Getting More Performance with Polymorphism from Emerging Memory Technologies

Iyswarya Narayanan
Penn State

Aishwarya Ganesan
UW-Madison

Anirudh Badam
Microsoft

Sriram Govindan
Microsoft

Bikash Sharma
Facebook

Anand Sivasubramaniam
Penn State

ABSTRACT

Storage-intensive systems in data centers rely heavily on DRAM and SSDs for the performance of reads and persistent writes, respectively. These applications pose a diverse set of requirements, and are limited by fixed capacity, fixed access latency, and fixed function of these resources as either memory or storage. In contrast, emerging memory technologies like 3D-Xpoint, battery-backed DRAM, and ASIC-based fast memory-compression offer capabilities across several dimensions. However, existing proposals to use such technologies can only improve either read or write performance but not both without requiring extensive changes to the application, and the operating system. We present PolyEMT, a system that employs an emerging memory technology based cache to the SSD, and transparently morphs the capabilities of this cache across several dimensions – persistence, capacity, latency – to jointly improve both read and write performance. We demonstrate the benefits of PolyEMT using several large-scale storage-intensive workloads from our datacenters.

CCS CONCEPTS

• **Information systems → Storage class memory**; **Hierarchical storage management**; *Data compression*; • **Computer systems organization → Cloud computing**; • **Software and its engineering → Memory management**;

ACM Reference Format:
Iyswarya Narayanan, Aishwarya Ganesan, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Anand Sivasubramaniam. 2019. Getting More Performance with Polymorphism from Emerging Memory Technologies. In *Proceedings of The 12th ACM International Systems and Storage Conference (SYSTOR '19).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3319647.3325826

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SYSTOR '19, June 3–5, 2019, Haifa, Israel
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6749-3/19/06…$15.00
https://doi.org/10.1145/3319647.3325826

1 INTRODUCTION

Data-intensive applications like key-value stores, cloud storage, back-ends of popular social media services that run on cloud datacenters are latency critical, and require fast access to store and retrieve application data. The memory and storage requirements of these data-intensive applications are fast outpacing the limits of existing hardware. For instance, growing data volume for these applications pose increasing pressure on the capacity needs of the memory and storage subsystems, and adversely impact application latency. This is further exacerbated in applications with persistent writes to storage (file writes/flushes, and msyncs) as even the fastest storage resource in contemporary servers (SSD) is an order of magnitude slower than its volatile counterpart (DRAM). Further, cloud applications are diverse in their resource requirements (e.g. memory and storage working set sizes). In contrast, existing memory and storage resources are rigid in their characteristics in terms of persistence capability, accesses latency, and statically provisioned capacity. Therefore, datacenter operators are faced with the question of how to better serve data-intensive cloud applications that pose diverse resource requirements across multiple dimensions.

Emerging memory technologies offer better performance along several dimensions (e.g. higher density compared to DRAM [63, 64], higher performance than SSDs [4, 5, 9], or even both). Realizing their potential, prior works [18, 28, 70] have exploited Non-Volatile Memory (NVM) technologies to jointly improve both read and write performance[1] of an application by relying on their byte addressability and non-volatility. NVMs that offer higher density than DRAM benefit applications bottle-necked by memory capacity, and improve their read performance. And, their non-volatility benefits applications bottle-necked by writes to the persistent storage. While these solutions offer attractive performance for both reads and writes, these are insufficient for widespread adoption in current datacenters for two main reasons.

[1] We refer to all persistent writes as writes.

First, many of these proposals consider file systems that reside entirely on NVMs[39, 75, 77] to benefit both reads and writes. Note that the cost per unit capacity of NVMs is still an order of magnitude higher than SSDs. Therefore, replacing a large fleet of SSDs with NVM will be cost prohibitive for applications requiring massive storage capacities. Instead, we require a design that requires only a limited NVM capacity.

Second, to extract high performance, many of these proposals require extensive code changes to port the application [18, 28, 70], operating system [39, 77] or the filesystem [74–76] which may not be immediately feasible at scale. These limit faster and wider on-boarding of these technologies in current datacenters. Therefore, we require transparent mechanisms and policies that can effectively utilize limited NVM capacity available within each server.

While there are several proposals that enable transparent integration of limited capacity NVMs into existing memory and storage stack [15, 63, 79], they are insufficient in the context of cloud datacenters. The reason is, they focus on single aspect whereas cloud applications pose diverse set of requirements which can be met effectively by tapping into the potential of these technologies to morph across several dimensions – persistence, latency, and capacity.

NVMs with performance similar to that of DRAM and persistence similar to that of SSDs can morph dynamically either as persistent block cache to SSDs or as additional byte-addressable volatile main memory – we call this functional polymorphism. Existing transparent solutions focus on one function by employing these devices either as storage caches for SSDs [15, 16, 50] or as extension of DRAM to augment main memory capacity[27, 63, 79]. Employing NVM as storage cache accelerates persistent accesses to the storage device. But, these incur additional software overheads for read accesses [17, 36, 73], despite their ability to allow direct hardware access using byte addressability.

Transparently integrating NVM as byte addressable memory is beneficial only for volatile memory access, and does not benefit any persistent writes to the storage medium despite being non-volatile. However, dynamically re-purposing all or part of NVM as memory or storage can benefit a diverse set of applications. Unlike existing such proposals [39, 67], we target a design that neither requires high NVM provisioning costs to replace entire SSD with NVM as file system storage medium nor requires changes to existing file system. Therefore, we employ limited NVM capacity as a storage cache to SSD, where the challenge is to carefully apportion this resource between competing memory and storage access streams, whereas in existing proposals it is only determined by one of those.

There exists other trade-offs in emerging memory technologies that offer fast access times at lower density or slow access times at higher density depending on data representation (e.g. Compressed memory [68], SLC vs. MLC [62]). We refer to this as representational polymorphism. This poses opportunity to improve tail performance for latency sensitive applications that are limited by fixed latency and capacity of existing hardware. While this has been studied previously [23, 49, 62, 68], a holistic solution that exploits polymorphism across several dimensions is missing.

Given the rigidity of existing hardware and the diversity of cloud applications, it is essential to fully benefit from the polymorphic capabilities of these technologies to effectively serve applications. Towards that, we employ NVM as a cache to the SSD, and exploit its capability to morph in its function (as volatile vs. non-volatile cache) and data representation (high density/slow vs. low density/fast). But, to benefit from polymorphism, we need to navigate the trade-offs along several dimensions to adapt the cache characteristics based on the needs of individual applications. Moreover, we need mechanisms to transparently enforce these capabilities without requiring any application, OS or file system level changes.

