Software Wear Management for Persistent Memories

Vaibhav Gogte, *University of Michigan;* William Wang and Stephan Diestelhorst, *ARM;* Aasheesh Kolli, *Pennsylvania State University and VMware Research;* Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch, *University of Michigan*

https://www.usenix.org/conference/fast19/presentation/gogte

This paper is included in the Proceedings of the
17th USENIX Conference on File and Storage Technologies (FAST '19).

February 25–28, 2019 • Boston, MA, USA

978-1-939133-09-0

Software Wear Management for Persistent Memories

Vaibhav Gogte[1], William Wang[2], Stephan Diestelhorst[2], Aasheesh Kolli[3,4], Peter M. Chen[1], Satish Narayanasamy[1], and Thomas F. Wenisch[1]

[1]University of Michigan
[2]ARM
[3]Pennsylvania State University
[4]VMware Research

Abstract

The commercial release of byte-addressable persistent memories (PMs) is imminent. Unfortunately, these devices suffer from limited write endurance—without any wear management, PM lifetime might be as low as 1.1 months. Existing wear-management techniques introduce an additional indirection layer to remap memory across physical frames and require hardware support to track fine-grain wear. These mechanisms incur storage overhead and increase access latency and energy consumption.

We present *Kevlar*, an OS-based wear-management technique for PM that requires no new hardware. Kevlar uses existing virtual memory mechanisms to remap pages, enabling it to perform both *wear leveling*—shuffling pages in PM to even wear; and *wear reduction*—transparently migrating heavily written pages to DRAM. Crucially, Kevlar avoids the need for hardware support to track wear at fine grain. Instead, it relies on a novel *wear-estimation* technique that builds upon Intel's Precise Event Based Sampling to approximately track processor cache contents via a software-maintained Bloom filter and estimate write-back rates at fine grain. We implement Kevlar in Linux and demonstrate that it achieves lifetime improvement of 18.4× (avg.) over no wear management while incurring 1.2% performance overhead.

1 Introduction

Forthcoming Persistent Memory (PM) technologies, such as 3D XPoint [3, 46], promise to revolutionize storage hierarchies. These technologies are appealing in many ways. For example, they are being considered as cheaper, higher capacity and/or energy-efficient replacements for DRAM [5, 64, 87, 119], low-latency and byte-addressable persistent storage [22, 23, 83, 101], and even as hardware accelerators for neural networks [89, 94]. We focus on systems with heterogeneous memory—with both DRAM and PM connected to the memory bus. Such systems may use PM for persistent data storage or to replace some or all of DRAM with a cheaper/higher-capacity technology.

Nevertheless, PM's limited write endurance [21, 64, 87, 114, 119] may hinder adoption. Just like erase operations wear out Flash cells, PM devices may also wear out after a certain number of writes. The expected PM cell write endurance varies significantly across technologies. For example, a phase-change memory is expected to endure $10^7 - 10^9$ writes [64, 85, 87] while resistive RAM may sustain over 10^{10} writes [106]. So, system developers must consider PM cell write frequency and manage wear to ensure memory endures for the expected system lifetime.

PM wear-management techniques employ *wear leveling*, spreading writes uniformly over all memory locations, and/or *wear reduction*, reducing the number of writes with additional caching layers [26, 64, 85, 88, 92, 119]. Unfortunately, prior techniques rely on various kinds of hardware support. Some proposals [85, 119] add an additional programmer-transparent address translation mechanism in the PM memory controller. These mechanisms periodically remap memory locations to uniformly distribute writes across the PM. Other techniques [26, 88, 114] perform wear reduction by remapping contents of frequently-written PM page frames to higher-endurance DRAM. Such techniques depend on hardware support to estimate wear, for example, via per-page counters or specialized priority queues/monitoring in the memory controller. Unfortunately, PM-based mechanisms [26, 88, 114] that rely on higher-endurance but volatile DRAM to reduce wear do not support applications [77] that require crash consistency when using PM as storage.

The indirection mechanisms proposed for PMs are analogous to the translation layer [33, 58, 65] in Flash firmware, which perform functionalities such as garbage collection [33, 58, 109] and out-of-place updates [33, 58, 65, 67] in addition to wear leveling, and incur high erasure latency [33, 53, 67]. Additional translation layers increase design complexity and incur higher access latency and power/energy consumption. Indeed, recent work [12, 15, 40, 41, 50, 66, 82, 115] aims to eliminate complexity and overhead associated with a Flash translation layer by combining its features in either the virtual

memory system in the OS [12, 15, 40, 41, 115], or in file-system applications [15, 50, 66, 82]. We would prefer to avoid additional indirection mechanisms for byte-addressable PMs, which have lower access latency and offer a direct load/store interface.

We note that the OS already maintains a mapping of virtual to physical memory locations and that these mappings can be periodically updated to implement wear management without an additional translation layer. We build upon virtual memory to implement *Kevlar*, a software wear-management system for fast, byte-addressable persistent memories. Kevlar performs both wear leveling, by reshuffling pages among physical PM frames, and wear reduction, by judicious migration of wear-heavy pages to DRAM, to achieve a configurable lifetime target.

A critical aspect of wear management is to estimate the wear to each memory location. Existing hardware tracks PM writes only at the granularity of memory channels—too coarse to be useful for wear management. Tracking PM writes at finer granularity is complicated by write-back hardware caches; an update to a memory location leads to a PM write only when a dirty cache block is evicted from the processor's caches.

Kevlar relies upon a novel, low-overhead *wear-estimation* mechanism by using Intel's Precise Events Based Sampling (PEBS) [44], which allows us to intercept a sample of store operations. Kevlar maintains an approximate representation of hardware cache contents using Bloom filters [16], and uses it to estimate relative fine-grain writeback rates. We demonstrate that our estimation strategy incurs less than 1% performance overhead.

Kevlar enables wear management for applications that employ PMs for capacity expansion [5, 55, 88] and/or durability [77]. When a PM device is used for capacity expansion, Kevlar exploits memory device heterogeneity and migrates frequently updated PM pages to the neighboring DRAM—a system-level option that cannot be exploited by device-level wear-management schemes [85, 92, 119]. We show that migrating as few as 1% of pages from PM to DRAM is sufficient to achieve our target PM lifetime. For pages that require durability, Kevlar relies on reserve PM capacity and performs directed migrations of frequently written pages across the nominal and reserve capacity.

We implement Kevlar in Linux version 4.5.0 and evaluate its impact on performance and PM lifetime. To summarize, the contributions of Kevlar are:

- *Wear leveling:* We first develop an analytical framework to show that even a simple, wear-oblivious random page shuffling is sufficient to achieve near-ideal (uniform) wear over the memory device lifetime at negligible (< 0.1%) performance overhead. Unfortunately, even ideal wear leveling provides insufficient lifetime for lower-endurance PMs.
- *Wear estimation:* We demonstrate how to estimate wear at fine grain by using Intel's PEBS to approximate cache contents via a Bloom filter, thereby estimating the cache write-backs to each page. We show that this mechanism is 21.7× more accurate than naive write sampling.
- *Wear reduction:* We demonstrate Kevlar, which uses our wear-estimation technique to apply both wear leveling and wear reduction, reducing wear by migrating less than 1% of the application working set to neighboring DRAM (when durability is not needed) incurring 1.2% (avg.) performance overhead.

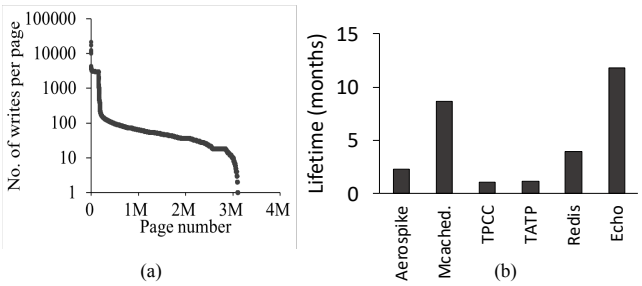


Figure 1: (a) **Pages sorted by number of writes (program entirety) in Aerospike**: There is a large disparity between most and least written pages. (b) **PM lifetime with no wear leveling**: The lifetime until 1% of pages sustain 10^7 writes can be as short as 1.1 months.

2 Background and Motivation

We briefly describe PM use cases and their drawbacks.

2.1 Persistent Memories (PMs)

Persistent memory technologies, such as Phase Change Memory [64, 87], Memristor [106], and Spin Torque Transfer RAM [111] are byte-addressable, achieve near-DRAM performance, and are denser and cheaper than DRAM. These characteristics allow systems to leverage PMs in exciting new ways. We focus on two well-studied use cases: (1) capacity expansion and (2) memory persistency.

Capacity expansion: Owing to their higher density and lower power consumption, PMs are projected to be cheaper than DRAM [5, 31, 55, 64, 87, 119] on a dollar per GB basis. Higher density enables greater peak capacity: Intel expects to soon offer servers with up to *6TB* of PM [3, 38]. System designers can use this capacity to manage larger in-memory data-structures [9, 42, 72].

