Remap-SSD: Safely and Efficiently Exploiting SSD Address Remapping to Eliminate Duplicate Writes

You Zhou, Qiulin Wu, and Fei Wu, *Huazhong University of Science and Technology;* Hong Jiang, *University of Texas at Arlington;* Jian Zhou and Changsheng Xie, *Huazhong University of Science and Technology*

https://www.usenix.org/conference/fast21/presentation/wu-qiulin

This paper is included in the Proceedings of the 19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

Open access to the Proceedings of the 19th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

Remap-SSD: Safely and Efficiently Exploiting SSD Address Remapping to Eliminate Duplicate Writes

You Zhou[†], Qiulin Wu[†], Fei Wu[†,*], Hong Jiang[‡], Jian Zhou[†], and Changsheng Xie[†]
[†]*Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology*
[‡]*Department of Computer Science and Engineering, University of Texas at Arlington*

Abstract

Duplicate writes are prevalent in diverse storage systems, originating from data duplication, journaling, and data relocations, etc. As flash-based SSDs have been widely deployed, these writes can significantly degrade their performance and lifetime. To eliminate duplicate writes, prior studies have proposed innovative approaches that exploit the *address remapping* utility inside SSDs. However, remap operations lead to a mapping inconsistency problem, which may cause data loss and has *not* been properly addressed in existing studies.

In this paper, we propose a novel SSD design, called *Remap-SSD*, with two notable features. First, it provides a *remap* primitive, which allows the host software and SSD firmware to perform logical writes of duplicate data at almost zero cost. Second, a *hybrid* storage architecture is employed to maintain the mapping consistency. Small byte-addressable non-volatile RAM is used to persist remapping metadata in a log-structured manner and is managed synergistically with flash memory. We verify Remap-SSD on a software SSD emulator with three case studies: intra-SSD deduplication, SQLite journaling, and F2FS cleaning. Experimental results show that Remap-SSD can realize the full potential of address remapping to improve SSD performance and lifetime.

1 Introduction

Duplicate writes are pervasive in real-world storage systems. Not only data duplication is common [16, 51, 62, 64], but also a broad spectrum of system software and applications introduce duplicate writes. For example, many databases and file systems employ double-write journaling to guarantee write atomicity [24, 46, 55]; data relocations are required for space cleaning in log-structured/copy-on-write systems [35, 46] and for file defragmentation [23]; file copy and snapshotting operations are common behaviors [60, 66].

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)* [21, 42]. 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. Notice that writes are harmful to both the performance and lifetime of SSDs [14, 43]. This situation deteriorates, as flash technologies are scaling rapidly to increase the bit density but at the cost of degraded write speed and endurance [33].

To eliminate duplicate writes on flash memory, innovative approaches have been proposed to exploit the SSD *address remapping* functionality [16, 17, 22–24, 28, 34, 45, 46, 60]. By directly modifying the L2P mapping table, copies and moves of data pages as well as duplicate writes of repeating data pages 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 SSDs 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 needs to store some *house-keeping metadata* including the relevant LPN either in the *out-of-band (OOB)* area of the same flash page [21, 41] or in another reserved flash page [8]. These persistent *physical-to-logical (P2L)* mappings are indispensable for completing data relocations during each GC operation and for recovering L2P mappings after sudden power failures (see Section 2). 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,

*Corresponding author. Email: wufei@hust.edu.cn.

compromising data consistency.

This mapping inconsistency problem, although crucial, has *not* been properly addressed in prior studies. The common solution in [16, 22, 23, 34, 45, 46] is to persist new P2L mappings generated by remap operations in a dedicated 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 [24, 28]. These solutions and their drawbacks are discussed thoroughly in Section 3.3.

In this paper, we propose a novel SSD design, called *Remap-SSD*, to safely and efficiently exploit SSD address remapping for reducing duplicate writes. Its two notable features are: (1) providing a *remap* primitive, which allows the host software and SSD firmware to conduct logical writes of duplicate data at almost zero cost; and (2) employing a *hybrid* storage architecture, where small byte-addressable *non-volatile RAM (NVRAM)* is employed to store remapping metadata in a log-structured manner and is managed synergistically with flash storage. Remap-SSD not only ensures that 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 on FEMU (a software SSD emulator [38]) with three case studies: intra-SSD deduplication, SQLite journaling, and F2FS cleaning. Experimental results show that Remap-SSD can realize the full potential of address remapping for improving SSD performance and lifetime.

2 Background

Mappings in flash-based SSDs: Modern SSDs generally employ a page-level FTL, powered by embedded processors and DRAM, for high performance [20, 21]. Since 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 4KB and each mapping entry takes 4B, the table size is about 0.1% of the SSD capacity. The table is persisted on flash memory and usually 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 [42]. 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 [30]. It has been a common practice, especially for high-performance SSDs, to organize flash storage in *superblocks* [8, 14, 20, 54, 58]. A superblock consists of flash blocks with the same offset across multiple dies. Both space allocations for data writes and GC operations 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 [14, 33, 67]. 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 [8]. Then, only a small amount of tail flash pages need to be scanned, rather than all the flash pages.

Non-volatile RAM: NVRAM technologies (e.g., PCRAM and MRAM) have received much attention and their developments are advancing [47]. Compared to flash technologies, they offer attractive benefits, such as lower latency and byte-addressability, but have lower bit density and higher cost. Therefore, NVRAM complements flash memory well and has opened up new opportunities to enhance flash-based SSDs for various purposes [26, 28, 40, 44]. Notably, SSDs with hybrid storage architectures have entered the market since 2019 (e.g., Intel Optane memory H10 with Optane memory and QLC flash [7]) and will gain increased popularity in the near future.

3 Motivation

SSDs have been deployed in diverse storage systems [18, 19], where duplicate writes are prevalent. We illustrate this with several examples in Section 3.1. Although duplicate writes degrade the performance, lifetime, and space utilization of SSDs, they can be eliminated by exploiting SSD address remapping. We detail where and how prior studies leverage SSD address remapping in Section 3.2 and their drawbacks in ensuring mapping consistency in Section 3.3.

3.1 Duplicate Writes

Data Duplication. One major source of duplicate writes is *data duplication*, which is commonplace [10, 39, 51, 62, 64]. For instance, in the disk images of some departmental working environments [16] and file system images collected from smartphones [64], 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% [22].