Contributions: Towards these goals, we present:

- A detailed characterization of production storage traces to show that cloud operators can effectively serve heterogeneous applications by exploiting polymorphic emerging memory technology based cache in front of SSD.
- A functional polymorphism solution that allows transparent integration of limited capacity NVM as persistent block-based write-back cache to SSDs. It dynamically configures available cache capacity in volatile and non-volatile forms to simultaneously improve performance for reads and persistent writes.
- A representational polymorphism knob that helps tune the latency and capacity within each function to further optimize performance for applications with working set sizes exceeding the physical capacity of the resources.
- A dynamic system that traverses the design space offered by both functional and representational polymorphism, and apportions resources across persistence, latency, and capacity dimensions to maximize application performance.
- A systematic way of reaching good configurations in the above dynamic system: starting with the full capacity in one single form that addresses the most significant bottleneck first and then gradually morphing into other forms until the performance increases, enables a way to search for the ideal configuration systematically.
- A prototype in real hardware available in today's datacenters as a transparent memory and storage-management runtime in C++ for Linux. Our solution *does not require any application, OS, or file system changes* for its usage. Our

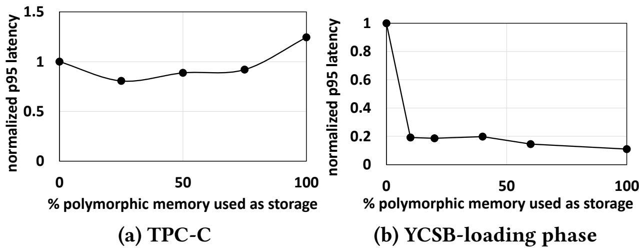


Figure 1: Applications benefit from different capacities of NVM in memory and storage. This split is different across applications.

experiments with the prototype show significant improvements to the tail latencies for representative workloads, by up to 57% and 70% for reads and writes, respectively.

2 THE NEED TO EXPLOIT POLYMORPHISM

Our goal is to aid datacenter operators to effectively serve data-intensive cloud applications using emerging memory technologies without incurring significant capital investments and intrusive software changes. While there are prior proposals that meet these constraints, they often extract sub-optimal performance as they focus on one aspect, whereas these resources can morph across several dimensions - persistence, capacity and latency. In this work, we consider battery-backed DRAM (BB-DRAM) [4, 9, 54] and fast memory compression as the Emerging Memory Technologies (EMTs), backed up by SSDs [58], as seen in today's datacenters. It presents us with the following two trade-offs[2].

2.1 Why Functional Polymorphism?

Much like other NVMs, BB-DRAM can function both as a regular volatile medium and as a specialized non-volatile medium using an energy storage device (e.g., ultra-capacitor). The battery capacity is used to flush data from DRAM to SSDs upon power loss. This can be dynamically configured to flush only a portion of (non-volatile) DRAM with the remaining used as volatile memory, thereby creating a seamless transition between these modes. The battery-backed portion offers persistence at much faster latency than SSDs.

One can transparently add such NVM to existing servers as an extension of main memory or as a block-granular cache to the SSD. While the former implicitly benefits reads, the latter is expected to implicitly benefit both reads and persistent-writes. However, note that *all* reads to NVM would then incur software penalty by having to go through the storage stack, despite their ability to allow direct CPU-level access for reads. So, explicitly partitioning it between the volatile memory and storage tiers would offer better read performance.

[2]These trade-offs exist for other emerging memory technologies as well.

In Fig. 1, we show the benefits of such explicit partitioning using TPC-C, a ready heavy workload as well as a write-heavy YCSB workload using a server provisioned with 24 GB volatile DRAM and 24 GB non-volatile BB-DRAM. We manually varied the non-volatile capacity between an SSD block cache (managed by dm-cache [69]) and physical memory (managed by OS). Fig. 1a shows that TPC-C suffers a loss of 30% in performance when using all of the BB-DRAM as SSD-cache compared to when using it as main memory. In contrast, Fig. 1b shows that using BB-DRAM as SSD-cache is optimal for the write-heavy YCSB workload.

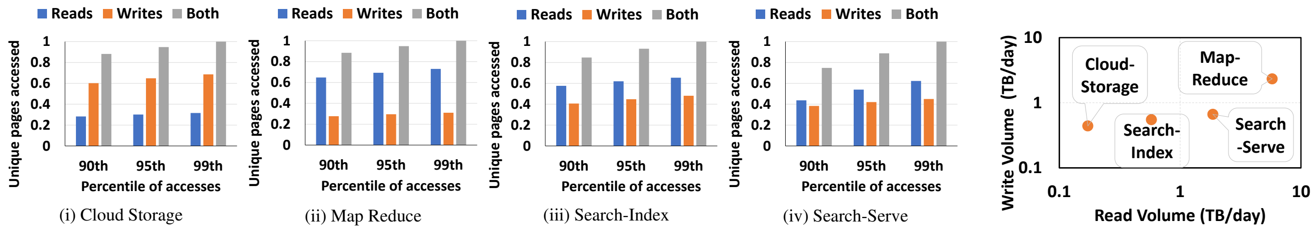
The results demonstrate that different "static" splits not only change the performance of applications differently, but also that the best split is different across applications. Furthermore, the results also show that the extremes are not always optimal. For instance, we see that the write-intensive YCSB-loading phase requires entire BB-DRAM capacity in the storage tier, whereas read-intensive TPCC requires a split of 75% and 25% between memory and storage to meet both read and persistent-write requirements satisfactorily.

This is especially important in cloud datacenters, as we observe that most real-world applications have a heterogeneous mix of reads and writes. We show this by analyzing the storage access characteristics of four major production applications from our datacenters: (i) A public cloud storage (Cloud-Storage); (ii) A map-reduce framework (Map-Reduce); (iii) Search data indexing application (Search-Index), and (iv) Search data serving application (Search-Serve).

We observe that file system read and write working set sizes vary across applications. Fig. 2a shows that the total number of unique pages required to serve 90th, 95th, and 99th percentile of access for reads and writes respectively, normalized to total unique pages accessed for a duration of 30 minutes. As we can see, the working set sizes vary across applications. Similarly, Fig. 2b captures the difference in the access intensities of total read and write volumes per day; the read and write volumes varies between 0.18 TB to 6 TB and 0.45 TB to 4 TB, respectively. Moreover, these access intensities vary over time; Fig. 3 shows this in Search-Serve for a 12 hour period as an example. Together, these motivate the need to exploit the ability of these technologies to morph dynamically between storage and memory functions to effectively serve diverse set of applications.

2.2 Why Representational Polymorphism?

In addition to the above trade-off, there exist trade-off between data representation and access latency in emerging technologies. For example, fast memory compression (using ASICs [2]) enable larger capacity using compressed high-density data representation. However, it incurs longer latency to work with the compressed representation. Thus, the



(a) Heterogeneity in read vs persistent-write capacity requirements to serve 90th, 95th and 99th percentile of read, persistent-write and total accesses over a 30 minutes window.

(b) Heterogeneity in the volume of read and persistent-write accesses.

Figure 2: Need for application aware memory and storage resource provisioning.

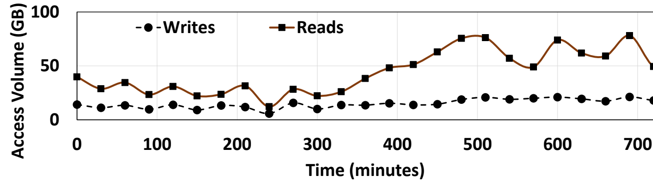


Figure 3: Temporal variations in read and persistent-write access intensities in Search-Serve.

same memory can provide either the higher capacity (with compression) or, the faster access (without compression).

This is especially important in the context of tail sensitive applications constrained by static capacities and latencies of today's hardware. For instance, existing servers have limited capacities of DRAM (in the order of 10s of GBs) and are backed up using SSDs (in the order of 100s of GBs) or over the network which are 500× slower. The static capacities of DRAM, non-volatile EMT, and SSDs form strict latency tiers. Consequently, the tail latency of data-intensive applications with working sets that do not fit in the DRAM is now determined by the slower tier. We illustrate this using Fig. 4a. It plots latency vs probability of pages accessed at this latency when the working set of an application is spread between two latency tiers. Here, the tail latency of the application is determined by the SSDs which are order of magnitudes slower than DRAM. One way to optimize the tail performance in such systems is to increase the effective capacity of the faster tier using high density data representation (e.g. compression) while maintaining latency well below that of the slowest tier using fast compression techniques as illustrated in Fig. 4a.

