DIDACache: An Integration of Device and Application for Flash-based Key-value Caching

ZHAOYAN SHEN, The Hong Kong Polytechnic University
FENG CHEN, Louisiana State University
YICHEN JIA, Louisiana State University
ZILI SHAO, The Hong Kong Polytechnic University

Key-value caching is crucial to today's low-latency Internet services. Conventional key-value cache systems, such as Memcached, heavily rely on expensive DRAM memory. To lower Total Cost of Ownership (TCO), the industry recently is moving toward more cost-efficient flash-based solutions, such as Facebook's McDipper [14] and Twitter's Fatcache [56]. These cache systems typically take commercial SSDs and adopt a Memcached-like scheme to store and manage key-value cache data in flash. Such a practice, though simple, is inefficient due to the huge *semantic gap* between the key-value cache manager and the underlying flash devices.

In this paper, we advocate to reconsider the cache system design and directly open device-level details of the underlying flash storage for key-value caching. We propose an enhanced flash-aware key-value cache manager, which consists a novel unified address mapping module, an integrated garbage collection policy, a dynamic over-provisioning space management, and a customized wear-leveling policy, to directly drive the flash management. A thin intermediate library layer which provides a slab-based abstraction of low-level flash memory space and an API interface for directly and easily operating flash devices. A special flash memory SSD hardware that exposes flash physical details is adopted to store key-value items. This co-design approach bridges the semantic gap and well connects the two layers together, which allows us to leverage both the domain knowledge of key-value caches and the unique device properties. In this way, we can maximize the efficiency of key-value caching on flash devices while minimizing its weakness. We implemented a prototype, called DIDACache, based on the Open-Channel SSD platform. Our experiments on real hardware show that we can significantly increase the throughput by 35.5%, reduce the latency by 23.6%, and remove unnecessary erase operations by 28%.

CCS Concepts: •**Information systems →Storage management;**

Additional Key Words and Phrases: NAND flash memory, key-value caching, open-channel SSD

ACM Reference format:
Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. 2018. DIDACache: An Integration of Device and Application for Flash-based Key-value Caching. *ACM Trans. Storage* 14, 3, Article 0 (XXX 2018), 32 pages.
DOI: 0000001.0000001

1 INTRODUCTION

High-speed key-value caches, such as Memcached [38] and Redis [45], are the "first line of defense" in today's low-latency Internet services. By caching the working set in memory, key-value cache systems can effectively remove time-consuming queries to the backend data store (e.g., MySQL

This is a revised version. A preliminary version of this work appeared in the Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 2017).

Authors' addresses: Zhaoyan Shen is with the Department of Computing, the Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong; Yichen Jia and Feng Chen are with the Department of Computer Science and Engineering, Louisiana State University, Baton Rouge, Louisiana, USA; Zili Shao is with the Department of Computing, the Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong.

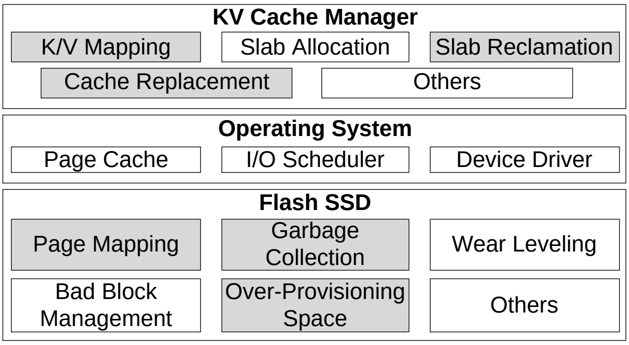


Fig. 1. Architecture of flash-based key-value cache.

or LevelDB). Though effective, the in-memory key-value caches heavily rely on large amount of expensive and power-hungry DRAM for high cache hit ratio [21]. As the workload size rapidly grows, an increasing concern with such memory-based cache systems is their cost and scalability [2]. A possible alternative is to directly replace DRAM with byte-addressable non-volatile memory (NVM), such as PCM [27, 31], however, these persistent memory devices are not yet available for large-scale deployment in commercial environment. Recently, a more cost-efficient alternative, *flash-based key-value caching*, has raised high interest in the industry [14, 56].

NAND flash memory provides a much larger capacity and lower cost than DRAM, which enables a low Total Cost of Ownership (TCO) for a large-scale deployment of key-value caches. Facebook, for example, deploys a Memcached-compatible key-value cache system based on flash memory, called McDipper [14]. It is reported that McDipper allows Facebook to reduce the number of deployed servers by as much as 90% while still delivering more than 90% "get responses" with sub-millisecond latencies [29]. Twitter also has a similar key-value cache system, called Fatcache [56].

Typically, these flash-based key-value cache systems directly use commercial flash SSDs and adopt a Memcached-like scheme to manage key-value cache data in flash. For example, key-values are organized into slabs of different size classes, and an in-memory hash table is used to maintain the key-to-value mapping. Such a design is simple and allows a quick deployment. However, it disregards an important fact – the key-value cache systems and the underlying flash devices both have very *unique properties*. Figure 1 shows a typical flash-based key-value cache architecture. The key-value cache manager that runs at the application level serves incoming requests and manages the cache space for allocation and replacement. The flash SSD at the device level manages flash chips and hides the unique characteristics of flash memory from applications. Simply treating flash SSDs as a faster storage and the key-value cache as a regular application not only fails to exploit various optimization opportunities but also raises several critical concerns: *Redundant mapping*, an application-level key-value-to-cache mapping and a device-level logical-to-physical flash space mapping; *Double garbage collection*, an application-level garbage collection process at the key-value item granularity to reclaim cache space and a device-level garbage collection process at the block granularity to reclaim flash space; and *Over-overprovisioning*, an application-level cache space reservation policy and a device-level over-provisioning space reservation. All these issues cause enormous inefficiencies in practice, which have motivated us to reconsider the software/hardware structure of the current flash-based key-value cache systems.

