FlatFlash: Exploiting the Byte-Accessibility of SSDs within A Unified Memory-Storage Hierarchy

Ahmed Abulila
UIUC

Vikram Sharma Mailthody
UIUC

Zaid Qureshi
UIUC

Jian Huang*
UIUC

Nam Sung Kim
UIUC

Jinjun Xiong
IBM Research

Wen-mei Hwu
UIUC

Abstract

Using flash-based solid state drives (SSDs) as main memory has been proposed as a practical solution towards scaling memory capacity for data-intensive applications. However, almost all existing approaches rely on the paging mechanism to move data between SSDs and host DRAM. This inevitably incurs significant performance overhead and extra I/O traffic. Thanks to the byte-addressability supported by the PCIe interconnect and the internal memory in SSD controllers, it isfeasible to access SSDs in both byte and block granularity today. Exploiting the benefits of SSD's byte-accessibility in today's memory-storage hierarchy is, however, challenging as it lacks systems support and abstractions for programs.

In this paper, we present FlatFlash, an optimized unified memory-storage hierarchy, to efficiently use byte-addressable SSD as part of the main memory. We extend the virtual memory management to provide a unified memory interface so that programs can access data across SSD and DRAM in byte granularity seamlessly. We propose a lightweight, adaptive page promotion mechanism between SSD and DRAM to gain benefits from both the byte-addressable large SSD and fast DRAM concurrently and transparently, while avoiding unnecessary page movements. Furthermore, we propose an abstraction of byte-granular data persistence to exploit the persistence nature of SSDs, upon which we rethink the design primitives of crash consistency of several representative software systems that require data persistence, such as file systems and databases. Our evaluation with a variety of applications demonstrates that, compared to the current unified memory-storage systems, FlatFlash improves the performance for memory-intensive applications by up to 2.3×, reduces the tail latency for latency-critical applications by up to 2.8×, scales the throughput for transactional database by up to 3.0×, and decreases the meta-data persistence overhead for file systems by up to 18.9×. FlatFlash also improves the cost-effectiveness by up to 3.8× compared to DRAM-only systems, while enhancing the SSD lifetime significantly.

*The corresponding author of this work is Jian Huang (jianh@illinois.edu).

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304061

CCS Concepts • **Hardware** → **Memory and dense storage**; *Non-volatile memory*; • **Software and its engineering** → **Memory management**.

Keywords byte-addressable SSD; unified memory management; page promotion; data persistence

ACM Reference Format:
Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. 2019. FlatFlash: Exploiting the Byte-Accessibility of SSDs within A Unified Memory-Storage Hierarchy. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304061

1 Introduction

Using SSDs as main memory [5, 20, 27, 60] has been presented as a practical approach to expanding the memory capacity for data-intensive applications, as the cost of SSDs is fast approaching that of hard disk drives, and their performance has improved by more than 1,000× over the past few years [23, 29, 59]. Also, SSDs today can scale up to terabytes per PCIe slot [44], whereas DRAM scales only to gigabytes per DIMM slot [26].

To overcome the DRAM scaling issue, the state-of-the-art approaches leverage the memory mapped interface and paging mechanism in operating systems and treat SSDs as fast backing storage for the DRAM [20, 27, 60]. Although these approaches simplify the development, they suffer from three drawbacks. First, the paging mechanism incurs performance overhead. For each memory access to data that is not present in DRAM, a page fault is triggered and a software page handler moves the accessed page from SSD to main memory, resulting in an execution delay. Second, the paging mechanism faces the thrashing problem caused by data-intensive applications whose working set sizes are significantly larger than the available DRAM capacity. Third, the page granularity of data access incurs a tremendous amount of extra I/O traffic by moving the whole page even if a small portion within that page is needed [47], which not only affects SSD performance but also hurts the SSD lifetime.

Thanks to the byte-addressability provided by the PCIe interconnect and the internal memory in SSD controllers, it is feasible today to have a byte-addressable SSD by leveraging the available Base Address Registers (BARs) in the SSD controller through the PCIe memory-mapped I/O interface (MMIO) [6]. Henceforth, we can access SSDs in both byte and block granularity. However, this inevitably increases the complexity of managing SSDs in modern memory-storage hierarchy and it is still unclear how systems software and applications will benefit from this new property.

In this paper, we exploit the byte-accessibility of SSDs and rethink the design of a unified memory-storage hierarchy to address the aforementioned challenges. We first map the SSD page locations into the host address space and extend virtual memory management to make use of the physical pages across the byte-addressable SSD and host DRAM. Since NAND flash chips are not natively byte-addressable and can only be accessed in page granularity, we utilize the DRAM present in the SSD controller and use it as a cache for accessed pages. Thus, the host CPU can issue memory requests (*load/store*) to a unified memory space, which eases the programmability and management of byte-addressable SSDs and makes programs access data across SSD and host DRAM seamlessly.

Accessing the SSD for each memory request is, however, slower than accessing DRAM. We propose a lightweight, adaptive page promotion mechanism to determine which flash pages should be placed in the host DRAM, therefore, applications can transparently exploit the advantages of both the byte-addressable SSD and the faster DRAM. We support promoting multiple hot pages concurrently to the host DRAM for fast access while keeping cold pages in the SSD for direct, byte-granular access to avoid thrashing. To avoid program stalls caused by the page promotion, we propose a promotion look-aside buffer (PLB) in the host bridge for redirecting memory requests for a page being promoted to its current physical address. As PLB is only used for pages being promoted, its storage overhead is trivial.

Furthermore, to preserve the persistent nature of SSDs in the unified memory-storage hierarchy, we propose byte-granular data persistence to exploit the fine-grained durable write with a battery-backed DRAM in SSDs. Unlike conventional persistent storage that uses a block interface, such a new persistence feature of SSDs significantly reduces the crash consistency overhead for software systems that have strict requirement on data persistence. With case studies of file systems and transactional databases, we demonstrate the benefits of this new feature and its impact on the design primitives of storage systems. Overall, this paper makes the following contributions:

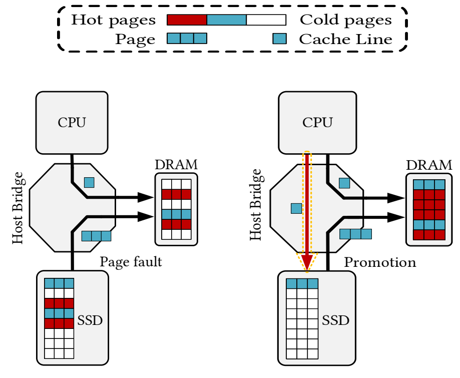
- We present FlatFlash, a unified memory and storage architecture with byte-addressable SSDs and DRAM. It presents a unified memory interface to simplify the management and programmability of the dual byte and block-accessible interfaces of SSDs.
- We propose an adaptive page promotion scheme that enables applications to benefit from both byte-addressable SSDs and DRAM simultaneously while incurring negligible performance and storage overhead.
- We exploit the byte-granular data persistence in FlatFlash, rethink the design primitives of ensuring crash consistency in several representative systems such as file systems and database, and demonstrate its performance benefits.

We implement FlatFlash in Linux on top of an SSD emulator with the new abstractions proposed. Compared to the state-of-the-art unified memory-storage solution, FlatFlash improves the performance of memory-intensive workloads such as the High-Performance Computing Challenge benchmark HPCC-GUPS [43] and graph analytics framework GraphChi [41] by up to 2.3×, reduces the tail latency for the NoSQL key-value store Redis [58] by up to 2.8×, scales the throughput of transactional database Shore-MT [34] by up to 3.0×, decreases the meta-data persistence overhead of file systems EXT4, XFS, and BtrFS by up to 18.9×. Beyond the performance and persistence benefits gained from FlatFlash, our evaluation also shows FlatFlash improves the cost-effectiveness by up to 3.8× compared to the DRAM-only system, while enhancing the SSD lifetime significantly.

The rest of the paper is organized as follows: § 2 explains the background and motivation of this work. The FlatFlash design and implementation are detailed in § 3 and § 4 respectively. Our evaluation methodology and results are presented in § 5. We discuss the related work in § 6 and conclude the paper in § 7.

2 Background and Motivation

To meet the ever-increasing demand for memory capacity from data-intensive applications, system designers either use a large amount of DRAM [42, 51] or leverage fast storage medium such as SSDs as backup store to scale up the application-usable memory capacity of systems [5, 20, 60]. Scaling up with DRAM is expensive, for example, a capacity of 512GB will cost about $8,950 with eight 64GB DDR4 DIMMs in 2018 [26]. And it is limited by the number of DIMM slots available on the servers [16, 66]. SSDs, on the other hand, scale to terabytes in capacity per PCIe slot [44] and are significantly cheaper than DRAM (e.g., a 1TB SSD



(a) Existing Approach **(b)** FlatFlash

Figure 1. (a) The state-of-the-art approach relies on paging mechanism to migrate pages; (b) FlatFlash provides direct memory accesses to the SSD with adaptive page promotion.

