Understanding and Exploiting the Full Potential of SSD Address Remapping

Qiulin Wu, You Zhou, Fei Wu, *Member, IEEE*, Hong Jiang, *Fellow, IEEE*, Jian Zhou, and Changsheng Xie, *Member, IEEE*

***Abstract*—Duplicate writes are prevalent in storage systems, originating from data duplication, journaling, and data relocations, etc. As flash-based solid state drives (SSDs) have been widely deployed, duplicate writes can significantly degrade their performance and lifetime. Prior studies have proposed innovative approaches that exploit the *address remapping* utility inside an SSD to eliminate duplicate writes. However, remap operations modify the logical-to-physical (L2P) address mapping table while the physical-to-logical (P2L) mappings persisted on flash memory remain unchanged. Such inconsistency between L2P and P2L mappings may cause data corruption and has long been a major obstacle to utilize SSD address remapping. In this article, we propose a novel SSD design, called *Remap-SSD-LH*, that realizes the full potential of SSD address remapping. It provides a *remap* primitive, which allows the host software and SSD firmware to perform logical writes of duplicate data at almost zero cost. To ensure mapping consistency as well as fast mapping lookups, Remap-SSD-LH employs a local log scheme based on hybrid storage. A local log is maintained for each flash garbage collection unit to record relevant P2L mapping changes induced by remap operations. The logs are stored in small nonvolatile RAM (NVRAM), e.g., capacitor-protected DRAM, and can be destaged to flash memory if NVRAM is full. We verify Remap-SSD-LH on a software SSD emulator with three case studies: 1) intra-SSD deduplication; 2) SQLite journaling; and 3) F2FS cleaning. The experimental results show that Remap-SSD-LH can maximally and efficiently exploit address remapping to improve SSD performance and lifetime.**

***Index Terms*—Address remapping, data consistency, duplicate writes, flash memory, flash translation layer (FTL), hardware/software co-design, solid state drive (SSD).**

Manuscript received 14 July 2021; revised 28 October 2021; accepted 6 January 2022. Date of publication 19 January 2022; date of current version 24 October 2022. This work was supported in part by the NSFC under Grant 61902137, Grant U2001203, Grant 61872413, and Grant 61821003; in part by the Fundamental Research Funds for the Central Universities, HUST under Grant 2021XXJS108; in part by the 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 Laboratory of Information Storage System, Ministry of Education of China. This article was recommended by Associate Editor C.-L. Yang. *(Corresponding author: You Zhou.)*

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

You Zhou is with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: zhouyou2@hust.edu.cn).

Hong Jiang is with the Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019 USA (e-mail: hong.jiang@uta.edu).

Digital Object Identifier 10.1109/TCAD.2022.3144617

I. Introduction

DUPLICATE writes are pervasive in real-world storage systems. Not only data duplication is common [1]–[4] but also a broad spectrum of system software and applications introduces duplicate writes. For example, many databases and file systems employ double-write journaling to guarantee write atomicity [5]–[7]; data relocations are required for space cleaning in log-structured/copy-on-write systems [6], [8] and for file defragmentation [9]; file copy and snapshotting operations are common behaviors [10], [11].

On the other hand, NAND flash-based solid state drives (SSDs) have been widely employed in various storage systems. Due to the idiosyncrasies of flash memory, the SSD-internal firmware, called flash translation layer (FTL), performs out-of-place updates. Logical pages written from the host are always mapped to new free flash pages, while obsolete flash pages are invalidated. Thus, a logical-to-physical (L2P) mapping table is maintained to translate logical page numbers (LPNs) to physical page numbers (PPNs) [12], [13]. For fast lookups, this table is typically cached in SSD-internal DRAM. The FTL also conducts garbage collection (GC) periodically to reclaim invalid pages in the granularity of flash blocks, where valid pages are relocated and then the blocks are erased. Note that writes are harmful to both the performance and lifetime of SSDs [14], [15]. This situation deteriorates, as flash technologies are scaling rapidly to increase the bit density but at the cost of degraded write speed and endurance [16].

To eliminate duplicate writes on flash memory, innovative approaches have been proposed to exploit the SSD *address remapping* functionality [1], [6], [7], [9], [10], [17]–[20]. By directly modifying the L2P mapping table, duplicate writes of repeating data pages (including moving and copying data) can be completed quickly without conducting physical writes. Also, data transfers between the host and SSD can be avoided. Although enabling such remapping requires minor modifications to the host software and SSD interface, the benefits are quite worthwhile. The performance, lifetime, and space utilization of an SSD can be improved significantly.

However, remap operations lead to a critical *mapping inconsistency* problem, which may cause data corruption. Whenever a logical data page is written to a flash page, the FTL stores *house-keeping metadata* including the relevant LPN either in the out-of-band (OOB) area of the same flash page [12], [21] or in another reserved flash page [22]. These persistent physical-to-logical (P2L) mappings are indispensable for

completing data relocations during GC and for recovering L2P mappings after sudden power failures (see Section II). Remap operations change the L2P mappings, but the relevant P2L mappings on flash memory cannot be updated accordingly. Due to such mapping inconsistency, wrong L2P mappings would be modified after data relocations during GC or be restored during power-off recovery, compromising data consistency.

This mapping inconsistency problem, although crucial, has *not* been properly addressed in prior studies. The common solution in [1], [6], [9], [18], and [20] is to persist new P2L mappings generated by remap operations in a global log on flash memory. Its main drawback is that the log size would increase continuously over time, incurring prohibitively high lookup overheads at last. Although limiting the log size could confine the lookup overheads, it would also restrict the usage of SSD address remapping. In addition, some other solutions have been proposed but only fit in very limited application scenarios of address remapping [7], [19]. These solutions and their drawbacks are discussed thoroughly in Section III-C.

In this article, we propose a novel SSD design, called *Remap-SSD-LH*, to maximally and efficiently exploit SSD address remapping for reducing duplicate writes. It provides a remap primitive, which allows the host software and FTL to conduct logical writes of duplicate data at almost zero cost. Notice that P2L mappings of a flash GC unit are always looked up together. Remap-SSD-LH maintains a local log for each flash GC unit on hybrid storage. The logs record P2L mapping changes induced by remap operations and are stored in small nonvolatile RAM (NVRAM), e.g., capacitor-protected DRAM. If NVRAM is full, log entries can be destaged to flash memory. This local log scheme not only ensures persistent P2L mappings are always consistent with the latest L2P mappings but also enables fast lookups of P2L mappings during GC. We verify Remap-SSD-LH on FEMU (a software SSD emulator [23]) with three case studies: 1) intra-SSD deduplication; 2) SQLite journaling; and 3) F2FS cleaning. The experimental results show that Remap-SSD-LH can realize the full potential of address remapping for improving SSD performance and lifetime.

II. Background

Mappings in Flash-Based SSDs: Modern SSDs generally employ a page-level FTL, powered by embedded processors and DRAM, for high performance [12], [24]. As a host logical page can be dynamically mapped to any flash page, an *L2P mapping* table is maintained for address translation. Assuming the page size is 4 kB and each mapping entry takes 4B, the table size is 0.1% of the device capacity. The table is persisted on flash memory and cached in DRAM for fast lookups, which locate on the critical path of I/O processing.

When a logical page is written to a flash page, the FTL transparently persists the reverse *P2L mapping* (i.e., the LPN) and *write timestamp* as house-keeping metadata on flash memory for two reasons. First, data pages are periodically migrated on flash memory for GC and wear leveling purposes. P2L mappings need to be retrieved to locate and modify the relevant L2P mappings after the migrations. Second, the mapping consistency needs to be guaranteed. The latest L2P mappings in DRAM may get lost after sudden power failures [13]. By scanning the persistent metadata, the FTL can obtain all the PPN-LPN entries and write order of PPNs, from which the latest L2P mappings can be restored.

Flash Management: SSDs are architected with a number of channels connecting many flash dies, each of which is a parallel unit for accesses [25]. It has been a common practice, especially for high-performance SSDs, to organize flash storage in *superblocks* [15], [22], [24], [26]. A superblock consists of flash blocks with the same offset across multiple dies. Both space allocations for data writes and GC are performed in the unit of a superblock. This has several advantages. First, the intra-SSD parallelism can be maximized. Second, flash management is simplified due to a large granularity. Third, it facilitates die-level RAID, as parity can be easily added in each superblock [15], [16], [27]. Finally, the FTL can accelerate the recovery speed of L2P mappings by storing house-keeping metadata of each superblock collectively in its tail flash pages [22]. Then, only a small amount of tail flash pages need to be scanned, rather than all the flash pages.

Nonvolatile RAM: NVRAM technologies (e.g., PCRAM) are advancing and have received much attention [28]. Compared to flash memory, they offer lower latency and byte-addressability but lower bit density and higher cost. NVRAM complements flash memory well and has opened up new opportunities to enhance SSDs for various purposes [19], [29].

III. Motivation

We illustrate the ubiquity of duplicate writes with several examples in Section III-A, and then detail where and how prior studies leverage SSD address remapping for reducing duplicate writes in Section III-B and their drawbacks in ensuring the mapping consistency in Section III-C.

A. Duplicate Writes

Data Duplication: One major source of duplicate writes is *data duplication* [2]–[4]. For instance, in the disk images of some departmental working environments [1] and file system images collected from smartphones [4], the data duplication rate is 8%–86% and an average of 33%, respectively, while duplicate writes account for 6%–28% and 22%–48% of total writes; in the three production systems at FIU, the ratios of duplicate writes range from 33% to 92% [18].

Journaling: To guarantee write atomicity, journaling approaches have been widely used in databases (e.g., MySQL and SQLite) and file systems (e.g., ext4 and XFS) [6], [7]. Either before-images (e.g., *rollback journaling*) or after-images [e.g., write-ahead logging (WAL)] of updated pages are written in a dedicated log, after which updates are applied to original locations in place. Such journaling introduces double writes of data, for example, causing a worst case slowdown of about 73% in ext4 compared to no journaling [5].

