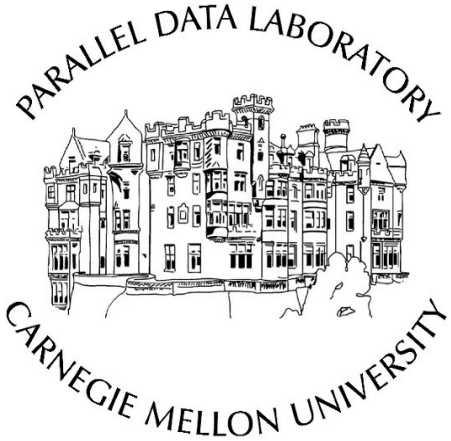


Vilamb: Low Overhead Asynchronous Redundancy for Direct Access NVM

Rajat Kateja, Andy Pavlo, Greg Ganger
Carnegie Mellon University

CMU-PDL-20-101
April 2020

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 Vilamb 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, Vilamb 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 A periodic background thread updates outdated redundancy at a dataset-specific frequency chosen to tune the performance vs. time-to-coverage tradeoff.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 Vilamb to maintain redundancy at just a few percent overhead. For example, MongoDB's YCSB throughput drops by less than 2% when using Vilamb with a 30 sec period and by only 3–7% with a 1 sec period. Compared to the state-of-the-art approach, Vilamb 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, Facebook, Google, Hewlett Packard Enterprise, Hitachi, IBM Research, Intel, Micron, Microsoft Research, NetApp, Oracle, Salesforce, Samsung, Seagate, Two Sigma) for their interest, insights, feedback, and support.

Keywords: NVM, DAX, asynchronous redundancy

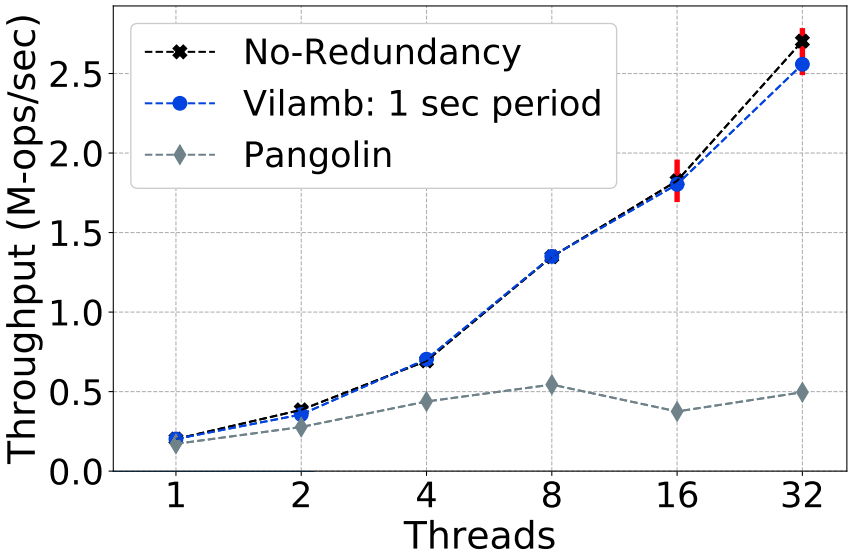


Figure 1: Throughput for a PMDK key-value store when using three system-redundancy options, as a function of the number of threads performing PMDK's insert-only benchmark workload. (Details in Section 4.3; RBtree results shown here.)

1 Introduction

Non-volatile memory (NVM) storage combines DRAM-like access latencies and granularities with disk-like durability [39, 54, 1, 11, 10]. Direct access (DAX) to NVM data exposes raw NVM performance to applications. Applications using DAX map NVM files into their address spaces and access data with load and store instructions, eliminating system software overheads associated with conventional storage interfaces.

Production storage demands fault tolerance in addition to non-volatility and performance. Whereas some fault tolerance mechanisms extend to DAX NVM storage trivially (e.g., background scrubbing), others do not. In particular, mechanisms for resilience against device-firmware-bug-induced data corruption fit poorly. FS-level page checksums enable detection of firmware-bug-induced data corruption, and cross-page redundancy enables recovery from such corruptions [53, 6, 29, 7, 60]. We use *system-redundancy* to refer to FS level checksums and cross-page redundancy.

Maintaining system-redundancy for DAX NVM storage, without forfeiting its performance benefits, is challenging for two reasons. First, accesses via load and store instructions bypass system software, removing the straightforward ability to detect and act on data changes (e.g., to update system-redundancy). Second, NVM's cache-line granular writes increase the overhead of updating system-redundancy (e.g., checksums) that is usually computed over sizeable data regions (e.g., pages) for effectiveness and space efficiency.

The state-of-the-art solution for DAX NVM system-redundancy is the Pangolin library [69]. Pangolin addresses the challenge of system software bypass by requiring applications to use its transactional API. This enables Pangolin to mediate and act on data accesses To address the incongruence in DAX write and system-redundancy granularities, Pangolin introduces micro-buffering and per-object checksums. Pangolin buffers application writes in DRAM and updates the NVM only on transaction commits. This buffering also enables Pangolin to use data diffs to make system-redundancy updates more efficient.

Even with Pangolin's well-optimized design, synchronous system-redundancy updates incur significant overhead. For example, Figure 1 shows that Pangolin reduces key-value insert throughput by 10–20% at low insert rates, compared to a No-Redundancy baseline, and by up to 80% at high rates. Fundamentally, any

software-based synchronous approach will struggle with high throughput updates because it must update system-redundancy on every operation. A recently proposed specialized hardware controller offers low-overhead synchronous DAX NVM system-redundancy [33], but it is unlikely to be available in systems soon.

This paper describes **Vilamb**[1], a user-space library for efficient asynchronous DAX NVM system-redundancy. Vilamb moves system-redundancy updates out of the critical path and delays them to amortize the overhead over multiple data updates. Delaying the system-redundancy updates creates a configurable trade-off between the delay before updated data is covered and performance. Figure 1 shows that updating system-redundancy every second with Vilamb reduces the No-Redundancy throughput by only 6%, even at the highest throughput level; this corresponds to 5× higher throughput than Pangolin. Although Vilamb leaves a fraction of data briefly uncovered, it increases the mean time to data loss (MTTDL) due to firmware-induced corruptions by 112× over No-Redundancy for this benchmark.

