INVITED PAPER

Software Support Inside and Outside Solid-State Devices for High Performance and High Efficiency

This paper is focused on the software aspects of the solid-state drives, from the Flash translation layer (FTL) to the operating system.

By Feng Chen, *Member IEEE*, Tong Zhang, *Senior Member IEEE*, and Xiaodong Zhang, *Fellow IEEE*

ABSTRACT | In the past decade, flash memory has been in the spotlight across a variety of research communities from circuits to computer systems, and significant progress has been accomplished. This has enabled flash memory to become increasingly pervasive across the entire information technology infrastructure, from consumer electronics to cloud and supercomputing. This paper aims to provide a comprehensive survey on the important advancements and milestones in the domains across flash translation layer (FTL), operating systems, and applications. As the storage device hardware has been quickly commoditized, software becomes increasingly important to tap the potential of flash memory to its full extent. Therefore, a comprehensive survey with a focus on software aspects will be very valuable to the research community and industry. It is our hope that this survey paper will serve as a good reference for system practitioners and researchers.

KEYWORDS | Flash translation layer; NAND flash memory; software; solid-state storage

I. INTRODUCTION

Computer storage systems have been dominated by rotating media for decades. In the past ten years, we have witnessed a landscape change in storage technologies—Solid-state storage, represented by NAND flash memory, quickly expands its application scope from consumer electronics (e.g., cell phones, PDAs, digital cameras) to personal computers, servers, and data center systems [27]. Today, flash storage can be found in various computing environments, from mobile systems, database, virtualization, Internet services to high-performance computing, and many others.

This grand landscape change has created an enormous space for innovations and attracted tremendous interest from both academia and industry. For example, a simple search of "flash memory" on Google Scholar returns over 2 000 000 results, and the industry has experienced a big wave of startups on solid-state data storage (e.g., SandForce, Fusion-io, Pure Storage, Kaminario, Nimble Storage, to name few). Over the past decade, the research community has accomplished significant progress on every aspect of flash-based storage devices and systems, spanning flash memory circuits, flash memory signal processing and error-correction coding (ECC), storage device firmware, OS and file systems, and applications. These advancements together have brought solid-state storage devices and systems into a well-established and researched domain with a broader impact in the real world. Therefore, it is imperative now to conduct a survey about the state of the art of this broad domain, which could not only provide a comprehensive reference to the practitioners but also facilitate future research efforts. This survey paper focuses on the software aspect, from the software inside storage device up to applications. We hope our discussion will serve as a guidance for researchers and system practitioners for a quick understanding on

Manuscript received November 28, 2016; revised January 27, 2017; accepted February 23, 2017. Date of publication April 12, 2017; date of current version August 18, 2017. This work was supported in part by Louisiana Board of Regents under Grant LEQSF(2014-17)-RD-A-01 and in part by the U.S. National Science Foundation under Grants CCF-1629218, CCF-1629201, CCF-1453705, CCF-1629291, CCF-1629403, CCF-1513944, and CNS-1162165.

F. Chen is with the Department of Computer Science and Engineering, Louisiana State University, Baton Rouge, LA 70803, USA.

T. Zhang is with the Electrical, Computer and Systems Engineering (ECSE) Department, Rensselaer Polytechnic Institute (RPI), Troy, NY 12180, USA. (e-mail: tong.zhang@ieee.org)

X. Zhang is with the Computer Science and Engineering Department, Ohio State University, Columbus, OH 43210, USA.

Digital Object Identifier: 10.1109/JPROC.2017.2679490

the impact of flash memory devices to today's computing systems. This paper will survey the prior research efforts and accomplishments in the following two categories.

- Software inside storage devices: As illustrated in Fig. 1, a solid-state storage device such as solid-state drive (SSD) mainly contains a controller and multiple NAND flash memory chips, where the software running on the controller is responsible for all the intelligence inside the SSD. To simplify the discussion, we call the entire software stack running on the controller as the flash translation layer (FTL), which provides a set of highly sophisticated mechanisms to address the technical limitations of flash memory and optimize the device performance. As discussed later, major FTL design objectives include 1) managing address mapping; 2) reducing write amplification; 3) dealing with device wearout; and 4) improving speed performance. In this paper, we will discuss representative studies across all these four aspects.
- Software outside storage devices: On top of FTL is the application layer. We note that applications here refer to a general scope of software, including those running at the system level, such as file systems and virtual machine hypervisor, rather than being strictly limited in "user mode" software. Most flash devices provide a backward-compatible block interface to the host, which enables a large-scale adoption of flash devices in the current computing systems without requiring significant software changes. However, fully exploiting the great potential and taking advantage of the unique properties of flash memory devices often demand efforts of removing or closing the so-called "semantic gap" between the device and the application, which refers to the weak ability for application to pass specific requests or hints to the device for high performance and high efficiency. A large body of research has been performed to optimize applications for flash devices. In this paper, we will discuss representative studies in three main flash-optimized applications, from caching systems, file systems, to database systems.

II. SOFTWARE INSIDE SSD

Inside SSD, a collection of sophisticated software components run on the SSD controller, which are together responsible for managing flash memory resources, handling flash memory operations, optimizing flash performance, and performing routine management and maintenance. This paper refers the entire software stack inside SSD as FTL. How well FTL is implemented directly determines the overall quality of SSD service. This section will discuss the fundamentals of the FTL design and survey its state of the art.

A. FTL Design Objectives

We first briefly discuss the major objectives of FTL design, for which it is necessary to review the very basic device characteristics of NAND flash memory. Each NAND flash memory cell is a floating gate transistor whose threshold voltage can be configured (or programmed) by injecting certain amount of charges into the floating gate. Before one memory cell can be programmed, it must be erased (i.e., its threshold voltage is set to the lowest voltage window). NAND flash memory is subject to gradual memory cell wearout caused by programming/erase (P/E) operations. This leads to a P/E cycling endurance limit that continuously degrades with technology scaling to increase bit density and further reduce the cost.

NAND flash memory cells are organized in an array→block →page hierarchy, as illustrated in Fig. 2, where one NAND flash memory array is partitioned into blocks, and each block contains a number of pages. Within one NAND flash memory block, each memory cell string contains a number of flash memory cells (typically 64–128), and all the memory cells driven by the same wordline are programmed and sensed at the same time. All the memory cells within the same block must be erased at the same time. Data are programmed and fetched in the unit of page, where the page size could range from 4 to 32 kB. All the memory blocks share the bitlines and an on-chip page buffer that holds the data being programmed or fetched. In summary, the above description identifies the following important characteristics of SSD.

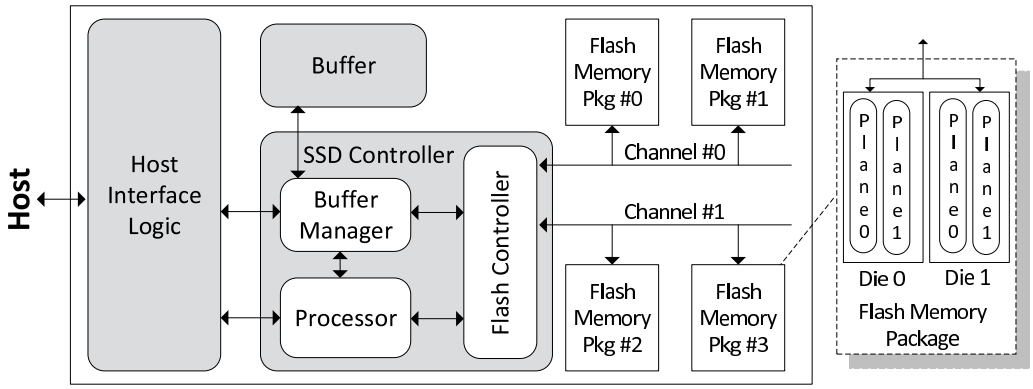

Fig. 1. ***An illustration of SSD architecture [18].***

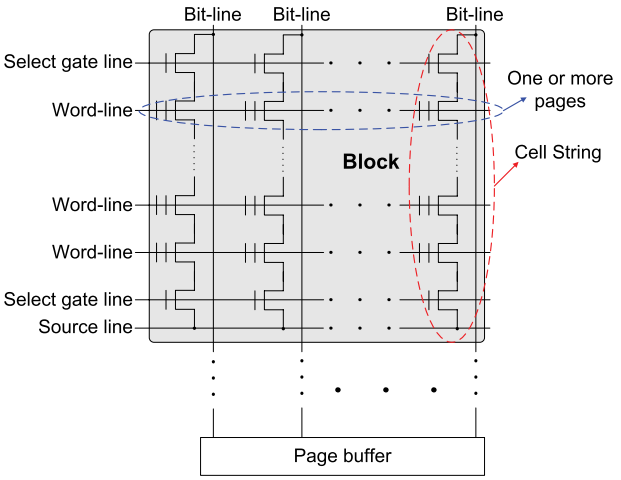

Fig. 2. ***NAND flash memory structure.***

- Absence of update-in-place feature: SSD cannot directly in-place rewrite or update the content of one individual flash memory page. Hence, SSD has to use the same principle of copy-on-write [78] to write any updated content to a new physical flash memory page. As a result, address mapping constantly changes, which makes the mapping information management a nontrivial task.
- Block-based erase: SSD has to erase one entire block, which stores hundreds of pages, at a time. Before one block is erased, any pages that still contain live data must be copied to other locations, leading to write amplification.
- Device wearout: NAND flash memory device storage reliability gradually deteriorates, and the wearout closely relates the P/E cycling number endured by each individual memory block. The overall SSD lifetime is typically defined in terms of total amount of data (e.g., hundreds of terabytes) that can be written to SSD over the time before SSD can no longer ensure its specified storage capacity (e.g., tens of gigabytes).