We observe that real-world applications in our datacenter can increase their effective capacity by 2–7× (see Fig. 4b) using compression while incurring significantly lower latency compared to accessing the slowest tier. Fig. 4c shows that in contrast to SSDs which incur more than 80μs and 100μs for reads and writes, compressed DRAM based memory incurs 4μs for reads (decompression), and 11μs for writes (compression). This can be further tuned based on the application needs by exploiting parallelism.

While existing works have studied such latency vs. capacity trade-off only in isolation, the additional challenge for us here is to identify which functional layer can benefit from exploiting representational polymorphism. Towards that, we explore a holistic approach for flexible memory and storage provisioning within a server to exploit both kinds of polymorphism, not only for static application configurations but also for dynamic variations in application behavior.

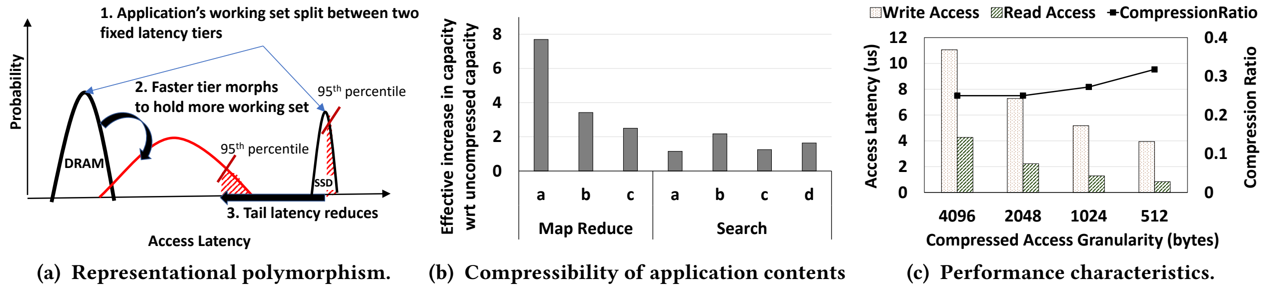
3 POLYMORPHIC EMT DESIGN

Our goal is to identify the optimal configuration of a limited-capacity EMT across its various polymorphic forms, both spatially and temporally, to extract maximum application performance. The challenge here is to navigate a huge search space as jointly determined by the polymorphism knobs. We use the following insight to decouple the search space: start with the full capacity in one single form that addresses the most significant bottleneck in the system. Then gradually morph into other forms to further improve effectiveness.

3.1 Using Functional Polymorphism

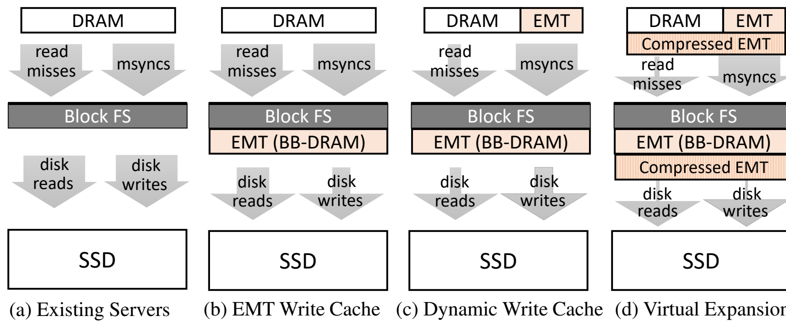
In today's servers (Fig. 5(a)), all persistent writes (msyncs) must hit the SSD synchronously, while reads have some respite since only those that miss DRAM buffer cache hit the SSD. Also, SSDs by nature have higher tail latencies for writes compared to reads. Fig. 6 shows that reads to be 2× faster than the writes even in their average latencies, and up to 8× faster at the 95th percentile for the *fio* benchmark [14]. This can be alleviated by using BB-DRAM as a persistent cache, to avoid accessing the SSD in the critical path.

However, in a persistent storage cache (Fig. 5(b)), both reads (those that miss in the DRAM buffer cache) and writes to the SSD will be cached. But, caching SSD reads in this storage layer is sub-optimal, as going through software adds significant latency penalty [17, 36, 73]. Ideally, such reads should be served directly via CPU load/store by bypassing the software stack. Fig. 5(c) shows such a system, where the BB-DRAM based write cache is just large enough to absorb bursts of application writes, buffer them, and write data to the



(a) Representational polymorphism. (b) Compressibility of application contents (c) Performance characteristics.

Figure 4: Representational polymorphism (compression) to increase working set in the fast tier. Latencies measured on 2.2 GHz Azure VM.



(a) Existing Servers (b) EMT Write Cache (c) Dynamic Write Cache (d) Virtual Expansion

Figure 5: PolyEMT design: (a) Today's servers have a high read and write traffic to SSD in the critical path. (b) We begin with EMT as Write-Cache to reduce persistent writes to SSD. (c) We then re-purpose underutilized Write-Cache blocks to extend DRAM capacity reducing DRAM read misses. (d) We then employ EMT compression to further reduce SSD traffic.

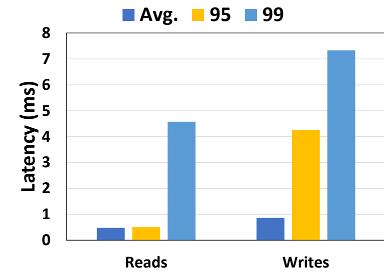


Figure 6: Read/Write asymmetry: 4K reads incur an average of 400μs with 95th percentile at 500μs, whereas average 4K writes incur as high as 850μs with 95th percentile at 4200μs at high load.

SSD in the background such that the applications do not have to experience the high write-tails of SSDs in the critical path. Its remaining capacity can be morphed to extend volatile main memory to accelerate reads. While the system shown in Figure 5(c) conceptually exploits functional polymorphism, its benefits depends on the ability to match the volatile and non-volatile cache sizes to application needs as discussed in Section 2.1 (see Fig. 2). Further, as applications themselves evolve temporally (see Fig. 3), these capacities need to be adapted to continuously meet changing application needs.

Therefore, our solution (PolyEMT) starts with the entire emerging memory capacity as write-cache, while gradually re-purposing some of its capacity to function as volatile (even though physically non-volatile) main memory. To minimize performance consequences of such a dynamic reduction in the write-cache capacity, we need to effectively utilize available write-cache capacity. Towards that, PolyEMT uses a perfect LRU policy to identify blocks for revocation. It periodically writes them back to the SSD in the background to make those blocks free and adds them to the free list.

Simultaneously, the re-purposed EMT pages (moved out of the free list of the storage cache) serve as DRAM extension managed by the OS. The virtual memory management component of the OS is well-equipped to effectively manage the available main memory capacity, and even handle heterogeneous latency tiers [40, 71]. Such buffer management policies maintain only the most recently accessed pages (volatile accesses) in memory, to improve access latency.

