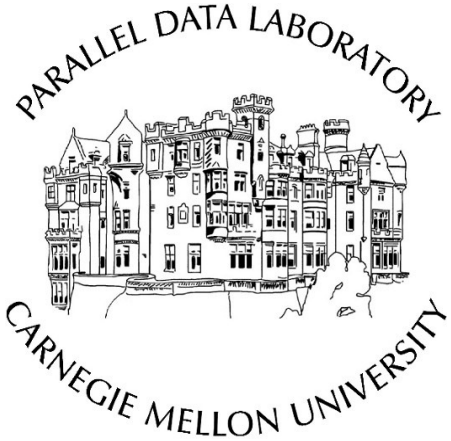


Lazy Redundancy for NVM Storage: Handing the Performance-Reliability Tradeoff to Applications

Rajat Kateja, Andy Pavlo, Greg Ganger
Carnegie Mellon University

CMU-PDL-19-101
April 2019

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

Abstract

Lazy redundancy maintenance can provide direct access non-volatile memory (NVM) with low-overhead data integrity features. The ANON *library lazily maintains redundancy (per-page checksums and cross-page parity) for applications that exploit fine-grained direct load/store access to NVM data. To do so,* ANON *repurposes page table dirty bits to identify pages where redundancy must be updated, addressing the consistency challenges of using dirty bits across crashes. A periodic background thread updates outdated redundancy at a dataset-specific frequency chosen to tune the performance vs. time-to-coverage tradeoff. This approach avoids critical path interpositioning and often amortizes redundancy updates across many stores to a page, enabling* ANON *to maintain redundancy at just a few percent overhead. For example, MongoDB's YCSB throughput drops by less than 2% when using* ANON *with a 30 sec period and by only 3–7% with a 1 sec period. Compared to the state-of-the-art approach,* ANON *with a 30 sec period increases the throughput by up to 1.8× for Redis with YCSB workloads and by up to 4.2× for write-only microbenchmarks.*

Acknowledgments: We thank the members and companies of the PDL Consortium (Alibaba Group, Amazon, Datrium, Dell EMC, Facebook, Google, Hewlett Packard Enterprise, Hitachi, IBM Research, Intel, Micron, Microsoft Research, NetApp, Oracle, Salesforce, Samsung, Seagate, Two Sigma, Veritas and Western Digital) for their interest, insights, feedback, and support.

Keywords: NVM, DAX, asynchronous redundancy

1 Introduction

Non-volatile memory (NVM) changes the way performance-sensitive applications interact with persistent data. NVM storage technologies combine DRAM-like access latencies and granularities with disk-like durability [40, 60, 1, 11, 9]. Applications using direct-access NVM storage map NVM files into their address spaces and access data with load and store instructions, rather than indirectly via file system or block I/O interfaces.

Production storage means more than just non-volatility and performance. A number of features that bolster and simplify storage management efforts are often expected. Whereas some features extend to direct access NVM storage trivially (e.g., background scrubbing, defragmentation), others with data access inter-dependencies do not. Most notably, conventional mechanisms for block checksums and cross-component redundancy fit poorly.

Maintaining such data integrity for direct-access NVM storage, without forfeiting NVM advantages, imposes two challenges. First, access via load and store instructions bypasses the system software, removing the straightforward ability to detect and act on data changes (e.g., to update redundancy information). Second, NVM's cache-line granular writes increase the overhead of updating redundancy information (e.g., checksums) that is computed over sizeable data regions (e.g., pages) to increase effectiveness and space efficiency.

This paper describes **ANON**, a framework providing data integrity features for direct access NVM storage: (i) per-page checksums and (ii) cross-page parity. ANON embraces a lazy approach to reduce overhead and creates a tunable trade-off between performance and quicker coverage for the newest data. Akin to asynchronous remote replication [18, 51, 37, 34], ANON moves redundancy updates out of the critical path, reducing interference with foreground accesses and amortizing the overhead over multiple fine-grained updates. For example, instead of re-computing a per-page checksum on each store or cache line flush, ANON invalidates the outdated checksum and recomputes it within the configured amount of time. While some deployments may demand full coverage and accept its high overheads (e.g., 48% lower Redis throughput), ANON provides control over this behavior.

A key challenge with software-maintained redundancy for direct access NVM storage is efficient and robust detection of updates. Approaches such as interpositioning libraries or use of write-protect mechanisms introduce unacceptable overheads (up to 73% throughput reduction in our experiments). Instead, ANON uses the page table's dirty bits. Conceptually straightforward, this repurposing of the dirty bits requires care to avoid gaps in detection of pages with outdated checksums and parity. Unlike the traditional use case of dirty bits, ANON must ensure that they are consistent with the NVM pages *across* power and kernel failures, and despite write-back caching of data and page table entries (in TLBs and/or CPU caches).

We implement ANON as a user-space library with a supporting kernel module to provide OS interfaces for bulk fetching and clearing of dirty bits. Experiments with two NoSQL DBMSs (Redis and MongoDB) and five microbenchmark applications confirm that ANON can efficiently provide data integrity for direct access NVM storage. Aggressively updating checksums and parity every second with ANON reduces MongoDB's YCSB throughput by only 3–7%. This overhead drops below 2% with a 30 sec window. For YCSB workloads with Redis, ANON is up to 1.8× faster than updating checksums and parity at transaction boundaries [73, 65].

This paper makes three main contributions. First, we identify lazy redundancy maintenance as an important addition to the toolbox of data integrity designs for direct access NVM, providing a tunable trade-off between performance and time-to-coverage. Second, we describe ANON and its efficient and robust lazy redundancy maintenance via careful use of dirty bits. Third, we quantify ANON's efficacy via extensive evaluation using seven macro- and micro-benchmarks, demonstrating unprecedented efficiency for per-page checksums and cross-page parity on direct access NVM storage.

2 Direct-access NVM & Storage Mgmt

This section describes direct-access (DAX) NVM usage model, and the impact of direct access on storage management features. NVM refers to a class of memory technologies that have access latencies comparable to DRAM and retain their contents across power outages like disks. Various NVM technologies, such as 3D-XPoint [1], Memristors [11], PCM [40, 60], and battery-backed DRAM [9, 16], are either already in-use or expected to be available soon [32]. In this paper, we focus on NVM that is accessible like DRAM DIMMs rather than like a disk [48]. That is, NVM that resides on the memory bus, with load/store accessible data that moves between CPU caches and NVM at a cache-line granularity. Accessing NVM via the file system interface incurs high overheads due to system calls, data copying and inefficient general-purpose file system code.

DAX NVM Usage Model: The interface that offers the best performance with NVM storage is direct-access (DAX), wherein NVM pages are mapped into application address spaces and accessed 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 [41, 22]. DAX enables applications to leverage the high performance durability of NVMs by eliminating the page cache and OS software overheads. DAX is widely used for adding persistence to conventionally volatile in-memory DBMSs [44, 62, 73, 57].

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 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 [33, 13, 67, 28, 8]. Alternatively, the system can be equipped with enough battery to allow flushing of cached writes to NVM before any shutdown [47, 75, 52]. Our work assumes this option, though we also evaluate the non-battery case in Section 5.4.

Storage Management for DAX NVM: Administrators rely on and expect a variety of storage management features to avoid data loss [64, 58, 74, 25] and theft [35, 69, 63, 7], reduce cost [76, 17, 59, 66, 39], and handle failures [50, 25, 51, 37, 34]. 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): (i) Background scan/reorganization features, like scrubbing and defragmentation, occur independently of reads and writes. (ii) Space efficiency features, like compression, deduplication, and tiering, track data access recency to estimate data temperature; they can potentially be involved with read operations (e.g., when reading compressed data). (iii) Data redundancy features like block checksums and cross-component redundancy are inter-dependent on writes; they may be involved with reads depending on how they are used (e.g., if reads require a checksum verification). (iv) Security features like encryption are directly involved in servicing of reads and writes.

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

Space efficiency features track data accesses, but the loss of this information does not affect their correctness—only performance (e.g., if a hot page is compressed). Consequently, supporting space efficiency features for DAX 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 warrant hardware support, because of their significant computational overheads and need in the critical path (e.g., to decrypt on read). Introducing

software encryption in the data path would annihilate the performance benefits of DAX NVM storage. Recent works have proposed hardware support for NVM encryption [10, 72, 42].