Unlike Pangolin, Vilamb does not require applications to adopt a particular access interface to identify data updates. Instead, Vilamb repurposes page table dirty bits to efficiently identify of data updates. Vilamb marks pages with updated system-redundancy as clean and identifies pages with outdated system-redundancy by checking their dirty bit. We implement a kernel module that Vilamb uses for batched fetching and clearing for dirty bits. Vilamb ensures atomic and consistent system-redundancy updates for all dirty pages by using shadow copies of dirty bits and leveraging batteries that are common in production environments [18, 31, 32, 46, 22, 63, 37, 64].

Extensive evaluation with eight macro- and micro-benchmarks demonstrate Vilamb's efficacy. Vilamb with a 1 sec delay between system-redundancy updates reduces single-threaded Redis' YCSB throughput by only 1.6–17%, compared to 13–18% for Pangolin. Increasing the delay to 10 seconds further reduces Vilamb's overhead to 0.1–6%. Similar to Figure 1, Vilamb offers 3–5× higher throughput than Pangolin at high insert rates for all five of Intel's PMDK key-value stores. By protecting the clean pages from firmware-bug-induced corruption, Vilamb increases the MTTDL over No-Redundancy. For example, Vilamb with a 1 sec system-redundancy update period increases Redis' MTTDL by 15× and 74× over No-Redundancy for a write-heavy and ready-heavy YCSB workload, respectively. Detailed timing breakdowns with fio microbenchmarks and battery cost analysis confirm Vilamb's design decisions.

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

2 Background and Related Work

This section provides background on direct-access (DAX) NVM and system-redundancy, and the challenges that DAX poses for maintaining system-redundancy. It then describes the solution space and how Vilamb and related work fit into it.

2.1 Direct-Access (DAX) NVM

NVM refers to a class of memory technologies that have access latencies comparable to DRAM and that retain their contents across power outages like disks. Various NVM technologies, such as 3D-XPoint [1, 27], Memristors [11], PCM [39, 54], and battery-backed DRAM [10, 15], are either already in-use or expected to be available soon. In this paper, we focus on NVM that is accessible like DRAM DIMMs rather than like a

[1] Vilamb means delay in Hindi.

Solution	Coverage Guarantees	Performance Overhead	Programming Model	Specialized Hardware Requirement
Pangolin [69]	Strong	Medium-to-High	Restrictive	None
Tvarak [33]	Strong	Negligible	Non-Restrictive	Yes
Vilamb	Configurable	Configurable	Non-Restrictive	None

Table 1: Solutions for DAX NVM system-redundancy and their trade-offs.

disk [45]. 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. Although applications can continue to access NVM via conventional FS interface, doing so incurs the overhead of system calls, and (potentially) data copying and inefficient general-purpose file system code [14, 66, 19, 65, 61, 30].

The DAX interface to NVM eliminates system software overheads, enabling applications to leverage raw NVM performance. With DAX, applications map NVM pages into their address spaces and access persistent data via load and store instructions. File systems that map a NVM file into the application address space (bypassing the page cache) on a `mmap` system call are referred to as DAX file systems and said to support DAX-mmap [40, 19, 67]. DAX is widely used for adding persistence to conventionally volatile in-memory DBMSs [41, 56, 70, 52] and is poised as the "killer use-case" for NVM.

DAX-mmap helps applications realize NVM performance benefits, but requires careful reasoning to ensure data consistency. Volatile processor caches can write-back data in arbitrary order, forcing applications to use cache-line flushes and memory fences for durability and ordering. Transactional NVM access libraries ease this burden by exposing simple transactional APIs to applications and ensuring consistency on their behalf [26, 13, 62, 24, 8]. Alternatively, the system can be equipped with enough battery to allow flushing of cached writes to NVM before a power failure [44, 72, 49]; our work assumes this option.

2.2 System-Redundancy

Many production storage systems implement system-redundancy, in the form of FS level page checksums and cross-page redundancy, to protect against firmware-bug-induced data corruption [58, 53, 71, 21]. Device firmwares are susceptible to bugs, like any software, because of their complex functionalities, such as address translation and wear leveling. A class of these bugs, namely lost write bugs and misdirected read or write bugs, can cause data corruption [53, 6, 29, 7, 60]. Lost write bugs cause the firmware to incorrectly consider a write as completed without actually writing the data on to the device media. Misdirected read or write bugs cause the firmware to access (read or write) data at a wrong location on the device media.

Firmware bugs can corrupt data that an application is actively accessing as well as data at rest. An example of a firmware bug affecting actively accessed data would be a misdirected read bug that causes the firmware to return incorrect data for an application read. On the other hand, lost write or address mapping bugs that are triggered when the firmware is performing wear-leveling could corrupt data at rest.

Storage systems can detect and recover from firmware-bug-induced corruption using system-redundancy [53, 71, 43]. For example, a FS can store and access page checksums separately from the data, making it unlikely for a firmware bug to affect both the data and its FS-level checksum in the same manner. An FS-level checksum mismatch can then flag firmware-bug-induced corruption, which the FS can recover from by using cross-page parity.

Many storage systems implement system-redundancy in addition to a variety of other fault-tolerance mechanisms [34, 28, 48, 21, 67, 71, 57, 23, 38]. In particular, storage systems implement system-redundancy even in the presence of device-level error correcting codes (ECCs) [68, 9, 35]. ECCs are designed for, and effective against, random bit flip induced corruption. However, they are ineffective against most firmware-

bug-induced corruption, because they are computed, stored, and accessed as a single unit with the data at a very low level of the device's firmware or hardware.

2.3 System-Redundancy for DAX NVM

Production NVM storage deployments will require similar levels of fault-tolerance as conventional storage deployments, including system-redundancy. Unsurprisingly, recently proposed NVM storage system designs include system-redundancy [67, 50, 69, 33]. Among these proposals, file systems like Nova-Fortis [67] and Plexistore [50] implement system-redundancy only for data that is accessed via the FS interface.