Hence, the sweet spot for partitioning the EMT is identified by incrementally revoking least recently written write-cache pages and re-purposing them as volatile main memory by probing application-level request latencies. PolyEMT stops the re-purposing when the performance of the application stops increasing; this is the point where the read bottleneck is mitigated and persistent writes re-start to impact the application's performance, as shown in Fig. 7. We observe a convex behavior, as in Fig. 11, for a wide range of applications. This is because LRU is a stack algorithm which does not suffer from Belady's anomaly, i.e., the capacity vs. hit rate has a monotonic relationship for both memory capacity and write cache capacity under steady access conditions. Therefore, a linear (incremental) search is suffices to identify the sweet spot between volatile and non-volatile capacities.

3.2 Using Representational Polymorphism

PolyEMT leverages representational polymorphism to virtually expand capacities when the read and write working set sizes of application exceed available physical memory and write-cache capacities. The compressed data representation reduces the capacity footprint of data (refer Fig. 4); thus more data can be held in EMT instead of being paged

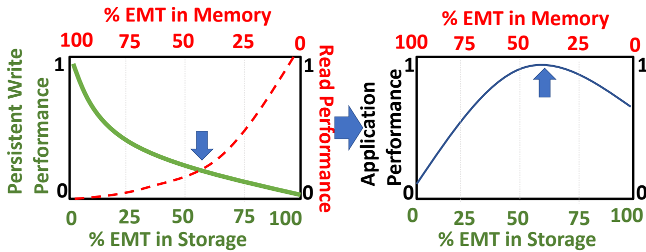


Figure 7: To partition EMT between memory and storage, incrementally re-purpose write-cache pages as volatile memory until performance start to decrease. This point balances the impact of persistent writes and volatile reads on application performance.

to the slowest tier (i.e., SSD). Towards that, PolyEMT uses a caching hierarchy as shown in Fig. 5(d) where 3 latency tiers – EMT, Compressed EMT and SSD – exist in both volatile and non-volatile forms. Here, a capacity bottleneck at the fast tier will move least-recently-used pages to the slower representation to make room for hot pages. And, a hit access at the slow tier will result in its conversion to fast representation.

Unlike existing compressed memory systems [12, 60, 68] and tiered storage systems [42, 59], we face the additional challenge of deciding where to spend the limited compute cycles available for compression/decompression - in the write cache or in the volatile memory? Exploiting representational polymorphism in either modes is viable when the benefits of holding more data using slower representation are higher than the benefits of holding fewer data using a faster representation. This is conceptually similar to the capacity partition problem of the functional polymorphism.

However, unlike functional polymorphism, asymmetry in read and write accesses are not important. This is because, the penalty of a cache miss is the same in both functions (the read or the write cache), which is primarily determined by the software latency to go and fetch the page/block to the faster tier. PolyEMT uses this key insight when determining the capacities of each tier in representational polymorphism – by using a combined faster/slower representation of both read and write caches rather than exploiting them independently (as in functional polymorphism). This has the natural side effect of using the minimum compute resources and EMT capacity to meet a target tail latency and throughput.

PolyEMT uses LRU based ranking of accesses across *all* the non-compressed pages in both volatile memory and write-back cache to decide data movement between the faster and slower tiers. Note that when using representational polymorphism in volatile main memory, PolyEMT does not have complete information on the data accessed in the fast tier - hardware loads and stores bypasses software stack to track exact LRU. Instead, PolyEMT uses page reference information in the page table and uses an approximate LRU algorithm in the fast tier (i.e., Clock [21, 37]). This information is used to not only identify EMT pages to move from memory to storage, but also to identify DRAM pages in volatile memory that can benefit from compression. The best split occurs when there is sufficient fast tier capacity to serve hot data accesses while optimizing the tail performance with the capacity reclaimed from using compressed representation.

To summarize PolyEMT design, the first three optimization steps - write-back cache, functional and representation polymorphisms are applied one after another sequentially, and the LRU-based capacity management is used in all components to cope with limited capacity and latency heterogeneities of the underlying resources. PolyEMT first exploits functional polymorphism and then within each of the functions it exploits representational polymorphism. The system re-configures the memory and storage characteristics to dynamic changes within an application. Once the change is detected, the PolyEMT runtime begins with the EMT as write-cache and finely tunes the capacities and latencies of memory and storage resources based on the current needs.

4 PROTOTYPING POLYEMT

We next describe the key aspects of PolyEMT prototype.

PolyEMT API: One of our goals is to retain application read/write API semantics to aid in faster and seamless onboarding of EMTs. Our prototype modifies the behavior of the traditional `mmap` and `msync` APIs commonly used in many cloud storage applications [1, 6–8]:

- `mmap(filename)` maps a file in persistent storage (SSD) into the application's virtual memory. This allows directly access to the mapped region using loads/stores, avoiding software overheads of read/write system calls.
- `msync(address, length)` invocation guarantees persistence of writes to the application's virtual memory. Modified memory pages in the application's mapped address space within the given *length* offset is persisted immediately as part of this API call. Typically, such `msync` operations are often bottlenecked by the writes to SSD. EMT used as a transparent persistent block cache below the file system layer can help improve `msync` performance.

PolyEMT Runtime: We implement PolyEMT logic as an application-level runtime without requiring any changes to application, file system, and OS. The initial allocation of DRAM, EMT/BB-DRAM, and SSD for an application is specified via a global configuration. The runtime tunes these allocations based on the instantaneous needs of the application, using the following modules for resource management.
Physical resource managers: To manage physical capacity, PolyEMT uses a DRAM manager and a BB-DRAM manager which maintains the list of free pages in DRAM and BB-DRAM, respectively. The DRAM and BB-DRAM memory regions allocated to PolyEMT are pinned to physical frames.

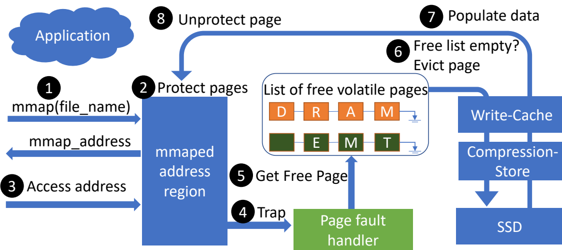


Figure 8: PolyEMT operation for volatile accesses

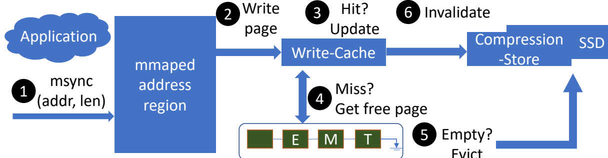


Figure 9: PolyEMT operation for persistent accesses

Buffer-Cache: This cache serves as the set of DRAM and BB-DRAM pages used as volatile memory by the application, accessible by the CPU directly via loads/stores.
Write-Back Cache: This cache is a block storage device which caches persistent writes to SSD (write-back). The application `msyncs()` calls are served using this Write-Cache.
Compression-Store: This resides in BB-DRAM so it can serve as a slow (but higher capacity) tier for both volatile buffer cache and persistent write-back cache. It employs dictionary-based Lempel-Ziv compression [81] at 4KB granularity as it provides higher effective capacity (compared to pattern-based ones [12, 60]). To handle the resulting variable-sized compressed pages which increase fragmentation, we compress 4K pages down only to multiples of a fixed sector size by appending zeros when needed. We determine the best sector size to be 0.5 KB based on the distribution of compression benefits for applications analyzed in Sec. 2.2.
Enabling fast access to volatile memory. PolyEMT places the most frequently read pages from the SSD in DRAM and the EMT/BB-DRAM extension of volatile memory. The following process shown in Fig. 8 explains how this works: An application initially maps a file from the persistent storage (SSD) into main memory ❶ using `mmap`. PolyEMT creates an anonymous region in virtual memory without allocating physical memory resources, set its page protection bits ❷, and returns a virtual address. When the application first accesses any page in this virtual address region ❸, it incurs a page fault ❹, which is handled by a user-level fault handler (using `sigaction` system call).