In this paper, we will discuss the above-mentioned three key issues (Section 3) caused by the huge *semantic gap* between the key-value caches and the underlying flash devices, and further present a cohesive cross-layer design to fundamentally address these issues. Through our studies, we advocate to open the underlying details of flash SSDs for key-value cache systems. Such a co-design effort not only enables us to remove the unnecessary intermediate layers between the cache manager and the storage devices, but also allows us to leverage the precious domain knowledge of key-value cache systems, such as the unique access patterns and mapping structures, to effectively exploit the great potential of flash storage while avoiding its weakness.

By reconsidering the division between software and hardware, a variety of new optimization opportunities can be explored: (1) A single, unified mapping structure can directly map the "keys" to physical flash pages storing the "values", which completely removes the redundant mapping table and saves a large amount of on-device memory; (2) An integrated Garbage Collection (GC) procedure, which is directly driven by the cache system, can optimize the decision of when and how to recycle *semantically invalid* storage space at a fine granularity, which removes the high overhead caused by the unnecessary and uncoordinated GCs at both layers; (3) An on-line scheme can determine an optimal size of Over-Provisioning Space (OPS) and dynamically adapt to the workload characteristics, which will maximize the usable flash space and greatly increase the cost efficiency of using expensive flash devices; (4) A wear-leveling policy that cooperates with GC to evenly wear out underlying flash blocks.

We implement a fully functional prototype, called *DIDACache*, based on a PCI-E Open-Channel SSD hardware, and provide an performance analysis for both the conventional key-value cache system and our proposed DIDACache. A thin intermediate library layer, `libssd`, is created to provide a programming interface to facilitate applications to access low-level device information and directly operate the underlying flash device. Using the library layer, we developed a flash-aware key-value cache system based on Twitter's Fatcache [56], and carried out a series of experiments to demonstrate the effectiveness of our new design scheme. Our experiments show that this approach can increase the throughput by 35.5%, reduce the latency by 23.6%, and remove erase operations by 28%.

The rest of paper is organized as follows. Section 2 and Section 3 give background and motivation. Section 4 describes the design and implementation. Experimental results are presented in Section 5. Section 7 gives the related work. The final section concludes this paper.

2 BACKGROUND

This section briefly introduces three key technologies, flash memory, SSDs, and the current flash-based key-value cache systems.

• **Key-value Cache**. Key-value caching is the backbone of many systems in modern web-server architecture. A cache can be deployed anywhere in the infrastructure where there is congestion with data delivery. The two main cache models are *look-aside cache* and *inline cache*. The main difference of these two is that for inline cache, applications write new data or update the existing data in cache, which synchronously (write through) or asynchronously (write behind) write data to the backend data store. However, for look-aside cache, applications write new data to the backend data store, and then update the data in cache, if existing. In practice, key-value cache systems typically adopt the look-aside cache model, such as Memcached [38] and McDipper [14].

Figure 2 illustrates the basic workflow of a look-aside style key-value cache. In the example, the browser is the client, it sends requests to the application server, and the application server stores or accesses data from the key-value cache or the backend database. For writing a new data item, the application server directly stores the data to the backend database. For retrieving a data item, the application server first checks the key-value cache, if it is a cache hit, the data is returned from the cache without requesting the database; otherwise, the application server obtains data from the

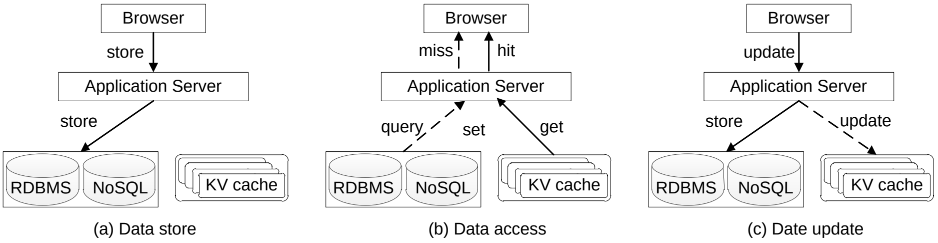


Fig. 2. A look-aside key-value caching example.

backend database and then writes it to the cache for future requests. For update operations, the application server updates existing data in both the key-value cache and the backend database. In this model, the data consistency is maintained by the application server.

• **Flash Memory**. NAND flash memory is a type of EEPROM device. A flash memory chip consists of two or more dies and each die has multiple *planes*. Each plane contains thousands of *blocks* (a.k.a. erase blocks). A block is further divided into hundreds of *pages*. Flash memory supports three main operations, namely `read`, `write`, and `erase`. Reads and writes are normally performed in units of pages. A read is typically fast (e.g., 50µs), while a write is relatively slow (e.g., 600µs). A constraint is that pages in a block must be written sequentially, and pages cannot be overwritten in place, meaning that once a page is programmed (written), it cannot be written again until the entire block is erased. An erase is typically slow (e.g., 5ms) and must be done in block granularity.