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) [24, 46]. Either before-images (e.g., *rollback journaling*) or after-images (e.g., *write-ahead logging*) of updated pages are

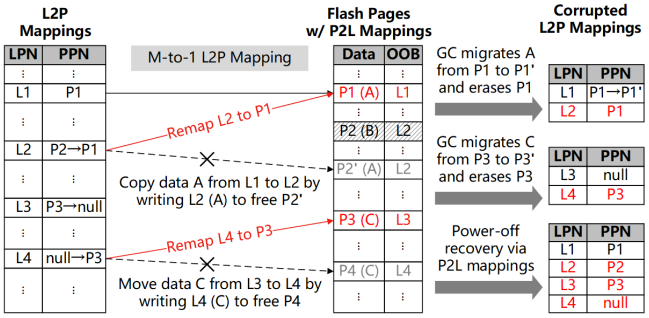


Figure 1: **Examples of SSD remap operations.** Duplicate writes to LPNs *L2* and *L4* can be completed through address remapping without writing flash pages. However, L2P and P2L mappings become inconsistent, causing data corruption.

written in a dedicated log, after which updates are applied to home/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 [55].

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) [35, 46]. 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* [23]. 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 [56]. These operations may introduce duplicate writes to create physical data copies.

3.2 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 Figure 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 application scenarios, as summarized in Figure 2 and Table 1. Among the studies, a body of works integrate a data deduplication engine inside SSDs [16, 22, 34, 50, 63, 65].[1] The 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 [17, 24, 45, 46, 60]. After data pages to be updated are written to the log, they can be applied by remapping LPNs of their original locations to the relevant flash pages storing the log. Using remapping for snapshotting files [60] is straightforward, like copying data *A* in Figure 1. Data relocations for cleaning [28], compaction [46], and defragmentation [23] can be accomplished similarly to moving data *C* in Figure 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 Figure 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 ($\{L1, L2\} \rightarrow P1$ vs. $P1 \rightarrow L1$). 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 3.3, 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* equals 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 write-ahead logging (*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), applications of SSD address remapping are divided into three types (*PD*, *PN*, and *UN*), as shown in Figure 2. The *UD* type is not applicable because the *U* type and *D* type contradict with each other.

[1] Intra-SSD deduplication presents a drop-in solution that is highly desirable for two reasons. First, the detrimental effects of writes on SSDs can be substantially alleviated without modifying the host software and consuming host computing and memory resources. Second, data deduplication can be implemented efficiently by utilizing the FTL's functionalities (e.g., address remapping and GC) [16]. Also, a hardware hash unit can be employed [22].

Table 1: 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 [60]	Snapshots, data deduplication, WAL		
CAFTL [16], CA-SSD [22]	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 [23]	File system defragmentation		
Copyless copy [45]	WAL, intra-SSD data deduplication		
SHARE [46]	WAL, compaction, tree wandering in copy-on-write databases		
PebbleSSD [28]	Cleaning in log-structured file systems	Replace (fixed-size) flash OOB with byte-addressable NVRAM	Only apply in *P*-type remapping scenarios
WAL-SSD [24]	WAL	Write the predetermined LPN for future remapping to flash OOB	Only apply in *PD*-type remapping scenarios

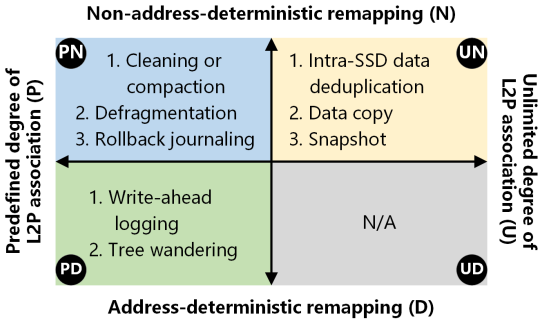


Figure 2: **Applications of SSD address remapping.** They can be classified according to characteristics of remapping.

3.3 Schemes for Mapping Consistency

To address the mapping inconsistency problem caused by remapping, several schemes have been proposed, as listed in Table 1. Taking all types of remapping scenarios into consideration, the common scheme adopted in [16,22,23,45,46] is to maintain a dedicated log on flash memory for persisting the P2L mappings changed by remapping. This scheme is referred to as *Remap-SSD-FLog* in Section 5. 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 are used. Assume the SSD capacity is 4TB, page size is 4KB, 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 (these ratios are quite reasonable, considering the popularity of duplicate writes discussed in Section 3.1), the log size is as large as 600MB or 2.4GB, 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 add high-speed NVRAM for storing the log (denoted as *Remap-SSD-NLog* in Section 5). This is because 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 [23] sets a limit on the log size and reclaims obsolete mapping entries periodically. However, remap operations have to be disabled when the number of valid entries reaches the limit. Additionally, high reclamation overheads are introduced, i.e., reading and re-writing the entire log on flash memory.

PebbleSSD [28] 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 NVRAM OOB area would be required. This would greatly increase the cost. Moreover, NVRAM space utilization would be low, since not all flash pages have high degrees of L2P association.

By utilizing the property of *PD*-type remapping, WAL-SSD [24] 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 full potential of SSD address remapping is largely underutilized.