Data Relocation: Copy-on-write and log-structuring mechanisms are popular means to provide write atomicity and write sequentiality (e.g., in Couchbase and F2FS) [6], [8].

TABLE I
PRIOR STUDIES EXPLOITING SSD ADDRESS REMAPPING

Name	Applications of remapping	Schemes for mapping consistency guarantee	Major drawbacks
JFTL [17]	Write-ahead logging (WAL)	None	N/A
ANViL [10]	Snapshots, data deduplication, WAL		
CAFTL [1], CA-SSD [18]	Intra-SSD data deduplication	Maintain a dedicated log on flash memory to record P2L mappings changed by address remapping	High lookup overheads of P2L mappings during GC, poor scalability
Janusd [9]	File system defragmentation		
Copyless copy [20]	WAL, intra-SSD data deduplication		
SHARE [6]	WAL, compaction, tree wandering in copy-on-write databases		
PebbleSSD [19]	Cleaning in log-structured file systems	Replace (fixed-size) flash OOB with byte-addressable NVRAM	Only apply in *P*-type remapping scenarios
WAL-SSD [7]	WAL	Write the predetermined LPN for future remapping to flash OOB	Only apply in *PD*-type remapping scenarios

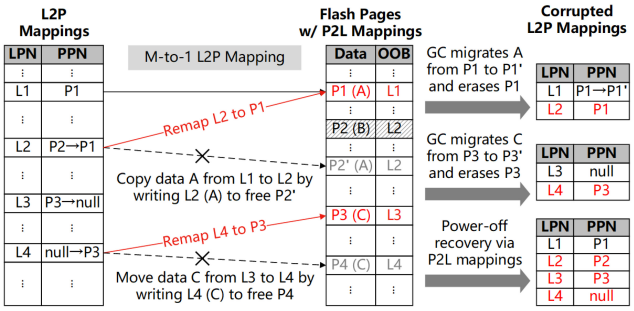


Fig. 1. Examples of SSD address remapping. Duplicate writes to LPNs *L2* and *L4* can be performed through remapping without flash writes. However, L2P and P2L mappings become inconsistent, causing data corruption.

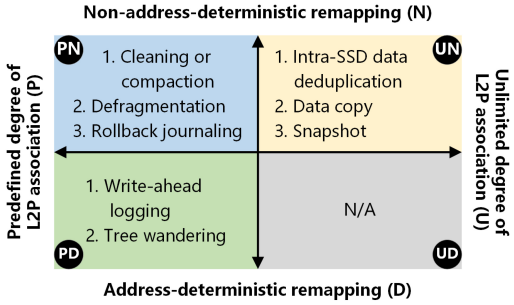


Fig. 2. Applications of SSD address remapping. They can be classified according to characteristics of remapping. The *UD* type is not applicable because the *U* type and *D* type contradict with each other.

They conduct out-of-place updates, so periodical *cleaning* or *compaction* operations are required to reclaim storage space occupied by stale data. In addition, file fragmentation has been a long-standing problem that degrades the performance of file systems. Many file systems recommend periodical *defragmentation* [9]. Both cleaning/compaction and defragmentation cause data relocations and thus duplicate writes.

Data Copy and Snapshot: Data copy is a frequent behavior of users and applications. *Snapshotting*, which provides point-in-time states of data volumes, is an important feature and a common routine in storage systems [30]. These operations may introduce duplicate writes to create physical data copies.

B. Exploiting SSD Address Remapping

To eliminate duplicate writes, the SSD address remapping functionality can be utilized. Assume LPN *Ly* is written with a duplicate data page copied or moved from LPN *Lx*. The FTL can realize the write by remapping *Ly* to the flash page storing *Lx*, rather than by writing a new free flash page. Such remap operations, as shown in Fig. 1, can be done quickly by updating the relevant L2P mappings in SSD-internal DRAM.

Many prior studies have proposed to exploit SSD address remapping in a spectrum of scenarios, as summarized in Fig. 2 and Table I. A body of works applies data deduplication on/inside SSDs [1], [18], [31], [32]. The deduplication engine identifies duplicate data pages written from the host (through hashing fingerprints). Instead of writing them to flash memory, they can be remapped to existing flash pages that store the same contents. Address remapping is also attractive for reducing journaling overheads [6], [7], [10], [17], [20]. After data updates are written to the log, they can be applied by remapping LPNs of original locations to the flash pages storing the logged updates. Using remapping for snapshotting [10] is straightforward, like copying data *A* in Fig. 1. Data relocations for cleaning [19], compaction [6], and defragmentation [9] can be accomplished similarly to moving data *C* in Fig. 1.

However, address remapping causes a critical *mapping inconsistency problem*. Remap operations modify the L2P mappings, but the relevant P2L mappings on flash memory cannot be updated accordingly (because flash memory does *not* support in-place updates). Such inconsistency between L2P and P2L mappings would finally cause data corruption, since L2P mappings would be altered incorrectly during GC or be rebuilt falsely during power-off recovery. For example, in Fig. 1, after remapping LPN *L2* (previously mapped to PPN *P2*) to PPN *P1* (already referenced by *L1*), the L2P and P2L mappings of *P1* become inconsistent ($\{L1, L2\} \rightarrow P1$ versus $P1 \rightarrow L1$). Then, after a GC operation migrates the data page on PPN *P1* to $P1'$ and erases *P1*, *L2* would still be mapped to *P1* wrongly. Consider another scenario where L2P mappings need to be restored after a sudden power outage. An improper L2P mapping, i.e., $L2 \rightarrow P2$, would be recovered from the P2L mapping, i.e., $P2 \rightarrow L2$, persisted on flash memory.

Although several schemes have been proposed in existing studies to cope with the mapping inconsistency, they suffer from severe drawbacks. To facilitate in-depth analysis of the

drawbacks in Section III-C, we classify the applications of remapping in two dimensions. Note that remap operations change the L2P mapping regularity from conventional 1-to-1 to *M-to-1*. In the first dimension, a remapping scenario is considered as *P*-type, if the maximum *M*, namely, *degree of L2P association*, is predefined. Otherwise, it is *U*-type. For example, data relocation and journaling are P-type (*M* is equal to 1 and 2, respectively), while deduplication and file copy are U-type (*M* depends on content popularity and user behaviors, respectively). In the second dimension, a remapping scenario is *D*-type, if the LPNs and PPNs for future remapping are deterministic at the time of the PPNs being written. Otherwise, it is *N*-type. For instance, in WAL (*D*-type), when data pages being updated are written to the log, the LPNs of their original locations are already known.

Combining the two dimensions (P/U-type and D/N-type), application scenarios of SSD address remapping are divided into three types (*PD*, *PN*, and *UN*), as shown in Fig. 2.

C. Schemes for Mapping Consistency

To address the mapping inconsistency problem caused by remapping, several schemes have been proposed, as listed in Table I. Taking all types of remapping scenarios into consideration, the common scheme adopted in [1], [6], [9], [18], and [20], referred to as *Remap-SSD-GF* in Section V, is to maintain a global log on flash memory for persisting the P2L mappings changed by remapping. Its major drawback is that it requires scanning the entire log to retrieve certain P2L mappings during every GC operation and power-off recovery. Especially, the log size increases continuously and could grow very large as remap operations occur. Assume the SSD capacity is 4 TB, page size is 4 kB, and each log entry for a page remap operation takes at least 12B (e.g., 4B PPN + 4B LPN + 4B timestamp). When 5% or 20% of data pages have been remapped, the log size is as large as 600 MB or 2.4 GB, respectively. Hence, the lookup overheads of P2L mappings would increase over time and finally become exceedingly high. It would *not* be an effective solution to store the global log on high-speed NVRAM (denoted as *Remap-SSD-GN* in Section V), since the scanning process would still be very time consuming, e.g., from tens of milliseconds to seconds when the log size is hundreds of megabytes.

To confine the lookup overheads, Janusd [9] sets a limit on the log size and reclaims obsolete mapping entries periodically. However, remap operations have to be disabled when the log reaches the size limit. Additionally, high reclamation overheads are introduced, i.e., reading and rewriting the entire log on flash memory.

PebbleSSD [19] proposes an NVRAM-enhanced scheme, which replaces the fixed-size OOB area in flash pages with byte-addressable NVRAM. Therefore, P2L mappings of remapped data pages can be updated in place in the NVRAM OOB, retaining consistent with the L2P mappings. However, due to the limited OOB size, this scheme only fits in *P*-type remapping scenarios, where the maximum degree of L2P association is limited and small. For *UN*-type remapping, where the degree of L2P association may be high, large and high-cost NVRAM OOB area would be required. Also, NVRAM space utilization would be low for flash pages with low degrees of L2P association.

By utilizing the property of *PD*-type remapping, WAL-SSD [7] writes the predetermined LPN for future remapping to the OOB area when the relevant flash page is written. Thus, the L2P and P2L mappings of the flash page are consistent after the predefined remap operation. This scheme is only applicable for *PD*-type remapping scenarios, because the LPNs for future remapping are totally uncertain in *N*-type scenarios.

In summary, existing SSD designs that exploit address remapping restrict the application scenarios and/or usage frequency of remapping severely, mainly due to the L2P and P2L mapping inconsistency problem. Furthermore, simply enhancing the SSD with extra NVRAM is inadequate to remove the restrictions. As a consequence, the potential of SSD address remapping is largely underutilized.

IV. Design

A. Overview of Remap-SSD-LH

In this section, we present a novel SSD design, called Remap-SSD-LH. The goal is to maximize the utilization of address remapping in all application scenarios and meanwhile, maintain the L2P and P2L mapping consistency efficiently.

Remap-SSD-LH provides a *remap* primitive at the firmware/FTL level, which embodies the address remapping utility, as shown in Fig. 3. The primitive is exposed to the host software as a vendor specific command, which is supported inherently in current interface techniques (e.g., NVMe and SATA). Through the primitive, applications and file systems can copy or relocate data pages without performing flash writes. Also, the FTL, if an intra-SSD deduplication engine is employed, can eliminate writes of duplicate data.