The page fault handler obtains a free volatile page from the physical memory manager ❺ and maps its physical address to the faulting virtual address referenced by the application using a kernel driver for managing application page table in software [34, 41]. In case the free list is empty, PolyEMT evicts a page ❻. Next, to populate the contents of the page, it brings in the data to the relevant physical page by searching from slower tiers in the order of Write-Cache, Compression-Store (if used) and SSD ❼. It then unsets the protection bits in page table and returns the control to the application ❽.

Pages are evicted from the buffer-cache as follows. A least recently used page is chosen for eviction based on an LRU-like Clock [21, 37] policy implemented using the `referenced bit` in the page table which is reset periodically. Eviction maintains the look-up order of Write-Cache, Compression-Store, and SSD during page faults. Towards that, dirty pages that are already present in write-cache (from a previous `msync` call to the page) are updated in place. Otherwise, they are directly evicted to the SSD. The process is similar when using compression. But, even clean pages are evicted to the compression store.
Handling persistent writes. To persist a page, applications call `msync()` with a virtual address and length, as shown in Fig. 9 ❶. PolyEMT handles this call by writing ❷ the pages in this region from Buffer-Cache to Write-Cache ❸. For write misses at the write cache, it allocates a new page from the free list of non-volatile pages ❹; in rare cases when there are no free pages ❺ it evicts a least-recently-used page to the slower tier ❻ (either Compression-Store or SSD).

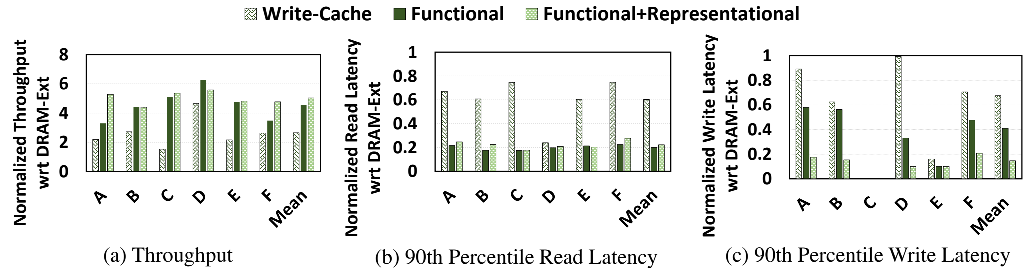
The candidate for eviction is chosen based on LRU for persistent writes in the Write-Cache. Note that, much like traditional `msync`, PolyEMT does not remove the synced pages from the Buffer-Cache unless they were retired implicitly by the higher tier. However, for correctness, the virtual addresses involved are write-protected first, written to the Write-Cache, marked as clean, and then unprotected again. Pages flushed from compressed volatile memory to Write-Cache are maintained in a compressed state until further read operations to save resources.
Handling dynamic capacity variations: PolyEMT dynamically manages the capacity of Buffer-Cache, Write-Cache, and Compression-Store by pursuing two knobs at runtime – the knob that splits BB-DRAM between volatile and non-volatile layers and the knob that controls how much BB-DRAM is compressed across both volatile and non-volatile layers. The runtime described in the previous section sweeps through different values of these two knobs and moves the system to higher-performance configurations – detected transparently by monitoring latencies of SSD operations. It maintains the configurations until such latencies change significantly (a configurable threshold).

5 EVALUATION

We run all experiments using Azure VMs running Linux 4.11 on E8-4s_v3 instances (4 CPU cores, 64GB memory and 128GB SSD at 16000 IOPS). We emulate a part of DRAM as BB-DRAM using Linux persistent memory emulation [3]. In addition to using the workloads presented in Sec. 2, we use

Bench mark	Properties
A	50% reads, 50% updates
B	95% reads, 5% updates
C	100% reads
D	95% reads, 5% inserts
E	95% scans, 5% inserts
F	50% reads, 50% read-modify-writes

Table 1: YCSB benchmark characteristics.



(a) Throughput (b) 90th Percentile Read Latency (c) 90th Percentile Write Latency

Figure 10: Performance of different transparent EMT integration designs normalized to DRAM-Ext.

the YCSB [20] cloud storage benchmark (see Table 1) on Redis [10] populated with 1 million 1KB records, and we report the performance for 1 million operations. We use a persistent version of Redis built using the file `mmap` interface that we transparently hijack to invoke PolyEMT's `mmap()/msync()` APIs. We evaluate the performance of PolyEMT on the metrics of application throughput, read and write tail latencies.

5.1 PolyEMT convergence time

We first study the optimal allocations of the EMT between memory and storage cache under an offline search and PolyEMT for the storage traces analyzed in Sec.- 2. The online run time of PolyEMT converges to the same allocation of BB-DRAM between memory and storage functions compared to the optimal partitions identified offline. Table 2 shows that it takes 4 to 8 minutes to identify the partition sizes, with read-intensive applications requiring longer time as more of the write-cache is re-purposed as memory.

Storage Trace	Time (min.)
Cloud-Storage	4
Map-Reduce	7.5
Search-Index	6
Search-Serve	8

Table 2: Convergence time.

5.2 Performance benefits of polymorphism

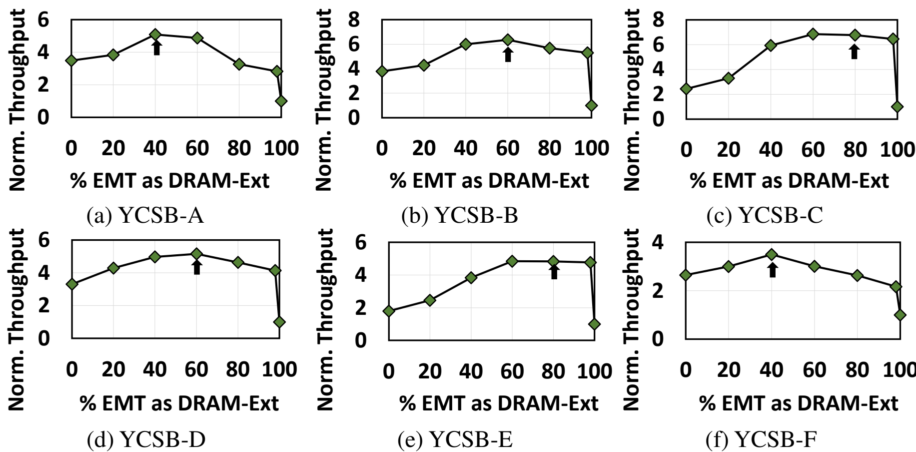
We study the performance benefits of polymorphism using YCSB benchmark. We use a server provisioned with capacities of DRAM (26 GB) and BB-DRAM (6 GB) for a total of 32GB. The application's dataset size ($\approx$ 38 GB) exceeds the capacity of these memories. We study the following designs to transparently utilize limited EMT available in the server[3]:
1. DRAM-Ext: EMT serves as main memory extension.
2. Write-Cache: EMT serves as persistent write-back cache.
3. Functional-only – performs a linear search to partition EMT between memory and storage cache functions.
4. Functional+Representational – partitions EMT between memory and storage functions and also identifies the fast and slow tier capacities within each function.