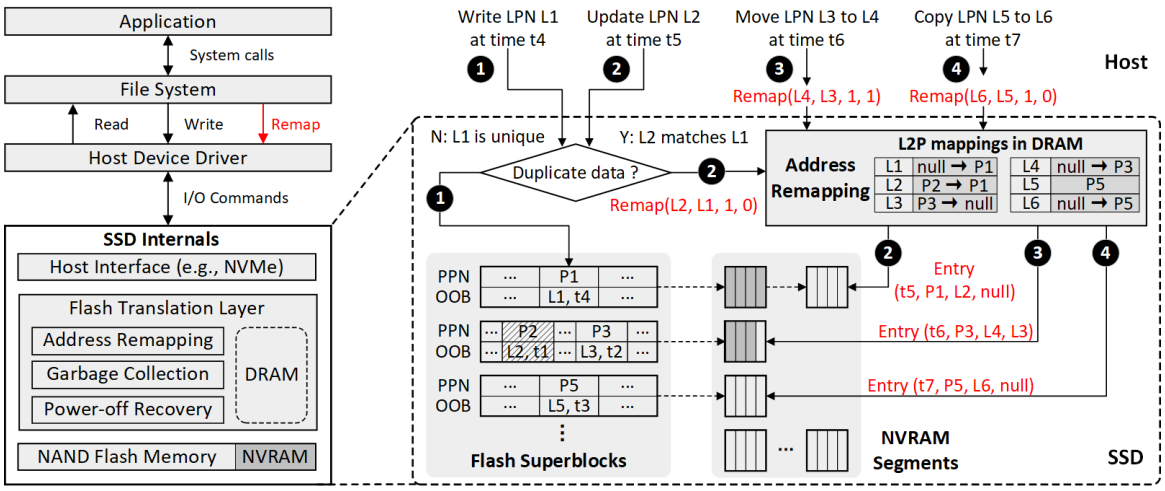


Figure 3: **Overview of Remap-SSD.** The SSD supports a `remap` primitive, which can be invoked by host software (❸❹) or the FTL internally (e.g., by an intra-SSD deduplication engine ❷). To guarantee the L2P and P2L mapping consistency, remapping metadata entries are persisted in NVRAM segments that are exclusively allocated to each flash superblock on demand.

4 Design

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

4.1 Overview of Remap-SSD

Remap-SSD provides a *remap* primitive at the firmware/FTL level, which embodies the address remapping utility, as shown in Figure 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. Furthermore, the primitive can be used internally by the FTL, e.g., to eliminate writes of duplicate data when an intra-SSD deduplication engine is employed.

The remap primitive is formatted as `remap(tgtLPN, srcLPN, length, remapFlag)` (*tgt*: target, *src*: 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`) and the degrees of L2P association of relevant flash pages increase by one. Both remapping and invalidation of LPNs are realized by directly modifying the L2P mapping table in SSD-internal DRAM.

Another notable feature of Remap-SSD is a hybrid storage architecture consisting of flash memory and byte-addressable NVRAM. Flash memory is organized in superblocks for data storage. Each superblock consists of flash blocks with the same offset across all flash dies. Besides P2L mappings and write timestamps that are persisted on flash memory along with data pages, Remap-SSD stores additional house-keeping metadata on NVRAM for address remapping, called *remapping metadata (RMM)*. Whenever an LPN is remapped to a flash page, a RMM entry that includes the changed P2L mapping is written to NVRAM. 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 on NVRAM. Modifications of L2P mappings are *not* required to be persisted because they can be recovered from house-keeping metadata (see Section 4.5). Thus, remap operations can be carried out quickly.

We introduce how to manage RMM entries on NVRAM in Section 4.2, which is the key to solve the problems of high-overhead lookups and poor scalability in the exiting solution (Remap-SSD-FLog). Details of RMM, which guarantee the mapping consistency and remapping atomicity, are described in Section 4.3. Sections 4.4 and 4.5 present how Remap-SSD performs GC operations and power-off recovery, respectively.

4.2 Co-management of Flash and NVRAM

Naively logging RMM entries would result in an expensive scan of the log for every lookup of P2L mappings, as analyzed in Section 3.3. To address this challenge, Remap-SSD takes advantage of a key observation that a flash superblock is the basic unit of free space allocations (for data writes) and GC operations. This observation delivers a favorable conclusion that *retrievals of P2L mappings are always performed in the granularity of a flash superblock*.

P2L mappings are retrieved during GC and power-off recovery. 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. The main process of power-off recovery is rebuilding the latest L2P mapping table based on house-keeping metadata of data pages that have been persisted on flash memory. This process starts with scanning the house-keeping metadata in write time order. Since data pages and their house-keeping metadata are written to flash memory superblock by superblock, P2L mappings of data pages in a superblock are examined together.

Based on the conclusion, Remap-SSD manages flash memory and NVRAM synergistically. The NVRAM volume is divided into fixed-size *segments*, which are exclusively allocated to a flash 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 flash superblock is remapped, the relevant RMM entry is appended in the free NVRAM segment allocated to the superblock (e.g., ❸ in Figure 3). If the superblock has no segments yet (e.g., ❹ in Figure 3) or the segment in use is full (e.g., ❷ in Figure 3), a new free segment is assigned first. We refer to the NVRAM segments that belong to a flash superblock as a *segment group*. A group contains zero or an unfixed number of segments, which are linked together.

An NVRAM segment group is actually a small and size-varied local log of remapping metadata for a flash superblock.[2] Compared with scanning a single global log for retrieving P2L mappings during GC in prior studies (i.e., Remap-SSD-FLog), Remap-SSD achieves fast lookups by scanning only a segment group. Meanwhile, Remap-SSD is adaptive to workloads and has high NVRAM utilization.

4.3 Remapping Metadata

Contents of RMM entries should be carefully designed to serve three goals: mapping consistency, atomicity of remap operations, and space efficiency.

First, the changed P2L mapping and timestamp of an LPN remapping should be recorded for power-off recovery of L2P mappings. 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.[3] 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: *remapping atomicity* and *command atomicity*. The former refers to the atomicity of remapping a single LPN, or more precisely, write atomicity of a 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 several 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 must be able 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 [68]. Remap-SSD configures RMM entry size to be a multiple of 8 bytes, say $K*8$ bytes. As K is larger than one, Remap-SSD adopts a simple *tornbit* mechanism implemented by Mnemosyne [59] 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 a RMM entry: the *start LPN* and *length* of the remap command, 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 outage. Atomic remap commands are similar to the advanced

[2] For SSDs that do not employ a superblock-based FTL, our design still applies and the only change is that the granularity becomes a flash block.

[3] The interface protocols may require an SSD to return an error or some deterministic value (e.g., zeros) when a deallocated LPN is read [4].

Table 2: Remapping metadata 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

atomic-write primitives proposed in [49,52] and NVMe specification [4]. 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 ensures only remapping atomicity by default.

The third goal of elaborating a RMM entry is to improve the space efficiency, which can be realized by compacting its 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 4TB and 4KB, respectively, a 4B LPN field can spare two bits for holding the torn bit and/or remapping flag. Table 2 shows an example layout of a 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 stores a *flash superblock ID* which the segment is associated with, the current *write/remap sequence number*, a *segment sequence number* among the segments allocated to the superblock, and a *next segment ID* that links the segments in a group. The former three fields are written immediately when the segment is allocated, while the next segment ID is written when the segment is full and a next free segment is allocated. The association relationships between flash superblocks and NVRAM segments can be restored from segment metadata entries.

