu, and T. Moscibroda. Log-structured Non-volatile

Main Memory. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 703–717, Berkeley, CA, USA, 2017. USENIX Association.

[37] *PMDK's libpmemobj Library.* `https://pmem.io/pmdk/libpmemobj/`.

[38] *Intel and Micron Produce Breakthrough Memory Tehcnology.* `https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/`.

[39] *PMDK: Intel Persistent Memory Development Kit.* `http://pmem.io`.

[40] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR*, abs/1903.05714, 2019.

[41] M. Ji, A. C. Veitch, and J. Wilkes. Seneca: remote mirroring done write. In *USENIX Annual Technical Conference, General Track*, ATC'03, pages 253–268, 2003.

[42] X. Jian and R. Kumar. Adaptive Reliability Chipkill Correct (ARCC). In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 270–281, Feb 2013.

[43] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky. Are Disks the Dominant Contributor for Storage Failures?: A Comprehensive Study of Storage Subsystem Failure Characteristics. *Trans. Storage*, 4(3):7:1–7:25, Nov. 2008.

[44] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP 19, page 494508, New York, NY, USA, 2019. Association for Computing Machinery.

[45] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 185–201, Berkeley, CA, USA, 2016. USENIX Association.

[46] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 29–42, Berkeley, CA, USA, 2003. USENIX Association.

[47] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani. Designing a True Direct-access File System with DevFS. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 241–255, Berkeley, CA, USA, 2018. USENIX Association.

[48] R. Kateja, A. Badam, S. Govindan, B. Sharma, and G. Ganger. Viyojit: Decoupling Battery and DRAM Capacities for Battery-Backed DRAM. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 613–626, New York, NY, USA, 2017. ACM.

[49] R. Kateja, N. Beckmann, and G. R. Ganger. Tvarak: Software-Managed Hardware

Offload for Redundancy in Direct-Access NVM Storage. In *Proceedings of the 47th Annual International Symposium on Computer Architecture*, ISCA '20. ACM, 2020.

[50] R. Kateja, A. Pavlo, and G. Ganger. Vilamb: Low Overhead Asynchronous Redundancy for Direct Access NVM. *Parallel Data Lab Technical Report CMU-PDL-20-101*. `https://www.pdl.cmu.edu/PDL-FTP/NVM/CMU-PDL-20-101.pdf`.

[51] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for Disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST'04, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[52] H. Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 691–706, New York, NY, USA, 2015. ACM.

[53] V. Kontorinis, L. E. Zhang, B. Aksanli, J. Sampson, H. Homayoun, E. Pettis, D. M. Tullsen, and T. S. Rosing. Managing Distributed Ups Energy for Effective Power Capping in Data Centers. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 488–499, Washington, DC, USA, 2012. IEEE Computer Society.

[54] H. Kumar, Y. Patel, R. Kesavan, and S. Makam. High-performance Metadata Integrity Protection in the WAFL Copy-on-write File System. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, pages 197–211, Berkeley, CA, USA, 2017. USENIX Association.

[55] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, 2017. ACM.

[56] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[57] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP 19, page 462477, New York, NY, USA, 2019. Association for Computing Machinery.

[58] *LWN: Linux and 4K disk sectors.* `https://web.archive.org/web/20131005191108/http://lwn.net/Articles/322777/`.

[59] *Supporting filesystems in persistent memory.* `https://lwn.net/Articles/610174/`.

[60] S. Liu, A. Kolli, J. Ren, and S. Khan. Crash Consistency in Encrypted Non-volatile Main Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 310–323, Feb 2018.

[61] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan. Janus: Optimizing Memory and Storage Support for Non-volatile Memory Systems. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 143–156,

New York, NY, USA, 2019. ACM.

[62] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris. Persistent Memcached: Bringing Legacy Code to Byte-addressable Persistent Memory. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'17, pages 4–4, Berkeley, CA, USA, 2017. USENIX Association.

[63] P. J. Meaney, L. A. Lastras-Montanõ, V. K. Papazova, E. Stephens, J. S. Johnson, L. C. Alves, J. A. O'Connor, and W. J. Clarke. Ibm zenterprise redundant array of independent memory subsystem. *IBM J. Res. Dev.*, 56(1):43–53, Jan. 2012.

[64] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 3–14, Dec 2007.

[65] S. Narayan, J. A. Chandy, S. Lang, P. Carns, and R. Ross. Uncovering Errors: The Cost of Detecting Silent Data Corruption. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 37–41, New York, NY, USA, 2009. ACM.

[66] D. Narayanan and O. Hodson. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 401–410, New York, NY, USA, 2012. ACM.

[67] D. S. Palasamudram, R. K. Sitaraman, B. Urgaonkar, and R. Urgaonkar. Using Batteries to Reduce the Power Costs of Internet-scale Distributed Networks. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 11:1–11:14, New York, NY, USA, 2012. ACM.

[68] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[69] R. H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[70] *Deprecating the PCOMMIT instruction.* `https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction`.