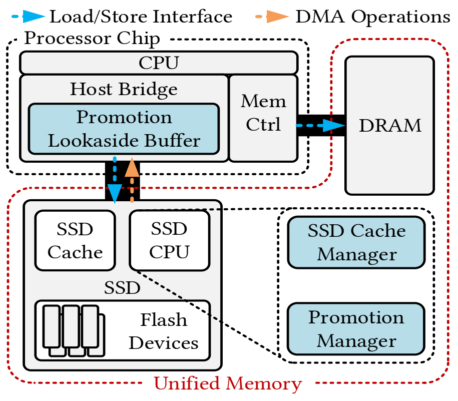


Figure 2. System architecture of FlatFlash.

costs around $600 in 2018 [49]). As SSD's capacity increases and cost decreases, using SSDs to expand memory capacity has become a cost-effective solution in practice [20, 27, 68]. System designers deploy SSDs as fast backing stores for memory mapped data and swap spaces for large datasets in virtual memory while leveraging the paging mechanism to move data between the DRAM and the SSD.

2.1 Using SSDs as Extended Memory

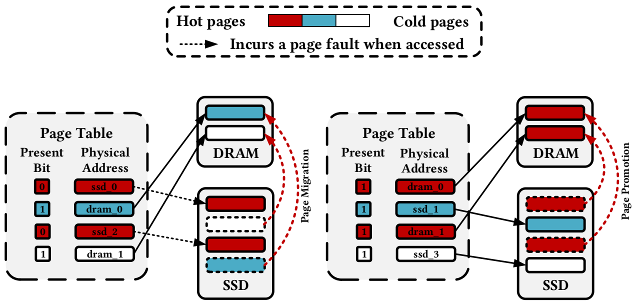
The state-of-the-art system using an SSD as extended memory (i.e., unified memory-storage hierarchy) relies on paging mechanism to manage SSDs, as shown in Figure 1a. Applications running on the CPU are allowed to access the DRAM in cache line granularity. However, a page fault occurs whenever an application accesses data in the extended memory region backed by SSD. The application is stalled until the OS's page fault handler migrates the requested page from the SSD to DRAM and updates the page table. For applications with large datasets that cannot fit in the DRAM, a significant number of page faults are triggered and pages are frequently swapped between DRAM and SSD [27]. For workloads with random page access patterns to a large data set, such as the HPCC-GUPS [43], the traffic between the SSD and DRAM is increased drastically due to the thrashing, a phenomenon where pages brought into DRAM are simply replaced before they can be accessed again.

2.2 Byte vs. Block-Accessible Interface for SSDs

The cost of page migration is exacerbated for workloads accessing only a few bytes or cache lines of a page, as any access to a page in SSD requires migration of the whole page between SSD and DRAM in the current memory-storage hierarchy. Being able to issue memory requests (i.e., *load/store*) directly to the SSD eliminates the need to migrate pages between the SSD and the DRAM. With the existing interface standards like PCIe (or NVMe) [48, 53], CCIX [12], QPI [4], and OpenCAPI [50], CPUs are capable of issuing *load/store* accesses, including atomic operations, directly to the SSD using memory-mapped I/O. However, NAND Flash chips, the memory technology commonly used in SSDs, are not byte-addressable. We can leverage the DRAM (a few GBs) present inside SSDs, originally for the purpose of storing the flash translation layer (FTL) and data buffering, to service the CPU's memory requests [6, 49, 59]. With the byte-addressable SSD memory extension (see Figure 1b), applications with a random access pattern can issue memory requests directly to the SSD in cache line granularity, thus reducing the I/O traffic. For an application that exhibits data locality, we can still promote its working set into fast DRAM for better performance. We will discuss how FlatFlash manages and exploits the byte-addressable SSD in detail in § 3.

3 FlatFlash Design

An overview of the FlatFlash system architecture is shown in Figure 2. FlatFlash addresses the challenges associated with exploiting the byte-accessibility of SSDs in five steps. First, we discuss the techniques used to enable byte-accessibility of SSDs in FlatFlash (§ 3.1). Second, we combine the SSD and DRAM into a unified and flat memory space. That is, a virtual memory page can be mapped either to the DRAM or the SSD. Such a unified memory interface simplifies the programmability of byte-addressable SSDs. Applications can directly access the SSD using regular memory requests without the need of paging mechanism (§ 3.2). Third, to gain benefits from the faster accesses to the DRAM, we propose a mechanism for promoting pages from the SSD to host DRAM. The mechanism keeps the page promotion activities from stalling application execution and ensures data consistency during the promotion process (§ 3.3). Fourth, to further bring benefits for applications, we develop an adaptive promotion policy that is dependent on their access patterns. The adaptive promotion mechanism is an integral part of Promotion Manager and interacts with the SSD-Cache to determine which pages to promote (§ 3.4). Fifth, FlatFlash enables byte-granular data persistence which facilitates critical-data persistence for systems software and applications that have strict requirement on data persistence (§ 3.5).



(a) Memory-Mapped SSD **(b)** FlatFlash

Figure 3. Page table support for memory-mapped SSD vs. FlatFlash. Both of them support unified address translation, however, FlatFlash enables direct memory access without relying on paging.

3.1 Enabling Byte-Addressability of SSDs

The PCIe standard defines a set of Base Address Registers (BARs) for end-point devices like SSDs to advertise the memory mappable region to the host at the system reset stage. During boot time, the BIOS and OS check the BAR registers of the PCIe-based end-point devices to add the extended memory mapped regions to the host. All memory requests to these extended memory-mapped regions are redirected to the respective PCIe end-point devices by the host bridge in Figure 2. The end-point device is responsible for mapping internal resources to the respective address ranges. With PCIe MMIO, the memory requests including atomic reads and writes, can be directly issued to the PCIe end-point device. FlatFlash exploits one of the PCIe BARs to expose the SSD as a byte-addressable memory mapped region to the host. During PCIe enumeration, the flash memory region is mapped into the memory space in the host.

Since the host PCIe bridge does not support cache coherence between the host machine and PCIe devices yet, the PCIe MMIO accesses are not cached in the host processor cache, which would miss the chance of exploiting the performance benefit of processor cache for applications. This is not much of a concern as commodity PCIe-based devices are increasingly employing the cache coherent protocol such as CAPI/OpenCAPI [25], CCIX [12], and GenZ [65] to accelerate applications. We leverage the cache coherent protocol in CAPI to enable cache-able memory accesses in FlatFlash.

Although PCIe MMIO supports memory requests in byte granularity, NAND flash memory chips have to be accessed in page granularity. To fill this gap, we leverage the DRAM (normally used for FTL but no longer needed since the FTL has been merged with the page table in the host, see § 3.2) inside the SSD as a cache (SSD-Cache in Figure 2) for the memory-mapped flash memory region. Therefore, SSD-Cache provides the necessary bridge between the byte-addressable interface and NAND Flash chips. The SSD-Cache Manager is responsible for handling all the operations related to SSD-Cache. Note that we organize SSD-Cache in page granularity. We will discuss the SSD-Cache details in § 3.4.

3.2 Unified Memory with Byte-Addressable SSD

SSDs can be used as memory via the memory-mapped interface provided by operating systems. In the traditional system, the page table entries point to the physical DRAM addresses in the host. For an access to the page in SSD, a page fault will occur, resulting in the page migration from the SSD to DRAM. Recently, Huang et al. [27] proposed the unified address translation for memory-mapped SSDs. It combines the traditional system's memory, storage, and device-level address translation (i.e., flash translation layer) into page tables in the virtual memory system. Therefore, the page table entries can point to the physical addresses in SSD as shown in Figure 3a. However, it still relies on paging mechanism to use DRAM as the cache when applications access the memory-mapped SSD.

FlatFlash differs from these existing work that require a page to be moved to the host DRAM before it can be accessed through the memory interface, it provides direct cache line access to the SSD as shown in Figure 3b. FlatFlash leverages the unified address translation mechanism in [27] to reduce the address translation overhead, and it further enables applications to issue memory requests directly to the pages in the SSD. This removes the need for the paging mechanism for a regular memory access to the memory-mapped SSD. FlatFlash handles memory requests to the SSD in the following ways.

- *memory read request:* The SSD-Cache manager in the SSD controller serves the memory *read* request by searching the SSD-Cache with the given physical address. For an SSD-Cache hit, it issues a PCIe MMIO response to the host with the data. If it is a cache miss, it reads the page from the flash device with the physical address. The SSD-Cache manager then issues a PCIe MMIO response with the requested cache line from the page.
- *memory write request:* For memory *write* request, if it is a SSD-Cache hit, SSD-Cache manager updates the page in the SSD-Cache with the new data. If it is a cache miss, it loads the page from the flash device into SSD-Cache and updates it with the new data.