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

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

Cache-line Granular Writes: Incongruence in the size of DAX writes and the size of pages over which system-redundancy is usually maintained increases the overhead of maintaining system-redundancy. Most storage systems maintain system-redundancy over sizeable blocks (e.g., 4K page checksums) for space efficiency. Cache-line granular writes require reading (at least) an entire page to update the system-redundancy. Whereas RAID systems solve a similar "small write" problem by reading the data before updating it [47], a DAX NVM storage system software cannot use this solution. As discussed above, direct access to NVM bypasses system software, prohibiting the use of pre-write values for incremental system-redundancy updates.

2.4 Related Work: Solution Design Space

Table 1 summarizes the design space of DAX NVM system-redundancy solutions and the tradeoffs among the three options (including Vilamb) in the toolbox.

Pangolin [69] is a user-space library that maintains DAX NVM system-redundancy synchronously by requiring applications to explicitly inform it about their data updates; applications piggyback these notifications on Pangolin's transactional interface. Pangolin offers strong coverage (immediate system-redundancy updates and verification) and does not require any specialized hardware resources (because it is a software-based solution). Pangolin addresses the mismatch of fine-grained DAX updates with large checksum ranges by requiring explicit object definitions and maintaining per-object checksums instead of per-page checksums.

Pangolin is well-tuned, including several overhead-reducing mechanisms, making it the state-of-the-art for an in-line software-only solution. Yet, Pangolin still incurs significant performance overhead (up to 80%) in many cases. Fundamentally, Pangolin's synchronous system-redundancy update design requires updating system-redundancy at the same rate at which an object is being modified; this becomes costly for the high update rates enabled by NVM. Pangolin's per-object checksums also incur higher space overhead for small data objects. Also, importantly, Pangolin only works for applications that can be and are modified to use its object-based transactional interface. Applications that manage NVM data themselves using other data models, such as NVM-optimized databases [3], may not be easily fit to Pangolin's interface.

Tvarak [33] is a hardware controller co-located with the last level cache (LLC) that the FS can offload system-redundancy maintenance work onto. Tvarak is able to identify data updates by the virtue of being interposed in the data path. Tvarak offers synchronous system-redundancy updates and verification, does not restrict applications to any specific library/API, and is low-overhead. However, it requires specialized hardware resources, including a controller, on-controller cache, and shared LLC partitions. The need for dedicated (and newly proposed) hardware resources implies that Tvarak is not available for immediate use,

and may not be part of commodity servers for many years. Further, Tvarak introduces cache-line granular checksums for DAX-mapped data, increasing the space overhead.

Prioritizing strong coverage at the expense of performance and a restrictive programming model (with Pangolin [69]), or cost and near-term availability (with Tvarak [33]), will not be the preferred choice for all applications. Many applications prioritize performance and use storage systems wherein some of the fault-tolerance mechanisms (e.g., remote replication or even persistence) are asynchronous—the fault-tolerance is still desired, and the more coverage the better, but not at a high performance cost [16, 48, 34, 28].

Vilamb is a software library that embraces an asynchronous approach to updating system-redundancy for updated data. Like other asynchronous redundancy-update approaches, it identifies and completes required system-redundancy updates in the background. Indeed, it does both aspects (identifying and updating) outside the critical path of application accesses. As such, Vilamb can provide low-overhead DAX NVM system-redundancy. Also, Vilamb does not impose any programming model restrictions and does not require any specialized hardware resources. But, Vilamb reduces the data coverage guarantees by delaying system-redundancy updates. Specifically, recently modified pages may not be covered when a firmware bug affects them. So, Vilamb can be a good option when applications desire high performance and/or are not a good fit for Pangolin-like API. and view partial system-redundancy coverage is as better than none.

3 Vilamb Design and Implementation

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

3.1 Asynchronous System-Redundancy

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

Figure 2 illustrates how Vilamb reduces work for per-page checksums (cross-page parity is not shown in the example, but is updated at the same time as the page checksum). The figure shows a DAX NVM page and its checksum; the checksum can either be up-to-date (✓) or outdated (x). In the initial state, the checksum is up-to-date with the data. The first write to the page makes the checksum stale. Instead of updating the checksum immediately, Vilamb delays the update until after two more writes. By delaying the update Vilamb performs a single checksum (and parity, not shown in the figure) computation, instead of three.

Vilamb scrubs the data using a separate background thread to detect data corruption. Upon mismatch between the page data and checksum for a clean page, Vilamb raises an error and halts the program. The OS can recover corrupted pages using the parity pages, with potential re-mapping to different physical pages [67, 69].

3.2 Repurposing Dirty Bits

The conventional use-case of dirty bits is irrelevant for DAX NVM pages, making them available for re-purposing. 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 [19, 40]. Consequently,

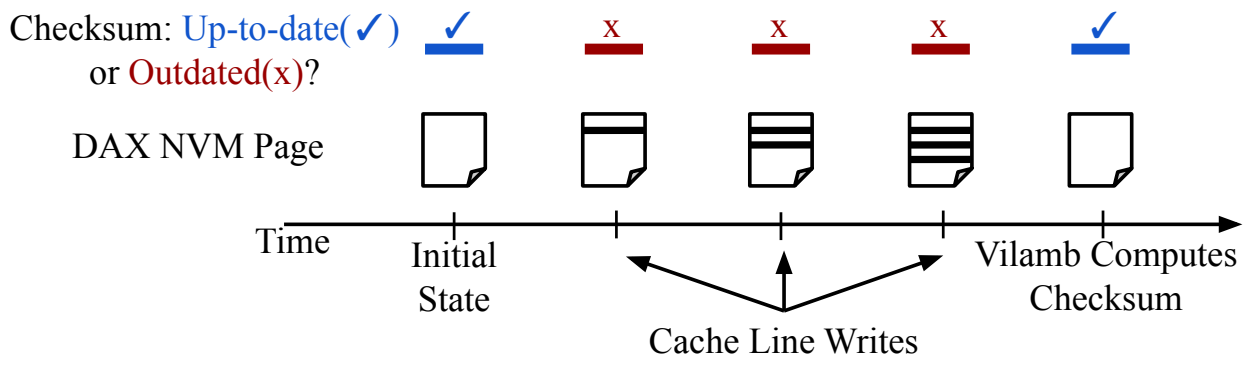


Figure 2: Delayed Checksum Computation Example – By computing per-page checksums asynchronously, Vilamb amortizes the computation overhead over multiple cache-line writes to the same NVM page.

even though each mapped page has a corresponding dirty bit, the conventional semantic of these dirty bits is irrelevant because the pages already reside in persistent NVM storage.

