LiveSSD: A Low-Interference RAID Scheme for Hardware Virtualized SSDs

You Zhou, Fei Wu, *Member, IEEE*, Weizhou Huang, and Changsheng Xie, *Member, IEEE*

***Abstract*—Hardware virtualization has been increasingly used to provide performance isolation between multiple tenants sharing an SSD. It exploits the SSD's highly parallel architecture by allocating dedicated flash dies to each tenant. On the other hand, intra-SSD RAID, which stripes data and parity across flash dies, is essential to enhance storage reliability, such as protecting data against die failures and read errors. However, parity updates introduce I/O interference, degrading tenants' performance significantly, and violating performance isolation. To solve this problem, we propose a low-interference RAID scheme for hardware virtualized SSDs, called LiveSSD. Flash pages with the same offset across dies constitute a stripe in a RAID-4 manner. High-speed NVRAM is employed as parity storage. Thus, LiveSSD allows each tenant to read/write its flash die(s) independently and avoids parity updates being a performance bottleneck. Nonetheless, parity updates introduce I/O interference during garbage collection, i.e., extra reads of invalid flash pages. LiveSSD actively conducts parity updates in advance by utilizing both page access feature of flash memory and idle time in workloads. Extensive simulation results show that LiveSSD enables RAID protection in a hardware-virtualized SSD with minimum I/O interference caused by parity updates.**

***Index Terms*—Intra-SSD RAID, NAND flash memory, nonvolatile RAM (NVRAM), solid-state drive (SSD), virtualization.**

I. Introduction

NAND flash-based solid-state drives (SSDs) are commonplace in data centers, hosting a wide variety of data-related services [1]. To provide large capacity and high bandwidth, modern SSDs employ a highly parallel architecture. Many flash dies, which can store data and operate independently, are interconnected to form a single storage device [2]. For example, a latest enterprise SSD has 8TB flash memory consisting of 128 dies of 64 GB [3].

To improve storage utilization and cost effectiveness, *SSD virtualization* has been increasingly used [1], [4]–[6]. A virtualized SSD is shared by multiple tenants, e.g., applications, virtual machines, and database instances. However, tenants with different I/O patterns as well as flash management routines, i.e., flash translation layer (FTL) tasks, compete for limited storage resources, e.g., the bandwidth and storage space. Subsequent *I/O interference* fluctuates tenants' I/O performance unexpectedly and may result in the violation of service-level objectives (SLOs) [5]–[7].

Both software-based and hardware-based techniques for SSD virtualization have been proposed by academic and industrial researchers. The goal is to provide performance isolation between multiple tenants. Note that a die is the minimum parallel unit of independently executing flash operations in an SSD [8]. *Software virtualization* allocates dedicated flash blocks to each tenant and a die can be shared and accessed by multiple tenants [9]–[13]. Although I/O throughput and storage space utilization can be improved by striping data of each tenant across flash dies, significant I/O interference between tenants exists on each die.

By contrast, *hardware virtualization* allocates a fixed number of dedicated dies to each tenant [5], [6], [14], [15]. Each tenant handles its I/O requests and *local* FTL tasks on its die(s) in a strongly isolated environment. Local FTL tasks are flash management routines triggered by I/O activities of each tenant itself, such as garbage collection (GC) [9]. Hence, hardware virtualization achieves performance isolation between tenants. However, *global* FTL tasks, i.e., flash management routines triggered by I/O activities of all the tenants, still interfere with tenants' I/O performance. For example, Huang *et al.* identified wear leveling as such a global FTL task [5]. As tenants may write and thus wear their die(s) at different rates, FlashBlox was proposed to swap dies between tenants periodically.

In this article, we identify parity updates of die-level RAID as such a global FTL task. Flash memory suffers from reliability issues, including bit errors (caused by device and circuit noises) and failures of storage elements (such as dies and blocks) [16], [17]. For example, an industrial research reports a die-failure probability of 0.035% [17]. Although error correction codes (ECCs) are used to reduce bit errors, they cannot cope with element failures [18]. Also, intra-SSD RAID can correct bit errors beyond the capability of ECCs. For example, the uncorrectable bit error rate (UBER) is reduced to 10^{-18} when the UBER with an ECC is 10^{-11} and one parity page is

Manuscript received September 9, 2019; revised January 5, 2020, April 8, 2020, and June 11, 2020; accepted July 31, 2020. Date of publication August 11, 2020; date of current version June 18, 2021. This work was supported in part by the NSFC under Grant 61902137, Grant U1709220, Grant 61821003, Grant 61872413; in part by Key-Area Research and Development Program of Guangdong Province under Grant 2019B010107001; in part by the 111 Project under Grant B07038; and in part by the Key Project of Shandong Wisdom Joint Fund under Grant ZR2019LZH009. This article was recommended by Associate Editor C. L. Yang. *(Corresponding author: Fei Wu.)*

You Zhou is with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China, and also with the School of Optical and Electronic Information, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: zhouyou2@hust.edu.cn).

Fei Wu, Weizhou Huang, and Changsheng Xie are with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: wufei@hust.edu.cn; huangweizhou@hust.edu.cn; cs_xie@hust.edu.cn).

Digital Object Identifier 10.1109/TCAD.2020.3015908

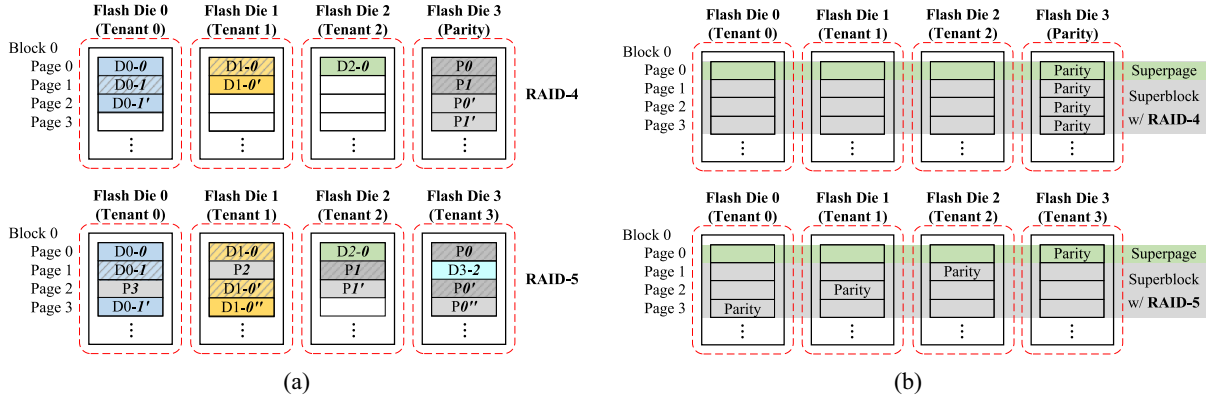


Fig. 1. Problems of applying die-level RAID in hardware-virtualized SSDs. Assume the SSD has four flash dies assigned to multiple tenants; each flash block contains 4 pages; boxes filled with gray lines indicate invalid data. Logical grouping-based RAID in (a): Dx-**y**: logical page *y* of Tenant *x*, P**y**: parity page of logical stripe *y*; if parity is stored in a dedicated die like RAID-4, the parity die receives extensive updates and thus becomes a performance bottleneck; if parity is distributed across dies like RAID-5, data writes of a tenant causes parity updates in other tenants' dies, introducing significant I/O interference between tenants. Physical grouping-based RAID in (b): a superpage constitutes a stripe with parity and this design requires multiple dies are written at the same offset; however, this requirement cannot be met, as each tenant writes its die(s) independently.

added for 128 data pages. Hence, intra-SSD RAID is essential to protect data against ECC-uncorrectable bit errors and element failures.[1] Data pages are grouped into stripes and each stripe including the parity page is stored across many flash dies. In a hardware-virtualized SSD, flash dies are allocated exclusively to multiple tenants. It would consume too much storage space to implement per-tenant RAID protection. For example, assuming each tenant requires two dies for data storage, on average, three dies (50% more space) need to be allocated. Therefore, RAID should be implemented at the device level in a cross-tenant manner.

When die-level RAID meets hardware virtualization in an SSD, contradictions emerge. Generally, die-level RAID schemes can be classified into logical grouping-based RAID, called LG-RAID, and physical grouping-based RAID, called PG-RAID. As shown in Fig. 1(a), LG-RAID groups a fixed set of logical data pages into a stripe. If all parity pages are placed in one flash die in a RAID-4 manner, the parity die would become a performance bottleneck due to extensive parity updates. If parity pages are distributed across flash dies in a RAID-5 manner [20]–[24], data writes in a tenant's die(s) would cause parity updates in other tenants' die(s). This introduces significant I/O interference between tenants, while hardware virtualization aims at performance isolation. Hence, LG-RAID is not an option for a hardware-virtualized SSD.

PG-RAID dynamically groups logical data pages into a stripe based on the write order. Modern SSDs usually organize and manage flash storage in *superpages/superblocks*, based on which lightweight PG-RAID schemes are implemented [16], [19], [25]–[27].[2] Flash memory performs reads/writes and erases in the units of a page and a block, respectively. A *superpage/superblock* refers to flash pages/blocks with the same offset across multiple dies. Data is written to flash memory in a granularity of a superpage and GC is conducted in a granularity of a superblock. Each superpage constitutes a RAID stripe, where one or even multiple parity pages are generated for the remaining pages. However, flash memory cannot be updated in place and flash pages in a block can only be written sequentially. The superpage design requires multiple dies to maintain the same write offset. Such write requirements contradict hardware virtualization, where dies are written independently by tenants at different rates, as shown in Fig. 1(b). As a result, PG-RAID cannot be directly applied in a hardware-virtualized SSD.