• **Flash SSDs**. A typical SSD includes four major components (Figure 3): A *host interface logic* connects the device to the host via an interface connection (e.g., SATA or PCI-E). An *SSD controller* is responsible for managing flash memory space, handling I/O requests, and issuing commands to flash memory chips via a *flash controller*. A dedicated *buffer* holds data or metadata, such as the mapping table. Most SSDs have multiple channels to connect the controller with flash memory chips, providing internal parallelism [9]. Multiple chips may share one channel. Actual implementations may vary in commercial products. More details about the SSD architecture can be found in prior work [3, 12]. A *Flash Translation Layer* (FTL) is implemented in SSD controller firmware to manage flash memory. and hide all the complexities behind a simple Logical Block Address (LBA) interface. which makes an SSD similar to to a disk drive. An FTL has three major roles: (1) *Logical block mapping*. An in-memory mapping table is maintained in the on-device buffer to map logical block addresses to physical flash pages dynamically. (2) *Garbage collection*. Due to the erase-before-write constraint, upon a write, the corresponding logical page is written to a new location, and the FTL simply marks the old page invalid. A GC procedure recycles obsolete pages later, which is similar to a Log-Structured File System [46]. (3) *Wear Leveling*. Since flash cells could wear out after a certain number of Program/Erase cycles, the FTL shuffles read-intensive blocks with write-intensive blocks to even out writes over flash memory. A previous work [15] provides a detailed survey of FTL algorithms.

• **Flash-based key-value caches**. In-memory key-value cache systems, such as Memcached, adopt a slab-based allocation scheme. Due to its efficiency, flash-based key-value cache systems, such as Fatcache, inherit a similar structure. Here we use Fatcache as an example; based on open documents [14], McDipper has a similar design. In Fatcache, the SSD space is first segmented into *slabs*. Each allocated slab is divided into *slots* (a.k.a. chunks) of equal size. Each slot stores a "value" item. According to the slot size, the slabs are categorized into different classes, from Class 1 to Class n, where the slot size increases exponentially. A newly incoming item is accepted into a class whose slot size is the best fit of the item size (i.e., the smallest slot that can accommodate the item). For quick access, a *hash mapping table* is maintained in memory to map the keys to the

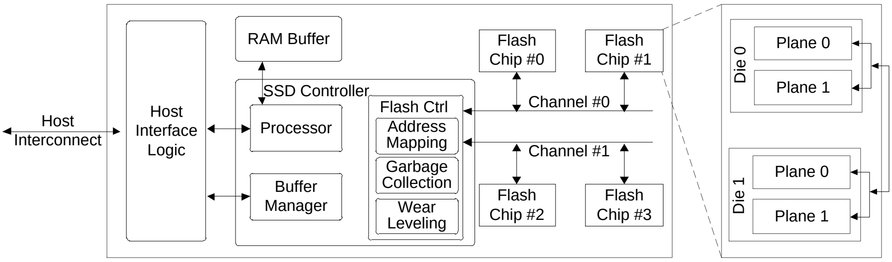


Fig. 3. Illustration of SSD architecture [3].

slabs containing the values. Querying a key-value pair (GET) is accomplished by searching the in-memory hash table and loading the corresponding slab block from flash into memory. Updating a key-value pair (SET) is realized by writing the updated value into a new location and updating the key-to-slab mapping in the hash table. Deleting a key-value pair (DELETE) simply removes the mapping from the hash table. The deleted or obsolete value items are left for GC to reclaim later.

Despite the structural similarity to Memcached, flash-based key-value cache systems have several distinctions from their memory-based counterparts. First, the I/O granularity is much larger. For example, Memcached can update the value items individually. In contrast, Fatcache has to maintain an in-memory slab to buffer small items in memory first and then flush to storage in bulk later, which causes a unique "large-I/O-only" pattern on the underlying flash SSDs. Second, unlike Memcached, which is byte addressable, flash-based key-value caches cannot update key-value items in place. In Fatcache, all key-value updates are written to new locations. Thus, a GC procedure is needed to clean/erase slab blocks. Third, the management granularity in flash-based key-value caches is much coarser. For example, Memcached maintains an object-level LRU list, while Fatcache uses a simple slab-level FIFO policy to evict the oldest slab when free space is needed.

3 MOTIVATION

As shown in Figure 1, in a flash-based key-value cache, the *key-value cache manager* and the *flash SSD* run at the application and device levels, respectively. Both layers have complex internals, and the interaction between the two raises three critical issues, which have motivated the work presented in this paper.

• **Problem 1: Redundant mapping**. Modern flash SSDs implement a complex FTL in firmware. Although a variety of mapping schemes, such as *block-level mapping* [19] and *page-level mapping* [20], exist, high-end SSDs often still adopt fine-grained *page-level mapping* for performance efficiency.As a result, for a 1TB SSD with a 4KB page size, a page-level mapping table could be as large as 1GB. Integrating such a large amount of DRAM on device not only raises production cost but also reliability concerns [20, 65, 66]. In the meantime, at the application level, the key-value cache system also manages another mapping structure, an in-memory hash table, which translates the keys to the corresponding slab blocks. The two mapping structures exist at two levels simultaneously, which unnecessarily doubles the memory consumption.

A fundamental problem is that the page-level mapping is designed for general-purpose file systems, rather than key-value caching. In a typical key-value cache, the slab block size is rather large (in Megabytes), which is typically 100-1,000x larger than the flash page size. This means that the fine-grained page-level mapping scheme is an *expensive over-kill*. Moreover, a large mapping table also incurs other overheads, such as the need for a large capacitor or battery, increased design complexity, reliability risks, etc. If we could directly map the hashed keys to the physical flash pages, we can completely remove this redundant and highly inefficient mapping for lower cost, simpler design, and improved performance.

• **Problem 2: Double garbage collection**. GC is the main performance bottleneck of flash SSDs [3, 8]. In flash memory, the smallest read/write unit is a page (e.g., 4KB). A page cannot be overwritten in place until the entire erase block (e.g., 256 pages) is erased. Thus, upon a write, the FTL marks the obsolete page "invalid" and writes the data to another physical location. At a later time, a GC procedure is scheduled to recycle the invalidated space for maintaining a pool of clean erase blocks. Since valid pages in the to-be-cleaned erase block must be first copied out, cleaning an erase block often takes hundreds of milliseconds to complete. A key-value cache system has a similar GC procedure to recycle the slab space occupied by obsolete key-value pairs.