The remap primitive is formatted as `remap (tgtLPN, srcLPN, length, remapFlag)` (*tgt/src*: target/source). It remaps a range of LPNs between `tgtLPN` and `tgtLPN + length - 1` to the flash pages currently mapped to the range of LPNs between `srcLPN` and `srcLPN + length - 1`. The `remapFlag` parameter is a 1-bit flag indicating whether the source LPNs should be deallocated/invalidated or not after remapping. For data relocations, the corresponding flash pages should no longer be mapped to source LPNs (`remapFlag = 1`). Regarding data copies, the L2P mappings of source LPNs are retained (`remapFlag = 0`). With the `remapFlag`, our proposed remap primitive has better expressivity and performance than traditional remap-like primitives (e.g., SHARE [6] and copyless copy (CC) [20]), which enable only data copies. After invoking a traditional primitive for data relocations, the host software needs to explicitly deallocate the source LPNs by issuing TRIM commands.

During remap operations, both remapping and invalidation of LPNs are realized by directly modifying the L2P mapping table cached in SSD-internal DRAM. Recall that when a data page is written to a flash page, the FTL stores the relevant P2L mapping and write timestamp as house-keeping metadata on flash memory. To address the mapping inconsistency and support fast lookups of P2L mappings (as discussed in

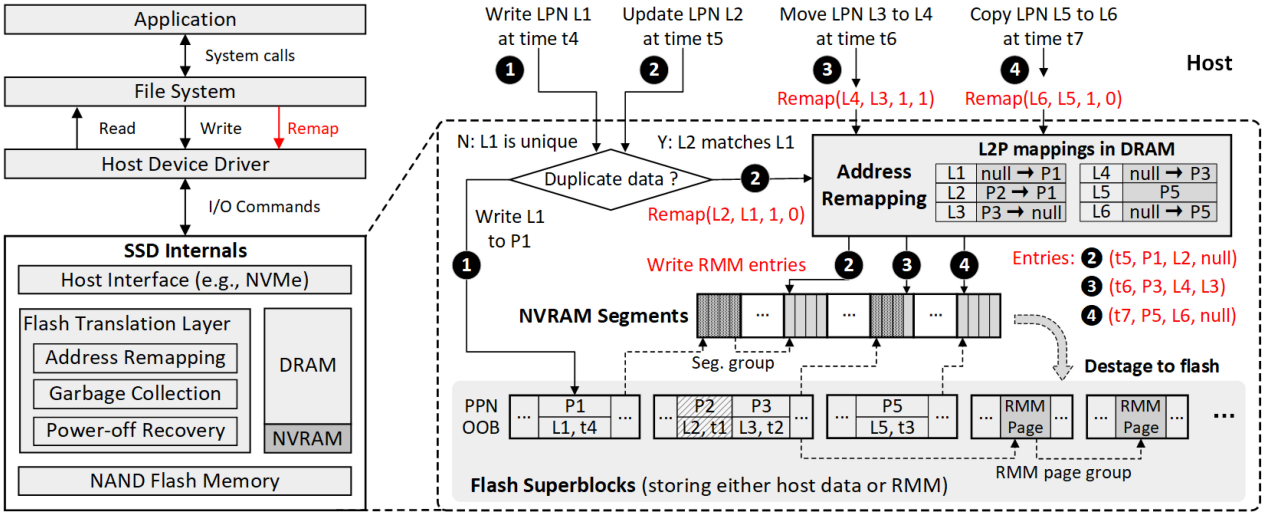


Fig. 3. Overview of Remap-SSD-LH. A `remap` primitive is provided and can be invoked by host software (❸ and ❹) or the FTL (e.g., an intra-SSD deduplication engine ❷). To guarantee the L2P and P2L mapping consistency, RMM entries are persisted in NVRAM segments and can be destaged to reserved flash pages when NVRAM is full.

Section III), Remap-SSD-LH employs a local log scheme that persists additional house-keeping metadata, called remapping metadata (RMM). Whenever an LPN is remapped to a flash page, an RMM entry including the changed P2L mapping is generated. A remap command is considered to be completed successfully only after the involved L2P mappings have been modified in DRAM and the relevant RMM entries have been persisted. Modifications of L2P mappings are *not* required to be persisted because they can be recovered from house-keeping metadata (see Section IV-E). Thus, remap operations can be carried out quickly and persistent P2L mappings are always consistent with the latest L2P mappings.

We introduce the details of the local log scheme and RMM content in Sections IV-B and IV-C, respectively. Sections IV-D and IV-E present how Remap-SSD-LH performs GC operations and power-off recovery, respectively.

B. Local Logging on Hybrid Storage

Naively appending RMM entries in a global log would incur expensive log scans for querying P2L mappings (see Section III-C). To address this challenge, Remap-SSD-LH takes advantage of a key observation that *retrievals of P2L mappings are always performed in the granularity of a flash GC unit*. We assume Remap-SSD-LH adopts the popular superblock-based flash management [24], [26]. The flash GC unit is a superblock containing flash blocks with the same offset across all flash dies. In each GC operation, the FTL selects a victim flash superblock, where valid data pages are read out and written to a free flash superblock. Before the migrations, valid P2L mappings of the victim superblock need to be retrieved so that the involved L2P mappings can be updated to point to new physical locations. After the migrations, the victim superblock can be erased and become free.

Based on the observation, Remap-SSD-LH maintains a local log to record RMM entries for each *data superblock*, i.e., a flash superblock storing host data. Whenever an LPN is remapped to a flash page, the corresponding RMM entry is appended in the local log of the superblock where the flash page resides. The size of a local log varies according to the number of LPNs remapped to the relevant superblock. Instead of a large global log, only a small local log needs to be scanned for lookups of P2L mappings during the GC of a data superblock. The local logs are persisted first in small byte-addressable NVRAM and can also be stored on flash memory if NVRAM capacity is short. The NVRAM media can be one of emerging NVRAM technologies like PCRAM (if available in the SSD) or a portion of DRAM protected by supercapacitors (which commonly exists in modern SSDs).

The NVRAM volume is divided into fixed-size *segments*, which are exclusively allocated to a data superblock *on demand* to store its RMM entries. A *segment validity bitmap (SV-bitmap)* is maintained in DRAM or NVRAM to indicate whether each segment is used or free. Each segment is partitioned into slots, which are written with RMM entries in a *log-structured* manner. When any data page in a superblock is remapped, the relevant RMM entry is appended in the free NVRAM segment allocated to the superblock (e.g., ❸ in Fig. 3). If the superblock has no segments yet (e.g., ❹ in Fig. 3) or the segment in use is full (e.g., ❷ in Fig. 3), a new free segment is assigned first. We refer to the NVRAM segments that belong to a data superblock as a *segment group*.

As remap operations are conducted, the RMM size increases and free NVRAM segments are consumed. When NVRAM is full, Remap-SSD-LH takes one of two measures to produce free NVRAM segments for future remap operations. First, *NVRAM GC* operations can be performed to reclaim NVRAM segment groups containing invalid RMM entries, which will be discussed in Section IV-D. Second, RMM storage can be extended beyond the NVRAM capacity with flash memory through *destage* operations. Remap-SSD-LH triggers NVRAM GC if the ratio of valid RMM entries is lower than a high watermark (95% by default). Otherwise, the benefit of NVRAM GC is very small and the NVRAM capacity is considered to be short. In this case, Remap-SSD-LH chooses the largest NVRAM segment group and destages RMM entries

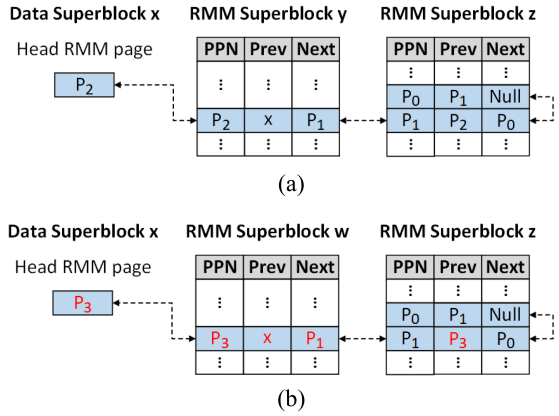


Fig. 4. Indexing of RMM on flash memory. The RMM of a data superblock can be stored in a group of flash pages indexed by a double-linked list. (a) Indexed RMM pages of data superblock *x*. (b) Index updates after the GC of RMM superblock *y*.

in the group to flash pages in dedicated superblocks, called *RMM pages/superblocks*; then, the segments in the victim group can be freed. An RMM page only accommodates the RMM entries of a specific data superblock. As a flash page is usually larger than an NVRAM segment, RMM pages may be partially filled. We refer to the RMM pages that belong to the same data superblock as an *RMM page group*.

The local log of a data superblock consists of an NVRAM segment group and an RMM page group (a group may contain zero segments/pages). To retrieve a local log, Remap-SSD-LH employs a lightweight indexing method. First, NVRAM segments or RMM pages in each group are linked together in the order of their write time. Each NVRAM segment has a head metadata entry that points to the next segment in the same group (see Section IV-C). We refer to the indexing structure of RMM pages as *RP-index*. It connects RMM pages in each group using a double-linked list. Assume a number of, say *R*, RMM superblocks are reserved. The RP-index contains *R* subtables and each subtable is indexed by the PPNs in an RMM superblock. A subtable entry, which represents an RMM page, records the PPNs of the previous and next RMM pages in the same group, as shown in Fig. 4(a). For the head RMM page in a group, its previous pointer stores the ID of the data superblock, which the group belongs to. Then, Remap-SSD-LH tracks the head NVRAM segment and head RMM page for each data superblock. Such superblock metadata and the RP-index are kept in DRAM or NVRAM for fast accesses. Their space overheads are discussed in Section IV-F.