In this article, we propose a low-interference RAID scheme for hardware virtualized SSDs, called LiveSSD, to achieve both reliability enhancement and performance isolation. LiveSSD adopts the superpage-based RAID-4 design and employs high-speed nonvolatile RAM (NVRAM) [e.g., phase change memory (PCM)] as parity storage. Thus, each tenant can write its flash die(s) independently at different rates and parity updates are *not* a performance bottleneck. Nonetheless, parity updates still introduce I/O interference to tenants and degrade their performance (by an average of 18.3% as shown in Section III). During GC, extra flash reads of invalid pages are needed for parity updates. To reduce such I/O interference, LiveSSD actively removes invalid flash pages from parity protection in advance by utilizing both page access feature of flash memory and idle time in workloads. Extensive simulation results show that LiveSSD enables RAID protection in a hardware-virtualized SSD with minimum I/O interference caused by parity updates.

The reminder of this article is organized as follows. Section II presents an overview of the background and related work. Motivation and our design are detailed in Sections III and IV, respectively. We evaluate the design in Section V and conclude this article in Section VI.

[1]RAID within an SSD is also called redundant array of independent NAND (RAIN) [19]. It can be implemented at the chip or die level. We assume die-level RAID is used in this article and the design will be no difference if chip-level RAID is used.

[2]The superpage/superblock design has several advantages. First, SSD internal parallelism is fully exploited to maximize the throughput. Second, it enables lightweight PG-RAID schemes. When data pages are written, one or more parity pages can be calculated incidentally and written together to a superpage. Third, flash management is simplified, since FTL metadata can be maintained at the superblock level rather than block level.

II. Background and Related Work

Basics of Flash Memory: A NAND flash memory cell stores bit states by holding a number of electrons, which determine its threshold voltage [16]. An array of flash cells are organized into *blocks*, each of which contains hundreds of *pages* (e.g., 16 KB). Flash memory performs read operations (by sensing cells' threshold voltages) and program/write operations (by injecting electrons into cells) in a unit of a page. Erase operations, which eject electrons from cells, are performed in a unit of a block. Notably, a block must be erased before being written and its pages must be written sequentially. In addition, flash memory has limited endurance/lifetime and can typically sustain thousands of program/erase (P/E) cycles.

Flash Translation Layer: Due to unique features, flash memory is managed by an FTL [28]. It performs out-of-place updates, where new data is written to free flash pages and old data pages are invalidated. Address mapping is performed to dynamically translate host logical page addresses to physical/flash page addresses. Modern SSDs usually employ a page-level mapping for high write performance and addressing flexibility [29]. A logical page can be mapped to any flash page. To reclaim invalid flash pages, GC is carried out by migrating valid data in a block and then erasing the block [24]. In addition, wear leveling is performed to prolong SSD lifetime by distributing P/E cycles over flash blocks evenly [5].

Architecture of an SSD: To improve I/O throughput and storage capacity, flash-based SSDs are architected with abundant parallelism. An SSD contains multiple channels, each of which connects multiple flash chips [8]. Each chip consists of several dies. A die is the basic parallel unit that can process flash operations independently. The SSD throughput can be scaled up by adding more dies. In addition, each die is composed of several planes and each plane contains many flash blocks. We do not further discuss the channel-, chip-, and plane-level hierarchies in this article for simplicity, as they do not essentially impact our design.

Flash Storage Reliability: Reliability of flash storage has been a critical concern, since flash memory is vulnerable to bit errors and suffers from occasional element failures. Bit errors are caused by various types of noises, such as P/E cycling, charge leakage (i.e., retention errors), and read/program disturbances [16]. Especially, the raw bit error rate (RBER) has been increasing as flash technologies aggressively scale to increase storage density [30]. For example, more layers are stacked in 3-D structures and more bits are stored per cell. The flash controller employs ECCs to detect and correct bit errors in flash pages [31], so the UBER can be lowered to an acceptable level (e.g., 10^{-14}). However, the RBER and UBER significantly grow as P/E cycles are consumed [32], [33]. Furthermore, flash storage elements (e.g., dies) may fail, like any other integrated circuits [17]. Such failures, which cannot be handled by ECCs, become serious as more dies are integrated into a single device.

Die-Level RAID: The failure of a die does *not* impact the functionality of other dies, so die-level RAID is widely employed to enhance the SSD reliability [16]. Both logically grouping-based RAID-5 (LG-RAID-5) and superpage-based PG-RAID schemes have been proposed. LG-RAID-5 statically groups logically consecutive data pages into stripes and spreads parity pages over flash dies. Since parity updates caused by updates of logical pages degrade the performance, several techniques have been proposed to delay parity updates on flash memory by utilizing idle time [20] or caching parity in NVRAM [21] or logging small writes in hidden mirroring dies [23]. By contrast, PG-RAID dynamically groups logical data pages into stripes in the write order and stores each stripe on flash pages with the same offset across dies [16], [19], [25], [26]. This design maximizes the write throughput and results in little parity updating overheads. In addition, RAID stripes can be stored dynamically according to the wear status of flash blocks to improve storage reliability [22], [27]; RAID construction can be utilized to serve data reads when the destination die is occupied by GC operations [24].

To maintain I/O isolation between tenants in a hardware-virtualized SSD, we adopt the superpage-based RAID-4 design with NVRAM as a parity storage. Different from the existing techniques that delay parity updates, our design actively conducts parity updates in advance to eliminate the RAID-induced I/O interference. Besides, the enhancements proposed in the existing PG-RAID techniques (e.g., permitting partial stripe writes in [26] and wear-aware stripe organization in [27]) are orthogonal to our proposed parity updating techniques.

SSD Virtualization: SSDs have become indispensable in data centers due to many advantages over traditional hard drives, such as high performance, high reliability, and low energy consumption. Note that virtualization is the key technique in data centers to improve storage utilization and simplify storage management [1], [34]. The SSD is a great carrier to implement storage virtualization for two reasons. First, its highly parallel architecture and built-in software layer (i.e., FTL) enable efficient performance isolation. For example, hardware isolation can be achieved by assigning separate dies to each tenant and software isolation can be achieved by I/O scheduling [5]. Second, SSD virtualization is promoted by recent technical advances. Flash technologies scaling has been significantly increasing storage density and thus the SSD capacity, so each SSD is likely to host more tenants. Moreover, interface techniques facilitate SSD virtualization. The NVMe protocol and PCIe interface have been widely used for SSDs to communicate with the host system. They support features that can virtualize a drive as multiple virtual instances, such as namespaces and I/O determinism in NVMe and single-root I/O virtualization (SR-IOV) in PCIe [35].

Both software and hardware virtualization techniques have been proposed to improve performance isolation between tenants sharing an SSD. Software virtualization allows a flash die to be shared by multiple tenants. Thus, GC operations globally affect all tenants' I/O performance. According to each tenant's I/O characteristics, GC-induced interference can be balanced through over-provisioning space isolation [9] and time budget compensation [10], or be isolated through block allocation within different sets of dies [12] and die reservation for reads, writes, and GC operations [11]. By contrast, hardware virtualization achieves strong isolation through die-based allocation [6], [14]. As tenants write their dies at different

rates, FlashBlox periodically exchanges dies between tenants to improve wear leveling [5]. However, none of these hardware virtualization techniques consider the I/O interference caused by die-level RAID schemes.

Nonvolatile RAM: In this article, we refer to NVRAM as high-speed and byte-addressable nonvolatile memory not including flash memory, such as PCM and magnetic RAM (MRAM) [36]. Despite lower density and higher cost, NVRAM provides many benefits over NAND flash memory. For example, reads/writes can be conducted at the byte level in nanoseconds and data can be updated in place. Therefore, integrating NVRAM into flash-based SSDs has become a popular solution to achieve various goals (e.g., a persistent data cache and fast metadata storage) [37], [38]. Recently, Intel introduced the Intel Optane memory H10 device which combines the responsiveness of Intel Optane technology (i.e., PCM) with the storage capacity of quad-level cell (QLC) 3-D flash in an M.2 form factor [39]. Likewise, we employ NVRAM to store RAID-4 parity to avoid parity updates being a performance bottleneck in this article.

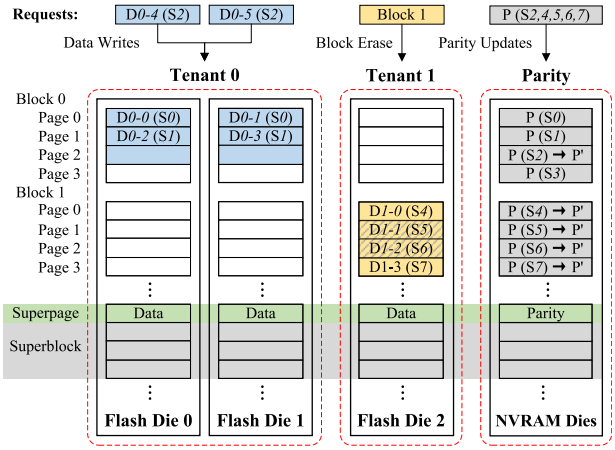


Fig. 2. Parity updates in NV-PG-RAID-4. Assume the SSD contains three flash dies (two are assigned to Tenant 0 while one to Tenant 1) for data storage and NVRAM dies for parity storage. D*x*–*y*: data *y* of Tenant *x*; S*z*: RAID stripe *z*; P: parity; boxes filled with gray lines indicate invalid data. Data writes of Tenants 0 and a block erase of Tenant 1 generate parity updates on NVRAM.

III. Motivation

In a multitenant SSD, performance isolation is achieved through isolating each tenant from other tenants and FTL tasks. Although hardware virtualization provides strong isolation, there is a fundamental conflict between it and die-level RAID, as shown in Fig. 1. LG-RAID results in coupled performance between tenants when LG-RAID-5 is adopted or parity updates being a performance bottleneck when LG-RAID-4 is adopted, while the superpage-based writing requirement of PG-RAID cannot be met in hardware virtualization.