Running at different levels (application vs. device), these two GC processes not only are redundant but also could interfere with one another. For example, from the FTL's perspective, it is unaware of the semantic meaning of page content. Even if no key-value pair is valid (i.e., no key maps to any value item), the entire page is still considered as "valid" at the device level. During the FTL-level GC, this page has to be moved unnecessarily. Moreover, since the FTL-level GC has to assume all valid pages contain useful content, it cannot selectively recycle or even aggressively invalidate certain pages that contain semantically "unimportant" (e.g., LRU) key-value pairs. For example, even if a page contains only one valid key-value pair, the entire page still has to be considered valid and cannot be erased, although it is clearly of relatively low value. Note that TRIM command [54] cannot address this issue as well. If we merge the two-level GCs and control the GC process based on semantic knowledge of the key-value caches, we could completely remove all the above-mentioned inefficient operations and create new optimization opportunities.

• **Problem 3: Over-overprovisioning**. In order to minimize the performance impact of GC on foreground I/Os, the FTL typically reserves a portion of flash memory, called Over-Provisioned Space (OPS), to maintain a pool of clean blocks ready for use. High-end SSDs often reserve 20-30% or even larger amount of flash space as OPS. From the user's perspective, the OPS space is nothing but an expensive unusable space. We should note that the factory setting for OPS is mostly based on a conservative estimation for worst-case scenarios, where the SSD needs to handle extremely intensive write traffic. In key-value cache systems, in contrast, the workloads are often read-intensive [5]. Reserving such a large portion of flash space is a significant waste of expensive resource. In the meantime, key-value cache systems possess rich knowledge about the I/O patterns and have the capability of accurately estimating the incoming write intensity. Based on such estimation, a suitable amount of OPS could be determined during runtime for maximizing the usable flash space for effective caching. Considering the importance of cache size for cache hit ratio, such a 20-30% extra space could significantly improve system performance. If we could leverage the domain knowledge of the key-value cache systems to determine the OPS management at the device level, we would be able to maximize the usable flash space for caching and greatly improve the overall cost efficiency as well as system performance.

In essence, all the above-mentioned issues stem from a fundamental problem in the current I/O stack design: the key-value cache manager runs at the application level and views the storage abstraction as a sequence of sectors; the flash memory manager (i.e., the FTL) runs at the device firmware layer and views incoming requests simply as a sequence of individual I/Os. This abstraction, unfortunately, creates a huge *semantic gap* between the key-value cache and the underlying flash storage. Since the only interface connecting the two layers is a strictly defined block-based interface, no semantic knowledge about the data could be passed over. This enforces the key-value cache manager and the flash memory manager to work individually and prevents any collaborative

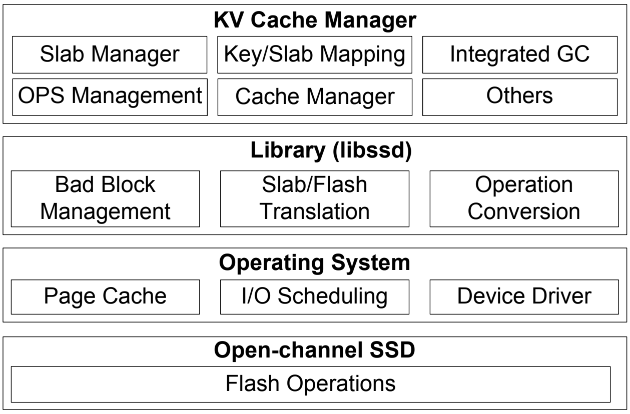


Fig. 4. The architecture overview of DIDACache.

optimizations. This motivates us to study how to bridge this semantic gap and build a highly optimized flash-based key-value cache system.

4 DESIGN

As an unconventional hardware/software architecture (see Figure 4), our key-value cache system is highly optimized for flash and eliminates all unnecessary intermediate layers. Its structure includes three layers.

- *An enhanced flash-aware key-value cache manager*, which is highly optimized for flash memory storage, runs at the application level, and directly drives the flash management;
- *A thin intermediate library layer*, which provides a slab-based abstraction of low-level flash memory space and an API interface for directly and easily operating flash devices (e.g., `read`, `write`, `erase`);
- *A specialized flash memory SSD hardware*, which exposes the physical details of flash memory medium and opens low-level *direct* access to the flash memory medium through the `ioctl` interface.

With such a holistic design, we strive to completely bypass multiple intermediate layers in the conventional structure, such as file system, generic block I/O, scheduler, and the FTL layer in SSD. Ultimately, we desire to let the application-level key-value cache manager leverage its domain knowledge and directly drive the underlying flash devices to operate only necessary functions while leaving out unnecessary ones. In this section, we will discuss each of the three layers.

4.1 Application Level: Key-value Cache

Our key-value cache manager has four major components: (1) a *slab management module*, which manages memory and flash space in slabs; (2) a *unified direct mapping module*, which records the mapping of key-value items to their physical locations; (3) an *integrated GC module*, which reclaims flash space occupied by obsolete key-values; and (4) an *OPS management module*, which dynamically adjusts the OPS size.

4.1.1 Slab Management. Similar to Memcached, our key-value cache system adopts a slab-based space management scheme – the flash space is divided into equal-sized *slabs*; each slab is divided

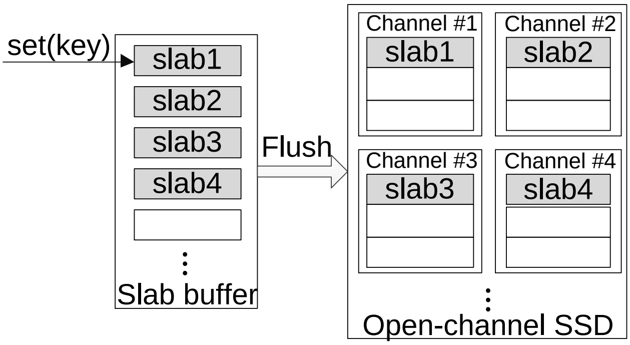


Fig. 5. Mapping slabs to flash blocks.

into an array of *slots* of equal size; each slot stores a key-value item; slabs are logically organized into different *slab classes* according to the slot size.