Memory persistency: Since PMs are non-volatile, they blur the traditional distinctions between memory and storage. Recent research leverages PM non-volatility by accessing persistent data directly in memory via loads and stores [22, 23, 28, 36, 48, 52, 60, 61, 63, 77, 83, 101]. The byte-addressable load-store PM interface enables fined-grained accesses to persistent data and avoids the expensive serialization and de-serialization layer of conventional storage [54].

PM drawbacks: Whereas PMs exhibit many useful properties, they also have two key drawbacks. First, PM cells have limited write endurance. For example, PCM endures only 10^7 - 10^9 writes [85]. In contrast, DRAM endurance is essentially unbounded (> 10^{15} writes) [87]. Limited PM endurance may lead to rapid capacity loss for write-intensive applications. Figure 1(a) shows the disparity between writes seen by the hottest and coldest pages for Aerospike (see Section 5 for our methodology). Absent wear management, frequently written-back addresses wear out sooner, compromising lifetime. Figure 1(b) shows the lifetime until 1% of memory locations wear out in a device with a write endurance of 10^7 writes (such as PCM) under the write patterns of various applications assuming no efforts to manage wear. For example, we observe that TPCC can wear out a PCM memory device within 1.1 months.

Second, PM access latency and bandwidth, while close to DRAM, fall short [64, 87, 106]. So, applications sensitive to memory performance might still prefer DRAM. Prior works [5, 55, 84] mitigate this challenge by identifying hot/cold regions of applications' footprints and placing hot regions in DRAM and cold regions in PM. Unlike these works [5, 55, 84], we exploit memory device heterogeneity to improve device lifetime when PMs are employed for capacity expansion and/or memory persistency. To this end, we propose Kevlar, a wear-management mechanism to improve low-endurance PM device lifetime.

2.2 Wear-aware virtual memory system

Prior PM wear-management mechanisms [85–87, 92, 119] require an additional indirection layer in hardware to uniformly wear PM cells. However, these mechanisms suffer from several drawbacks. First, these mechanisms [85–87, 92] use volatile DRAM caches to reduce wear to PM. These mechanisms do not readily support applications [77] that rely on PM durability, since the volatile DRAM caches lose data upon power failure. Second, these mechanisms perform additional DRAM cache lookups and address translation for each memory access, delaying PM loads/stores. Third, wear leveling alone sometimes achieves PM lifetime of only 2.3 years (as shown later in Section 6.2)—lower than the desired system lifetimes. These device-level mechanisms are unable to exploit memory system heterogeneity for applications that employ PMs for capacity expansion.

We explore low-overhead OS wear-management mechanisms that can extend PM device lifetime to a desired target without any additional indirection layers. Indeed, our approach is analogous to similar ongoing efforts [12, 15, 40, 41, 50, 66, 82, 115] in Flash-based systems to identify and eliminate performance bottlenecks in the Flash translation layer (FTL). These works avoid FTL complexities and overheads by folding its features either into the virtual memory system [12, 15, 40, 41, 115], or into file system applications [15, 50, 66, 82]. Like these works, we aim to build PM wear-management into the virtual memory system. Note that, contrary to block-based access to Flash, PM updates arise from LLC write-backs. Unfortunately, there are no straightforward mechanisms to measure LLC write-backs directly at fine grain—a critical challenge that we solve in Kevlar.

3 Kevlar

We detail wear-management approaches in Kevlar.

3.1 Wear leveling

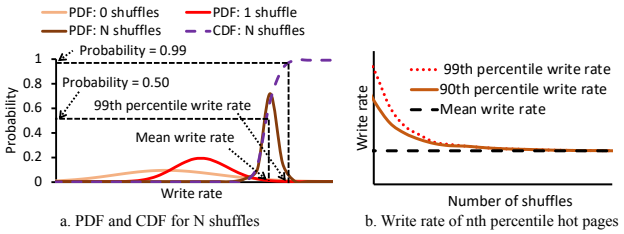
Modern OSes, such as Linux, manage memory via a paging mechanism to translate virtual to physical memory addresses. Linux manages the page tables used by the hardware translation mechanism, and already reassigns virtual-to-physical mappings for a variety of reasons (e.g., to improve NUMA locality).

Kevlar's *Wear-Leveling* (WL) mechanism uses existing OS support to periodically remap virtual pages to spread writes uniformly. Kevlar makes a conservative assumption that a write to a physical PM page modifies all locations within that page. Thus, Kevlar does not need an additional intra-page wear-leveling mechanism. We observe that periodic random shuffling of virtual-to-physical mappings—migrating each virtual page to a randomly selected physical page frame—is sufficient to uniformly distribute writes to PM provided shuffles are frequent enough. A key advantage of this approach is that it is wear oblivious—it requires no information about the wear to each location; it only requires the aggregate write-back rate to memory, which is easily measurable on modern hardware. Surprisingly, we find that this simple approach may be acceptable for PM devices with a sufficiently high endurance (e.g., 10^9 writes).

We consider a scheme that periodically performs a random *shuffle* of all virtual pages, reassigning each virtual page to a randomly selected physical page. Whereas our analysis assumes all pages are shuffled at once for simplicity, in practice, pages are shuffled continuously and incrementally over the course of the shuffle period. Our analysis poses the question: How many times must the address space be shuffled for the expected number of writes to each page to approach uniformity? Furthermore, at what point does the wear incurred by shuffling exceed the wear from the application? To simplify discussion, we use "write" to mean write-back from the last-level cache to the PM throughout this section.

Analysis. Let W represent the write distribution to physical pages and W_i be the write rate to i^{th} physical page in the memory. We define an equality function E as:

$$E(x,y) = \begin{cases} 1 & x == y \\ 0 & x != y \end{cases} \quad (1)$$



a. PDF and CDF for N shuffles

b. Write rate of nth percentile hot pages

Figure 2: **(a) Write-back rate distribution**: We use an application's write distribution to derive 99^{th} percentile write rate after N shuffles. **(b) Write-back rate vs. shuffles**: The disparity in page write rates shrinks with the increase in shuffles.

Given a write distribution W over n physical pages, P_n^k represents the probability density function (PDF) for W after k shuffles. Using the distribution W, we can compute the probability $P_n^0(x)$ of physical page with the write rate x with 0 shuffles (initial state) as:

$$P_n^0(x) = \frac{1}{n} \times \sum_{i=1}^{n} E(W_i, x) \tag{2}$$

With no shuffles, one can easily compute the expected life of each physical page by dividing the expected endurance (in number of writes) by the write rate x, yielding an expected lifetime distribution over pages. When we consider a shuffle's effect, each page will experience an average write rate x' of two write rates x_1 and x_2 chosen uniformly at random from W. Since the PDF of the sum of two random variables is the convolution of their respective PDFs, we can calculate the expected distribution of write rates after S shuffles, P_n^S, as:

$$P_n^S(X = x'/2) = \sum_{k=-\infty}^{\infty} P_n^{S-1}(X = k) P_n^{S-1}(X = x' - k) \tag{3}$$

Note the normalization by one half, since we want the average (rather than the sum) of the random variables.

We illustrate the PDF P_n^0 (expected write rate without shuffles) of the page write distribution as expressed by Eq. 2 in Fig. 2 (a). The PDF P_n^0 has a heavy right-tailed distribution with high variance (*i.e.* the write-rate of few pages is high as compared to the mean write rate), a characteristic typical of the applications we have studied. Moreover, due to high variance, there is a wide write-rate range that might occur for any given page. Next, we compute the PDF P_n^S using Eq. 3 for shuffles ranging from one to N. With each shuffle, the PDF variance shrinks, while the probability of a near-mean write rate increases. Note that the PDF mean P_n^1 *appears* to be higher than the PDF P_n^0 due to the heavy right-tail of P_n^0. The mean in fact stays constant after each shuffle.

Fig. 2 (a) illustrates how the PDF after N shuffles converges to the mean write rate (equivalently, writes become uniformly distributed over the physical pages). In Figure 2 (a), we also show the cumulative distribution function (CDF) for N shuffles where the CDF C_n^N is used to compute the top n^{th} percentile of pages with the highest write rate after N shuffles (i.e., the "hottest" pages). $C_n^N(p)$ provides the minimum expected write rate of the most heavily written $(1-p)*100\%$ of the pages. For example, in Fig. 2 (a), we mark with a dotted line the 99^{th} percentile. The $C_n^N(p = 0.99)$ gives the minimum expected write rate of the most heavily written 1% of pages after N shuffles. From this rate, we can estimate when we expect this 1% of pages to have worn out. As the number of shuffles grows, the variance shrinks and $C_n^N(p = 0.99)$ approaches the mean write rate.