Note that NVRAM segments are linked in single-linked lists while the RP-index uses double-linked lists. This is because when valid RMM pages are migrated to new locations during the GC of an RMM superblock, pointers that target the pages must be found and updated. Fig. 4 shows an example where an RMM page is moved from P_2 to P_3. In contrast, single-linked lists of NVRAM segments are reconstructed during NVRAM GC, since the GC granularity is a segment group.

C. Remapping Metadata

The content of an RMM entry should be carefully designed to serve three goals: 1) mapping consistency; 2) atomicity of remap operations; and 3) space efficiency.

First, the changed P2L mapping and timestamp of an LPN remapping should be recorded for the mapping consistency. Recall that a remap operation is to remap a target LPN to the PPN that is currently mapped to a source LPN; if it is a relocation-based remapping (`remapFlag=1`), the source LPN needs to be deallocated. The P2L mapping contains four fields: a pair of *target LPN and PPN*, a *remapping flag*, and an *alterable field*, i.e., a *source LPN* if the flag is set or *null* value otherwise. Without the last two fields, deallocations of source LPNs could not be recognized and then L2P mappings of source LPNs may be revived undesirably after power-off recovery. The *timestamp* can be virtual time. In the current implementation, we use the number of host write/remap operations that have been performed in the SSD, i.e., *write/remap sequence number* for short.

Second, atomicity of remap operations should be maintained, as their executions may be disrupted by sudden power outages. We distinguish two atomicity levels: 1) *remapping atomicity* and 2) *command atomicity*. The former refers to the atomicity of remapping a single LPN, or more precisely, write atomicity of an RMM entry on NVRAM. A partially updated or written RMM entry would result in improper power-off recovery of L2P and P2L mappings and thus data corruption. A remap command includes one or multiple RMM entries that may scatter in different NVRAM segments. Command atomicity implies atomic remap commands. If the write of any RMM entry in a remap command fails, all the mapping changes caused by the command should be discarded.

Partial updates of RMM entries have been avoided by the log structure of NVRAM segments. Remap-SSD-LH needs to further detect incomplete writes of RMM entries on NVRAM for remapping atomicity, and moreover, recognize whether all the RMM entries of a remap command have been persisted successfully for command atomicity. This can be achieved by adding extra fields in each RMM entry.

Modern processors generally support 8-byte atomic writes to NVRAM [33]. Remap-SSD-LH configures RMM entry size to be a multiple of 8 bytes, say $K * 8$ bytes. As K is larger than one, Remap-SSD-LH adopts a simple *torn-bit* mechanism implemented by Mnemosyne [34] to guarantee atomic writes of RMM entries. In every 8 bytes, a single torn bit is preserved. NVRAM segments are initialized to zeros when allocated for use. Completely written entries will have all K torn bits set as ones, while incomplete entries, which have at least one zero torn bit, will be discarded during power-off recovery.

If command atomicity is desired, three more fields are required in an RMM entry: the *start LPN* and *length* of the remap command, and a *command atomicity flag* indicating whether the remap command is required to be atomic. Each remap command can be identified by its write/remap sequence number. When RMM entries on NVRAM are scanned during power-off recovery, a remap command is successfully executed only if all the RMM entries in its LPN range are found to be intact. Otherwise, the remap command is partially performed and will be abandoned to guarantee command atomicity.

Current applications commonly require remapping atomicity. This resembles regular SSDs, where single-page write atomicity is guaranteed and maybe only some of data pages in a write command are persisted after a sudden power

TABLE II
RMM ENTRY

First 8 bytes		Second 8 bytes	
0	Torn bit	64	Torn bit
1-21	Flash page offset in superblock	65-95	Target LPN
		96	Remapping flag
22-63	Write/Remap sequence number	97-127	Null or source LPN

outage. Atomic remap commands are similar to the advanced atomic-write primitives proposed in [35] and [36] and NVMe specification [37]. Although these atomic commands are not widely used yet, they provide an option to reduce the complexity and overheads for atomicity assurance in the host software. In the current implementation, Remap-SSD-LH ensures only remapping atomicity by default.

The third goal of designing RMM entries is to improve the space efficiency, which can be realized by compacting the fields. The target PPN is replaced by its *physical page offset* in the resident flash superblock, as each NVRAM segment is dedicated to a specific superblock. Also, the unused bits in LPN fields can be utilized. Assuming the SSD capacity and page size are 4 TB and 4 kB, respectively, a 4B LPN field can spare two bits for holding the torn bit and/or remapping flag. Table II shows an example layout of an RMM entry, whose size is 16B. The entry size can be extended to 24B, if any fields demand more bits or command atomicity is required.

Besides RMM entries, each NVRAM segment contains a *segment metadata entry* in its head slot. This entry includes a *data superblock ID* that the segment is associated with, the current *write/remap sequence number*, a *segment sequence number* among the segments allocated to the data superblock, and a *next segment ID* that links the segments in a group. In every RMM page, the data superblock ID that the page belongs to is also stored, e.g., in the head slot or OOB area. Hence, association relationships between data superblocks and NVRAM segments or RMM pages can always be restored from persistent metadata.

D. Garbage Collection

Both flash memory and NVRAM are written in a log-structured fashion. Remap-SSD-LH logically divides flash superblocks into a data superblock pool and an RMM superblock pool. We refer to the number of RMM superblocks as R, while the rest are data superblocks. An upper limit is set for R, say $R_{\max}$. A flash superblock is dynamically allocated to store RMM or host data on demand (wear leveling can be conducted across all flash superblocks), while the number limit in each pool is guaranteed. When free space in one superblock pool or NVRAM is running out, GC is performed in data/RMM superblocks or NVRAM segments to reclaim invalid data. Both superblock pools reserve at least one free superblock for migrating valid data during GC.

Assume the sizes of an RMM entry, a superblock, and NVRAM are S_{entry}, S_{sb}, and S_{nvram}, respectively, and there are N_{rp} unique logical pages that have been remapped (i.e., the size of valid RMM is $N_{rp} * S_{\text{entry}}$). If NVRAM is large enough for RMM storage ($S_{\text{nvram}} >= N_{rp} * S_{\text{entry}}$), $R_{\max}$ is equal to zero. Otherwise

$$R_{\max} = \left\lceil \left(N_{rp} * S_{\text{entry}} - S_{\text{nvram}}\right) \,/\, S_{sb} \right\rceil + O_{\max}. \quad (1)$$

The rounding-up part refers to a minimum number of superblocks for the storage of valid RMM and may change over time along with the data duplication rate in the workload. $O_{\max}$ indicates a maximum number of RMM superblocks that can be overprovisioned for improving GC efficiency.

At runtime, an increasing number of RMM superblocks are allocated on demand until the upper limit $R_{\max}$ is reached. RMM superblocks account for only a very small fraction of flash storage due to the small size of an RMM entry. For example, assume the logical/physical SSD capacity is 1 TB/1.25 TB, page size is 4 kB, $S_{\text{entry}} = 16B$, $S_{\text{nvram}} < S_{sb} = 512$ MB, and $O_{\max} = 4$. When the data duplicate ratio is 12.5%, 50%, or 100% (N_{rp} is equal to 128 GB/4 kB, 512 GB/4 kB, or 1 TB/4 kB), $R_{\max}$ can be calculated as $1+4$, $4+4$, or $8+4$, respectively. The overprovisioning ratio for RMM storage is 400%, 100%, and 50%, respectively. In contrast, the total number of flash superblocks is calculated as 2560. In the current implementation, we statically set $O_{\max}$ to provide an overprovisioning ratio of no smaller than 100% when the data duplicate ratio is no higher than 50% (this condition holds true in general workloads). We could increase $O_{\max}$ in extreme cases where the data duplicate ratio is unusually high.

When free data superblocks run out, a victim data superblock with the most invalid pages is chosen for GC. A flash page may be referenced by multiple LPNs due to address remapping. The FTL maintains a reference counting table (RC-table) to track the number of references for each flash page. A flash page is invalidated only when its reference count is zero. Considering most flash pages have small reference counts (e.g., smaller than ten [1]), four-bit counters are used by default. During the GC, valid data pages are moved to free flash pages of a new data superblock. Meanwhile, new RMM entries are generated by updating the valid RMM entries (precisely, PPN fields) of migrated data pages and then written to the local log of the new data superblock. An RMM entry is identified as valid only when its P2L mapping is consistent with the latest L2P mapping. Afterward, the victim data superblock is erased to be free and its local log is invalidated.

The GC of RMM is conducted on both NVRAM segments and RMM superblocks. Instead of recording the validity status of every RMM entry, the FTL tracks the numbers of invalid RMM entries in the NVRAM segment groups and in the RMM page group for each data superblock. Specifically, a bitmap is maintained for LPNs and each entry contains two bits, called *LR-bitmap*. One bit indicates whether the current L2P mapping of an LPN is established by a remap or write operation. For a remapped LPN, the other bit records whether the relevant RMM entry is stored in the NVRAM segment group or the RMM page group. When an LPN previously remapped to a PPN is remapped again or written to a new PPN, the stale RMM entry becomes invalid and the number of invalid entries of the corresponding group increases by one. In addition, the FTL maintains the overall ratios of valid entries in NVRAM and in all the RMM superblocks, respectively.

An NVRAM GC operation is triggered, when free NVRAM segments are exhausted and the ratio of valid RMM entries in NVRAM is below a high watermark (see Section IV-B). A segment group with the largest number of invalid RMM entries is chosen as the victim. Then, valid entries in the group are moved to a new group of free segments, after which the victim segment group is zeroed to be free.

An RMM superblock GC operation is carried out, when $R_{\max}$ RMM superblocks have been used up. The RMM superblock with the most invalid entries is chosen to be reclaimed. Invalid entries in victim RMM pages are discarded, while valid entries are compacted to construct new RMM pages and are written to a reserved free RMM superblock. Afterward, the RP-index is updated accordingly for data superblocks whose RMM pages have been reconstructed.