Data redundancy features, on the other hand, do not need hardware support. Unlike data security features, they have no impact on read accesses if redundancy is not checked on every read, but instead used for background scrubbing to detect corruptions and reconstruction to repair erasures. The benefits of software-managed redundancy features, such as their broader coverage and configurability [58, 74, 25], remain a viable target for DAX NVM. Current approaches rely on interposing on writes [73, 65], introducing significant performance and programming limitations. Section 3 motivates ANON's alternate approach, that provides a robust low-overhead solution by weakening the reliability guarantees.

3 Data Redundancy for DAX NVM

This section describes the challenges in maintaining redundancy[1] for DAX NVM, the solution design space and the state-of-the art solutions. We then motivate ANON's lazy redundancy approach that exposes the performance-reliability tradeoff to applications.

3.1 DAX NVM Redundancy Challenges

Maintaining 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 system to identify updated NVM pages that need to be checksummed and made redundant. A hardware implementation of data reliability features in the memory subsystem could address this challenge, but may not provide the configuration flexibility or end-to-end data reliability required of production storage [58, 74, 25].

Cache-line Granular Writes: Incongruence in the size of writes and the size of blocks over which checksums and redundancy are usually computed increases the overhead of maintaining them for DAX NVM storage. Storage systems compute redundancy over sizeable blocks (e.g., per-page checksums and cross-page redundancy) for space efficiency. Cache-line granular writes require reading (at least) an entire block to update the checksum and redundancy. Whereas RAID systems solve a similar "small write" problem by reading the data before updating it [50], DAX NVM storage systems cannot use this solution. As discussed above, direct access to NVM bypasses system software, prohibiting the use of pre-write values for incremental checksum and redundancy updates.

3.2 Solution Design Space and Tradeoffs

Table 1 summarizes the design space of DAX NVM data redundancy solutions and their tradeoffs. The design space is defined by two choices (i) how are data updates identified? and (ii) when is data redundancy updated?

Previous solutions require applications to explicitly inform the storage software about data updates. For example, Mojim [73] and HotPot [65] replicate updated data at transaction boundaries (referred to as commit points). This explicit notification enables strong reliability guarantees but has two drawbacks. First, it imposes restrictions on the programming model, requiring the use of a file system or transactional library interface. All change points in NVM-accessing software must be modified to report updates, since any that are missed can lead to blind use of stale redundancy. Second, updating data redundancy for each notification

[1] We use redundancy to mean information required to both detect and correct corruptions. Typically, this would be block/page checksums for detection and parity or replicated blocks/pages for correction.

DAX NVM Redundancy Solution	Data Update Identification	Redundancy Updates	Critical Path Interpositioning	Programming Model	Max. Window of Non-Redundancy
Mojim [73], HotPot [65]	Application Instrumentation	Transaction Boundaries	Yes	Restrictive	None
Mojim, HotPot augmented with Lazy Redundancy	Application Instrumentation	Lazily	Yes	Restrictive	Configurable
ANON w/ Volatile Dirty Bits (Section 5.4)	Write Protection	Lazily	Yes	Non-restrictive	Configurable
ANON	Dirty Bits	Lazily	No	Non-restrictive	Configurable

Table 1: Design space of solutions for DAX NVM data redundancy and associated consequences.

imposes a high overhead due to the small update granularities. Our experiments in Section 5.1 demonstrate up to 48% lower throughput for YCSB workloads with Redis using this approach.

Prioritizing greater reliability over performance and ease of programming may not be the desired trade-off for all applications. Indeed, many applications use storage systems that relax some reliability guarantees in favour of performance [18, 51, 37, 34]. A lazy redundancy scheme, wherein redundancy is updated with a configurable delay, allows applications to choose their desired performance-reliability tradeoff. Increasing the delay amortizes redundancy update overheads over multiple writes, but also increases the window not covered by that redundancy.

Delaying the redundancy updates in existing methods improves their performance but does not eliminate the restricted programming model. Moreover, it requires updating metadata at each notification to identify pages. Our measurements indicated that this overhead on the critical path negates most of the benefits of a lazy scheme.

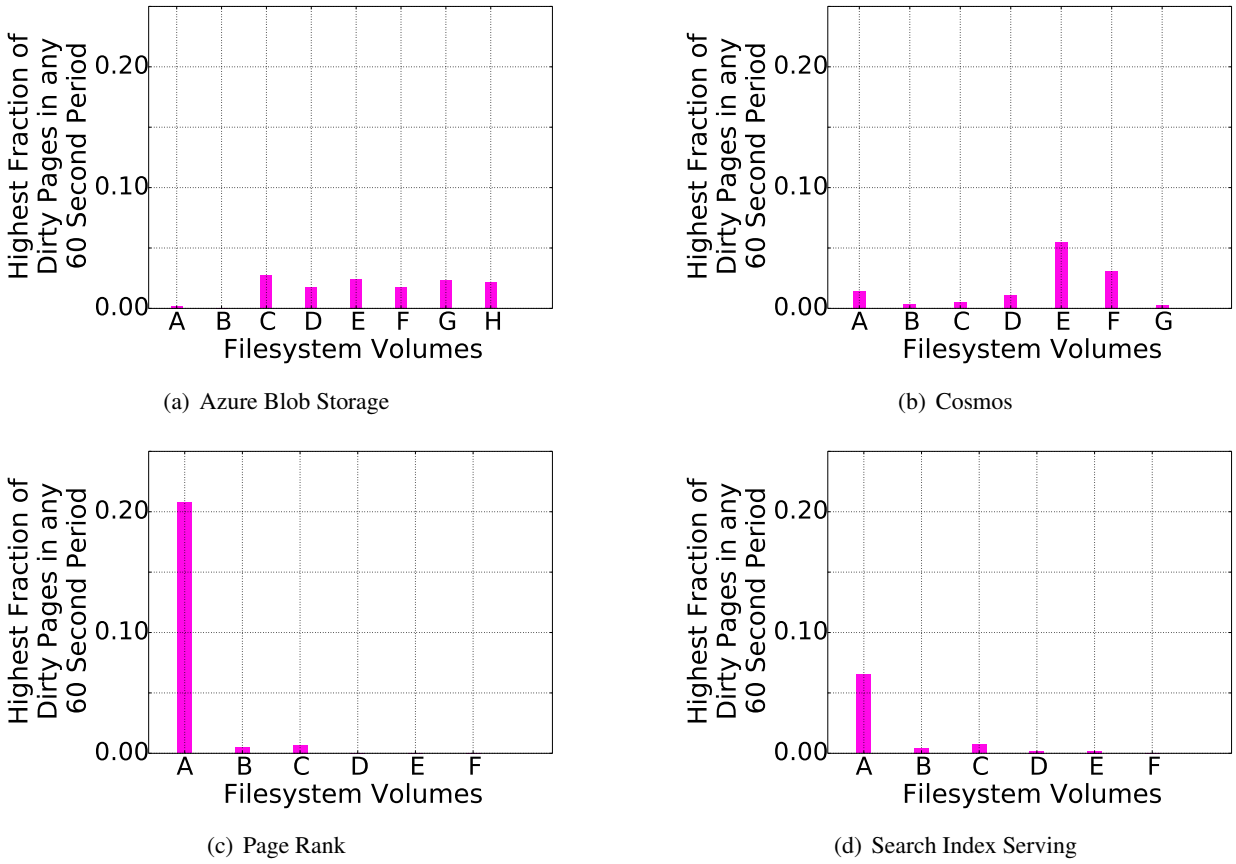
Updated pages can instead be identified by using page write protections. The storage system can map NVM data as read-only and use page faults to identify data updates. The storage system can record pending modifications before making the page writable, and then lazily make the page read-only again and update its redundancy. This approach does not impose any programming restrictions and performs better that state-of-the-art solutions, even when the latter also delay redundancy computations. But, processing the write protection page faults still incurs substantial overhead, as shown in Section 5.4.

ANON embraces an asynchronous approach to identifying updated data, in addition to delaying the redundancy updates. ANON repurposes page table entry dirty bits to identify updated pages after-the-fact. Removing critical path processing (either in the form of notifications or write protection page faults) enables ANON to perform better than both of the delayed redundancy computation approaches described above.

