One Size Never Fits All: A Flexible Storage Interface for SSDs

Zhaoyan Shen*, Feng Chen†, Gala Yadgar‡, Zili Shao §
*Shandong University, †Louisiana State University, ‡Technion, §The Chinese University of Hong Kong

ABSTRACT

The rapid adoption of solid-state drives (SSDs) as a major storage component has been made possible thanks to their ability to export a standard block I/O interface to file system and application developers. Meanwhile, this high-level abstraction has been shown to limit the utilization of the devices and the performance of applications running on top of them. Indeed, many optimizations of performance-critical applications bypass the standard block interface and rely on low-level control over SSD internal processes. However, the need to directly manage the physical device significantly increases development complexity and cost, and reduces its portability. Thus, application developers must choose between two extreme options, either *easy development* or *optimal performance*, without a real possibility to balance between these two objectives.

To bridge this gap, we propose a *flexible storage interface* that exports the SSD hardware in three levels of abstraction: as a raw flash media with its low-level details, as a group of functions to manage flash capacity, or as a configurable block device. This multi-level abstraction allows developers to choose the degree in which they desire to control the flash hardware in a manner that best suits their applications' semantics and performance objectives.

We demonstrate the usability of this new model with *Prism-SSD*—a prototype of this interface as a user-level library on the Open-Channel SSD platform. We use each of the interface's three abstraction levels to modify the I/O module of three representative applications: a key-value cache system, a user-level file system, and a graph processing engine. Prism-SSD improves application performance by 5% to 27%, at varying development costs, between 200 and 3,500 lines of code.

I. INTRODUCTION

The complexity of today's hierarchical and distributed storage systems is hidden from the users by numerous layers of abstraction that make up the storage stack, including device drivers, file systems, databases, file sharing applications, etc. This abstraction is realized by well-defined interfaces that expose a standard set of functions between the layers of the stack. These standard interfaces facilitate portability of applications and other software-based components to different hardware platforms, backward compatibility and adoption of new hardware, and competitive development times.

At the same time, fixed interfaces prevent the components of the storage stack from sharing valuable information. This gap is particularly crucial for performance-critical systems, where application-specific information is exploited to maximize their utilization of the hardware they are built on. Many ad-hoc approaches for bypassing layers in the storage stack and for passing information between them have been proposed. Examples include pinning pages in memory according to the query execution plan [1], passing file and priority information to the storage server's cache manager [2] and disk scheduler [3], leveraging application hints for exploiting caching opportunities in databases [4], file systems [5], [6] and hadoop applications [7].

The wide adoption of solid-state drives (SSDs) has exposed well-known issues, such as 'log-on-log' [8], [9] and high tail latency [10]–[12], that result from the semantic gap caused by accessing SSDs via the standard block I/O interface. A recent trend directly opens the SSD hardware details to the applications [13]–[17]. Open-Channel SSD is one such representative and popular example [14].

With Open-Channel SSD, the physical layout details (e.g., channels, chips, and blocks) are directly exposed to applications, which manage them via direct access to the core flash operations—page-read, page-write, and block-erase. This low-level control allows applications to optimize their performance through a tight integration of software and hardware. Indeed, several studies showed how such direct access can be used to optimize key-value caches [16], key-value stores [18]–[20], and LSM-trees [15].

The drawback of this approach is that it introduces significant challenges into software development. For example, a strong expertise in SSD hardware is required for application developers; the development process becomes more complex, involving both software and hardware . As a result, application optimizations become ad-hoc and hardware dependent, with limited portability.

Currently, developers must choose between these two extreme usage modes, neither of which is ideal. They can adopt the easy-to-use block interface, but suffer the long-term consequences on their applications' performance, or directly control and optimize every aspect of their SSDs by taking on excessive development burden. In practice, many application types and development scenarios call for a finer-grained compromise. For example, a developer may wish to

The work described in this paper is partially supported by the grants from the State Key Program of National Natural Science Foundation of China No. 61533011, the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 15222315, GRF 15273616, GRF 15206617, GRF 15224918), Direct Grant for Research, the Chinese University of Hong Kong (Project No. 4055096), the U.S. National Science Foundation under Grants CCF-1453705 and CCF-1629291.

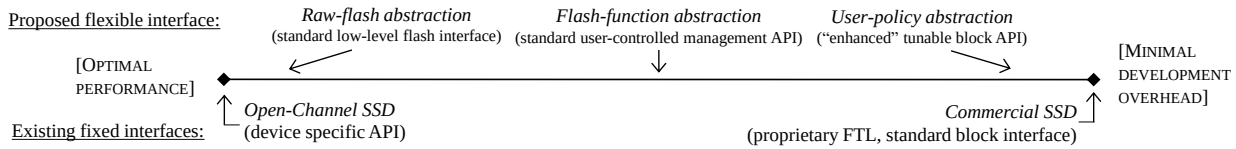

Fig. 1: Existing (bottom) and proposed (top) storage interfaces.

parallelize I/Os without being interested in explicit control of garbage collection (GC). These two extreme choices, depicted in the bottom of Figure 1, does not allow developers to strike the best balance between development cost and performance.

In this work, we advocate for a *flexible storage interface* for SSDs that will provide standard access to SSDs that expose their physical layout to applications, while supporting the versatile needs of applications and developers. Specifically, we will focus on three abstraction levels, two of which are close to the existing extreme usage modes, and one that introduces a novel tradeoff point between them. As a proof-of-concept, we implement *Prism-SSD* — a user-level library framework that exports the SSD via three levels of abstraction, depicted in Figure 1: (1) The *raw-flash level abstraction* directly exposes the low-level flash details, including physical structures and core flash operations; (2) The *flash-function level abstraction* presents the SSD as a group of flash management functions that can be scheduled and custom-defined by applications, such as GC, wear-leveling, etc.; (3) The *user-policy level abstraction* presents flash hardware as a block device that is configurable by selecting predefined high-level policies.

We implemented a Prism-SSD prototype on the Open-Channel SSD platform. We enhanced the I/O modules of three representative applications, using each of the three abstraction levels provided by the Prism-SSD library. We modified a key-value cache based on Twitter's Fatcache [21], a user-level log-structured file system based on Linux FUSE [22], and a graph computing engine based on GraphChi [23]. Together, these three use cases well demonstrate the flexibility and efficiency of our model. Our results show that Prism-SSD allows developers to flexibly choose the most suitable storage abstraction for optimizing their applications, at different tradeoff points between performance and development cost.

Our main contributions are as follows. (1) We introduce the concept of a flexible storage interface, that will allow developers to interact with flash-based SSDs in a standardized API with varying layers of abstraction. (2) We present a fully functional prototype of Prism-SSD on the real Open-Channel hardware platform. (3) We demonstrate the efficacy of our approach in three use cases, with a range of development costs and performance benefits.