4.4 Garbage Collection

Both the writes of data pages to flash superblocks and RMM entries to NVRAM segments are conducted in a log-structured fashion. Thus, GC is required to reclaim invalid flash pages and invalid RMM entries.

When free flash superblocks run out, a flash GC operation is triggered on a victim superblock (e.g., with the most invalid pages). Since address remapping is enabled, a flash page may be referenced by multiple LPNs. Only flash pages without any references are invalid and can be recycled. The FTL maintains a *reference counting table (RC-table)* to track the number of references to each flash page. Consider the number of writes on most duplicate data is small (e.g., smaller than ten [16,34]). Four-bit counters are used by default.

NVRAM GC is performed both passively and actively. Reclamation of a flash superblock causes a passive recycle on its NVRAM segment group. An active recycle is triggered when free NVRAM segments run out. The NVRAM segment group with the most invalid RMM entries will be selected as the victim. Invalid RMM entries are those whose P2L mappings are *not* consistent with the latest L2P mappings. The FTL tracks the number of invalid RMM entries in each segment group. Specifically, a bitmap is used to indicate whether the current L2P mapping of each LPN is established by a remap or write operation, called *LR-bitmap*. When a remapped LPN is remapped again or written to a new PPN, the number of invalid RMM entries in the segment group of the flash superblock where the stale PPN resides increases by one. To recycle a segment group, RMM entries in it are checked, where valid entries are migrated to a new group of free segments and invalid ones are discarded. Then, the stale segment group is zeroed to be free.

The usage of address remapping is limited by both the reference counting capability and the NVRAM capacity. If the counter of a flash page reaches its maximum, remapping to this page is prohibited. Also, if all the NVRAM segments are filled with valid RMM entries, remap operations are disabled. It is important to note that these two cases do *not* mean Remap-SSD would return a failure on the relevant remap command and require the host software to perform error handling. Instead, Remap-SSD 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 remap commands without concerning the operational details inside the SSD. In addition, to restrain NVRAM GC overheads, remap operations are disabled when the ratio of valid RMM entries is larger than a high watermark (95% by default).

4.5 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 most recent L2P mapping table can be rebuilt from P2L mappings.

Remap-SSD maintains *head and tail metadata* in each flash superblock for fast power-off recovery, similar to conventional SSDs [8]. 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 FTL metadata or other vendor-specific information. 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 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

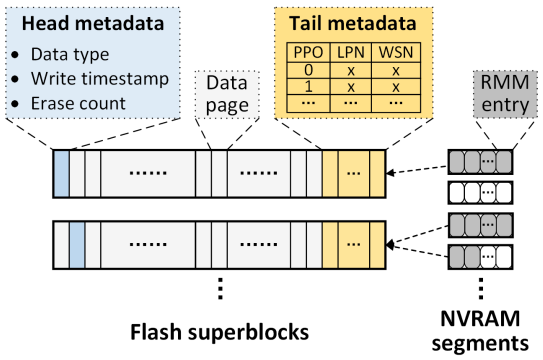


Figure 4: **Persistent metadata for power-off recovery.** *WSN*: write/remap sequence number, *PPO*: physical page offset in the superblock. The latest L2P mappings can be rebuilt from persistent metadata in flash superblocks and NVRAM.

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 relies on the head and tail metadata in flash superblocks and remapping metadata in NVRAM segments, as shown in Figure 4. The main recovery procedure includes three steps. First, head metadata of all flash superblocks are read to identify superblocks storing host data, which are organized in write time order. Second, tail metadata of superblocks are scanned in write time order, from which we can obtain the L2P mapping table established by data page writes and these writes' timestamps. The power-off recovery of traditional SSDs ends after this step. Third, Remap-SSD examines all NVRAM segments. Based on intact RMM entries whose timestamps are more recent than the write timestamps of relevant data pages, the changes to L2P mappings caused by the latest remap operations are applied. As the latest L2P mapping table has been recovered, the RC-table is also acquired. Moreover, segment metadata entries are used to restore the SV-bitmap and association relationships between flash superblocks and NVRAM segment groups.

4.6 Discussion

Hybrid Storage Architecture. One might wonder whether it is necessary for Remap-SSD to employ a hybrid storage architecture or whether NVRAM segments can be replaced by reserved flash pages. We argue that pure flash storage is *not* adequate to address the mapping inconsistency problem. This is mainly due to the size discrepancy between RMM entry and flash write unit. If NVRAM is not adopted, for each flash superblock containing remapped data, its RMM entries would have to be cached in DRAM and accumulate to a page size before being written to a flash page. Then, there would be a large amount of cached entries (from many superblocks) facing the risk of loss if sudden power outages occur. It is feasible to use supercapacitors for some level of power loss protection and periodically flush cached entries. However, this would lead to write amplification and underutilized storage space when cached entries of a superblock cannot fill a page. Also, supercapacitors increase the cost and raise new reliability concerns (e.g., aging effect [11]).

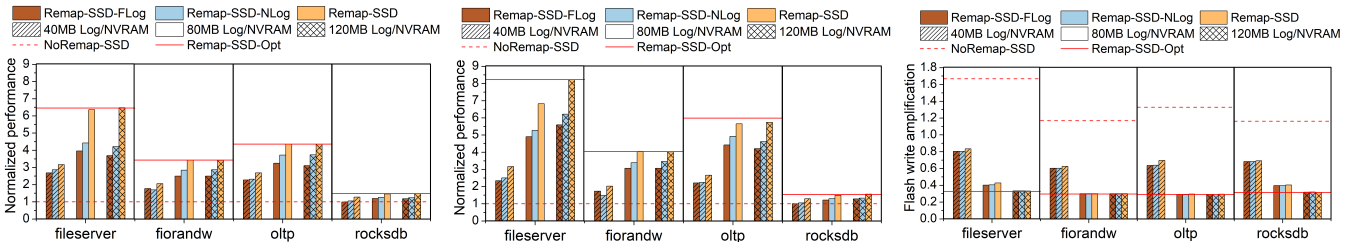
We should note that adding NVRAM in Remap-SSD has high cost-efficiency. The requirement for NVRAM capacity is small. Writes of every 1GB duplicate data through address remapping only produce 4MB RMM. In contrast, the utilization of remapping brings large savings on storage space and cost. Assume PCRAM, whose bit cost is roughly 5 times that of flash memory [47], is in use. The cost of storing RMM on PCRAM is only about 2% of the cost of storing duplicate data on flash memory. On the other hand, given 1GB NVRAM, which can accommodate a maximum of 256GB duplicate data, its cost can be compensated as long as 5GB flash storage space is saved. In addition, the NVRAM lifetime is not a concern, since NVRAM has more than 1,000 times better write endurance than flash memory.