A naive solution to enable parity protection in a hardware-virtualized SSD is to adopt the superpage-based RAID-4 design and employ high-speed NVRAM as parity storage, called the *NV-PG-RAID-4* scheme. NVRAM provides high throughput and supports in-place updates, so parity updates are *not* a performance bottleneck. The parity storage is as large as a flash die. Pages with the same offset across flash dies and parity storage constitute a superpage, i.e., a RAID stripe. Parity updates are caused by *data writes* on free flash pages (which set some bits from "1" to "0") and GC operations (which erase all bits to 1) in victim flash blocks, as shown in Fig. 2. The read–modify–write (RMW) method is used for parity updates, because the read–construct–write (RCW) method violates performance isolation between tenants. The parity construction caused by a tenant's writes requires data reads on other tenants' dies. Note that LG-RAID-4 with NVRAM as parity storage is not considered, because whenever a logical data page is updated, its old version needs to be read from flash memory first to modify the relevant parity page. By contrast, PG-RAID-4 dynamically groups newly written data pages into stripes, where parity is calculated without reading "old data pages."

Although NV-PG-RAID-4 isolates parity updates from data writes of tenants, parity updates caused by GC operations interfere with tenants' I/O performance. In a hardware-virtualized SSD, each tenant writes data to its flash die(s) independently, which leads to parity updates on NVRAM storage. As flash memory does not support in-place updates, data is always written to free flash pages and old data pages are invalidated. Parity updates during tenant writes are conducted by accessing only NVRAM storage: 1) reading old parity; 2) calculating new parity based on old parity and newly written data; and 3) writing new parity. Hence, no I/O interference occurs between tenants during data writes. Parity updates caused by flash block erases during GC operations are performed as follows: 1) read both valid and *invalid* data pages in victim blocks; 2) reading old parity; 3) calculating new parity based on old parity and victim pages; and 4) writing new parity. Valid pages need to be read no matter RAID is employed or not, because they need to be migrated to free flash pages. However, reading invalid pages introduces extra flash reads, degrading tenants' I/O performance. In the worst case, where all pages in the victim block are invalid, the extra flash reads are expensive. Assuming the time flash page read and block erase is 50 μs and 5 ms and each block contains 1024 pages, reading the whole block takes 10 times higher time than erasing the block. We refer to such performance overheads as *RAID-induced I/O interference*. A higher ratio of invalid pages in victim flash blocks indicates larger I/O interference.

We have conducted experiments to quantify the RAID-induced I/O interference in a hardware-virtualized SSD under various traces. Detailed experimental configurations are shown in Section V-A. NV-PG-RAID-4 is compared to the *RAID-0* scheme, which has no parity protection and thus no RAID-induced I/O interference. As shown in Fig. 3, NV-PG-RAID-4 increases the average I/O response time by an average of 21.9%. In a group of synthetic write-only traces, the increase of I/O response time is as high as 43.7% under 100% sequential writes (*w_100_seq*), while between 20.3% and 24.3% under traces with random writes (*w_70_seq, w_30_seq, w_0_seq*). This reveals that sequential writes cause larger

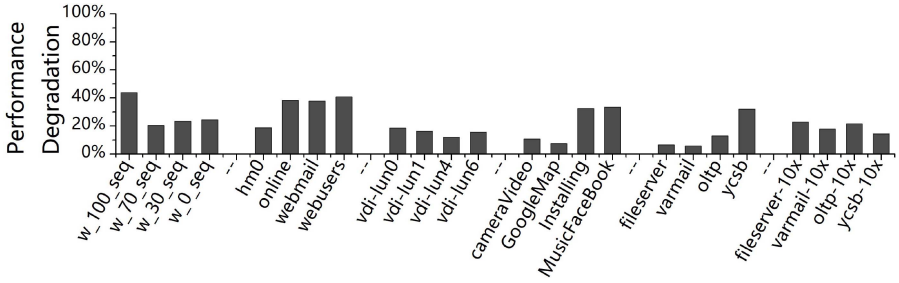


Fig. 3. Performance degradation caused by RAID-induced I/O interference in NV-PG-RAID-4. Six groups of various traces are used in the experiments (detailed configurations are shown in Section V-A). Each group has four traces, which indicate four individual tenants and run simultaneously on separate dies. Experimental results show that the RAID-induced I/O interference increases I/O response time by an average of 21.9% and up to 43.7%.

RAID-induced I/O interference than random writes, because sequential writes lead to more invalid flash pages in victim blocks during GC operations. Meanwhile, we notice that many popular applications and file systems are designed with I/O sequentiality in mind. For example, log-structured file systems [40] and LSM-tree-based key-value stores [41] transform random writes to sequential writes, which are friendly to storage devices. These observations demonstrate that NV-PG-RAID-4 would introduce nontrivial RAID-induced I/O interference.

Neither existing RAID schemes nor NV-PG-RAID-4 can be directly applied in hardware-virtualized SSDs. We are thus motivated to design a novel RAID scheme. To our best knowledge, this is the first paper that targets the problem of integrating die-level RAID into a hardware-virtualized SSD.

IV. Design

A. Overview of LiveSSD

LiveSSD is a hardware-virtualized SSD shared by multiple tenants and employs RAID to improve storage reliability, as shown in Fig. 4. At a high level, LiveSSD functions as several virtual SSDs (vSSDs) and each vSSD is used exclusively by a tenant for data storage. Inside LiveSSD, embedded processors and volatile RAM are utilized to execute two major software components, host interface logic (HIL) and FTL, like any other modern SSDs. The HIL provides necessary interfaces to the host for managing (e.g., allocating and deallocating) and accessing vSSDs. This is easily supported by the existing interface techniques. For example, the SR-IOV protocol allows a PCIe SSD to be exposed as multiple virtual functions; an NVMe SSD can be exposed as multiple sets or namespaces [6]. Each virtual function or NVMe set/namespace corresponds to a vSSD. Each vSSD runs a virtual FTL with the same algorithms, including address mapping, GC, wear leveling, and RAID management. We focus on *RAID management* in this article, as the goal is to eliminate the RAID-induced I/O interference. The other algorithms are no difference from those in traditional SSDs. For performance isolation, virtual FTLs are executed in a round-robin manner by processors and are allocated with dedicated RAM space to cache FTL metadata and buffer data.

Avoiding competitions on storage resources is critical to provide performance isolation between tenants. LiveSSD allocates each vSSD with dedicated flash dies for data storage, so

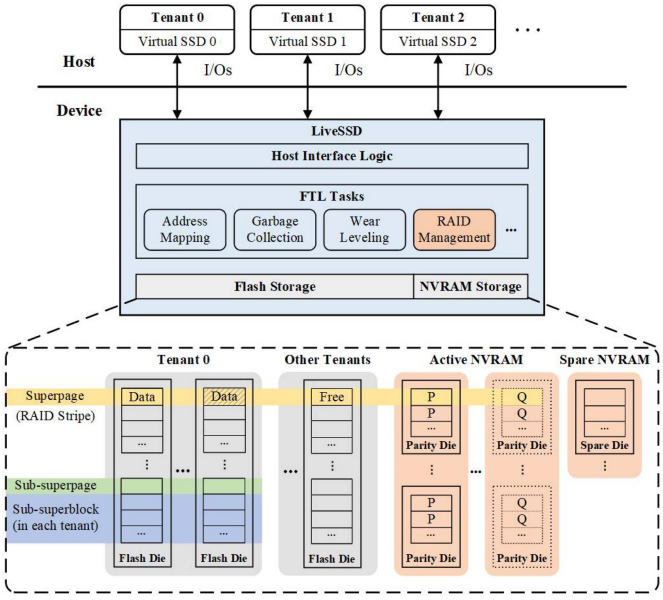


Fig. 4. Overview of LiveSSD. Flash dies are exclusively assigned to multiple tenants for data storage and high-speed NVRAM dies are employed for parity storage. LiveSSD maintains superpage-level parity in a PG-RAID manner (one or more parity can be included on demand). Parity updates are merged for each tenant via sub-superpage-based writes/reclamation and are conducted actively utilizing page access feature of flash memory and idle time in workloads.

each tenant can read/write its die(s) independently. The storage capacity and throughput of a vSSD/tenant can be scaled up on demand by allocating more flash dies [5]. To guarantee storage reliability, the superpage-based die-level PG-RAID design is employed. High-speed and byte-addressable NVRAM dies are employed to store parity, so parity updates will *not* become a performance bottleneck. Since an NVRAM die is much smaller than a flash die due to lower storage density [42], multiple NVRAM dies are grouped to serve as parity storage, called *active NVRAM*. Pages with the same offset across flash dies and active NVRAM storage constitute a superpage/stripe.

An SSD may have different levels of reliability requirements, for example, an enterprise SSD with many dies may need double- or even triple-parity RAID. The architecture of LiveSSD is inherently compatible with different reliability levels, i.e., each superpage can include one or more parity pages on demand. Assuming the SSD has N flash dies and adopts M-parity RAID, the stripe size is N:M. M groups of NVRAM dies would be employed and each group, as large as a flash die, would

accommodate an instance of parity. When M is equal to 1 or 2, the reliability level is equivalent to RAID-4/5 or RAID-6, respectively. Improving the reliability level would increase the hardware cost, i.e., the number of NVRAM dies, but would *not* increase the RAID-induced I/O interference. When N data pages are written to a superpage across flash dies, M parity pages can be calculated and written to M NVRAM dies in parallel. During GC operations, RAID-protected invalid flash pages are read once and then M parity pages in each relevant superpage can be updated on M NVRAM dies simultaneously. If a larger M is required and channel resources are limited, more NVRAM dies may be connected to the same channel bus, where data transfers to NVRAM dies are interleaved. This may incur trivial performance overheads, as the data transfer speed is high (e.g., 800 MB/s [43]). In the following sections, we describe technical details and experimental results of LiveSSD with single parity (i.e., PG-RAID-4) by default.

B. Low-Interference RAID Scheme

Although high-speed NVRAM is used as parity storage, parity updates still interfere with tenants' I/O performance, as discussed in Section III. LiveSSD employs a low-interference RAID scheme to alleviate the I/O interference. Specifically, the *tenant-level parity updating technique* reduces the total number of parity updates; the *active parity updating technique* reduces extra flash reads of invalid pages during GC operations and thus the RAID-induced I/O interference.