Meanwhile, speed performance is one important metric of SSDs, for which FTL plays a critical role as well. Therefore, SSD FTL design essentially centers around the following four objectives: 1) managing the address mapping at reasonable implementation cost; 2) reducing the write amplification; 3) dealing with the device wearout; and 4) improving the speed performance. In the remainder of this section, we will discuss and survey the FTL design techniques from these four aspects. For the reference to the readers, Fig. 3 illustrates these four FTL design objectives and lists major options for achieving these objectives, which will be discussed throughout the remainder of this section.

B. Managing Address Mapping

Storage devices internally manage the data being stored on their physical storage media (e.g., platters in HDD and flash memory chips in SSD) in the unit of constant-size *sectors* (e.g., 512 B or 4 kB). Each physical sector is assigned with one unique physical block address (PBA). Instead of directly exposing the PBAs to external host, storage devices expose an array of logical block address (LBA) and internally manage/maintain an injective mapping between LBA and PBA. The reason for introducing such an extra layer of address mapping can be multifold and vary among different types of storage devices (e.g., HDD versus SSD). In the context of HDD, the primary reason is to facilitate the tolerance of defective sectors, which may be caused by disk surface scratches, insufficient magnetic coating material, and deterioration of magnetic materials. Moreover, regardless to their causes, defects in HDDs can be either primary defects, which are detected during the HDD manufacturing, or grown defects, which gradually develop over the time in the field. The internal LBA–PBA mapping makes it possible for HDD controller to mask out the PBAs of those defective sectors while still exposing a continuous storage address space to external host.

SSD employs the extra layer of address mapping not only for defect tolerance, but also, more importantly, for embracing unique device characteristics of NAND flash memory as discussed above. In spite of the seemingly simple task of LBA–PBA mapping management, it is far beyond trivial and

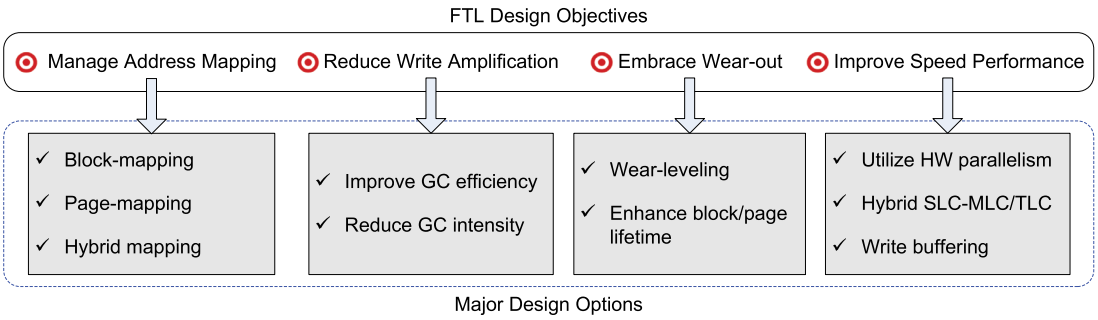

Fig. 3. ***Illustration of the four FTL design objectives and the major options for achieving these objective.***

involves a large design space. Various mapping strategies have been surveyed in [26]. On one side of the design spectrum is the so-called block mapping, which simply uses a linear LBA–PBA mapping within each flash memory block. For example, suppose each memory block stores 2^m sectors, and each LBA contains n bits expressed as $[\mathbf{l}_{n-m}, \mathbf{l}_m]$, where $\mathbf{l}_{n-m}$ and $\mathbf{l}_m$ are $(n-m)$-bit and m-bit vectors. Under block mapping, the 2^m LBAs sharing the same $\mathbf{l}_{n-m}$ are linearly mapped to the 2^m PBAs belonging to the same NAND flash memory block. Let $\mathcal{L}_b$ denote the set consisting of all the 2^{n-m} different possible values of l_{n-m}, and $\mathcal{P}$ denote the set consisting of all the PBAs. Hence, under block mapping, the FTL only needs to manage the $\mathcal{L}_b \rightarrow \mathcal{P}$ mapping. On the other side of the design spectrum is the so-called page mapping, which can map one LBA to any PBA. Let $\mathcal{L}$ denote the set consisting of all the 2^n LBAs, then page-mapping FTL has to manage the $\mathcal{L} \rightarrow \mathcal{P}$ mapping.

Compared with its page-mapping counterpart, the block-mapping FTL manages 2^m times less mapping entries, leading to a much smaller implementation complexity (especially when the FTL aims to hold the entire mapping table in SRAM or DRAM). Meanwhile, the drawback of block mapping is also very clear, i.e., the much worse write amplification due to its block-mapping nature, which leads to much worse SSD speed performance and lifetime. Intuitively, one may expect to achieve a graceful SSD performance versus FTL implementation complexity tradeoffs by appropriately combining block mapping and page mapping, which has been well studied by the research community [32], [46], [51], [56], [57], [74], [92], [96]. Regardless of the specific design details, most prior work shares the following common themes: 1) apply page mapping to a small number of flash memory blocks and combine it with appropriate data write scheduling in order to largely reduce the write amplification; and 2) apply block mapping to the bulk of flash memory blocks to keep the size of the mapping table relatively small. The key challenge is the development of the data write scheduling that can best leverage the page mapping to reduce the write amplification. In [32] and [96], the FTL actively estimates the data hotness by observing the workload characteristics, and accordingly decide whether the data are written to the block-mapping region or page-mapping region. The prior work [46], [51], [56], [57], [74], [92] focused on using the page-mapping region as the write log buffer for the block-mapping region. Data are first written to the write log buffer and then migrated to the normal memory blocks. Such a two-tier structure prevents the memory blocks, which are managed by block mapping, from being directly exposed to random write, leading to much less write amplification.

Since page-mapping FTL manages a large number of mapping table entries, its straightforward realization demands a large memory (SRAM or DRAM) holding the entire mapping table. The rule of thumb is that the size of the page-mapping table is roughly 1/1000 of the SSD storage capacity, e.g., 1 GB of page-mapping entries for each 1-TB storage capacity. Leveraging the runtime workload spatial and temporal locality inherent in many real-world applications, prior research [28], [40], [76], [107] has presented solutions that only cache a small subset of the entire page-mapping table to reduce the memory cost. The key concept is essentially the same as the use of translation lookaside buffer (TLB) in CPU [31]. For example, Gupta *et al.* [28] developed a so-called demand-based FTL (DFTL) design solution that selectively caches the page-mapping table content using the least recently used (LRU) policy. More sophisticated and workload-adaptive cache replacement policies have been presented in [107].

Regardless of the specific address mapping scheme, it is important to guarantee the consistency between the mapping information and the data being stored in SSD. Because the size of each mapping table entry is much smaller than the size of flash memory page, FTL should aggregate many updated mapping table entries together in SRAM/DRAM before flushing them into flash memory. As a result, a sudden power loss could cause the loss of the up-to-date mapping information, especially for SSDs without internal energy storage devices such as supercapacitor. To mitigate this issue, FTL always integrates the inverse mapping information into each sector being written to the flash memory, i.e., FTL records the corresponding LBA information in the spare space within each flash memory physical page. Hence, in the case of mapping table corruption/loss, FTL can scan all the flash memory pages to reconstruct the mapping table.

C. Reducing Write Amplification

Write amplification occurs when one NAND flash memory block to be erased still contains live data. Inside FTL, a process called garbage collection (GC) is responsible for choosing the block(s) to be reclaimed/erased and copying live data out to other blocks. Generally, to reduce the write amplification inside SSD, we have only two options: 1) to improve the efficiency of GC, i.e., reduce the amount of live data inside memory blocks to be erased; and 2) to reduce the intensity of GC process, i.e., reduce the frequency of invoking the GC process. Accordingly, all the existing FTL design techniques for reducing the write amplification fall into the two categories as described through the rest of this subsection.

1) Techniques to Improve the Efficiency of GC: Most straightforwardly, one can use the so-called greedy GC policy [14], which aims to always reclaim the memory block that contains the minimal amount of live data among all the memory blocks in SSD. Typically, FTL invokes the GC process when the number of free memory blocks falls below a threshold that is much less than the total number of memory blocks. As a result, aiming to find the best candidate among all the filled memory blocks, the strictly greedy

GC tends to consume a large number of CPU cycles, especially for large-capacity SSDs. To reduce the CPU stress, a few variations of greedy GC can be deployed, which simply chooses the blocks to be reclaimed from a subset of all the memory blocks. Notable examples are the age-based greedy GC strategies [35], [67] that search for the block with minimal amount of live data only among those blocks that were not recently written. Meanwhile, because of the flash memory wearout caused by P/E cycling, how the memory blocks are reclaimed by GC also affects the wearout pace of all the memory blocks. In order to maximize the SSD lifetime, all the memory blocks should have roughly the same wearout pace and approach their end-of-lifetime almost at the same time. Nevertheless, since greedy GC does not explicitly take into account of flash memory block wearout, it cannot guarantee equalized wearout pace among all the memory blocks. Reducing write amplification and equalizing flash memory wearout could be conflicting at times. Hence, many prior efforts focused on developing GC algorithms that cohesively consider both write amplification and memory wearout, which will be discussed in details in Section II-D.

The efficiency of GC can be significantly improved if the FTL can gain certain knowledge about the characteristics of the data being stored. The TRIM command [49], [83] has been introduced for this purpose. Through the TRIM command, file system can notify the FTL about which data have been deleted by the file system. Suppose the file system writes a sector to the LBA L_i at the time t_0, which is mapped to the PBA P_i in the FTL mapping table, deletes the data from LBA L_i at the time t_1, and then writes a new sector to the LBA L_i later at the time t_2 (where $t_0 < t_1 < t_2$). Without the TRIM command, the FTL is not aware of the deletion operation inside file system at the time t_1, hence must treat the data being stored at the PBA P_i alive until the time t_2. As a result, if GC reclaims the block that contains the PBA P_i anytime between time t_1 and t_2, it has to copy the data from P_i to another location. In comparison, with the TRIM command, such data copy can be directly obviated, leading to reduced write amplification.