Metadata Overheads. Compared with traditional SSDs whose address remapping ability is not exposed, Remap-SSD introduces extra metadata overheads. First, remapping metadata and segment metadata are stored in NVRAM. The NVRAM capacity limits the maximum number of valid RMM entries and thus unique LPNs that can be remapped. The segment metadata size is inversely proportional to the segment size, for example, 1.6% of NVRAM capacity when the segment size is 1KB. Second, the SV-bitmap (see Section 4.2), RC-table, and LR-bitmap (see Section 4.4) are maintained in DRAM or NVRAM (if DRAM is too small). The SV-bitmap size is negligible. 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 4TB and 5TB, respectively, and the page size is 4KB. The RC-table (with 4-bit counters) size in Remap-SSD is 640MB, while that (with 1-bit counters) in conventional SSDs is 160MB. The LR-bitmap occupies 128MB space and can be embedded into the L2P mapping table if its PPN field has any unused bit.

5 Case Studies and Evaluation

5.1 Experimental Setups

To evaluate Remap-SSD, we perform three case studies with various applications: *intra-SSD deduplication*, *write-ahead logging in SQLite*, and *cleaning in F2FS*. Remap-SSD is compared with one scheme, called *NoRemap-SSD*, which does *not* exploit SSD address remapping, and three other schemes, which exploit SSD address remapping but differentiate in how to guarantee the mapping consistency. *Remap-SSD-FLog* maintains a dedicated log of RMM entries stored on parallel



(a) Normalized performance (10% duplicate data). (b) Normalized performance (30% duplicate data). (c) Flash write amplification (30% duplicate data).

Figure 5: **Intra-SSD deduplication with 10% and 30% data duplication ratios.** Performance (bandwidth or throughput) numbers are normalized to those of NoRemap-SSD, which does *not* perform deduplication. Remap-SSD-FLog, Remap-SSD-NLog, and Remap-SSD are evaluated in each workload with three log/NVRAM sizes, i.e., 40MB, 80MB, and 120MB. Flash write amplifications (lower is better) with 10% duplicate data are not shown as they present similar insights to Figure (c).

flash dies. This scheme corresponds to the commonly adopted solution in existing studies listed in Table 1. *Remap-SSD-NLog* enhances Remap-SSD-FLog by using NVRAM to store the log. *Remap-SSD-Opt* is an optimal case assuming RMM entries can always be retrieved in O(1) time. It also represents prior studies (i.e., PebbleSSD [28] and WAL-SSD [24]) that target only specific applications of remapping. The maximum usage of remapping in Remap-SSD-FLog, Remap-SSD-NLog, and Remap-SSD is restricted by the log/NVRAM size, while Remap-SSD-Opt has no limit. The NVRAM segment size is set as 1KB by default in Remap-SSD.

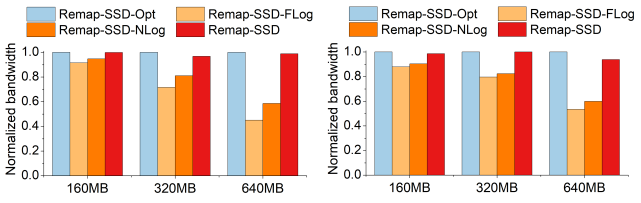
Most experiments are conducted on FEMU, a QEMU-based NVMe SSD emulator [38]. FEMU runs in a machine with 3.80GHz 16-core Intel i7-9800X CPU and 64GB DRAM. The emulated SSD is configured with 32GB logical capacity plus 4GB over-provisioning space (the total capacity is limited by DRAM size of the machine). Every flash block has 1024 pages whose size is 4KB. Each superblock contains 16 blocks, since the SSD consists of 16 parallel dies (each die has one plane). The flash read, write, and erase latencies are 50µs, 500µs, and 5ms, respectively. The NVRAM read and write latencies are 50ns and 500ns per 64B, respectively. In addition, we carry out some experiments of intra-SSD deduplication on SSDsim, a popular SSD simulator [25], to evaluate the schemes with a larger SSD and real-world traces. The simulated SSD has 256GB/288GB logical/physical capacity and 32 dies, while the flash block size remains unchanged. Write-dominant workloads are used for evaluation, since our work aims to reduce duplicate writes.

5.2 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 [16]. 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 32µs [22]. If a fingerprint hits the store, the remap primitive is used 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.

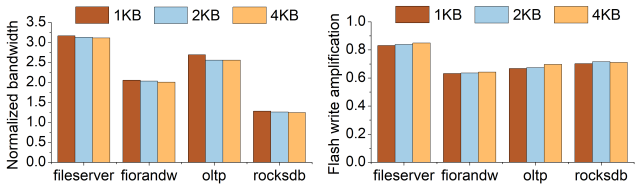
We conduct two sets of experiments on FEMU-SSD running benchmark tools and on SSDsim running real-world traces. Benchmarks include the *fileserver* and *oltp* workloads in *filebench* [2], updating *RocksDB* with a zipfian request distribution in *YCSB* [6], and random-write workload (*fiorandw* for short) in *fio* [3]. 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 [22]. 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, a is the *zipf* parameter representing the skewness in content popularity. We set a as 0.2 and the data duplication ratio as 10% or 30% (N equals to 90% or 70% of the total number of logical data pages, respectively). Real-workload traces include *homes* and *mail*, collected from production systems at FIU [22]. They contain real fingerprints of data pages, which can be used for deduplication.

Figure 5 shows the performance and flash *write amplifications (WAs)* of the five schemes when data duplication ratio is 10% and 30%. The performance metric is bandwidth or throughput (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. Compared to NoRemap-SSD, the other schemes significantly improve the storage performance (e.g., by 1.5~8.2 times in Remap-SSD-Opt) and reduce the WA below one (e.g., by 40.5%~80.4% in Remap-SSD-Opt). Such benefits stem from intra-SSD data dedupli-



(a) SSD bandwidth in *homes*. (b) SSD bandwidth in *mail*.