1) Tenant-Level Parity Updating: This technique merges the parity updates of each tenant. A *sub-superpage/sub-superblock* refers to flash pages/blocks with the same offset across the flash dies of a tenant. Tenants with different numbers of flash dies have different sub-superpage sizes. Each superpage/stripe contains one sub-superpage from each tenant. The FTL writes data in a granularity of a sub-superpage and conducts GC in a granularity of a sub-superblock. Therefore, it results in one parity page update to write or reclaim multiple flash pages in a sub-superpage, as shown in lines 12 and 28 of Algorithm 1. This technique also inherits the advantages of the superpage-based design employed in traditional SSDs, such as fully exploiting the parallelism [16], [44]. Parity updates of different tenants are not merged. Whenever a sub-superpage is written or reclaimed by a tenant, parity on NVRAM is updated. Nonetheless, parity updates are *not* a performance bottleneck as analyzed in Section IV-C.

Additionally, a tenant may *not* have enough buffered dirty data to form a sub-superpage write. Meanwhile, dirty data needs to be persisted periodically as a sudden power failure may cause data loss. In this case, LiveSSD writes dirty data to the flash page(s) of a sub-superpage and updates the corresponding parity immediately. When new dirty data is being written, the partially written sub-superpage is filled before the next free sub-superpage is chosen for writing.

2) Active Parity Updating: This technique actively removes invalid flash pages from parity protection in advance without degrading tenants' I/O performance and storage reliability. The FTL performs out-of-place updates due to the erase-before-write feature of flash memory. Old data pages become invalid

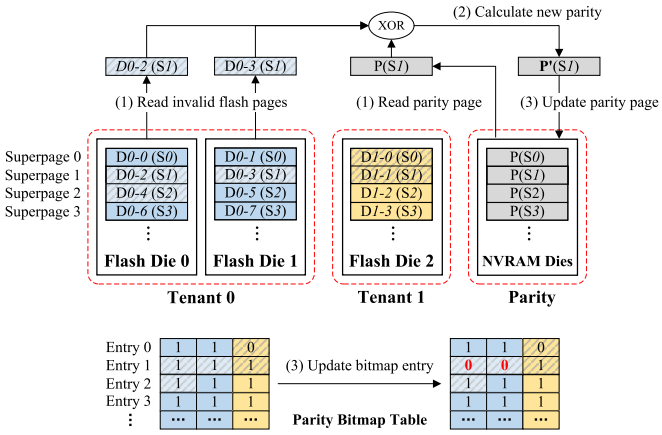


Fig. 5. Example of active parity updates. Assume the SSD has three flash dies (assigned to Tenants 0 and 1) and each flash block contains 4 pages. D*x*–*y*: data *y* of Tenant *x*; S*z*: RAID stripe *z*; P: parity; boxes filled with gray lines indicate invalid data. Two flash pages in sub-superpage 1 of Tenant 0 are invalid and can be removed from parity protection in advance as follows: 1) reading target invalid flash pages (T0-2 and T0-3) and the corresponding parity page on NVRAM (P(S1)); 2) calculating new parity as the XOR results of them; and 3) updating the parity page and its bitmap entry on NVRAM.

when data is updated and need to be removed from parity protection before being erased/reclaimed during GC. With active parity updates for invalid flash pages, a superpage may have only some flash pages in parity protection while not for the other. LiveSSD maintains a *parity bitmap table*, where each entry records whether flash pages in each superpage are parity-protected or not. Assume an SSD has 8-TB flash storage with 128 flash dies and the page size is 16 KB. Each bitmap entry contains 128 bits and the table is as small as 64 MB. LiveSSD stores each entry along with the corresponding parity page on NVRAM, as it is persistent and supports in-place updates. Fig. 5 illustrates how to perform active parity updates.

Active parity updates are conducted by utilizing both the *page access feature of flash memory* and *idle time* in workloads. Flash memory performs reads/writes in a unit of a page whose typical size is 16 KB, while typical sizes of I/O requests are in a unit of 4 KB [45]. When a data page is *partially* updated by a tenant, the old data page needs to be read first to pad the new data page. Then, this new page is written to a free flash page and the old data page is invalidated. LiveSSD utilizes such prereads during partial page updates to perform active parity updates, as shown in line 6 of Algorithm 1.

Idle time commonly exists between I/O requests in real-world workloads [46], [47]. LiveSSD predicts the idle time length between I/O requests of each tenant. If the predicted length of the remaining idle time between the completing of the last I/O request and the arriving of the next I/O request is longer than a threshold, background operations can be triggered. Specifically, LiveSSD utilizes the idle time to read RAID-protected invalid flash pages in sub-superblocks and performs active parity updates, as shown in line 19 of Algorithm 1. Sub-superblocks are chosen in the same way as choosing GC victims, so invalid pages that most likely cause I/O interference can be removed from parity protection timely. The threshold is set as long as the time of reading a flash page (e.g., 0.05 ms) by default. These background operations

Algorithm 1 Parity Updating of LiveSSD for Each Tenant. Parity Updates Are Conducted During Data Writes of Tenants (Lines 1–13), Idle Time (Lines 14–22), and GC (Lines 23–38)

```
1: procedure TenantWriteUpdate
                    ▷ Write a sub-superpage in Superpage C
2:     Read parity page C from NVRAM
3:     for each written data page do
4:         if it is a partial page update then
5:             Read old data page from Superpage P on flash
6:             Update parity page P and its bitmap on NVRAM by
   removing old data page from parity protection
7:             Merge newly written data with old data
8:         end if
9:         Calculate parity page C by adding new data page
10:    end for
11:    Write the sub-superpage and invalidate old pages on flash
12:    Update parity page C and its bitmap on NVRAM
13: end procedure
14: procedure IdleTimeUpdate
      ▷ Remove invalid flash pages from protection at idle time
15:    Predict idle time length after completing an I/O request
16:    if it is longer than a threshold and no I/Os arrive then
17:        Check the parity bitmap and page validity table
18:        if a flash page is invalid and RAID-protected then
19:            Update its parity page and bitmap on NVRAM by
   removing it from parity protection
20:        end if
21:    end if
22: end procedure
23: procedure GCUpdate
    ▷ Reclaim a victim sub-superblock during garbage collection
24:    for each sub-superpage in the sub-superblock do
25:        Check the parity bitmap table and page validity table
26:        if the sub-superpage has RAID-protected pages then
27:            Read protected pages from Superpage P on flash
28:            Update parity page P and its bitmap on NVRAM by
   removing them from parity protection
29:            for each protected flash page do
30:                if it contains valid data then
31:                    Buffer this valid page
32:                end if
33:            end for
34:        end if
35:    end for
36:    Flush buffered valid pages to flash via sub-superpage writes
37:    Erase the victim sub-superblock
38: end procedure
```

are suspended whenever I/O requests arrive and then resumed whenever another idle time cycle is predicted. The exponential smoothing method is adopted to predict the length of next (ith) idle time cycle, as shown in (1) (α is set as 0.5) [48]. t^{i-1}_{real} and t^{i-1}_{predict} refer to the real and predicted lengths of last idle time cycle, respectively

$$t^{i}_{\text{predict}} = \alpha * t^{i-1}_{\text{real}} + (1-\alpha) * t^{i-1}_{\text{predict}}. \tag{1}$$

C. NVRAM Capacity Configuration

NVRAM capacity is determined by flash storage capacity and reliability requirement. The reliability of both flash storage and NVRAM storage should be considered. Assume each stripe contains N data pages on flash memory and one parity page on NVRAM. A smaller N results in higher reliability of flash storage but a larger requirement for NVRAM capacity. Furthermore, a number of *spare* NVRAM dies should be provisioned to tolerate occasional failures of NVRAM dies. Therefore, this is a tradeoff between hardware cost and storage reliability. An NVRAM die is typically multiple times smaller and more expensive than a flash die [36], [42]. Assume NVRAM has C_{nvram} times higher cost (\$/Gb) than flash memory. When NVRAM storage is over-provisioned by a percentage (P_{nvram}) of the capacity of a flash die, the ratio of NVRAM cost to flash cost is $(1 + P_{\text{nvram}}) * C_{\text{nvram}}/N$. For example, when PCM, whose C_{nvram} is about 5.3 [36], is used and N and P_{nvram} are 128 and 25%, respectively, the cost ratio is 5.2%. The SSD reliability can be improved by decreasing N and/or increasing P_{nvram} at an increased cost, and vice versa. To improve storage utilization, spare NVRAM can be used to cache FTL metadata and buffer data [37], [38].

NVRAM storage receives intensive read and write requests, as data writes and GC operations of each tenant lead to parity updates. Nonetheless, NVRAM storage does *not* become a performance bottleneck for three reasons. First, NVRAM has orders of magnitude higher access speed than flash memory. Second, parity pages are distributed across multiple NVRAM dies and can be updated in parallel. More active NVRAM dies should be provisioned if higher performance is required. Third, NVRAM supports byte addressability and in-place updates, so only the changed parts of parity pages need to be updated.

To give an example for analysis, we assume that PCM dies are used to store parity pages. Typical read/write time of PCM is 20 ns/75 ns and the granularity is a cache line (e.g., 64 B) [36]. By contrast, typical read/write and block erase time of flash memory is 50 μs/500 μs (in a granularity of a flash page, e.g., 16 KB) and 5000 μs (in a granularity of a flash block, e.g., 1024 pages), respectively [18]. It takes 25 μs to update (i.e., read and then write) an entire parity page on PCM. Assume a severe case where tenants simultaneously write sub-superpages in different superpages. Eight active PCM dies can support up to 160 ($8 * 500/25$) tenants without being the performance bottleneck. Assume another severe case where tenants simultaneously erase sub-superblocks with different offsets in their dies and each block contains 1024 RAID-protected invalid pages. Eight active PCM dies can support up to 17 ($8 * (50 * 1024 + 5000)/(25 * 1024)$) tenants. This severe case rarely happens, since LiveSSD actively removes invalid flash pages from parity protection in advance and the probability of simultaneous erases is low.