Despite these similarities to in-memory key-value caches, caching key-value pairs in flash has to deal with several unique properties of flash memory, such as the "out-of-place update" constraint. By directly controlling flash hardware, our slab management can be specifically optimized to handle these issues as follows.

• **Mapping slabs to blocks**: Our key-value cache directly maps (logical) slabs to physical flash blocks. We divide flash space into equal-sized slabs, and each slab is statically mapped to one or several flash blocks, as shown in Figure 5. There are two possible mapping schemes: (1) *Per-channel mapping*, which maps a slab to a sequence of contiguous physical flash blocks in one channel, and (2) *Cross-channel mapping*, which maps a slab across multiple channels in a round-robin way. Both have pros and cons. The former is simple and allows to directly infer the logical-to-physical mapping, while the latter could yield a better bandwidth through channel-level parallelism.

We choose the simpler per-channel mapping for two reasons. First, key-value cache systems typically have sufficient slab-level parallelism. Second, per-channel allows us to directly translate "slabs" into "blocks" at the library layer with minimal calculation. For cross-channel mapping, a big slab whose size is of several flash blocks may lead to flash space waste and make the slab to block mapping more complicated. A small slab in cross-channel mapping may pollute several flash blocks upon operations of invalidating slabs, which contributes to device-level GC overhead. In fact, in our prototype, we directly map a flash slab to a physical flash block, since the block size (8MB) is appropriate as one slab. For flash devices with a smaller block size, we can group multiple contiguous blocks in one channel into one slab.

• **Slab buffer**: Unlike DRAM memory, flash does not support random in-place overwrite. As so, a key-value item cannot be directly updated in its original place in flash. For a SET operation, the key-value item has to be stored in a new location in flash (appended like a log), and the obsolete item will be recycled later. To enhance performance, we maintain some *in-memory slabs* as buffer for flash slabs. Upon receiving a SET operation, the key-value pair is first stored in the corresponding in-memory slab and completion is immediately returned. When the in-memory slab is full, it is flushed into an *in-flash slab* for persistent storage. (the "Flush" process shown in Figure 5).

The slab buffer brings two benefits. First, the in-memory slab works as a write-back buffer. It not only speeds up accesses but also makes incoming requests asynchronous, which greatly improves the throughput. Second, and more importantly, the in-memory slab merges small key-value slot writes into large slab writes (in units of flash blocks), which completely removes the unwanted

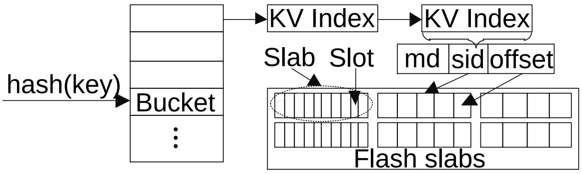


Fig. 6. The unified direct mapping structure.

small flash writes. Thus, from the device's perspective, all I/Os seen at the device level are in large-size slabs, which renders the unnecessarity of the generic GC at the FTL level. For this reason, flash writes in our system are all large writes, in units of flash blocks. Our experiments show that a small slab buffer is sufficient for performance.

• **Channel selection and slab allocation**: For load balance considerations, when an in-memory slab is full, we first select the channel with the lowest load. The load of each channel is estimated by counting three key flash operations (`read`, `write`, and `erase`). Once a channel is selected, a free slab is allocated. For each channel, we maintain a *Free Slab Queue* and a *Full Slab Queue* to manage clean slabs and used slabs separately. The slabs in a free slab queue are sorted in the order of their erase counts, and we always select the slab with the lowest erase count first for wear-leveling purposes. The slabs in a full slab queue are sorted in the Least Recently Used (LRU) order. When running out of free slabs, the GC procedure is triggered to produce clean slabs, which we will discuss in more details later.

With the above optimizations, a fundamental effect is, all I/Os seen at the device level are shaped into large-size slab writes, which completely removes small page writes as well as the need for generic GC at the FTL level.

4.1.2 Unified Direct Mapping. In order to address the double mapping problem, a key change is to remove all the intermediate mappings, and directly map the SHA-1 hash of the key to the corresponding physical location (i.e., the slab ID and the offset) in the in-memory hash table.

Figure 6 shows the structure of the in-memory hash table. Each hash table entry includes three fields: `<md, sid, offset>`. For a given key, `md` is the SHA-1 digest, `sid` is the ID of the slab that stores the key-value item, and `offset` is the slot number of the key-value item within the slab. Upon a request, we first calculate the hash value of the "key" to locate the bucket in the hash table, and then use the SHA-1 digest (`md`) to retrieve the hash table entry, in which we can find the slab (`sid`) containing the key-value pair and the corresponding slot (`offset`). The found slab could be in memory (i.e., in the slab buffer) or in flash. In the former case, the value is returned in a memory access; in the latter case, the item is read from the corresponding flash page(s).