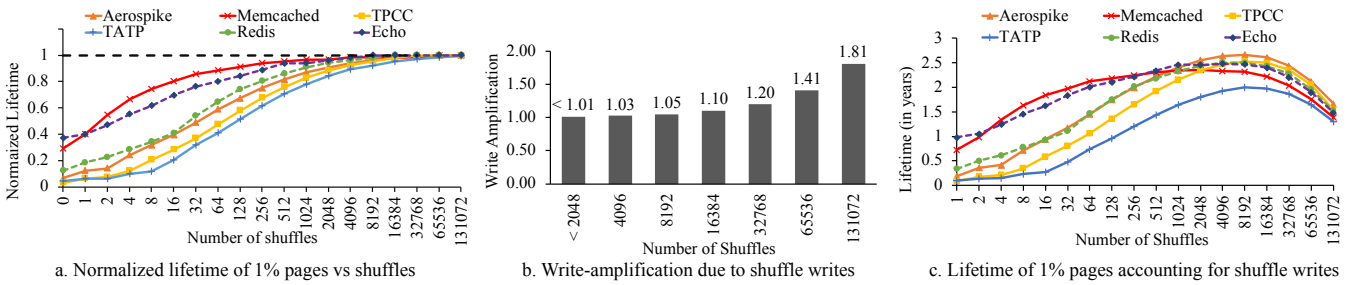
We illustrate how the write rate of the hottest pages compares to the mean as a function of the number of shuffles in Fig. 2 (b). Note that our approach can estimate the wear rate at any percentile, but we present results primarily for the 99^{th} percentile. Without shuffles, there is a large disparity between the most-written 1% of pages and the mean. The gap rapidly shrinks with additional shuffles. Given the hottest pages' write rates in Fig. 2(b), we compute lifetime of a device with a 10^7 write endurance.

Tracing Methodology. We collect write-back traces for a set of applications (detailed in Section 5) using the DynamoRio [17] instrumentation tool and its online cache simulation client `drcachesim`. Since `drcachesim` can simulate only a two-level cache hierarchy with power-of-two cache sizes, we model an 8-way 256KB L2 cache and 32MB 16-way associative L3 cache, which is close to the configuration of the physical system on which we evaluate our Kevlar prototype (described in Table 1). We instrument loads and stores to trace all memory references and run `drcachesim` online to simulate the system's cache hierarchy. We record writebacks from the simulated LLC to PM. We then extract write rate distributions to analyze expected PM lifetime under shuffling.

Determining optimal shuffles. In Fig. 3(a), we show the lifetime, normalized to what is possible under ideal wear leveling, as a function of the number of shuffles. We assume some redundancy in the PM device similar to prior works [85, 86] and define its lifetime as the time when 1% of pages are expected to fail. Note that the lifetime under ideal wear leveling is the device endurance divided by the application's average write-back rate. As shown in Figure 3(a), frequently written virtual pages are mapped to a different set of physical pages after every shuffle, leading to improved device lifetime with more shuffles. Interestingly, for all applications, after about 8192 shuffles, the expected lifetime converges to that of ideal wear leveling (i.e., the write distribution is uniform). Note that we do not consider the additional writes incurred due to remapping virtual-to-physical page mappings after each shuffle in Figure 3(a).

Figure 3(b) shows the write amplification caused due to the shuffle operations. The write amplification shows the ratio of the total writes incurred after shuffling as compared to the application's PM writes. The write amplification can be higher than 1.4x (40% additional writes) for greater than 2^{16} shuffles as shown in Figure 3(b).

Peak lifetimes occur when memory is shuffled 8192 times over the device lifetime. With 8192 shuffles, we perform 5% additional writes for wear leveling. Fig. 3(c) shows the writes



a. Normalized lifetime of 1% pages vs shuffles

b. Write-amplification due to shuffle writes

c. Lifetime of 1% pages accounting for shuffle writes

Figure 3: **(a) Lifetime of 1% of pages vs. shuffles**: The expected lifetime converges to the ideal lifetime for shuffles > 8192, **(b) Write-amplification due to shuffle writes**: Kevlar performs 5% additional writes with 8192 shuffles, **(c) Lifetime of 1% of pages, accounting for shuffle writes**: The lifetime of PM peaks at 8192 shuffles, following which shuffle writes become significant.

due to shuffle operations, which may grow to dwarf the application's writes if shuffles are too frequent (*i.e.* >16384).

Discussion. Shuffling memory 8192 times over the PM device lifetime uniformly distributes PM writes. However, the lifetime achievable via even ideal wear leveling is limited by an application's average write rate. For our applications, this lifetime is only 2.3 to 2.8 years for a device that wears out after 10^7 writes (see Fig. 3(c)). Wear leveling alone may be insufficient to meet lifetime targets.

To achieve desired lifetimes, we must augment Kevlar's wear-leveling mechanism with a wear-reducing mechanism. The key challenge for wear reduction is to monitor the wear to each virtual page at low overhead. There is no straightforward mechanism for the OS to directly monitor device wear at fine granularity. PM devices incur wear only when writes reach the device. Write-back caches absorb much of the processor write traffic, so the number of stores to a location can be a poor indicator of actual device wear. Current x86 hardware can count writebacks per memory channel, but provides no support for finer-grain (e.g., page or cache line) monitoring. Mechanisms that monitor writes via protection faults (e.g., [5, 34]) incur high performance overhead and fail to account for wear reduction by writeback caches, grossly overestimating wear for well-cached locations. Instead, Kevlar builds a software mechanism to estimate per-page wear intensity.

3.2 Wear Estimation

We design a wear-estimation mechanism that approximately tracks hardware cache contents to estimate per-page PM write-back rates. Our mechanism builds upon Intel's PEBS performance counters [45] to sample store operations executed by the processor. Note that, although we focus on Intel platforms, other platforms—AMD Instruction Based Sampling [29] and ARM Coresight Trace Buffers [7]—provide analogous monitoring mechanisms. Kevlar's write estimation mechanism monitors the retiring stores to maintain an estimate of hardware cache contents.

Monitoring stores. PEBS captures a snapshot of processor state upon certain configurable events. We configure PEBS to monitor `MEM_UOPS_RETIRED.ALL_STORES` events. As stores retire, PEBS can trigger an interrupt to record state into a software-accessible buffer; we record the virtual address accessed by the retiring store.

Although accurate, sampling every store with PEBS is prohibitive. Instead, we rely on systematic sampling to reduce performance overhead: we configure PEBS with a *Sample After Value* (SAV). For a SAV of n, PEBS captures only every n^{th} event. Like prior work [71], we choose prime SAVs to avoid bias from periodicities in the systematic sampling. We explore the accuracy and overhead of SAV alternatives in Section 6.1.

We obtain the virtual addresses of sampled stores to estimate per-page write-back rates. A naive strategy to compute write-back rates is to assume that each sampled store results in a write-back. However, with write-back hardware caches, a PM write occurs only when a dirty block is evicted from the cache hierarchy; many stores coalesce in the caches. Indeed, in our applications, the naive strategy drastically overestimates writebacks (see Section 6.1). Consequently, we design an efficient software mechanism that estimates temporal locality due to hardware caches to predict which stores incur write-backs.

Estimating temporal locality. Prior mechanisms have been proposed to estimate temporal locality in storage [102, 103] or multicore [13, 90, 91] caches. These mechanisms maintain stacks or hashmaps to compute reuse distances for accesses to sampled locations. Instead, we focus on modeling temporal locality in hardware caches to estimate LLC write-backs using sampled stores. We estimate temporal locality by using a Bloom filter [16] to approximately track dirty memory locations stored in the caches. For each store sampled by PEBS, we insert its cache block address into the Bloom filter. (Algorithm 1: Line 12-14). Whenever a new address is added to the filter, we assume it is the store that dirties the cache block, and hence will eventually result in a writeback. Further stores to the same cache block will find their address already present in the Bloom filter; we assume these hit in the cache and hence do not produce additional write-backs. Thus, the Bloom filter maintains a compact representation of likely dirty blocks present in the cache.

Bloom filters have a limited capacity; after a certain num-

ber of insertions into the set, their false positive rate increases rapidly. We size the Bloom filter such that it can accurately (less than 1% false positives) track a set as large as the capacity of the processor's last-level cache (LLC), which is roughly 700K cache blocks on our evaluation platform. We clear the Bloom filter when the number of insertions reaches this size (Algorithm 1: Line 19-29).

Of course, after clearing the filter, Kevlar would predict a sudden false spike in writeback rates. We address this by using two Bloom filters; Kevlar probes both filters but inserts into only one "active" filter at a time (Algorithm 1: Line 3, 12-17). When the active filter becomes full, we clear the inactive filter and then make it active. As such, at steady state, one filter contains 700K cache block addresses, while the other is active and being populated (Algorithm 1: Line 12-17). We assume a cache block will result in a store hit (no additional writeback) if it is present in either filter (Algorithm 1: Line 6-10).