II. Background

Generic flash SSDs. Conventional flash SSDs typically provide a generic block I/O interface to the host. An SSD controller is used to process I/O requests, and manage flash memory by issuing commands to the *flash memory controller*. A *Flash Translation Layer* (FTL) is usually implemented in the device firmware to manage flash memory and hide its complexities behind the *Logical Block Address (LBA)* interface. An FTL mainly consists of three components: an *address mapping table* translating logical addresses to flash physical pages, a *garbage collector (GC)* reclaiming invalid flash blocks, and a *wear-leveler* spreading the wear of flash blocks evenly across the flash memory chips. The details of FTL algorithms can be found in prior work [24]–[26].

Open-Channel SSDs. Open-Channel SSDs expose device-level details and raw flash operations directly to applications. The host is responsible for utilizing SSD resources with primitive functions through a simplified I/O stack. The following design principles of Open-Channel SSDs open up new prospects for SSD management. (1) Internal geometry details, such as the layout of channels, LUNs, and chips, are exposed to user-level applications. With this knowledge, applications can effectively organize their data and schedule accesses to fully exploit the raw flash performance. (2) Applications can directly operate the device hardware through the `ioctl` interface, allowing them to bypass the intermediate OS components, such as file systems. (3) FTL-level functions, such as address mapping, GC, and wear-leveling, are removed from the device firmware. Applications are responsible for dealing with flash physical constraints. For example, applications are responsible for allocating physical flash pages, ensuring a block being erased before it is overwritten. Thus, it can avoid issues, such as the 'log-on-log' problem [8], by directly issuing commands to erase physical blocks [16].

III. Design Goals

Open-Channel SSDs have been deployed with various kinds of applications, such as file systems [6], key-value stores and caches [15], [16], and virtualization environments [27], where they help achieve significant performance improvements. However, the prohibitive development overhead associated with Open-Channel SSDs hinders them from a much wider adoption, especially by applications that require special but only minor deviations from the standard block I/O interface (e.g., erase a block). This unrealized potential motivates our new model of a flexible storage interface for SSDs. We define our model with the following design goals.

- *Flexibility*: Applications should be able to flexibly choose the degree of control they require over the SSD operations. We achieve this goal by providing multiple levels of abstraction for programmers to choose from.
- *Generality*: The application design, and most of the interface's implementation, should be general and portable to different hardware and OS platforms. In this work, we show how this can be achieved by encapsulating the low-level flash accesses within a user-level flash monitor, and decoupling it from a standard user-perceived storage abstraction.

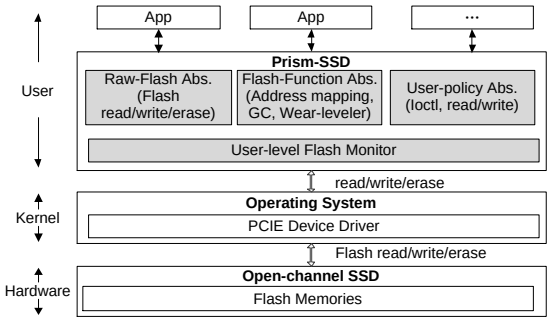

Fig. 2: Overview of Prism-SSD architecture.

• *Efficiency*: Applications should experience minimal overhead by using the interface. We achieve this goal in our prototype by implementing it in user-space, thus bypassing most of the kernel's I/O stack and the latencies it entails.

By following these goals, the definition of our flexible storage interface is designed to provide users a fine-grained control on the tradeoff between performance and development cost, while incurring minimal overhead.

IV. The Design of Prism-SSD

To achieve the above-said goals, we argue that much of the intelligence and complexity of the device driver should be moved into the user level, closer to the applications. This can provide a customizable, easy-to-use storage abstraction, and bypass multiple intermediate layers in the traditional I/O stack, such as the file system, generic block I/O interface, the scheduler, and the FTL in the firmware. We demonstrate the power of this novel approach with a proof-of-concept implementation, a design based on a user-level implementation of a flexible storage interface—*Prism-SSD*.

Prism-SSD consists of three main components: (1) *a specialized flash-based SSD hardware*, which exposes the physical details of flash memory, (2) a *user-level abstraction library*, which provides a comprehensive set of storage I/O stack abstractions, and (3) *applications*, which customize their software design with flash memory management at different integration degrees through the library's abstraction interface. In our prototype implementation, the library includes *three* sets of abstraction APIs with different degrees of hardware exposure to applications: a *raw-flash level*, a *flash-function level*, and a *user-policy level*. All accesses to an SSD managed by Prism-SSD have to go through the library. The three levels offer a set of choices for developers to optimize their application performance at different integration levels with distinct complexity and customizability. The library also includes a flash monitor running as a user-level module, which is responsible for allocating flash capacity to different applications sharing the same SSD hardware, and for isolating them from one another [27].

Figure 2 depicts the overall architecture of Prism-SSD, and Figure 3 presents the APIs in our prototype. Below, we focus on their details required for demonstrating our flexible storage interface approach. We note, however, that the specific three-layer design is only one of many possible realizations of our approach. Other designs may define different or additional abstraction layers to allow for finer-grained tradeoff points between development cost and performance.

A. The User-level Flash Monitor

At the bottom layer of the Prism-SSD library is a user-level flash monitor. Its main role is two-fold. First, as a storage capacity manager, it allocates the required flash capacity to applications and ensures space isolation. Second, it is responsible for sharable services, such as OPS allocation, global wear-leveling, bad block management, etc.

Applications request storage space through the user-level flash monitor. The monitor uses LUNs as the basic allocation unit[1] for satisfying applications' capacity requirements. Prism-SSD allocates LUNs in a round-robin fashion across channels. Consider an Open-Channel SSD with 12 channels, each providing access to 16 LUNs of 1GB. If an application requests a capacity of 24GB, the device manager will allocate two LUNs from each channel. The monitor also allocates an amount of over-provisioning space (OPS) as specified by the application. The developer can determine the size of OPS based on the application's properties. For write-intensive applications, the OPS can be set larger (e.g., 25%, similar to a typical high-end SSD); for read-intensive applications, the percentage can be smaller. The over-provisioning space is also allocated in units of LUNs. In the above example, for an OPS of 25%, six extra flash LUNs will be allocated to the application. As the flash monitor tracks the channels and LUNs allocated to each application, the flash capacity of different applications is completely isolated. Applications access the flash space allocated to them using the address format <channel_id, LUN_id, block, page>.

Workload patterns of different applications may vary considerably, causing the erase counts of flash blocks in different channels to vary as well. To handle uneven wear of the flash device, the design of Prism-SSD includes a global wear leveling module, which is based on FlashBlox [27]: Global wear leveling is applied in LUN granularity—it calculates the average erase count of each LUN to distinguish between 'hot' and 'cold' LUNs. If the difference in average erase counts exceeds a threshold, a hot LUN will be shuffled with a cold LUN, and their allocation status will be updated. This module is not implemented in our current prototype.

For bad block management, the flash monitor maintains a list of blocks that are detected faulty and marked ineligible, hiding them from applications.