FlatFlash relies on the garbage collection (GC) of the SSD to collect dirty pages in SSD-Cache and write them back to the SSD periodically. The GC is discussed in details in § 4.

Since accesses to SSD are slower than accesses to DRAM, pages that are frequently accessed (i.e., hot pages) can be promoted to the host DRAM for better performance. However, promoting a page from the SSD to the host DRAM is not free (it takes 12.1 μs as shown in Table 2). Also, a write request to the page that is being promoted would result in inconsistent view of the data to the application. To overcome these challenges, FlatFlash performs off-critical path page promotion to avoid application stall.

3.3 Off-Critical Path Page Promotion

To facilitate effective page promotion and ensure data consistency during the promotion procedure, we add a Promotion Look-aside Buffer (PLB, see Figure 2) to the host bridge (a.k.a., Root Complex) that connects the CPU, memory controller, and I/O interfaces. Note that the page promotion is executed in cache line granularity and concurrent promotions for multiple cache lines are enabled.

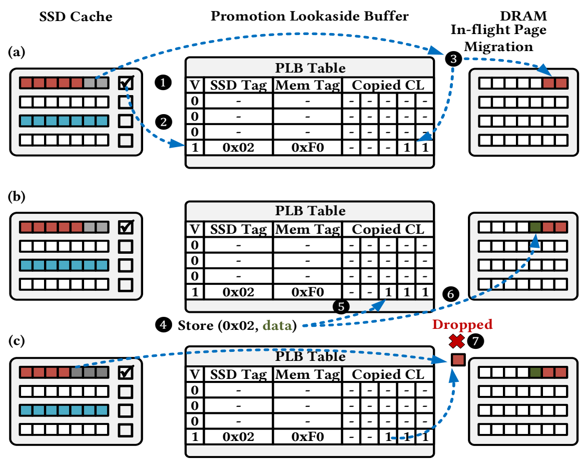


Figure 4. An example for the Promotion Look-Aside Buffer.

The PLB consists of a PLB table and a controller to manage any memory requests to the in-flight promotion page. As shown in Figure 4a, each in-flight page promotion has an entry in the PLB table with its source SSD physical address (`SSD tag`) and its destination DRAM physical address (`Mem tag`). The PLB entry also has a valid bit (`V`) and a bit vector (`Copied Cache Line`) to indicate if each cache line (CL) in the page is currently residing in the host DRAM or not. When a page promotion is requested by the promotion manager, the PLB will get a free page from a reserved memory region in host DRAM for caching the page being promoted and initialize a PLB entry with the SSD and DRAM addresses.

PLB uses the `Copied CL` field to ensure the data consistency between the SSD and host DRAM. During the page promotion, for each cache line that is promoted from the SSD to the host, the PLB controller checks if the corresponding bit in `Copied CL` field is set or not. If it has been set by an evicted cache line from the CPU during the promotion, PLB controller will cancel the promotion of that cache line from the SSD. Otherwise, the PLB controller first sets the corresponding bit in the `Copied CL` field to inform that the most recent copy of that cache line exists in the host DRAM and then all the memory operations to this cache line will be forwarded to the host DRAM. When PLB receives requests to the same cache line from both the host CPU (for regular memory operations) and the SSD controller (for page promotion), it gives higher priority to the requests from host CPU to avoid conflicts.

Once the promotion of a page is completed, its corresponding entry in PLB is cleaned, and the corresponding page table entry (PTE) and translation lookaside buffer (TLB) entry are updated. As the latency of accessing data in the SSD is much higher than the latency of TLB shootdown, the overhead of TLB shootdown is small and has negligible impact on the performance of the whole system.

In FlatFlash, the PLB table has 64 entries. Each entry has 24 bytes (8 bytes for each tag) and 1 valid bit. Thus, its storage overhead is trivial. Once the promotion of a page is completed, its corresponding entry in the PLB table will be cleaned for future use. As we index the PLB entries following the principles of the content-addressable memory (CAM) design, each PLB entry lookup takes only one CPU cycle [52], thus its performance overhead is negligible. Since PLB has multiple entries, it enables concurrent promotion of multiple hot flash pages.

As host DRAM capacity is limited, the least-recently used pages will be evicted out for free space in host DRAM and written back to SSD with page granularity, and the corresponding TLB entries will be updated to the flash addresses. Since the latency of writing a page to SSD is much higher (16 μs for ultra-low latency Z-SSD [67]) than the latency of TLB shootdown [3], the TLB shootdown overhead is relatively small and has negligible effect.

We use an example to demonstrate the in-flight page promotion process. As shown in Figure 4, a hot page (❶) is identified for promotion. The SSD promotion manager initiates the promotion by informing the PLB controller with the source and destination physical addresses. The PLB controller then inserts a valid entry to the PLB table (❷) as shown in 4a. For each CL that is promoted, the PLB controller sets the corresponding bit in the `Copied CL` field (❸). Figure 4b illustrates the case for a store (cache eviction) to a CL within the in-flight page promotion (❹). PLB controller set the `Copied CL` field (❺) in the PLB entry and redirects the CL to the host DRAM using the `Mem Tag` field (❻). As for the promotion of an inbound CL from the SSD promotion manager, the PLB controller checks the `Copied CL` flag to determine if the CL is updated or not. As shown in Figure 4c, if the `Copied CL` is set, it means the most recent copy of the corresponding CL is in the host DRAM, the inbound CL from the SSD will be discarded (❼).

3.4 Adaptive Page Promotion Scheme

We now discuss how we identify pages for promotion within SSD-Cache. SSD-Cache in FlatFlash uses a set-associative cache structure and leverages Re-reference Interval Prediction (RRIP) as its replacement policy since it can achieve a better cache hit rate [31], especially for random page accesses. Each entry in SSD-Cache has a valid bit (`V`), tag (`Tag`), re-reference interval value counter (`RRPV`), page hit counter (`pageCnt`), and page data (`Page`). The `pageCnt` increments when a cache line of a corresponding page is accessed.

```
Variables: PageCntArray({0}), NetAggCnt(0), AccessCnt(0), AggPromotedCnt(0), LwRatio(0.25), HiRatio(0.75), MaxThreshold(7), ResetEpoch(10K), CurrThreshold(7)
 1: procedure ADJUST_CNT($set$,$way$)
 2:     NetAggCnt ← NetAggCnt - PageCntArray[$set$][$way$]
 3:     PageCntArray[$set$][$way$] ← 0
 4: procedure UPDATE($pageSet$,$pageWay$)
 5:     NetAggCnt++
 6:     AccessCnt++
 7:     pageCnt ← ++PageCntArray[$pageSet$][$pageWay$]
 8:     promoteFlag ← pageCnt = CurrThreshold
 9:     if promoteFlag then
10:         AggPromotedCnt ← AggPromotedCnt + pageCnt
11:         PROMOTE($pageSet$, $pageWay$)
12:     currRatio ← AggPromotedCnt / AccessCnt
13:     if currRatio ≤ LwRatio then
14:         if CurrThreshold < MaxThreshold then
15:             CurrThreshold++
16:     else if currRatio ≥ HiRatio then
17:         if CurrThreshold > 1 and promoteFlag then
18:             CurrThreshold--
19:     if AccessCnt = ResetEpoch then                ▷ reset counters
20:         AccessCnt ← NetAggCnt
21:         AggPromotedCnt ← 0
22:         CurrThreshold ← maxThreshold
```

Algorithm 1. The adaptive page promotion algorithm. The variables are listed with their initial values. `UPDATE` procedure is called on every memory access to the SSD and `ADJUST_CNT` is invoked on a page eviction in SSD-Cache.

A naive approach, taken from the paging mechanism, promotes every accessed page. This can pollute the DRAM as many of these pages have low reuse. To manage page promotions, an access counter can be added for each page in the SSD-Cache, `pageCnt`, whose value is compared against a threshold, `maxThreshold`, on every access to determine if it should be promoted or not. However, comparison against a fixed threshold is insufficient to dynamically adapt to different memory access patterns. Thus an adaptive threshold, `CurrThreshold`, is needed so that pages are promoted frequently when there is high page-reuse and infrequently when there is low page-reuse.

In order to detect page re-use pattern, we set `currRatio` to $\frac{\texttt{AggPromotedCnt}}{\texttt{AccessCnt}}$, where `AggPromotedCnt` is the sum of the page access counters that have reached `CurrThreshold` and `AccessCnt` is the total number of accesses to the SSD-Cache. A high value of `currRatio` signifies high page-reuse as many pages' access counters reached the `CurrThreshold` value and were promoted. Similarly, a low value of `currRatio` signifies low page-reuse as not many pages' access counters reached the `CurrThreshold`. The adaptive promotion algorithm adapts the `CurrThreshold` value based on whether the `currRatio` is high or low. If `currRatio` is greater than or equal to `HiRatio`, then `CurrThreshold` is decremented so that the pages are promoted frequently. If `currRatio` is less than or equal to `LwRatio`, then `CurrThreshold` is incremented so that the pages are promoted infrequently.