Figure 6: **Intra-SSD deduplication with real-world traces.** Bandwidth values are normalized to those of Remap-SSD-Opt. Different log/NVRAM sizes, 160MB, 320MB, and 640MB, are evaluated (SSD capacity is 256GB). Bandwidths of NoRemap-SSD are 6~40 times lower than those of the other schemes and are not shown in the figures.



(a) Normalized performance. (b) Flash write amplification.

Figure 7: **Impacts of NVRAM segment size in Remap-SSD under intra-SSD deduplication (10% duplicate data).** The NVRAM size is 40MB. With a larger NVRAM, the impacts of segment size decrease.

cation, which completes host writes of duplicate data through quick remap operations without performing flash page writes. Moreover, deduplication reduces the GC overheads since it results in smaller storage space consumption and thus larger over-provisioning space.

For the three schemes that log remapping metadata (i.e., Remap-SSD-FLog, Remap-SSD-NLog, and Remap-SSD), the log/NVRAM size is a critical factor that affects their performance and WA. When the log/NVRAM size is enlarged, the performance increases because more RMM entries or remap operations can be afforded. With 30% data duplication ratio, 17%~34% of remap operations are demoted to regular flash writes (because the log/NVRAM is full) when the log/NVRAM size is 40MB. The percentages become up to 4.5% and 0%, respectively, when the log/NVRAM sizes are 80MB and 120MB. Compared to Remap-SSD-FLog and Remap-SSD-NLog, Remap-SSD improves the performance by an average of 20.2% and 17%, respectively, when the log/NVRAM size is 40MB. The improvements increase to 38.5% and 24.3% for an 80MB log/NVRAM, and further to 44.3% and 26.8% for a 120MB log/NVRAM. The main reason behind these performance improvements is that Remap-SSD-FLog and Remap-SSD-NLog suffer from high overheads of scanning the entire log, no matter on flash memory or faster NVRAM, in every GC operation. The larger the log size is, the higher the overheads are. In contrast, Remap-SSD always achieves fast lookups by maintaining a small local log for each GC unit on demand, rather than a global log.

On the other hand, Remap-SSD has slightly higher WAs than Remap-SSD-FLog and Remap-SSD-NLog when the log/NVRAM size is small, such as an average of 4.5% and 2.3% for log/NVRAM sizes of 40MB and 80MB, respectively. When the log/NVRAM size increases to 120MB, the three schemes obtain similar WAs. This is because Remap-SSD allocates NVRAM segments for separate local logs and may leave some segments underutilized, while Remap-SSD-FLog and Remap-SSD-NLog can fully utilize the flash/NVRAM log space and undertake more remap operations. When a larger log/NVRAM is used, the gaps on space utilization and remapping efficiency narrow.

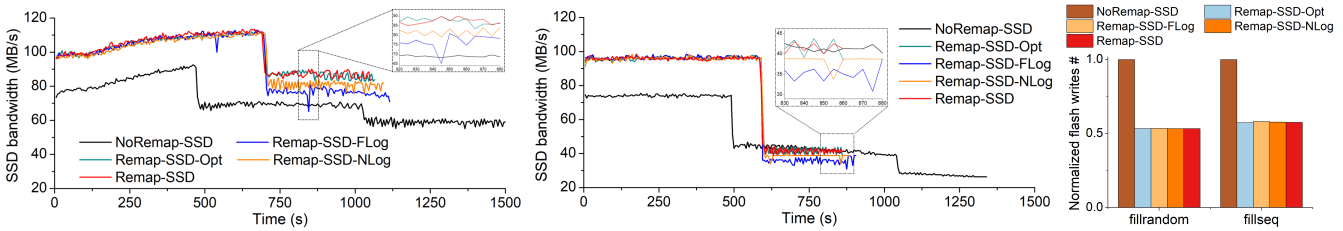
We also study the performance of Remap-SSD with a larger SSD and real-world traces, as shown in Figure 6. Before running each trace, we age the SSD by issuing random writes until flash GC is triggered and by filling NVRAM with 70% valid RMM entries with random LPNs. When the log/NVRAM size is 160MB, 320MB, and 640MB, Remap-SSD averagely improves the performance by 10.7%, 32.1%, and 97.3%, compared to Remap-SSD-FLog, and 7.2%, 22%, and 62.6% compared to Remap-SSD-NLog, respectively. Furthermore, Remap-SSD has close performance to Remap-SSD-Opt, e.g., an average of 2.1% and up to 6.2% lower performance. These results demonstrate rapidly increasing performance overheads of employing a global log when the log size grows and, on the other hand, the good scalability of Remap-SSD. Besides, the three schemes have similar WAs (not shown in figures), as segmenting large NVRAM in Remap-SSD negligibly degrades the space utilization.

Figure 7 shows sensitivity studies on the NVRAM segment size in Remap-SSD. A larger segment size results in trivial performance degradations and slight WA increases. This is because space utilization of NVRAM decreases as the allocation unit is enlarged. We set the segment size as 1KB by default, despite marginally higher segment metadata overheads.

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

5.3 Write-ahead Logging in SQLite

Write-ahead logging (WAL) is a widely used approach for transactional atomicity in databases and file systems [24]. All modifications on the database file are written to a WAL file and then applied to original locations during checkpoint operations. With Remap-SSD, checkpointing writes can be



(a) SSD bandwidth under *fillrandom* workload. (b) SSD bandwidth under *fillseq* workload. (c) Flash page writes.

Figure 8: **Performance results of SQLite.** Numbers of flash page writes are normalized to those of NoRemap-SSD.

realized through the remap primitive, i.e., remapping LPNs of original locations to those in the WAL file. We use SQLite, a popular database [5], to verify Remap-SSD 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 [37]. 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 [1] 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 16KB. Figure 8 shows the SSD bandwidth over time and the numbers of total flash page writes of different schemes. Remap operations are counted in measuring the bandwidth. The log/NVRAM size is 80MB.