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

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

3.3 Failure Coverage

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

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

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

- For the first scenario, consider an application write that is evicted from the CPU caches to the NVM device, is stored in the on-device write-back cache, and then lost by the firmware before Vilamb updates the corresponding page's checksum. This would lead to a silent corruption because Vilamb would use the incorrect (old) data to compute the checksum.

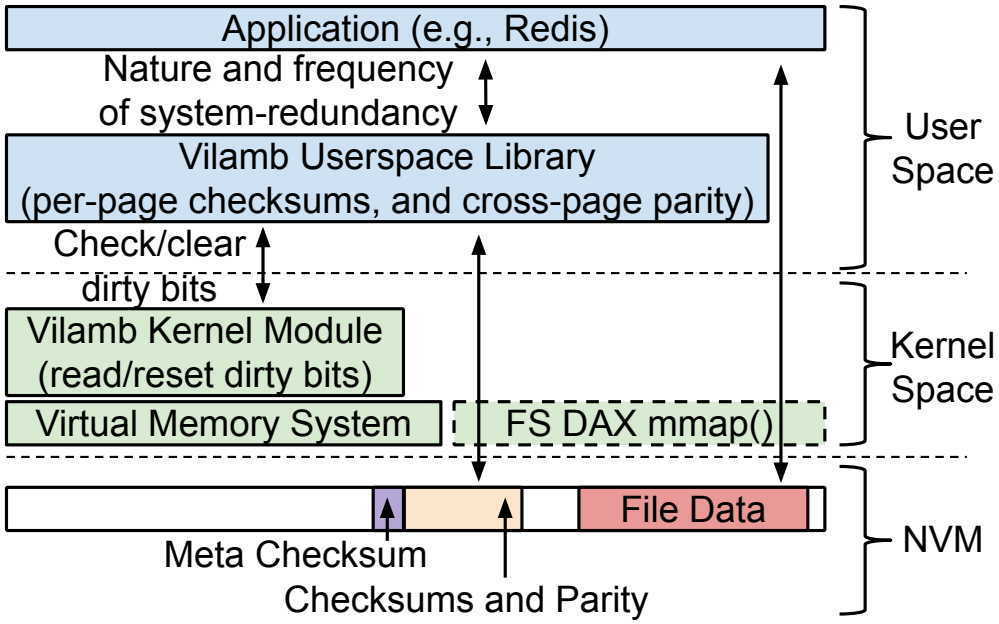


Figure 3: Vilamb's Implementation: The user space library performs the checksum and parity computations with a period that is set by the application. The kernel module checks and clears the dirty bits when requested by the user space library.

- For the second scenario, consider that Vilamb updates the page's checksum before the firmware bug is triggered (i.e., while the data is in the CPU caches or in the on-device cache). Vilamb would update the checksum correctly in this scenario and detect the subsequent corruption because of a data checksum mismatch at a later point.

- For the third scenario, imagine the bug affects a clean page while the firmware is performing wear leveling. Vilamb would be able to detect this data loss in its scrubbing thread.

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

Power Failures: Vilamb avoids any inconsistencies between data and its system-redundancy by ensuring that the system-redundancy is made up-to-date if there is a power failure. To that end, Vilamb leverages battery backups that are common in production environments [18, 31, 32, 23, 22, 63, 37]. Conventional storage systems use batteries to flush DRAM to a persistent medium upon a power failure [18, 31, 32, 23]. NVM does not need batteries to make its contents persistent, because they are already persistent. Vilamb instead leverages the battery backup to update system-redundancy upon a power failure, ensuring that no pages are left uncovered. Given that batteries are also used to address other issues, including brief power losses and spikes [46], we believe that Vilamb can exploit them for updating system-redundancy.

NVM DIMM Failures or Machine Failures: Vilamb's system-redundancy is not intended for protection against DIMM or machine failures; the storage system can protect against these using remote replication [70, 59]. Being a machine-local fault-tolerance mechanism, system-redundancy, independent of its implementation, is ineffective against machine failures. For DIMM failures, Vilamb's asynchronous system-redundancy design makes it unable to reconstruct the fraction of the pages in the failed DIMM that belonged to a stripe with outdated system-redundancy. Although the storage system could still recover a large fraction of the data (Section 4.8), it would need other redundancy to recover the remaining data.

We implement Vilamb as a user-space library. The library exposes an API that applications can use to

configure the nature of system-redundancy (e.g., type of checksum and number of pages in a stripe) and its update frequency. The library uses a periodic background thread that checks and clears the dirty bits using new system calls that we implement, and performs the system-redundancy updates for the dirty pages. Our implementation uses a stripe size of five pages by default, with four consecutive data pages and one parity page. The stripes are statically determined at the time of initialization. Figure 3 shows the components of our implementation.

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

Batched Checking and Clearing: Vilamb checks and clears dirty bits for multiple NVM pages (e.g., 512 in our experiments) as a batch for efficiency. Both checking and clearing of dirty bits require a system call and traversing the hierarchical page table; clearing dirty bits further requires invalidating the corresponding TLB entries. Each of these is a costly operation, as evinced by prior research [2], and demonstrated by our experiments (Section 4.6). Batching allows pages to share the system call, fractions of the page table walk, and the TLB invalidation. We found that batching reduced the amount of time spent in checking/clearing dirty bits by up to two orders of magnitude.