Algorithm 4.1 shows the `SET` operation procedure in DIDACache with this unified mapping structure. When a `SET` request of one key-value item comes, DIDACache first checks wether it is an update operation or not. If it is an update operation, DIDACache removes the mapping record and updates the information associated with the operation of invalidating an obsolete key-value item (e.g., valid data ratio of its slab). Then, DIDACache allocates one slab whose slot size best fits this key-value pair, stores this key-value item in one slot, and updates the mapping with the slab and slot address. When there is not enough free memory slabs, the background "drain" process will be triggered to flush memory slabs to disk slabs. Similarly, an asynchronous integrated application-driven GC process will be called once there is not enough flash disk slabs inside SSD. Algorithm 4.2 presents the `GET` operations procedure, which is much simpler. When a `GET` request with one key comes, DIDACache searches the hash table, if the mapping record does not exist, an non-exist value is returned. Otherwise, DIDACache gets the ID of the slab ("sid") that stores the key-value item with the mapping structure. If the slab is in memory, the value is returned with

Algorithm 4.1 The Key-value SET Procedure

```
Input: $key$: Key for this key-value item
$value$ :Value for this key-value item
$CH_{num}$: Channel number in SSD
function BOOL SET($key$, $value$)
    if hash(key) exists then //for update operation
        remove(hash(key));
        Update the invalid information;
    end if
    Select one memory slab whose slot size best fits the KV size;
    Insert the KV item to the slot, establish an index for hash(key);
    if number of free memory slab < $free_{threshold}$ then
        slab_drain_thread(); //trigger the background slab drain process
    end if
    return true;
end function

function VOID SLAB_DRAIN_THREAD()
    while full_memory_slab > $full_{threshold}$ do
        if $channel(CH_{num})$ does not have free disk slab then
            $CH_{num} \leftarrow CH_{num} + 1$;
        end if
        Drain one memory slab to disk slab;
        if number of free disk slab < $W_{high}$ then
            Integrated_GC_thread();
        end if
    end while
end function
```

one memory load operation. If the slab is in disk, DIDACache needs to read the flash page which contains the key-value item, and return the value.

The unified direct mapping brings two benefits. First, it removes the redundant lookup in the intermediate mapping structures, which speeds up the query processing. Second, and more importantly, it dramatically reduces the demand for a large and expensive on-device DRAM buffer. Since the mapping tables at different levels are collapsed into one single must-have in-memory hash table, the FTL-level mapping table becomes unnecessary and can be completely removed from the device. This saves hundreds of Megabytes to even Gigabytes of on-device DRAM space. We could either reduce production cost or make a better use of on-device DRAM, such as on-device caching/buffering.

4.1.3 Garbage Collection. Garbage collection is a must-have in key-value cache systems, since operations (e.g., SET and DELETE) can create obsolete value items in slabs, which need to be recycled at a later time. When the system runs out of free flash slabs, we need to reclaim their space in flash.

With the semantic knowledge about the slabs, we can perform a fine-grained GC in one single procedure, running at the application level only. There are two possible strategies for identifying a victim slab: (1) *Space-based eviction*, which selects the slab containing the largest number of

Algorithm 4.2 The Key-value GET Procedure

```
Input: key: Key for this key-value item
function VALUE GET(key)
    if hash(key) does not exist then
        return -1; //key does not exist
    end if
    $sid = hash(key)$
    if $sid$ is in memory then
        return value; //return value with one memory load
    else
        $flash\_read(dev, sid)$; //read the data from flash
        return value;
    end if
end function
```

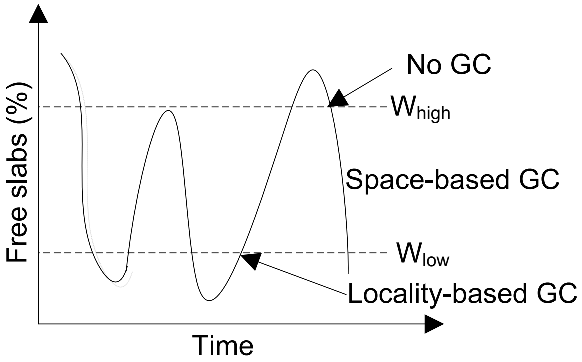


Fig. 7. Low and high watermarks.

obsolete values, and (2) *Locality-based eviction*, which selects the coldest slab for cleaning based on the LRU order. Both policies are used depending on the runtime system condition.

• **Space-based eviction**: As a greedy approach, this scheme aims to maximize the freed flash space for each eviction. To this end, we first select a channel with the lowest load to limit the search scope, and then we search its *Full Slab Queue* to identify the slab that contains the least amount of valid data. As the slot sizes of different slab classes are different, we use the number of valid key-value items times their size to calculate the valid data ratio for a given flash slab. Once the slab is identified, we scan the slots of the slab, copy all valid slots into the current in-memory slab, update the hash table mapping accordingly, then erase the slab and place the cleaned slab back in the *Free Slab Queue* of the channel.

• **Locality-based eviction**: This policy adopts an aggressive measure to achieve fast reclamation of free slabs. Similar to *space-based eviction*, we first select the channel with the lowest load. We then select the LRU slab as the victim slab to minimize the impact to hit ratio. This can be done efficiently as the full flash slabs are maintained in their LRU order for each channel. A scheme, called *quick clean*, is then applied by simply dropping the entire victim slab, including all valid slots. It is safe to remove valid slots, since our application is a key-value cache (rather than a key-value store) – all clients are already required to write key-values to the backend data store first, so it is safe to aggressively drop any key-value pairs in the cache without any data loss.