In addition, NVRAM lifetime/wear is *not* concern. NVRAM has more than three orders of magnitude higher write endurance than flash memory [36]. In the worst case, each data page write on flash memory leads to a parity page update on NVRAM. Assume each superpage/stripe contains N flash pages and one NVRAM page, the NVRAM page endures writes of at most N times of flash page writes. Thus, NVRAM can support at least 1000 tenants without concerning the wear.

D. Discussion

RAID Reconstruction: Whenever data corruption occurs, LiveSSD recovers victim data through RAID reconstruction

and then migrates the data to reliable storage. Since the reconstruction needs to read victim stripes across flash dies, I/O interference is introduced to tenants. Regarding a flash page read failure (due to ECC-uncorrectable bit errors), only one stripe is read and the interference is small. After being recovered, the victim data are written to a healthy flash page to avoid future read failures. Regarding a die failure, numerous stripes need to be read and the interference is significant. However, die failures rarely happen [17] and data integrity should be given a higher priority than performance isolation.

Data Consistency: A concern is how to ensure data consistency after a power failure if volatile RAM is used to cache metadata and buffer data. Both hardware and software techniques can be applied. SSDs are usually equipped with capacitors, which provide backup energy [49]. After a power failure, the backup energy can be utilized to complete pending operations of flash writes/erases and NVRAM writes. Hence, the consistency between data and parity is guaranteed. From the software perspective, the FTL flushes dirty data to flash/NVRAM storage periodically and conditionally. To ensure the mapping table consistency, the FTL writes the relevant logical page address to the spare area of each flash page. If the table is corrupted, it can be reconstructed by scanning the spare area of flash pages [50].

V. Experiments

A. Experiment Setup

We evaluate LiveSSD on SSDsim [8], a popular trace-driven simulator whose accuracy has been validated against a real hardware platform. We compare LiveSSD with a hardware-virtualized SSD adopting the RAID-0 scheme, called Optimal-SSD, and a hardware-virtualized SSD adopting the NV-PG-RAID-4 scheme (discussed in Section III), called *NVRAM-SSD*. Optimal-SSD has no parity protection, so it does *not* suffer from any RAID-induced I/O interference and has optimal performance. In NVRAM-SSD, pages with the same offset across flash dies (assigned to multiple tenants) constitute a RAID stripe and parity is stored in NVRAM. We regard NVRAM-SSD as the baseline, where the tenant-level parity updating technique is enabled but invalid flash pages are *not* actively removed from parity protection in advance. Its performance deviation from Optimal-SSD indicates the RAID-induced I/O interference.

To provide deep insights about the active parity updating technique in LiveSSD, we also evaluate two variants of LiveSSD: 1) LiveSSD w/partial utilizes partial page updates of tenants to perform active parity updates but disables the utilization of idle time and 2) LiveSSD w/idle utilizes only the idle time but disables the utilization of partial page updates. In all the schemes, the page-level mapping is used and mapping tables of tenants are entirely cached; the greedy GC policy is used to select the sub-superblock with the least valid data as the victim.

The SSD configurations are shown in Table I. The logical capacity of flash storage is 64 GB and the ratio of over-provisioning space is 12.5%. Data buffer is shared by four tenants equally. The logical-to-physical mapping table is entirely cached to avoid I/O interference caused by address translation. PCM is used as active NVRAM, which is as large as a flash die (spare NVRAM is not involved in the experiments). Sixteen flash dies are evenly partitioned into four vSSDs. Each vSSD is allocated to a tenant, which runs a specific trace. Before running each trace to obtain performance results, we age the vSSD by: 1) sequentially writing 75% of the logical space and 2) writing it with the modified trace, where logical addresses are shifted by a random offset, one or several times until GC operations have been triggered. This aged state makes experimental results more practical.

TABLE I
SSD Configurations [18], [36]

Flash	Logical/Physical capacity	64GB/72GB
	Data buffer size	4MB
	# of channels	4
	# of dies per channel	4
	# of blocks/die	288
	# of pages/block	1024
	Page size	16KB
	Read/Write/Erase time	50μs/500μs/5ms
	# of tenants	4 (each with 4 dies)
PCM	Physical Capacity	4.5GB
	Access granularity	64B
	Read/Write time	20ns/75ns
RAID	Stripe size	16 : 1

We choose six groups of traces with diverse characteristics and each time one group is used to evaluate the schemes. Each group includes four different traces, representing four individual tenants. The RAID-induced I/O interference is caused by reading invalid flash pages in victim blocks during GC operations. Therefore, write-dominant traces are preferred. I/O characteristics of these traces are shown in Table II. The average I/O interval time is calculated excluding the tail beyond 95th percentile (P95).

A group of synthetic traces (SYN) contain only write requests, where percentages of sequential/random writes are 100%/0% (*w_100_seq*), 70%/30% (*w_70_seq*), 30%/70% (*w_30_seq*), and 0%/100% (*w_0_seq*), respectively. Three groups of publicly available traces are included: *MSR–FIU* traces are collected in servers at Microsoft Research Cambridge [51] and Florida International University [52]; VDI traces and *Phone* traces are collected from virtual machines in large-scale virtual desktop infrastructure [53] and smartphone applications [54], respectively. Original VDI traces have very large logical address space, ranging from 3 TB to 5 TB. To fit in the small vSSD in our experiments, we regulate the logical page addresses of I/O requests into 12 GB (75% of the SSD capacity) using the modulus (*mod*) operation. Although VDI traces are collected on slow hard disks, we believe such regulation drastically amplifies the I/O intensity and thus makes the traces more suitable for testing SSDs.

We also collected a group of traces (APP) by ourselves on a 256-GB Intel 545 s SSD running a popular file system benchmark, *filebench* [55], and a popular database benchmark, *YCSB* [56].[3] As *filebench* workloads stress storage systems

[3]Specifically, we run three tests (*fileserver*, *varmail*, and *oltp*) in *filebench* on ext4 file system and one test in *YCSB* on RocksDB for minutes one after another. Meanwhile, the Linux *blktrace* tool [57] is used to obtain I/O traces.

TABLE II
SPECIFICATION OF TRACES [51], [52]

Groups of tenants		# of request	Write ratio	Avg. R/W size	Avg. interval time within P95
SYN	w_100_seq	2,000,000	100%	-/4KB	1ms
	w_70_seq	2,000,000	100%	-/4KB	1ms
	w_30_seq	2,000,000	100%	-/4KB	1ms
	w_0_seq	2,000,000	100%	-/4KB	1ms
MSR-FIU	hm0 (MSR)	2,244,798	66.4%	8.3KB/8.9KB	2.84ms
	online (FIU)	2,313,799	73.9%	4KB/4KB	4.23ms
	webmail (FIU)	3,138,222	82.8%	4KB/4KB	0.58ms
	webusers (FIU)	2,827,077	91%	6.6KB/4KB	15.27ms
VDI	vdi-lun0	1,128,115	37.9%	33KB/18.6KB	1.81ms
	vdi-lun1	986,063	38.2%	31.9KB/21.2KB	1.6ms
	vdi-lun4	1,570,527	31.2%	32KB/27KB	1.28ms
	vdi-lun6	2,039,705	32%	27.4KB/26.6KB	0.97ms
Phone	cameraVideo	9,348	29.5%	38.6KB/180.7KB	15.73ms
	GoogleMap	12,603	86.8%	28.6KB/13.1KB	12.06ms
	Installing	17,952	98.3%	22.2KB/42.5KB	6.61ms
	MusicFaceBook	35,136	87.7%	38.5KB/9.2KB	2.27ms
APP / APP-10xTime	fileserver	2,000,000	100%	-/21.5KB	0.036ms / 0.36ms
	varmail	2,000,000	99.1%	14.7KB/4.3KB	0.01ms / 0.1ms
	oltp	2,000,000	94.8%	4KB/4KB	0.018ms / 0.18ms
	ycsb	2,000,000	3.1%	13.9KB/499.6KB	0.9ms / 9ms

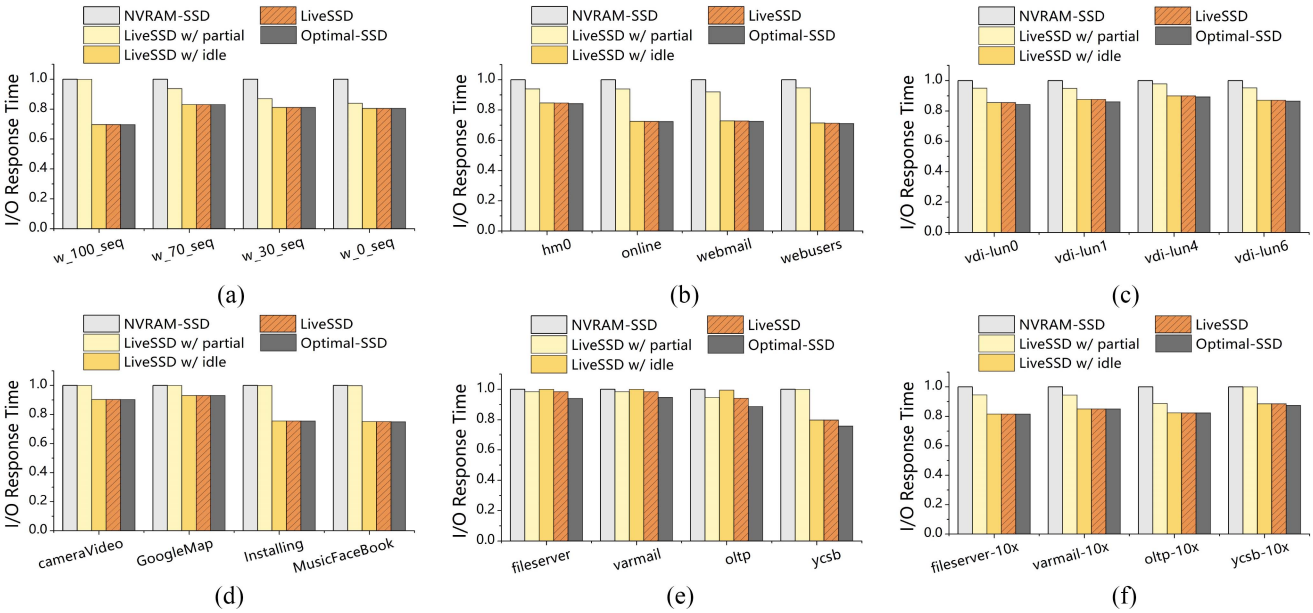


Fig. 6. Average I/O response time normalized to that of NVRAM-SSD. The SSD is evenly shared by four tenants, each of which is represented by a specific trace. LiveSSD achieves similar performance to Optimal-SSD by largely eliminating the RAID-induced I/O interference. (a) Synthetic traces. (b) MSR-FIU traces. (c) VDI traces. (d) Phone traces. (e) APP traces. (f) APP-10xTime traces.

