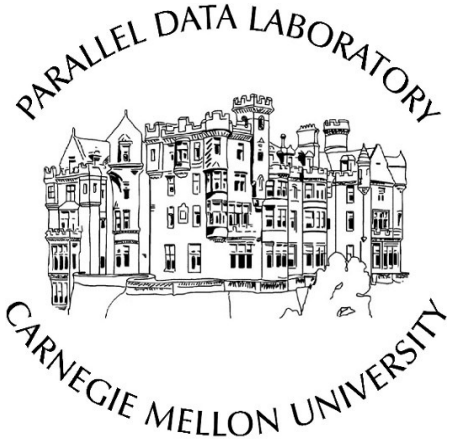


TVARAK: Software-Managed Hardware Offload for DAX NVM Storage Redundancy

Rajat Kateja, Nathan Beckmann, Greg Ganger
Carnegie Mellon University

CMU-PDL-19-105
August 2019

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

TVARAK *efficiently implements system-level redundancy for direct-access (DAX) NVM storage. Production storage systems complement device-level ECC (which covers media errors) with system-checksums and cross-device parity. This system-level redundancy enables detection of and recovery from data corruption due to device firmware bugs (e.g., reading data from the wrong physical location). Direct access to NVM penalizes software-only implementations of system-level redundancy, forcing a choice between lack of data protection or significant performance penalties. Offloading the update and verification of system-level redundancy to* TVARAK*, a hardware controller co-located with the last-level cache, enables efficient protection of data from such bugs in memory controller and NVM DIMM firmware. Simulation-based evaluation with seven data-intensive applications shows* TVARAK*'s performance and energy efficiency. For example,* TVARAK *reduces Redis set-only performance by only 3%, compared to 50% reduction for a state-of-the-art software-only approach.*

Acknowledgments: We thank the members and companies of the PDL Consortium (Alibaba Group, Amazon, Datrium, Facebook, Google, Hewlett Packard Enterprise, Hitachi, IBM Research, Intel, Micron, Microsoft Research, NetApp, Oracle, Salesforce, Samsung, Seagate, and Two Sigma) for their interest, insights, feedback, and support. Nathan Beckmann is supported by a Google Faculty Research Award.

Keywords: NVM, DAX, redundancy, hardware offload

1 Introduction

Non-volatile memory (NVM) storage improves the performance of stateful applications by offering DRAM-like performance with disk-like durability [7, 8, 16, 19, 74]. Applications that seek to leverage raw NVM performance eschew conventional file system and block interfaces in favor of direct access (DAX) to NVM. With DAX, an application maps NVM data into its address space and uses load and store instructions to access it, eliminating system software overheads from the data path [15, 19, 40, 69, 70].

Production storage systems 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 [10, 11, 31, 52, 65]. 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 corruption will go unnoticed even in the presence of device-level ECC, because that 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-checksums for detection and cross-device 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 [10, 52, 75]. 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 two. It can then trigger recovery using the cross-device parity [35, 43, 47, 83]. In this paper, we use the term *redundancy* to refer to the combination of system-checksums and cross-device parity.

Production NVM-based storage systems will need such 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, most existing NVM storage system designs provide insufficient protection. Although fault-tolerant NVM file systems [50, 75] efficiently cover data accessed through the file system interfaces, they do not cover DAX-mapped data. The Pangolin [1] library is an exception, implementing system-checksums and parity for applications that use its transactional library interface. However, software-only approaches for DAX NVM redundancy incur significant performance overhead (e.g., 50% slowdown for a Redis set-only workload, even with Pangolin's streamlined design).

This paper proposes TVARAK[1], a software-managed hardware offload that efficiently maintains redundancy for DAX NVM data. TVARAK co-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 redundancy for each DAX NVM cache-line write-back.

TVARAK's design relies on two key elements to achieve efficient 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 redundancy information. Applications' data access locality leads to

[1] TVARAK means accelerator in Hindi.

reuse of system-checksum and parity cache-lines; TVARAK leverages this reuse with a small dedicated on-controller cache and configurable LLC partitions for redundancy information.

Simulation-based evaluation with seven applications, each with multiple workloads, demonstrates TVARAK's promise of efficient DAX NVM storage redundancy. For Redis, TVARAK incurs only a 3% slowdown for maintaining redundancy with a set-only workload, in contrast to 50% slowdown with Pangolin's efficient 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-only alternatives. The efficiency benefits are seen in both application runtimes and energy.

This paper makes three primary contributions. First, it motivates the need for architectural support for DAX NVM storage redundancy, highlighting the limitations of software-only approaches. Second, it proposes TVARAK, a low-overhead, software-managed hardware offload for DAX NVM storage redundancy. It describes the challenges for efficient hardware DAX NVM redundancy and how TVARAK overcomes these challenges with straightforward, effective design. Third, it reports on extensive evaluation of TVARAK's runtime, energy, and memory access overheads for seven applications, each under multiple workloads, showing its efficiency especially in comparison to software-only alternatives.

2 Redundancy Mechanisms and NVM Storage

This section provides background and discusses related work. First, it describes conventional storage redundancy mechanisms for firmware bug resilience. Second, it discusses the need for these mechanisms in NVM storage systems, the direct-access (DAX) interface to NVM storage, and the challenges in maintaining the required redundancy with DAX. Third, it discusses related work and where TVARAK fits.

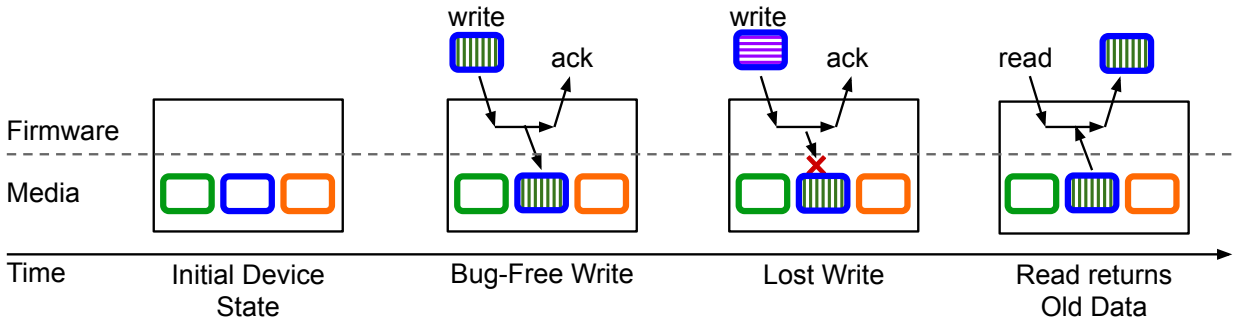
2.1 Redundancy for Firmware Bug Resilience

Production storage systems employ a variety of redundancy mechanisms to address a variety of faults [22, 29, 35, 37, 45, 47, 48, 75, 79, 81]. In this work, we focus on redundancy mechanisms used to detect and recover from firmware-bug-induced data corruption (specifically, per-page system-checksums and cross-device parity).