E. Power-Off Recovery

Power-off recovery aims to recover the FTL to a consistent state with the latest mappings after sudden power outages. The key is to ensure P2L mappings of data pages that have been written on flash memory are persistent. Then, the L2P mapping table can be rebuilt from the P2L mappings.

Remap-SSD-LH maintains *head and tail metadata* in each flash superblock for fast power-off recovery, similar to conventional SSDs [22]. When a flash superblock is allocated, head metadata are written first before any data writes, including at least the type, write timestamp, and erase count of the superblock. The type indicates whether the superblock stores host data pages or RMM metadata or other FTL metadata. Write timestamps preserve the write order of superblocks. Note that flash pages in a block must be written sequentially and blocks in a superblock can be written in parallel. Remap-SSD-LH chooses the first flash page of the Xth block in a superblock to keep head metadata, where X is the modulus of superblock ID and the number of blocks each superblock contains. This enables concurrent reads to head metadata of different superblocks. Tail metadata are retained in the last several flash pages in each data superblock. They collectively hold the P2L mappings and write/remap sequence numbers of data pages that have been written in the superblock.

Power-off recovery of Remap-SSD-LH relies on the head and tail metadata in flash superblocks and RMM in NVRAM segments and RMM superblocks. The main recovery procedure includes three steps. First, head metadata of all flash superblocks are read to identify data superblocks and RMM data superblocks. Second, tail metadata of data superblocks are scanned in write time order, from which we can recover an L2P mapping table indexing data pages that have been written to flash memory. The power-off recovery of traditional SSDs ends after this step. Third, Remap-SSD-LH examines all NVRAM segments and RMM superblocks to obtain intact RMM entries whose timestamps are more recent than the write timestamps of relevant data pages. These entries record the mapping changes caused by the latest remap operations. The up-to-date L2P mapping table can be established by applying the changes. Meanwhile, some other metadata, such as the RC-table, LR-bitmap, and indexes of RMM on NVRAM and flash memory, can be restored.

F. Discussion

Utilization of Address Remapping: In Remap-SSD-LH, the usage of address remapping is only limited by the reference counting capability (e.g., up to 15 for 4-bit counters). If the counter of a flash page reaches its maximum, which rarely happens, remapping to this page is prohibited. In this case, the SSD would *not* return a failure on the relevant remap command and require the host software to perform error handling. Instead, Remap-SSD-LH internally transforms the prevented remap operations to regular physical writes of duplicate data pages, which is transparent to the host. Therefore, host software can maximize the utilization of remapping without concerning SSD-internal details.

It is important to notice that address remapping improves the cost efficiency of data storage, even if NVRAM is employed for RMM storage. Writes of every 1-GB duplicate data through remapping only produce 4-MB RMM. Assume PCRAM, whose bit cost is roughly five times that of flash memory [28], is in use. The cost of storing RMM on PCRAM is only about 2% of the cost of storing duplicate data on flash memory. We also note that NVRAM lifetime is *not* a concern, because NVRAM has more than three orders of magnitude higher write endurance than flash memory (e.g., five orders for PCRAM [28]). In all our experiments, the ratio between the maximum number of write cycles on an NVRAM segment and that on a flash superblock is much smaller than 1000. Moreover, NVRAM GC overheads and thus, write cycles can be largely reduced by migrating valid RMM entries from NVRAM to flash memory.

In addition, address remapping increases the number of references to and thus, read hotness of some flash pages. Such flash pages become more vulnerable to read disturb errors. This concern can be addressed by using existing read refresh mechanisms [15], which migrate victim data to new flash blocks, similar to data migrations during GC. As read disturb errors rarely happen (e.g., after hundreds of thousands of repetitive read operations), implementing a read refresh mechanism in Remap-SSD-LH would incur negligible overheads.

Metadata Overheads: Compared to traditional SSDs where address remapping is not exploited, Remap-SSD-LH introduces extra metadata and thus, increases DRAM or NVRAM consumption. The SV-bitmap and RP-index are kept for local logging (see Section IV-B), while the RC-table and LR-bitmap are maintained for GC (see Section IV-D). Also, the superblock metadata are extended, e.g., recording the locations of local logs and statistics of invalid RMM entries. The SV-bitmap and additional superblock metadata incur negligible space overheads. The sizes of RC-table and LR-bitmap are proportional to the physical and logical capacities of the SSD, respectively. Assume the logical and physical capacities of the SSD are 1 and 1.25 TB, respectively, and page size is 4 kB. The RC-table (with 4-bit counters) size in Remap-SSD-LH is 160 MB, while that (with 1-bit counters) in conventional SSDs is 40 MB. The LR-bitmap takes 64-MB space and may be embedded into existing L2P mapping entries if the PPN field has unused bits. The RP-index size depends on the number of RMM superblocks. Suppose the superblock size is 512 MB and there are six RMM superblocks. The RP-index, which

tracks two 4-byte pointers for every RMM page, is as small as 6 MB.

V. Case Studies and Evaluation

A. Experimental Setups

To evaluate Remap-SSD-LH, we perform three case studies with various applications: 1) *intra-SSD deduplication*; 2) *WAL in SQLite*; and 3) *cleaning in F2FS*. Remap-SSD-LH is compared with the following five schemes.

1) *NoRemap-SSD*, which is a conventional SSD and does *not* exploit SSD address remapping.
2) *Remap-SSD-GF*, which represents a state-of-the-art SSD enabling address remapping (i.e., the commonly adopted scheme in existing studies listed in Table I) and addresses the mapping consistency problem by persisting a global log of RMM entries on parallel flash dies.
3) *Remap-SSD-GN*, which enhances Remap-SSD-GF by storing the global log on high-speed NVRAM.
4) *Remap-SSD-LN*, which maintains local logs for data superblocks like Remap-SSD-LH but employs only NVRAM to store the logs.
5) *Remap-SSD-Opt*, which is an optimal case assuming no limitations on the usage of address remapping and O(1) time in retrieving RMM entries.

To confine the lookup overhead of a global log, Remap-SSD-GF and Remap-SSD-GN have to restrict the log size and thus, the maximum usage of remapping. Remap-SSD-LN also cannot maximize the usage of remapping due to the hardware limit of NVRAM capacity. When the global log or NVRAM has no more space for new RMM entries, remapping has to be forbidden. In contrast, Remap-SSD-LH removes such limitations by employing local logging (which provides fast lookups of RMM) and extending RMM storage with flash memory. The NVRAM segment size in Remap-SSD-LH and Remap-SSD-LN is set as 1 kB by default.

The majority of experiments are conducted on FEMU, a QEMU-based NVMe SSD emulator [23]. FEMU runs in a machine with 3.80 GHz 16-core Intel i7-9800X CPU and 64-GB DRAM. The emulated SSD is configured with 32-GB logical capacity plus 4-GB overprovisioning space (the capacity is limited by DRAM size of the machine). A flash block has 1024 pages whose size is 4 kB. A superblock contains 16 blocks, as the SSD consists of 16 parallel dies (each die has one plane). The flash read, write, and erase latencies are 50 μs, 500 μs, and 5 ms, respectively. The NVRAM read and write latencies are 50 and 500 ns per 64B, respectively. In addition, we carry out some experiments of intra-SSD deduplication on SSDsim, a popular SSD simulator [38], to evaluate the schemes with a larger SSD and real-world traces. The simulated SSD has 256/288 GB logical/physical capacity and 32 dies, while the other configurations remain unchanged. Write-dominant workloads are used for evaluation, since our work aims to reduce duplicate writes.

B. Intra-SSD Deduplication

Intra-SSD deduplication is a case worthwhile for studying for two reasons. First, data duplication incurs extensive duplicate writes, demanding the exploitation of address remapping. Second, deduplication generates complex *UN*-type remapping behaviors, similar to those in copying or snapshotting files. Such behaviors challenge the schemes for maintaining mapping consistency, so their efficiency differences can be clearly presented. In all the schemes excluding NoRemap-SSD, we implement a deduplication engine in the FTL, similar to CAFTL [1]. The FTL maintains a hash-based fingerprint store and computes the fingerprint of each logical data page written from the host. We assume a hardware hash unit is used and the computational overhead is 32 μs [18]. If a fingerprint hits the store, the FTL invokes the remap primitive to map the logical page to be written to the existing logical page that has the same content. Otherwise, the fingerprint is unique and added to the store and the logical page is written to flash memory. Note that we can also deploy the deduplication engine and then call the remap primitive at the host side. This makes no differences for verifying the schemes.

We conduct two sets of experiments on FEMU-SSD running benchmark tools and on SSDsim running a real-world trace. Benchmarks include the *fileserver* and *oltp* workloads in *filebench* [39], and random-write workload (*randw* for short) in *fio* [40]. These benchmarks do *not* include content locality in their data sets. Thus, we use their I/O patterns and simulate contents of logical data pages using a *zipf* distribution, which has been verified in characterizing the content popularity [18]. The distribution is expressed by $P(t_i) = C/t_i^a$, where $C = 1/(\sum_{i=1}^{N} t_i^{-a})$, N is the number of unique contents in the data set and a is the *zipf* parameter representing the skewness in content popularity. We set a as 0.2 and the data duplication ratio as 10% (N is equal to 90% of the total number of logical data pages). The real-workload trace, called *mail*, is collected from production systems at FIU [18]. It contains real fingerprints of data pages for deduplication.

Fig. 5 shows the performance and flash write amplification (WA) of the five schemes when the maximum size of the global log or NVRAM ranges from 2 to 120 MB. The performance metric is bandwidth or operations per second, which is measured by benchmark tools. The WA results from valid data migrations during GC and is calculated as the ratio between total flash page writes and host page writes. The performance of Remap-SSD-GF, Remap-SSD-GN, and Remap-SSD-LN continuously improves, as the maximum log/NVRAM size increases from 2 to 80 MB gradually. Compared with NoRemap-SSD, these three schemes enhance the performance by smaller than 50% under log/NVRAM sizes within 10 MB while by 1.5 to 5.4 times when the log/NVRAM size is 80 MB. This is because a larger log/NVRAM size can afford more remap operations, which reduce duplicate writes.