to acquire the maximum performance, APP traces excluding *ycsb* are extremely I/O intensive for a 16-GB vSSD with a small write buffer and only four flash dies. The *ycsb* trace is less I/O intensive because the storage bandwidth is not fully exploited in the YCSB test due to host software overheads. By amplifying the timestamp of APP traces tenfold, we obtain another group of traces with more moderate (but still high) I/O intensity (*APP-10xTime*).

B. Performance Evaluation

We first present experimental results about tenants' I/O performance of the five schemes in Figs. 6 and 7, indicated by the average and tail (99th percentile) response time of I/O requests. Tail latency is an important metric of Quality of Service (QoS) and reducing it is one of the goals of hardware virtualization. Fig. 8 shows the average numbers of invalid flash pages per GC block (all the schemes have the same numbers). Fig. 9 shows the percentages of parity updates during different phases in LiveSSD. Fig. 10 shows the prediction accuracy of idle time of (1). These three figures provide insights into the performance results. Experimental results about SSD write amplification and flash block erases are not presented because our design does *not* have any impacts on them.

1) RAID-Induced I/O Interference: Compared to Optimal-SSD, NVRAM-SSD increases the average I/O response time by an average of 21.9% (see Fig. 6), while an average

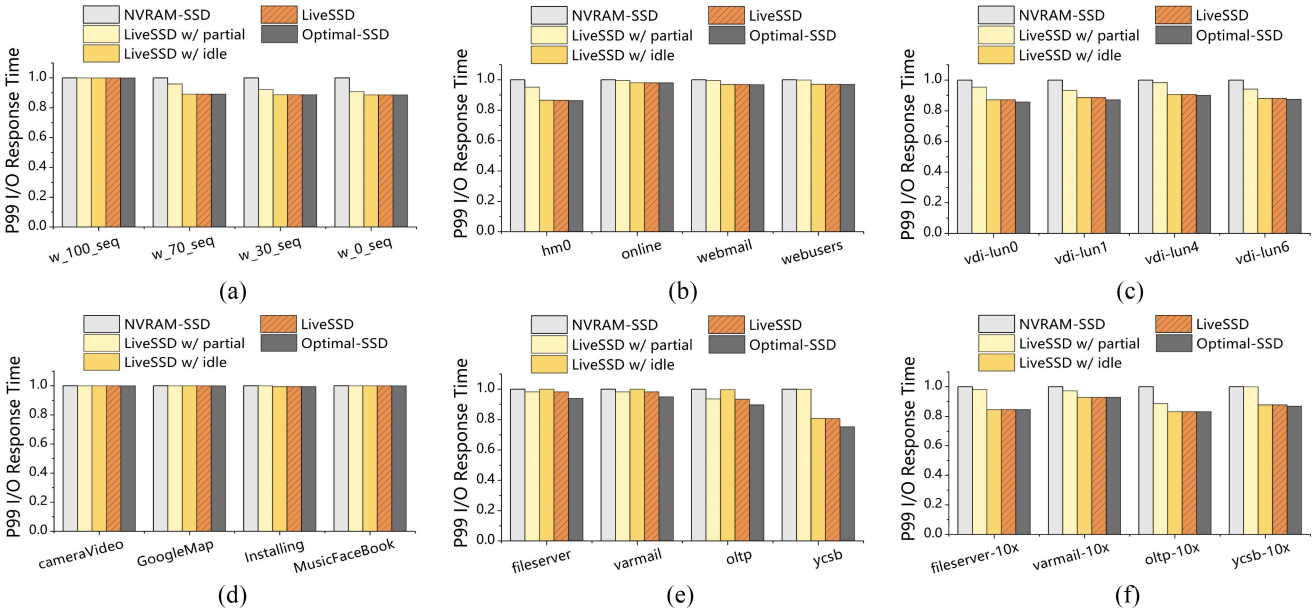


Fig. 7. 99th percentile (P99) I/O response time normalized to that of NVRAM-SSD. The SSD is evenly shared by four tenants, each of which is represented by a specific trace. LiveSSD incurs little increases on tail I/O response time, compared to Optimal-SSD. (a) Synthetic traces. (b) MSR-FIU traces. (c) VDI traces. (d) Phone traces. (e) APP traces. (f) APP-10xTime traces.

of 9.8% in terms of tail I/O response time (see Fig. 7). These performance degradations indicate the RAID-induced I/O interference, caused by extra reads of invalid flash pages in victim blocks during GC. The maximum increase on average I/O response time is 43.7% and occurs in the *w_100_seq* trace, while the maximum increase on tail I/O response time is 32.8% and occurs in the *ycsb* trace. Both traces have strong write sequentiality, so invalid pages gather on a small number of flash blocks. This can be verified in Fig. 8, where 93.5% and 84.4% of flash pages are invalid in GC blocks, respectively. Similarly, NVRAM-SSD suffers from significant (larger than 30%) increases on average I/O response time in *FIU* traces [*online*, *webmail*, and *webusers* in Fig. 6(b)] and some *Phone* traces [*Installing* and *MusicFaceBook* in Fig. 6(d)]. In three SYN traces (*w_70_seq*, *w_30_seq*, and *w_0_seq*), despite they have different percentages of sequential writes, NVRAM-SSD shows similar performance degradations [see Fig. 6(a)]. This is because random writes in these traces, including those in the aging phase before each test, distribute consecutive logical data pages across scattered physical pages. Besides, performance degradations are moderate (e.g., 15.5% on average) in VDI traces, where read requests dominate and write requests do not show strong sequentiality (48.5% of flash pages are invalid in GC blocks, as shown in Fig. 8).

A higher number of invalid pages per GC block does not necessarily lead to a larger performance degradation, because some other factors also matter, such as I/O intensity and the ratio of write requests. Especially, I/O intensity has complicated impacts on the performance, like a double-edged sword. For example, in Fig. 6(d) (*Phone* traces), NVRAM-SSD has more invalid pages per GC block but much smaller performance degradations (e.g., 10.8% versus 33.3%) in *cameraVideo* and *GoogleMap* than in *Installing* and *MusicFaceBook*. This is because RAID-induced I/O

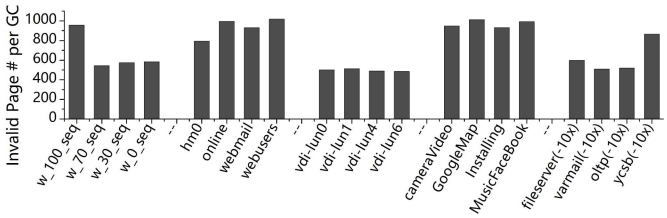


Fig. 8. Average numbers of invalid pages per GC block. Each flash block has 1024 pages. The numbers of invalid pages are the same in the five schemes because they employ the same GC algorithm and parity updates cause no flash writes. A larger number indicates larger RAID-induced I/O interference.

interference causes longer queuing/waiting time in *Installing* and *MusicFaceBook*, which have higher I/O intensity. On the other hand, higher I/O intensity may also result in a smaller performance degradation. As shown in Fig. 6(e) and (f) (APP traces have ten times higher I/O intensity than *APP-10xTime* traces), the average increase of I/O response time is 20.6% in *fileserver-10x*, *varmail-10x*, and *oltp-10x*, while 8.3% in *fileserver*, *varmail*, and *oltp*. The main reason is that, under extremely high I/O intensity, waiting time and thus I/O response time have already been very large. Relatively, the impact of RAID-induced I/O interference becomes small. This also explains why RAID-induced I/O interference has a smaller impact on increasing tail I/O response time than on increasing average I/O response time (Fig. 6 versus Fig. 7).

In summary, these results under various workloads indicate that RAID-induced I/O interference not only increases I/O response time of tenants significantly but also degrades the tail latency QoS. It is necessary to eliminate such I/O interference so that storage reliability can be guaranteed without sacrificing performance isolation in a hardware-virtualized SSD.