In each test, NoRemap-SSD sustains two sharp performance drops, e.g., at the time around 500s and 1000s in Figure 8(a). The first drop is because the SSD has undergone a full disk write and begins to conduct GC operations. At this time, the working set (i.e., the number of valid unique LPNs) size is moderate. As invalid flash pages has accumulated to a high level, GC overheads are small. Then, the working set grows and invalid flash pages are reclaimed over time, increasing the GC overheads significantly. This leads to the second performance drop. We can see the schemes that exploit SSD address remapping postpone the first performance drop and avoid the second drop, because remapping enables single-write WAL and largely reduces flash writes, e.g., by 44.5% on average (see Figure 8(c)). Also, the schemes with remapping finish the tests much faster than NoRemap-SSD. In addition, SSD bandwidth increases over time up to the first drop in Figure 8(a). The reason is that the ratio of reads, which originate from read-modify-write operations for small random updates, rises and the SSD processes reads faster than writes.

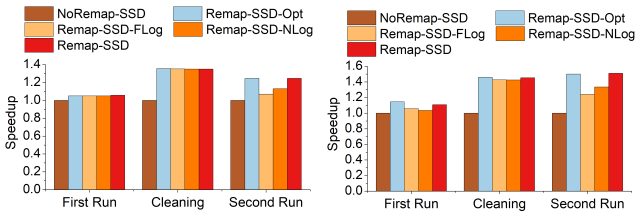
Remap-SSD always outperforms Remap-SSD-FLog and Remap-SSD-NLog, e.g., by an average of 15.1% and 7.8%, respectively, in the two workloads after GC has been triggered. Notably, Remap-SSD-FLog suffers from two bandwidth drops at time 540s and 845s in *fillrandom*. This owes to reclaiming invalid RMM entries in the log on flash memory, which is slower than that in Remap-SSD-NLog. The reclamation requires reading the entire log, writing back valid entries, and erasing flash blocks. In contrast, Remap-SSD looks up and reclaims RMM entries in a small unit, i.e., a segment group, whose largest size is found to be 117KB in the experiments of SQLite. These results exhibit the efficiency of RMM management in Remap-SSD.

We notice that there is a performance inversion between the schemes with remapping and NoRemap-SSD after the first performance drop at around 600s in Figure 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. By 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 between it and Remap-SSD ends.

5.4 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 [35]. 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 [35], to utilize the remap primitive for migrating valid data blocks at almost zero cost.

Two workloads are used for testing F2FS: the *fileserver* workload in *filebench*, updating *MongoDB* with a zipfian request distribution in *YCSB* [6]. Each test consists of three successive phases: (1) running the workload to generate in-



(a) *fileserver in filebench.* (b) *YCSB on MongoDB.*

Figure 9: **Speedups in F2FS.** Performance is normalized to that of NoRemap-SSD. The log/NVRAM size is 80MB.

valid data blocks in F2FS; (2) manually triggering cleaning operations until all invalid data blocks in F2FS are reclaimed; (3) running the workload for the second time for performance evaluation. Figure 9 shows the speedups of the schemes with remapping over NoRemap-SSD on above three phases. The utilization of SSD address remapping accelerates the cleaning process (i.e., the second phase) by an average of 28.3% and improves F2FS performance at runtime by up to 50%. The cleaning process includes a large number of remap operations. Then, Remap-SSD-FLog and Remap-SSD-NLog contain much more RMM entries in the log in the third phase than in the first phase. As a result, average performance improvements of Remap-SSD over Remap-SSD-FLog and Remap-SSD-NLog are 2.8% and 4% in the first phase but increase to 19.1% and 11.6% in the third phase, respectively. These results verify the efficiency and scalability of Remap-SSD in exploiting SSD address remapping.

6 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, 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 [16, 17, 23, 24, 28, 45, 46, 60]. Compared to these designs, Remap-SSD avoids their limitations on the usage of remapping (see Section 3) 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 [31, 49, 52]. 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 ZNS (zoned namespaces) interfaces allow the host to directly manipulate data layout on flash memory [13, 36, 48]. Recently, key-value (KV) interfaces [29, 32, 61] and dual block- and byte-addressable interfaces [9, 12] 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 [53] proposed a user-programmable SSD that enables flexible interactions between the host and SSD. These schemes and Remap-SSD 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 [26], to absorb small updates to data pages on flash memory [57], to replace flash OOB for supporting byte-addressable metadata [28], and to store intra-SSD RAID parity for reducing parity updating overheads [27, 67]. These efforts along with Remap-SSD demonstrate the large design space and great 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 [15]. 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.

7 Conclusion

Reducing flash writes has been a long-standing goal in deploying SSDs. In this paper, we present Remap-SSD, which exports a remap interface and employs a flash and NVRAM hybrid storage architecture. It allows the host and FTL to maximally exploit the address remapping facility for eliminating duplicate writes. Meanwhile, Remap-SSD ensures the latest mappings can always be retrieved quickly and recovered from house-keeping metadata persisted on flash memory and NVRAM together with written or remapped data. Through three practical case studies, we demonstrate Remap-SSD delivers a safe, efficient, and scalable solution that exploits SSD address remapping for performance and lifetime improvements.

Acknowledgments

We would like to thank our shepherd, Patrick P. C. Lee, and the anonymous reviewers for their valuable feedback. This work was supported in part by the NSFC under Grant No. 61902137, No. U2001203, No. 61872413, No. 61821003, Key Area Research and Development Program of Guangdong Province under Grant No. 2019B010107001, National Key Research and Development Program of China under Grant No.2018YFB1003305, the 111 Project (No. B07038), and Key Laboratory of Information Storage System, Ministry of Education of China.

References

[1] Database Microbenchmarks. http://www.lmdb.tech/bench/microbench/.

[2] Filebench Benchmark. https://github.com/filebench/filebench/wiki.

[3] Fio Benchmark. https://github.com/axboe/fio.

[4] NVM express base specification. https://nvmexpress.org/resources/specifications/.

[5] SQLite Home Page. https://www.sqlite.org/index.html.

[6] YCSB Benchmark. https://github.com/brianfrankcooper/YCSB.

[7] Intel Optane Memory H10 with Solid State Storage. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-memory/optane-memory-h10.html, 2019.

[8] Flash translation layer in the storage performance development kit (SPDK). https://spdk.io/doc/ftl.html, 2020.

[9] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. FlatFlash: Exploiting the byte-accessibility of SSDs within a unified memory-storage hierarchy. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, 2019.

[10] Mohammadamin Ajdari, Pyeongsu Park, Joonsung Kim, Dongup Kwon, and Jangwoo Kim. CIDR: A cost-effective in-line data reduction system for terabit-per-second scale SSD arrays. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*, pages 28–41, 2019.