When the log/NVRAM size continues to increase from 80 to 120 MB, the performance of Remap-SSD-GF and Remap-SSD-GN no longer increases and may decrease, while the performance of Remap-SSD-LN keeps the same as that of Remap-SSD-LH and close to that of Remap-SSD-Opt. Compared with Remap-SSD-GF and Remap-SSD-GN, Remap-SSD-LN or Remap-SSD-LH improves the performance by 45.6% and 28.7% on average, respectively, when the log/NVRAM size is 80 MB. The performance improvements increase to 53% and 31.4%, respectively, when

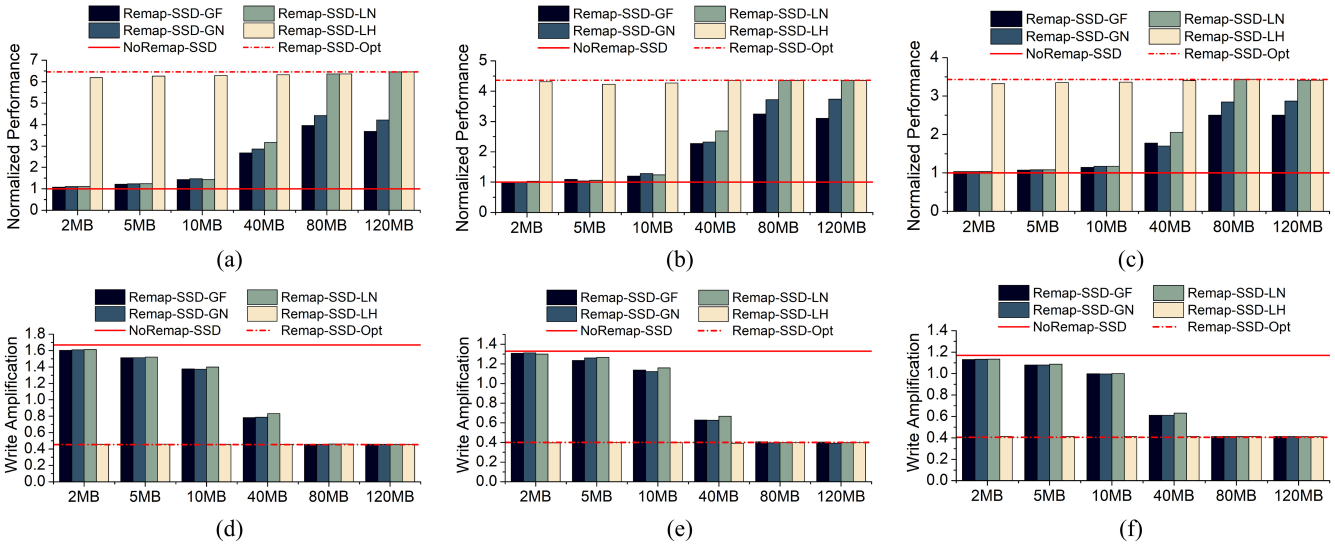


Fig. 5. Performance and flash WA under three traces and different sizes of the global log or NVRAM in the deduplication case. Performance (bandwidth or throughput) numbers are normalized to those of NoRemap-SSD, which does *not* perform data deduplication. (a) Performance in *fileserver* workload. (b) Performance in *oltp* workload. (c) Performance in *randw* workload. (d) WA in *fileserver* workload. (e) WA in *oltp* workload. (f) WA in *randw* workload.

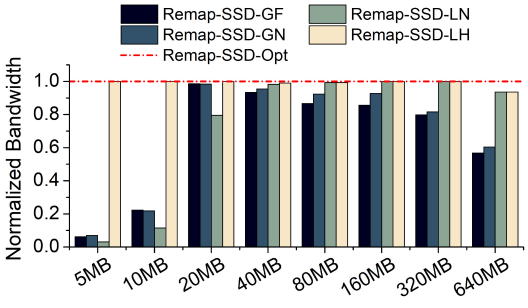


Fig. 6. Normalized bandwidth in intra-SSD deduplication with *mail* trace. Bandwidth is normalized to that of Remap-SSD-Opt. Different log/NVRAM sizes ranging from 5 to 640 MB are evaluated (SSD capacity is 256 GB). NoRemap-SSD is not shown due to large bandwidth gaps with others.

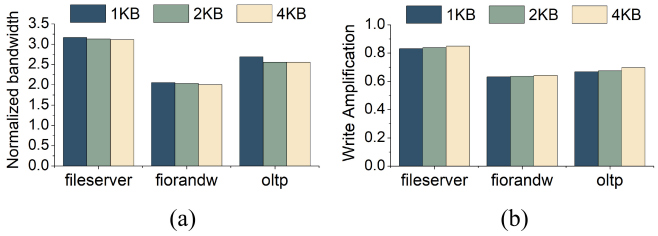


Fig. 7. Impacts of NVRAM segment size in Remap-SSD-LN under intra-SSD deduplication (10% duplicate data). The NVRAM size is 40 MB. (a) Normalized performance. (b) Flash WA.

the log/NVRAM size is enlarged to 120 MB. These results exhibit that the lookup overhead of a large global log becomes significant (even NVRAM is used for logging). The equivalence between Remap-SSD-LN and Remap-SSD-LH reveals that 80-MB log/NVRAM space is large enough to hold the RMM for maximizing the usage of remapping in the workloads (no need for a flash extension).

The superior performance of Remap-SSD-LN owes to the proposed local logging scheme, which enables fast lookups of RMM despite large log space. Notice that Remap-SSD-LH further enhances Remap-SSD-LN by extending RMM storage with flash memory and thus, removing the limitation of NVRAM capacity. As a result, Remap-SSD-LH always achieves close-to-optimal performance. Especially, when the log/NVRAM size is small, ranging from 2 to 10 MB, Remap-SSD-LH improves the performance by 1.8 to 4.7 times, compared with Remap-SSD-GF, Remap-SSD-GN, and Remap-SSD-LN. The utilization of remapping can also significantly reduce the flash WA, since duplicate writes are eliminated and the following GC operations are reduced. As shown in Fig. 5, the WA of Remap-SSD-GF, Remap-SSD-GN, and Remap-SSD-LN decreases as the log/NVRAM size increases. By comparison, the WA of Remap-SSD-LH keeps close to that of Remap-SSD-Opt regardless of the NVRAM size, as the usage of remapping can always be maximized.

In addition, Remap-SSD-LN has the same size of log space as but slightly higher WA than Remap-SSD-GF and Remap-SSD-GN, when the log/NVRAM size is small. The reason is that Remap-SSD-LN allocates NVRAM segments for separate local logs and may leave the segments underutilized. Thus, Remap-SSD-GF and Remap-SSD-GN, which can fully utilize the space of a single global log, can undertake more remap operations. When log space grows, the gaps on space utilization and remapping usage narrow and become negligible.

We also study the performance of Remap-SSD-LH with a 256-GB SSD and a real-world trace, as shown in Fig. 6. The log/NVRAM size varies from 5 to 640 MB. Before running the trace, we age the SSD by issuing random writes until flash GC is triggered and by filling NVRAM with 70% valid RMM entries with random LPNs. The results are consistent with those shown in Fig. 5, where Remap-SSD-LH matches Remap-SSD-Opt under a range of NVRAM sizes. When the log/NVRAM size is no larger than 20 MB, Remap-SSD-LN performs worse than Remap-SSD-GF and Remap-SSD-GN due to lower log space utilization. As the log space increases beyond 20 MB, the performance of Remap-SSD-GF and

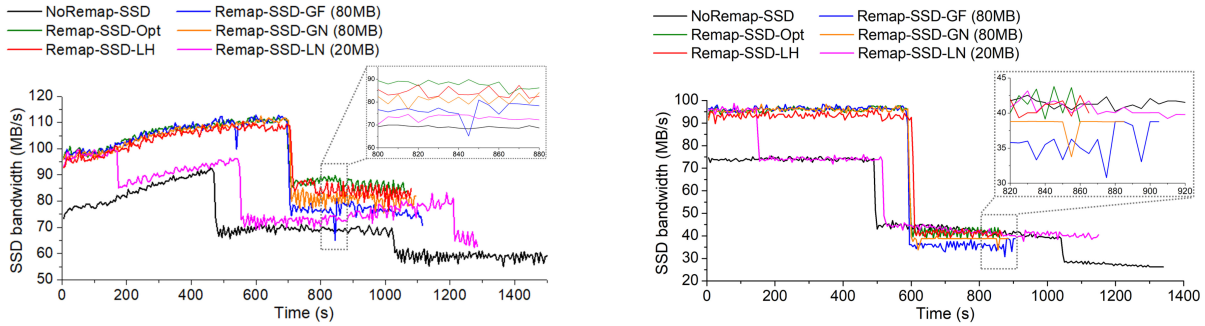


Fig. 8. SSD bandwidth under SQLite with *fillrandom* and *fillseq* workloads. The log/NVRAM sizes in Remap-SSD-GF, Remap-SSD-GN, and Remap-SSD-LN are set as 80, 80, and 20 MB, respectively, while the NVRAM size in Remap-SSD-LH is configured as small as 2 MB. (a) SSD bandwidth in the *fillrandom* workload. (b) SSD bandwidth in the *fillseq* workload.

Remap-SSD-GN degrades and Remap-SSD-LN outperforms them because of low-overhead lookups of RMM.

Fig. 7 shows sensitivity studies on the NVRAM segment size in Remap-SSD-LN. A larger segment size results in a trivial performance degradations and a slight WA increase. This is because NVRAM space utilization degrades (i.e., segments contain unused slots) as the allocation unit is enlarged. We set the segment size as 1 kB by default, despite marginally higher segment metadata overheads.