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

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

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

Leveraging Hardware Support: Our implementation of Vilamb leverages hardware-support whenever possible. We use CRC-32C checksums and employ the `crc32q` instruction when available. Similarly, we use SIMD instructions for computing the parity whenever possible (e.g., by operating on 256-byte words in our experiments). We never flush cache lines for persistence because we assume battery-backed servers. We do, however, use fences to ensure ordering between updates. For example, the fence at Line 5 ensures that the shadow copy of the dirty bits and current batch's starting page number writes are completed before the dirty bits are cleared. Similarly, the fence at Line 19 ensures that system-redundancy is written before

Algorithm 1: System-Redundancy 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;
    memoryFence;
    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();
```

the dirty bits' shadow copy is cleared. We extend the same performance benefits (e.g., no cache line flushes and SIMD parity computations) to the alternatives that we compare Vilamb with in our evaluation.

4 Evaluation

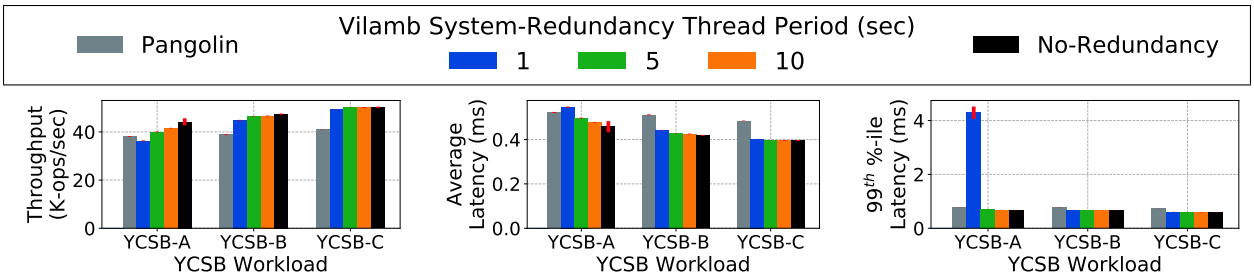
This section evaluates Vilamb and compares it to No-Redundancy and Pangolin, using eight macro- and micro-benchmarks. No-Redundancy serves as the baseline, providing the best performance but not implementing any system-redundancy. Pangolin is a state-of-the-art userspace library that updates system-redundancy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

We obtained Pangolin's code from the authors and run it with checksum and parity updates enabled but checksum verification disabled (referred to as Pangolin-MLPC in the Pangolin paper [69]). We run Vilamb also with checksum and parity updates enabled and checksum verification disabled. As shown in the evaluation of Pangolin [69], and confirmed by our experiments, checksum verification via scrubbing at reasonable frequencies incurs negligible overhead. Pangolin can also verify checksums on object reads, which Vilamb cannot, but doing so reduces throughput by up to 50% for large objects [69].

Unless mentioned otherwise, Vilamb uses a 512-page batch size for checking/clearing dirty bits. To accurately quantify Vilamb's overheads, we pin it to the same core(s) as the application. For single threaded applications such as Redis, this means that the application and Vilamb run on the same logical core (i.e., same hyper-thread). Each data point in our results is an average of three runs with root mean square error bars. We use a dual-socket Intel Xeon Silver 4114 machine with Linux 4.4.0 kernel for our experiments. The system has 192 GB DRAM, from which we use 64 GB as emulated NVM [51].

4.1 Key Evaluation Takeaways

Key takeaways from our evaluation include:



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

Figure 4: YCSB with Redis – Throughput and read latency of YCSB workloads with Redis.

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

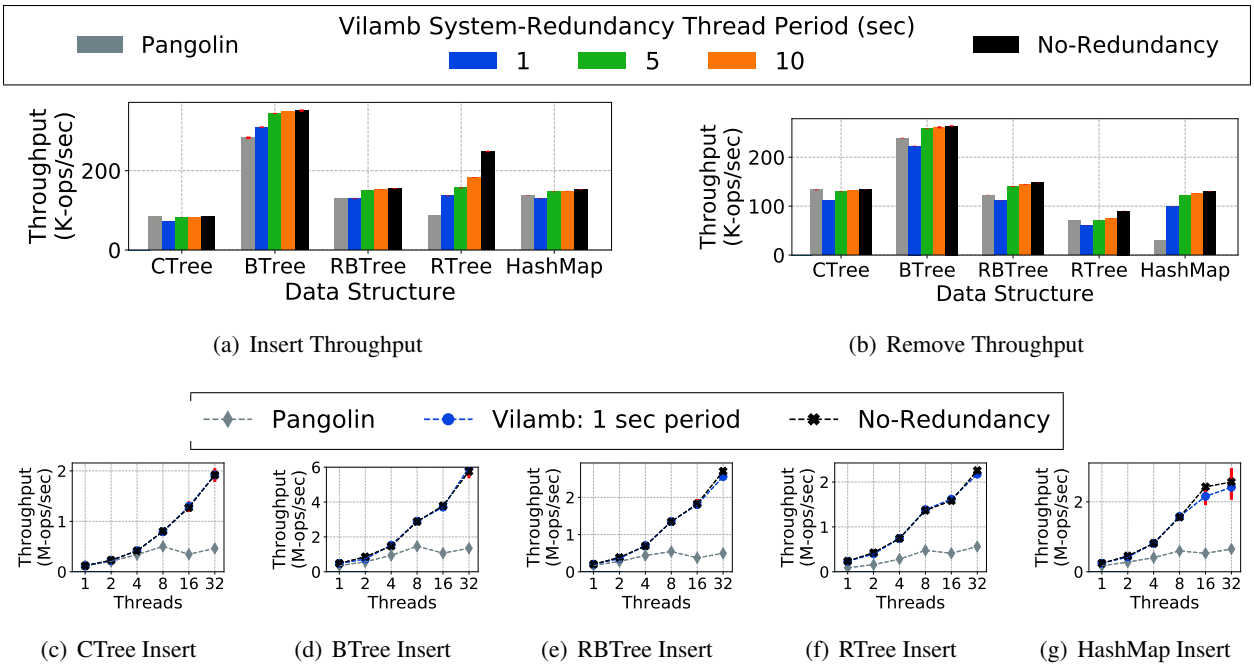
4.2 YCSB with Redis

Redis [55] is a widely used open-source NoSQL DBMS. We modify it to use a DAX NVM file for its data heap. Our implementation uses the `libpmemobj` library [25] from the Intel persistent memory development kit (PMDK) [26] for No-Redundancy.

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

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

Results: Figure 4 presents throughput and read latencies. Vilamb reduces the throughput, in comparison to No-Redundancy, by 0.1–6% for a system-redundancy update period of 10 sec and by 1.6–17% for a period of 1 sec. Increasing the delay for system-redundancy updates improves Vilamb's performance because it performs fewer system-redundancy updates and hogs less CPU. With aggressive system-redundancy updates every second, Vilamb increases the tail latency for YCSB-A because it stalls Redis while updating



(a) Insert Throughput

(b) Remove Throughput

(c) CTree Insert

(d) BTree Insert

(e) RBTree Insert

(f) RTree Insert

(g) HashMap Insert

Figure 5: PMDK Key-Value Stores – Throughput for insert-only, remove-only benchmarks with different PMDK key-value stores.

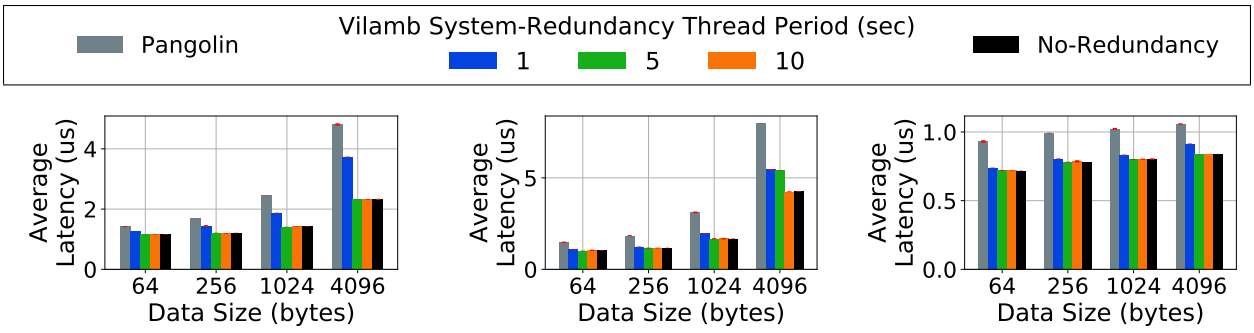
system-redundancy on the same core. This effect can be mitigated if Vilamb and Redis were to run on separate cores.

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

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

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

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



(a) Allocation (b) Overwrite (c) Deallocation