3.3 Implications of Lazy Redundancy

Delaying redundancy updates naturally reduces the coverage of that redundancy. ANON's lazy redundancy scheme protects data at rest from corruption or loss (e.g., due to a firmware bug that loses/misdirects a page write when wear leveling) at all times other than when it is both updated and non-redundant. For a data corruption to go undetected, a page has to be updated after a redundancy verification and corrupted before a redundancy update and verification. While any such windows without coverage are unacceptable for some data, we believe that many cases cannot tolerate the corresponding performance overheads and would benefit from covering most data most of the time at low overheads.

To put the coverage of lazy redundancy into context, we evaluate the fraction of uncovered data for



(a) Azure Blob Storage

(b) Cosmos

(c) Page Rank

(d) Search Index Serving

Figure 1: Highest fraction of pages written in any 60 second period for four datacenter applications.

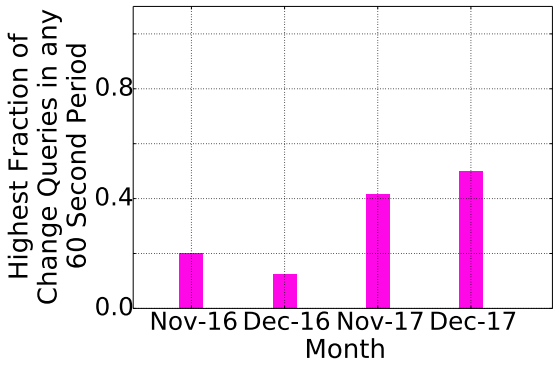
four Microsoft datacenter applications (based on the data reported in Viyojit [36]) and an online transaction processing (OLTP) web application. Figure 1 presents the maximum fraction of pages updated within any 60 second period over the span of 24 hours (3.5 hours for the Cosmos case) for 27 filesystem volumes. The maximum amount of data that would be at risk with a 60 second redundancy update period is less than 7% for all volumes except one in the Page Rank trace, for which it is 20%[2].

We also analyzed a trace of SQL queries to a large university's admissions website over two admission cycles [43]. We look at the fraction of "change queries" (i.e., INSERT, DELETE, and UPDATE queries) that alter the database state. Figure 2(a) shows the maximum fraction of change queries; no 60 second period has more than 50% of change queries. Figure 2(b) shows a CDF of the fraction of change queries; 32% of all the 60 second periods in the four months had no change queries, implying that no part of the database would be at risk due to a lazy redundancy scheme in those periods.

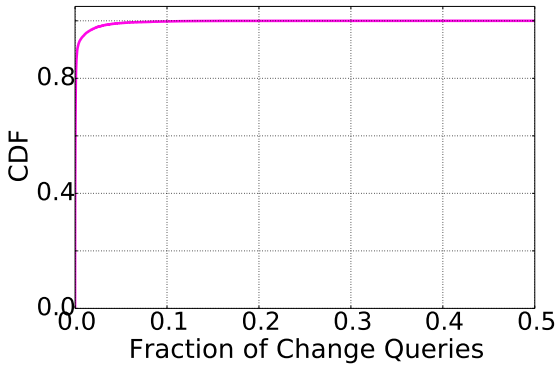
4 ANON Design and Implementation

ANON lazily maintains redundancy for direct access NVM storage using per-page checksums and cross-page parity. By delaying the redundancy computation, ANON amortizes the overhead over multiple NVM writes. The core idea of ANON is to repurpose the page table's dirty bits to identify pages that require re-computation of their redundancy. ANON clears a page's dirty bit when it updates page redundancy;

[2]Viyojit's [36] data only allows us to report the worst-case values and for a period of at least 60 seconds. The fraction updated would be significantly smaller for shorter periods and for non-worst-case periods.



(a) Highest fraction of change queries in any 60 seconds period.



(b) CDF of fraction of change queries in all 60 second periods.

Figure 2: Fraction of change queries (i.e., INSERT, UPDATE, and DELETE queries) that alter the database state for a large university's graduate school admissions' website.

any subsequent write to the page sets the dirty bit that ANON uses to check if the previously computed redundancy is outdated.

4.1 Lazy Checksums and Parity

ANON uses a background thread to periodically update per-page checksums and cross-page parity. Figure 3 illustrates how delaying the computation of per-page checksums (and cross-page parity, not shown in the example) amortizes the computation overhead over multiple cache line writes. The time line shows three cache line writes to a particular page before ANON checksums the page, performing a single computation, instead of three, for the three writes. The reduced computation, and the associated performance benefit, come with a (tunable) window of vulnerability. As shown in the figure, lazy checksumming leave a page written to between two successive updates with an outdated checksum. ANON verifies page checksums and repairs corrupted pages using parity pages in a background scrubbing thread.

ANON's lazy redundancy maintenance requires identifying pages that are updated between two successive invocations of its redundancy-updating thread. Writes to DAX NVM bypass system software, leaving NVM storage systems unaware of data updates that outdate their redundancy. We now describe how ANON repurposes the dirty bits, that are set by the CPU when executing store instructions, to identify pages with stale redundancy.

4.2 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 [22, 41]. 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.

ANON repurposes dirty bits to determine pages that have been written to since their checksum and parity were last computed. When a file is first DAX mapped, its page's dirty bits are unset and checksums and parity are up-to-date (potentially computed during initialization for newly created files). Each successive invocation of ANON's background redundancy-updating thread computes checksum and parity only for

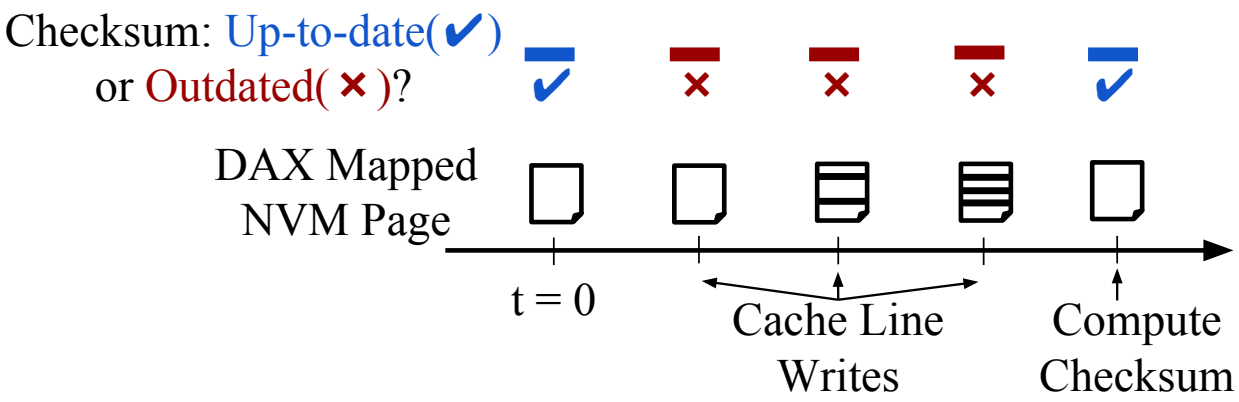


Figure 3: Lazy Checksum Computation Example – By computing per-page checksums lazily, ANON amortizes the computation overhead over multiple cache-line writes to the same NVM page. The reduced computations, and associated performance benefit, come with a window of vulnerability.

pages with their dirty bit set and clears their dirty bit again.

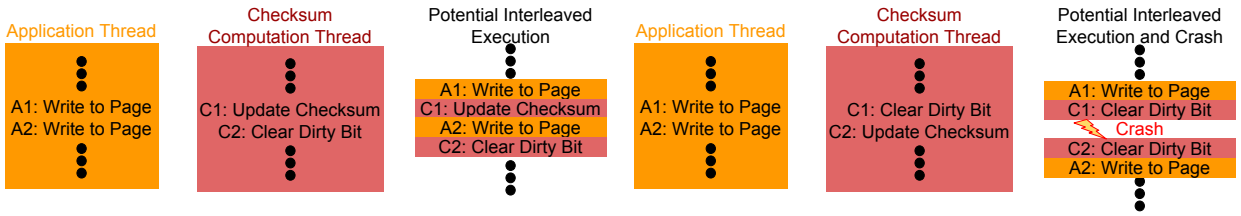
Crash-Consistency Challenge: Repurposing dirty bits to identify outdated redundancy poses a non-trivial challenge of storing dirty bits crash-consistently. In their conventional usage, losing dirty bits across crashes (e.g., power failures) is acceptable because the data that remains on the persistent storage after a crash is self-identifiable as not-dirty. Such self-identification is not possible in our repurposed use of dirty bits. A straightforward extension of identifying pages as dirty or not depending on whether they match their checksum defeats the purpose of checksums; if a non-dirty page (i.e., with up-to-date checksum and parity) gets corrupted and does not match its checksum, it will be classified as dirty, whereas in a correct design, it should be flagged as corrupted and repaired using the parity. Ensuring crash-consistency of dirty bits requires that dirty bits survive various failures, including but not limited to power failures and kernel crashes. We delineate how this can be achieved with non-intrusive hardware and OS support.