To mitigate the slow unlearning rate of the adaptive promotion algorithm, we reset the counters of `CurrThreshold`, `AggPromotedCnt`, and `AccessCnt` at every epoch, `ResetEpoch`. To preserve the access pattern for the pages currently in the SSD-Cache, we set `AccessCnt` to `NetAggCnt`, the sum of the `pageCnt` for all pages present in the SSD-Cache, avoiding the overhead of scanning the `PageCntArray` which has 512K entries for a 2GB SSD-Cache. The storage overhead of the page promotion mechanism is 0.2% of the SSD-Cache size, mostly contributed by the `PageCntArray`. We show the adaptive promotion scheme in Algorithm 1.

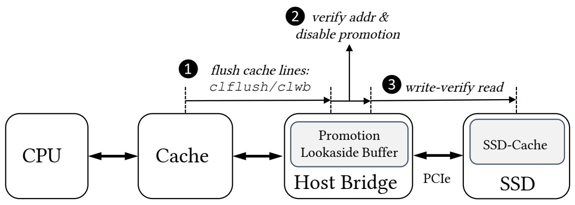
3.5 Byte-Granular Data Persistence

As discussed, the unified memory interface simplifies the programmability with byte-addressable SSDs, and programs can transparently obtain the performance benefits from both byte-addressable large SSDs and fast DRAM. As a new interface enabled in SSDs, the byte-accessibility also helps programs achieve fine-grained data persistence by exploiting the byte-granular durable write in combination with the persistence nature of SSDs.

Unlike conventional persistent storage that uses a block interface, FlatFlash enables byte-granular data persistence, which significantly reduces the I/O traffic to SSDs. This feature can be used to achieve fine-grained data persistence for specific data structures in software, such as log persistence in database and meta-data persistence in file systems.

To implement this feature, FlatFlash leverages the existing PCIe-based atomic memory operations (see § 3.1). As many modern SSDs employ battery-backed DRAM or large capacitors [2, 6, 57] in their controllers, the received memory write requests via PCIe MMIO will be persistent without much hardware modifications. FlatFlash employs battery-backed DRAM inside the SSD to simplify its implementation for data persistence. It allows applications to create a dedicated persistent memory region with the provided function: *create_pmem_region (void* vaddr, size_t size)*. All of the virtual addresses in the persistent memory region are mapped to the address space of the SSD.

However, ensuring the data persistence is challenging, because (1) the update to the persistent memory region could be cached in processor cache and (2) a page in SSD could be promoted to the volatile DRAM in the host. To overcome the first challenge, FlatFlash enforces applications to flush the corresponding cache lines when they write to the persistent memory region and employs the "write-verify read" approach [6, 11] which functions similarly to *mfence* to enforce the ordering of writes and cache flushing in host bridge. To overcome the second challenge, FlatFlash leverages one of the reserved bits in the PTE as the `Persist (P)` bit to indicate whether a page should be promoted or not. For every memory access to the SSD, during address translation, the physical address is prefixed with the `P` bit, and this new physical address is transferred to the host bridge. When the



2 verify addr & disable promotion
1 flush cache lines: clflush/clwb
3 write-verify read
CPU
Cache
Promotion Lookaside Buffer
Host Bridge
PCIe
SSD-Cache
SSD

Figure 5. The workflow of enforcing byte-granular data persistence.

host bridge detects a memory request for the SSD, it creates a PCIe packet. In this packet, the Address field is set to the memory address with the P bit masked out, and the Attribute field is set to the value of the P bit [53]. When the SSD receives a packet with the P bit set, it will not execute UPDATE in Algorithm 1 to avoid the promotion of these pages. The entire workflow is presented in Figure 5.

For decades, the upper-level storage software and applications were built based on the block I/O interface. We are motivated to rethink their design primitives with the byte-granular data persistence enabled in SSDs. To facilitate our discussion, we use two representative systems as examples.

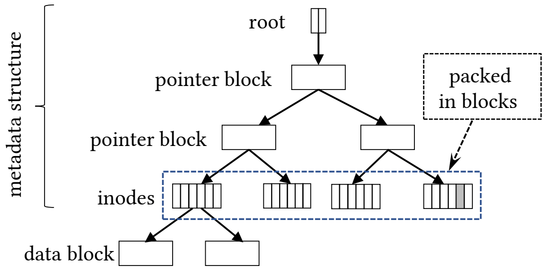


Figure 6. The metadata structure of a file system.

Improving Metadata Persistency in File Systems. In almost any type of file system, maintaining the metadata (e.g., *inode*) persistence is critical for ensuring data consistency. Although metadata operations incur only small updates (usually 8–256 bytes in current file systems), they are on the critical path, which significantly affects the storage performance. A small metadata update inevitably causes a page update, resulting in large write amplification as shown in Figure 6. For example, file creation requires the allocation of a new *inode* and a update to the parent directory, it generates 16-116 KB write IO and 24-40 KB of read IO in different file systems [47]. With FlatFlash, we can allocate persistent memory regions for the critical data structures like *inode* and metadata journal, and persist their updates with byte-granular data persistence.

Decentralizing the Logging for Databases. Transactional databases require logging to preserve the ACID properties for transactions. In the traditional storage system stack, databases usually have a centralized write-ahead log to group logs (64-1,424 bytes per transaction according to our study on database workloads TPCC, TPCB, and TATP) for avoiding frequent I/O operations and generating sequential data

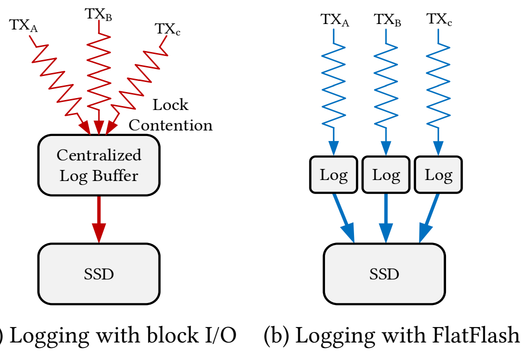


Figure 7. Per-transaction logging with FlatFlash.

access pattern [46]. Such a design causes serious logging contention in multi-core era [69] which limits the scalability of transaction operations, especially when the persistent storage device like SSD becomes faster. With FlatFlash, we can decentralize the log buffer and apply the per-transaction logging (similar to the logging approach proposed in [28]) to issue multiple atomic and persistent log *writes* concurrently (see Figure 7). Therefore, the logging bottleneck can be reduced for better scalability. We integrate the discussed approaches in both examples and evaluate them in § 5.

4 FlatFlash Implementation

FlatFlash Memory Management: We build a unified address translation layer in the memory manager using a similar approach as described in [27], where all of the indirection layers of the memory mapping of an SSD have been combined into a single layer. FlatFlash uses the memory-mapped interface to create a unified memory address space while providing direct access to any data that is mapped to the SSD in cache line granularity. Since the FTL of the SSD is integrated into the page table in the virtual memory system in the host, FlatFlash allows the SSD controller to update the address mapping in PTEs and TLB entries for memory-mapped regions when garbage collection (GC) of the SSD moves pages to new flash blocks. To avoid frequent TLB shootdown, FlatFlash maintains a mapping table in SSD that maps the old physical address to the new one. Thus, it can serve memory requests from the host machine using the old physical address, and the entries in the mapping table will be lazily propagated to the page table entries and TLB entries in batches using a single interrupt [3, 8, 39].

Byte-Addressable SSD Emulation: To emulate the byte-addressable SSD, we convert a real SSD to a byte-addressable SSD and emulate the promotion look-aside buffer in the host bridge (the only hardware modification required by FlatFlash). The host memory is used to model the SSD-cache while the real SSD is utilized to model the raw flash device. The host memory has been divided into two different regions: the first region represents a regular main memory; the second region models the SSD-Cache. We use a red-black tree to index the pages in these regions separately. The SSD emulation has been implemented as part of the memory manager which keeps track of the pages cached in the SSD-Cache and the pages promoted to host DRAM, and fulfills the PLB functions and the page promotion algorithm as described in § 3. To reflect the memory access timings of the SSD regions, the memory manager leverages the protection bit in the page table entries for the SSD regions using `mprotect`. When the application accesses any page within the SSD regions, an exception is raised and the handler introduces additional latency and clears the protection bit. To emulate the SSD-cache, we flush the TLB entries of the recently accessed pages and reset the read protection bit in these PTE entries for the SSD region with a dedicated thread.