The efficiency of GC can be further improved if the FTL could write data with similar lifetime into the same memory blocks. Nevertheless, it is very difficult, if not impossible, for FTL to accurately infer or predict data lifetime on its own. Therefore, to enable its practical realization, applications must convey the data lifetime information to the FTL, which of course demands modification and enhancement of existing storage I/O interface. Researchers at Samsung have recently demonstrated the feasibility by prototyping so-called multistreamed SSDs [45], [102]. With multistreamed SSD, the host can explicitly open different streams through the enhanced I/O interface, and write data with similar expected lifetime to the same stream. The FTL of multistreamed SSDs tries to place data within the same stream onto the same memory blocks. Experiments show that such SSDs can significantly improve the efficiency of GC.

2) Techniques to Reduce the Intensity of GC Process: The intensity of GC process (i.e., how frequently the GC process is invoked) is inversely proportional to the amount of empty memory blocks inside SSDs. As the most straightforward option to reduce the intensity of GC process, overprovisioning [84] has been pervasively used by commercial SSDs. Through overprovisioning, SSDs reserve certain storage space from being exposed to the user, e.g., one SSD with 1-TB storage space available to the users may internally contain 1.2-TB storage capacity in total, representing 20% overprovisioning.

Beyond explicit overprovisioning, FTL can internally deploy data reduction techniques (e.g., lossless data compression and data deduplication) to create more empty memory blocks for reducing the intensity of GC process. This can be considered as opportunistic overprovisioning without demanding additional physical flash memory devices. Since FTL-based data reduction is completely transparent to the external host, SSDs always expose the same storage capacity to the users regardless the efficiency of their internal data reduction. Data reduction can be realized by either lossless data compression or data deduplication, or even their combination. FTL-based lossless data compression has been first implemented in SandForce[1] SSD controllers. The biggest challenge of implementing FTL-based compression is to realize a cost-effective address mapping due to the variable length of compressed sectors. If we record the full location information of each compressed sector (i.e., its head location and its length) in the address mapping table, it could significantly increase the address mapping table size. A hybrid design solution was presented in [23], which sheds a light on how SandForce addressed this issue. The key idea is to split the full location information of each compressed sector between SRAM/DRAM and flash memory. Each physical NAND flash memory contains a fixed number of ECC units (e.g., suppose each ECC unit protects 2 kB, each 16 kB flash memory pages contains eight ECC units). Each ECC unit is assigned with a unique physical ECC unit address (PEUA), similar to the PBA. Given the size of ECC unit, we can calculate the maximum number of ECC units over which one compressed sector can span, denoted as n_{sp}. Since the size of ECC unit typically ranges between 1 and 4 kB in commercial SSDs, the value of n_{sp} is very small (e.g., less than 6). FTL maintains an in-SRAM/DRAM LBA-PEUA mapping table, in which each entry contains one PEUA and additional $\lceil \log_2 n_{sp} \rceil$-bit span information. Inside each ECC unit on the flash memory, there is a header that stores the metadata for recovering the full location information of each compressed sector. Such a hybrid design solution can support FTL-based data compression with almost the same mapping table size as the one without using compression.

Although data deduplication has been well studied (e.g., see [11], [60], [87], and [108]), FTL-based data deduplication

[1]SandForce Inc. is one of the pioneers on developing commercial SSD controllers, which was acquired by LSI Inc. in 2012.

inside SSDs faces several unique challenges, including 1) limited computing and memory resources of SSDs; 2) relatively lower redundancy, especially compared with backup and archival systems; 3) lack of semantic hints from host file systems; and 4) stringent requirement on low performance overhead. How to practically address these challenges was first studied in [19] and [29]. For example, Chen *et al.* [19] presented a design framework, referred to as content-aware FTL (CAFTL), to comprehensively address these challenges. In particular, CAFTL combines both inline and out-of-line (also known as postprocessing or out-of-band) deduplication. Inline deduplication proactively examines the incoming data and cancels duplicate writes before committing a write request to flash, while out-of-line deduplication periodically scans the flash memory and coalesces redundant data. CAFTL employs a two-level mapping mechanism to coalesce redundant data. In order to minimize the performance impact caused by computing hash values, CAFTL further incorporates a set of acceleration methods to speed up fingerprinting. In addition to identical data deduplication, Wu and He [98] applies delta compression (or similarity-based data deduplication) to better exploit the inherent data redundancy at the cost of a higher FTL design complexity, which could further reduce the intensity of GC process.

D. Dealing With Device Wearout

We can handle the cycling induced device wearout to improve the SSD lifetime from two aspects: 1) to equalize the device wearout across all the memory blocks/pages through wear leveling; and 2) to increase the lifetime of each memory block/page. This section reviews prior research efforts from these two aspects.

1) Wear Leveling: To maximize the lifetime of SSD, one would expect that the reliability of all the memory blocks should degrade to the point beyond the ECC tolerance almost the same time. The process of leveling (or equalizing) the reliability of all the memory blocks is called wear leveling. Ideally, wear leveling aims to schedule the usage of memory blocks in such a way that all the memory blocks maintain the same data storage reliability throughout the SSD lifetime. As one of the most critical components in FTL, wear leveling has been very well studied (e.g., see [13], [14], [21], [33], [50], [69], [86], [99], and [103]) and surveyed in [10] and [26].

Since wear leveling and GC schedule the usage of memory blocks with different objectives (i.e., equalizing the memory block reliability versus reducing the write amplification), their integration and interaction could largely affect the overall quality and complexity of FTL. The design schemes, which more cohesively consider the wear leveling and GC, are called static wear leveling. In particular, static wear leveling aims to equalize the wearout across all the memory blocks through proactively moving cold data from less worn out blocks to more worn out blocks. In comparison, the so-called dynamic wear leveling represents loose coupling between wear leveling and GC, i.e., GC first reclaims/erases memory blocks based upon the write amplification efficiency, then wear leveling chooses which memory blocks to use. As a result, memory blocks storing cold data are worn out slowly relative to other blocks. Assuming a 20%:80% ratio of dynamic data versus static data in one SSD, all the data write traffic is accommodated by 20% of flash memory blocks under dynamic wear-leveling. In comparison, static wear-leveling utilizes all the flash memory blocks to absorb the write traffic. As a result, when using dynamic wear leveling, 20% of flash memory blocks will be worn out much more quickly, leading to five times less lifetime than the case of using static wear leveling. A large variety of specific wear-leveling algorithms has been reported in the open literature [13], [14], [21], [33], [50], [69], [86], [99], [103]).

Because the reliability of memory blocks monotonically degrades with the P/E cycling, most prior work on wear leveling uses the P/E cycling number as the reliability metric, i.e., they aim to equalize the P/E cycling number across all the memory blocks. Nevertheless, due to the inevitable fabrication process variation (especially under highly scaled technology nodes), different memory blocks could exhibit different storage reliability even under the same P/E cycling number, particularly among memory blocks in different flash memory chips. Therefore, memory block could have (largely) different levels of P/E cycling endurance. As a result, simply using P/E cycling number as the equalization target could lead to suboptimal wear leveling. A few recent works [41], [73], [94], [104] investigated the design of wear leveling that explicitly embraces such interblock P/E cycling endurance variation. Pan *et al.* [73] and Yang *et al.* [104] proposed wear-leveling algorithms that directly track and use the runtime bit error statistics of memory block as the equalization target. Woo and Kim [94] presented design schemes that combine the bit error statistics and other flash memory operational characteristics such as programming latency to more accurately quantify the reliability of memory blocks. A so-called wear-unleveling solution is presented in [41] that exploits the wearout pace variation at the page level instead of block level to further improve the overall SSD lifetime.

2) Improving Memory Block/Page Lifetime: An effective option for improving the endurance of each memory block/page is to deploy more powerful ECC such as low-density parity-check (LDPC) codes. Since the ECC module in SSD is largely independent from FTL and has been discussed in other papers in this special issue, we will not discuss it in this paper. Beyond using more powerful ECC to increase the lifetime of individual memory pages, one may recycle worn out memory blocks/pages by leveraging certain flash memory device characteristics. For example, motivated by the intra-page reliability variation (e.g., different segments within the same physical page may exhibit noticeably different reliability), Lin and Hsieh [61] developed a half-level-cell (HLC)

scheme that logically combines segments from two worn out physical pages to form a new usable logic page. Leveraging the self-recovery effect of flash memory cells, a recovery-aware throttling technique was presented in [53] to enhance FTL for improving flash memory lifetime. Motivated by the observation that slowing down the erase process at a lower erase voltage could significantly reduce the cycling-induced device damage, Jeong *et al.* [39] presented an FTL design strategy that can enhance flash memory cycling endurance with minimal impact on system speed performance by dynamically adjusting the memory block erase voltage/latency.

The cycling-induced physical damage also depends on the data content being programmed into memory cells. Hence, it is desirable to transform or manipulate the data content in a damage-aware manner, which however demands extra storage space to enable data content transformation. Li *et al.* [59] proposed to apply data compression approach to create extra space for enabling damage-aware data content transformation. A set of mathematical formulations have been further derived in [59] that can quantitatively estimate flash memory physical damage reduction gained by the proposed design strategies for various compression schemes.

E. Improving Speed Performance

FTL can improve SSD speed performance from three aspects: 1) better utilization of internal hardware parallelism at the channel, package, die, and/or plane level; 2) exploring the high-speed performance of the flash memory pages operating in the SLC mode; and 3) intra-SSD write buffering. The importance of exploiting and utilizing SSD internal hardware parallelism has been well demonstrated [15], [16], [36]. Appropriate LBA–PBA mapping and request scheduling are critical for better utilizing SSD internal parallelism. A variety of techniques (e.g., see [30], [44], [55], [65], [93], and [101]) have been developed to improve the SSD speed performance by enhancing the address mapping and/or request scheduling. A common theme is for FTL to on-the-fly infer the access characteristics of data being stored and utilize such inferred information to guide the address mapping and request scheduling.