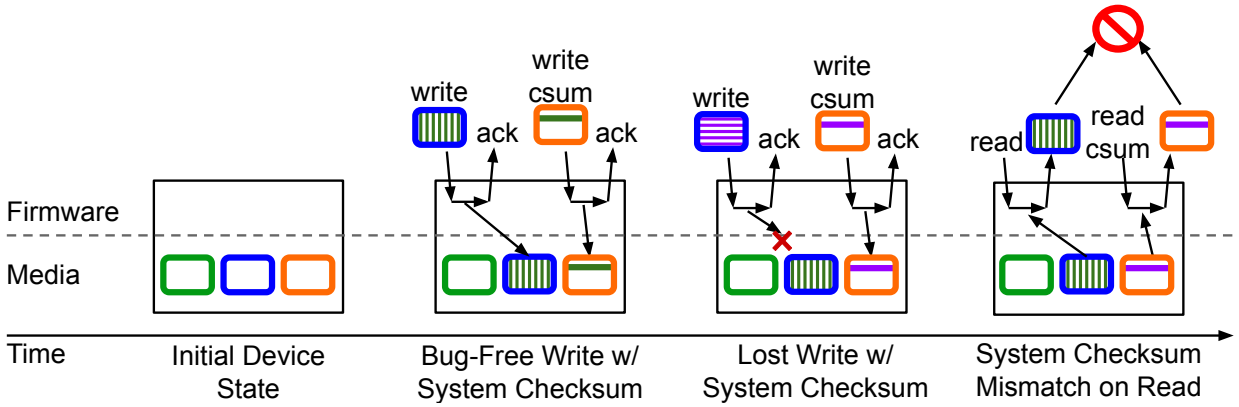
Firmware-bug-induced data corruption: Large-scale studies of deployed storage systems show that device firmware bugs sometimes lead to data loss or corruption [10, 11, 31, 52, 65]. Device firmware, like any software, is prone to bugs because of its complex responsibilities (e.g., address translation, 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 [32] and application-specific functionalities [2, 13, 59, 60, 71] 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. Figure 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. Figure 2(a) illustrates a misdirected write bug. As before, the first



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



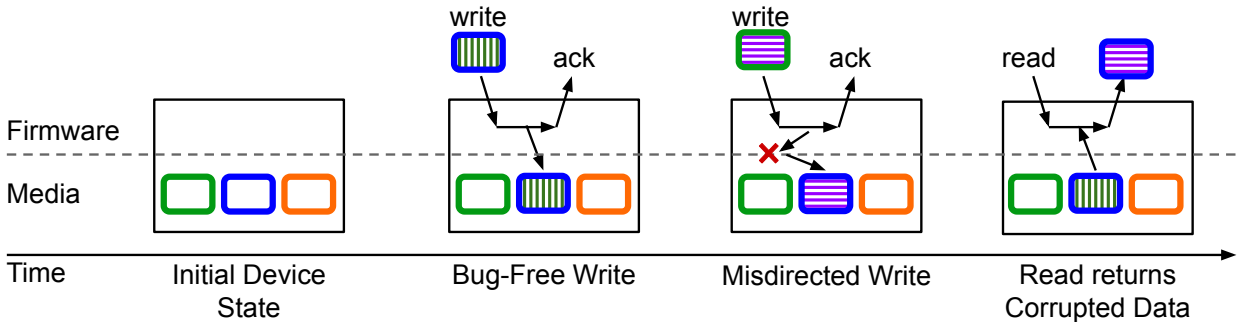
(b) The fix: having the higher-level system update and verify system-checksums when writing or reading data, in separate requests to the device, enables detection of a lost write because of mismatch between the data and the system-checksum.

Figure 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. (a) shows how the higher-level system can consume incorrect (old) data if it trusts the device to never lose an acknowledged write. (b) shows how the higher-level system can detect a lost write with system-checksums.

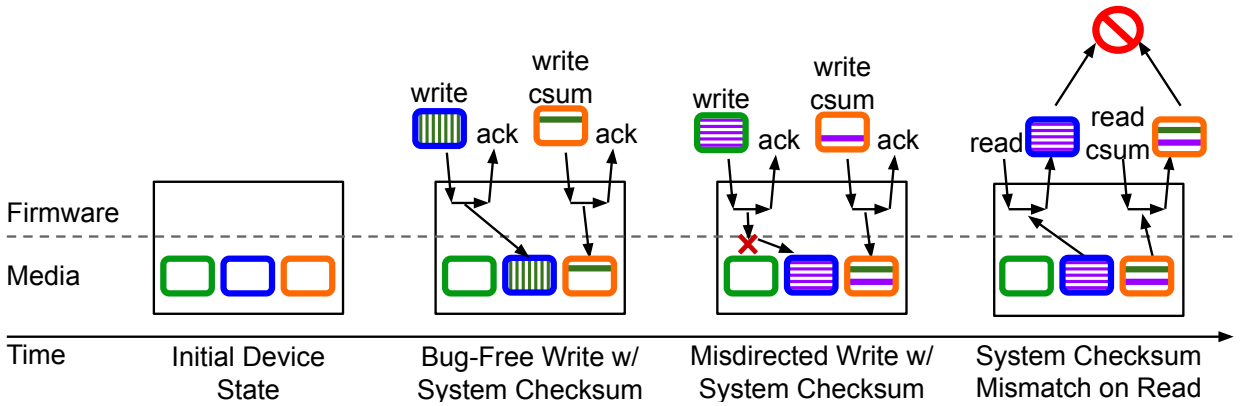
write to the block stored in the blue location is performed correctly by the firmware. For this example, the second write request shown is for the block stored in the green location. But, it encounters a misdirected write bug wherein the data is incorrectly written to the blue media location. Notice 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 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 [17, 30, 68, 77], these ECCs cannot detect firmware-bug-induced corruption [10, 52]. Device-level ECCs are stored together with the data and computed and verified inline by the same firmware during the actual media update/access. So, in the case of a lost write, the firmware loses the ECC update along with the corresponding 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.

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 blocks (each containing checksums for many blocks) separate from the data, and read and written using I/O requests separate from the corresponding



(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.



(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: Misdirected write bug example. Similar construction to Figure 1, but with the second operation being a write intended for the green location that is misdirected by the firmware to the blue location.

data I/O requests [22, 37, 45, 75, 79, 81]. 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 probability of a bug affecting both in a consistent fashion (e.g., losing both or misdirecting both to another corresponding data and system-checksum pair) is even lower.

Figure 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.