B. Abstraction 1: Raw-Flash Level

The raw-flash level abstraction of Prism-SSD exposes the device geometry and allows applications to control the low-level flash hardware. Applications can directly operate on flash pages or blocks through page read/write and block erase commands. To use this level's API, application developers should be fully aware of the unique characteristics of flash memories, such as the out-of-place update constraint, to operate the device correctly. While the low-level details and

[1] A *channel* usually consists of multiple *LUNs*, which are the smallest parallelization units [28]. Each LUN includes multiple flash blocks.

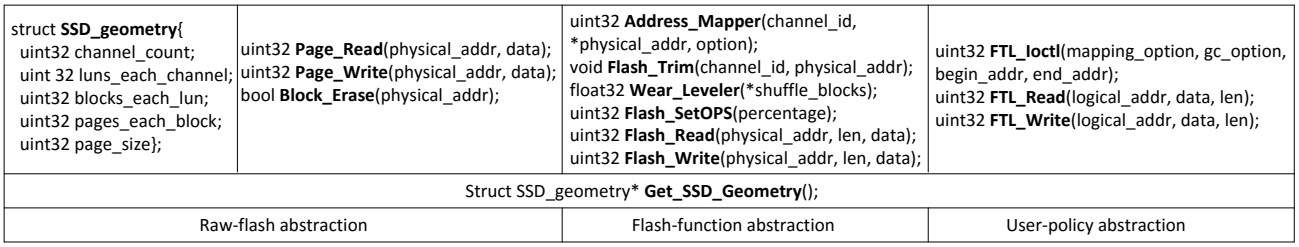

Fig. 3: APIs of Prism-SSD.

Algorithm IV.1 A GC process with the raw-flash abstraction.

```
1: CH_num ← 0;
2: while (GC) do //application determines GC status
3:     Select Block_i ∈ CH_num with the least valid data;
4:     while Page_j is valid in Block_i do
5:         page_read(Page_j, page_data);
6:         page_write(Page_new, page_data);
7:         //Page_new is selected by the application
8:     end while
9:     block_erase(Block_i);
10:    CH_num ← CH_num + 1;
11:    if CH_num ≥ CH_count then
12:        CH_num ← 0;
13:    end if
14: end while
```

control exposed by this layer are similar to those exposed by existing low-level interfaces, its interface functions are standard and decoupled from any specific SSD hardware, providing an additional degree of portability for developers.

With this abstraction level, typical FTL functions, such as address mapping, GC and wear-leveling, are not provided by the library. Whether to implement them or not depends on the application's requirements. The application should also be responsible for its own flash space allocation and management, and for integrating them with its software semantic. The library simply delivers function calls from applications to the device driver through the `ioctl` interface.

Figure 3 shows the APIs provided by the raw-flash level abstraction. `Get_SSD_Geometry` returns the SSD layout information to the application. The SSD layout is described by the number of channels, LUNs in each channel, blocks in each LUN, pages in each block, and the page size. This layout information is exposed to all abstraction layers via the same interface. Applications use the API functions, `Page_Read` and `Page_Write` to directly read and write flash physical pages, and `Block_Erase` to erase a specified block.

Algorithm IV.1 gives a garbage collection process implemented with the raw-flash abstraction. This process reclaims flash blocks in each channel in a round-robin manner, and the blocks with the least valid data are selected as victims. Valid pages in the victim blocks are copied to newly allocated physical pages, after which the victim blocks are erased.

The raw-flash abstraction gives applications full knowledge and direct control of the low-level flash device, at the cost of considerable development effort. The applications that will likely benefit most from this abstraction are those with special, regular, and well-defined access patterns.

C. Abstraction 2: Flash-Function Level

Our flash-function level abstraction models the flash storage as a collection of *core functions* for flash management, such as GC, wear-leveling, etc. Application developers can compose them and implement more sophisticated and complex management tasks. Thus, they can maintain a certain low-level control, while avoiding the need to handle other irrelevant details of the SSD hardware. Figure 3 shows the core APIs of the flash-function level. These APIs are used to divide the main components of flash management between the application and the library, as follows.

Space allocation. At this level, applications directly read and write flash physical addresses via functions `Flash_Read` and `Flash_Write`, while the library is responsible for erasing blocks and for allocating them to applications. The application requests physical blocks via `Address_Mapper`, specifying the channel in which the block should be allocated, and the mapping scheme (i.e., page-level or block-level) for that block. It then maps the physical address returned by the library to an application-managed logical address. This function call returns the amount of free space available for the application, allowing the application to invoke GC according to its needs.

Garbage collection (GC). At this level, the application is responsible for selecting the victim blocks for GC, and for identifying the valid data on these blocks. The granularity of the valid data is determined by the application and can be as small as a tuple of several hundred bytes. Thus, the application is also responsible for copying the valid data to a new location. By calling the `Flash_Trim` command, the application notifies the library that a block is ready to be erased, in the background, and reallocated.

Wear-leveling. At this level, the application manages the logical-to-physical block mapping while the library maintains the blocks' erase counts. Thus, wear leveling is triggered by the application and executed by the library, as follows. The application invokes the `Wear_Leveler` in a suitable time. The library identifies the hottest blocks and the coldest ones, and swaps the data written on them. It returns these block addresses via the "shuffle_block" parameter, as well as the maximum variance between erase counts of the application's allocated blocks. The application then updates its mapping of the two blocks, and potentially invokes another wear leveling operation according to its target variance.

OPS management. The application can dynamically determine the over-provisioning space it requires according to its current workload via `Flash_SetOPS`. The library reserves

Algorithm IV.2 Block allocation and GC with the flash-function abstraction.

```
1: FBN; //free block space
2: PBN; //physical address
3: LBN; //logical address
4: len ← 10 × Blocksize; //length
5: while len > 0 do
6:     CH_id ← choose a channel with the least workload;
7:     FBN ← Address_Mapper(CH_id, &PBN, "Block");
8:     LBN ← PBN; // map logical to physical
9:     //allocate physical block in channel CH_id
10:    if FBN < GC_Threshold then
11:        //Free space is under a GC threshold
12:        APP_GC(CH_id);
13:    end if
14:    len ← len − 1;
15: end while
16: function VOID APP_GC(CH_id)
17:    while FBN < GC_Threshold do
18:        PBN_victim ← victim block in CH_id;
19:        //select block by "Greedy", "LRU", etc.
20:        //copy valid data from the victim block elsewhere
21:        Flash_Trim(CH_id, PBN_victim);
22:    end while
23: end function
```

the specified OPS for this application. The library cannot provide the requested OPS if too many blocks are currently mapped by the application. In this case, the application must first release sufficient flash space.

Algorithm IV.2 shows an example of block allocation and GC implemented with the flash-function level. In this simple example, the application requests 10 flash blocks in an idle channel by repeatedly calling the address mapping function with block-level address mapping ("Block"). This function call returns the number of free blocks currently available in this channel. If the available free space is below a predefined threshold, the application triggers an application-controlled background GC process in this channel. The GC process selects a victim block, copies its valid data elsewhere, and releases this block for erasure by the library.