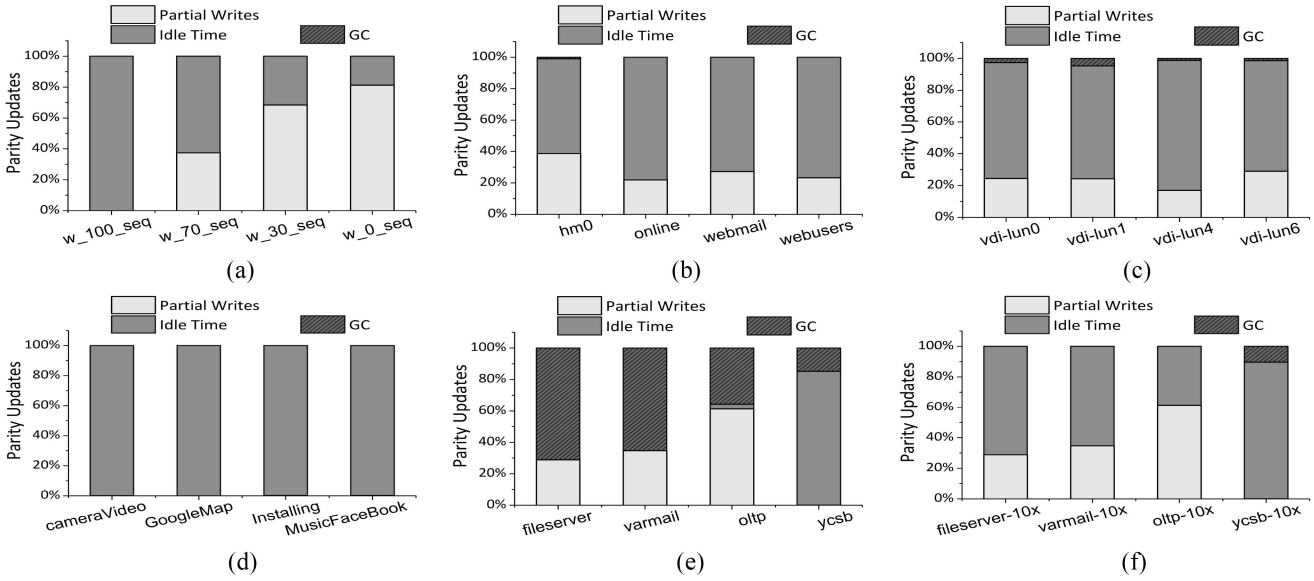


Fig. 9. Parity updates for invalid flash pages during different phases in LiveSSD. Most parity updates can be done during tenants writes (i.e., partial page updates) and idle time, while the rest conducted during GC introduce I/O interference to tenants. (a) Synthetic traces. (b) MSR-FIU traces. (c) VDI traces. (d) Phone traces. (e) APP traces. (f) APP-10xTime traces.

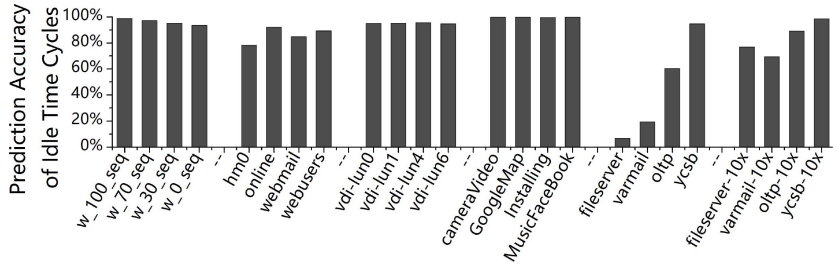


Fig. 10. Prediction accuracy of idle time. Most idle cycles between I/O requests can be predicted and utilized for background parity updates by LiveSSD.

2) Reducing RAID-Induced I/O Interference: Compared to NVRAM-SSD, LiveSSD utilizes partial page updates of tenants and idle time to actively remove invalid flash pages from parity protection without blocking I/O requests. RAID-protected invalid flash pages in GC blocks remain the source of RAID-induced I/O interference. As show in Figs. 6 and 7, LiveSSD reduces the average I/O response time by an average of 16.3% and up to 30.4%, while by an average of 7.6% and up to 19.2% for the tail I/O response time. Compared to Optimal-SSD, LiveSSD has negligible performance degradations in all the traces except APP traces. Specifically, the increases on average and tail I/O response time are only 0.4% and 0.3%, on average, and up to 1.2% and 1.7%, respectively.

APP traces are obtained using stress test tools running on an SSD that has higher performance than the simulated vSSD. Therefore, they except *ycsb* present a worst-case scenario, where I/O requests are very intensive and little idle time can be utilized. In this scenario, RAID-induced I/O interference in LiveSSD increases the average and tail I/O response time by an average of 5% and 4% and up to 6.1% and 4.5%, respectively. The reason why the *ycsb* trace is an exception is that it is read-dominant and less I/O intensive. In a more moderate scenario, e.g., *APP-10xTime* traces, where I/O intensity is 10 times lower than APP traces but is still high (compared to VDI and MSR-FIU traces), LiveSSD is on a par with Optimal-SSD.

LiveSSD w/partial utilizes only partial page updates of tenants to alleviate the RAID-induced I/O interference, so it is effective under small random writes. As shown in Fig. 9, we can see a non-negligible portion of invalid flash pages in GC blocks which are removed from parity protection through partial page updates in traces except those dominated by sequential writes (e.g., *w_100_seq*, *ycsb*, and *Phone* traces). Especially, the portion becomes greater, from 0% to 81.2%, as the percentage of sequential writes decreases, from 100% to 0%, in SYN traces. The portion is as large as 61.4% in the *oltp* trace, since this trace includes extensive small random writes. As a result, LiveSSD w/partial reduces the average I/O response time by an average of 4.5% and up to 16%, compared to NVRAM-SSD (see Fig. 6). The maximum decrease occurs in the *w_0_seq* trace, which contains only small random writes. In APP traces excluding *ycsb*, which include little idle time, LiveSSD w/partial matches LiveSSD.

LiveSSD w/idle utilizes only the idle time to alleviate the RAID-induced I/O interference. From Table II and Fig. 9, we can see I/O idle time commonly exists in real-world workloads, from large-scale storage systems (e.g., VDI traces) to enterprise servers (e.g., *MSR-FIU* traces) and to mobile

devices (e.g., *Phone* traces). In LiveSSD, a large portion (73.6% on average) of invalid flash pages in GC blocks are removed from parity protection during idle time in all the traces excluding *fileserver*, *varmail*, and *oltp*. In LiveSSD w/idle, the average portion rises to 98.3% (not shown in the figures), as invalid flash pages that are removed from parity protection through partial page updates in LiveSSD are included. Such effectiveness also owes to the lightweight and efficient background operations. Specifically, reading flash pages (for parity updates) is fast, so even short idle periods can be utilized; moreover, invalid flash pages that most likely cause RAID-induced I/O interference are chosen as targets in priority. As a consequence, LiveSSD w/idle rivals LiveSSD in most traces but falls behind by an average of 2.9% and up to 5.7% in APP traces except *ycsb*.

In addition, we evaluate the prediction accuracy of idle time of the exponential smoothing method [see (1)] in Fig. 10. When an I/O request is completed and next I/O request does not arrive yet, if the predicted idle time length is longer than a threshold (i.e., 0.05 ms), background parity updates can be triggered by LiveSSD. The prediction accuracy is defined as the ratio between the number of available idle time cycles that have been predicted successfully and the maximum number of available idle time cycles, rather than as the deviation of predicted idle time length to real idle time length. We can see, on average, 84.3% of idle time periods between I/O requests can be predicted and utilized. This verifies the exponential smoothing method is competent for idle time prediction.

To summarize, these results demonstrate that LiveSSD can efficiently exploit partial page updates and idle time, which extensively exist in various workloads, to eliminate the RAID-induced I/O interference to tenants.

VI. Conclusion

Modern SSDs, architected with many parallel flash dies, have been increasingly employed to host data related services in multitenant storage systems. Hardware virtualization and die-level RAID are critical techniques to improve storage utilization and reliability, respectively. This article identifies the RAID-induced I/O interference as an obstacle to integrate these two techniques. To solve this problem, we propose LiveSSD, a hardware-virtualized SSD with a low-interference die-level RAID scheme. High-speed NVRAM is employed as parity storage without being a performance bottleneck. Parity updates are merged for each tenant and are conducted in advance without blocking I/O requests. Despite a higher hardware cost, LiveSSD delivers an effective solution that enhances storage reliability of a hardware-virtualized SSD without violating the goal of performance isolation.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments.

References

[1] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash storage disaggregation," in *Proc. 11th Eur. Conf. Comput. Syst. (EuroSys)*, 2016, pp. 1–15.

[2] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *Proc. 20st IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2014, pp. 524–535.

[3] B. Tallis. (2019). *The Memblaze PBlaze5 C916 Enterprise SSD Review: High Performance and High Capacities*. [Online]. Available: https://www.anandtech.com/show/14070/the-memblaze-pblaze5-c916-ssd-review

[4] N. Zhang, J. Tatemura, J. Patel, and H. Hacigumus, "Re-evaluating designs for multi-tenant OLTP workloads on SSD-based I/O subsystems," in *Proc. ACM SIGMOD Int. Conf. Manag. Data (SIGMOD)*, 2014, pp. 1383–1394.

[5] J. Huang *et al.*, "Flashblox: Achieving both performance isolation and uniform lifetime for virtualized SSDs," in *Proc. 15th USENIX Conf. File Stor. Technol. (FAST)*, 2017, pp. 1–17.

[6] C. Petersen, W. Zhang, and A. Naberezhnov, "Enabling NVMe I/O determinism @ scale," in *Proc. Flash Memory Summit*, 2018, pp. 1–32.

[7] B. S. Kim, H. S. Yang, and S. L. Min, "AutoSSD: An autonomic SSD architecture," in *Proc. USENIX Annu. Techn. Conf. (ATC)*, 2018, pp. 1–14.

[8] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity," in *Proc. ACM Int. Conf. Supercomput. (ICS)*, 2011, pp. 96–107.

[9] J. Kim, D. Lee, and S. H. Noh, "Towards SLO complying SSDs through OPS isolation," in *Proc. 13th USENIX Conf. File Stor. Technol. (FAST)*, 2015, pp. 1–8.

[10] B. Jun and D. Shin, "Workload-aware budget compensation scheduling for NVME solid state drives," in *Proc. IEEE Non-Volatile Memory Syst. Appl. Symp. (NVMSA)*, 2015, pp. 1–6.

[11] S.-M. Huang and L.-P. Chang, "Providing SLO compliance on NVMe SSDs through parallelism reservation," *ACM Trans. Design Autom. Electron. Syst.*, vol. 23, no. 3, pp. 1–26, Feb. 2018.

[12] B. S. Kim, "Utilitarian performance isolation in shared SSDs," in *Proc. 10th USENIX Workshop Hot Topics Stor. File Syst. (HotStorage)*, 2018, pp. 1–16.