Figure 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.

Cross-device parity for recovery: To recover from a detected page corruption, storage systems store parity pages [29,35,43,47,48,83]. Although parity across arbitrarily selected pages suffice for recovery from firmware-bug-induced corruption, storage systems often implement cross-device parity that enable recovery

NVM Storage Redundancy Design	Checksum Granularity	Checksum/Parity Update for DAX data	Checksum Verification for DAX data	Performance Overhead
Nova-Fortis [75], Plexistore [50]	Page	No updates	No verification	None
Mojim [80], HotPot [64]	Page[2]	On application data flush	Background scrubbing	Very High
Pangolin [1]	Object	On application data flush	On NVM to DRAM copy	High
Anon [34]	Page	Periodically	Background scrubbing	Configurable
TVARAK	Page	On LLC to NVM write	On NVM to LLC read	Low

Table 1: Trade-offs among TVARAK and previous DAX NVM storage redundancy designs.

from device failures as well.

2.2 NVM Storage Redundancy and Direct Access (DAX)

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 [3, 14, 26, 38, 56]. NVM devices have orders of magnitude lower latency and higher bandwidth than conventional storage devices, thereby improving stateful applications' performance [7, 8, 16, 19, 36, 73, 74].

Need for firmware-bug resilience in NVM storage: NVM storage systems will be prone to firmware-bug-induced data corruption and require corresponding redundancy mechanisms, like 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 [53, 55, 56, 63]. Looking forward, its complexity will only increase as more NVM-specific functionality is embedded into the firmware (e.g., reducing NVM writes and wear [12, 21, 41, 76, 84]) and as the push towards near-data computation [2, 5, 6, 13, 20, 25, 32, 59, 60, 71, 78] continues.

Direct access NVM storage redundancy challenges: Direct-access (DAX) interface to NVM storage exposes raw NVM performance to applications [1, 19, 34, 40, 42, 51, 58, 64, 69, 75, 80]. DAX-enabled file systems map NVM-resident files directly into application address spaces; such direct mapping is possible because of NVM's DRAM-like access characteristics. DAX enables applications to access persistent data with load and store instructions, eliminating system software overheads from the data path.

These characteristics, however, pose challenges for maintaining firmware-bug resiliency mechanisms [34]. First, the lack of interposed system software in the data path makes it difficult to efficiently identify data reads and writes that should trigger a system-checksum verification and system-checksum/parity updates, respectively. Second, updating and verifying system-checksums for DAX data incurs high overhead because of the mismatch between DAX's fine-grained accesses and the typically large blocks (e.g., 4KB pages) over which checksums are computed for space efficiency.

2.3 Related Work on DAX NVM Storage 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 1 summarizes these trade-offs. Two recent fault-tolerant NVM file systems, Nova-Fortis [75] and Plexistore [50], update and check redundancy during explicit FS calls but do not update or verify redundancy while data is DAX mapped. Interposing library-based solutions, such as Mojim [80], HotPot [64], and Pangolin [1], can protect DAX-mapped data if applications use the given library's transactional interface for all data accesses and updates. But, software-based redundancy updates on every data update incur large performance over-

head. Mojim [80] and HotPot [64] would incur very high overhead because of DAX's fine-grained writes [2]. Pangolin [1] reduces such overhead by eschewing per-page checksums in favor of per-object checksums, accepting higher space overhead instead, but still incurs performance overheads do to redundancy updates/verifications in software. Anon [34] reduces the performance overhead, potentially arbitrarily, by delaying and batching the per-page checksum updates. In doing so, however, Anon reduces the coverage guarantees by introducing windows of vulnerability wherein data can get corrupted silently.

Most existing redundant NVM storage system designs do not verify DAX application data reads with the corresponding checksum. As shown in the fourth column of Table 1, some do no verification while data is DAX-mapped, while others' designs would only accommodate verifying checksums as part of background scrubbing. Pangolin does verify the checksums when it reads an object into a DRAM buffer, providing significantly tighter verifications.

The remainder of this paper describes and evaluates TVARAK, a software-managed hardware controller that provides in-line redundancy maintenance at low overheads and without programming restrictions based on required use of a given library's interface. TVARAK updates the redundancy for every write to the NVM device and verifies system-checksums for every read from the NVM device.

3 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 redundancy controller design, and improve its design to reduce its overheads, leading to TVARAK's design. We end with TVARAK's architecture, area overheads, and walk through examples.

3.1 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, (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 Figure 3.

TVARAK's 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 [22, 29, 35, 48], snapshots for user errors [23, 61, 75, 81], and inline sanity checks for file system bugs [37].

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 ECCs [17, 30, 68] 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.

[2] The original Mojim and HotPot designs do not include checksums, only replication, but their designs extend naturally to include per-page checksums.

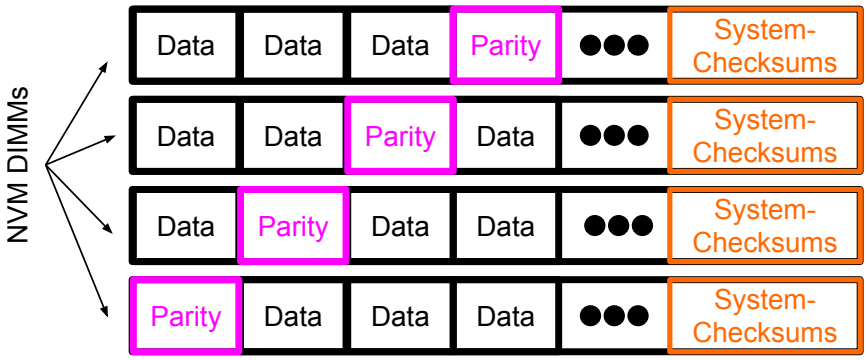


Figure 3: TVARAK coordinates with the file system to maintain per-page system-checksums and cross-DIMM parity akin to RAID-5 with page striping.

3.2 Basic Redundancy Controller Design

Figure 4 illustrates a basic redundancy controller design that satisfies the requirements for detecting firmware-bug induced corruptions, as described in Section 2.1. We refer to this basic design as NAIVE, and will improve NAIVE's design to build up to TVARAK. NAIVE resides above the device firmware in the data path (with the LLC bank controllers). The file system informs NAIVE 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, NAIVE performs an address range matching. NAIVE does not do anything for cache-lines that do not belong to a DAX-mapped regions, as illustrated in the leftmost read/write access in Figure 4. The file system continues to maintain the redundancy for such data [50, 75].

For DAX-mapped cache-lines, NAIVE updates and verifies redundancy using separate accesses from the corresponding data. The request in the center of Figure 4 shows a DAX cache-line read. To verify the read, NAIVE 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 Figure 4 shows a cache-line write. NAIVE 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[3]. It then writes the new data, new system-checksum, and the new parity to NVM. NAIVE's cross-DIMM parity design and the use of data diffs to update parity is similar to recently proposed RAIM-5b [83].