The flash-function level abstraction exports basic flash functions that application developers can use to configure different flash-management policies and to invoke them at the most suitable timing according to their current workloads. At the same time, it hides the low-level device details, such as LUNs and erase counts, from the application level. This abstraction level is suitable for applications that can leverage their software semantics for specific optimizations but are not willing to handle the low-level management details. To the best of our knowledge, it is the first general-purpose implementation to provide this fine-grained tradeoff between application management and ease-of development.

D. Abstraction 3: User-policy Level

The user-policy level abstraction hides all the flash related management operations from users, allowing them to manage the SSD as a simple block device. To some extent, it can be regarded as a user-level FTL that handles address mapping, GC, wear-leveling, etc. This abstraction level is designed to provide the highest generality for SSDs. However, unlike conventional device-level FTLs, this "FTL" runs as part of the user-level library and is configurable, allowing applications to select their preferred policies for managing flash space.

Algorithm IV.3 Example code of application initialization with the user-policy level abstraction.

```
1: start_addr ← 0;
2: split_addr ← 10GB;
3: end_addr ← 100GB;
4: FTL_Ioctl("Block", "FIFO", start_addr, split_addr);
5: // We can now read/write addresses between 0 and 10GB
6: FTL_Ioctl("Page", "Greedy", split_addr, end_addr);
7: // We can now read/write addresses between 10GB and 100GB
```

The applications use their semantic knowledge about the data usage patterns to choose the best policies for optimizing their specific objectives. Thus, these configuration parameters serve as application 'hints' to the FTL. Meanwhile, the full device layout information is exposed to applications, allowing them to optimize the size of their data structures or level of I/O parallelism for the underlying device.

Figure 3 lists the APIs provided in the user-policy level abstraction. Logical addresses are read and written via the `FTL_Read` and `FTL_Write` block I/O interfaces. Applications configure the key flash management policies, address mapping and GC, via the `FTL_Ioctl` function. The same policies implemented in the flash-function level (see Section IV-C) are available for selection.

Algorithm IV.3 presents an example code of an application initialization progress with the user-policy level abstraction. In this example, the application divides its flash logical space into two parts. The first part is configured with block-level address mapping and greedy garbage collection. The second part is configured with page-level address mapping and FIFO-based garbage collection. The application uses `Flash_Write` and `Flash_Read` to access logical addresses within each of these logical partitions.

The user-policy level abstraction is similar to existing host-level FTLs. However, it is managed as part of a general-purpose user-level library which also exposes additional abstraction layers. Furthermore, it allows application developers to leverage their semantic knowledge to configure the FTL policies. This abstraction level requires the lowest integration overhead, which makes it suitable for applications that only demand certain hardware/software cooperation through a configurable interface, but are sensitive to development cost.

V. Implementation and Prototype System

We have built a prototype of Prism-SSD on the Open-Channel SSD hardware platform manufactured by Memblaze [29]. This PCI-E based SSD contains 12 channels, each of which connects to two Toshiba 19nm MLC flash chips. Each chip consists of two planes and has a capacity of 66GB. The SSD exports its physical space to the upper level as one volume, with access to 192 LUNs. The 192 LUNs are evenly mapped to the 12 channels in a channel-by-channel manner. That is, channel #0 contains LUNs 0-15, channel #1 contains LUNs 16-31, and so on. Thus, the physical mapping of flash memory LUNs to channels is known.

This interface is different from that of Open-Channel SSDs used in other studies, which exports flash space as 44 individual volumes [13]. The hardware used in our prototype allows the upper level to directly access raw flash memory via the `ioctl` interface, by specifying the LUN ID, block ID, or page ID in the commands sent to the device command queue. Standard FTL-level functions, such as address mapping and GC, are not provided. In Prism-SSD, they are implemented in the library. The user-level flash monitor is responsible for conveying the I/O operations to the device driver via `ioctl`.

Our prototype implements the user-level flash monitor and the three abstraction levels, accounting for 4,460 lines of C code. Specifically, the user-level flash monitor module accounts for 560 lines, the raw-flash abstraction for 380 lines, and the flash-function abstraction and the user-policy abstraction for 2,580 and 940 lines, respectively. We deployed our prototype on a Linux workstation with an Intel i7-5820K 3.3GHZ processor and 16GB memory. We use Ubuntu 14.04 with Linux kernel 3.17.8 as our operating system.

VI. Case Studies

The Prism-SSD model offers a powerful tool for developers to optimize their performance with SSDs. Choosing the abstraction level that best suits their needs, application developers can integrate their software design with the hardware management at the right balance between performance and development cost.

In this section, we demonstrate the versatility of this approach in typical software development scenarios, mainly from the perspective of developers. We carefully selected three major applications as our case studies: a key-value cache system based on Twitter's Fatcache (Section VI-A), a user-level log-structured file system based on Linux FUSE (Section VI-B), and a graph computing engine based on GraphChi (Section VI-C). Due to space constraint, we use key-value caching as our main case study and the other two as examples demonstrating the applicability of Prism-SSD.

A. Case 1: In-flash Key-value Caching

Background and challenges. Flash-based key-value cache systems, such as Facebook's McDipper [30] and Twitter's Fatcache [21], are becoming increasingly common in industry. These caches typically run on commercial flash-based SSDs and adopt a slab-based allocation scheme, similar to Memcached [31], to manage key-value pairs. For example, Fatcache divides the SSD space into large `slabs` (e.g., 1MB), each dedicated to a different range of value sizes. Slabs are further separated into slots, each is used to store one key-value item. An in-memory hash table is used to record the mapping between key-value items and slabs.

The design of flash-based key-value caches is typically made "aware" of the underlying SSD to certain extent and maintains several flash-friendly properties. The SSD is treated as a log-structured object store, and key-value updates are implemented as out-of-place updates. In addition, I/O operations are issued in large units of entire slabs. This is accomplished by maintaining buffering small items in memory and flushing them to the underlying SSD in bulk. Finally, the cache is managed with coarse granularity, evicting entire slabs when the cache is full.

Despite this simple "flash-aware" design, the decoupled implementation of application-level cache management and device-level FTL results in redundant mechanisms that significantly degrade the cache's utilization: (1) redundant mapping between the in-memory hash table that maps key-values to slabs and the FTL that maps logical addresses to physical flash pages, (2) redundant garbage collection between the cache's slab eviction and the device-level GC that reclaims flash blocks occupied by invalid pages, and (3) redundant over-overprovisioning, allocated both at the cache management module and the FTL.

A recent study addressed these redundancies in the design of DIDACache [16]: a modified version of Fatcache implemented on top of Open-Channel SSD. DIDACache directly drives the SSD hardware while exploiting the semantic knowledge of Fatcache, augmented with the following major components. (1) a slab/block management module, which directly translates slabs into one or more blocks; (2) a unified mapping module, which records the mapping of key-value items to their physical flash locations; (3) an integrated garbage collection module, which reclaims flash space occupied by obsolete key-value items; and (4) a dynamic over-provisioning space (OPS) management module, which dynamically adjusts the OPS size based on a queuing-theory based model.

The full details of the DIDACache policies appear in the original study [16]. Its implementation based on Fatcache required adding 2,100 lines of code in total , and is tightly coupled with Open-Channel SSD's architecture and API. Below, we demonstrate how this development effort and dependency can be avoided with Prism-SSD, allowing the developer to choose from several tradeoff points between the cache's performance and the development effort.