From the above results, we can make two conclusions. First, maintaining a global log of RMM causes significant performance overheads, which are proportional to the log size. Second, Remap-SSD-LH provides an efficient and scalable scheme that can maximize the utilization of SSD address remapping while ensuring the mapping consistency. When the NVRAM size is small or the log space becomes large, Remap-SSD-LH's performance does *not* degrade and keeps comparable with that of Remap-SSD-Opt.

C. Write-Ahead Logging in SQLite

WAL is a widely used approach for transactional atomicity in databases and file systems [7]. All modifications on the database file are written to a WAL file and then applied to original locations during checkpoint operations. With Remap-SSD-LH, checkpointing writes can be realized through the remap primitive, i.e., remapping LPNs of original locations to those in the WAL file. We use SQLite, a popular database [41], to verify Remap-SSD-LH on reducing WAL overheads. One issue is that data pages in the SQLite WAL file are not page aligned because they are interleaved with frame headers [42]. To make data pages aligned, we simply store frame headers collectively in reserved pages. The remap primitive is implemented as a new NVMe command and is invoked by SQLite through an extended *ioctl* system call.

We use the *db_bench* benchmark [43] to test SQLite (*synchronous* = NORMAL). Two tests are conducted: one writes 1.6 million values in random key order (*fillrandom*) and the other writes 1.5 million values in sequential key order (*fillseq*). The value size is 16 kB. Fig. 8 shows the SSD bandwidth of different schemes over time. Remap operations are counted in measuring the bandwidth like read/write operations. The log/NVRAM size configuration varies in different schemes,

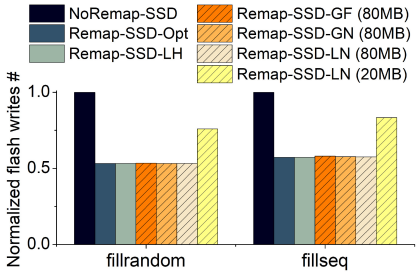


Fig. 9. Numbers of flash page writes in SQLite (normalized to those of NoRemap-SSD). The log/NVRAM size is set as 80 or 20 MB in Remap-SSD-GF, Remap-SSD-GN, and Remap-SSD-LN, but 2 MB in Remap-SSD-LH.

e.g., 80 MB in Remap-SSD-GF and Remap-SSD-GN, 20 MB in Remap-SSD-LN, and only 2 MB in Remap-SSD-LH.

In each test, NoRemap-SSD sustains two sharp performance drops, e.g., at the time around 500 and 1000 s in Fig. 8(a). The first drop is because the SSD has undergone a full disk write and begins to conduct GC operations. At this time, GC overheads are small, because the working set (i.e., the number of valid unique LPNs) size is moderate and the number of invalid flash pages has accumulated to a high level. As the working set grows and invalid flash pages are reclaimed over time, the second performance drop occurs due to increased GC overheads. Compared to NoRemap-SSD, Remap-SSD-LN with 20-MB NVRAM has an additional performance drop at the time around 200 s in Fig. 8(a). This is because the NVRAM log space has been exhausted and remap operations are demoted to expensive flash writes of duplicate data. In contrast, in Remap-SSD-GF and Remap-SSD-GN with a large log/NVRAM size of 80 MB, and Remap-SSD-LH with 2-MB NVRAM, address remapping is fully exploited to enable single-write WAL. Hence, compared to NoRemap-SSD, these three schemes achieve significantly higher SSD bandwidth, postpone the first performance drop, and avoid the second drop. Moreover, the number of flash page writes is reduced by an average of 44.5%, as shown in Fig. 9. Among the schemes, Remap-SSD-LH always outperforms Remap-SSD-GF and Remap-SSD-GN, e.g., by 12.6% and 5.4%, respectively, in the two workloads after GC has been triggered.

It is noticed that there is a performance inversion between the schemes enabling address remapping and NoRemap-SSD after the first performance drop at around 600 s in Fig. 8(b).

This is attributed to higher GC overheads in the schemes with remapping. On the one hand, the schemes with remapping have a larger working set size than NoRemap-SSD at that time due to higher write bandwidth. On the other hand, despite eliminating WAL overheads, remapping reduces the number of invalid flash pages and thus, GC efficiency. In NoRemap-SSD, the WAL file is overwritten repeatedly when it becomes full and its contents have been applied to the database file. Such overwrites lead to invalidation of flash pages that store obsolete WAL contents. In contrast, these flash pages remain valid in the schemes with remapping, because they are remapped to and referenced by relevant logical pages in the database file. As the working set size grows and invalid flash pages are reclaimed by GC over time in NoRemap-SSD, its GC overheads increase and the performance inversion ends.

D. Cleaning in F2FS

Considering the detrimental effects of random writes on SSDs, log-structured file systems naturally fit for SSDs and have drawn close attention [8]. They provide write sequentiality by organizing data in logs. However, cleaning is required to reclaim invalid data blocks. Similar to and independent from intra-SSD GC, the log cleaning process includes migrating valid data blocks and thus, introduces duplicate writes. We modify F2FS, a state-of-the-art and popular log-structured file system designed for flash devices [8], to utilize the remap primitive for migrating valid data blocks at almost zero cost.

Two workloads are used for testing F2FS: 1) the *fileserver* workload in *filebench* and 2) updating *MongoDB* with a zipfian request distribution in *YCSB* [44]. Each test consists of three successive phases: 1) running the workload to generate invalid data blocks in F2FS; 2) manually triggering cleaning operations until all invalid data blocks in F2FS are reclaimed; and 3) running the workload for the second time for performance evaluation. Fig. 10 shows the speedups of the schemes that utilize SSD address remapping over NoRemap-SSD on the above three phases. Remap-SSD-LH accelerates the cleaning process by 39.9% and improves F2FS runtime performance (i.e., the second run) by 32.8%, on average. The cleaning process includes many remap operations and thus, accumulates a large number of RMM entries. Compared with Remap-SSD-GF and Remap-SSD-GN, Remap-SSD-LH has similar performance to them before the cleaning phase but an average of 14.9% and 7.6% higher performance, respectively, after the cleaning phase (due to fast lookups of RMM). These results verify the efficiency and scalability of Remap-SSD-LH in exploiting SSD address remapping.

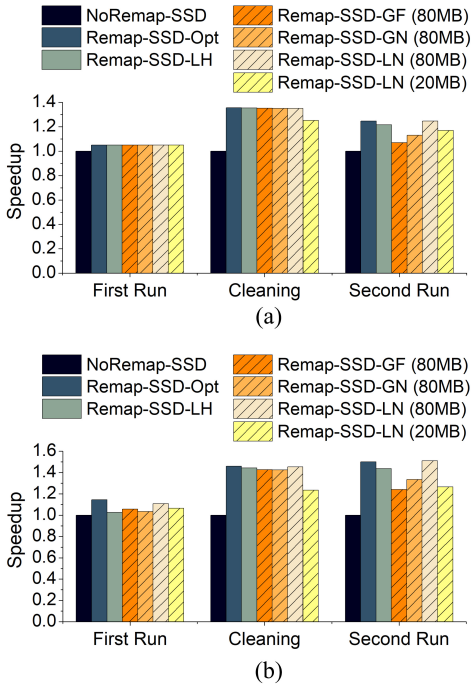


Fig. 10. Speedups in F2FS. Performance is normalized to that of NoRemap-SSD. The log/NVRAM size is set as 80 or 20 MB in Remap-SSD-GF, Remap-SSD-GN, and Remap-SSD-LN, but 2 MB in Remap-SSD-LH. (a) *fileserver in filebench*. (b) *YCSB on MongoDB*.

VI. Related Work

Innovative SSD architectures have been an active field of study in both academia and industry. Below, we discuss some representative designs in two areas related to Remap-SSD-LH, i.e., *novel SSD interfaces* and *hybrid SSD architectures*.

Novel SSD Interfaces: The conventional block interface impedes hardware-software co-designs that can maximally exploit the performance characteristics of flash storage. Hence, several new SSD interfaces have been devised. A number of designs employ remap or similar primitives to reduce duplicate writes by utilizing the SSD address remapping utility [1], [6], [7], [9], [10], [17], [19], [20]. Compared to these designs, Remap-SSD-LH avoids their limitations on the usage of remapping (see Section III) by solving the mapping inconsistency problem in an efficient manner.

Atomic-write interfaces have also been proposed by leveraging the copy-on-write nature of the FTL [35], [36], [45]. Through the interfaces, the burden of ensuring transactional atomicity can be removed from the host software. To eliminate redundant log layers across the storage stack and provide predicable performance, the open-channel and zoned namespaces (ZNSs) interfaces allow the host to directly manipulate data layout on flash memory [46]–[48]. Recently, key-value (KV) interfaces [49]–[51] and dual block- and byte-addressable interfaces [52], [53] have been presented for SSDs. KV-SSDs consolidate KV management with the FTL to provide high-performance and scalable KV stores. Dual-interface SSDs open a fast and fine-grained path to access SSDs. Besides, Willow [54] proposed a user-programmable SSD that enables flexible interactions between the host and SSD. These schemes and Remap-SSD-LH share the same design philosophy of breaking the block interface.

Hybrid SSD Architectures: To address the idiosyncrasies of flash memory and take advantage of emerging NVRAM technologies, hybrid SSD architectures have been studied. NVRAM can be used in different ways for various purposes, e.g., to store the L2P mapping table for fast and energy-efficient address translation [29], to absorb small updates to data on flash memory [55], to replace flash OOB for supporting byte-addressable metadata [19], and to store intra-SSD RAID parity for reducing parity updating overheads [27], [56].

These efforts along with Remap-SSD-LH demonstrate the large design space and potentials of hybrid SSD architectures.

In addition, our design on the co-management of NVRAM and flash storage is partially inspired by the co-management of reserved space and value storage in HashKV [57]. As a KV store built on KV separation, HashKV divides value storage into fix-sized partitions and allows a partition to grow on demand by allocating segments in reserved space.