NAIVE, and consequently TVARAK, assume that the storage servers are equipped with backup power to flush CPU caches in case of a power failure. The backup power guarantees that NAIVE and TVARAK can complete the system-checksum and parity writes corresponding to a data write in case of a power failure, even if they cache this information, as we will describe later. This 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 [4, 18, 23, 33, 46, 49, 82]. Backup power also eliminates the need for durability-induced cache-line flushes and improves performance [46, 82]. We extend this assumption, and the corresponding performance benefits, to the all the designs we compare TVARAK to in Section 4.

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

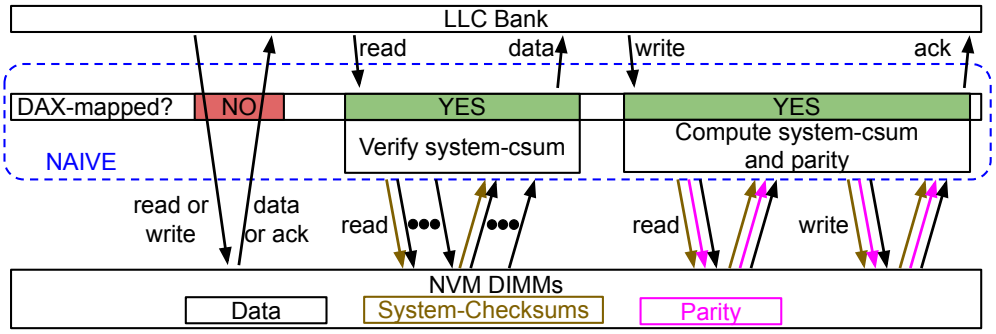


Figure 4: Basic Design: NAIVE operates only on DAX-mapped data. For DAX-mapped cache-line reads, NAIVE reads the entire page to compute the checksum, reads the system-checksum, and verifies that the two match. For cache-line writes, NAIVE reads the old data, system-checksum, and parity, computes the data diff, uses that to compute the new system-checksum and parity, and writes them back to NVM.

3.3 Efficient Checksum Verification

Verifying system-checksums in NAIVE incurs a high overhead because it has to read the entire page, as shown in Figure 4. For typical granularities of 4KB checksum pages and and 64B cache lines, NAIVE reads 65× more cache lines (64 cache-line in a page and one for the checksum). 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× more space than per-page checksums. Indeed, the trend in storage system designs is to move towards larger, rather than smaller, checksum granularities [39, 67, 72].

We introduce *DAX-CL-checksums* to reconcile the performance overhead of page-granular checksum verification with the space overhead of cache-line checksums. Adding DAX-CL-checksums to NAIVE results in the EV (Efficient Verification) design shown in Figure 5. As the name suggests, DAX-CL-checksums are cache-line granular checksums that EV maintains only when data is DAX-mapped. The read request in the middle of Figure 5 illustrates that using DAX-CL-checksums reduces the read amplification to only 2× from 65× for NAIVE—EV only needs to read the DAX-CL-checksum in addition to the data to verify the read. The additional space for DAX-CL-checksums is required only for the fraction of NVM that is DAX-mapped, in contrast to maintaining cache-line-granular or object-granular checksums for all NVM data at all times [1].

EV 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, EV updates the corresponding DAX-CL-checksum as well, using a similar process as that for system-checksums and parity (rightmost request in Figure 5).

Managing DAX-CL-checksums is simple because EV 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 file system and EV can afford to lose DAX-CL-checksums in case of a failure. When the file system DAX-maps a file, EV 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.

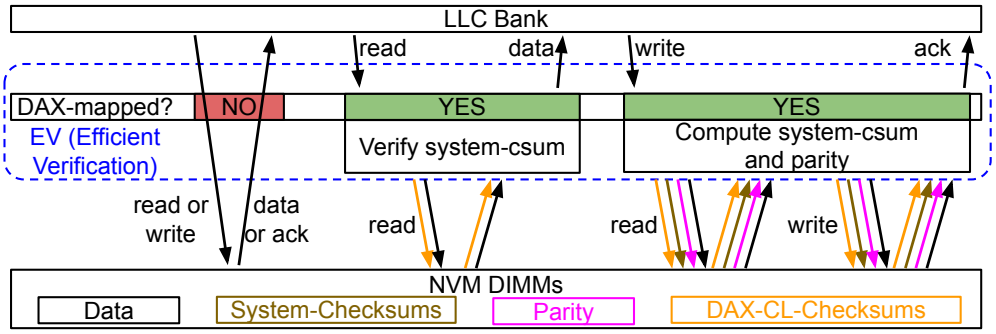


Figure 5: Efficient Checksum Verification: DAX-CL-checksums eliminate the need to read the entire page for DAX cache-line read verification. Instead, EV only reads the cache-line and its corresponding DAX-CL-checksum.

3.4 Efficient Checksum and Parity Updates

The rightmost request in Figure 5 shows that EV incurs 4 extra NVM reads and writes for each cache-line write to update the redundancy. To reduce these NVM accesses, we note that 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 reuse of parity cache-lines because they belong to the same RAID stripe.

Figure 6 shows EVU (Efficient Verification and Updates) that, in addition to EV, caches the redundancy data, i.e., system-checksums, DAX-CL-checksums, and parity, in a small on-controller cache. EVU does not cache the corresponding NVM data because the LLC already does that. EVU also uses a partition of the LLC to increase its cache space for redundancy information (not shown in the figure). Using a reserved LLC partition for caching redundancy information limits the interference with application data. EVU can insert up to 3 redundancy cache-lines (checksum, DAX-CL-checksum, and parity) per data cache-line write-back. If EVU were to share the entire LLC for caching redundancy information, each of these redundancy cache-lines could force out application data. Reserving a partition for redundancy information eliminates this possibility because EVU can only evict a redundancy cache-line when inserting a new one.

EVU also eliminates the need 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, EVU uses it to compute the data diff and stores this diff in a LLC partition. This enables EVU to directly use this data diff upon a LLC cache-line write back (shown as maroon arrows from EVU to LLC in the rightmost request in Figure 6). Upon an eviction from the LLC data diff partition (e.g., to insert a new data diff), EVU 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 EVU to read the old data either, while allowing for reuse of the data in the LLC.

EVU's LLC partitions (for caching redundancy and storing data diffs) are completely decoupled from the application data partitions. The cache controllers do not lookup application data in EVU's partitions, and EVU does not look up redundancy or data diff cache-lines in application data partitions.