Algorithm 4.3 The Integrated Application Driven Garbage Collection Procedure

```
Input: $F_{dslab}$ : The number of free disk slab
  $W_{low}$ : Low watermark
  $W_{high}$ : High watermark
  $CH_{num}$ : Channel number in SSD
Output: Reclaim disk slabs.
  if Timer then
  Space-based eviction:
      if $F_{dslab}$ is less than $W_{high}$ and larger than $W_{low}$; then
          Choose a slab with maximum invalid data from the full slab queue of channel $CH_{num}$;
          Scan the slab and do valid key-value pair copy;
          Erase the slab and insert it into the free slab queue $CH_{num}$;
          $CH_{num} \leftarrow CH_{num} + 1$;
          if $CH_{num}$ equals to $Total\_CH$; then
              $CH_{num} \leftarrow 0$;
          end if
      end if
      if idle and $F_{dslab}$ is less than $W_{high}$; then
          goto Space-based eviction
      end if
  Locality-based eviction:
      while $F_{dslab}$ is less than $W_{low}$; do
          Choose a victim disk slab which is recently least accessed from the
          LRU full disk slab queue $CH_{num}$;
          Erase the slab and insert it into the free slab queue $CH_{num}$;
          $CH_{num} \leftarrow CH_{num} + 1$;
          if $CH_{num}$ equals to $Total\_CH$; then
              $CH_{num} \leftarrow 0$;
          end if
      end while
  end if
```

Comparing these two approaches, *space-based eviction* needs to copy still-valid items in the victim slab, so it takes more time to recycle a slab but retains the hit ratio. In contrast, *locality-based eviction* allows to quickly clean a slab without moving data, but it aggressively erases valid key-value items, which may reduce the cache hit ratio. To reach a balance between the hit ratio and GC overhead, we apply these two policies *dynamically* during runtime – when the system is under high pressure (e.g., about to run out of free slabs), we use the fast but imprecise *locality-based eviction* to quickly release free slabs for fast response; when the system pressure is low, we use *space-based eviction* and try to retain all valid key-values in the cache for hit ratio.

To realize the above-mentioned dynamic selection policies, we set two watermarks, low (W_{low}) and high (W_{high}). We will discuss how to determine the two watermarks in the next section. As shown in Algorithm 4.3, the GC procedure checks the number of free flash slabs, S_{free}, in the current system periodically. If S_{free} is between the high watermark, W_{high}, and the low watermark, W_{low}, it means that the pool of free slabs is running low but under moderate pressure. So we activate the less aggressive *space-based eviction* policy to clean slabs. This process repeats until

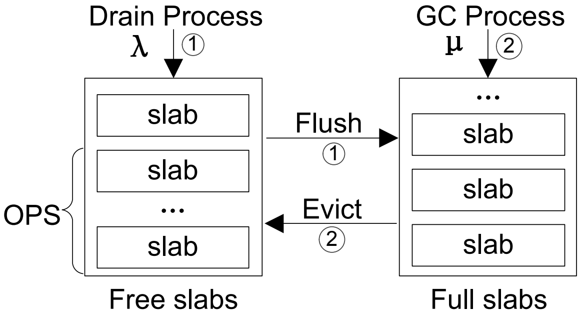


Fig. 8. Throughput for key-value items of size 256 bytes with different SET/Get ratios.

the number of free slabs, S_{free}, reaches the high watermark. If S_{free} is below the low watermark, which means that the system is under high pressure, the aggressive *space-based eviction* policy kicks in and uses *quick clean* to erase the entire LRU slab and discard all items immediately. This fast-response process repeats until the number of free slabs in the system, S_{free}, is brought back to W_{low}. If the system is idle, the GC procedure switches to the *space-based eviction* policy and continues to clean slabs until reaching the high watermark. Figure 7 illustrates this process.

4.1.4 Over-Provisioning Space Management. In conventional SSDs, a large portion of flash space is reserved as OPS, which is invisible and unusable by applications. In our architecture, applications can access all the physical flash blocks. We aim to leverage the application's domain knowledge to dynamically adjust the reserved space and maximize the usable flash space for caching. In the following, we refer to this dynamically changeable reserved space as OPS, and build a model to adjust its size during the run time.

In our system, the two watermarks, W_{low} and W_{high}, drive the GC procedure. The two watermarks effectively determine the available OPS size – W_{low} is the dynamically adjusted OPS size, and W_{high} can be viewed as the upper bound of allowable OPS. We set the difference between the two watermarks, $W_{high} - W_{low}$, as a constant (15% of the flash space in our prototype). Ideally, we desire to have the number of free slabs, S_{free}, fluctuating in the window between the two watermarks.

Our goal is to keep just enough flash space for over-provisioning. However, it is challenging to appropriately position the two watermarks and make them adaptive to the workload. It is desirable to have an automatic, self-tuning scheme to dynamically determine the two watermarks based on runtime situation. In our prototype, we have designed two schemes, a *feedback-based heuristic model* and a *queuing theory based model*.

Our heuristic scheme is simple and works as follows: when the low watermark is hit, which means that the current system is under high pressure, we lift the low watermark by doubling W_{low} to quickly respond to increasing writes, and the high watermark is correspondingly updated. As a result, the system will activate the aggressive *quick clean* to produce more free slabs quickly. This also effectively reserves a large OPS space for use. When the number of free slabs reaches the high watermark, which means the current system is under light pressure, we linearly drop the watermarks. This effectively returns free slabs back to the usable cache space (i.e., reduced OPS size). In this way, the OPS space automatically adapts to the incoming traffic.

The second scheme is based on the well-known queuing theory, which builds slab allocation and reclaim processes as a M/M/1 queue. As Figure 8 shows, in this system, we maintain queues for free flash slabs and full flash slabs for each channel, separately. The slab drain process consumes free slabs, and the GC process produces free slabs. Therefore we can view the drain process as the consumer process, the GC process as the producer process, and the free slabs as resources. The drain process consumes flash slabs at a rate λ, and the GC process generates free flash slabs at a rate μ. Prior study [5] shows that in real applications, the incoming of key-value pairs can be seen as a Markov process, so the drain process is also a Markov process. For the GC process, when S_{free} is less than W_{low}, the locality-based eviction policy is adopted. The time consumed for reclaiming one slab is equal to the flash erase time plus the schedule time. The flash block erase time is a constant, and the schedule time can be viewed as a random number. Thus the locality-based GC process is also a Markov process with a service rate μ. Based on the analysis, the process can be modeled as a M/M/1 queue with arrival rate λ, service rate μ, and one server.