Optimizing Fatcache with Prism-SSD. The Prism-SSD library facilitates three different approaches.

- *Deep Integration*: Using the raw-flash level abstraction provided by Prism-SSD, we allow the key-value cache manager to fully exploit the semantic knowledge of Fatcache and directly drive the SSD hardware. To that end, we have augmented the key-value cache manager as described in DIDACache [16], with the four major components described above. As a low-level integration, our implementation uses the library's basic APIs and accounts for 1,450 lines of code. As the library provides a standard API over the hardware, it is more portable than DIDACache to different platforms.
- *Function-level Integration*: The second implementation, based on the flash-function level abstraction, allows the key-value cache manager to design cache-friendly policies without managing all low-level details.

In contrast to the raw-level approach, the four major components in this implementation are as follows: (1) A slab to block mapping module. At the function level, the application can still see and manage the physical flash blocks

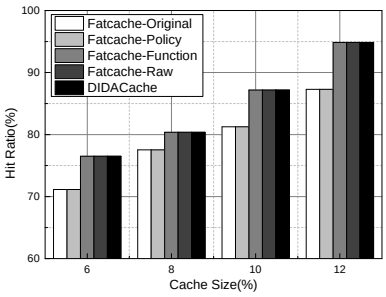

Fig. 4: Hit ratio vs. cache size.

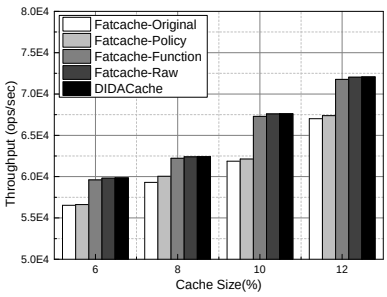

Fig. 5: Throughput vs. cache size.

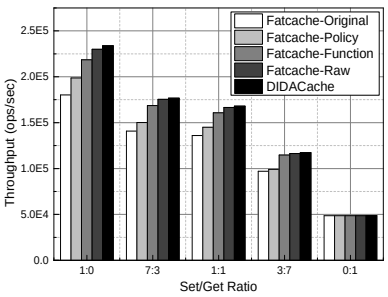

Fig. 6: Throughput vs. `Set/Get` ratio.

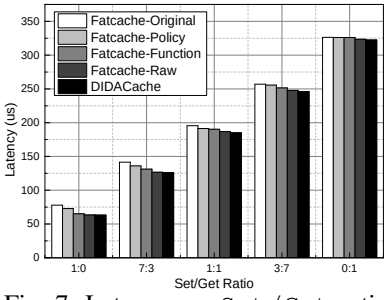

Fig. 7: Latency vs. `Set/Get` ratio.

TABLE I: Garbage collection overhead.

GC Scheme	Key-values	Flash Pages	Erase Counts
Fatcache-Original	13.27 GB	7.15 GB	8,540
Fatcache-Policy	13.27 GB	0	7,620
Fatcache-Function	3.63 GB	0	6,017
Fatcache-Raw	3.49 GB	N/A	5,994
DIDACache	3.45 GB	N/A	5,985

via the library APIs. Thus, it is responsible for the mapping between slabs and flash physical blocks; (2) A hash-key-to-slab mapping module. The key-value cache manager also records the mapping of key-value items to their slab locations, which is identical to the stock Fatcache implementation. (3) A garbage collection module. The key-value cache reclaims slab space occupied by obsolete (deleted or updated) key-value items. The flash physical blocks are invalidated and recycled via the library API; (4) A dynamic OPS management module, which estimates the preferred OPS based on a queuing theory based model. Note that at this level, the slab-to-flash-block mapping is still maintained by the application, but the block allocation, reclamation, and status are maintained by the library. This implementation consists of 860 lines of code.

• *Light Integration*: For comparison, we also implemented a light-weight optimization for Fatcache by using the user-policy level abstraction. In this implementation, the key-value cache manager is nearly identical to the stock Fatcache. Our implementation only replaces the device initialization process with the library APIs. The change requires 210 lines of code.

Implementation and Evaluation. Our implementation is based on Twitter's Fatcache [21]. For fair comparison, we added non-blocking slab allocation and eviction to the stock Fatcache. We use this version as our baseline and denote it as "Fatcache-Original". We refer to our implementation with the raw-flash level, flash-function level, and user-policy level abstractions as "Fatcache-Raw", "Fatcache-Function", and "Fatcache-Policy", respectively. For comparison, we run Fatcache-Original on a commercial PCI-E SSD, which has the same hardware as the Open-Channel SSD. We also show the results of DIDACache [16], denoted as "DIDACache", which directly integrates the hardware management into the Fatcache application, representing the ideal case.

• *Overall performance*. We first evaluate the key-value cache system in a simulated production data-center environment. This setup includes a front-end client, a key-value cache server, and an MySQL database in the backend. The key-value workload is generated using a model based on real Facebook workloads [32], [33], which is also used in prior work [16].

Figure 4 shows the hit ratios of the five cache systems with cache sizes of 6%–12% of the data set size. As the cache size increases, the hit ratio of all schemes improves significantly. Fatcache-Original and Fatcache-Policy have the same hit ratio because they both reserve 25% flash capacity as static OPS. In contrast, DIDACache, Fatcache-Raw, and Fatcache-Function have a higher hit ratio thanks to their adaptive OPS policy, which adaptively tunes the reserved space according to the workload, saving more space for caching. This extends the available cache space to accommodate more cache data. As a result, they outperform Fatcache-Original and Fatcache-Policy substantially: their hit ratios range between 76.5% and 94.8%, while those of Fatcache-Original and Fatcache-Policy are between 71.1% and 87.3%.

Figure 5 shows the throughput, i.e., the number of operations per second (ops/sec). We can see that as the cache size increases from 6% to 12%, the throughput of all the four schemes improves significantly, due to the improved cache hit ratio. Fatcache-Raw has the highest throughput, and Fatcache-Function is slightly lower. With a cache size of 10% of the data set (about 25GB), Fatcache-Raw outperforms Fatcache-Original, Fatcache-Function, and Fatcache-Policy by 9.2%, 0.4%, and 8.8%, respectively.

• *Cache server performance*. In our next set of experiments, we study the performance details of the cache server. We first populate the cache server with 25GB key-value items, and then directly issue `Set` and `Get` operations to the cache server. Figure 6 shows the throughput of the five cache systems with different `Set`/`Get` ratios. Fatcache-Raw

achieves the highest throughput across the board. Fatcache-Original achieves the lowest throughput. With 100% `Set` operations, the throughput of Fatcache-Raw is 27.6% higher than that of Fatcache-Original, 5.2% higher than that of Fatcache-Function, and 15.5% higher than that of Fatcache-Policy. The performance gain of Fatcache-Raw is mainly due to its unified slab management policy and the integrated application-driven GC policy, and the better use of the SSD's internal parallelism. Fatcache-Raw achieves almost the same performance as DIDACache. The throughput of Fatcache-Raw is only 1.7% lower than that of DIDACache in the worst case, which also demonstrates that the overhead of the Prism-SSD library is negligible compared to its benefits.