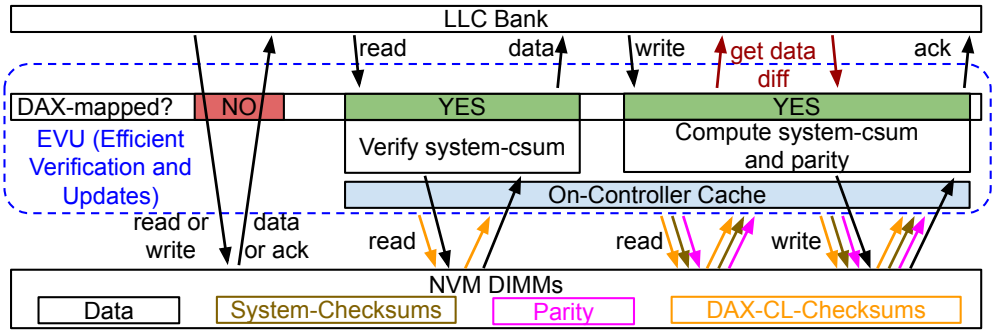


Figure 6: Efficient Checksum and Parity Updates: EVU caches redundancy cache-line in an on-controller cache and a LLC partition (not shown). EVU also uses a LLC partition to store data diffs, eliminating the need to read the old data from NVM upon cache-line write-backs.

3.5 Putting it all together with TVARAK

Figure 7 shows TVARAK's components, which are based on EVU's design. One TVARAK controller co-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 for redundancy data and uses LLC way-partitions for caching redundancy data and storing data diffs. The controllers use MESI coherence protocol for sharing the redundancy cache-lines between their private caches.

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 area is only 0.2% of the LLC. TVARAK's design of caching 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).

Life of DAX-mapped cache-lines 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 redundancy partition. If it misses in the LLC partition as well, TVARAK reads the DAX-CL-checksum from NVM and caches it. TVARAK read 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 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 redundancy information in the on-controller cache, and writes-back the data cache-line to NVM. TVARAK can safely cache the updated redundancy information because it assumes that servers are equipped with backup power to flush caches to persistence in case of a power failure (Section 3.2).

TVARAK fills an important gap in NVM storage redundancy with simple architectural changes. We believe that TVARAK can be easily integrated in storage server chips, specially because integrating NVM devices into servers already requires changing the on-chip memory controller to support the new DDR-T protocol [28].

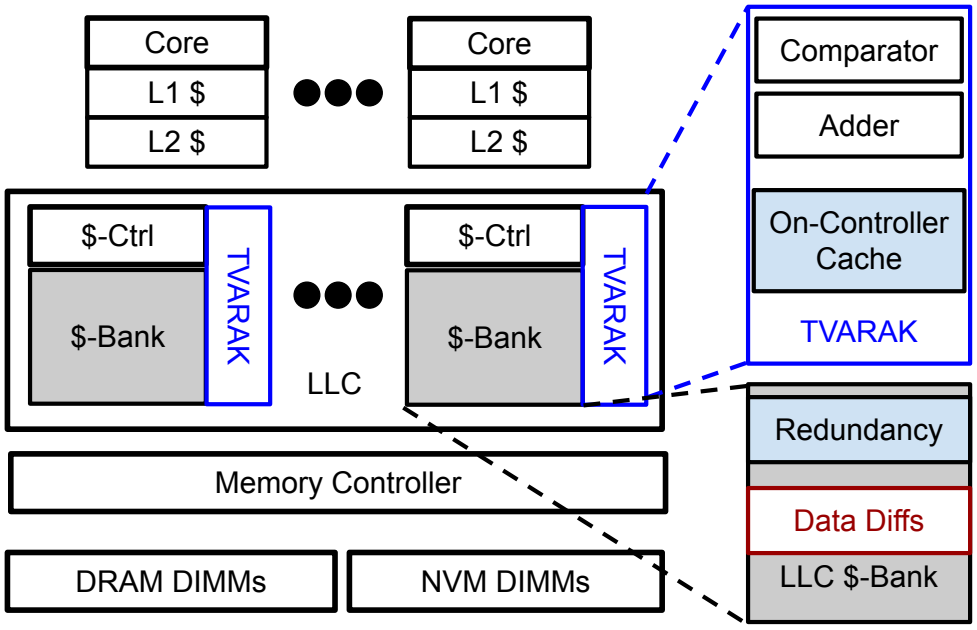


Figure 7: TVARAK is co-resides with the LLC bank controllers. It includes comparators to identify cache-line that belong to DAX-mapped pages and adders to compute checksums and parity. It includes a small on-controller redundancy cache that is backed by a LLC partition. TVARAK also stores the data diffs to compute checksums and parity.

4 Evaluation

We evaluate TVARAK with 7 applications and with multiple workloads for each application. Table 2 describes our applications and their workloads. Redis [58], Intel PMDK's [27] tree-based key-value stores (C-Tree, B-Tree, and RB-Tree), and N-Store [7] are NVM applications with complex access patterns. We also use fio [9] to generate synthetic sequential and random access patterns, and stream [66] for sequential access memory-bandwidth intensive microbenchmarks.

We compare TVARAK with three alternatives: *Baseline*, *TxB-Object-Csums*, and *TxB-Page-Csums*. Baseline implements no redundancy mechanisms. TxB-Object-Csums and TxB-Page-Csums are software-only redundancy approaches; TxB-Object-Csums is based on Pangolin [1] and TxB-Page-Csums is based on Mojim [80] and HotPot [64]. 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; it computes the new system-checksum from the new data directly. However, TxB-Object-Csums has higher space overhead because of object-granular checksums. Neither TxB-Page-Csums nor our TxB-Object-Csums verify the data read by an application with the corresponding system-checksum; thus, our TxB-Object-Csums should overpredict performance for Pangolin's approach (which includes read verification). 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 3.2, we assume battery-backed CPU caches and none of the designs flush cache-lines for durability.

Methodology: We use zsim [62] to simulate a system similar to Intel Westmere processors [62]. Table 3 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 sim-

Redis	Set-only and get-only with 1–6 parallel instances
C-Tree	Insert-only, update-only, 50:50 updates:reads, and read-only with 12 parallel instances
B-Tree	Insert-only, update-only, 50:50 updates:reads, and read-only with 12 parallel instances
RB-Tree	Insert-only, update-only, 50:50 updates:reads, and read-only with 12 parallel instances
Fio	Sequential and random reads and writes with 12 threads
Stream	4 memory bandwidth intensive kernels with 12 threads

Table 2: Applications and their workloads.