Impact on application performance: We first study the impact of these designs on application throughput for YCSB benchmarks in Fig. 10(a). The x-axis shows the benchmark, and the y-axis shows its throughput normalized with respect to the DRAM-Ext policy. We find that using the non-volatile BB-DRAM as Write-Cache provides an average speed-up of around 2.5× over using it as a memory extension. This implies that even when using this resource readily without any software modifications, adding it to the most bottlenecked function (i.e. persistent msyncs) delivers better performance. Further partitioning it between memory and storage by exploiting functional polymorphism improves it by up to 70%. Adding representational polymorphism improves performance by 90% compared to Write-Cache policy. This indicates the importance of exploiting polymorphic properties of EMTs to derive higher performance from hardware infrastructure by adapting their characteristics at runtime.

We next present the impact on the tail performance of these benchmarks in Figures 10(b) and 10(c). The x-axis presents the YCSB benchmark and the y-axis presents 90th percentile tail latency for writes and reads, respectively, normalized to that of DRAM-Ext. As can be seen, the tail latency for reads and writes when using Write-Cache reduces by 40% and 36%, respectively, when compared to EMT as DRAM-Ext. In contrast, employing functional polymorphism reduces read tail latency by an average of 80%. This reduction is significant across all benchmarks as more of their working size fits in memory and access to EMT memory extension is done via load/stores without any kernel/storage stack overhead. Compared to read performance, the reduction in write latency at tail is only modest, as the write misses at the write-cache still has to go to the slowest tier (i.e. SSD). However, by incorporating both functional and representational polymorphism write performance at tail improves significantly, with an average latency drop of 85%. Workload C, being read-only is not included for the write latency results in Fig. 10(c).

Resulting memory and storage configurations: We next study the resulting configuration of memory/storage resources for individual applications. Fig. 11 shows the application performance when exploiting functional polymorphism. The x-axis shows the % of EMT morphed as memory and the y-axis shows the average application performance. We see that performance initially increases as more EMT is converted to memory, and beyond which it starts to decrease. As

[3]As the data set size exceeds BB-DRAM capacity, transparent NVM based file system solutions are infeasible.



(a) YCSB-A (b) YCSB-B (c) YCSB-C

(d) YCSB-D (e) YCSB-E (f) YCSB-F

Figure 11: Exploiting functional polymorphism only. PolyEMT search space when traversing different EMT capacity splits between memory and storage.

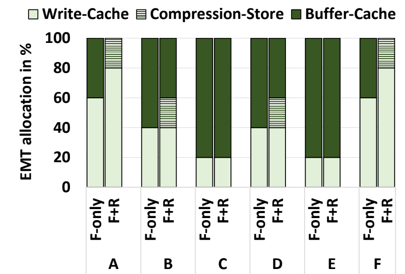
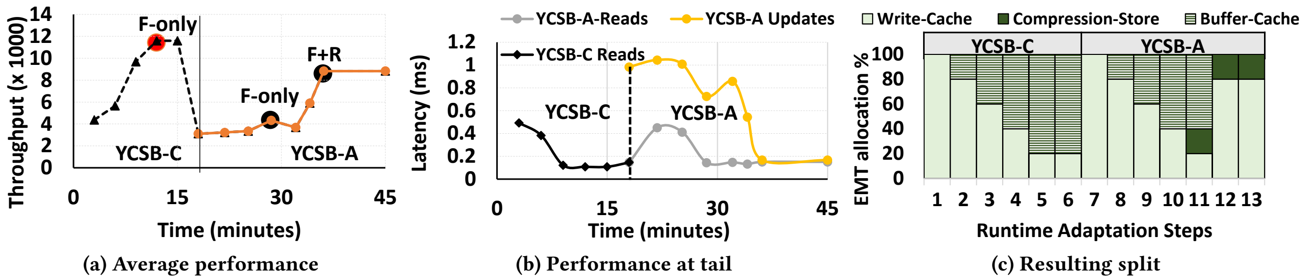


Figure 12: Comparison of ideal EMT split across Buffer-Cache, Write-Cache and Compression-Store between functional polymorphism only (F-only) and functional+Representational polymorphism (F+R).



(a) Average performance (b) Performance at tail (c) Resulting split

Figure 13: Adapting to dynamic changes as YCSB transitions from read-only YCSB-C workload to a read-write YCSB-A workload. PolyEMT identifies YCSB-C benefits from direct access to volatile EMT pages via load/store, where-as YCSB-A benefits from 80% of EMT as Write-Cache and 20% as Compression-Store within twenty minutes. F-only: Functional-only; F+R: Functional+Representational.

expected, the best split (indicated with up-arrow) is different across these applications as expected. However, as the search space is convex with a single maximal point, linear search finds the ideal split with regard to overall throughput.

Fig. 12 presents the partitions identified across Write-Cache, Compression-Store, and Buffer-Cache. Exploiting functional polymorphism alone suffices for read intensive YCSB-C and YCSB-E; this is because the re-purposed capacity fits the entire working size in DRAM. In contrast, benchmarks with write operations employ representational polymorphism as their combined working set for reads and writes exceed the total physical capacity. e.g., YCSB-A and YCSB-F use representational polymorphism with 20% of EMT in compressed form and the rest as write-cache. This shows the ability of the PolyEMT to tune memory and storage resource characteristics based on the application needs.

5.3 Adapting to dynamic phase changes

We next illustrate run time re-configuration using PolyEMT in Fig. 13 for a server running YCSB-C. PolyEMT runtime begins with a full Write-Cache, and subsequently exploits functional polymorphism. As YCSB-C is read-only, direct access to volatile BB-DRAM pages via hardware increases its throughput from 4K operations per second to 11.5K operations per second (F-only for YCSB-C in Fig. 13a) resulting in 80% of BB-DRAM as volatile memory extension and 20% in Write-Cache (refer to runtime adaptation step-5 in Fig. 13c).

At this point, the server gets a burst of writes as its phase changes to 50% reads and 50% updates (YCSB-A). So, PolyEMT starts with a full Write-Cache and exploits functional polymorphism, until it reaches a split of 40% EMT in storage and 60% in memory. This partition balances the relative impact of the reads and writes and achieves highest throughput (F-only for YCSB-A in Fig. 13a). However, PolyEMT observes that the tail latency of update requests are much higher than volatile reads as shown in Fig. 13b at around 30 minutes time in x-axis. At this point, PolyEMT exploits representational polymorphism within the storage cache to alleviate the high update latency. It identifies 20% of total EMT pages in compressed form to have better tail performance without impacting the average performance. PolyEMT further re-apportions EMT between non-volatile region (80%) and the shared compressed region (20%) to bring down the tail latency of both reads and writes within 180μs. This demonstrates the ability of PolyEMT to adapt to dynamic changes.

5.4 Cost benefits of polymorphism

We next study the cost benefits of PolyEMT by comparing the following provisioning strategies:
(i) Application-specific static provision: Each application runs on a dedicated server with the DRAM, BB-DRAM and SSD capacity right-sized to meet the the application needs. This hardware right-sizing is primarily suited for applications that run on private datacenters to reduce costs. But, it is impractical at scale. Hence, we consider two other strategies that provision uniform set of physical servers for all applications.
(ii) Application-oblivious static provision: This employs an identical set of servers to meet the needs of all applications. Here, both DRAM and BB-DRAM capacities are sized based on the maximum capacity needs across all applications.
(iii) Application-oblivious dynamic provision: Here, the servers are provisioned with BB-DRAM based on the peak Write-Cache needs of the applications. However, the servers are under-provisioned in their DRAM capacity as BB-DRAM can also function as DRAM using PolyEMT. This strategy can serve diverse applications using a uniform fleet as in the second design, but with reduced upfront capital costs.