As we expected, the performances of Fatcache-Function and Fatcache-Policy are lower than that of Fatcache-Raw. The performance of Fatcache-Function is slightly lower than that of Fatcache-Raw. This is because although Fatcache-Function cannot operate with full low-level controls, it can still integrate the cache semantics within flash management, such as the GC process. Fatcache-Policy outperforms Fatcache-Original by 10.2%, due to its simplified I/O stack and block-level mapping, which reduces the overhead. As the portion of `Get` operations increases, the raw flash read latency becomes the main bottleneck, and this performance gain decreases.

Figure 7 shows the average latency of the cache systems with different `Set`/`Get` ratios. Fatcache-Original suffers the highest latency, while Fatcache-Raw, implemented with Prism-SSD, has the lowest latency. For example, with 100% `Set` operations, Fatcache-Raw reduces the average latency of Fatcache-Original, Fatcache-Function, and Fatcache-Policy by 22.9%, 2.8% and 12.1%, respectively.

• *Effect of garbage collection*. We also evaluate the effect of optimized GC on the flash erase counts, which directly affect the device lifetime. We configure the available SSD size to 30GB, and preload it with 25GB data. We then issue 140M `Set` operations following the Normal distribution, writing approximately 50GB of logical data. To retrieve the erase counts of Fatcache-Original, which runs on a commercial SSD, we collect its I/O trace and replay it with the widely used SSD simulator from Microsoft Research [34]. Table I shows the GC overhead in terms of valid data copies (key-values and flash pages) and block erases of the four schemes.

Fatcache-Original suffers from the highest GC overhead. It uses the greedy GC policy and the SSD hardware uses page-level mapping. As a result, blocks selected for erasure store a mix of valid and invalid pages, incurring flash page copies by the device-level GC. In contrast, the block-level mapping used by Fatcache-Function and Fatcache-Policy maps each slab directly to one flash block, thus eliminating all page copies caused by the device-level GC. By aggressively evicting valid clean items as part of the cache management policy, Fatcache-Function, Fatcache-Raw, and DIDACache further leverage application semantics to reduce the key-value copies to only 3.63 GB, 3.49 GB and 3.45 GB, respectively.

We further perform experiments to evaluate the GC latencies of these three schemes, and we find that the GC overheads of Fatcache-Raw and Fatcache-Function are basically the same. For Fatcache-Raw and Fatcache-Function, 88% and 86.2% percent of the GC invocations finish in less than 100ms, respectively. Fatcache-Policy is more affected by the GC due to the lack of deep optimization, and with 84% of the GC invocations finish in 100-1000ms.

TABLE II: File system GC overhead.

File system	File copy	Flash copy	Erase
ULFS-SSD	9.82GB	7.24GB	6,594
ULFS-Prism	9.82GB	N/A	5,280
MIT-XMP	N/A	9.37GB	5,429

This case study demonstrates the effectiveness of Prism-SSD by comparing four implementations of an optimized in-flash key-value caching. With the raw-flash abstraction, the developer can tightly control the low-level flash operations and optimize flash physical layout. With the flash-function abstraction, the software integrates its software semantics into hardware management without handling low-level details, and the performance can be close to the raw-flash implementation. With the user-policy abstraction, the application achieves noteworthy performance gains with minimal development overhead (210 lines of code). This successfully demonstrates the flexibility of our proposed storage interface.

B. Case 2: Log-structured File System

User-space file systems are often used to prototype and evaluate new approaches for file system design, and to develop complex systems that are difficult to maintain in kernel-space. Although user-space file systems achieve lower performance than that of their kernel-level counterparts, they are easier to develop and maintain, and enjoy better portability [40].

Log-structured file systems achieve high write throughput. A log-structured file system divides its space into equal-sized segments and writes its data and metadata sequentially to segments. Such log-structured design is expected to be flash-friendly. However, an oblivious log-structured file system running atop an SSD's log-structured FTL results in inefficient storage management [6], [8]. Specifically, the GC logic is duplicated without coordination, writes may not be properly aligned to segment boundaries, and semantic knowledge about data validity is unavailable at the FTL.

This inefficiency can be addressed by controlling the allocation of flash blocks to segments. Further performance improvement can be gained by explicit utilization of the device's channels. However, to achieve these benefits while still enjoying the relatively low development cost of user-level file systems, developers should avoid low-level details such as address mapping and GC. Such flexibility is provided by Prism-SSD's flash-function abstraction, which we use to build ULFS-Prism—a user-level log-structured file system.

ULFS-Prism directly allocates flash physical blocks to files. It maintains only the block-to-file mapping. Similarly, ULFS-Prism triggers greedy GC policy when the number of free blocks drops below a threshold, using the library's implementation of the greedy scheme. ULFS-Prism implements the channel-level parallelism and load balancing explicitly, by utilizing the channel information provided by

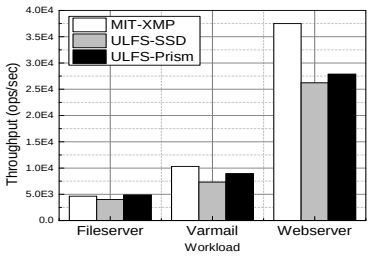

Fig. 8: Performance evaluation.

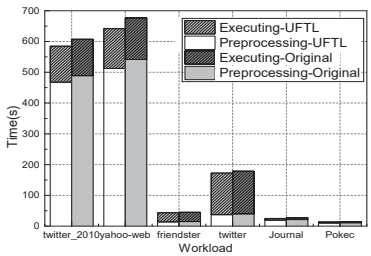

Fig. 9: Pagerank performance.

TABLE III: Graph workloads.

Graph Name	Nodes	Edges	Size
Twitter2010 [35]	41.7 m	1.4 b	26.2GB
Yahooweb [36]	1.4 b	6.6 b	50GB
Friendster [37]	6.6 m	1.8 b	211MB
Twitter [38]	81,306	1.8 m	44MB
LiveJournal [37]	4.0 m	34.7 m	1.1GB
Soc-Pokec [39]	1.6 m	30.6 m	404MB

the function-level abstraction. It maintains a queue for each channel, and counts the read/write/erase operations in each queue. A similar scheme was implemented in ParaFS [6] as a kernel-level file system with a specialized device-level FTL.

We also implemented a user-level log-structured file system, *ULFS-SSD*, and ran on a commercial PCI-E SSD, which has the same hardware as our Open-Channel SSD. For performance reference, we compare both log-structured file systems to MIT-XMP—a user level file system implemented as a FUSE wrapper for the host Ext4 file system [41] that ran on the commercial SSD.

In our first experiment, we use Filebench [42] to compare the results of the three file systems with three workloads, namely fileserver, webserver, and varmail. Figure 8 shows the throughput (operations per second, ops/sec) of the three file system implementations. The throughput of the two log-structured file systems is of a similar order of magnitude with MIT-XMP. ULFS-Prism outperforms ULFS-SSD in all three workloads, thanks to the cooperation between the hardware and software. Its throughput is 21.5% higher than that of ULFS-SSD on the varmail workload.