Cores	12 cores, x86-64 ISA, 2.27 GHz, Westmere-like OOO [62]
L1-D caches	32KB, 8-way set-associative, 4 cycle latency, LRU replacement, 15/33 pJ per hit/miss [44]
L1-I caches	32KB, 4-way set-associative, 3 cycle latency, LRU replacement, 15/33 pJ per hit/miss [44]
L2 caches	256KB, 8-way set-associative, 7 cycle latency, LRU replacement, 46/94 pJ per hit/miss [44]
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 [44]
DRAM	6 DDR DIMMs, 15ns reads/writes
NVM	4 DDR DIMMs, 60ns reads, 150ns writes [38] 1.6/9 nJ per read/write [38]
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 3: Simulation Parameters

ulated 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. [38] (60/150 ns read/write latency, 1.6/9 nJ per read/write). We evaluate the impact of changing the number of NVM DIMMs and the underlying NVM technology (and the associated performance characteristics) in Section 4.8. We use a fixed-work methodology and perform the same amount of application work for each design: baseline, 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.

4.1 Key Evaluation Takeaways

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

- TVARAK provides efficient redundancy updates for application data writes, e.g., with only 1.5% overhead over baseline that provides no redundancy for a insert-only workload with tree-based key-value

stores (C-Tree, B-Tree, RB-Tree).

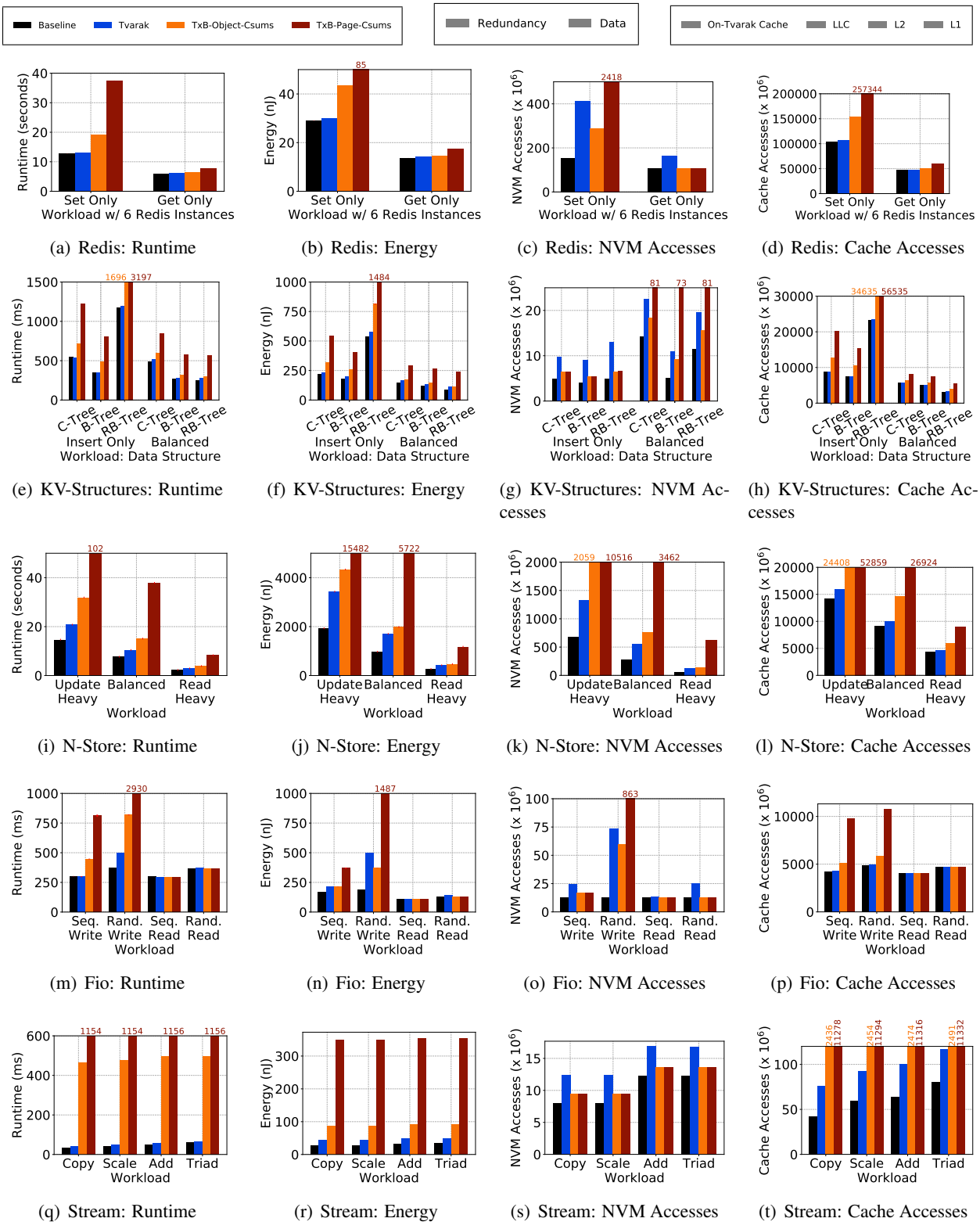
- TVARAK verifies all application data reads, unlike most existing solutions, and does so efficiently. For example, in comparison to baseline that does not verify any reads, TVARAK slows down Redis get-only workload by only 3%.
- TVARAK benefits from application data access locality because that leads to better cache usage for 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 baseline.
- TVARAK outperforms existing software-only 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 Redis

Redis is a widely used single-threaded in-memory key-value store that uses a hashtable as its primary data structure [57]. We modify Redis (v3.1) to use a persistent memory heap using Intel PMDK's libpmemobj library [27], building upon an open-source implementation [58]. 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 or presentation; the trends are the same for 1–6 Redis instances that we evaluated.

Figure 8(a) shows the runtime for Redis set-only and get-only workloads. In comparison to baseline, that maintains 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 the baseline for the set-only workload. For get-only workloads, TxB-Object-Csums and TxB-Page-Csums increase the runtime for by a maximum of 5% and 28% over baseline, respectively. This increase for TxB-Object-Csums and TxB-Page-Csums, despite them not verifying any application data reads, is because Redis use libpmemobj transactions for get requests as well; 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.

Figures 8(b) to 8(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 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 200× more cache accesses than baseline), 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.



(a) Redis: Runtime (b) Redis: Energy (c) Redis: NVM Accesses (d) Redis: Cache Accesses

(e) KV-Structures: Runtime (f) KV-Structures: Energy (g) KV-Structures: NVM Accesses (h) KV-Structures: Cache Accesses

(i) N-Store: Runtime (j) N-Store: Energy (k) N-Store: NVM Accesses (l) N-Store: Cache Accesses

(m) Fio: Runtime (n) Fio: Energy (o) Fio: NVM Accesses (p) Fio: Cache Accesses

(q) Stream: Runtime (r) Stream: Energy (s) Stream: NVM Accesses (t) Stream: Cache Accesses

Figure 8: Runtime, energy, and NVM and cache accesses for various Redis (Figures 8(a) to 8(d)), tree-based key-value data structures (Figures 8(e) to 8(h)), N-Store (Figures 8(i) to 8(l)), Fio (Figures 8(m) to 8(p)), and Stream (Figures 8(q) to 8(t)) workloads. We divide NVM accesses into data and redundancy information accesses, and cache accesses into L1, L2, LLC, and on-TVARAK cache.