Saving Dirty Bits Across Power Failures: We envision systems that maintain dirty bits across power failures by storing page tables persistently in NVM Upon reboot after a power failure, ANON extracts the dirty bit information from the persistent page tables. The OS then discards the page tables, because the processes that the page tables belong to do not survive across reboots.

For an efficient design of persistent page table, we envision systems that employ a battery to include TLBs and CPU caches in the persistence domain. If the TLBs and caches were to be volatile, the memory management unit (MMU) would need to ensure that every page table entry modification is synchronously written back to NVM. The MMU would also need to complete the write-back of dirty bits no later than the write-back of the corresponding page data. Such synchronous write-backs are not guaranteed [31], and supporting them would adversely affect performance and NVM device wear [47, 75]. Using battery-backed TLBs and caches eliminates the need for synchronously flushing page table updates, reducing the performance penalty and NVM device wear.

Saving Dirty Bits Across Kernel Crashes: Dirty bits must also be saved across reboots, even due to kernel failures. To do so, following the Otherworld approach [19], the kernel would flush TLBs and processor caches in appropriate handlers prior to rebooting the system, e.g., in the panic and non-maskable interrupt (NMI) handler for kernel panics and deadlocks respectively (x86 systems support using a a watchdog that generates a NMI to trigger reboot for deadlocked kernels [53]). Such OS support requires only small modifications (e.g., a single `wbinvd` instruction to flush processor caches), and imposes no overhead during normal system operation.

Shadow Dirty Bits for Synchronization-Free Foreground Accesses: Even with hardware and OS support for crash-consistent dirty bits, ANON needs to carefully orchestrate the non-atomic two-step process of



(a) A write can happen after updating the checksum and before clearing the dirty bit. This would cause the dirty bit to be cleared even when the checksum is not up-to-date with the latest write to the page.

(b) A crash after clearing the dirty bit and before updating the checksum would cause an incorrectly flagged data corruption. (Upon reboot, the dirty bit would be unset but the page data will not match its checksum).

Figure 4: Performing the checksum update and clearing of dirty bit in either order, without any safeguards, is incorrect.

updating a page's redundancy and clearing its dirty bit; performing these two steps without any additional safeguard would be incorrect. As shown in Figure 4(a), dirty bit could be incorrectly cleared for a page with outdated checksums because of interleaving of ANON's background and application's foreground thread. Reversing the order of the two steps is not safe either, as shown in Figure 4(b). If the system crashes after the dirty bit is cleared but before the checksum is updated, ANON would incorrectly identify a data corruption upon reboot (dirty bit is cleared but page's checksum is outdated). Write-protecting pages before updating their checksum can solve this problem, but would impact foreground data accesses. Instead, ANON makes a persistent shadow copy of the dirty bit before clearing it, and clears this shadow copy only after completing the redundancy update; if ANON finds either of the dirty bit or its shadow copy to be set for a page, it knows that the page's redundancy is outdated.

4.3 Implementation

We implement ANON as a user-space library with an API that allows applications to configure the type and frequency of checksum and parity computations for their DAX NVM files. The library uses a periodic background thread that checks and clears the dirty bits using new system calls that we implement, and performs the checksum and parity updates for the dirty pages. Our implementation uses a stripe size of five pages, with four consecutive data pages and one parity page. The stripes are statically determined at the time of file creation. Figure 5 shows the key 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 ANON is unaware of pages dirtied in between the checking and clearing and will not update their redundancy, it clears the dirty bits only for pages that were dirty when initially checked.

Batched Checking and Clearing: ANON's userspace library 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 dirty bits further requires invalidating the corresponding TLB entries. Each of these is a costly operation, as evinced by prior research [3], and demonstrated by our experiments (Section 5.3). 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/clearing dirty bits by up to two orders of magnitude.

Algorithm: Algorithm 1 details the steps that ANON's userspace background thread performs on each invocation. ANON loops over all the N pages in a given DAX NVM file in increments of B pages; B being

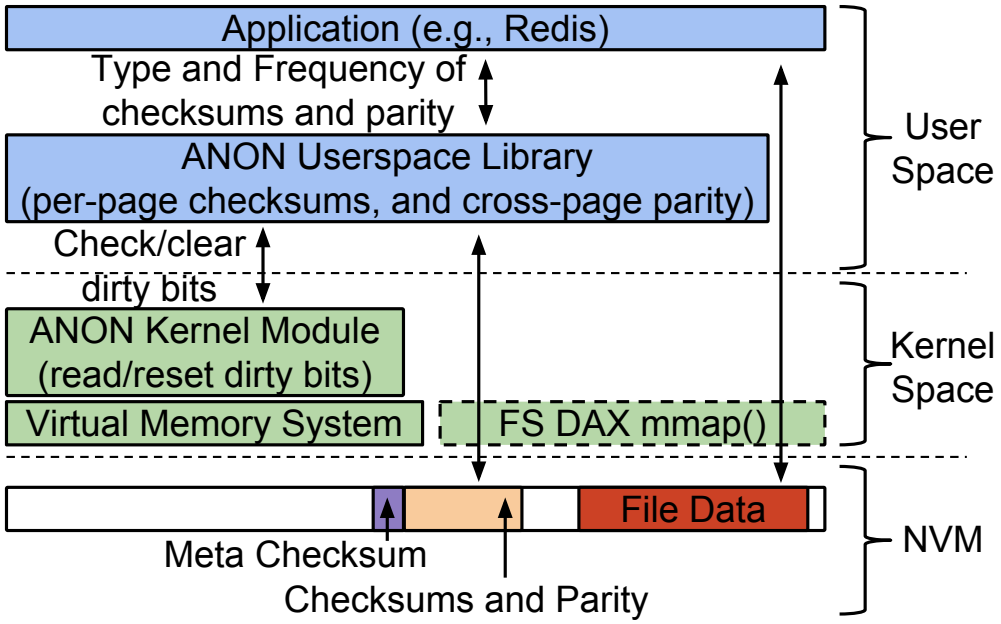


Figure 5: ANON's Implementation: The user space library performs the checksum and parity computations with a period that is set by the application. The kernel module checks and clears the dirty bits when requested by the user space library.

the batch size for which ANON checks the dirty bits using a single system call (Line 2). ANON stores a persistent shadow copy of the dirty bits (Line 3) and then clears them (Line 5). ANON updates the checksum of each dirty page (Line 11), and the parity of a group of P page if either of them is dirty (Line 15). ANON stores the checksums and parity separately from the data (Figure 5) and then clears the shadow copy of the dirty bits (Line 19). ANON then updates a meta-checksum (checksum of the page checksums) after every iteration (Line 21 and Figure 5).

As a performance optimization, instead of recording 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.

Notes: Our implementation of ANON 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 our assumed system model consists of battery-backed processor caches (and TLBs) as discussed earlier. We do, however, use fences to ensure ordering between updates. For example, the fence at Line 18 ensures that checksums and parity are 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 ANON with in our evaluation.