Figure 6: NVM Transaction Latencies – Latencies for transactional allocation, overwriting, and deallocation.

4.3 PMDK Key-Value Stores

Intel persistent memory development kit (PMDK) [26] implements NVM-optimized key-value stores and includes performance benchmarks.

Experimental Setup: Similar to Pangolin [69], we use insert-only, and remove-only benchmarks for five key-value stores: Crit-Bit Tree (CTree), BTree, Red-Black Tree (RBTree), Range Tree (RTree) and chaining hashmap (HashMap). We first re-create the experiment and results from Pangolin [69] with a single-thread that performs 5 million operations. We then use multiple threads (1 to 32) with 100,000 operations per thread.

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

Results: Figures 5(a) and 5(b) show the throughput for the insert-only and remove-only workloads when using a single thread for the key-value store. Pangolin's overheads are similar to those reported in their paper [69]. Vilamb's performance improves with increasing delay in system-redundancy updates. Of the five key-value stores, both Pangolin and Vilamb have the highest overhead in comparison to No-Redundancy for RTree because RTree's insertion touches the largest amount of data. For the remove-only workload, Pangolin outperforms Vilamb with 1 sec system-redundancy update period because removing objects touches only a small amount of data and Pangolin can efficiently update system-redundancy using the diffs for small data.

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

4.4 NVM Transaction Microbenchmarks

Pangolin [69] introduced micro-benchmarks to measure the latency of transactional operations (allocation, overwrite, and deallocation), and to measure the scalability of overwriting NVM regions with multiple threads.

Experimental Setup: We perform each transactional operation (allocation, overwrite, deallocation) 1

million times for different sized objects in a single thread and report the average latency. We use an NVM file of 10 GB for this. For scalability, we increase the number of threads with each thread overwriting a 64-byte and 4 KB regions 200,000 times.

Results: Figure 6 shows the latency for performing the transactional operations using a single thread. For 64-byte objects, Pangolin incurs 23%, 44%, and 30% higher latency than No-Redundancy for allocation, overwrite, and deallocation, respectively. In contrast, Vilamb with a system-redundancy update period of 1 sec increases the corresponding latencies by only 9%, 5%, and 3%; increasing the system-redundancy update period further reduces Vilamb's latencies. Increasing the object sizes increases the latency for all configurations, because more data is touched (except for deallocation, in which only metadata is updated). However, even for 4 KB objects, Vilamb with a system-redundancy update period of 1 sec has 13%–31% lower latencies than than Pangolin.

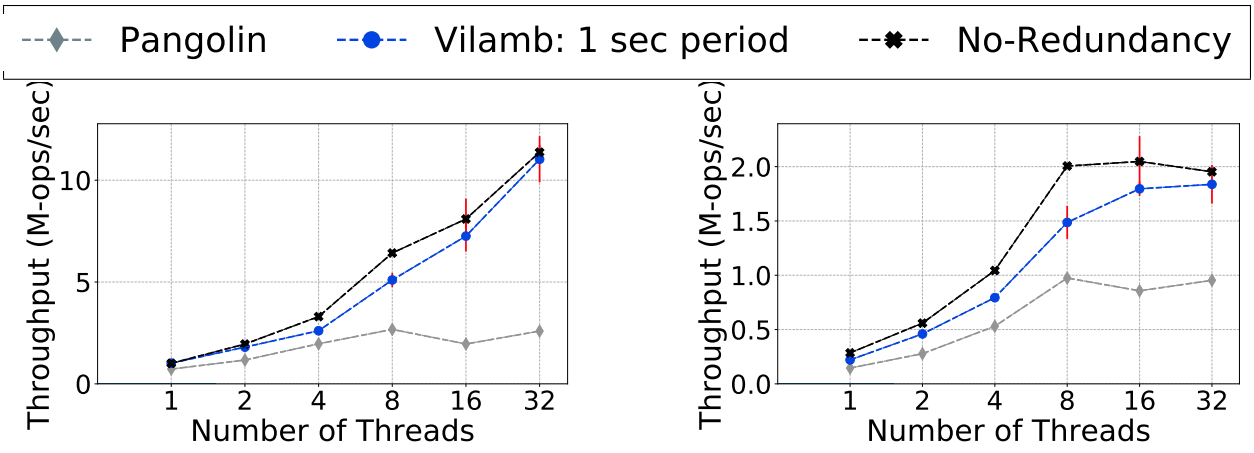
Figure 7 shows the throughput for overwriting 64-byte and 4 KB regions with increasing number of threads. Vilamb scales close to No-Redundancy, with only up to 25% lower throughput. In contrast, Pangolin has up to 77% lower throughput. Pangolin's experiments with real NVM (in contrast to our DRAM-based emulation) showed that No-Redundancy performance does not scale well beyond 8 threads because of NVM's limited bandwidth [69]. However, even with 8 threads Vilamb's throughput is double of Pangolin's. As NVM performance improves and gets closer to DRAM performance, the benefits of Vilamb's asynchronous redundancy maintenance will become more pronounced. We also evaluated overwriting with other intermediate data sizes (256 and 1024 bytes) and obtained similar trends.

4.5 Fio Microbenchmarks

This section evaluates Vilamb's performance using fio [5] microbenchmarks. We cannot evaluate Pangolin using fio because fio's NVM engine [20] does not use object based transactions. Rather fio treats the entire DAX-mapped file as a raw sequence of bytes. This illustrates Pangolin's programming model restriction. Applications that manage DAX-mapped data themselves, either as raw data as in fio microbenchmarks or in more complex fashion like NVM databases [3], can benefit from Pangolin only if they can be and are modified to use its APIs.

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