4.3 Key-value Data Structures

We use three persistent memory key-value data structures, namely C-Tree, B-Tree, and RB-Tree, from Intel PMDK [27]. 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.

Figures 8(e) to 8(h) show the runtime, energy, and NVM and cache accesses 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 the baseline while updating the redundancy for all inserted tuples. In contrast, TxB-Object-Csums and TxB-Page-Csums increase the runtime by 43% and 171% over the baseline, respectively. For the balanced workload, TVARAK updates the redundancy for tuple updates and also verifies tuple reads with system-checksums with only 5% increase in runtime over the baseline for C-Tree and B-Tree. TxB-Object-Csums incurs a 20% increase in runtime over baseline for just updating the redundancy upon tuple updates; TxB-Page-Csums performs even worse.

4.4 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) [7]. We use 4 client threads to drive the workload and perform a total of 800000 transactions. For N-Store, we present results from a single run with no error bars.

Figures 8(i) to 8(l) show runtime and energy, and NVM and cache accesses. TVARAK increases the runtime by 27% and 41% over the baseline 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 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 baseline, respectively. This is because the TxB-Object-Csums and TxB-Page-Csums also incur misses for for redundancy information in the L1, L2 and LLC caches and have to perform more NVM accesses for random writes.

4.5 Fio Benchmarks

Fio is a file system benchmarking tool that supports multiple access patterns [9]. 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.

Figures 8(m) to 8(p) show the results for fio. As already discussed above in the context of N-Store, random access pattern in the application hurt TVARAK because of poor reuse for redundancy cache-lines with random accesses. This trend is visible for fio as well—whereas TVARAK has essentially the same runtime as baseline for sequential accesses, TVARAK increases the runtime by 2% and 33% over baseline 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.

4.6 Stream Benchmarks

Stream is a memory bandwidth stress tool [66] that is part of the HPC Challenge suite [24]. 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.

Figures 8(q) to 8(t) show 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 the baseline, respectively. The absolute value of the overheads are higher for all the designs because the baseline 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 clear even across the four microbenchmarks: copy is the simplest kernel, followed by scale, add, and triad. Consequently, the overheads for all the designs are highest for the copy kernel and lowest for the triad kernel.

4.7 Impact of TVARAK's Design Choices

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

Figure 9 shows the performance for the naive design, and then adds individual design elements, i.e., DAX-CL-checksums, 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, 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 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 redundancy cache-lines.

This evaluation highlights the importance of choosing the LLC partition space that TVARAK uses to cache redundancy information or to store data diffs. We leave dynamically adapting the partition sizes based on the workload characteristics for future work. The partition sizes can be adapted either by TVARAK using set duelling [54], or by the OS by application profiling.

4.8 Sensitivity Analysis

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

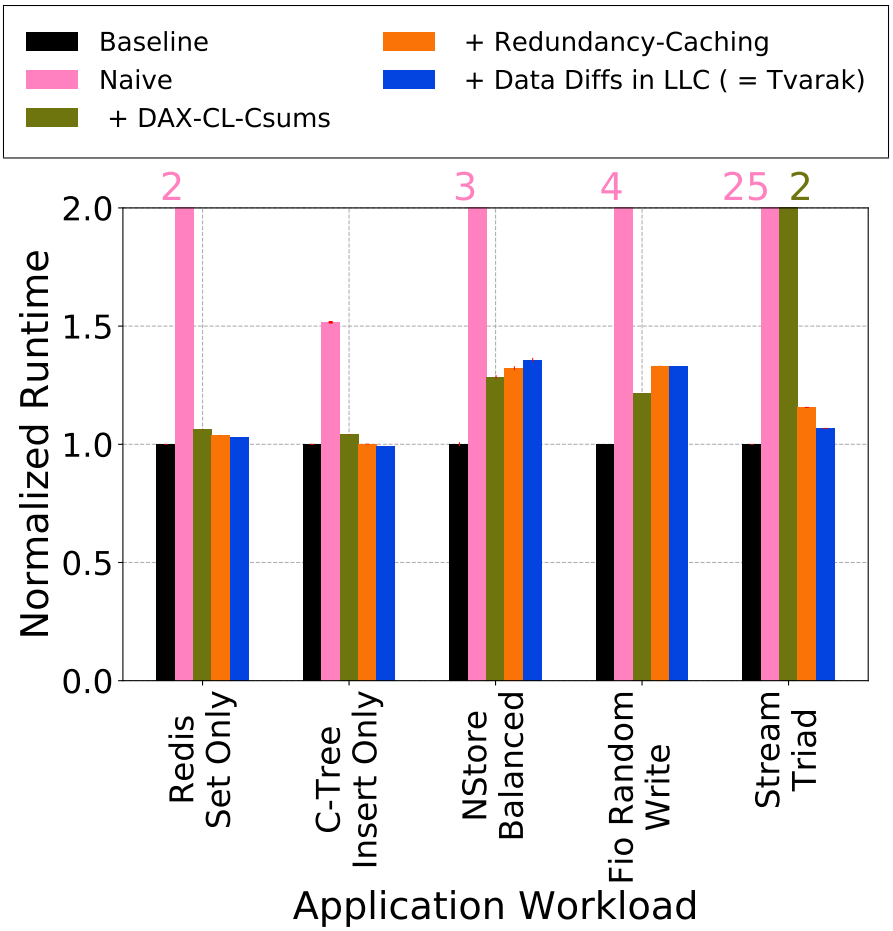
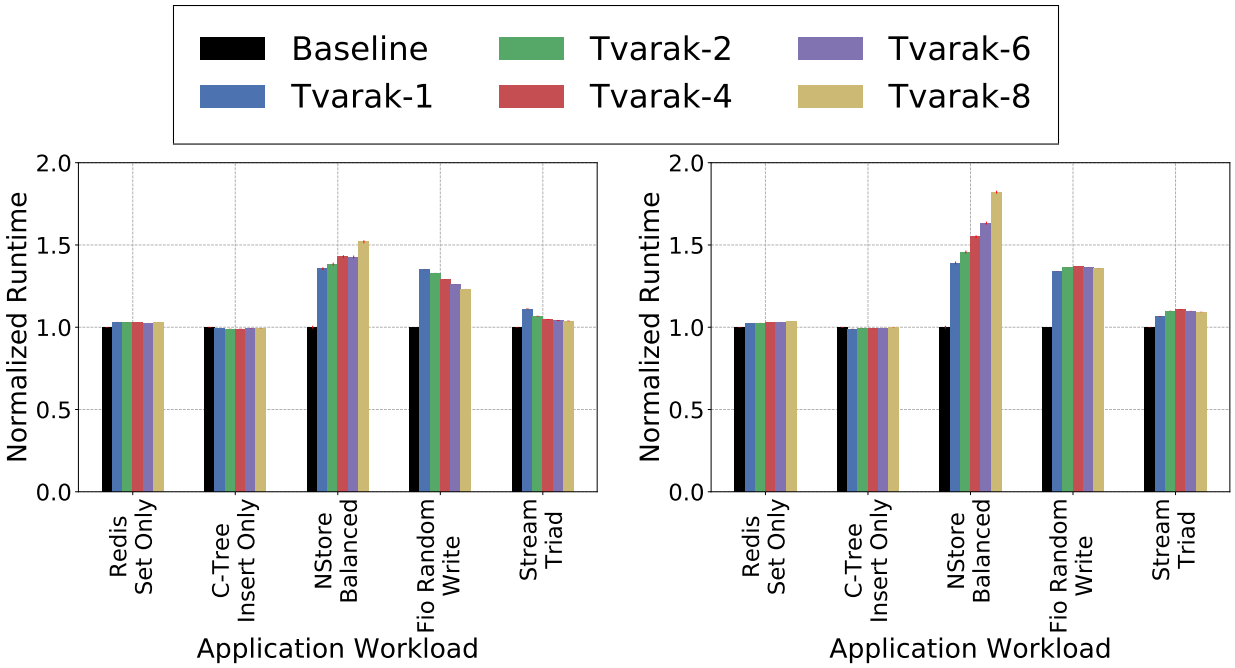