5 Evaluation

We demonstrate ANON's efficacy using seven macro- and micro-benchmarks. We first evaluate ANON using Yahoo! Cloud Serving Benchmark (YCSB) [15] workloads with Redis [61] and MongoDB [45]. We

Algorithm 1: Checksum and Parity Update Thread

```
Parameter: Batch Size, B
Parameter: Number of Pages in File, N
Parameter: Number of Pages in a Parity Group, P
for i ← 0 to N increment by B do
    dirtyBitvector ← checkDirtyBits(i, i + B);
    dirtyBitvectorCopy ← dirtyBitvector;
    currentBatchStartingPage ← i;
    clearDirtyBits(i, i + B, dirtyBitvector);
    for j ← i to i + B increment by P do
        for k ← j to j + P increment by 1 do
            updateParity ← False;
            if bitIsSet(dirtyBitvector, k − i) then
                updateParity ← True;
                computePageChecksum(k);
            end
        end
        if updateParity then
            computeParity(j, j + P);
        end
    end
    memoryFence;
    dirtyBitvectorCopy ← 0;
end
computeMetaChecksum();
```

then evaluate ANON using fio microbenchmarks [6] and get-/put-only microbenchmarks with four key-value (KV) data structures. Following that, we dissect the cost of checking and clearing dirty bits, and demonstrate the benefit of batching these operations. Lastly, we describe and evaluate ANON's fallback write protection mechanism to identify stale redundancy for systems that lack the hardware or OS support for crash-consistent dirty bits.

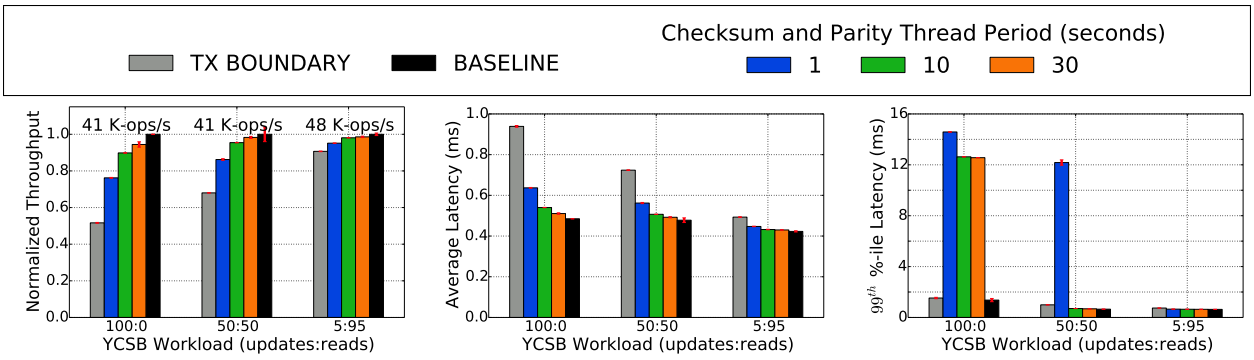
We compare ANON with two alternate designs: (i) Baseline—no checksums and parity, and (ii) TxBoundary—update checksums and parity at transaction boundaries. Baseline corresponds to the best performance but provides no redundancy guarantees. TxBoundary ensures that redundancy is updated when data is persisted, and is the design used by Mojim [73] and HotPot [65]. Unless mentioned otherwise, we use a 512 page batch size for checking/clearing dirty bits.

To fairly quantify ANON's overhead, we run the application (e.g., Redis, MongoDB) and ANON's background redundancy update thread on the same single core. Each data point in our results is an average of three runs with root mean square error bars. We perform all our experiments on a dual-socket Intel Xeon Silver 4114 machine with Linux 4.4.0 kernel. The system has 192 GB DRAM, from which we emulate and use 64 GB as NVM [56].

5.1 YCSB with Redis and MongoDB

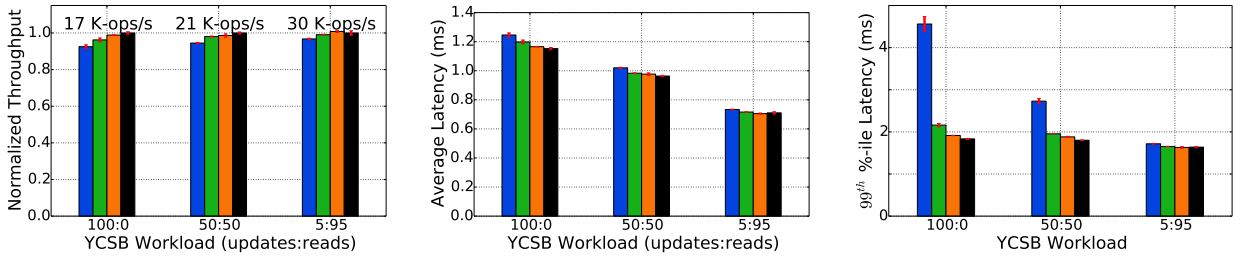
Redis [61] and MongoDB [45] are widely used open-source NoSQL DBMSs. We modify Redis (v3.1) to use DAX NVM files for its data heap. We use an open-source MongoDB (v3.5) storage engine (PMSE [57]) that does the same. Our implementation for Redis uses Intel PMDK's `libpmemobj` library [33]. We implement TxBoundary in `libpmemobj` allowing us to compare ANON with Baseline and TxBoundary for Redis. For MongoDB, we compare ANON with only Baseline; PMSE uses Intel PMDK's `libpmemobj++` library for which we we did not implement TxBoundary.

Experimental Setup: We use YCSB workloads and measure the impact on workload throughput, and update latency. We use update-only (100% updates), balanced (50% updates) and read-heavy (5% updates)



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

Figure 6: YCSB (Zipf Access) with Redis – Throughput and update latency of YCSB workloads with Redis. We normalize throughputs to that of Baseline and annotate the graph with Baseline throughput.



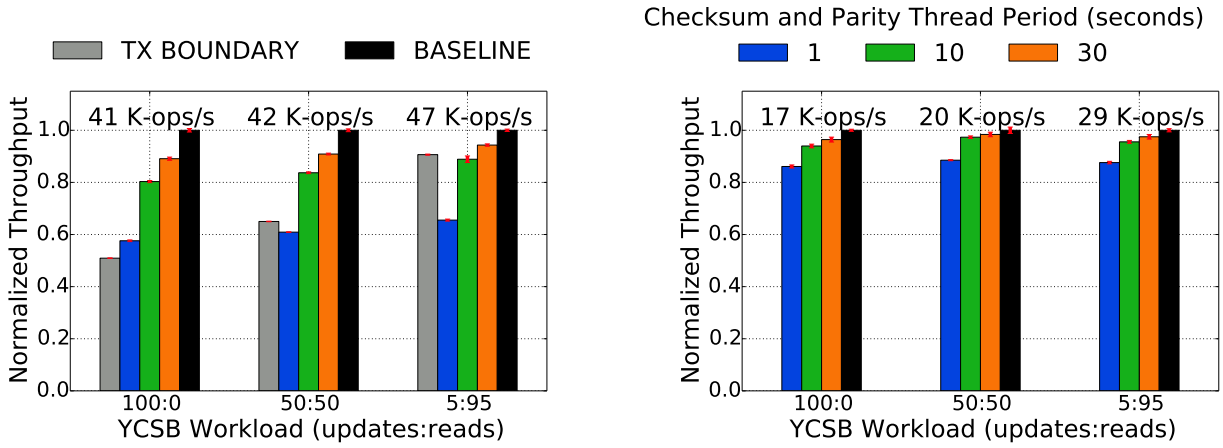
(a) Throughput (b) Average Latency (c) Tail Latency

Figure 7: YCSB (Zipf Access) with MongoDB – Throughput and update latency of YCSB workloads with MongoDB. We normalize throughputs to that of Baseline and annotate the graph with Baseline throughput.

workloads with Zipf and uniform random access patterns. We initialize the DBMSs with 1M (1×2^{20}) key-value pairs and run the workloads for five minutes. YCSB workload generator and the DBMSs run on different sockets.

YCSB (Zipf Access) with Redis Results: Figure 6 presents throughput and update latencies for YCSB Zipf workloads with Redis. We normalize the throughputs to that of Baseline and annotate the Baseline throughput. ANON reduces throughput by 1–5% for a checksum and parity computation period of 30 sec and by 4–23% for a period of 1 sec. Increasing the delay between computations allows ANON to amortize the computations over more writes to the same pages, improving its performance. TxBoundary imposes 9–48% reduction in throughput because it updates redundancy at the end of every transaction, performing up to 14× more computations than ANON. TxBoundary's inline redundancy updates increase the average latency by 16–93%. In contrast, ANON's out-of-critical-path redundancy updates increase the average latency by a maximum of 5% for a period of 30 sec. The increase in tail latency with ANON is because it shares the same core with Redis – when the OS schedules ANON, ongoing Redis transactions stall, increasing the tail latency. This can be avoided if ANON was run on a separate dedicated core. As expected, the throughput and latency overheads are lowest, and Baseline performance is best for the read-heavy workload.