FlatFlash uses a similar read-modify-write garbage collection (GC) scheme as used in other SSDs [7, 24, 73], to collect dirty pages in SSD-Cache and write them back to the SSD periodically. In the read phase, the GC reads a flash block into memory. In the modify phase, it overwrites the invalid pages in the memory copy of the flash block with the dirty pages from the SSD-Cache. In the write phase, it writes the in-memory copy of the flash block into a free flash block and updates the page table entries and TLB entries lazily (as discussed previously) for these flushed dirty pages in SSD-Cache.

5 Evaluation

Our evaluation demonstrates that: (1) FlatFlash improves performance for data-intensive applications by adaptively using a byte-addressable SSD and the host DRAM according to workload characteristics (§ 5.1, § 5.2, § 5.3, and § 5.4); (2) It minimizes the persistence overhead with byte-granular data persistence and enables new optimization opportunities for software systems with persistence requirement (§ 5.5 and § 5.6); (3) It provides a cost-effective solution to physically extend memory capacity in a transparent way (§ 5.7). We compare FlatFlash against the following systems:

- **Separated Memory-Storage Stack (TraditionalStack):** The traditional memory and storage stack consisted of DRAM with byte-addressability and SSD with a block I/O interface. Systems software, such as the virtual memory manager and EXT4 file system, are used to manage DRAM and SSD, respectively. An unmodified *mmap* interface is used to map the storage into virtual memory and the paging mechanism is used by default to swap pages between DRAM and SSD. We consider *TraditionalStack* with placing the SSD FTL in the host DRAM for high performance, which is similar to ioMemory [20]. All indirections are separated but used on demand for performance.
- **Unified Memory-Mapped Storage (UnifiedMMap):** Similar to the previous work such as FlashMap [27] that combines all three indirection layers into a unified layer, it reduces the address translation overhead for memory-mapped storage and utilizes the DRAM resource by lowering the storage cost of maintaining metadata for each indirection layer. *UnifiedMMap* bypasses the conventional storage software stack to access data in SSDs.

Table 1. Real workloads used in our evaluation. We summarize FlatFlash's average improvements on both performance and SSD lifetime compared to UnifiedMMap.

Applications	Benchmarks	FlatFlash Improvement (Avg.)	
		Performance	SSD Lifetime
HPC Challenge (§ 5.2)	GUPS	1.6×	1.3×
Graph Analytics: GraphChi (§ 5.3)	PageRank	1.3×	1.5×
	Connected Component	1.5×	1.9×
Key-Value Store: Redis (§ 5.4)	YCSB-B	2.1×	1.3×
	YCSB-D	2.2×	1.3×
File Systems EXT4, XFS, BtrFS (§ 5.5)	CreateFile	3.6-11.2×	2.2-8.4×
	RenameFile	2.6-6.7×	1.5-12.1×
	CreateDirectory	3.6-15.3×	1.4-9.8×
	VarMail	3.2-6.2×	1.9-8.0×
	WebServer	5.3-18.9×	1.5-3.1×
Transactional Database: ShorMT (§ 5.6)	TPCC	1.9×	1.0×
	TPCB	2.8×	1.0×
	TATP	1.3×	1.0×

Table 2. Latency of the major components in FlatFlash

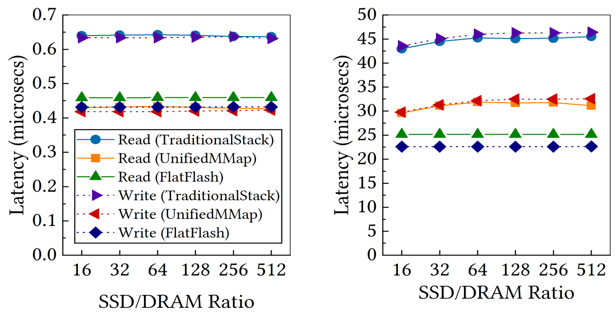
Overhead Source	Average (μsec)
Read a cache line in SSD-Cache via PCIe MMIO	4.8
Write a cache line in SSD-Cache via PCIe MMIO	0.6
Promote a page from SSD-Cache to host DRAM	12.1
Update PTE and TLB entry in host machine	1.4
Page table walking to get the page location	0.7

We use a server machine with a 24-core Intel Haswell based Xeon CPU running at 2.6 GHz with 64GB of DRAM and an Intel DC P3700 series PCIe-based SSD with the capacity of 1.6TB. We use the system described in § 4 to emulate FlatFlash. We set the size of the SSD-Cache to be 0.125% of the SSD capacity by default. To quantify the performance benefits of FlatFlash in different aspects, we use a variety of data-intensive applications that contain both high-performance computing and enterprise workloads. We summarize the experimental results in Table 1.

To measure the read/write latency of accessing a cache line via PCIe MMIO, we used a Xilinx Virtex-7 [70] and annotated the driver of a reference design. We used the measured numbers (see Table 2) in our byte-addressable SSD emulator. The MMIO write operation is a posted transaction which is completed when the written data reaches the write buffer [11]. Therefore, the latency of the write transaction is significantly lower than that of the read transaction.

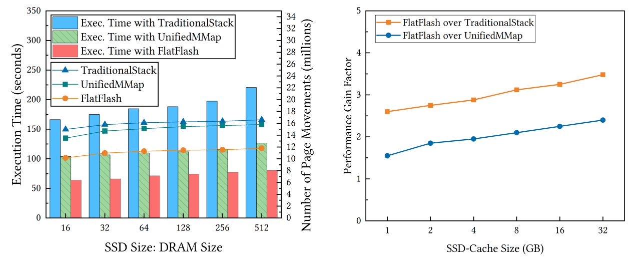
5.1 Byte-Addressable SSD vs. Paging Mechanism

We first evaluate the performance of FlatFlash with synthetic workloads. We vary the SSD size from 32GB to 1TB while keeping the host DRAM size at 2 GB for the workloads (without including the main memory required by host OS). We allocate 2 million pages (4 KB) that distribute uniformly from



(a) Sequential Access **(b)** Random Access

Figure 8. Average latencies of accessing a cache line (64 bytes) in sequential and random manner respectively.



(a) HPCC-GUPS Performance **(b)** Sensitivity to SSD-Cache Size

Figure 9. (a) FlatFlash performs 1.6× and 2.7× faster than *UnifiedMMap* and *TraditionalStack* for HPCC-GUPS. The lines represent the number of page movements between SSD and host DRAM. (b) FlatFlash benefits more from the increased SSD-Cache size.

a file that spans the entire SSD. We perform both sequential and random memory accesses against these pages in cache line granularity (64 Bytes). At the beginning, we randomly access the 2 million pages to warm up the system.

We compare the read/write latencies of FlatFlash with *TraditionalStack* and *UnifiedMMap* as described above and report the average latencies of accessing 64 bytes of data in Figure 8. For sequential memory access, the latency of accessing a cache line in FlatFlash is close to that of *UnifiedMMap* with a slight overhead as shown in Figure 8a. The additional overhead is introduced by the off-critical path page promotion (see § 3.3) which takes 12.1 μs on average for a 4KB page. Both FlatFlash and *UnifiedMMap* performs much better than *TraditionalStack* because they bypass the storage software stack with the unified address translation.

For random memory access, FlatFlash reduces the latencies by 1.2-1.4× and 1.8-2.1× compared to the *UnifiedMMap* and *TraditionalStack* respectively, because accessing a cache line of a page in SSD-Cache via PCIe MMIO is more efficient than moving a flash page with low reuse to host DRAM. FlatFlash will automatically switch between the PCIe MMIO mode and page promotion according to workload patterns.

In summary, for memory accesses with high page-reuse, FlatFlash promotes pages to host DRAM for better performance. For low page-reuse cases, FlatFlash outperforms others because it adaptively issues memory requests to the SSD over PCIe rather than using the conventional paging mechanism. We evaluate FlatFlash with real applications as follows.

5.2 Performance Benefit for HPC Applications

For high-performance computing applications, a representative benchmark is the High-Performance Computing Challenge (HPCC-GUPS) [43]. We use its *RandomAccess* benchmark in our experiment, as it is usually used to test the capability of memory systems. In this benchmark, multiple threads cooperate to solve a large scientific problem by updating random locations of a large in-memory table. We set the table size to be 32GB, which is larger than the host DRAM available for GUPS (2GB). Therefore, SSD will be used to expand the memory space when the benchmark runs.

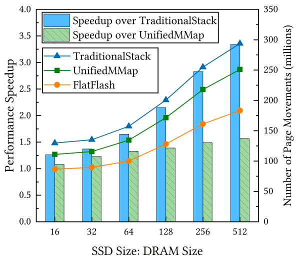
FlatFlash performs 1.5-1.6× and 2.5-2.7× faster than *UnifiedMMap* and *TraditionalStack* respectively as shown in Figure 9a. FlatFlash outperforms *UnifiedMMap* because many of the random memory accesses are issued to the SSD directly with PCIe MMIO. In this case, fewer pages are moved between the SSD and host DRAM. FlatFlash reduces page movement by 1.3-1.5× compared to the *UnifiedMMap* and *TraditionalStack* solutions (see the lines in Figure 9a). *UnifiedMMap* has slightly fewer page movements than that of *TraditionalStack* because it has more available DRAM for the application's working set by combining the address translation layers across the storage stack and thus reducing the size of the page index. For applications that demand a large amount of memory and have random memory accesses, FlatFlash provides a practical solution by leveraging the SSD to expand the memory capacity while exploiting its byte-accessibility for better performance.