Figure 9: Impact of TVARAK's Design Choices: We evaluate the impact of TVARAK's design optimizations with one workload for each application. We present the results for the naive design and then add optimizations, DAX-CL-checksums, redundancy caching, and data diffs in LLC. With all the optimizations enabled, we get TVARAK.

Figure 10(a) shows the impact of changing the number of LLC ways (out of 16) that TVARAK can use for caching redundancy information. Redis and C-Tree are largely unaffected by the 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 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 redundancy hurts its performance.

Figure 10(b) shows the sensitivity of TVARAK to the number of ways reserved for storing data diffs. As with the sensitivity to 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 effect. 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.

We also evaluate the impact of increasing the number of NVM DIMMs and changing the underlying NVM technology on baseline, TVARAK, TxB-Object-Csums, and TxB-Page-Csums. The relative perfor-



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

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

Figure 10: Impact of changing the number of LLC ways (out of 16) that TVARAK can use for caching redundancy data and for storing data diffs.

mance trends stay the same with both of these changes; we do not show the results here for brevity. As an example, even with 8 NVM DIMMs or with improved NVM performance by considering battery-backed DRAM as NVM, TVARAK continues to outperform TxB-Object-Csums and TxB-Page-Csums by orders of magnitude for the stream microbenchmarks.

5 Conclusion

TVARAK efficiently maintains system-checksums and cross-device parity 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. Since system-level redundancy is expected from production storage, TVARAK is an important step towards the use of DAX NVM as primary storage.

References

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

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

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

[5] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung 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.

[6] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung 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.

[7] Joy Arulraj, Andrew Pavlo, and Subramanya 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.

[8] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind Logging. *Proc. VLDB Endow.*, 10(4):337–348, November 2016.

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

[10] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An Analysis of Data Corruption in the Storage Stack. *Trans. Storage*, 4(3):8:1–8:28, November 2008.

[11] Mary Baker, Mehul Shah, David S. H. Rosenthal, Mema Roussopoulos, Petros Maniatis, TJ Giuli, and Prashanth 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.

[12] 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.

[13] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory 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.

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

[15] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven 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.

[16] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick 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.

[17] Timothy J Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory. *IBM Microelectronics Division*, 11:1–23, 1997.

[18] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel 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.

[19] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff 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.

[20] 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.

[21] Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel 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.

[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak 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.

[23] Dave Hitz, James Lau, and Michael 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.

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

[25] 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.

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

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

[28] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR*, abs/1903.05714, 2019.

[29] Minwen Ji, Alistair C Veitch, and John Wilkes. Seneca: remote mirroring done write. In *USENIX Annual Technical Conference, General Track*, ATC'03, pages 253–268, 2003.

[30] 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.

[31] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady 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, November 2008.

[32] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath 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.

[33] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg 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.

[34] Rajat Kateja, Andy Pavlo, and Greg Ganger. Lazy redundancy for nvm storage: Handing the performance-reliability tradeoff to applications. *Parallel Data Lab Technical Report CMU-PDL-19-101*. https://www.pdl.cmu.edu/PDL-FTP/NVM/CMU-PDL-19-101.pdf.

[35] Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John 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.

[36] Hideaki 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.

[37] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith 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.

[38] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug 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.

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

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

[41] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira 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.

[42] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim 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.

[43] 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, January 2012.

[44] 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.

[45] Sumit Narayan, John A. Chandy, Samuel Lang, Philip Carns, and Robert 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.

[46] Dushyanth Narayanan and Orion 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.

[47] David A. Patterson, Garth Gibson, and Randy 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.

[48] R. Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane 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.

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

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

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

[52] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi 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.

[53] 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.

[54] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel 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.

[55] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent 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.

[56] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude 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.

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

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

[59] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active Disks for Large-Scale Data Processing. *Computer*, 34(6):68–74, June 2001.

[60] Erik Riedel, Garth A. Gibson, and Christos 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.

[61] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[62] Daniel Sanchez and Christos 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.

[63] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin 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.

[64] Yizhou Shan, Shin-Yeh Tsai, and Yiying 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.

[65] Gopalan Sivathanu, Charles P. Wright, and Erez 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.

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

[67] *4K Sector Disk Drives: Transitioning to the Future with Advanced Format Technologies*. https://storage.toshiba.com/docs/services-support-documents/toshiba_4kwhitepaper.pdf.

[68] 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.

[69] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael 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.

[70] Haris Volos, Andres Jaan Tack, and Michael 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.

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

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

[73] Xiaojian Wu and A. L. Narasimha 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.

[74] Jian Xu and Steven 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.

[75] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy 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.

[76] Vinson Young, Prashant J. Nair, and Moinuddin 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.

[77] Da Zhang, Vilas Sridharan, and Xun 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.

[78] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael 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.

[79] 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.

[80] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven 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.

[81] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi 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.

[82] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman 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.

[83] Ruohuang Zheng and Michael 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.

[84] Pengfei Zuo, Yu Hua, Ming Zhao, Wen Zhou, and Yuncheng 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.