Table II shows the erase counts and valid data copied—file copies in the file system level and flash page copies in the device level—of each file system. ULFS-Prism and ULFS-SSD incur the same amount of file copies, but ULFS-Prism does not incur any flash page copies, because it does not require any additional device-level GC. However, with ULFS-SSD, the device-level GC may choose segments that still include valid data as victims blocks. MIT-XMP performs in-place updates at the file-system level, but incurs high GC cost at the device level.

This use case demonstrates that the flash-function level abstraction allows developers to quickly optimize their software design with relatively lower development cost. ULFS-SSD was implemented from scratch with 2,880 lines of code, and ULFS-Prism required only 660 more lines for integrating its flash management. This medium-level development effort is well paid off for a 21.5% speedup.

C. Case 3: Graph Computing Engine

Graph data processing is essential in big data systems. Graph computing platforms play an important role in analyzing massive graph data and extracting valuable information, such as social, biological, healthcare, information and cyber-physical systems. Many distributed graph processing systems like Pregel [43], GraphLab [44], PowerGraph [45], GraphX [46], are proposed to handle large scale graphs. These platforms adopt in-memory processing model, which is costly and difficult to scale. To address this issue, external memory graph processing systems are proposed, such as GraphChi [23], X-Stream [47] and GridGraph [48]. In this use case, we have enhanced a popular graph computing platform, GraphChi, with the Prism-SSD library.

We modify GraphChi with Prism-SSD using the user-policy level abstraction, as a showcase of quick, light-weight integration. In the initialization process, we divide the allocated logical space into two parts, and use one to store the shard data, and the other to store the results. We divide the data of each shard into block-sized segments (instead of files), and record the mapping between shards, intervals, and segments. We configure the logical space for shards with block-level mapping. The GC policy is irrelevant because this data is never updated. Similarly, we divide the result data into segments and record their mapping information in the application. We configure the logical space for result data with block-level mapping and greedy GC.

We compare the original GraphChi platform to our optimized implementation by running the "pagerank" algorithm on the graphs shown in Table III. Figure 9 shows the total run time of both GraphChi versions on each graph, divided into preprocessing time and execution time. In general, our optimized implementation outperforms the stock GraphChi in both preprocessing and execution steps across the board. For example, on the *Soc-Journal* graph, the optimized GraphChi reduces the preprocessing and execution times of the original platform by 5.2% and 7.6%, respectively, resulting in an overall 5.7% reduction.

This performance improvement is limited compared to our previous use cases. This is mainly due to the highly optimized nature of the original GraphChi: its I/O stack is simplified and its access pattern has been carefully optimized for SSDs. Also, I/O is not a major bottleneck, and optimizing it does not have a major impact on overall runtime. Nevertheless, we still were able to noticeably reduce this run time with a small development effort with 490 lines of code.

Summary. Table IV summarizes the development cost and main characteristics of our three use cases. These cases clearly demonstrate the flexibility and versatility provided by Prism-SSD—applications built with the raw-flash abstraction require the most development effort, while the user-policy abstraction requires the least code adaptation. With Prism-SSD, developers can choose how to integrate application software with low-level hardware management, according to their design goals.

TABLE IV: Use case summary.

Application	Level	Code Lines	Library Services	Application Responsibilities
Key-value caching	Raw-flash	1,450	Transfer to flash operations	Slab-to-block mapping, block allocation, garbage collection, OPS management
	Flash-function	860	Block allocation, Wear leveling, Asynchronous block erase	Slab-to-block mapping, Dynamic OPS, Valid data copies
	User-Policy	210	Wear leveling, garbage collection, block allocation, block mapping	Slab allocation, item-to-slab mapping
User-level LFS	Flash-function	(2,880+) 660	Wear leveling, block erasure, block allocation, block mapping	File-to-segment mapping, segment-based garbage collection, load balancing
Graph computing	User-Policy	490	Wear leveling, garbage collection, block allocation, block mapping	Shard-to-segment mapping, logical space partitioning

VII. Discussion

We have demonstrated the applicability and performance implications of Prism-SSD on several representative use cases. Here, we discuss additional aspects of our proposed interface.

Flexible extension. The flexibility of our storage interface can be easily extended to include more than just three abstraction levels. For example, the raw-flash level abstraction can be extended to develop and export a key-value set/get interface. The flash-function level can be extended to support asynchronous I/O operations by adding a scheduling algorithm for read, write and GC operations. Similarly, the user-policy level can be extended to implement a "container" abstraction for dividing the flash space into several partitions, facilitating separation of data according to lifetime or access frequency.

Alternative implementations. Our user-space implementation brings several performance benefits, as we have demonstrated above. Nevertheless, existing kernel-level frameworks can be leveraged to export the flexible storage interface to application developers. The LightNVM subsystem is an obvious candidate for this task [14]. The current implementation exposes three abstractions levels which can be classified into two categories: the "NVMe Device Driver" and the "LightNVM Subsystem" both require low-level management and control of all the aspects of the flash-based device, while the "High-level I/O Interface" exposes a standard block I/O interface to applications. These building blocks can be used to export a wider range of abstraction layers, such as those provided by PrismSSD, as well as additional layers. The performance implications of a kernel-space implementation of the entire interface remain subject for future work.

Hardware development: From the perspective of hardware vendors, a flexible storage interface in the form of a user-level library is a powerful tool for reducing development costs and the time-to-market. Moving complex internal firmware into the library layer will allow hardware vendors to quickly roll out new features in the form of library updates, and to accelerate the development cycle thanks to reduced coding, debugging, and testing requirements at the user level. These advantages are particularly appealing with the increasing complexity of hardware storage devices. Furthermore, hardware vendors can easily offer custom-built hardware/software solutions addressing various applications' requirements. Such solutions are currently prohibitive in terms of development time and costs. Combined, these advantages offer hardware vendors the means to build and own a complete vertical stack to closely connect with applications, creating more business opportunities in a new model.

VIII. Related Work

Flash SSDs have been extensively studied and optimized in the last decades. Besides the works mentioned previously, we focus here on the works related to the storage interface and their integration with file systems and applications.

Several recent works have been proposed to expose some or all of the internal flash details to applications. The Open-Channel SSD used in our implementation is one such example. Another example is SDF [13], which exposes the channels in commodity SSD hardware to the software, allowing it to fully utilize the device's raw bandwidth and storage capacity. FlashBlox [27], based on Open-Channel SSD, utilizes flash parallelism to improve isolation between applications. It runs them on dedicated channels and dies, and balances wear within and across different applications.

Other designs, implemented on customized SSDs or FPGAs, follow a similar approach. ParaFS [6] exposes device physical information to the file system, which in turn exploits its internal parallelism and coordinates the GC processes to minimize its overhead. AMF [7] provides a new out-of-place block I/O interface, reducing flash management overhead and pushing management responsibilities to the applications.