The second option aims to leverage the fact that SLC flash memory sustains a much higher write/read speed than its MLC/TLC counterparts. It is well known that the same physical NAND flash memory blocks can operate in either SLC, MLC, or TLC mode with different tradeoffs among speed, density, and reliability. Leveraging this feature, researchers have developed design techniques (e.g., see [12], [38], and [70]–[71]) that enhance FTL to improve the overall SSD speed performance. The underlying rationale is that, if FTL configures a certain portion of MLC/TLC flash memory to operate in SLC mode and uses the SLC region to handle most data access traffic, the overall SSD speed performance could significantly improve. In spite of the simple concept, its practical realization may not be trivial and could noticeably complicate the FTL implementation, especially for FTL that aims to dynamically conthe SLC and MLC/TLC partition during the runtime. Wang *et al.* [91] studied the optimal SLC versus MLC capacity ratio and implemented an FTL that can dynamically adjust the SLC versus MLC capacity ratio according to runtime workload characteristics.

Write buffering has been widely used in SSDs to improve the overall speed performance. Current commercial SSD controllers use either embedded SRAM or standalone DRAM as write buffers. Most SSDs acknowledge the completion of write operation to the host as soon as the incoming data reach the write buffer (i.e., before the data are eventually written to flash memory), even for synchronous write requests. To mitigate the volatile nature of SRAM and DRAM, most SSDs (and all the enterprise-grade SSDs) integrate a supercapacitor to prevent data loss in the case of sudden power loss. The size of write buffer is limited by the energy storage capacity of supercapacitors. Kim and Kang [48] presented a technique that applies delta encoding to enable the use of large write buffer with a small supercapacitor. Prior works [34], [37], [42], [82], [97] have developed various write buffer management algorithms that can further effectively leverage the write buffer to improve SSD speed performance.

III. SOFTWARE OUTSIDE SSD

Application software has been heavily tuned and optimized for underlying disk storage for decades. While migrating to flash storage, application software designers need to address three critical challenges for a successful adoption of such an unconventional technology into the existing I/O stack.

- Challenge #1: The high cost and relatively small capacity of flash devices limit the affordability. When commercial flash SSDs initially emerged in the storage market, a significant obstacle for a large-scale adoption of flash storage devices was their high cost and relatively small capacity, compared to magnetic disk drives. A critical research issue is how to achieve high cost- and performance-efficient operations. Thus, building hybrid storage and leveraging flash SSDs as caching media has been an important research direction at the software level. In this section, we will discuss several such representative work [17], [63], [81].
- Challenge #2: Flash memory has more distinct properties than rotating media. As a semiconductor device, flash memory is fundamentally different from magnetic disks. On one hand, flash memory has several unique technical advantages, such as high random read speed [16] and rich internal parallelism [18]. On the other hand, flash memory also has several significant constraints, such as the high-overhead garbage collection and device lifetime

issues. How to exploit the benefits and mitigate the problems is important and challenging. A large body of prior research has studied these aspects in various systems, such as databases [62], [75] and file systems [43], [47], [52].

- Challenge #3: The block interface raises an information barrier between applications and storage. On one hand, the standard LBA interface allows the legacy software systems to easily migrate to flash storage without any change. On the other hand, this interface creates a huge semantic gap and greatly limits the capability of exploiting the semantic knowledge at the application level and the rich hardware resources at the device level. In this section, we will select and discuss several efforts made to address this issue [43], [63], [62], [75], [81].

In this section, we select three key software applications of flash storage in practical systems, namely caching systems, file systems, and database systems, and we discuss several representative works in each of them. We hope through these discussions we can gain insight from these prior studies on how to efficiently adopt flash memory technology in application software.

A. Caching Systems

Flash memory has been considered as an ideal media for caching—its speed, capacity, and price fall nicely between traditional disk drives and DRAM memory [58]. Prior studies have discussed flash-based caching at different levels, from operating system kernels, virtual machine hypervisors, to applications. Here we discuss three representative ones.

1) Block-Level Caching: A natural consideration is to add flash-based caching layer at the block level. Hystor [17] is a kernel-level hybrid block storage solution that integrates flash memory and conventional disk drives together. Hystor regards compatibility as the top design priority. Similar to redundant array of independent disks (RAID), Hystor provides a virtual block device to the upper level components and hides the complex internals from users. So users can use the hybrid device as any directly attached disk drive, such as creating partitions and making file systems. Internally, the Hystor driver runs as an operating system kernel module, tracks I/O accesses during runtime, and decides the data placement in either the flash SSD or the disk drive. The caching and data placement decisions are completely based on the I/O accesses observed at the block level, which is simply a stream of LBAs.

The key challenge that Hystor addresses is to identify the blocks that are most appropriate for caching. A naïve consideration is to directly use standard caching policies, such as the well-known least recently used (LRU) replacement algorithm. However, on flash memory, the "value" of a block for caching is determined not only by locality but also access pattern. For example, randomly accessed data are considered more valuable for caching than sequentially accessed data, since the former incur higher latencies on disk. Thus, Hystor attempts to cache data that can bring the most benefit, randomly and frequently accessed data. To identify such blocks during runtime, Hystor maintains a data structure, called block table, which is akin to the page table in memory management. By traversing the block table, Hystor can quickly find the hottest region, the hottest subregion, and the hottest block for caching. Eventually, Hystor splits I/O traffic into different devices based on their access patterns—small and random I/O requests are served from the flash SSD; large and sequential I/O requests are served from the disk drive.

Industry has also made efforts in flash caching. For example, Apple has released a hybrid drive product, called Fusion Drive [8], which has been influenced by Hystor and combines a hard drive with an NAND flash SSD. Microsoft Windows Vista includes a feature called ReadyBoost [68] to use flash devices, such as flash thumb drives, as memory extension and disk cache. Intel Turbo Memory (ITM) [66] uses a PCI-Express based flash device and a special driver to cache small request data and buffer dirty data. These industrial solutions share a similar principle, using the high-speed flash memory device to accelerate storage I/O.

2) VM-Level Caching: As a middle layer residing between the host hardware and guest operating systems, virtual machine (VM) is required to deliver high performance, maximize resource utilization, and also provide system isolation. S-CAVE [63] is a hypervisor-based flash caching solution for virtual machine environment. In S-CAVE, the caching decision is made at the hypervisor level, based on its global view of the entire system and all guest VMs. S-CAVE maintains a cache monitor for each VM and a cache space allocator to collect and analyze cache usage information from all VMs to make a global caching decision. Essentially S-CAVE measures the working-set size of each VM during runtime, and according to the trend of hit ratio and working-set size, S-CAVE can decide whether increase or decrease the cache allocation. During this process, each guest VM does not directly involve in caching decision, and the hypervisor makes a global decision based on the observed behaviors of all guest VMs.

To some extent, S-CAVE adopts a relatively conservative approach by respecting the existing interface, similar to Hystor, and attempts to contain changes at the hypervisor level and avoid leaving burdens to the upper-level components. Such a design choice is understandable, since avoiding penetrative changes to guest VM is highly desirable for virtualization environment.

3) Application-Level Caching: Flash devices are also highly desirable to application-level cache systems. In-flash key-value cache systems (e.g., Facebook's McDipper [5] and Twitter's Fatcache [6]) are becoming popular recently. Such application-level cache systems adopt a Memcached-like mechanism to manage key-value data: The flash space

is sliced into fixed-sized slabs. Each slab is further divided into fixed-sized slots, or chunks, each of which is used for storing a key/value pair. A hash mapping table is maintained in memory to map a hashed key to the slab that contains the corresponding key/value item. Clients use get, set, and delete operations to read, write, and remove a specified key/value pair. A garbage collection (GC) procedure routinely runs to reclaim the space of the obsolete and deleted key-value items by examining slabs in a certain manner (e.g., FIFO).

Such a scheme raises several unique issues as it runs on flash SSDs.

1) Redundant mapping table: The FTL in the device firmware maintains a mapping structure to translate logical block addresses to physical flash memory pages, and similarly, the key-value cache running at the application level also maintains an in-memory hash table to map the hashed key to the corresponding slab block in flash.
2) Redundant GC procedures: The FTL implements a device-level GC to recycle the invalidated flash pages before erasing the entire flash erase block, and the key-value cache also has a GC procedure to reclaim the invalidated key-value slots in a slab.
3) Over-overprovisioning: Modern flash SSDs often reserve a large over-provisioning space (OPS), typically 20%–30% of flash space. Such a large amount of reserved flash memory is a hidden and unusable space from the perspective of users. Since key-value caches are often read intensive [9], such a large OPS becomes an expensive overkill.

DIDACache [81] addresses these issues by deeply integrating device and application together, which allows the application-level key-value cache manager to directly drive the low-level flash memory operations. In this solution, the device is only responsible for minimum hardware-level functions, such as scheduling and operating flash memory commands. Other functions that are traditionally handled at the FTL level, such as device-level buffering, FTL-level mapping, wear leveling, garbage collection, are moved to the upper layers. An intermediate library layer is responsible for bad block management, translates slabs to flash blocks, and provides an API interface to the key-value cache system. The key-value cache manager sees and directly manages the flash space, such as mapping the hashed key to the physical location in flash, arranging data layout to exploit the internal parallelism, recycling obsolete key-value items and determining a proper OPS space based on workload demands, etc.

Hystor, S-CAVE, and DIDACache represent three different approaches for flash-based caching. Hystor is a very general-purpose caching scheme and its optimization goal is to maximize the flash space utilization while still retaining the highest backward compatibility by hiding all details from applications at the block level. S-CAVE leverages its domain knowledge at the VM hypervisor level. It requires a change in the hypervisor but the caching details are hidden from the guest VMs, which exploits certain level of semantic knowledge but is still unaware of semantic details of applications. DIDACache represents a more recent trend in flash-optimized applications [72], [90], [105]–[106]—break the traditionally strictly defined interface, open the underlying device details to applications, and exploit the semantic knowledge available at the application level and also the device-level knowledge about the flash memory media. A great deal of application- and device-specific knowledge can be utilized, but the tradeoff is its limited applicable scope and the dependence on special hardware support.