[13] L. Parker, "Optimizing SSDs for multiple tenancy use," in *Proc. Flash Memory Summit*, 2018, pp. 1–10.

[14] X. Song, J. Yang, and H. Chen, "Architecting flash-based solid-state drive for high-performance I/O virtualization," *IEEE Comput. Archit. Lett.*, vol. 13, no. 2, pp. 61–64, Jul.–Dec. 2014.

[15] D.-W. Chang, H.-H. Chen, and W.-J. Su, "VSSD: Performance isolation in a solid-state drive," *ACM Trans. Design Autom. Electron. Syst.*, vol. 20, no. 4, pp. 1–33, Sep. 2015.

[16] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error characterization, mitigation, and recovery in flash-memory-based solid-state drives," *Proc. IEEE*, vol. 105, no. 9, pp. 1666–1704, Sep. 2017.

[17] N. R. Mielke, R. E. Frickey, I. Kalastirsky, M. Quan, D. Ustinov, and V. J. Vasudevan, "Reliability of solid-state drives based on NAND flash memory," *Proc. IEEE*, vol. 105, no. 9, pp. 1725–1750, Sep. 2017.

[18] B. S. Kim, J. Choi, and S. L. Min, "Design tradeoffs for SSD reliability," in *Proc. 17th USENIX Conf. File Stor. Technol. (FAST)*, 2019, pp. 1–15.

[19] S. Shadley. (2011). *NAND Flash Media Management Through RAIN*. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/technical-marketing-brief/brief_ssd_rain.pdf

[20] Y. Lee, S. Jung, and Y. H. Song, "FRA: A flash-aware redundancy array of flash storage devices," in *Proc. 7th IEEE/ACM Int. Conf. Hardw. Softw. Codesign Syst. Synth. (CODES+ISSS)*, 2009, pp. 163–172.

[21] S. Im, D. Shin, D. Shin, D. Shin, and D. Shin, "Flash-aware RAID techniques for dependable and high-performance flash memory SSD," *IEEE Trans. Comput.*, vol. 60, no. 1, pp. 80–92, Jan. 2011.

[22] J. Guo, W. Wen, Y. Z. Li, S. Li, H. Li, and Y. Chen, "DA-RAID-5: A disturb aware data protection technique for NAND flash storage systems," in *Proc. Design Autom. Test Eur. Conf. Exhibit. (DATE)*, 2013, pp. 380–385.

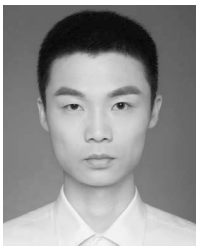
[23] Y. Wang, W. Wang, T. Xie, W. Pan, Y. Gao, and Y. Ouyang, "CR5M: A mirroring-powered channel-RAID5 architecture for an SSD," in *Proc. 30th Symp. Mass Stor. Syst. Technol. (MSST)*, 2014, pp. 1–28.

[24] S. Yan *et al.*, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs," in *Proc. 15th USENIX Conf. File Stor. Technol. (FAST)*, 2017, pp. 1–15.

[25] M.-C. Yang, Y.-H. Chang, and T.-W. Kuo, "Virtual flash chips: Rethinking the layer design of flash devices to improve data recoverability," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, 2015, pp. 1–6.

[26] J. Kim, E. Lee, J. Choi, D. Lee, and S. H. Noh, "Chip-level RAID with flexible stripe size and parity placement for enhanced SSD reliability," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1116–1130, Apr. 2016.

[27] S. Wang, F. Wu, Z. Lu, J. Zhou, and C. Xie, "WARD: Wear aware RAID design within SSDs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2918–2928, Nov. 2018.
[28] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Comput. Surveys*, vol. 37, no. 2, pp. 138–163, 2005.
[29] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. 14th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2009, pp. 1–50.
[30] A. S. Spinelli, C. M. Compagnoni, and A. L. Lacaita, "Reliability of NAND flash memories: Planar cells and emerging issues in 3D devices," *Computers*, vol. 6, no. 2, pp. 1–55, 2017.
[31] K. Zhao, W. Zhao, H. Sun, X. Zhang, N. Zheng, and T. Zhang, "LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives," in *Proc. 11th USENIX Conf File Stor. Technol. (FAST)*, 2013, pp. 1–30.
[32] Q. Xiong *et al.*, "Characterizing 3D floating gate NAND flash: Observations, analyses, and implications," *ACM Trans. Stor.*, vol. 14, no. 2, pp. 1–31, 2018.
[33] F. Wu *et al.*, "Characterizing 3D charge trap NAND flash: Observations, analyses and applications," in *Proc. 36th IEEE Int. Conf. Computer Design (ICCD)*, 2018, pp. 381–388.
[34] R. Birke, M. Bjoerkqvist, L. Y. Chen, E. Smirni, and T. Engbersen, "(Big)data in a virtualized world: Volume, velocity, and variety in cloud datacenters," in *Proc. 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014, pp. 177–189.
[35] B. Peng, H. Zhang, J. Yao, Y. Dong, Y. Xu, and H. Guan, "MDev-NVMe: A NVMe storage virtualization solution with mediated pass-through," in *Proc. USENIX Annu. Techn. Conf. (ATC)*, 2018, pp. 1–13.
[36] M. Oros, "Analysts weigh in on persistent memory," in *Proc. Persistent Memory Summit*, 2018, pp. 1–28.
[37] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1537–1550, May 2016.
[38] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "Improving SSD lifetime with byte-addressable metadata," in *Proc. Int. Symp. Memory Syst. (MEMSYS)*, 2017, pp. 374–384.
[39] Intel. (2019). *Intel Optane Memory H10 With Solid State Storage*. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-memory/optane-memory-h10.html
[40] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. USENIX Conf. File Stor. Technol. (FAST)*, 2015, pp. 1–15.
[41] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in SSD-conscious storage," in *Proc. 14th USENIX Conference on File and Storage Technologies (FAST'16)*, 2016, pp. 1–17.
[42] K. Suzuki and S. Swanson, "A survey of trends in non-volatile memory technologies," in *IEEE International Memory Workshop (IMW)*, 2015, pp. 2000–2014.
[43] *Open NAND Flash Interface (ONFI)*. Accessed: Jun. 11, 2020. [Online]. Available: http://www.onfi.org/
[44] Y. Y. Tai, "High Performance FTL for PCIe/NVMe SSDs," in *Proc. Flash Memory Summit*, 2016, pp. 1–20.
[45] H. Lv *et al.*, "Exploiting minipage-level mapping to improve write efficiency of NAND flash," in *Proc. IEEE Int. Conf. Netw. Archit. Stor. (NAS)*, 2018, pp. 1–10.
[46] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, 2008, pp. 119–128.
[47] S. Lee and J. Kim, "Improving performance and capacity of flash storage devices by exploiting heterogeneity of MLC flash memory," *IEEE Trans. Comput.*, vol. 63, no. 10, pp. 2445–2458, Oct. 2014.
[48] Q. Wei, B. Gong, S. Pathak, B. Veeravalli, L. Zeng, and K. Okada, "WAFTL: A workload adaptive flash translation layer with data partition," in *Proc. 27th Symp. Mass Stor. Syst. Technol. (MSST)*, 2011, pp. 1–12.
[49] Samsung Electronics Co. (2014). *Power Loss Protection (PLP): Protect Your Data Against Sudden Power Loss*. [Online]. Available: http://insignis-tech.com/wp-content/uploads/2017/05/NS_PLPv0.7_20170516.pdf
[50] D. Ma, J. Feng, and G. Li, "LazyFTL: A page-level flash translation layer optimized for NAND flash memory," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2011, pp. 1–12.
[51] (2007). *MSR Cambridge Traces*. [Online]. Available: http://iotta.snia.org/traces/388
[52] (2010). *FIU Traces*. [Online]. Available: http://iotta.snia.org/traces/390
[53] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara, "Understanding storage traffic characteristics on enterprise virtual desktop infrastructure," in *Proc. 10th ACM Int. Syst. Stor. Conf. (SYSTOR)*, 2017, pp. 1–11.
[54] D. Zhou, W. Pan, W. Wang, and T. Xie, "I/O characteristics of smartphone applications and their implications for EMMC design," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, 2015, pp. 12–21.
[55] *Filebench Benchmark*. Accessed: Apr. 2, 2020. [Online]. Available: https://github.com/filebench/filebench/wiki
[56] *YCSB Benchmark*. Accessed: Apr. 2, 2020. [Online]. Available: https://github.com/brianfrankcooper/YCSB
[57] *Blktrace: Generate Traces of the I/O Traffic on Block Devices*. Accessed: Apr. 2, 2020. [Online]. Available: https://linux.die.net/man/8/blktrace



You Zhou received the B.E. degree in computer science and technology and the Ph.D. degree in computer architecture from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2011 and 2017, respectively.

He is currently a Postdoctoral Researcher with the Wuhan National Laboratory for Optoelectronics, HUST. His research interests include flash storage systems, emerging storage architectures, and file systems.

Fei Wu (Member, IEEE) received the B.E. and M.E. degrees in electrical automation, control theory, and control engineering from Wuhan Industrial University, Wuhan, China, in 1997 and 2000, respectively, and the Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, in 2005.

She is currently a Professor with the Wuhan National Laboratory for Optoelectronics, HUST. Her research interests include computer architecture and nonvolatile memory.

Weizhou Huang received the B.E. degree in communication engineering from the Wuhan University of Technology, Wuhan, China, in 2019. He is currently pursuing the Ph.D. degree with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, under the supervision of Prof. F. Wu.

His research interests include flash storage and nonvolatile memory systems.

Changsheng Xie (Member, IEEE) received the B.E. and M.E. degrees in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1982 and 1988, respectively.

He is currently a Professor with the Wuhan National Laboratory for Optoelectronics, HUST. His research interests include computer architecture and emerging storage technologies and systems.