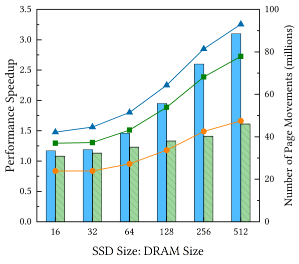
To understand how SSD-Cache size affects the FlatFlash performance, we vary the SSD-Cache size while keeping the same working set size for GUPS, and maintaining the SSD:DRAM ratio at 512. Figure 9b shows the performance speedup of FlatFlash over *UnifiedMMap* and *TraditionalStack* is increased as we increase the SSD-Cache size. This is because both *UnifiedMMap* and *TraditionalStack* have to migrate pages from the SSD to host DRAM irrespective of the SSD-Cache size. FlatFlash can utilize the SSD-Cache capacity and directly access the data in SSD without migrating pages.

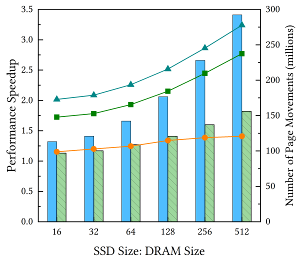
5.3 Performance Benefit for Graph Analytics

Beyond HPC workload, we also evaluate the performance benefit of FlatFlash for enterprise graph analytics applications. These graph analytics applications are typically used for large social networks and genomics analysis, which are memory intensive. The experiments with graph analytics aim to demonstrate that FlatFlash can also benefit memory-intensive applications that preserve certain levels of data locality (e.g., power-law distribution [21]).

We use GraphChi [41], which is a graph analytics framework that partitions large graphs such that each partition


(a) PageRank + Twitter


(b) PageRank + Friendster


(c) Conn-Component + Twitter

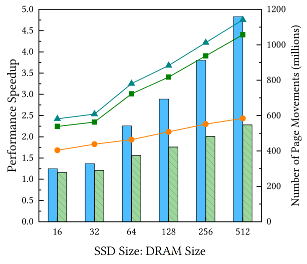

(d) Conn-Component + Friendster

Figure 10. Performance of graph analytics on Twitter and Friendster graph datasets with various DRAM size. Compared to *UnifiedMMap*, FlatFlash provides 1.1-1.6× better performance for PageRank and 1.1-2.3× better performance for Connected-Component Labeling algorithm. The lines represent the number of page movement between SSD and host DRAM.

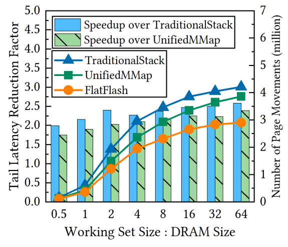
can fit in DRAM. We modify GraphChi to place the entire graphs in FlatFlash and run graph analytic algorithms with various DRAM sizes. The size of the graph data is beyond the available amount of DRAM, thus, the SSD is used as memory. We run two graph analytics algorithms, PageRank and Connected Component Labeling, on two different graphs: (1) Twitter graph dataset [40], which has 61.5 million vertices and 1.5 billion edges; (2) Friendster graph dataset [71], which has 65.6 million vertices and 1.8 billion edges.

We first run the PageRank algorithm on both Twitter and Friendster datasets. FlatFlash outperforms *UnifiedMMap* by 1.1-1.6× as shown in Figure 10. As we increase the SSD:DRAM ratio, the benefit brought by FlatFlash over *UnifiedMMap* is increased, because the thrashing of the limited DRAM is reduced since the host can directly issue memory requests to the SSD to avoid page movement between the SSD and host DRAM. As FlatFlash performs 1.2-3.3× better than *TraditionalStack*, as it leverages the unified address translation to improve DRAM efficiency.

We next evaluate the Connected-Component Labeling algorithm. FlatFlash performs 1.1-2.3× and 1.3-4.8× better than *UnifiedMMap* and *TraditionalStack*, respectively. For the Friendster graph dataset, a significant number of page movements are incurred due to misses in host DRAM, as shown in Figure 10d. FlatFlash improves such types of workload significantly as it enables the host CPU to access flash pages directly without paging them to host DRAM.

5.4 Latency Benefit for Key-Value Store

We now demonstrate the benefit of FlatFlash for latency-critical applications. We use the in-memory key-value store Redis [58] as a representative for such applications. We run Yahoo Cloud Serving Benchmark (YCSB) [15] workloads that represent the typical cloud services to test the latencies of accessing Redis. In our evaluation, we use workloads B and D. Workload B consists of 95% reads and 5% updates, modeling a photo tagging application. Workload D consists of 95% reads and 5% inserts, modeling social media status updates. Both workloads issue requests with Zipfian distribution.


(a) YCSB-B

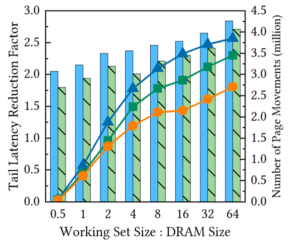
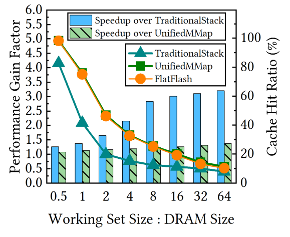

(b) YCSB-D

Figure 11. Tail latency reduction for key-value store Redis. FlatFlash reduces the tail latency by 1.8-2.7× and 2.0-2.8× compared to *TraditionalStack* and *UnifiedMMap*, respectively.


(a) YCSB-B

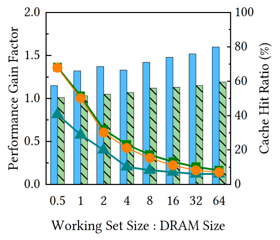

(b) YCSB-D

Figure 12. Average latency reduction for key-value store Redis with YCSB. FlatFlash reduces the average latency by 1.2-3.2× and 1.1-1.4× compared to *TraditionalStack* and *UnifiedMMap*, respectively. The lines represent cache hit ratio.

We use 16 client threads to issue key-value operations against Redis. Each key-value pair is 64 bytes. We conduct 64 million operations for each workload and adjust the working set sizes by setting the request distribution parameter in YCSB. We maintain the SSD:DRAM ratio at 256 while varying the ratio of working set size to DRAM size.

In these workloads, the tail latency is important as it dictates the performance guarantees for 99% of the requests (i.e., 99th percentile latency). FlatFlash reduces the 99th percentile latency by 2.2× and 2.5× on average compared to *UnifiedMMap* and *TraditionalStack*, respectively (see Figure 11). The benefit mainly comes from FlatFlash's promotion algorithm, which decreases page movement between the SSD and host

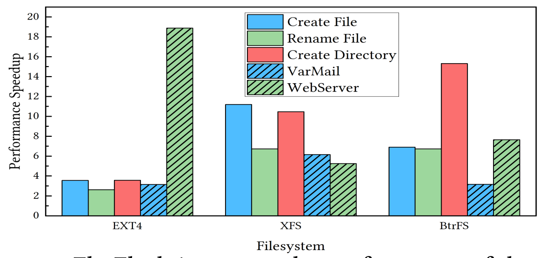


Figure 13. FlatFlash improves the performance of the common file system operations by up to 18.9×. *VarMail* benchmark emulates a mail server that stores each email in a file; *WebServer* emulates the storage operations in a web-server.

DRAM as it avoids promoting low-reuse pages. Taking the YCSB workload B for example, when its working set size is 16× larger than the available host DRAM size, the number of pages moved between SSD and host DRAM is reduced from 3.9 million in *TraditionalStack* to 2.7 million in FlatFlash as shown in Figure 11a. Such a policy can avoid pollution in the host DRAM and reduce the I/O traffic to the SSD, therefore, the performance interference is reduced. Similar results have been seen in YCSB workload D as shown in Figure 11b.

FlatFlash improves the average latency of Redis by up to 3.2× and 1.4× compared to *TraditionalStack* and *UnifiedMMap* respectively, as demonstrated in Figure 12. For YCSB workloads that have certain levels of data locality, FlatFlash further improves application performance by reducing the latency of the remaining random requests (see Figure 8) while exploiting the DRAM speed for requests with data locality.