B. File Systems

Designing a flash file system is nontrivial, since it needs to consider several important hardware properties of flash memory, such as the read/write disparity, slow random writes, and wearout problem, just to name a few. Traditional file systems are heavily optimized and tuned for rotating media. Switching to flash storage thus has incited a lot of studies in file systems. Here we discuss three representative ones.

1) File System for Flash Memory: As early as in 1995, Kawaguchi *et al.* presented a design of flash-memory-based file system [47]. Many elements in this paper have influenced the design of modern FTLs and flash-based file systems. For example, in order to create sequential writes on flash, the file system adopts a structure similar to log-structured file system [79]. The data are appended to the tail of the log-like structure. A translation table (i.e., mapping table) is maintained to map a block number to a flash address. Upon a read, the translation table is looked up to locate the corresponding location in flash memory. Upon a write, a new flash page is allocated and the data are written, the original flash page is simply marked as invalid, and the mapping table is updated to point to the new location. A cleaner (i.e., garbage collector) is triggered when the number of available flash memory blocks drops to a low level. The cleaner selects a flash block as a victim, in which the valid pages are first copied to a new location and then the entire block is erased.

As a pioneering work, this paper outlines several important components for flash-based file systems, such as the log-like structure, the logical-to-physical mapping table, the periodic garbage collection, etc. All are essential to optimizing flash storage and can be found in today's FTLs and flash-based file systems, such as YAFFS [64] and JFFS2 [95].

2) Direct File System (DFS): DFS [43] is designed for Fusion-io's ioDrive [1]. Without being constrained by the traditional disk interface (e.g., SATA and SCSI), such PCI-E-based flash devices can not only provide high performance but also allow more direct control over the flash hardware. Leveraging the flexible PCI-E interface, DFS divides the flash management and file system functions among hardware, device driver, and file system in a new way.

The core part is virtualized flash storage layer (VFSL), an intermediate layer running as a device driver in OS kernel between the device and the file system. VFSL provides a large virtualized block address space and implements core FTL functions, such as block abstraction, logical-to-physical translation, space allocation and reclamation, garbage collection, wear leveling, etc. Accordingly, the device firmware is simplified, and so is the file system. For example, VFSL provides a huge 64-b virtual block address space, which is orders of magnitude larger than actual physical flash space. The file system, DFS, can take advantage of the huge virtual address space by directly placing file system objects (files) sparsely and contiguously, as single logical extents, in the logical space. This greatly simplifies the file system design by leaving the complicated allocation and reclamation work to VFSL. Other complex issues, such as crash recovery, buffer cache designs, can also be handled by leveraging the capability of VFSL.

Compared to Kawaguchi's design, DFS simplifies the file system and device firmware design. However, since most flash-related management logic is moved into the device driver level, a special device hardware and a custom file system are needed, which limits its applicable scope.

3) Flash-Friendly File System (F2FS): A more recent work that has been practically used in production systems is F2FS [52] developed by Samsung. Unlike Kawaguchi's early work, which was designed for managing raw flash memory, F2FS is designed for modern flash devices, such as eMMC and SSD. Unlike raw flash memory, these devices are already equipped with a sophisticated FTL to handle flash-specific operations at the firmware level, such as address translation, wear leveling, garbage collection, etc. Therefore, F2FS can be freed from directly handling the low-level technical constraints of flash memory and can focus more on creating a "flash friendly" I/O patterns to indirectly influence the performance.

F2FS does not strictly follow the requirement of flash memory for sequential writes, although it still attempts to organize sequential write patterns. In F2FS, the storage space is sliced into multiple zones, which is divided into sections and further into segments. Data are written in segments in a log-like manner, similar to log-structured file system [79] and Kawaguchi's solution [47]. However, in certain cases, it allows in-place writes. For example, F2FS has a special logging scheme, called threaded logging. Normal logging policy always writes data sequentially into clean segments, and when running out of space, the cleaning procedure copies valid blocks out and reclaims the space occupied by invalidated blocks. Such a process removes random writes but the cleaning is time consuming. Threaded logging, in contrast, directly writes data to the "holes" (i.e., the invalidated blocks) in the dirty segments. Allowing such in-place writes would create flash-unfriendly random writes but lowers the cleaning overhead. F2FS dynamically switches between the two policies depending on the system status by examining if the system is under high pressure of clean segments or not.

Another design consideration in F2FS is to purposefully exploit the rich internal parallelism resources in modern flash devices. For example, F2FS maintains multiple active log segments simultaneously and appends data and metadata to different segments based on the update frequency. So the multiple logs can work simultaneously and the hot and cold data can be physically separated on the flash media. In particular, F2FS classifies node and data blocks into three temperature levels (hot, warm, and cold). For example, direct node blocks are considered hotter than indirect node blocks, and directory blocks are considered hotter than file blocks. In total, six logs are maintained, three logs (hot, cold, and warm) for node blocks and the other three logs for data blocks. Since the six active logs can run simultaneously, the internal parallelism of modern flash devices can be effectively exploited.

If we compare the three file systems above, they all take many key design elements from log-structured file system [79] for flash management, such as the log-structured write, periodic garbage collection, etc. The distinction is where these functions are implemented. Kawaguchi's solution directly manages raw flash memory and thus most functions are implemented in the file system level; DFS moves much of the flash management into the virtualized flash storage layer at the device driver level; F2FS is built on the strength of modern flash devices and relies on a highly sophisticated FTL at the device firmware level. These are three representative approaches with different emphasis on interoperability and efficiency.

C. Database Systems

Database system is a typical data-intensive application. High-speed flash storage provides a long-awaited storage technology that could significantly enhance the data retrieval speed for database applications. However, successfully adopting flash into databases is nontrivial. An important research issue is how to leverage the rich semantic knowledge of databases (e.g., access patterns of data, specific requests for storage management and accesses) and how to exploit the unique properties of flash devices to effectively serve various database operations. Here we discuss two representative work.

1) Hybrid-Storage-Based Database (hStorage-DB): Flash storage is desirable to database designers and users because of its high speed. However, building the entire database on flash is excessively expensive. For this reason, a hybrid storage that integrates both high-speed flash devices and large-capacity disk drives becomes a practically appealing choice.

A straightforward hybrid solution is to directly use a hybrid storage, such as Hystor [17], and run database systems atop. Such an approach relies on the underlying hybrid storage to identify hot data and cache them in the

flash devices. For databases, this solution is suboptimal. First, completely relying on observing I/O patterns at the storage level takes a long ramp-up time to recognize hot data and is difficult to make a timely caching decision. Second, and more importantly, the rich semantic knowledge available at the database level cannot be exploited, not to mention that many semantic information is important but not necessarily related to locality, such as the data lifetime information.

hStorage-DB [62] is designed to exploit the database-level knowledge while still retaining minimal changes to the existing block interface. The key idea of hStorage-DB is to pass semantic hints to the hybrid storage. In current database, the storage manager simply translates a data request into a storage I/O request. In hStorage-DB, the I/O requests are first classified into various predefined classes, and each I/O request is associated with a class ID. The class ID and the request are sent together down to the storage layer. At the hybrid storage layer, each class is linked to a quality of services (QoS) policy, which defines the caching policy. Upon receiving an I/O request, the storage layer uses the class ID as an index to locate the corresponding policy and enforces it in the cache management. Fig. 4 illustrates this process.

An example is how the temporary data are handled in hStorage-DB. In databases, some queries can generate temporary data, which are written into storage, consumed, and then deleted. Due to its weak locality and short lifetime, such temporary data are a challenge for traditional locality-based caching schemes. In hStorage-DB, when the temporary data are generated, the data are assigned the highest priority to stay in the cache, so the following data consumption operations can be satisfied in the high-speed flash cache; after the data are consumed and not useful any more, the database labels the data as low value for caching, and the cache manager can quickly evict the data. In this way, the underlying cache manager can be aware of the temporary data lifetime and make a proper caching decision rather than blindly follow the standard locality principle, such as LRU.

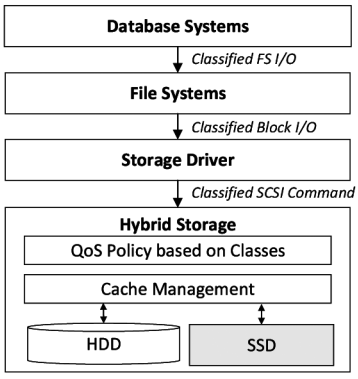

Fig. 4. ***An illustration of passing database hints to hybrid storage.***

hStorage-DB represents a "baby-step" strategy in the database domain: use a hinting approach to allow semantic information flow between the two layers, and in the meantime, avoid radical changes to the standard interface.

2) Transactional Processing on Flash Storage: A more aggressive approach is to tightly connect database requirements to flash storage properties. An example of such efforts is Transactional Flash (TxFlash) [75]. TxFlash aims to provide new abstractions to better utilize the unique properties of flash memory for device-level transaction support. By providing a transactional construct to allow users to perform atomic writes to flash, TxFlash helps reduce the complexity of upper level components, such as databases and file systems.

A TxFlash device is composed of commodity flash packages. Besides the block interface, TxFlash also exports a transaction interface, WriteAtomic and Abort. This interface allows an application to specify a set of page writes in one transaction. The device ensures either all or none of the pages to be written and also guarantees the isolation between transactions and allows to abort uncommitted ones.

The design of TxFlash directly takes advantage of several inherent structural properties of flash SSDs, which make them effective to support transactions natively. For example, the FTL uses copy-on-write (COW) and a log-like method to handle writes. So the "out-of-place writes" requirement by flash memory ensures that write operations in a transaction could be undone if uncommitted. In TxFlash, it leverages the indirection provided by the mapping table to remap the logical pages to the new locations when a transaction is committed. Also, in recovery process, TxFlash can identify the old version and uncommitted intention records and directly garbage collect them. Thus, the implementation of supporting transactions in TxFlash is tightly integrated with the flash SSD internals, rather than a simple change to the interface.