In essence, our tracking strategy filters out cache blocks that have write reuse distances [56] of about 700K or less, as such writes are likely to be cache hits. Effectively, we assume that dirty blocks are flushed from the cache primarily due to capacity misses, which is typically the case for large associative LLCs [39, 113]. Note that our estimate of the cache contents is approximate. For example, the Bloom filters do not track read-only cache blocks. Moreover, due to SAV, only a sample of writes are inserted. The mechanism works despite these approximations because: (1) frequently written addresses are likely to be sampled and inserted into the filters—it is these addresses that are most critical to track; and (2) few addresses have reuse distances near 700K—reuse distances are typically much shorter or longer, so the filters are effective in estimating whether or not a store is likely to hit. Although Kevlar approximates writebacks by sampling retiring stores, our goal in Kevlar is to measure relative hotness of the pages as opposed to absolute writebacks per page. We show the accuracy of our estimation mechanism to identify writeback intensive pages later in Section 6.1.

Estimating write-backs. PEBS provides the virtual address of sampled stores. Our handler then walks the software page table to obtain the corresponding physical frame (Alg. 1: Line 7). In our Linux prototype, we maintain a writeback count in `struct page`, a data-structure associated with each page frame. When we sample a store, we update the counter for the corresponding physical page as shown in Alg. 1: Line 8. Kevlar uses the estimated writebacks to identify writeback-intensive pages.

3.3 Wear Reduction

As shown in Sec. 3.1, Kevlar's wear-leveling mechanism can achieve only 2.3- to 2.8-year lifetime for a PM device that wears out after 10^7 writes. Our goal is to achieve a lifetime target for a low-endurance PM device by migrating heavily written pages to DRAM. We assume a nominal lifetime goal of four years. This target is software-configurable; we discuss longer targets in Section 6.2.

Consider an application with a memory footprint of N physical PM pages and a given lifetime target, the write rate to the PM B writes/sec to achieve the lifetime target can be computed as:

$$B = \frac{Endurance \times N}{Lifetime} \tag{4}$$

We use Eq. 4 to compute the number of writes the application may make per 1GB (*i.e.* N = 256K small pages) of PM footprint. For a given lower-bound endurance of 10^7 writes and a 4-year lifetime, writebacks must be limited to 20K writes/sec/GB. Configuring a different target lifetime or device endurance changes the allowable threshold.

One approach is to use wear leveling (as described in Sec. 3.1) by provisioning additional reserve capacity such that the target lifetime is met. This strategy is applicable both when PM is used for persistent storage or capacity expansion. For instance, with N pages in an application, and average write rate of B' writes/sec/GB, the reserve capacity R to achieve a 4-year lifetime is given by:

$$R = \frac{N \times B'}{2 \times 10^4} \tag{5}$$

When the application write rate is high relative to the device endurance, the required reserve can undermine any cost advantages, as we show later in Section 6.3. Instead, for capacity expansion, we propose wear reduction by migrating the hottest pages to high-endurance memory (DRAM). Kevlar regulates the average write rate to the pages that remain in PM to 20K writes/GB/sec such that we achieve the desired lifetime of four years.

3.3.1 Page migration

Kevlar uses its write-back estimation mechanism to measure per-page PM writeback rates and migrate the most write-intensive pages to DRAM. Kevlar must regulate average PM writeback rate to 20K writes/GB/sec to achieve a 4-year lifetime. Kevlar uses `IMC.MC_CHy_PCI_PMON_CTR` counters in the Intel memory controller to count `CAS_COUNT.WR` events, which measure write commands issued on the memory channels. Such counters already exist in DRAM controllers, and analogous counters exist on other hardware platforms (*e.g.* ARM's L3D_CACHE_WB performance monitoring unit counter [8]). This aggregate measure allows us to determine whether pages must be migrated from PM to DRAM (or can be migrated back) to maintain the target average rate of 20K writes/GB/sec.

Migrating hot-pages to DRAM. Kevlar computes the PM writeback rate at a fixed 10-second interval. If the average writeback rate exceeds 20K writes/GB/sec during an interval, Kevlar enables PEBS and samples the retiring stores as explained in Section 3.2. Kevlar estimates the PM writeback rate

Algorithm 1 Write-back estimation mechanism

```
1: Inputs:
       PEBS record rec, Bloom Filter filterA, Bloom Filter filterB
2:
3: Initialize:
       filterA.isActive = True
       filterB.isActive = False
       activate = LLC_CACHE_BLOCKS
4:
5: blockAddr = rec.strAddr » log2(LLC_BLOCK_SIZE)
6: if !filterA.isPresent(blockAddr) and !filterB.isPresent(blockAddr) then
7:     pageStruct = doPageWalk(blockAddr)
8:     pageStruct.WBCount+=1
9:     memRef+=1
10: end if
11:
12: if filterA.isActive and !filterA.isPresent(blockAddr) then
13:     filterA.add(blockAddr)
14: end if
15: if filterB.isActive and !filterB.isPresent(blockAddr) then
16:     filterB.add(blockAddr)
17: end if
18:
19: if activate == memRef then
20:     filterA.isActive = !filterA.isActive
21:     filterB.isActive = !filterB.isActive
22:     if filterA.isActive then
23:         filterA.clear()
24:     end if
25:     if filterB.isActive then
26:         filterB.clear()
27:     end if
28:     activate+=LLC_CACHE_BLOCKS
29: end if
```

at 4KB-page granularity. When migration is needed, Kevlar scans writeback counters for all page frames and sorts them by their estimated write-back counts. Kevlar then migrates the hottest 10% of pages to DRAM. It continues monitoring for an additional interval. Kevlar ceases migration, disables PEBS monitoring, and clears write-back counters when the write-back rate falls below 20K writes/GB/sec. With this monitoring and migration control loop, Kevlar achieves our lifetime target with 1.2% performance impact.

Migrating cold pages to PM. An application's access pattern might change over its execution, so pages migrated to DRAM may become cold. To minimize the application footprint in DRAM, it is desirable to migrate cold pages back to PM. If Kevlar observes five consecutive intervals with a PM writeback rate below 20K writes/GB/sec, it re-enables PEBS for a 10-second interval, estimates the write-back rate of pages in DRAM, and migrates 10% of cold pages from DRAM back to PM.

4 Implementation

We implement Kevlar in Linux kernel version 4.5.0. We use the Linux control group mechanism [74] to manage Kevlar specific configuration parameters.

Wear leveling. Kevlar should shuffle the entire application footprint once every 4.2 hours to achieve uniform wear leveling over a lifetime of 4 years. Instead of gang-scheduling the shuffle operations together every 4.2 hours, Kevlar periodically shuffles a fraction of application footprint. Kevlar maintains a shuffle bit in the `struct page` associated with each page frame to indicate whether the page was shuffled within the current shuffle interval. Kevlar scans the application pages every 300-sec *shuffle interval* to identify the pages that are yet to be shuffled. It randomly chooses a fraction of pages to be shuffled in this shuffle interval by equally apportioning the total number of pages yet to be shuffled to the time remaining in a 4.2 hour shuffle operation.

The fraction of pages are then shuffled following these steps: (1) Kevlar selects a pair of application pages in PM to be swapped. (2) It locks the page table entries for both pages so that any intermediate application accesses stall on page locks. (3) It allocates a temporary page in DRAM (for capacity workloads) to aid in swapping the contents of the two pages in PM. (4) Once the pages are swapped, Kevlar restores the page table entries so that the virtual addresses now map to the swapped pages, unlocks the pages, and deallocates the temporary DRAM page. (5) Once shuffled, Kevlar records this event in the shuffle bit in page frame's `struct page` of the two pages.

Note that, we use a temporary page mapped in DRAM to limit wear in PM due to shuffle. For persistent applications, we map the temporary page in PM to ensure that the page contents are persistent in case of intermediate failure. Once all the pages are shuffled, Kevlar clears the shuffle bit in `struct page` and initiates the next shuffle.

Wear estimation. Kevlar initializes PEBS to monitor the `MEM_UOPS_RETIRED.ALL_STORES` event and a SAV to sample the retiring stores for wear estimation. We determine SAV empirically to ensure that the monitoring has negligible performance overhead. Kevlar implements two Bloom Filters, each of size 840KB and a capacity of 700K cache blocks, corresponding to the 45MB LLC of our system. We size the Bloom filter to achieve less than 1% false positives. As explained in Section 3.3.1, Kevlar performs a software page table walk to identify the page frames being accessed by the sampled store, and records writeback counts in `struct page`.

Wear reduction. Kevlar monitors PM writeback rate at a 10-second *migration interval* to determine if it needs to initiate hot/cold page migration between DRAM and PM. If the PM writeback rate triggers a migration, Kevlar scans the application pages and identifies the top 10% hot (or cold) pages to be migrated to DRAM (or PM). It performs migration using a mechanism similar to the page shuffles in wear leveling: it locks the page to be migrated, copies its contents to a newly allocated page in DRAM (or PM), updates page table entries, and unlocks the page. If no migration is triggered, Kevlar disables PEBS sampling counters to minimize performance monitoring overhead.

5 Methodology

We next discuss details of our prototype and evaluation.