We consider SSD to cost 25 cents per GB [65], DRAM to cost $10 per GB [32], and BB-DRAM to cost 1.2× to 2× compared to DRAM [25, 55]. Table 3 presents the datacenter level total cost of ownership (TCO) [31] when hosting the workloads under each of these designs. As expected, right sizing the server for each application results in the lowest TCO – which increases by 1.19% even if the cost of BB-DRAM increases from 1.2× to 2× as DRAM. Although attractive from cost perspective, this strategy is difficult in practice due to diversity applications. Among the practical alternatives, App-Oblivious-Dynamic translates to 1.97% reduction in datacenter TCO compared to App-Oblivious-Static.

Provisioning Strategy	Cost-ratio (BB-DRAM/DRAM) 1.2×	1.5×	2×
App-Specific-Static	0	0.435%	1.19%
App-Oblivious-Static	2.18%	2.925%	4.15%
App-Oblivious-Dynamic	0.205%	0.947%	2.18%

Table 3: Datacenter TCO under various provisioning strategies and cost ratios. Baseline: App-Specific-Static with cost-ratio of 1.2×.

6 RELATED WORK

Datacenter resource provisioning: To overcome the challenges in scaling the physical capacity of datacenters [66, 72], prior efforts optimized the static provisioned capacity [29, 56, 57], enable dynamic sharing of capacity using novel techniques such as re-purposing remote memory [52], memory disaggregation [47, 48], compute reconfiguration [38, 80], etc. We complement these works by exploring a knob for dynamic provisioning of memory and storage resources within a server using polymorphic emerging memory technologies.

Employing Emerging Memory Technologies: There is a rich body of work on readily exploiting emerging memory technologies [4, 5, 9, 11, 12, 24, 30, 41, 45, 51, 60, 61, 68] as DRAM replacement [22, 43, 45] or extension [22, 53], storage caches [16, 50] and flash/storage replacement [39]. This includes systems and mechanisms to exploit heterogeneous latency when employing these technologies as DRAM extension [27, 40] or storage caches [42, 59]. Further, there is a large body of research on using byte-addressable persistent programming techniques at the file system [19, 26, 33, 44, 46, 75, 76] and application levels [13, 18, 25, 28, 35, 70, 74, 78] to achieve performance close to that of the raw hardware. Complementary to such works, we exploit the polymorphic nature of these technologies at system software level without changes to file systems or applications to get closer to raw hardware performance than existing transparent approaches.
Exploiting Polymorphism: Prior works [39, 49, 62, 67] have studied EMT polymorphism. Morphable memory [62] proposes new hardware to trade capacity for latency in PCM, by exploiting single level (SLC) and multi level (MLC) representations. Memorage [39] and SAY-Go [67] propose memory and file system changes to increase the effective volatile memory capacity by using the free space from an NVM based file system volume. In these approaches, the capacity of NVM available for volatile memory decreases over time as the file system is used, and is also determined only by memory pressure in a two-tier architecture. In PolyEMT, we focus on a three-tier architecture where NVM is used as a cache to SSDs running on unmodified file systems and hardware.

7 CONCLUSION

Low tail latency of reads and writes is critical for the predictability of storage-intensive applications. The performance and capacity gaps between DRAM and SSDs unfortunately lead to high tail latency for such applications. Emerging memory technologies can help bridge these gaps transparently. However, existing proposals have targeted solutions that either benefit read or write performance but not both since they exploit only one of the many capabilities of such technologies. PolyEMT on the other hand exploits several polymorphic capabilities of EMTs to alleviate both read and write bottlenecks. It dynamically adjusts the capacity of EMT within each capability layer to tailor to application needs for maximal performance. Our evaluation with a popular storage benchmark shows that PolyEMT can provide substantial improvements in throughput and tail latencies of both read and write operations. In the future, we want to include more forms of polymorphism like latency vs. retention.

8 ACKNOWLEDGMENTS

This work has been supported in part by NSF grants 1439021, 1526750 1763681, 1629129, 1629915, and 1714389.

REFERENCES

[1] [n. d.]. ArangoDB. https://www.arangodb.com/.

[2] [n. d.]. Helion LZRW Compression cores. https://www.heliontech.com/comp_lzrw.htm.

[3] [n. d.]. How to emulate Persistent Memory. https://pmem.io/2016/02/22/pm-emulation.html/.

[4] [n. d.]. HPE Persistent Memory. https://www.hpe.com/us/en/servers/persistent-memory.html.

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

[6] [n. d.]. Lightning Memory-Mapped Database Manager (LMDB). http://www.lmdb.tech/doc/.

[7] [n. d.]. MapDB. http://www.mapdb.org/.

[8] [n. d.]. MonetDB. https://www.monetdb.org/Home.

[9] [n. d.]. Netlist Expressvault PCIe (EV3) PCI Express (PCIe) Cache Data Protection. http://www.netlist.com/products/vault-memory-storage/expressvault-pcIe-ev3/default.aspx.

[10] [n. d.]. Redis. https://redis.io/.

[11] Bulent Abali, Hubertus Franke, Xiaowei Shen, Dan E Poff, and T Basil Smith. 2001. Performance of hardware compressed main memory. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on.* IEEE, 73–81.

[12] Alaa R Alameldeen and David A Wood. 2004. Frequent pattern compression: A significance-based compression scheme for L2 caches. (2004).

[13] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* ACM, 707–722.

[14] Jens Axboe. 2014. Fio-flexible IO tester. URLhttp://freecode.com/projects/fio.

[15] Mary Baker, Satoshi Asami, Etienne Deprit, John Ouseterhout, and Margo Seltzer. 1992. Non-volatile memory for fast, reliable file systems. In *ACM SIGPLAN Notices.* ACM, 10–22.

[16] Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson. 2013. Bankshot: Caching slow storage in fast non-volatile memory. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads.* ACM, 1.

[17] Adrian M Caulfield, Arup De, Joel Coburn, Todor I Mollow, Rajesh K Gupta, and Steven Swanson. 2010. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 385–395.

[18] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices* 46, 3 (2011), 105–118.

[19] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* ACM, 133–146.

[20] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing.* ACM, 143–154.

[21] Fernando J Corbato. 1968. *A paging experiment with the multics system.* Technical Report. Massachusetts Institute of Technology.

[22] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. 2009. PDRAM: A hybrid PRAM and DRAM main memory system. In *Proceedings of the 46th Annual Design Automation Conference.* ACM, 664–469.

[23] Xiangyu Dong and Yuan Xie. 2011. AdaMS: Adaptive MLC/SLC phase-change memory design for file storage. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference.* IEEE Press, 31–36.

[24] Fred Douglis. 1993. The Compression Cache: Using On-line Compression to Extend Physical Memory. Usenix Winter.

[25] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles.* ACM, 54–70.

[26] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems.* ACM, 15.

[27] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems.* ACM, 15.

[28] Ru Fang, Hui-I Hsiao, Bin He, C Mohan, and Yun Wang. 2011. High performance database logging using storage class memory. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering.* IEEE Computer Society, 1221–1231.

[29] Inigo Goiri, Kien Le, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. 2011. Intelligent placement of datacenters for internet services. In *2011 31st International Conference on Distributed Computing Systems.* IEEE, 131–142.