Results: Figure 8 shows the throughput for the two workloads with three access patterns each. For write-only workloads, Vilamb reduces throughput by 0.5–56% with higher overheads for more frequent system-redundancy updates. Vilamb's overheads are highest for the random workload and lowest for the sequential workload; sequential workloads offer the best opportunity to reduce computations, because successive cache line writes belong to the same page. Even for random workloads, the overhead is only 10% with a system-redundancy update delay of 60 seconds. Vilamb reduces the throughput by only up to 3% for read-only workloads, demonstrating the efficacy of its checking of dirty bits. Vilamb's througput is higher than No-Redundancy for the read-only sequential workload with an update period of more than 10 seconds; this is an artifact of the experimental setup. While checking for dirty bits, Vilamb populates the page table entries and reduces the number of soft page faults. The performance benefit of reduced soft page faults outweigh the overhead of checking the dirty bits infrequently (i.e., with a period of more than 10 seconds). This anamoulous inversion of performance can be resolved by pre-populating the page table entries for Vilamb as well.



(a) 64 Byte Writes

(b) 4096 Byte Writes

Figure 7: NVM Overwrite Throughput

4.6 Cost of Checking/Clearing Dirty Bits

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

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

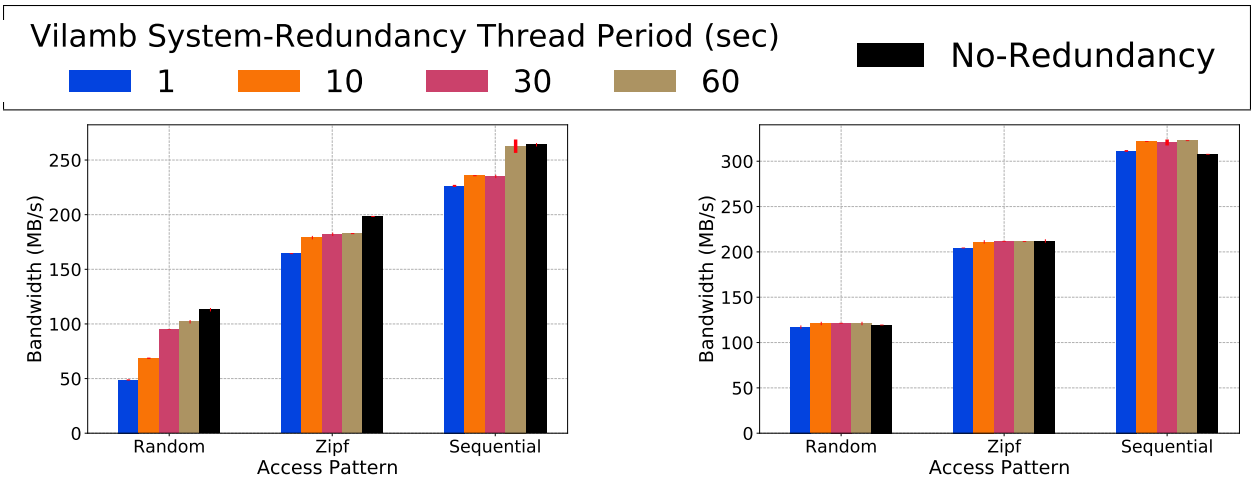
Results: Figure 9(a) presents the time spent in various components of checking and clearing dirty bits. The batch size is set to 512 pages for this experiment. Doubling the file size, and consequently the total number of pages, roughly doubles the amount of time spent in each of the components. This is because the number of system calls, page walks, and reads of the dirty bits are all directly proportional to the total number of pages. The number of pages for which the dirty bit is cleared and the number of TLB invalidations depend on the workload's access pattern. For the uniform random access workload, these are also directly proportional to the total number of pages.

Figure 9(b) presents the impact of batch size for a 16 GB file. As the batch size increases, the time spent in checking/clearing dirty bits decreases with diminishing marginal returns. This decrease is because the number of system calls reduce and larger fractions of the page table walks are shared between the pages in the same batch. The benefits are diminishing with increasing batch size, because of the fixed cost of reading all the dirty bits and resetting the ones that are found to be set.

4.7 Battery Capacity Requirements

This section analyzes the cost of batteries required for Vilamb to update the system-redundancy after a power failure for various workloads. We consider two kinds of batteries: ultra-capacitors that cost $2.85/KJ [44, 64], and lithium-ion batteries that cost $0.02/KJ [64, 46]. Conventionally, datacenters use lithium-ion batteries; modern datacenters additionally use ultra-capacitors because of their higher energy efficiency and density [64]. We consider servers with 500W [64] power usage.

For Redis with the write-heavy workload YCSB-A, one iteration of Vilamb's system-redundancy updates takes 143 ms when performed every second and 562 ms when performed every 10 seconds. These correspond to less than 1 KJ of energy required, i.e. the cost would be less than $2.85 when using ultra-



(a) Write Only Workload

(b) Read Only Workload

Figure 8: Fio Microbenchmarks – Throughputs for write-only and read-only workloads with different access patterns.

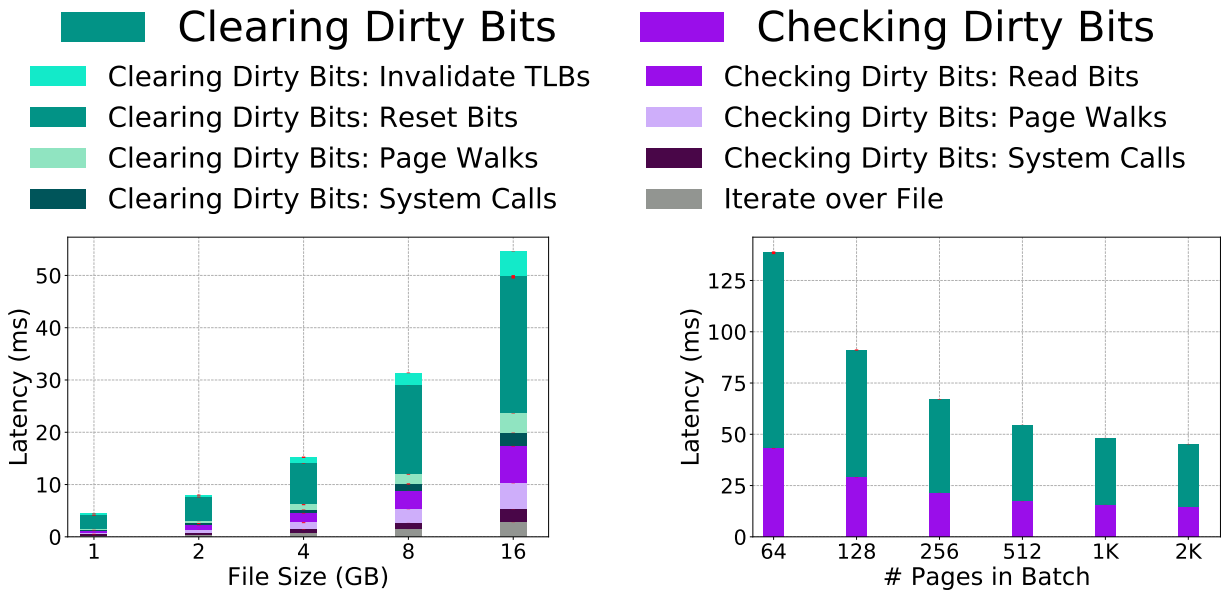
capacitors and less than $0.02 when using the conventional lithium-ion batteries. This is the case for all PMDK key-value stores except RTree as well. For RTree, because of its sparse and large writes, Vilamb can require up to 5 seconds to update the system-redundancy upon a power failure, requiring 2.5 KJ of energy. This corresponds to $7.2 in ultra-capacitor cost or $0.05 lithium-ion battery cost. For fio, even with the adversarial random write workload with a system-redundancy update period of every 60 seconds, Vilamb requires only 4.5 seconds after a power failure. This translates to 2.25 KJ of required energy and $6.4 in ultra-capacitor cost or $0.04 in lithium-ion battery cost. The battery requirement, and the associated cost, can be further reduced by limiting the number of pages that can be dirty (i.e., with outdated system-redundancy) using Viyojit's [32] design.

4.8 Reliability Analysis

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

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

We use the above to compute the increase in the MTTDL with Vilamb over No-Redundancy for the various applications and workloads described in Section 4. Workload access patterns, i.e., the rate and locality of their data updates determine the number of vulnerable stripes. We emperically measure the average number of vulnerable stripes for the various workloads and use that to compute the increase in MTTDL. For Redis, Vilamb with a system-redundancy update period of 1 sec increases the MTTDL by 15× for the write-heavy workload YCSB-A and 74× for the ready-heavy workload YCSB-B. Increasing the delay reduces the MTTDL, because a larger fraction of data remains dirty (e.g., 21× and 13× for YCSB-