While the semantics of the interfaces exported by these systems vary, they all export a fixed interface of the device to the application level. However, as we have demonstrated in our use cases and discussion, many applications can benefit from a flexible interface that will allow developers to balance their performance and development cost. At the same time, we believe that the designs in these works can be implemented and made portable with Prism-SSD.

IX. Conclusion

In this paper, we presented a flexible storage interface and prototype implementation, Prism-SSD, which exports SSDs to applications in three abstraction levels. This interface allows developers to choose how tightly they want to integrate flash management into their application, providing more than just the two extreme options with current fixed interfaces. We demonstrated the usability of our model by comparing application performance improvement and development cost of three representative use cases. Our evaluation results reveal potential optimization opportunities that are facilitated by our model in a wide range of applications.

REFERENCES

[1] X. Ouyang, N. S. Islam, R. Rajachandrasekar, J. Jose, M. Luo, H. Wang, and D. K. Panda, "SSD-assisted hybrid memory to accelerate memcached over high performance networks," in *International Conference on Parallel Processing (ICPP'12)*, 2012.

[2] G. Yadgar, M. Factor, K. Li, and A. Schuster, "Management of multilevel, multiclient cache hierarchies with application hints," *ACM Transactions on Compututer Systems*, vol. 29, no. 2, pp. 1–51, 2011.

[3] H. Liu and H. H. Huang, "Graphene: fine-grained io management for graph computing." in *USENIX Conference on File and Storage Technologies (FAST'17)*, 2017.

[4] T. Luo, R. Lee, M. Mesnier, F. Chen, and X. Zhang, "hStorage-DB: Heterogeneity-aware data management to exploit the full capability of hybrid storage systems," in *ACM International Conference on Very Large Databases (VLDB'12)*, 2012.

[5] M. Mesnier, J. Akers, F. Chen, and T. Luo, "Differentiated storage services," in *ACM Symposium on Operating System Principles (SOSP'11)*, 2011.

[6] J. Zhang, J. Shu, and Y. Lu, "ParaFS: a log-structured file system to exploit the internal parallelism of flash devices," in *USENIX Annual Technical Conference (ATC'16)*, 2016.

[7] S. Lee, M. Liu, S. W. Jun, S. Xu, J. Kim, and A. Arvind, "Application-managed flash." in *USENIX Conference on File and Storage Technologies (FAST'16)*, 2016.

[8] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, "Don't stack your log on my log," in *USENIX Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*, 2014.

[9] Y. Lu, J. Shu, W. Zheng *et al.*, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[10] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[11] M. Hao, G. Soundararajan, D. R. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The tail at store: a revelation from millions of hours of disk and SSD deployments," in *USENIX Conference on File and Storage Technologies (FAST'16)*, 2016.

[12] S. Yan, H. Li, M. Hao, H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-tail flash: near-perfect elimination of garbage collection tail latencies in NAND SSDs," in *USENIX Conference on File and Storage Technologies (FAST'17)*, 2017.

[13] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "SDF: software-defined flash for web-scale internet storage systems," in *ACM International Conference Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)*, 2014.

[14] M. Bjørling, J. Gonzalez, and P. Bonnet, "LightNVM: the Linux open-channel SSD subsystem," in *USENIX Conference on File and Storage Technologies (FAST'17)*, 2017.

[15] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *ACM European Conference on Computer Systems (EuroSys'14)*, 2014.

[16] Z. Shen, F. Chen, Y. Jia, and Z. Shao, "DIDACache: a deep integration of device and application for flash-based key-value caching," in *USENIX Conference on File and Storage Technologies (FAST'17)*, 2017.

[17] S. Seshadri, M. Gahagan, M. S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: a user-programmable SSD." in *USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[18] R. J. Yang and Q. Luo, "FlashTKV: a high-throughput transactional key-value store on flash solid state drives," in *International Conference on Advanced Communications and Computation (INFOCOMP'12)*, 2012.

[19] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "NVMKV: a scalable, lightweight, FTL-aware key-value store." in *USENIX Annual Technical Conference (ATC'15)*, 2015.

[20] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "KAML: A flexible, high-performance key-value SSD," in *IEEE International Symposium on High Performance Computer Architecture (HPCA'2017)*, 2017.

[21] Twitter, "Fatcache," https://github.com/twitter/fatcache.

[22] N. R. Miklos Szeredi, "Filesystem in userspace," http://fuse.sourceforge.net.

[23] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: large-scale graph computation on just a PC," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, 2012.

[24] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of flash translation layer," *Journal of Systems Architecture*, vol. 55, no. 5, pp. 332–343, 2009.

[25] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," in *ACM Computing Survey (CSUR)*, 2005.

[26] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *USENIX Annual Technical Conference (ATC'08)*, 2008.

[27] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, "FlashBlox: achieving both performance isolation and uniform lifetime for virtualized SSDs," in *USENIX Conference on File and Storage Technologies (FAST'17)*, 2017.

[28] 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 *IEEE International Symposium on High Performance Computer Architecture (HPCA'11)*, 2011.

[29] Memblaze, "Memblaze," http://www.memblaze.com/en/.

[30] Facebook, "McDipper: a key-value cache for flash storage," https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920.

[31] Memcached, "Memcached: a distributed memory object caching system," http://www.memcached.org.

[32] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review (SIGMETRICS'12)*, 2012.

[33] D. Carra and P. Michiardi, "Memory partitioning in memcached: an experimental performance analysis," in *IEEE International Conference on Communications (ICC'14)*, 2014.

[34] S. S. G. G. John Bucy, Jiri Schindler, "DiskSim 4.0," http://www.pdl.cmu.edu/DiskSim/.

[35] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *International conference on World Wide Web (WWW'10)*, 2010.

[36] "yahoo-web," http://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

[37] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *IEEE International Conference on Data Mining (ICDN'12)*, 2012.

[38] J. McAuley and J. Leskovec, "Learning to discover social circles in ego networks," in *International Conference on Neural Information Processing Systems (NIPS'12)*, 2012.

[39] L. Takac and M. Zabovsky., "Data analysis in public social networks," in *International Scientific Conference and International Workshop on Present Day Trends of Innovations*, 2012.

[40] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: performance of user-space file systems." in *USENIX Conference on File and Storage Technologies (FAST'17)*, 2017.

[41] "Xmp," https://github.com/libfuse/libfuse/releases.

[42] "Filebench benchmark," http://sourceforge.net/apps/mediawiki/filebench.

[43] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *ACM SIGMOD International Conference on Management of data (SIGMOD'2010)*, 2010.

[44] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: a new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[45] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: distributed graph-parallel computation on natural graphs." in *USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, 2012.

[46] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on Spark," in *International Workshop on Graph Data Management Experiences and Systems (GRADES'13)*, 2013.

[47] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: edge-centric graph processing using streaming partitions," in *ACM Symposium on Operating Systems Principles (SOSP'13)*, 2013.

[48] X. Zhu, W. Han, and W. Chen, "GridGraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning." in *USENIX Annual Technical Conference (ATC'2015)*, 2015.