YCSB (Zipf Access) with MongoDB Results: Figure 7 presents the throughput and update latencies for YCSB Zipf workloads with MongoDB. We normalize the throughput to that of Baseline and annotate the Baseline throughput. MongoDB has lower throughput than Redis for both Baseline and ANON. This is because MongoDB's storage engine, PMSE [57], uses a B+Tree index whereas Redis uses a chaining hashtable index; our workloads consist of only point-queries, for which hashtables perform better than B+Tree. Even with aggressive checksum and parity updates every second, ANON reduces the throughput by only 3–7%, and increases the average latency by only 3–8%. Increasing ANON's redundancy update delay further im-



(a) Throughput with Redis

(b) Throughput with MongoDB

Figure 8: YCSB (Random Access) with Redis and MongoDB – Throughput normalized to that of Baseline.

proves performance by reducing the number of computations. The high impact on tail latency (e.g., up to 1.4× for the update-only workload) is because ANON and MongoDB share the same core. For the read-heavy workload, the increase in tail latency is only 0.4–5% because ANON's computation thread runs, and stalls MongoDB, only for small durations.

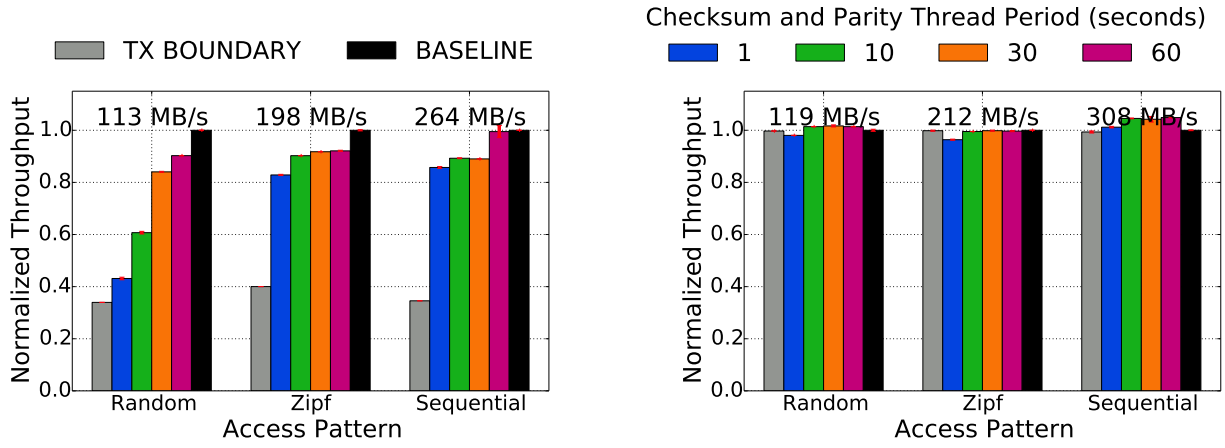
YCSB (Random Access) with Redis and MongoDB: Figure 8 presents the normalized throughput for YCSB workloads with uniform random access patterns with Redis and MongoDB. For Redis, ANON reduces the throughput by 5–10% for a period of 30 sec compared to 9–49% with TxBoundary. When updating redundancy every second, ANON suffers from a higher overhead than TxBoundary for read-heavy and balanced workloads. This is because the reduction in the number of computations is not enough to offset the overhead of periodic checking/clearing of dirty bits. For MongoDB, ANON imposes a throughput reduction of 2–13%. ANON reduces throughput more for random access workloads than Zipf access workloads with both Redis and MongoDB. This is because, as corroborated by microbenchmarks in Section 5.2, Zipf workloads offer more opportunities for ANON to reduce computations as a larger fraction of the writes go to the same page.

5.2 Fio and Key-Value Microbenchmarks

We now evaluate ANON's performance using fio [6] microbenchmarks and synthetic get-/put-only microbenchmarks with four key-value (KV) data structures.

Fio Experimental Setup: We use fio with its libpmem engine [24]. The 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 compare ANON with Baseline and TxBoundary.

Fio Results: Figure 9 shows the normalized throughput for the two workloads with three access patterns each. We annotate the graphs with Baseline throughput. For write-only workloads, ANON reduces throughput by 0.5–56% with higher overheads for more frequent computations. TxBoundary has a 6–66% throughput reduction. ANON'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 redundancy update delay of 60 secs. ANON reduces the throughput by only up to 3% for read-only workloads, demon-



(a) Write Only Workload

(b) Read Only Workload

Figure 9: Fio Microbenchmarks – Throughputs (normalized to that of Baseline) for write-only and read-only workloads with different access patterns.

strating that ANON's background checking of dirty bits is low-overhead.

KV Microbenchmark Experimental Setup: We use synthetic get- and put-only microbenchmarks with four KV data structures: B+Tree, chaining hashmap, Red-Black Tree (RB Tree) and FP Tree [49]. We use PMDK's [33] B+Tree, hashmap, and RB Tree implementations and its `pmembench` utility to generate workloads. We populate these three data structures with one million KV pairs, run the workloads for 60 secs, and compare ANON with Baseline and TxBoundary. We use the FP Tree implementation from Intel's `pmemkv` [30] library and its `db_bench` utility to generate the workloads. We initialize the FP Tree with 20 million KV pairs, run the workloads for 60 secs, and compare ANON with Baseline (we did not implement TxBoundary for `pmemkv`).

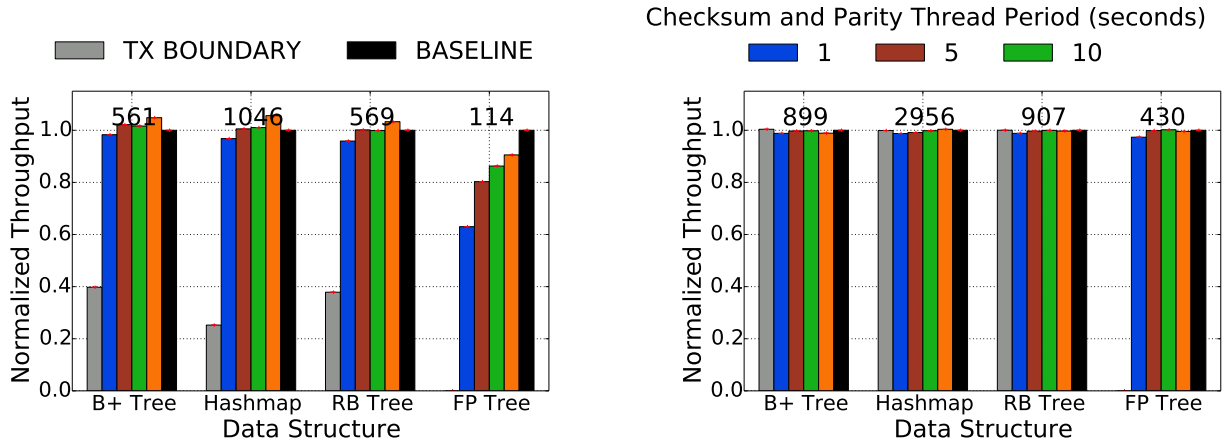
KV Microbenchmark Results: Figure 10 shows the normalized throughput for the four data structures with put-only and get-only workloads, with annotated Baseline throughput in K-ops/s. For the put-only workload, ANON reduces throughput by a maximum of 4% when updating redundancy every second for B+Tree, hashmap and RB Tree, compared to 60–74% reduction with TxBoundary. For FP Tree, ANON reduces throughput by 13–37% for computation periods of 1 to 10 secs. Increasing the period to 30 secs reduces ANON's overhead for FP Tree to 9% (not shown in the graph).

Among the four data structures, hashmap performs the best because our workloads only contain point-queries. Hashmap's superior performance implies that it performs the highest number of writes for our fixed duration put-only workload. Hence the overhead of TxBoundary is highest for hashmap. FP Tree Baseline has the lowest throughput because its implementation deletes and re-allocates a key-value pair instead of overwriting the value for put operations. This also causes higher overhead when using ANON because deletes and allocations write to the persistent allocator state. For get-only workload, ANON imposes only 1–2% throughput reduction, reinforcing ANON's efficacy in checking dirty bits.