VII. Conclusion

Reducing flash writes has been a long-standing goal in deploying SSDs. In this article, we presented Remap-SSD-LH, which exports a remap interface and employs a local logging scheme on hybrid storage for RMM. It allows the host and FTL to maximally exploit SSD-internal address remapping utility for eliminating duplicate writes. Meanwhile, Remap-SSD-LH ensures the latest mappings can always be retrieved quickly and recovered from house-keeping metadata persisted on flash memory and NVRAM. Through three practical case studies, we demonstrated Remap-SSD-LH delivers a safe, efficient, and scalable solution that exploits SSD address remapping for performance and lifetime improvements.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable feedback.

References

[1] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proc. 9th USENIX Conf. File Stroage Technol. (FAST)*, 2011, pp. 77–90.

[2] J. A. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," *ACM Comput. Surveys*, vol. 47, no. 1, p. 11, 2014.

[3] W. Xia *et al.*, "A comprehensive study of the past, present, and future of data deduplication," *Proc. IEEE*, vol. 104, no. 9, pp. 1681–1710, Sep. 2016.

[4] Q. Yang, R. Jin, and M. Zhao, "SmartDedup: Optimizing deduplication for resource-constrained devices," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2019, pp. 633–646.

[5] K. Shen, S. Park, and M. Zhu, "Journaling of journal is (almost) free," in *Proc. 12th USENIX Conf. File Storage Technol. (FAST)*, 2014, pp. 287–293.

[6] G. Oh, C. Seo, R. Mayuram, Y.-S. Kee, and S.-W. Lee, "Share interface in flash storage for relational and NoSQL databases," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2016, pp. 343–354.

[7] K. Han, H. Kim, and D. Shin, "WAL-SSD: Address remapping-based write-ahead-logging solid-state disks," *IEEE Trans. Comput.*, vol. 69, no. 2, pp. 260–273, Feb. 2020.

[8] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 273–286.

[9] S. S. Hahn *et al.*, "Improving file system performance of mobile storage systems using a decoupled defragmenter," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2017, pp. 759–771.

[10] Z. Weiss, S. Subramanian, S. Sundararaman, N. Talagala, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "ANViL: Advanced virtualization for modern non-volatile memory devices," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 111–118.

[11] Y. Zhan *et al.*, "How to copy files," in *Proc. 18th USENIX Conf. File Storage Technol. (FAST)*, 2020, pp. 75–89.

[12] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2009, pp. 229–240.

[13] "How Micron SSDs Handle Unexpected Power Loss." Micron. 2014. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/white-paper/ssd_power_loss_protection_white_paper_lo.pdf

[14] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random write considered harmful in solid state drives," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2012, pp. 1–16.

[15] 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.

[16] B. S. Kim, J. Choi, and S. L. Min, "Design tradeoffs for SSD reliability," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST)*, 2019, pp. 281–294.

[17] H. J. Choi, S.-H. Lim, and K. H. Park, "JFTL: A flash translation layer based on a journal remapping for flash memory," *ACM Trans. Storage*, vol. 4, no. 4, p. 14, 2009.

[18] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing NAND flash-based SSDs," in *Proc. 9th USENIX Conf. File Stroage Technol. (FAST)*, 2011, pp. 1–13.

[19] 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.

[20] F. Ni, X. Wu, W. Li, L. Wang, and S. Jiang, "Leveraging SSD's flexible address mapping to accelerate data copy operations," in *Proc. IEEE 21st Int. Conf. High Perform. Comput. Commun. IEEE 17th Int. Conf. Smart City IEEE 5th Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, 2019, pp. 1051–1059.

[21] 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. Manage. Data*, 2011, pp. 1–12.

[22] "Flash Translation Layer in the Storage Performance Development Kit (SPDK)." 2020. [Online]. Available: https://spdk.io/doc/ftl.html

[23] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator," in *Proc. 16th USENIX Conf. File Storage Technol. (FAST)*, 2018, pp. 83–90.

[24] D. Gouk *et al.*, "Amber*: Enabling precise full-system simulation with detailed modeling of all ssd resources," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2018, pp. 469–481.

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

[26] Y. Y. Tai, "High performance FTL for PCIe/NVMe SSDs," in *Proc. Flash Memory Summit*, 2016, pp. 1–20.

[27] Y. Zhou, F. Wu, W. Huang, and C. Xie, "LiveSSD: A low-interference RAID scheme for hardware virtualized SSDs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 7, pp. 1354–1366, Jul. 2021.

[28] M. Oros, "Analysts weigh in on persistent memory," in *Proc. Persistent Memory Summit*, 2018, pp. 1–28.

[29] Y. Hu *et al.*, "Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation," in *Proc. IEEE Symp. Mass Storage Syst. Technol. (MSST)*, 2010, pp. 1–12.

[30] S. Subramanian, S. Sundararaman, N. Talagala, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Snapshots in a flash with ioSnap," in *Proc. 9th ACM Eur. Conf. Comput. Syst. (EuroSys)*, 2014, pp. 1–14.

[31] J. Park, S. Lee, and J. Kim, "DAC: Dedup-assisted compression scheme for improving lifetime of NAND storage systems," in *Proc. Design Autom. Test Europe Conf. Exhibition (DATE)*, 2017, pp. 1249–1252.

[32] M.-C. Yen, S.-Y. Chang, and L.-P. Chang, "Lightweight, integrated data deduplication for write stress reduction of mobile flash storage," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2590–2600, Nov. 2018.

[33] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2018, pp. 461–476.

[34] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2011, pp. 91–104.

[35] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash," in *Proc. 8th USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, 2008, pp. 147–160.

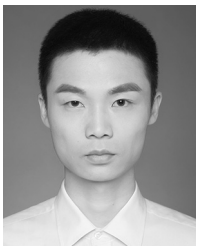
[36] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block I/O: Rethinking traditional storage primitives," in *Proc. 17th IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2011, pp. 301–311.

[37] "NVM Express Base Specification." [Online]. Available: https://nvmexpress.org/resources/specifications/ (accessed May 2021).

[38] 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.
[39] "Filebench Benchmark." [Online]. Available: https://github.com/filebench/filebench/wiki (accessed May 2021).
[40] "Fio Benchmark." [Online]. Available: https://github.com/axboe/fio (accessed May 2021).
[41] "SQLite Home Page," [Online]. Available: https://www.sqlite.org/index.html (accessed May 2021).
[42] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won, "WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2015, pp. 235–247.
[43] "Database Microbenchmarks," [Online]. Available: http://www.lmdb.tech/bench/microbench/ (accessed May 2021).
[44] "YCSB Benchmark." [Online]. Available: https://github.com/brianfrankcooper/YCSB (accessed May 2021).
[45] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, "X-FTL: Transactional FTL for SQLite databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2013, pp. 97–108.
[46] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "SDF: Software-defined flash for Web-scale Internet storage systems," in *Proc. 19th Int. Conf. Archit. Support Programm. Lang. Oper. Syst. (ASPLOS)*, 2014, pp. 471–484.
[47] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind, "Application-managed flash," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 339–353.
[48] M. Bjørling, J. Gonzalez, and P. Bonnet, "LightNVM: The linux open-channel SSD subsystem," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 359–373.
[49] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "KAML: A flexible, high-performance key-value SSD," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2017, pp. 373–384.
[50] S.-M. Wu, K.-H. Lin, and L.-P. Chang, "KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store," in *Proc. Design Autom. Test Europe Conf. Exhibition (DATE)*, 2018, pp. 563–568.
[51] Y. Kang *et al.*, "Towards building a high-performance, scale-in key-value storage system," in *Proc. 12th ACM Int. Conf. Syst. Storage (SYSTOR)*, 2019, pp. 144–154.
[52] D.-H. Bae *et al.*, "2B-SSD: The case for dual, byte- and block-addressable solid-state drives," in *Proc. 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2018, pp. 425–438.
[53] A. Abulila *et al.*, "FlatFlash: Exploiting the byte-accessibility of SSDs within a unified memory-storage hierarchy," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2019, pp. 971–985.
[54] S. Seshadri *et al.*, "Willow: A user-programmable SSD," in *Proc. 11th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2014, pp. 67–80.
[55] G. Sun *et al.*, "A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement," in *Proc. 16th Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2010, pp. 1–12.
[56] S. Im 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.
[57] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "HashKV: Enabling efficient updates in KV storage via hashing," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2018, pp. 1007–1019.

Qiulin Wu received the B.S. degree in computer science and technology from the Wuhan University of Technology, Wuhan, China, in 2017. He is currently pursuing the Ph.D degree with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan.

His research interests include flash storage systems and file systems.



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.

From 2018 to 2020, he worked as a Postdoctoral Researcher with the Wuhan National Laboratory for Optoelectronics, HUST, where he is currently an Associate Professor with the School of Computer Science and Technology. His research interests include nonvolatile memory, solid-state storage architectures and systems, storage quality of service, and hardware-software co-designs.

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, nonvolatile memory, and intelligent storage.

Hong Jiang (Fellow, IEEE) received the B.Sc. degree in computer engineering from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the M.A.Sc. degree in computer engineering from the University of Toronto, Toronto, ON, Canada, in 1987, and the Ph.D. degree in computer science from Texas A&M University, College Station, TX, USA, in 1991.

He is currently the Chair and the Wendell H. Nedderman Endowed Professor of Computer Science and Engineering Department, University of Texas at Arlington, Arlington, TX, USA. His present research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, big data computing, cloud computing, and performance evaluation.

Jian Zhou received the first Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2016, and the second Ph.D. degree in computer engineering from the University of Central Florida, Orlando, FL, USA, in 2018.

In 2021, he joined as an Associate Professor with the Wuhan National Laboratory for Optoelectronics, HUST. He worked as a Postdoctoral Fellow with the University of Central Florida from 2018 to 2020. His research interests include advanced computer architecture, nonvolatile memory technologies, hardware security, solid-state storage drive technologies, big data, and near data processing.

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. He is also the Directors of the Data Storage Systems Laboratory of HUST and the Key Laboratory of Ministry of Education of China. His research interests include computer systems and networks, and emerging storage technologies.