[11] G. Alcicek, H. Gualous, P. Venet, R. Gallay, and A. Miraoui. Experimental study of temperature effect on ultracapacitor ageing. In *Proceedings of the European Conference on Power Electronics and Applications*, 2007.

[12] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2B-SSD: The case for dual, byte- and block-addressable solid-state drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*, 2018.

[13] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, 2017.

[14] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, 105(9):1666–1704, 2017.

[15] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. HashKV: Enabling efficient updates in KV storage via hashing. In *Proceedings of the USENIX Annual Technical Conference (ATC'18)*, 2018.

[16] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*, pages 77–90, 2011.

[17] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. Jftl: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage*, 4(4), 2009.

[18] Kevin Conley. Flash: The Great Disruptor. Flash Memory Summit, 2015.

[19] Bob Fine. Mckesson mixes SSDs with HDDs for optimal performance and ROI. Flash Memory Summit, 2016.

[20] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. Amber*: Enabling precise full-system simulation with detailed modeling of all ssd resources. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[21] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS'09)*, 2009.

[22] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*, 2011.

[23] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of the USENIX Annual Technical Conference (ATC'17)*, pages 759–771, 2017.

[24] Kyuhwa Han, Hyukjoong Kim, and Dongkun Shin. WAL-SSD: Address remapping-based write-ahead-logging solid-state disks. *IEEE Transactions on Computers*, 69(2):260–273, 2020.

[25] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of ACM International Conference on Supercomputing (ICS'11)*, 2011.

[26] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Shuping Zhang, Jingning Liu, Wei Tong, Yi Qin, and Liuzheng Wang. Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation. In *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies (MSST'10)*, 2010.

[27] Soojun Im, Dongkun Shin, Dongkun Shin, Dongkun Shin, and Dongkun Shin. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *IEEE Transactions on Computers*, 60(1):80–92, 2011.

[28] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. Improving ssd lifetime with byte-addressable metadata. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'17)*, page 374–384, 2017.

[29] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A flexible, high-performance key-value SSD. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*, 2017.

[30] Myoungsoo Jung and Mahmut T Kandemir. Sprinkler: maximizing resource utilization in many-chip solid state disks. In *Proceedings of the 20st IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*, 2014.

[31] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*, 2013.

[32] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR'19)*, 2019.

[33] Bryan S. Kim, Jongmoo Choi, and Sang Lyul Min. Design tradeoffs for SSD reliability. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*, 2019.

[34] Jonghwa Kim, Choonghyun Lee, Sangyup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. Deduplication in SSDs: Model and quantitative analysis. In *Proceedings of the 28th IEEE Symposium on Mass Storage Systems and Technologies (MSST'12)*, 2012.

[35] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: a new file system for flash storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'15)*, 2015.

[36] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, 2016.

[37] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proceedings of the USENIX Annual Technical Conference (ATC'15)*, 2015.

[38] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, 2018.

[39] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. Cachededup: In-line deduplication for flash caching. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 301–314, 2016.

[40] Lloyd Liu. MRAM based NVMe SSD Architecture. Flash Memory Summit, 2019.

[41] Dongzhe Ma, Jianhua Feng, and Guoliang Li. LazyFTL: A page-level flash translation layer optimized for NAND flash memory. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2011.

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

[43] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: random write considered harmful in solid state drives. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

[44] Sparsh Mittal and Jeffrey S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2016.

[45] Fan Ni, Xingbo Wu, Weijun Li, Lei Wang, and Song Jiang. Leveraging ssd's flexible address mapping to accelerate data copy operations. In *Proceedings of the IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS'19)*, pages 1051–1059, 2019.

[46] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. Share interface in flash storage for relational and nosql databases. In *Proceedings of the International Conference on Management of Data (SIGMOD'16)*, page 343–354, 2016.

[47] Michael Oros. Analysts Weigh In On Persistent Memory. Persistent Memory Summit, 2018.

[48] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, 2014.

[49] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA'11)*, 2011.

[50] Jisung Park, Sungjin Lee, and Jihong Kim. DAC: Dedup-assisted compression scheme for improving lifetime of NAND storage systems. In *roceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE'17)*, 2017.

[51] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys*, 47(1), 2014.

[52] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[53] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable SSD. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[54] Scott Shadley. NAND flash media management through RAIN. https://www.micron.com/-/media/client/global/documents/products/technical-marketing-brief/brief_ssd_rain.pdf, 2011.

[55] Kai Shen, Stan Park, and Men Zhu. Journaling of journal is (almost) free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, pages 287–293, 2014.

[56] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Snapshots in a flash with ioSnap. In *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys'14)*, 2014.

[57] Guangyu Sun, Yongsoo Joo, Yibo Chen, Dimin Niu, Yuan Xie, Yiran Chen, and Hai Li. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA'10)*, 2010.

[58] Ying Y. Tai. High Performance FTL for PCIe/NVMe SSDs. Flash Memory Summit, 2016.

[59] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, 2011.

[60] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. ANViL: advanced virtualization for modern non-volatile memory devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, 2015.

[61] Sung-Ming Wu, Kai-Hsiang Lin, and Li-Pin Chang. KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE'2018)*, 2018.

[62] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[63] Zhichao Yan, Hong Jiang, Song Jiang, Yujuan Tan, and Hao Luo. SES-Dedup: a case for low-cost ECC-based SSD deduplication. In *Proceedings of the 35th Symposium on Mass Storage Systems and Technologies (MSST'19)*, 2019.

[64] Qirui Yang, Runyu Jin, and Ming Zhao. SmartDedup: Optimizing deduplication for resource-constrained devices. In *Proceedings of the USENIX Annual Technical Conference (ATC'19)*, pages 633–646, 2019.

[65] Miao-Chiang Yen, Shih-Yi Chang, and Li-Pin Chang. Lightweight, integrated data deduplication for write stress reduction of mobile flash storage. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2590–2600, 2018.

[66] Yang Zhan, Alexander Conway, Yizheng Jiao, Nirjhar Mukherjee, Ian Groombridge, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. How to copy files. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 75–89, 2020.

[67] You Zhou, Fei Wu, Weizhou Huang, and Changsheng Xie. LiveSSD: A low-interference RAID scheme for hardware virtualized SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[68] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, 2018.