[71] *Plexistore keynote presentation at NVMW 2018.* `http://nvmw.ucsd.edu/nvmw18-program/unzip/current/nvmw2018-paper97-presentations-slides.pptx`.

[72] *Persistent Memory Emulation.* `http://pmem.io/2016/02/22/pm-emulation.html`.

[73] *Persistent Memory Storage Engine.* `https://github.com/pmem/pmse`.

[74] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-

Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 206–220, New York, NY, USA, 2005. ACM.

[75] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Data Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[76] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[77] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 14–23, New York, NY, USA, 2009. ACM.

[78] M. K. Qureshi, A. Seznec, L. A. Lastras, and M. M. Franceschini. Practical and secure PCM systems by online detection of malicious write streams. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 478–489, Feb 2011.

[79] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[80] *Redis: in-memory key value store*. `http://redis.io/`.

[81] *Redis PMEM: Redis, enhanced to use PMDK's libpmemobj*. `https://github.com/pmem/redis`.

[82] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active Disks for Large-Scale Data Processing. *Computer*, 34(6):68–74, June 2001.

[83] E. Riedel, G. A. Gibson, and C. Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[84] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST'02, pages 2–2, Berkeley, CA, USA, 2002. USENIX Association.

[85] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[86] D. Sanchez and C. Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 475–486, New York, NY, USA, 2013. ACM.

[87] B. Schroeder, S. Damouras, and P. Gill. Understanding Latent Sector Errors and How to Protect Against Them. *ACM Trans. Storage*, 6(3):9:1–9:23, Sept. 2010.

[88] N. H. Seong, D. H. Woo, and H.-H. S. Lee. Security Refresh: Prevent Malicious Wear-out and Increase Durability for Phase-change Memory with Dynamically Randomized Address Mapping. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 383–394, New York, NY, USA, 2010. ACM.

[89] Y. Shan, S.-Y. Tsai, and Y. Zhang. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 323–337, New York, NY, USA, 2017. ACM.

[90] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring data integrity in storage: Techniques and applications. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability*, StorageSS '05, pages 26–36, New York, NY, USA, 2005. ACM.

[91] *Stream Memory Bandwidth Benchmark*. `http://www.cs.virginia.edu/stream/`.

[92] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 373–386, Berkeley, CA, USA, 2015. USENIX Association.

[93] *4K Sector Disk Drives: Transitioning to the Future with Advanced Format Technologies*. `https://storage.toshiba.com/docs/services-support-documents/toshiba_4kwhitepaper.pdf`.

[94] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi. LOT-ECC: Localized and tiered reliability mechanisms for commodity memory systems. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 285–296, June 2012.

[95] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, and O. Mutlu. A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA 18, page 207220. IEEE Press, 2018.

[96] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[97] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[98] D. Wang, S. Govindan, A. Sivasubramaniam, A. Kansal, J. Liu, and B. Khessib. Underprovisioning Backup Power Infrastructure for Datacenters. In *Proceedings of the*

19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, pages 177–192, New York, NY, USA, 2014. ACM.

[99] D. Wang, C. Ren, A. Sivasubramaniam, B. Urgaonkar, and H. Fathy. Energy Storage in Datacenters: What, Where, and How Much? In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 187–198, New York, NY, USA, 2012. ACM.

[100] J. Wang, D. Park, Y. Papakonstantinou, and S. Swanson. Ssd in-storage computing for search engines. *IEEE Transactions on Computers*, pages 1–1, 2016.

[101] *Transition to Advanced Format 4K Sector Hard Drives*. `https://www.seagate.com/tech-insights/advanced-format-4k-sector-hard-drives-master-ti/`.

[102] C. P. Wright, M. C. Martino, and E. Zadok. Ncryptfs: A secure and convenient cryptographic file system. In *USENIX Annual Technical Conference, General Track*, ATC'03, pages 197–210, 2003.

[103] X. Wu and A. L. N. Reddy. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[104] J. Xu and S. Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.

[105] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, 2017. ACM.

[106] V. Young, P. J. Nair, and M. K. Qureshi. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 33–44, New York, NY, USA, 2015. ACM.

[107] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 85–98, New York, NY, USA, 2014. ACM.

[108] D. Zhang, V. Sridharan, and X. Jian. Exploring and optimizing chipkill-correct for persistent memory based on high-density nvrams. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 710–723. IEEE, 2018.

[109] L. Zhang and S. Swanson. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, 2019. USENIX Association.

[110] Y. Zhang, D. S. Myers, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Zettabyte reliability with flexible end-to-end data integrity. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, May 2013.

[111] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[112] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM.

[113] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.

[114] R. Zheng and M. C. Huang. Redundant memory array architecture for efficient selective protection. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 214–227, New York, NY, USA, 2017. ACM.

[115] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.

[116] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo. Improving the Performance and Endurance of Encrypted Non-volatile Main Memory Through Deduplicating Writes. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 442–454, Piscataway, NJ, USA, 2018. IEEE Press.