Core	Intel Xeon E5-2699 v3, 2.30GHz 36-core (72 hardware threads) Dual-socket x86 server
L1 D&I Cache	32KB, 8-way associative
L2 Cache	256KB, 8-way associative
Shared LLC	45MB, 20-way associative
DRAM	256GB per socket
Operating System	Linux Kernel 4.5.0

Table 1: **System Configuration.**

5.1 Emulating persistent memory

A system with byte-addressable persistent memory is not yet commercially available. Hence, we emulate a hybrid PM-DRAM memory system using a dual-socket server. We run the application under test on a single socket and treat memory local to that socket as DRAM. Conversely, we treat memory of the remote socket as PM. Note that the local and remote nodes are cache coherent across the sockets. Since each chip has its own memory controllers, we use the performance counters in each memory controller to monitor the total accesses to each device and distinguish "PM" and "DRAM" accesses.

Using this emulation, our Kevlar prototype incurs the actual performance overheads of monitoring and migration that would occur in a real hybrid-memory system. However, the latency and bandwidth differential between our emulated "PM" and "DRAM" is only the gap between local and remote socket accesses. The performance differential between DRAM and actual PM devices is technology dependent and remains unclear, but is likely higher than in our prototype. We expect relative performance overhead of our mechanism (as detailed later in Section 6.4) to be lower on a system with a high differential between DRAM and PM devices. Our results represent a high estimate of the Kevlar's performance overhead.

Nevertheless, our contributions with respect to wear management are orthogonal to the performance aspects of replacing DRAM with PM, which have been studied in prior work [5, 55, 84]. We focus our evaluation on quantifying the effectiveness and overheads of Kevlar's mechanisms.

5.2 System configuration

We run our experiments on a dual-socket server with the configuration listed in Table 1. We use the Linux control group mechanism [74] to isolate the application to a particular socket. We pin application threads to execute only on CPUs on the local node, but map all memory to initially allocate in the remote node using Linux's `memory` and `cpuset cgroups`, modeling a system where DRAM has been replaced by PM. Kevlar expects a lifetime goal for the PM device as an input, and performs wear leveling, estimation, and reduction for all the processes in the `cgroups`. The test applications use all 18 CPU cores of the local node with hyper-threading enabled. For client-server benchmarks, we run clients on another system to avoid performance interference.

As explained in Section 3.2, we use Intel's PEBS counters to estimate PM page writeback frequency. We isolate these counters to monitor only accesses from the application under test using Linux's `perf_event` cgroup mechanism. Thus, spurious store operations from background processes or the kernel do not perturb our measurements.

We measure the write rate to the PM (*i.e.* remote DRAM) using the performance counters in the memory controller. Unlike PEBS counters, these counters lie in a shared domain and cannot be isolated to count only events for a particular process. However, we have measured the write rate of the background processes in an idle system and find that they constitute less than 1% of the total writeback rate observed during our experiments.

5.3 Benchmarks

We study two categories of applications. We report memory footprints of the benchmarks under study in Figure 9.

5.3.1 Capacity Expansion Workloads

We evaluate both the wear-leveling and wear-reduction mechanisms of Kevlar for the following benchmarks in a "capacity expansion" PM use case.

NoSQL applications. Aerospike [1, 97], and Memcached [4] are popular in-memory NoSQL databases. We use YCSB clients [24] to generate the workload to Aerospike and Memcached. We evaluate 400M operations on 4M keys for Aerospike and 100M operations on 1M keys for Memcached. We configure each record to have 20 fields resulting in a data size of 2KB per record. As we are interested in managing wear in write-intensive scenarios, we configure YCSB for update-heavy workload with a 50:50 read-write ratio and Zipfian key distribution.

MySQL. MySQL is a SQL database management system. We drive MySQL using the open-source TPCC [98] and TATP [79] workloads from oltpbench [27]. TPCC models an order fulfillment business and TATP models a mobile carrier database. In each, we run default transactions with a scale-factor of 320 for 1800 secs.

5.3.2 Persistent Workloads

We evaluate persistent applications from the WHISPER benchmark suite [77], which use the Intel PMDK libraries [2] for persistence. These applications divide their address space into volatile and persistent subsets. The persistent subset must always be mapped to PM to ensure recoverability in the event of power failure. As such, Kevlar may not migrate pages in the persistent subset to DRAM. We instead rely only on wear leveling to shuffle these pages in PM. However, we allow pages

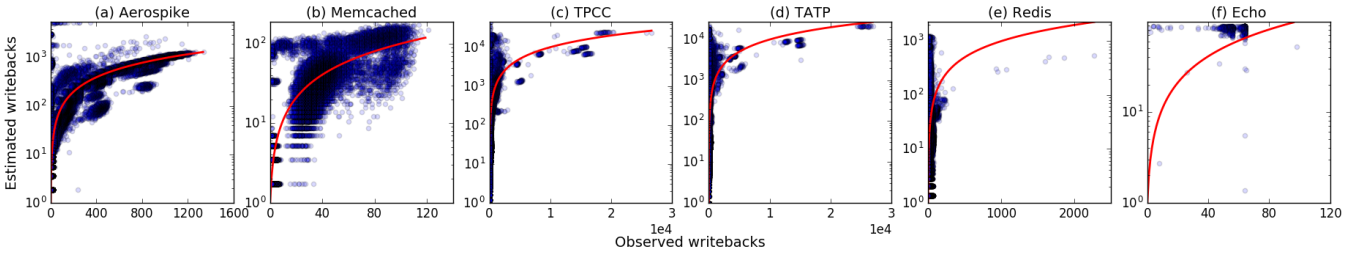


Figure 4: **Estimated writebacks vs. observed writebacks**. We compare the estimated writebacks with observed writebacks obtained from memory access tracing. Each point on the scatter plot represents the number of writebacks to a page. The red line on each plot represents the ideal prediction curve.

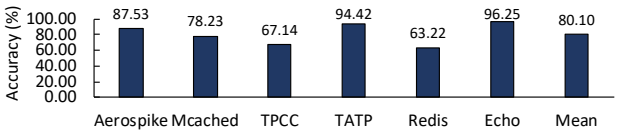


Figure 5: **Comparison of top 10% estimated hot pages to top 10% observed hot pages**. Kevlar's wear estimation identifies 80.10% (avg.) of the 10% hottest written pages correctly.

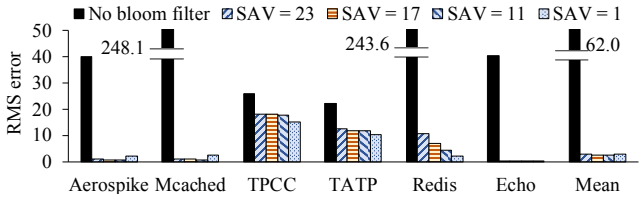


Figure 6: **RMS Error with cache modeling**. Kevlar achieves 20× lower RMS error than a mechanism without cache modeling.

in the volatile subset to migrate to DRAM if the aggregate write rate to all pages exceeds 20K writes/GB/sec.

Linux presently provides no mechanism to label pages as persistent or volatile. WHISPER benchmarks use Linux's `tmpfs` [96] memory mapped in DRAM to emulate persistency, and the persistent pages are allocated in a fixed address range. We hardcode this address range in our experiments to prevent page migrations to DRAM.

We select the two NoSQL applications, Redis and Echo, from WHISPER. Redis is a single-threaded in-memory key-value store. We configure a Redis database comprising 1M records, each with 10 fields. We use YCSB clients to perform key-value operations on the Redis server with a Zipfian distribution. For our evaluation, we run 40M operations with an update-heavy workload with a 50:50 read-to-write ratio. For echo, we use the configuration provided with the WHISPER benchmark suite and evaluate it using 2 client threads each running 40M operations.

6 Evaluation

We evaluate Kevlar's wear-management mechanisms.

6.1 Modeling Wear Estimation

We first evaluate the accuracy of Kevlar's wear-estimation mechanism as described in Section 3.2. We collect a ground-truth writeback trace for each application using the online cache simulator `drcachesim` in Dynamorio [17] with a tracing infrastructure described in Section 3.1. We model the PEBS sampling mechanism and bloom filters in `drcachesim` to record the estimated writeback rate. We compare the ground-truth writebacks against the estimates provided by the emulation of PEBS sampling and our Bloom filters.

Comparison with ideal mechanism. In Figure 4, we show estimated writebacks (vertical axis) and ground-truth observed writebacks (horizontal axis) for each application for one 10-sec sampling interval. We use log-linear scale[1] to highlight accuracy of our mechanism for higher write rate. As instrumentation results in application slowdown, we expand the 10-second sampling duration by the slowdown due to instrumentation measured for each workload. Due to the log-linear scale, we plot a red curve in the Figure to show the ideal prediction curve, where estimated and observed writebacks match. For all applications, Figure 4 (a-f) indicates that the estimated writebacks correlate closely to the ideal curve. Echo performs cache flush operation following each store to flush dirty cache blocks to PM. As a result, we observe 64 write-backs per page (owing to 64 cache blocks in a 4KB page) for nearly all pages. As shown in Figure 4(f), Kevlar is able to measure write-backs to these pages.