5.5 Persistency Benefit for File Systems

In this section, we demonstrate the benefit of FlatFlash on data persistence by applying the byte-granular data persistence to file systems as the case study discussed in § 3.5. We modified EXT4 (e.g., *ext4_setattr* in *inode.c*), XFS (e.g., *xfs_setattr_size* in *xfs_iops.c*), and BtrFS (e.g., *btrfs_setattr* in *inode.c*) to instrument their metadata persistence procedures and collect the storage traces of metadata and data operations when running a variety of file system benchmarks from FileBench [64]. Each benchmark executes about 50 million file system operations. As shown in Figure 13, FlatFlash improves the performance of these common file system operations such as file creation, file rename, and directory creation by 2.6-18.9×, 5.3-11.2×, 3.2-15.3× for EXT4, XFS, and BtrFS, respectively. The benefits of FlatFlash mainly come from the byte-accessibility of SSDs. Since each file system has its own implementation for data consistency, the performance improvement varies for the same workload.

Instead of persisting a page for each metadata operation, FlatFlash guarantees the atomicity and durability of the small updates with PCIe operations and their persistency with a battery-backed DRAM cache inside SSD. Beyond performance benefits, FlatFlash also significantly reduces the write amplification for file system operations by avoiding the redundant journaling (for EXT4 and XFS) and copy-on-write

Table 3. Cost-effectiveness of FlatFlash vs. DRAM-only.

Application	Workloads	Slow-down	Cost-Saving	Cost-Effectiveness
HPC Challenge	GUPS	8.9×	14.6×	1.6×
Graph Analytics	PageRank	11.0×	14.6×	1.3×
	Conn-Component	6.9×	14.6×	2.1×
Key-Value Store	YCSB-B	6.1×	15.0×	2.5×
	YCSB-D	5.5×	15.0×	2.7×
Transactional Database	TPCC	1.4×	2.4×	1.7×
	TPCB	1.9×	2.6×	1.4×
	TATP	1.2×	4.5×	3.8×

logging (for BtrFS), which further improves SSD lifetime as shown in Table 1.

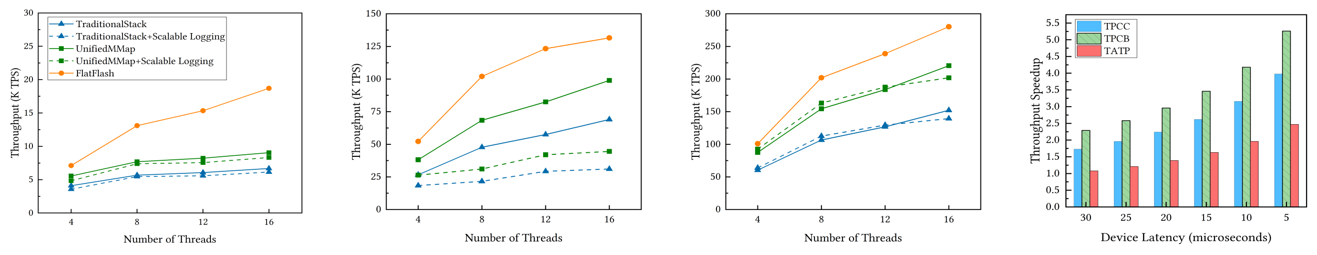
5.6 Persistency Benefit for Transactional Database

To evaluate the persistency benefit of FlatFlash for database, we modify the open-source database Shore-MT [34] and implement per-transaction logging with *TraditionalStack*, *UnifiedMMap*, and FlatFlash respectively. We use Shore-Kits benchmarks that include TPCC, TPCB, and TATP database workloads. We reserve 6GB for the memory manager of Shore-MT database engine. The database size used is 48GB. We vary the number of client threads from 4 to 16.

FlatFlash scales the throughput of transaction operations by up to 3.0× and 4.2× compared to *UnifiedMMap* and *TraditionalStack*, respectively, as demonstrated in Figure 14. Applying the per-transaction logging scheme to *TraditionalStack* and *UnifiedMMap* does not improve their (TPCC and TATP) throughput significantly, because they interact with SSD using page granularity and scalable logging has less chance to group logs and thus increases the I/O traffic as the transaction log entry is usually small (64–1,424 bytes). As TPCB is an update-intensive workload, the per-transaction logging brings benefit to *TraditionalStack* and *UnifiedMMap*, however, FlatFlash still performs the best. FlatFlash treats byte-addressable SSD as a non-volatile memory device, which enables new optimization opportunities for persistence-critical systems. As the SSD latency decreases with new memory technologies (e.g., PCM [56], 3D XPoint [1]), FlatFlash can achieve even more performance benefits as shown in Figure 14d. FlatFlash has the same write amplification factor for logging as other two schemes with centralized log buffer, thus its SSD lifetime is not improved (see Table 1).

5.7 FlatFlash vs. DRAM-Only Systems

In this section, we analyze the cost-effectiveness of FlatFlash in comparison to DRAM-only systems. We rerun the workloads in Table 1 by hosting their entire working sets in DRAM. Our analysis uses the ratio of the DRAM-only system's performance to FlatFlash's performance as the performance slowdown, with the DRAM and SSD cost for hosting all relevant data for the workloads. The unit prices for DRAM and the PCIe SSD used in our experiments are $30/GB and $2/GB, respectively.DRAM-only system increases the server's base cost by $1,500 as more DIMM slots are required. As shown



(a) TPCC **(b)** TPCB **(c)** TATP **(d)** Various device latency

Figure 14. Throughput of running database with scalable logging for TPCC, TPCB, and TATP. For Flash with 20 μs device latency, FlatFlash improves the throughput by 1.1-3.0× and 1.6-4.2× compared to *UnifiedMMap* and *TraditionalStack*. As we reduce the device latency in (d), FlatFlash outperforms *UnifiedMMap* by up to 5.3× when running database with 16 threads.

in Table 3, FlatFlash costs 2.4-15.0× less compared to the DRAM-only setup for different applications, improving the normalized performance per cost by 1.3-3.8×.

6 Related Work

Using SSDs as Memory. SSDs have been used to expand the main memory capacity with the memory-mapped interface and swapping mechanism in operating systems [5, 13, 17, 27, 35, 54, 60, 62, 63]. They treat SSDs as block devices and rely on paging mechanism to manage the data movement between the SSD and host DRAM. FlatFlash exploits the byte-accessibility of SSDs and investigates its impact on the unified memory-storage hierarchy. To improve the performance of accessing SSDs, previous solutions either bypass the storage software stack [9, 14, 27, 38, 55] or move system functions closer to the hardware [22, 36]. FlatFlash shares the similar performance goals with these work by bypassing the storage software stack, but it focuses on exploiting the performance benefits of byte-addressable SSDs.

Byte-addressable SSDs. Jacob *et al.* [30] proposed a memory architecture that leverages DRAM as a cache to provide quasi-byte-addressability for Flash. Jin *et al.* proposed PebbleSSD [33] that architects non-volatile memory inside SSD controller to reduce the metadata management overhead for SSDs. Bae *et al.* proposed a dual, byte- and block-addressable SSD with a persistent memory [6] by leveraging the byte-addressability of PCIe interconnect. FlatFlash acknowledges these work and moves further to rethink the unified memory system design to manage the byte-addressable SSD and exploit its byte-accessibility in computing systems. For instance, FlatFlash performs higher throughput for transactional databases by decentralizing the logging for databases as discussed in § 5.6.

Hybrid Memory Systems. To overcome the scaling limits of DRAM, alternative memory technologies such as non-volatile memories (NVMs) like PCM, STT-RAM, and 3D Xpoint have been proposed [18, 45, 56, 74]. As these technologies have different characteristics in terms of performance, capacity, lifetime, and cost, it is unlikely that a single memory technology will simply replace others to satisfy all the requirements of applications [6, 18, 19, 37]. The byte-addressable SSD has its unique properties. It is developed based on the commodity PCIe attached SSD and provides both byte and block-accessible interfaces. FlatFlash focuses these unique parts, rethinks the current system design, and investigates its performance and persistency benefits for systems software and applications. In hybrid memory systems, the page migration between different memory devices has been a classical topic [10, 32, 61, 72], FlatFlash proposes an adaptive page promotion mechanism dedicated for the byte-addressable SSD and host DRAM with the goal of exploiting their advantages concurrently and transparently. As NVM such as PCM is slower than DRAM, we believe FlatFlash techniques (e.g., page promotion) can shed light on the unified DRAM-NVM hierarchy.

7 Conclusion

In this paper, we exploit the byte-accessibility of SSDs in modern memory-storage hierarchy. We leverage a unified memory interface to simplify the management and programmability of byte-addressable SSDs. We develop an adaptive page promotion mechanism between the SSD and host DRAM, thus programs can exploit benefits from both of them concurrently. We also exploit the byte-granular data persistence of SSDs and apply it to representative software systems such as file systems and database. Experiments show that FlatFlash is a cost-effective solution, which brings significant performance and persistency benefits to a variety of applications.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments and feedback. This work was supported in part by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA, E2CDA-NRI, a funded center of NRI, a Semiconductor Research Corporation (SRC) program sponsored by NERC and NIST, IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizon Network.