[30] Erik G Hallnor and Steven K Reinhardt. 2005. A unified compressed memory hierarchy. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on.* IEEE, 201–212.

[31] James Hamilton. [n. d.]. Overall data center costs. https://perspectives.mvdirona.com/2010/09/overall-data-center-costs/.

[32] Sarah Harris and David Harris. 2015. *Digital design and computer architecture: arm edition.* Morgan Kaufmann.

[33] Y Hu, Z Zhu, I Neal, Y Kwon, T Cheng, V Chidambaram, and E Witchel. 2018. TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions. In *USENIX Annual Technical Conference (ATC)* .

[34] Jian Huang, Anirudh Badam, Moinuddin K Qureshi, and Karsten Schwan. 2015. Unified address translation for memory-mapped SSDs with FlashMap. In *ACM SIGARCH Computer Architecture News.* ACM, 580–591.

[35] Jian Huang, Karsten Schwan, and Moinuddin K Qureshi. 2014. NVRAM-aware logging in transaction systems. *Proceedings of the VLDB Endowment* 8, 4 (2014), 389–400.

[36] Intel. 2017. Application latency comparison, Introduction to Programming with Persistent Memory from Intel. https://software.intel.com/en-us/articles/introduction-to-programming-with-persistent-memory-from-intel.

[37] Theodore Johnson and Dennis Shasha. 1994. 2Q: a low overhead high performance bu er management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases.* 439–450.

[38] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on.* IEEE, 1–12.

[39] Ju-Young Jung and Sangyeun Cho. 2013. Memorage: Emerging persistent ram based malleable main memory and storage architecture. In *Proceedings of the 27th international ACM conference on International conference on supercomputing.* ACM, 115–126.

[40] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. Heteroos: Os design for heterogeneous memory management in datacenter. In *ACM SIGARCH Computer Architecture News*. ACM, 521–534.

[41] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. 2017. Viyojit: Decoupling battery and DRAM capacities for battery-backed DRAM. In *ACM SIGARCH Computer Architecture News*. ACM, 613–626.

[42] Youngjae Kim, Aayush Gupta, Bhuvan Urgaonkar, Piotr Berman, and Anand Sivasubramaniam. 2011. HybridStore: A cost-efficient, high-performance storage system combining SSDs and HDDs. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*. IEEE, 227–236.

[43] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 256–267.

[44] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 460–477.

[45] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*. ACM, 2–13.

[46] Eunji Lee, Hyokyung Bahn, and Sam H Noh. 2013. Unioning of the buffer cache and journaling layers with non-volatile memory.. In *FAST*, Vol. 13.

[47] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *ACM SIGARCH Computer Architecture News*. ACM, 267–278.

[48] K Lim, P Ranganathan, J Chang, C Patel, T Mudge, and S Reinhardt. 2008. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In *International Symposium on Computer Architecture*.

[49] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. 2014. NVM Duet: Unified working memory and persistent store architecture. In *n Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

[50] Zhuo Liu, Bin Wang, Patrick Carpenter, Dong Li, Jeffrey S Vetter, and Weikuan Yu. 2012. PCM-based durable write cache for fast disk I/O. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*. IEEE, 451–458.

[51] Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail D Flouris, and Angelos Bilas. 2010. Using transparent compression to improve SSD-based I/O caches. In *Proceedings of the 5th European conference on Computer systems*. ACM, 1–14.

[52] EP MARKATOS. 1996. Implementation of a Reliable Remote Memory Pager. In *USENIX 1996 Annual Technical Conference*.

[53] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. 2012. Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management. *IEEE Computer Architecture Letters* 11, 2 (2012), 61–64.

[54] Microsoft Server and Cloud Platform Team . 2015. Microsoft Reinvents Datacenter Power Backup with New Open Compute Project Specification. https://blogs.technet.microsoft.com/hybridcloud/2015/03/10/microsoft-reinvents-datacenter-power-backup-with-new-open-compute-project-specification/.

[55] Timothy Prickett Morgan. 2015. NVDIMM price. https://www.nextplatform.com/2015/12/07/nvdimm-cant-wait-for-3d-xpoint-cant-rely-on-dram-alone/.

[56] Iyswarya Narayanan, Aman Kansal, and Anand Sivasubramaniam. 2017. Right-Sizing Geo-distributed Data Centers for Availability and Latency. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 230–240.

[57] Iyswarya Narayanan, Aman Kansal, Anand Sivasubramaniam, Bhuvan Urgaonkar, and Sriram Govindan. 2014. Towards a leaner geo-distributed cloud infrastructure. In *6th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 14)*.

[58] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. 2016. SSD failures in datacenters: What? when? and why?. In *Proceedings of the 9th ACM International on Systems and Storage Conference*. ACM, 7.

[59] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. 2012. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems.. In *FAST*, Vol. 12.

[60] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2012. Base-delta-immediate compression: practical data compression for on-chip caches. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 377–388.

[61] Gennady Pekhimnko, Vivek Seshadri, Yoonqu Kim, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2013. Linearly compressed pages: a low-complexity, low-latency main memory compression framework. In *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*. IEEE, 172–184.

[62] Moinuddin K Qureshi, Michele M Franceschini, Luis A Lastras-Montaño, and John P Karidis. 2010. Morphable memory system: A robust architecture for exploiting multi-level phase change memories. In *ACM SIGARCH Computer Architecture News*. ACM, 153–162.

[63] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[64] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*. ACM, 85–95.

[65] Drew Robb. 2018. SSD and Flash in Enterprise Storage Environments. http://www.enterprisestorageforum.com/storage-technology/ssd-in-enterprise-storage-environments.html.

[66] Kevin Simon. 2018. Project Natick: Microsoft's Self-sufficient Underwater Datacenters. *IndraStra Global* 4, 6 (2018), 4.

[67] Hyeonho Song and Sam H. Noh. 2018. Towards Transparent and Seamless Storage-As-You-Go with Persistent Memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage18/presentation/song

[68] Irina Chihaia Tuduce and Thomas Gross. 2005. Adaptive main memory compression. In *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, 29–29.

[69] Eric Van Hensbergen and Ming Zhao. 2006. Dynamic policy disk caching for storage networking. (2006).

[70] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H Campbell. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX conference on File and stroage technologies*. USENIX Association, 5–5.

[71] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating system support for improving data locality on CC-NUMA compute servers. In *ACM Sigplan Notices*. ACM, 279–289.

[72] Kashi Venkatesh Vishwanath, Albert Greenberg, and Daniel A Reed. 2009. Modular data centers: how to design them?. In *Proceedings of the 1st ACM workshop on Large-Scale system and application performance*. ACM, 3–10.
[73] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. 2014. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 14.
[74] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*. ACM, 91–104.
[75] Jian Xu and Steven Swanson. 2016. 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*. USENIX Association, 323–338.
[76] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 478–496.
[77] Dongliang Xue, Chao Li, Linpeng Huang, Chentao Wu, and Tianyou Li. 2018. Adaptive Memory Fusion: Towards Transparent, Agile Integration of Persistent Memory. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 324–335.
[78] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A reliable and highly-available non-volatile memory system. In *ACM SIGARCH Computer Architecture News*. ACM, 3–18.
[79] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.
[80] Yanqi Zhou, Henry Hoffmann, and David Wentzlaff. 2016. CASH: Supporting IaaS customers with a sub-core configurable architecture. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 682–694.
[81] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory* 24, 5 (1978), 530–536.