Prediction accuracy. Next, we compare the top 10% heavily written pages as estimated by Kevlar's wear-estimation mechanism to the top 10% hottest observed (ground-truth) pages. Figure 5 shows the percentage of heavily written pages correctly estimated by Kevlar. Kevlar correctly estimates 80.1% hottest pages on average and up to 96.3% hottest pages in Echo as compared to the ground truth.

We also demonstrate the accuracy of Kevlar's prediction mechanism by measuring root-mean-squared (RMS) error between estimated and observed writebacks. The RMS error reports the standard-deviation of the difference between estimated and observed writebacks. We study the impact of hardware cache modeling using our Bloom filter mechanism

[1] We use log-linear scale to highlight estimated and observed writebacks to hot pages that are crucial for our study. In contrast, a log-log scale discretizes lower writeback values and hides comparison between observed and estimated writebacks for hot pages.

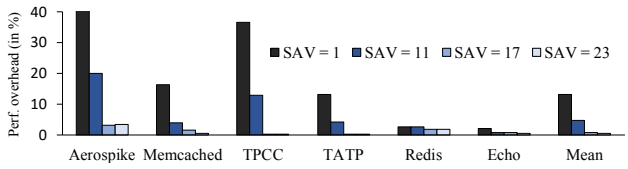


Figure 7: **PEBS sampling overhead**. Runtime overhead due to sampling every retiring store is 13.2% (avg.). We configure PEBS SAV = 17 in Kevlar with < 1% overhead.

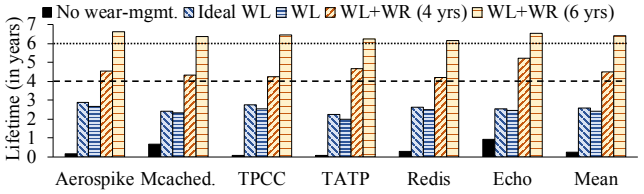


Figure 8: **PM Lifetime**: Kevlar achieves greater than 4 years of lifetime; 11.2× (avg.) higher than no wear leveling.

by comparing Kevlar's prediction mechanism with a mechanism without the Bloom filter. Figure 6 shows the RMS error of our writeback prediction mechanism normalized to the average writeback rate of the application for different PEBS SAV values. We choose prime numbers for PEBS SAV to avoid periodicities in systematic sampling.

As compared to a mechanism that does not model cache contents, we observe 100.0× and 106.8× improvement in RMS errors for Memcached and Redis, respectively, with our estimation mechanism (with SAV = 1). Overall, the Bloom filters can approximate the dirty cache contents well, allowing it to estimate writebacks with 21.6× lower RMS error on average. The Bloom filters are critical to avoiding overestimation of writebacks in Aerospike, Memcached, and Redis by estimating temporal locality of memory accesses. Note that, as shown in Figure 6, the standard deviation of the difference between absolute values of estimated and observed writebacks is 2.85× that of the mean for SAV of 1. Although the estimated writebacks are not accurate when compared to absolute values, our goal in Kevlar is to measure the relative hotness of the pages. As shown earlier in Figure 5, Kevlar identifies 80.1% of the 10% hottest pages correctly.

Configuring PEBS SAV. We study the RMS error in Figure 6 and runtime performance overhead in Figure 7 for different PEBS SAV values. Figure 7 shows the monitoring overhead for different SAVs when compared to the application runtime without PEBS monitoring. Upon sampling a store, PEBS triggers an interrupt and records architectural state in a software buffer, which can lead to a performance overhead. Taking an interrupt on every retiring store results in substantial performance overhead. Indeed, with SAV=1, the performance overhead due to PEBS sampling can be as high as 112.9% (in Aerospike), and 13.2% on an average. In contrast, the performance overhead in persistent applications, Redis and Echo, is less than 3% as we sample only stores to volatile pages, which may be migrated between PM and DRAM. Interestingly, with SAV of 17, the average performance overhead due to sampling is less than 1% (avg.) with no substantial degradation in RMS error. As we do not see any substantial performance gains for SAV > 17, we configure PEBS to sample one in every 17 stores in Kevlar.

6.2 PM Lifetime

We study Kevlar for lifetime targets of four and six years. We compare Kevlar's wear-management mechanisms to a baseline with no wear leveling. We make a conservative assumption that a write to a physical page modifies all locations within that page for Kevlar's wear-management mechanisms. In contrast, we measure lifetime for the baseline via precise monitoring at cache-line granularity.

Wear leveling alone. We first consider lifetime for the PM device achieved by Kevlar's wear-leveling mechanism alone. As discussed in Section 3.3, to achieve a four- (or six-) year lifetime until 1% of locations wear out on a PM device that can sustain only 10^7 writes, the average write rate must be below 20,000 (or 13,333) writes/GB/second. Even after wear leveling, all of the applications we study incur a higher average write rate when their entire footprints reside in PM. We also show lifetime due to ideal wear leveling in Figure 8 when writes are uniformly remapped in PM. Although wear leveling substantially improves PM lifetimes over a baseline of no wear leveling, it falls short of achieving the four-year and six-year lifetime targets for all applications. As compared to the baseline with no wear leveling, Kevlar with only wear leveling achieves an average lifetime improvement of 9.8× with 31.7× improvement in lifetime for TPCC.

Wear leveling + wear reduction. Wear reduction can improve application lifetimes to meet our target while moving only a remarkably small fraction of the application footprint to DRAM. Kevlar in wear leveling + wear reducing mode aims to limit the write-back rate to the PM at 20K (or 13.3K) write/GB/second for four (or six) year lifetime target, by identifying the "hottest pages" that are being frequently written back and migrating them to DRAM.

Owing to the writeback rate limit imposed by Kevlar's wear-reducing mechanism, as indicated in Figure 8, the lifetime with wear leveling + wear reduction exceeds the configured target of four and six years for all applications. Kevlar's wear leveling + wear reduction mode (for a 6-year lifetime configuration) achieves the highest lifetime improvement of 80.7× for TPCC, with an average improvement of 26.1× when compared to no wear leveling.

High-endurance PMs: Absent wear-management mechanisms, a PM device that can sustain 10^8 writes would wear out within 9.8 months. Moreover, for PM devices with endurance 10^8 - 10^9, wear-leveling mechanism would be sufficient to

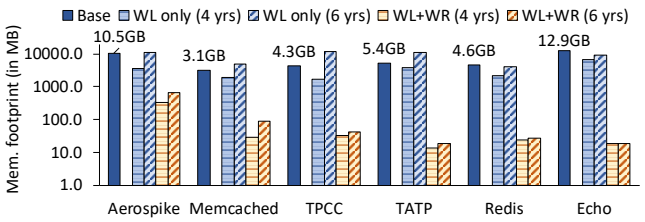


Figure 9: **Application footprint in PM and DRAM**: Kevlar migrates < 1% of application footprint to DRAM. Blue and orange bars represent application footprint in PM and DRAM respectively.

achieve the desired lifetimes of 4- and 6-years. For instance, our wear-leveling mechanism alone can achieve a lifetime of 24.0 years (average) for a PM device that can sustain 10^8 writes. Kevlar would not trigger wear-reduction mechanism for PMs with high write endurance as the application write-back rate would be lower than configured threshold. Nevertheless, the endurance numbers of commercial PM devices (i.e. Intel's 3D XPoint) are not publicly available. As such, we can configure the endurance of a PM device in Kevlar.

6.3 Memory Overhead

Figure 9 shows the baseline memory footprint of the applications, and an additional memory footprint in DRAM necessary to host the most frequently written PM pages that are migrated by Kevlar. In addition, we also show the reserve footprint that can be mapped in PM to achieve the lifetime targets using wear-leveling mechanism alone as outlined in Equation 5.

Wear reduction for persistency applications. For the WHISPER benchmarks that rely on persistency (Redis & Echo), the pages in the persistent set must always remain in PM. Nevertheless, some fraction of these applications' footprints are volatile and may reside in PM or DRAM. We initially map the entire footprint to the PM and allow only volatile pages to migrate to DRAM. As a majority of memory accesses are made to the volatile footprint in these applications [77], the wear-reducing mechanism can achieve a 4 year lifetime by migrating only 23.6MB of footprint to DRAM.

Reserve PM required can be significant. The amount of PM reserves required to ensure that the target lifetime be met are significant. It can be as high as 2.7× for TPCC and 2.0× for TATP for a six-year lifetime (1.3× average across all the benchmarks). The required reserve capacity may undermine the cost advantages of capacity expansion offered by PMs.