Comparing hStorage-DB and TxFlash, we can find that database systems with rich semantic knowledge can benefit greatly from a close interaction with the storage layer but the cost is the increased complexity and the loss of generality. hStorage-DB attempts to achieve a balance between efficiency and compatibility by passing a small amount of semantic hints; TxFlash deeply exploits various flash-only features, such as its COW nature, fast random reads, and internal parallelism. In essence, this is simply a design choice, and we believe both solutions well illustrate how challenging it is to fully exploit this new technology efficiently.

IV. THE DIVISION OF SOFTWARE

An important factor contributing to the huge success of modern flash devices is the backward-compatible block interface, which allows users to quickly adopt such a new technology without changing any applications.

The associated cost, however, is the straight division of software at two separate layers, inside and outside the device. As flash technologies further advance, especially the adoption of more flexible hardware interface (e.g., PCI-E), a quickly growing effort in industry and academia is to remove the constraint of the current strictly defined block interface to create opportunities for fully exploiting the device potential and satisfying the application needs.

Recently, there have been two sets of such research efforts made on different directions. One direction is to move certain computing functions from the application level down to the device level, represented by Active Flash [85] and Willow [80] ; the other is to expose the device-level details and move low-level device control and functions to the upper level, represented by software-defined flash [72], [81], [90] and application-managed flash [54].

A. Moving Functions From Applications to Device

Active Flash [85] is designed for data analysis for high-performance computing (HPC) applications. Traditional data analysis in HPC often demands a large amount of data movement from the storage system to the computing system, which raises not only performance but also power concerns. Active Flash adopts a concept similar to Active Disk [7], [77]. It tries to exploit the computing capabilities of low-power multicore storage controllers on flash devices by carrying certain analysis tasks directly on the device, which avoids transferring large amount of data across the device/host interface and consuming host-side computing resources. Willow [80] is a similar but more general-purpose solution. Willow provides a programmable interface to allow ordinary programmers to extend the SSD functionality through a mechanism based on remote procedure call (RPC). Willow potentially can satisfy a broad scope of applications that desire to directly offload customized data processing on device, such as Atomic Writes, through the programmable interface.

Both Active Flash and Willow essentially move certain functions from applications down to the device level. Such a "fat-device-thin-application" approach can bring three benefits. First, the abundant device resources can be better utilized. Second, large amount of cross-interface data transfer can be avoided. Third, applications can be made simpler and better modularized. In many cases, such an approach can lead to promising results for applications that can offload certain data processing, such as Nameless Writes [105].

B. Moving Functions From Device to Applications

Another technical trend is happening in the opposite direction [54], [72], [81], [90]. These studies take a "thin-device-fat-application" approach—the device exposes certain device-level information to the upper level applications and allows applications to directly see and carry out certain functions that originally belong to the device. In other words, move certain parts of the FTL software outside of the device and let applications directly handle them.

As an early effort, Ouyang *et al.* presented a scheme called software-defined flash (SDF) [72], which is designed for web-scale storage to maximize the bandwidth and usable capacity for a large-scale deployment of flash storage in data centers. SDF consists of the hardware part, which is based on a customized FPGA controller, and a software layer, which provides applications with accesses to low-level flash memory resources. SDF takes a minimal hardware design, including only features and functions that are absolutely necessary for the target application environment. For example, unlike many commercial SSD products, SDF hardware does not provide parity-based protection on device but uses software-managed data replication. Also, the SDF device exposes an asymmetric operation interface to the software, page-based read and block-based write, which eliminates the small write problem (all writes are in large erase blocks). SDF also unlocks erase operations to applications, which allows applications to directly implement garbage collection and schedule long-latency block erase operations. In SDF, the host software has a direct view and control over flash channels for exploiting internal parallelism resources inside the SSD. Each channel is presented to the application layer as an independent block device, which allows applications to decide data placement, I/O scheduling, parallelization, etc. Based on the same platform, Wang *et al.* further presented an LSM-tree based key-value store, called LOCS [90]. As discussed earlier, DIDACache [81] takes a similar principle for flash based key-value caching. Another similar effort toward the same direction is application-managed flash (AMF) [54]. Similar to SDF, AMF moves selected functions of flash management from the device level to the application level. AMF offers an append-only block interface to applications, which simplifies the device-level management. For example, the remapping and garbage collection can be moved out of the device, and the device-level complexity is significantly reduced and only focuses on core functions, such as bad block management and wear leveling. Applications, especially those that already employ append-only writes, such as log-structured file systems and LSM-tree-based databases, can easily adopt and benefit from such an architecture.

Though different, all these efforts attempt to reconsider a proper division of software functions inside and outside the device. These research studies are essentially driven by two facts. First, the flash device's capability, both computing and storage, is quickly growing and more abundant than ever. Second, the applications desire to gain more resources and controls on hardware for performance, power, and other reasons. As a result, the above-said research efforts in two distinct directions all attempt to remove the intermediate layers out from the I/O path, either bringing computing

components of applications closer to the hardware or opening the hardware controls to the application. A result of such efforts is more tightly integrated applications with devices [81], [106], [105].

On one hand, we are highly optimistic with such research attempts and strongly believe that they represent the future of flash storage and could deeply change the computing storage system. On the other hand, we should also note that when we approach to the goal of closing the gap between the two layers, we are not only tightening the connection between applications and devices but also making an increasingly blurrier boundary between the two—either making a special device for the application or making a special application for the device. As researchers and practitioners, we still need to consider how to retain a reasonable abstraction and balance between efficiency and interoperability, which will still remain a challenging research issue in the future.

V. CONCLUSION AND PROSPECTS

This paper surveys the progress over the past decade on the development of software stacks inside and outside SSDs. Software inside SSDs is designed and optimized centering around four objectives, including managing address mapping, reducing write amplification, embracing device wearout, and improving speed performance. We have accordingly discussed major design options and discussed representative prior work under these four categories. We have also surveyed recent advancement on rethinking the design of high-level software stacks to best embrace and exploit SSDs. We have discussed prior work in three particular domains, including caching, file system, and databases. As SSDs are quickly becoming commodity products, we expect that future innovations will more and more center around those high-level software stacks. It is our hope that this survey paper will serve as a comprehensive reference for academic researchers and industrial practitioners to continue to advance this exciting and broad domain in the big data era.

Beyond NAND flash, a recent technology breakthrough is nonvolatile memory (NVM), represented by phase change memory (PCM) and spin-transfer-torque RAM (STT-RAM). NVM is interesting, because it carries a combination of properties of both memory and storage—it provides near-memory access speed, DRAM-like byte addressability, and storage-like persistence. In other words, NVM draws a blurry line between volatile memory and persistent storage. A fundamental question is how to integrate such an unconventional technology into our existing I/O stack. A natural consideration is to insert NVM as a new caching layer between DRAM and NAND flash. For example, leveraging the byte addressability, we may directly integrate NVM and DRAM together and expose a huge unified memory space to accommodate memory-intensive workloads. Similarly, leveraging its persistence property, NVM can be integrated into the storage system as an on-device cache (or buffer) or can be directly used as a high-speed block device, such as PMBD [20]. Such an approach is conservative but demands little change to the existing system and application designs. To fully exploit the NVM properties, more aggressive approaches can be adopted. An example is NVM-based file systems, such as BPFS [24], PMFS [25], and SCMFS [100]. These file systems are designed with special consideration on the unique properties of NVM. For example, PMFS supports direct memory mapping of file data to process address space without extra memory copy. An even more aggressive approach is to directly open persistence and byte addressability properties to applications, which allows programmers to make certain in-memory structures/objects persistent. However, this implies nontrivial burden to application developers, especially on handling data integrity issues upon system failures. Prior studies, such as Mnemosyne [89], CDDCS [88], and NV-heap [22], all try to provide a programming interface to help programmers.

As a disruptive technology, NVM could change many aspects of system and application designs along with other emerging technologies, such as NVM-Express [2], CAPI [3], NVLink [4], etc. The experience that we learned through adopting NAND flash in the past decade can also shed a light on a smooth transition to NVM in the future: When integrating an unconventional technology into the existing stack, we must carefully balance between two equally important goals, fully exploiting the new technology's potential and minimizing disruptions to the current ecosystem. Though challenging, we are optimistic that this process will open numerous research opportunities for researchers and practitioners. ■

Acknowledgements

The authors would like to thank the reviewers for constructive comments.

REFERENCES

[1] [Online]. Available: http://www.fusionio.com/PDFs/Fusion_ioDriveDuo_datasheet_v3.pdf

[2] [Online]. Available: http://www.nvmexpress.org/

[3] [Online]. Available: https://www-304.ibm.com/webapp/set2/sas/f/capi/home.html

[4] [Online]. Available: http://www.nvidia.com/object/nvlink.html

[5] (2016). [Online]. Available: https://www.facebook.com/notes/facebook-engineering/mcdipper-akey-value-cache-for-flash-storage/10151347090423920

[6] (2016). [Online]. Available: https://github.com/twitter/fatcache

[7] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," *ACM SIGPLAN Notices*, vol. 33, no. 11, pp. 81–91, 1998.

[8] Apple. (Mar. 22, 2016). *Mac Mini (Late 2012 and Later), iMac (Late 2012 and Later): About Fusion Drive*. https://support.apple.com/en-us/HT202574

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, pp. 53–64, 2012.

[10] A. Ben-Aroya and S. Toledo, "Competitive analysis of flash-memory algorithms," in *Proc. Annu. Eur. Symp. Algorithms*, 2006, pp. 100–111.

[11] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge, "Extreme Binning: Scalable, parallel deduplication for chunk-based file backup," in *Proc. IEEE Int. Symp. Modeling, Anal. Simulation Comput. Telecommun. Syst. (MASCOT)*, Sep. 2009, pp. 1–9.