5.3 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 fio with its libpmem engine described above. We use the write-only workload with 64-byte writes and uniform random access pattern. We configure ANON to check/clear the dirty bits every second. We measure the average amount of time spent in each of the components for a single invoca-



(a) Put Only Workload

(b) Get Only Workload

Figure 10: KV Microbenchmarks – Throughput for KV workloads with various data structures. We normalize the throughputs to that of Baseline, which is annotated in K-ops/s.

tion of ANON's background thread. We vary the batch size to demonstrate the impact of batching.

Results: Figure 11(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 number of pages in a file, 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 number of pages in the file. The number of pages for which the dirty bit is cleared, and the number of TLB invalidations depends on the workload's access pattern. For the uniform random access workload, these are also directly proportional to the number of pages in the file. We also find that resetting the dirty bits is costlier than reading them.

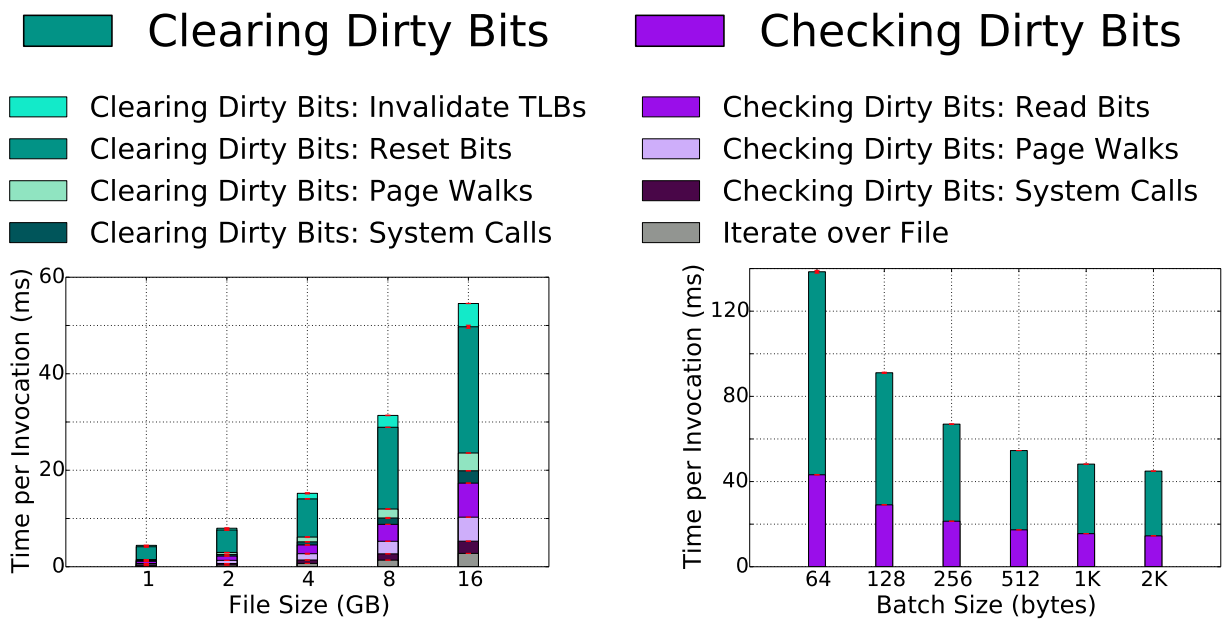
Figure 11(b) presents the impact of batch size for a 16 GB file. As the batch size increases, the time spent in checking/clearing dirty bits decreases with diminishing marginal returns. This decrease is because the number of system calls reduce and larger fractions of the page table walks are shared 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.

5.4 ANON with Volatile Dirty Bits

Repurposing dirty bits to identify pages with stale redundancy requires hardware and OS support that may not be universally available. In absence of such support, ANON uses write protections. We briefly discuss this design and present its evaluation using Redis with YCSB Zipf workloads and fio write-only microbenchmarks.

If ANON cannot rely on dirty bits across crashes, it write protects pages after updating their checksum and parity. ANON registers a page fault handler and gets notified when an application attempts to write to a read-only page. In the page fault handler, ANON persistently records that the page is going to be dirtied, out-dating its redundancy. In essence, ANON implements its own crash-consistent dirty bits. The rest of ANON's design remains the same as previously described.

Experimental Setup: We consider two system models, (i) Volatile Caches—systems with no battery backup for CPU caches, and (ii) Battery-Backed Caches—systems with battery backed CPU caches. Volatile caches necessitate using write protections; battery-backed caches may or may not depending on whether the OS offers the additional support for repurposing and storing dirty bits crash-consistently. Battery-backed caches offer performance benefits for ANON as well as for Baseline and TxBoundary by eliminating the need for cache line flushes.



(a) Breakdown of Time Spent

(b) Impact of Batch Size

Figure 11: Cost of Checking/Clearing Dirty Bits – 11(a) shows the time spent in each component of checking/clearing dirty bits for a batch size of 512 pages and increasing file sizes. 11(b) shows that increasing the batch size reduces the time spent in checking/clearing dirty bits with diminishing returns.

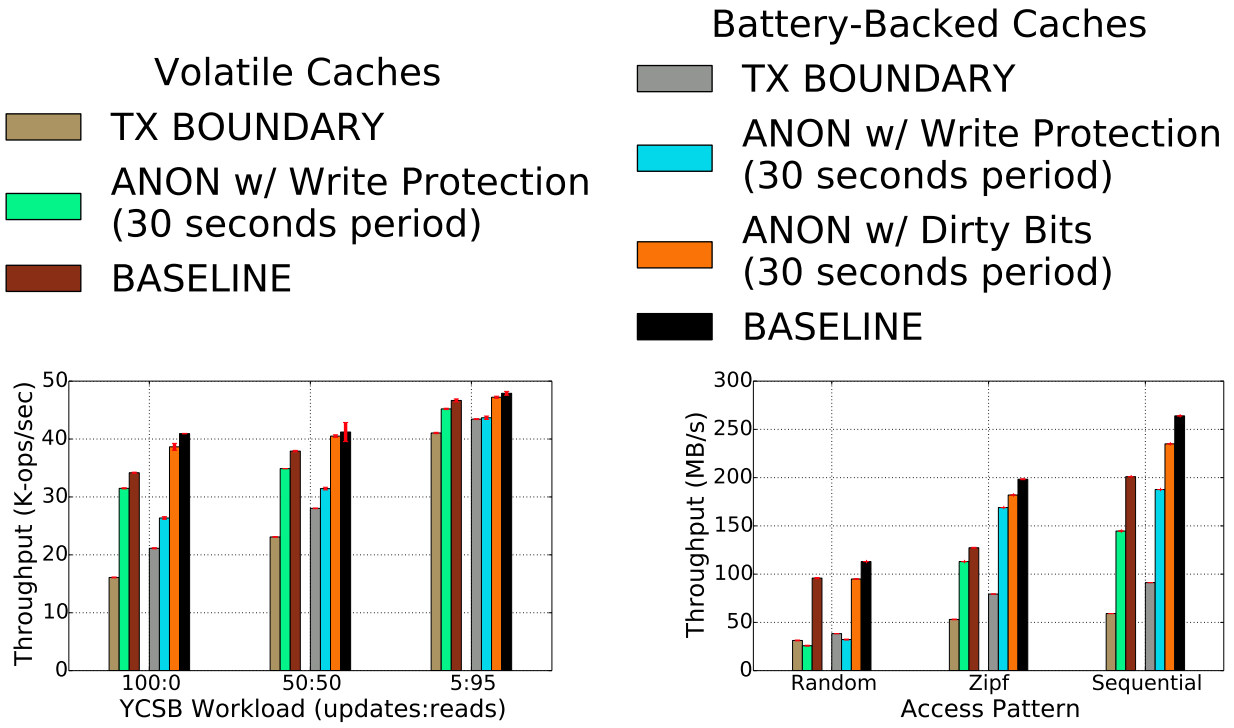
Results: Figure 12 presents the throughput for TxBoundary, ANON (with write protection and with dirty bits), and Baseline for Redis with YCSB Zipf workloads (Section 5.1) and fio random write-only workloads (Section 5.2). Using write protections reduces ANON's throughput by 4–66% compared to when using dirty bits (in the battery-backed caches case). With volatile caches, ANON also has to perform cache line flushes for durability This further reduces ANON's throughput by 2–35%. Baseline and TxBoundary throughput also reduce by 5–35% and 18–32% respectively when battery backup is removed. ANON with write protections performs worse than TxBoundary for fio's random write workload because the overhead of repeated write protections and page faults outweighs the benefits of lazy redundancy.