Reserve DRAM required is much smaller than reserve PM. As can be seen from Figure 9, the reserve DRAM required is much smaller than the reserve PM required. This difference is due to a difference in the write endurance of DRAM (practically infinite) and the cell endurance we assume for PM (10^7 writes). Note that Kevlar's goal is to limit wear while maximizing application footprint in PM (especially for the capacity expansion use-case) and achieve configured device lifetime. Thus, it migrates only the heavily written

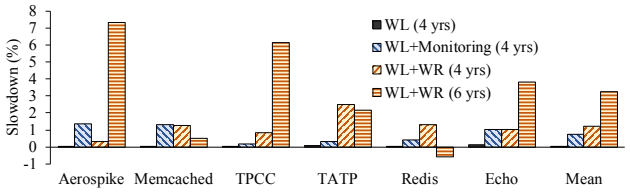


Figure 10: **Performance overhead**: Overhead of page monitoring and migration in Kevlar is 1.2% (avg.) in our applications.

application footprint from PM to DRAM. In contrast, prior mechanisms [5, 55] aggressively migrate pages to DRAM and limit application performance degradation resulting from slower PM accesses. Kevlar migrates less than 1% of the application's footprint to DRAM for four- and six-year lifetime targets, on average.

6.4 Performance Overhead

Next, we present application slowdown due to Kevlar.

Page shuffle overhead. Figure 10 illustrates the slowdown (lower is better) in applications resulting from our wear leveling, wear estimation, and page migration. The shuffle mechanism incurs a negligible average performance overhead of 0.04% (highest 0.1% in Echo) over the baseline with no wear leveling.

Overheads from Kevlar's monitoring and migration. As explained in Section 3.3, we configure PEBS with SAV of 17, and further reduce performance overhead by filtering store addresses using the Bloom filters. We observe up to 1.3% slowdown from our PEBS sampling in Aerospike, with even lower overheads in the remaining applications. Redis observes a net gain (as much as 0.9%) when we enable migration and relocate their frequently written pages to DRAM because the local NUMA node (representing DRAM) is faster than the remote node (representing PM) in our prototype. We expect the performance gains to be more pronounced with PMs that are anticipated to exhibit higher memory latency than remote DRAM in our prototype. On an average, we see 1.2% (or 3.2%) slowdown due to our wear-management mechanisms to achieve the lifetime goal of four (or six) years.

7 Related work

The adoption of PMs has been widely studied by both academia and industry in processor architectures [23, 28, 51, 52, 60, 77, 83, 95, 117], file systems [20, 23, 30, 100, 105, 107, 108], logging/databases [10, 11, 18, 19, 35–37, 59, 62, 63, 68, 73, 80, 81, 104], data structures [43, 78, 99], and distributed systems [57, 70, 116, 120]. We discuss the relevant works that address wear out problem in PMs.

7.1 Wear-reduction mechanisms

We first discuss techniques that reduce PM writes.

DRAM cache. Numerous works [32, 75, 87, 92] advocate placing a DRAM cache in front of PM. The DRAM cache absorbs most of the writes thereby reducing wear. A DRAM cache presents three disadvantages: (1) it sacrifices capacity that could instead be used to expand memory; (2) it increases the latency of PM writes; and (3) it is inapplicable to writes that require persistency, which must write through the cache. Like many prior works [23, 28, 36, 51, 52, 60, 77, 83, 95, 117], we assume that PM and DRAM are peers on the memory bus.

Page migration. Several works [6, 26, 88, 114] propose migrating pages from PM to DRAM to reduce wear. Dhiman et al. [26] use a software-hardware hybrid solution, where dedicated hardware counters (one per PM page) that track page hotness are maintained in PM and cached in the memory controller. RaPP [88] and Zhang et al. [114] use a set of queues in the memory controller to estimate write intensiveness and perform page migrations to DRAM. However, these mechanisms propose no wear-leveling solutions for the remaining pages in PM. As such, these mechanisms may still not achieve desired PM device lifetimes. For example, RaPP can achieve a device lifetimes exceeding 3 years only if the cell endurance exceeds 10^9 [88] – insufficient for PCM-based memories with endurance of only 10^7 - 10^9 writes. Moreover, these mechanisms do not support applications that require crash consistency when using PM as storage [77]. Kevlar incurs none of these hardware overheads and uses a novel sampling scheme to estimate wear completely in the OS.

Heterogeneous main memory: Several works [5, 55, 84] manage footprint between DRAM and PM for applications that prefer DRAMs for high performance. These works map heavily and least accessed regions of application footprint to DRAM and PM respectively. Unlike these works [5, 55, 84], Kevlar exploits heterogeneity to reduce PM wear.

Currently, Kevlar operates at a small (4KB) page granularity. However, huge (2MB) pages are increasingly being used to minimize performance penalties of using small pages (due to increased TLB pressure), especially in virtualized systems. Kevlar can be further extended to operate at a huge page granularity. For instance, Kevlar can be integrated with mechanisms such as Thermostat [5] to split a huge page into small pages, monitor write rate at granularity of small pages, and migrate pages between DRAM and PM. We leave evaluation of Kevlar's wear-reduction mechanism and development of shuffling strategies to operate at a huge page granularity to future work.

Other. DCW [119] and Flip-N-Write [21] perform read-compare-write operation to ensure that only the data bits that have changed are written. Bittman et al. [14] proposes data structures aimed at minimizing the number of bit-flips per PM write operation. Ferreira et al. [32] enable eviction of clean cache lines over dirty cache lines at the expense of potentially slowing down future reads to evicted cache lines. Recent works, MCT [25] and Mellow Writes [112], improve the endurance by reconfiguring memory voltage levels and slowing write accesses to the PM. These proposals can achieve high device lifetime but at a significant performance overhead, especially when write latency is critical to application performance [77]. NVM-Duet [69] employs a smart-refresh mechanism to eliminate redundant memory refresh operations thereby reducing PM wear. Others [49, 118] propose solutions to manage wear when using persistent memory technologies to build caches. These techniques are orthogonal to our proposal and can be used in conjunction with Kevlar.

7.2 Wear-leveling mechanisms

Qureshi et al. [87], Zhou et al. [119], Security refresh [92], Online Attack Detection [86] and Start-Gap [85] observe that cache lines within a PM page do not wear out equally and propose mechanisms to remap cache lines for uniform intra-page wear. All of these works rely on additional address indirection mechanisms in hardware. Unlike Kevlar, these mechanisms cannot exploit the heterogeneity of memory systems as discussed earlier in Section 2.2.

Error recovery. DRM [47] and SAFER [93] gracefully degrades PM capacity as memory cells wear out by reusing and remapping failed cells to store data. FREE-p [110] and NVMAlloc [76] leverage ECC and checksum mechanisms to tolerate wear out errors.

8 Conclusion

We have presented Kevlar, a wear-management mechanism for persistent memories. Kevlar relies on a software *wear-estimation* mechanism that uses PEBS-based sampling in a novel approach to estimate dirty cache contents and predict writebacks to PM. It uses a *wear-leveling* mechanism that shuffles PM pages every ~4 hours with an overhead of less than 0.10% achieving up to 31.7× higher lifetime as compared to PM with no wear leveling. Kevlar employs *wear-reduction* mechanism to further extend PM lifetime. It migrates the hottest pages to higher durability memory. Kevlar, implemented in Linux kernel (version 4.5.0), achieves four-year target lifetime with 1.2% performance overhead.

Acknowledgements

We would like to thank our shepherd, Carl Waldspurger, and the anonymous reviewers for their valuable feedback. We are grateful to Akshitha Sriraman, Kumar Aanjaneya, Neha Agarwal, Animesh Jain, and Amirhossein Mirhosseini for their suggestions that helped us improve this work. This work was supported by ARM and the National Science Foundation under the award NSF-CCF-1525372.

References

[1] Aerospike. http://www.aerospike.com/. [Online; accessed 17-Jun-2017].

[2] pmem.io: Persistent memory programming. https://pmem.io/pmdk/.

[3] Reimagining the Data Center Memory and Storage Hierarchy. https://newsroom.intel.com/editorials/re-architecting-data-center-memory-storage-hierarchy.

[4] Memcached - a distributed memory object caching system, 2012.

[5] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[6] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 62–77, New York, NY, USA, 2018. ACM.

[7] ARM. Embedded trace macrocell, 2011. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0014q/IHI0014Q_etm_architecture_spec.pdf.

[8] ARM. Arm architecture reference manual, 2017. https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf.

[9] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1753–1758, New York, NY, USA, 2017. ACM.

[10] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage and recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

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

[12] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 211–224, Berkeley, CA, USA, 2011. USENIX Association.

[13] N. Beckmann and D. Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–75, Feb 2015.

[14] Daniel Bittman, Mathew Gray, Justin Raizes, Sinjoni Mukhopadhyay, Matt Bryson, Peter Alvaro, Darrell D. E. Long, and Ethan L. Miller. Designing data structures to minimize bit flips on nvm. In *The 7th IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, August 2018.

[15] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, 2017. USENIX Association.

[16] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[17] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[18] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2014.

[19] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Vaglis. Rewind: Recovery write-ahead system for in-memory non-volatile data structures. *Proceedings of the VLDB Endowment*, 8(5), 2015.

[20] I-C. K. Chen, C-C. Lee, and T. N. Mudge. Instruction prefetching using branch prediction information. In *Proc. of the International Conference on Computer Design*, 1997.

[21] Sangyeun Cho and Hyunjin Lee. Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of the International Symposium on Microarchitecture*, 2009.