[12] L.-P. Chang, "A hybrid approach to NAND-flash-based solid-state disks," *IEEE Trans. Comput.*, vol. 59, no. 10, pp. 1337–1349, Oct. 2010.

[13] L.-P. Chang and C.-D. Du, "Design and implementation of an efficient wear-leveling algorithm for solid-state-disk microcontrollers," *ACM Trans. Design Autom. Electron. Syst.*, vol. 15, no. 1, pp. 6:1–6:36, Dec. 2009.

[14] L.-P. Chang and T.-W. Kuo, "Real-time garbage collection for flash-memory storage systems of real-time embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 3, no. 4, pp. 837–863, Nov. 2004.

[15] F. Chen, B. Hou, and R. Lee, "Internal parallelism of flash memory-based solid-state drives," *ACM Trans. Storage*, vol. 12, no. 3, pp. 13:1–13:39, May 2016.

[16] F. Chen, D. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proc. Int. Joint Conf. Meas. Modeling Comput. Syst. (SIGMETRICS)*, 2009, pp. 181–192.

[17] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the best use of solid state drives in high performance storage systems," in *Proc. 25th ACM Int. Conf. Supercomput. (ICS)*, Tucson, AZ, USA, May/Jun. 2011, pp. 22–32.

[18] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit. (HPCA)*, San Antonio, TX, USA, Feb. 2011, pp. 266–277.

[19] 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 Storage Technol. (FAST)*, vol. 11. 2011, pp. 1–14.

[20] F. Chen, M. P. Mesnier, and S. Hahn, "A protected block device for persistent memory," in *Proc. IEEE 30th Int. Conf. Massive Storage Syst. Technol. (MSST)*, Santa Clara, CA, USA, Jun. 2014, pp. 1–12.

[21] M.-L. Chiang and R.-C. Chang, "Cleaning policies in mobile computers using flash memory," *J. Syst. Softw.*, vol. 48, no. 3, pp. 213–231, 1999.

[22] J. Coburn *et al.*, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proc. Archit. Support Program. Lang. Oper. Syst. (ASLPOS)*, Newport Beach, CA, USA, Mar. 2011, pp. 1–13.

[23] E. T. Cohen, "Why variable-size matters: Beyond page-based flash translation layers," in *Proc. Flash Memory Summit*, 2012.

[24] J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Principles (SOSP)*, 2009, pp. 133–146.

[25] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, and R. S. J. Jackson, "System software for persistent memory," in *Proc. Eur. Conf. Comput. Syst. (EuroSys)*, Amsterdam The Netherlands, Apr. 2014, p. 15.

[26] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 138–163, Jun. 2005.

[27] G. Graefe, "The five-minute rule 20 years later, and how flash memory changes the rules," in *Proc. 3rd Int. Workshop Data Manage. New Hardw. (DaMon)*, Beijing, China, Jun. 2007, pp. 1–29.

[28] 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.

[29] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing NAND flash-based SSDs," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2011, pp. 91–103.

[30] S. S. Hahn, S. Lee, and J. Kim, "SOS: Software-based out-of-order scheduling for high-performance NAND flash-based SSDs," in *Proc. IEEE Symp. Mass Storage Syst. Technol. (MSST)*, May 2013, pp. 1–5.

[31] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.

[32] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient identification of hot data for flash memory storage systems," *ACM Trans. Storage*, vol. 2, no. 1, pp. 22–40, Feb. 2006.

[33] J. W. Hsieh, H. Y. Lin, and D. L. Yang, "Multi-channel architecture-based FTL for reliable and high-performance SSD," *IEEE Trans. Comput.*, vol. 63, no. 12, pp. 3079–3091, Dec. 2014.

[34] J. Hu, H. Jiang, L. Tian, and L. Xu, "PUD-LRU: An erase-efficient write buffer management algorithm for flash memory SSD," in *Proc. IEEE Int. Symp. Modeling, Anal. Simulation Comput. Telecommun. Syst. (MASCOTS)*, Aug. 2010, pp. 69–78.

[35] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proc. Israeli Experim. Syst. Conf. (SYSTOR)*, 2009, pp. 10:1–10:9.

[36] 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. Int. Conf. Supercomput. (ICS)*, 2011, pp. 96–107.

[37] S.-M. Huang and L.-P. Chang, "Exploiting page correlations for write buffering in page-mapping multichannel SSDs," *ACM Trans. Embedded Comput. Syst.*, vol. 15, no. 1, pp. 12:1–12:25, 2016.

[38] S. Im and D. Shin, "Storage architecture and software support for SLC/MLC combined flash memory," in *Proc. ACM Symp. Appl. Comput. (SAC)*, 2009, pp. 1664–1669.

[39] J. Jeong, S. S. Hahn, S. Lee, and J. Kim, "Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling," in *Proc. USENIX Conf. File Storage technol. (FAST)*, 2014, pp. 61–74.

[40] S. Jiang, L. Zhang, X. Yuan, H. Hu, and Y. Chen, "S-FTL: An efficient address translation for flash memory by exploiting spatial locality," in *Proc. IEEE 27th Symp. Mass Storage Syst. Technol. (MSST)*, May 2011, pp. 1–12.

[41] X. Jimenez, D. Novo, and P. Ienne, "Wear unleveling: Improving NAND flash lifetime by balancing page endurance," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2014, pp. 47–59.

[42] X. Jin, S. Jung, and Y. H. Song, "Write-aware buffer management policy for performance and durability enhancement in NAND flash memory," *IEEE Trans. Consum. Electron.*, vol. 56, no. 4, pp. 2393–2399, Nov. 2010.

[43] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li, "DFS: A file system for virtualized flash storage," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, San Jose, CA, USA, Feb. 2010.

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

[45] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *Proc. USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, Jun. 2014.

[46] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory," in *Proc. ACM/IEEE Int. Conf. Embedded Softw. (EMSOFT)*, Oct. 2006, pp. 161–170.

[47] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proc. USENIX Winter*, 1995, pp. 155–164.

[48] D. Kim and S. Kang, "Dual region write buffering: Making large-scale nonvolatile buffer using small capacitor in SSD," in *Proc. ACM Symp. Appl. Comput. (SAC)*, 2015, pp. 2039–2046.

[49] G. Kim and D. Shin, "Performance analysis of SSD write using TRIM in NTFS and EXT4," in *Proc. Int. Conf. Comput. Sci. Converg. Inf. Technol. (ICCIT)*, Nov. 2011, pp. 422–423.

[50] H. Kim and S. Lee, "An effective flash memory manager for reliable flash memory space management," *IEICE Trans. Inf. Syst.*, vol. 85, no. 6, pp. 950–964, 2002.

[51] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for CompactFlash systems," *IEEE Trans. Consum. Electron.*, vol. 48, no. 2, pp. 366–375, May 2002.

[52] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, Santa Clara, CA, USA, Feb. 2015, pp. 273–286.

[53] S. Lee, T. Kim, K. Kim, and J. Kim, "Lifetime management of flash-based SSDs using recovery-aware dynamic throttling," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2012, pp. 327–340.

[54] S. Lee *et al.*, "Application-managed flash," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 339–353.

[55] S. Lee, D. Shin, Y. Kim, and J. Kim, "Exploiting sequential and temporal localities to improve performance of NAND flash-based SSDs," *ACM Trans. Storage*, vol. 12, no. 3, pp. 15:1–15:39, May 2016.

[56] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, p. 18, 2007.

[57] Y.-G. Lee, D. Jung, D. Kang, and J.-S. Kim, "μFTL: A memory-efficient flash translation layer supporting multiple mapping granularities," in *Proc. ACM Int. Conf. Embedded Softw. (EMSOFT)*, 2008, pp. 21–30.

[58] A. Leventhal, "Flash storage memory," *Commun. ACM*, vol. 51, pp. 47–51, Jul. 2008.

[59] J. Li, K. Zhao, X. Zhang, J. Ma, M. Zhao, and T. Zhang, "How much can data compressibility help to improve NAND flash memory lifetime?" in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 227–240.

[60] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "Sparse Indexing: Large scale, inline deduplication using sampling and locality," in *Proc. 7th USENIX Conf. File Storage Technol. (FAST)*, vol. 9. 2009, pp. 111–123.

[61] H.-Y. Lin and J.-W. Hsieh, "HLC: Software-based half-level-cell flash memory," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, 2015, pp. 936–941.

[62] T. Luo, R. Lee, M. P. Mesnier, F. Chen, and X. Zhang, "hStorage-DB: Heterogeneity-aware data management to exploit the full capability of hybrid storage systems," in *Proc. 38th ACM Int. Conf. Very Large Databases (VLDB)*, Istanbul, Turkey, Aug. 2012, pp. 1076–1087.

[63] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou, "S-CAVE: Effective SSD caching to improve virtual machine storage performance," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Edinburgh, Scotland, Sep. 2013, pp. 103–112.

[64] C. Manning. (2004). *YAFFS: Yet Another Flash File System*. [Online]. Available: http://www.aleph1.co.uk/yaffs

[65] B. Mao and S. Wu, "Exploiting request characteristics and internal parallelism to improve SSD performance," in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, 2015, pp. 447–450.

[66] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud, "Intel turbo memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems," *ACM Trans. Storage*, vol. 4, no. 2, pp. 4:1–4:24, May 2008.

[67] J. Menon and L. Stockmeyer, *An Age-Threshold Algorithm for Garbage Collection in Log-Structured Arrays and File Systems*. New York, NY, USA: Springer, 1998, pp. 119–132.

[68] I. Moulster. (Apr. 6, 2006). *SuperFetch, ReadyBoost and ReadyDrive: Some New Feature Names for You*. [Online]. Available: https://blogs.msdn.microsoft.com/ianm/2006/04/06/superfetch-readyboost-and-readydrive-some-new-feature-names-for-you/

[69] M. Murugan and D. H. C. Du, "Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead," in *Proc. IEEE Symp. Mass Storage Syst. Technol. (MSST)*, May 2011, pp. 1–12.