References

[1] 3D XPoint™Technology. 2018. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[2] A Closer Look At SSD Power Loss Protection. 2019. https://www.kingston.com/us/ssd/enterprise/technical_brief/tantalum_capacitors.

[3] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. Santa Clara, CA.

[4] An Introduction to the Intel® QuickPath Interconnect. 2009. https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html.

[5] Anirudh Badam and Vivek S. Pai. 2011. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. Boston, MA, 211–224.

[6] Duck-Ho Bae, Insoon Jo, Youra A. Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2018. 2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives. In *Proceedings of the 45Th Annual International Symposium on Computer Architecture (ISCA '18)*. Los Angeles, CA, 425–438.

[7] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *Proc. 9th USENIX NSDI*. San Jose, CA.

[8] Batch TLB Flushes. 2015. https://lkml.org/lkml/2015/4/25/125.

[9] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR '13)*. New York, NY, USA, 22:1–22:10.

[10] Daniel P Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel: from I/O ports to process management*. Oreilly & Associates Inc.

[11] R. Budruk, D. Anderson, and T. Shanley. 2004. *PCI Express System Architecture*. Addison-Wesley.

[12] Cache Coherent Interconnect for Accelerators (CCIX). 2019. http://www.ccixconsortium.com.

[13] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. 2009. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. Washington, DC.

[14] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 2012. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. London, United Kingdom.

[15] 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 (SoCC '10)*. Indianapolis, IN.

[16] Elliott Cooper-Balis, Paul Rosenfeld, and Bruce Jacob. 2012. Buffer-on-board Memory Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. Portland, OR, 392–403.

[17] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. Athens, Greece.

[18] 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 9th European Conference on Computer Systems (EuroSys '14)*. Amsterdam, The Netherlands.

[19] 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 (EuroSys '16)*. London, United Kingdom.

[20] Fusion ioMemory™SX350 PCIe Application Accelerators. 2019. https://www.sandisk.com/business/datacenter/resources/data-sheets/fusion-iomemory-sx350_datasheet.

[21] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. Hollywood, CA, 17–30.

[22] Y. Gottesman and Y. Etsion. 2016. NeSC: Self-virtualizing nested storage controller. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[23] Laura M. Grupp, John D. Davis, and Steven Swanson. 2012. The Bleak Future of NAND Flash Memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. San Jose, CA, 2–2.

[24] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*. Washington, DC.

[25] How you can Boost Acceleration with OpenCAPI, Today!. 2017. http://opencapi.org/2017/11/can-boost-acceleration-opencapi-today/.

[26] HP 805358-512 PC4-19200 512GB LRDIMM. 2019. https://www.serversupply.com/MEMORY/PC4-19200/512GB/HP/805358-512.htm.

[27] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. 2015. Unified Address Translation for Memory-mapped SSDs with FlashMap. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. Portland, OR.

[28] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2015. NVRAM-award Logging in Transaction Systems. In *Proceedings of the 41th International Conference on Very Large Data Bases (VLDB'15)*. Kohala Coast, HI.

[29] Intel 900P Optane NVMe PCIe vs Samsung 960 Pro M.2. 2019. http://ssd.userbenchmark.com/Compare/Samsung-960-Pro-NVMe-PCIe-M2-512GB-vs-Intel-900P-Optane-NVMe-PCIe-280GB/m182182vsm315555.

[30] Bruce Jacob. 2016. The 2 PetaFLOP, 3 Petabyte, 9 TB/s, 90 kW Cabinet: A System Architecture for Exascale and Big Data. *IEEE Comput. Archit. Lett.* 15, 2 (July 2016).

[31] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). *SIGARCH Comput. Archit. News* 38, 3 (June 2010).

[32] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. 2010. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. In *Procedding of the 16th International Symposium on High-Performance Computer Architecture (HPCA '10)*. Bangalore, India.

[33] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. Improving SSD Lifetime with Byte-addressable Metadata. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '17)*. Alexandria, VA.

[34] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '09)*. Saint Petersburg, Russia.

[35] Myoungsoo Jung and Mahmut Kandemir. 2013. Revisiting Widely Held SSD Expectations and Rethinking System-level Implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '13)*. Pittsburgh, PA.

[36] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. 2018. Designing a True Direct-Access File System with DevFS. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*. Oakland, CA, 241–256.

[37] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS - OS Design for Heterogeneous Memory Management in Datacenter. In *the 44th International Symposium on Computer Architecture (ISCA'17)*. Toronto, Canada.

[38] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. Denver, CO.

[39] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Jan Vesely, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. 2018. LATR: Lazy Translation Coherence. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Williamsburg, VA.

[40] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. Raleigh, NC.

[41] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. Hollywood, CA, 31–46.

[42] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-performance Concurrency Control Mechanisms for Main-memory Databases. *Proc. VLDB Endow.* 5, 4 (Dec. 2011).

[43] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC Challenge (HPCC) Benchmark Suite. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. Tampa, FL.

[44] Chris Mellor. 2017. Samsung drops 128TB SSD and kinetic-type flash drive bombshells. https://www.theregister.co.uk/2017/08/09/samsungs_128tb_ssd_bombshell/.

[45] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan. 2012. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. *IEEE Computer Architecture Letters* (July 2012).

[46] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.* 17, 1 (March 1992).

[47] Jayashree Mohan, Rohan Kadekodi, and Vijay Chidambaram. 2017. Analyzing IO Amplification in Linux File Systems. *arXiv preprint arXiv:1707.08514* (2017).

[48] NVMe 1.3 Specification . 2019. http://nvmexpress.org/resources/specifications/.

[49] NVMe SSD 960 PRO/EVO | Samsung Consumer V-NAND SSD. 2019. http://www.samsung.com/semiconductor/minisite/ssd/product/consumer/ssd960/.

[50] OpenCAPI Technical Specifications. 2019. http://opencapi.org/technical/specifications/.

[51] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2011. The Case for RAMCloud. *Commun. ACM* 54, 7 (July 2011).

[52] K. Pagiamtzis and A. Sheikholeslami. 2006. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey. *IEEE Journal of Solid-State Circuits* 41, 3 (March 2006).

[53] PCIe 3.0 Specification. 2019. https://pcisig.com/specifications.

[54] Roger Pearce, Maya Gokhale, and Nancy M. Amato. 2010. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. Washington, DC.

[55] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO, 1–16.

[56] 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 36th Annual International Symposium on Computer Architecture (ISCA '09)*. Austin, TX.

[57] Rajat Kateja and Anirudh Badam and Sriram Govindan and Bikash Sharma and Greg Ganger. 2017. Viyojit: Decoupling Battery and DRAM Capacities for Battery-Backed DRAM. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA'17)*. Toronto, Canada.

[58] Redis. 2019. https://redis.io/.

[59] Samsung Z-NAND Technology Brief. 2017. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[60] Mohit Saxena and Michael M. Swift. 2010. FlashVM: Virtual Memory Management on Flash. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. Boston, MA, 187–200.

[61] Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. 2014. Transparent Hardware Management of Stacked DRAM As Part of Memory. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. Cambridge, United Kingdom.

[62] Yongseok Son, Hyuck Han, and Heon Young Yeom. 2015. Optimizing File Systems for Fast Storage Devices. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR '15)*. New York, NY, USA.

[63] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. 2016. Efficient Memory-Mapped I/O on Fast Storage Device. *Trans. Storage* 12, 4 (May 2016).

[64] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A flexible framework for file system benchmarking. *The USENIX Magazine* 41, 1 (2016).

[65] The Gen-Z Consortium. 2019. https://genzconsortium.org/.

[66] K. Therdsteerasukdi, G. S. Byun, J. Ir, G. Reinman, J. Cong, and M. F. Chang. 2011. The DIMM tree architecture: A high bandwidth and scalable memory system. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*.

[67] Ultra-Low Latency with Samsung Z-NAND SSD. 2017. *White Paper* (2017).

[68] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann. 2012. NVMalloc: Exposing an Aggregate SSD Store as a Memory Partition in Extreme-Scale Machines. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*.

[69] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB'14)*. Hangzhou, China.

[70] Xilinx Virtex-7 FPGA VC709. 2019. https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html#documentation.

[71] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics (MDS '12)*. Beijing, China.

[72] HanBin Yoon, Rachata Meza, Justin Ausavarungnirun, Rachael A. Harding, and Onur Mutlu. 2012. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD 2012) (ICCD '12)*. Washington, DC, USA, 8.

[73] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. De-indirection for Flash-based SSDs with Nameless Writes. In *Proc. 10th USENIX FAST*. San Jose, CA.

[74] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. Davis, CA.