(a) Breakdown of Time Spent

(b) Impact of Batch Size

Figure 9: Cost of Checking/Clearing Dirty Bits – 9(a) shows the time spent in each component of checking/clearing dirty bits for a batch size of 512 pages and increasing file sizes. 9(b) shows that increasing the batch size reduces the time spent in checking/clearing dirty bits with diminishing returns.

B with 5 sec and 10 sec period, respectively). For PMDK's key-value stores, Vilamb increases the MTTDL by up to two orders of magnitude (e.g., 112× for RBTree insert-only workload with 32 threads).

5 Conclusion

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

References

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

[2] 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.

[3] 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 Inter-*

national Conference on Management of Data, SIGMOD '15, pages 707–722, New York, NY, USA, 2015. ACM.

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

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

[6] 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.

[7] 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.

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

[9] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, 105(9):1666–1704, Sep. 2017.

[10] 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.

[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] 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.

[16] 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.

[17] 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.

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

[21] 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.

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

[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] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

[30] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. SplitFS: A File System that Minimizes Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Ontario, Canada, October 2019.

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

[32] 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.

[33] Rajat Kateja, Nathan Bechmann, and Greg Ganger. Tvarak: Software-managed hardware offload for dax nvm storage redundancy. *Parallel Data Lab Technical Report CMU-PDL-19-105*. https://www.pdl.cmu.edu/PDL-FTP/NVM/CMU-PDL-19-105.pdf.

[34] 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.

[35] Taeho Kgil, David Roberts, and Trevor Mudge. Improving nand flash based disk caches. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 327–338, Washington, DC, USA, 2008. IEEE Computer Society.

[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] Vasileios Kontorinis, Liuyi Eric Zhang, Baris Aksanli, Jack Sampson, Houman Homayoun, Eddie Pettis, Dean M. Tullsen, and Tajana Simunic Rosing. Managing Distributed Ups Energy for Effective Power Capping in Data Centers. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 488–499, Washington, DC, USA, 2012. IEEE Computer Society.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

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

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

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

[53] 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.

[54] 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.

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

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

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

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] Di Wang, Sriram Govindan, Anand Sivasubramaniam, Aman Kansal, Jie Liu, and Badriddine Khessib. Underprovisioning Backup Power Infrastructure for Datacenters. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 177–192, New York, NY, USA, 2014. ACM.

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

[65] 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.

[66] 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.

[67] 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.

[68] 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.

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

[70] 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.

[71] 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.

[72] 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.