5.5 Summary of Results

ANON's lazy redundancy approach increases Redis' YCSB throughput by up to 1.8×, and write-only microbenchmarks' throughput by up to 7.5× compared to the state-of-the-art TxBoundary design. ANON reduces MongoDB's YCSB throughput by only 3-7% for a period of one sec, and by <2% for a period of 30 secs. ANON's performance improves with increasing delay between computations, with increasing ratio of reads, and increasing spatial locality of writes. ANON's periodic checking of dirty bits reduces throughput by a maximum of 3%.

6 Related Work

State-of-the-art software support for NVM data redundancy falls into two categories. First, systems like NOVA-Fortis [71] and NetApp's Pelxistore [55] support checksums and replication only for data that is accessed via the file system interface. In contrast, ANON supports both for DAX NVM. Nova-Fortis does support snapshots of DAX NVM data via copy-on-write of write-protected snapshot pages. As shown



(a) YCSB (Zipf) with Redis

(b) Fio Workloads

Figure 12: ANON with Volatile Dirty Bits – Throughput of YCSB Zipf workloads and fio write-only workloads. Volatile caches correspond to systems that require applications to perform cache line flushes for durability. With volatile caches, ANON cannot use dirty bits and has to use write protections. Battery-backed caches do not require cache line flushes for durability. ANON can use dirty bits if the required OS support is present, or fall back to using write protections.

in Section 5.4, using write protection as a foundation for providing checksums and redundancy reduces throughput significantly (up to 73%).

Second, systems like Mojim [73] and HotPot [65] replicate DAX NVM data when the application requests that data be persisted, as with the TxBoundary approach in our experiments. In contrast, ANON does not require any application modifications and provides a tunable performance-reliability tradeoff (e.g., 1.8× higher Redis throughput with a 30 sec delay in redundancy updates).

Prior works have described mechanisms to protect DAX NVM data from stray writes. Privilege levels, process isolation, and SMAP protect applications and kernel from each others stray writes [22]. Mapping data as read only and toggling its write protection before and after a write reduces the probability of an application or kernel corrupting its own data [9]. Protection keys [27, 54, 29] reduce the overhead of toggling write protections. These mechanisms are complementary to, and can be used in conjunction with ANON's software managed redundancy.

ANON's system model of using battery-backed TLB and CPU caches conforms to the widespread use of batteries to treat inherently volatile mediums as durable. Systems have used battery-backup for DRAM [68, 9, 21, 16, 2, 36], CPU caches [52, 75], and the entire system state (including DRAM, CPU caches and registers, and TLBs) [47]. Similarly, ANON's modification of kernel panic handlers to flush page tables borrows from prior works on supporting OS/hypervisor microreboots [19, 23] and kernel dumps [26] using modified panic handlers.

Lazy enhancement of data reliability has a long history in HDD/SSD-based storage systems [18, 51, 37, 34]. For example, many systems with remote replication support treat a write as complete after it is stored on the primary machine, and then send it to the remote replica in the background. Some systems will

put data into a write-back cache and then store it redundantly across multiple disks in the background. Like ANON, these systems embrace a model in which the fullest form of supported reliability is in place only after some time, in order to avoid delaying foreground operation. Unlike ANON, these systems do not face the challenges of direct load/store access to NVM storage.

7 Future Work: NVM Controller Managed Dirty Bits

We discuss how additional hardware support can make ANON's design simpler and more efficient. Repurposing the dirty bits to identify stale checksums and parity poses correctness as well as performance challenges. Maintaining dirty bits durably in NVM using NVM controller (independent of the page table entry dirty bits, which remain as is) can address both of these challenges. The dirty bit corresponding to a given NVM page could be set by the NVM controller when the page is written to, and the OS would be able to clear it. This would also eliminate the need for safeguards against power failures and kernel crashes, simplifying the robustness of ANON's repurposed dirty bits. We leave the design of NVM controller managed dirty bits and exploration of its benefits for future work.

8 Conclusion

ANON provides a tunable performance-reliability tradeoff for direct access NVM data integrity features. ANON embraces an asynchronous approach to data redundancy, offloading per-page checksum and cross-page parity computations to a periodic background thread. ANON repurposes page table entry dirty bits to robustly identify pages with stale redundancy, despite power failures, OS crashes, and concurrent foreground application activity. As a result, the overhead of using ANON is small and tunable: 3–8% for MongoDB's YCSB throughput with a 1-second period and $<$2% with a 30-second period. Compared to state-of-the-art approaches, it allows 1.8$\times$ higher YCSB throughput for Redis and 4.2$\times$ higher for write-only microbenchmarks. ANON's ability to efficiently maintain data redundancy fills a critical gap as direct access NVM storage moves toward production use.

References

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

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

[3] Nadav 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.

[4] 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.

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

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

[7] Matt 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.

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

[9] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David 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.

[10] Siddhartha Chhabra and Yan 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.

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

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

[13] 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.

[14] 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.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10', pages 143–154, New York, NY, USA, 2010. ACM.

[16] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe ram. In *Proceedings of the 15th International Conference on Very Large Data Bases*, VLDB '89, pages 327–335, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

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

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[19] Alex Depoutovitch and Michael Stumm. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 181–194, New York, NY, USA, 2010. ACM.

[20] Mingkai Dong and Haibo 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.

[21] 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.

[22] 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.

[23] David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Mini-Ckpts: Surviving OS Failures in Persistent Memory. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 7:1–7:14, New York, NY, USA, 2016. ACM.

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

[25] 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.

[26] Vivek Goyal, Eric W Biederman, and Hariprasad Nellitheertha. Kdump, a kexec-based kernel crash dumping mechanism. In *Proc. of the Linux Symposium*. Citeseer, 2005.

[27] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. Itaniuma system implementors tale. In *Proceedings of the 2005 Annual USENIX Technical Conference*, ATC'05, pages 264–278, 2005.

[28] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas 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.

[29] *Intel Software Developer's Manual: describes protection keys in Section 4.6.2*. http://www.intel.com/sdm.

[30] *PMEMKV: Key/Value Datastore for Persistent Memory*. https://github.com/pmem/pmemkv.

[31] *Intel Software Developer's Manual: describes caching of page table entries in Section 4.10*. http://www.intel.com/sdm.

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

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

[34] 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.

[35] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin 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.

[36] 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.

[37] 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.

[38] 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.

[39] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas 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.

[40] 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.

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

[42] 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.

[43] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 631–645, New York, NY, USA, 2018. ACM.

[44] 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.

[45] *MongoDB*. http://www.mongodb.com/.

[46] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 135–148, New York, NY, USA, 2017. ACM.

[47] 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.

[48] *Intel Optane Memory SSDs*. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html.

[49] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. ACM.

[50] 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.

[51] 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.

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

[53] *Linux documentation: lockup-watchdogs*. https://github.com/torvalds/linux/blob/master/Documentation/lockup-watchdogs.txt.

[54] *Memory Protection Keys Man Page*. http://man7.org/linux/man-pages/man7/pkeys.7.html.

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

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

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

[58] 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.

[59] Sean Quinlan and Sean 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.

[60] 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.

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

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

[63] Erik Riedel, Mahesh Kallahalla, and Ram 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.

[64] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. *ACM Trans. Storage*, 6(3):9:1–9:23, September 2010.

[65] 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.

[66] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai 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.

[67] 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.

[68] An-I Andy Wang, Geoff Kuenning, Peter Reiher, and Gerald Popek. The conquest file system: Better performance through a disk/persistent-ram hybrid design. *Trans. Storage*, 2(3):309–348, August 2006.

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

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] Benjamin Zhu, Kai Li, and Hugo 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.