[70] M. Murugan and D. H. C. Du, "Hybrot: Towards improved performance in hybrid SLC-MLC devices," in *Proc. IEEE Int. Symp. Modeling Anal. Simulation Comput. Telecommun. Syst.*, Aug. 2012, pp. 481–484.

[71] Y. Oh, E. Lee, J. Choi, D. Lee, and S. H. Noh, "Hybrid solid state drives for improved performance and enhanced lifetime," in *Proc. IEEE Symp. Mass Storage Syst. Technol. (MSST)*, May 2013, pp. 1–5.

[72] 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 Program. Lang. Oper. Syst. (ASPLOS)*, Salt Lake City, UT, USA, Mar. 2014.

[73] Y. Pan, G. Dong, and T. Zhang, "Error rate-based wear-leveling for NAND flash memory at highly scaled technology nodes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 7, pp. 1350–1354, Jul. 2013.

[74] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim, "A reconfigurable FTL (Flash Translation Layer) architecture for NAND flash-based applications," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 4, pp. 38-1–38-23, Aug. 2008.

[75] V. Prabhakaran, T. L. Rodeheffeer, and L. Zhou, "Transactional flash," in *Proc. 8th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, San Diego, CA, USA, Dec. 2008.

[76] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "A two-level caching mechanism for demand-based page-level address mapping in NAND flash memory storage systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, Apr. 2011, pp. 157–166.

[77] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle, "Active disks for large-scale data processing," *Computer*, vol. 34, no. 6, pp. 68–74, Jun. 2001.

[78] O. Rodeh, "B-trees, shadowing, and clones," *ACM Trans. Storage*, vol. 3, no. 4, pp. 2-1–2-27, Feb. 2008.

[79] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Jun. 1992.

[80] S. Seshadri *et al.*, "Willow: A user-programmable SSD," in *Proc. 11th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2014, pp. 67–80.

[81] Z. Shen, F. Chen, Y. Jia, and Z. Shao, "DIDACache: A deep integration of device and application for flash based key-value caching," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, Santa Clara, CA, USA, Feb./Mar. 2017.

[82] L. Shi, J. Li, Q. Li, C. J. Xue, C. Yang, and X. Zhou, "A unified write buffer cache management scheme for flash memory," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 12, pp. 2779–2792, Dec. 2014.

[83] K. Smith, "Garbage collection," *Flash Memory Summit*, 2011.

[84] K. Smith, "Understanding SSD over-provisioning," *Flash Memory Summit*, Jan. 2012.

[85] D. Tiwari *et al.*, "Active Flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines," in *Proc. 11th USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 119–132.

[86] J. Tjioe, A. Blanco, T. Xie, and Y. Ouyang, "Making garbage collection wear conscious for flash SSD," in *Proc. IEEE Int. Conf. Netw., Archit. Storage (NAS)*, Jun. 2012. pp. 114–123.

[87] C. Ungureanu *et al.*, "HydraFS: A high-throughput file system for the hydrastor content-addressable storage system," in *Proc. 8th USENIX Conf. File Storage Technol. (FAST)*, vol. 10. 2010, pp. 225–239.

[88] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proc. 9th USENIX Conf. File Storage Technol. (FAST)*, San Jose, CA, USA, Feb. 2011, pp. 61–75.

[89] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proc. Archit. Support Programm. Lang. Oper. Syst. (ASLPOS)*, Newport Beach, CA, USA, Mar. 2011, pp. 91–104.

[90] P. Wang *et al.*, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *Proc. 9th Eur. Conf. Comput. Syst. (EuroSys)*, Amsterdam, The Netherlands, 2014, p. 16.

[91] W. Wang, W. Pan, T. Xie, and D. Zhou, "How many MLCs should impersonate SLCs to optimize SSD performance?" in *Proc. Int. Symp. Memory Syst. (MEMSYS)*, 2016, pp. 238–247.

[92] Y. Wang, D. Liu, M. Wang, Z. Qin, Z. Shao, and Y. Guan, "RNFTL: A reuse-aware NAND flash translation layer for flash memory," in *Proc. ACM SIGPLAN/SIGBED Conf. Lang., Compil., Tools Embedded Syst.*, 2010, pp. 163–172.

[93] Q. Wei, C. Chen, M. Xue, and J. Yang, "Z-MAP: A zone-based flash translation layer with workload classification for solid-state drive," *ACM Trans. Storage*, vol. 11, no. 1, pp. 4:1–4:33, Feb. 2015.

[94] Y.-J. Woo and J.-S. Kim, "Diversifying wear index for MLC NAND flash memory to extend the lifetime of SSDs," in *Proc. ACM Int. Conf. Embedded Softw. (EMSOFT)*, Sep. 2013, pp. 6:1–6:10.

[95] D. Woodhouse, *JFFS: The Journaling Flash File System*, 2001.

[96] C-H. Wu and T.-W. Kuo, "An adaptive two-level management for the flash translation layer in embedded systems," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2006, pp. 601–606.

[97] G. Wu, B. Eckart, and X. He, "BPAC: An adaptive write buffer management scheme for flash-based solid state drives," in *Proc. IEEE Symp. Mass Storage Syst. Technol. (MSST)*, May 2010, pp. 1–6.

[98] G. Wu and X. He, "ΔFTL: Improving SSD lifetime via exploiting content locality," in *Proc. ACM Eur. Conf. Comput. Syst. (EuroSys)*, 2012, pp. 253–266.

[99] M. Wu and W. Zwaenepoel, "eNVy: A nonvolatile main memory storage system," in *Proc. 4th Workshop Workstation Oper. Syst.*, Oct. 1993, pp. 116–118.

[100] X. Wu and A. L. Reddy, "SCMFS: A file system for storage class memory," in *Proc. Super Comput. (SC)*, Seattle, WA, Nov. 2011, p. 39.

[101] W. Xie, Y. Chen, and P. C. Roth, "Parallel-DFTL: A flash translation layer that exploits internal parallelism in solid state drives," in *Proc. IEEE Int. Conf. Netw., Archit. Storage (NAS)*, Aug. 2016, pp. 1–10.

[102] F. Yang, K. Dou, S. Chen, M. Hou, J.-U. Kang, and S. Cho, "Optimizing NoSQL DB on flash: A case study of RocksDB," in *Proc. IEEE Int. Conf. Ubiquitous Intell. Comput.*, Aug. 2015, pp. 1062–1069.

[103] M.-C. Yang, Y.-H. Chang, T.-W. Kuo, and F.-H. Chen, "Reducing data migration overheads of flash wear leveling in a

progressive way," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 5, pp. 1808–1820, May 2016.

[104] M.-C. Yang, Y.-H. Chang, C.-W. Tsao, and P.-C. Huang, "New ERA: New efficient reliability-aware wear leveling for endurance enhancement of flash storage devices," in *Proc. 50th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, May 2013, pp. 1–6.

[105] Y. Zhang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "De-indirection for flash-based SSDs with nameless writes," in *Proc. 10th USENIX Conf. File Storage Technol. (FAST)*, San Jose, CA, USA, Feb. 2012, p. 1.

[106] Y. Zhang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Removing the costs and retaining the benefits of flash-based SSD virtualization with FSDV," in *Proc. 31st Int. Conf. Massive Storage Syst. Technol.(MSST)*, Santa Clara, CA, USA, May 2015, pp. 1–7.

[107] Y. Zhou, F. Wu, P. Huang, X. He, C. Xie, and J. Zhou, "An efficient page-level FTL to optimize address translation in flash memory," in *Proc. Eur. Conf. Comput. Syst. (EuroSys)*, 2015, pp. 12:1–12:16.

[108] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. 6th USENIX Conf. File Storage Technol. (FAST)*, vol. 8. 2008, pp. 1–14.

ABOUT THE AUTHORS

Feng Chen (Member, IEEE) received the Ph.D. degree in computer science and engineering from The Ohio State University, Columbus, OH, USA, in 2010.

He is an Assistant Professor of Computer Science at Louisiana State University, Baton Rouge, LA, USA. Before joining LSU, he was a research scientist at Intel Labs, OR, USA. His research interests include operating systems, file and storage systems, and distributed systems.

Prof. Chen is a recipient of the Best Paper Award at the 25th ACM International Conference on Supercomputing in 2011 and the National Science Foundation (NSF) Faculty Early Career Development Award (CAREER) in 2015.

Tong Zhang (Senior Member, IEEE) received the Ph.D. degree in electrical engineering from the University of Minnesota, Minneapolis, MN, USA, in 2002.

Currently, he is a Professor in Electrical, Computer and Systems Engineering (ECSE) Department at Rensselaer Polytechnic Institute (RPI), Troy, NY, USA. He coauthored over 150 refereed papers in the broad areas of memory circuits and systems, VLSI signal processing, and computer architecture. He has graduated 15 Ph.D. students.

Prof. Zhang has served as an Associate Editor for the IEEE Transactions on Circuits and Systems—Part II: Express Letters, the IEEE Transactions on Signal Processing, *ACM Transactions on Storage*, and *Journal of Signal Processing Systems*. He was the Technical Program Co-Chair of the 2012 ACM Great Lakes Symposium on VLSI and the 2012 IEEE Workshop on Signal Processing Systems (SiPS), and the General Co-Chair of the 2013 ACM Great Lakes Symposium on VLSI.

Xiaodong Zhang (Fellow, IEEE) received the Ph.D. degree in computer science from the University of Colorado at Boulder, Boulder, CO, USA.

He is the Robert M. Critchfield Professor in Engineering and Chair of the Computer Science and Engineering Department at the Ohio State University, Columbus, OH, USA. His research interests focus on data management in computer and distributed systems.

Prof. Zhang received the Distinguished Engineering Alumni Award in 2011 from the University of Colorado at Boulder. He is a Fellow of the Association for Computing Machinery (ACM).