[22] 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 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[23] 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 22nd ACM Symposium on Operating Systems Principles*, 2009.

[24] 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*, pages 143–154. ACM, 2010.

[25] Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Frederic T. Chong. Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in nvms. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 232–244, New York, NY, USA, 2017. ACM.

[26] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. Pdram: A hybrid pram and dram main memory system. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, 2009.

[27] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[28] Kshitij Doshi, Ellis Giles, and Peter Varman. Atomic persistence for scm with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 77–89. IEEE, 2016.

[29] Paul J Drongowski. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. *Advanced Micro Devices*, 2007. http://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf.

[30] 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 9th European Conference on Computer Systems*, 2014.

[31] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 15:1–15:16, New York, NY, USA, 2016. ACM.

[32] Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. Increasing pcm main memory lifetime. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10.

[33] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, June 2005.

[34] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Badgertrap: A tool to instrument x86-64 tlb misses. *SIGARCH Comput. Archit. News*.

[35] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Failure-atomic synchronization-free regions, 2018. http://nvmw.ucsd.edu/nvmw18-program/unzip/current/nvmw2018-final42.pdf.

[36] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 46–61, New York, NY, USA, 2018. ACM.

[37] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software persistent memory. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 319–331, Boston, MA, 2012. USENIX.

[38] SAP HANA. Bringing persistent memory technology to sap hana: Opportunities and challenges, 2016. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2016/20160810_FR21_Caklovic.pdf.

[39] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12):1612–1630, December 1989.

[40] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan. Unified address translation for memory-mapped ssds with flashmap. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 580–591, June 2015.

[41] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, Santa Clara, CA, 2017. USENIX Association.

[42] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 967–979, Boston, MA, 2018. USENIX Association.

[43] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, 2018. USENIX Association.

[44] Intel. Intel microarchitecture codename nehalem performance monitoring unit programming guide (nehalem core pmu). https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf.

[45] Intel. Intel 64 and ia-32 architectures software developer's manual, 2018. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[46] Intel and Micron. Intel and micron produce breakthrough memory technology, 2015. http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.

[47] Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: Building reliable systems from nanoscale resistive memories. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010.

[48] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[49] Yongsoo Joo, Dimin Niu, Xiangyu Dong, Guangyu Sun, Naehyuck Chang, and Yuan Xie. Energy- and endurance-aware design of phase change memory caches. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, 2010.

[50] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. Dfs: A file system for virtualized flash storage. *Trans. Storage*, 6(3):14:1–14:25, September 2010.

[51] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, Feb 2017.

[52] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient persist barriers for multicores. In *Proceedings of the international symposium on Microarchitecture*, 2015.

[53] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for nand flash memory. In *Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '06, pages 161–170, New York, NY, USA, 2006. ACM.

[54] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, 2018. USENIX Association.

[55] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 521–534, New York, NY, USA, 2017. ACM.

[56] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *2007 25th International Conference on Computer Design*, pages 245–250, Oct 2007.

[57] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 297–312, New York, NY, USA, 2018. ACM.

[58] Jesung Kim, Jong Min Kim, S. H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

[59] 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, 2015.

[60] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P.M. Chen, and T.F. Wenisch. Delegated persist ordering. In *Proceedings of the 49th International Symposium on Microarchitecture*, 2016.

[61] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 481–493, New York, NY, USA, 2017. ACM.

[62] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Tarp: Translating acquire-release persistency, 2017. http://nvmw.eng.ucsd.edu/2017/assets/abstracts/1.

[63] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[64] 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*, 2009.

[65] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[66] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 339–353, Santa Clara, CA, 2016. USENIX Association.

[67] H. L. Li, C. L. Yang, and H. W. Tseng. Energy-aware flash memory management in virtual memory system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(8):952–964, Aug 2008.

[68] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[69] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. Nvm duet: unified working memory and persistent store architecture. In *Proceedings of the international conference on Architectural Support for Programming Languages an Operating Systems*, 2014.

[70] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, 2017. USENIX Association.

[71] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti. Laser: Light, accurate sharing detection and repair. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–273, March 2016.

[72] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[73] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.

[74] Paul Menage. Memory resource controller, 2016. http://elixir.free-electrons.com/linux/latest/source/Documentation/cgroup-v1/memory.txt.

[75] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. Enabling efficient and scalable hybrid memories using fine-granularity dram cache management. *IEEE Comput. Archit. Lett.*

[76] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, pages 1:1–1:17, New York, NY, USA, 2013. ACM.

[77] 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.

[78] Faisal Nawab, Dhruva Chakrabarti, Terence Kelly, and Charles B. Morey III. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. Technical Report HPL-2014-70, Hewlett-Packard, December 2014.

[79] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. Telecom application transaction processing benchmark, 2011. http://tatpbenchmark.sourceforge.net/.

[80] T. Nguyen and D. Wentzlaff. Picl: A software-transparent, persistent cache log for nonvolatile main memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 507–519, Oct 2018.

[81] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. Sofort: A hybrid scm-dram storage engine for fast data recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, 2014.

[82] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, New York, NY, USA, 2014. ACM.

[83] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceedings of the 41st International Symposium on Computer Architecture*, 2014.

[84] S. Phadke and S. Narayanasamy. Mlp aware heterogeneous memory system. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[85] Moinuddin K. Qureshi, Michele M. Franchescini, Vijayalakshmi Srinivasan, Luis A. Lastras, Bulent Abali, and John Karidis. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the International Symposium on Microarchitecture*, 2009.

[86] Moinuddin K. Qureshi, Andre Seznec, Luis A. Lastras, and Michele M. Franchescini. Practical and secure pcm systems by online detection of malicious write streams. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, 2011.

[87] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[88] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, 2011.

[89] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, 2016.

[90] D. L. Schuff, B. S. Parsons, and V. S. Pai. Multicore-aware reuse distance analysis. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010.

[91] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 53–64, New York, NY, USA, 2010. ACM.

[92] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S. Lee. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[93] Nak Hee Seong, Dong Hyuk Woo, Vijayalakshmi Srinivasan, Jude A. Rivers, and Hsien-Hsin S. Lee. Safer: Stuck-at-fault error recovery for memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, 2010.

[94] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, 2016.

[95] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: A flexible and fast software supported hardware logging approach for nvm.

In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 178–190, New York, NY, USA, 2017. ACM.

[96] Peter Snyder. tmpfs: A virtual memory file system. In *In Proceedings of the Autumn 1990 European UNIX Users' Group Conference*, pages 241–248, 1990.

[97] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. Aerospike: Architecture of a real-time operational dbms. *Proc. VLDB Endow.*, 9(13):1389–1400, September 2016.

[98] Transaction Processing Performance Council (TPC). Tpc benchmark C, 2010. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5-11.pdf.

[99] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the USENIX Conference on File and Storage Technologies*, February 2011.

[100] 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, 2014.

[101] Haris Volos, Andres Jaan Tack, and Michael M. Swift E. Mnemosyne: Leightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[102] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, Santa Clara, CA, 2017. USENIX Association.

[103] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, 2015. USENIX Association.

[104] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, June 2014.

[105] Xiaojian Wu and A. L. Narasimha Reddy. Scmfs: a file system for storage class memory. In *In Proceedings of the International Conference for High Performance Computing*, 2011.

[106] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the challenges of crossbar resistive memory architectures. In *In Proceedings of the International Symposium on High Performance Computer Architecture*, 2015.

[107] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, 2016.

[108] 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.

[109] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND ssds. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 15–28, Santa Clara, CA, 2017. USENIX Association.

[110] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P. Jouppi, and Mattan Erez. Free-p: Protecting non-volatile memory against both hard and soft errors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, 2011.

[111] H. C. Yu, K. C. Lin, K. F. Lin, C. Y. Huang, Y. D. Chih, T. C. Ong, J. Chang, S. Natarajan, and L. C. Tran. Cycling endurance optimization scheme for 1mb stt-mram in 40nm technology. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 224–225, Feb 2013.

[112] Lunkai Zhang, Brian Neely, Diana Franklin, Dmitri Strukov, Yuan Xie, and Frederic T. Chong. Mellow writes: Extending lifetime in resistive memories through selective slow write backs. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 519–531, Piscataway, NJ, USA, 2016. IEEE Press.

[113] Michael Zhang and Krste Asanovic. Highly-associative caches for low-power processors. In *Kool Chips Workshop, MICRO*, volume 33, 2000.

[114] Wangyuan Zhang and Tao Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures.

In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, 2009.

[115] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[116] 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.

[117] 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 46th International Symposium on Microarchitecure*, 2013.

[118] Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *ACM Trans. Archit. Code Optim.*

[119] Ping Zhou, Bo Zhao, Jun Yang, and Yutao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[120] Yanqi Zhou, Ramnatthan Alagappan, Amirsaman Memaripour, A Badam, and D Wentzlaff. Hnvm: Hybrid nvm enabled datacenter design and optimization. *Microsoft, Microsoft Research, Tech. Rep. MSR-TR-2017-8*, 2017.