According to Little's law, the expected number of slabs waiting for service is $\lambda/(\mu - \lambda)$. If we reserve at least this number of free slabs before the locality-based GC process is activated, we can always eliminate the synchronous waiting time. So, for the system performance benefit, we set

$$W_{low} = \lambda/(\mu - \lambda) \tag{1}$$

In the above equation, λ is the slab consumption rate of the drain process, and μ is the slab reclaim rate of GC, which equals $1/(t_{evict} + t_{other})$, where t_{evict} is the block erase time, and t_{other} is other system time needed for GC.

In Equation 2, the arrival rate is decided by the incoming rate of key-value pairs and their average size, which are both measurable. Assuming the arrival rate of key-values is λ_{KV}, the average size is S_{KV}, and the slab size is S_{slab}, λ can be calculated as follows.

$$\lambda = \frac{\lambda_{KV} \times S_{KV}}{S_{slab}} \tag{2}$$

So, we have

$$W_{low} = \frac{\lambda_{KV} \times S_{KV} \times (t_{evict} + t_{other})}{S_{slab} - \lambda_{KV} \times S_{KV} \times (t_{evict} + t_{other})} \tag{3}$$

By using the above-mentioned equations, we can periodically update the settings of the low and high watermarks. In this way, we can adaptively tune the OPS size based on real-time workload demands.

4.1.5 *Wear-leveling.*

Flash memory wears out after a certain number of Program/Erase (P/E) cycles. In our prototype, key-value update operations are performed in an out-of-place way, meaning that the updated key-value items are stored within the newly allocated slabs, and the stale key-value items need to be reclaimed through the GC process. For wear leveling, when allocating slabs in the drain process and reclaiming slabs in the GC process, we take the erase count of each slab into consideration and always use the block with the smallest erase count. Our locality-based GC that selects the least recently used blocks also helps evict those cold key-value items from their occupied flash blocks. Furthermore, as our channel-slab selection and slab-allocation scheme can evenly distribute the workloads across all channels, wears can be approximately distributed across channels as well.

Despite of these optimization policies, uneven aging still exists. For example, flash blocks which are filled with read intensive key-value items may be rarely erased. To further ensure uniform aging of all flash blocks, we adopt a simple yet effective approach by periodically invoking the wear-leveling procedure. Nonetheless, instead of swapping flash blocks that have higher wear number with those lower ones, we propose to incorporate this periodical wear-leveling procedure within the GC process.

In DIDACache, we maintain the total erase count and erase number of each flash slab. The wear-leveling process is periodically triggered when the total erase count exceeds m times of the total flash block number in the system. For example, we set $m = 2$ in our prototype. Suppose there are 1,000 flash blocks in the system, then the wear-leveling process will be triggered when the total erase count equals to 2,000. Once the wear-leveling process is triggered, we calculate the average wear number of flash blocks, and identify those flash blocks whose erase counts are far lower than the average number. These cold slabs are either seldom accessed or read-intensive. If a victim slab is seldom accessed, we can directly evict it out (just as quick-clean). If a victim slab is read-intensive, instead of simply swapping key-value items stored in the cold flash slab with a hot slab, DIDACache marks the cold block as victim block, and puts them into the GC queue. The GC process will reclaim these cold flash blocks and put them into the free slab queue to serve new incoming requests.

Traditional wear-leveling requires to shuffle frequently erased flash blocks with the less frequently erased ones, which involves a large amount of data copy, consuming I/O bandwidth and also increasing P/E cycles. In DIDACache, we are able to directly integrate wear-leveling within the GC procedure. This optimization policy reduces the amount of unnecessary data copy without defeating the purpose of GC and wear-leveling. In particular, since DIDACache does not support in-place update, if a slab has write-intensive key-value items, they must have already been copied out to other blocks, leaving obsolete slots ready for recycling. Thus, unlike traditional wear-leveling, we are able to skip copying these data. If a key-value items in the slab are not frequently read, as described in Section 4.1.3, DIDACache will devote the slab as "inactive" by checking its access count and use quick clean to directly erase this entire slab without moving data. Only if the key-value items are read-intensive, the GC process will find the slab active, and these hot items will be copied before erasing the slab. Thus, compared to traditional wear-leveling, this approach only needs to copy read-intensive data, achieving both effective wear-leveling and minimized data copy.

4.1.6 Crash Recovery. Crash recovery is also a challenge. As a typical key-value cache, all the key-value items have their persistent copy in the back database store. Thus, when system crash happens, we may simply drop the entire cache upon crashes. However, due to the excessively long warm-up time, it is preferred to retain the cached data through crashes [64]. In our system, all key-value items are stored in persistent flash but the hash table is maintained in volatile memory. There are two potential solutions to recover the hash table. One simple method is to scan all the valid key-value items in flash and rebuild the hash table, which is a time-consuming process. This approach demands more time for reconstructing the hash table. A more efficient solution is to periodically checkpoint the in-memory hash table into (a designated area of) the flash. Upon recovery, we only need to reload the latest hash table checkpoint into memory and then apply changes by scanning the slabs written after the checkpoint. Crash recovery is currently not implemented in our prototype. Applications use a persistent cache to improve repeated accesses. However, it is possible that the data in the backend data store are updated during the period of cache server downtime. Handling this situation is out of the scope of a look-aside cache, and applications or systems should implement certain methods to ensure that the data in the cache are still up-to-date after recovery. For example, when updating data, if the application finds the cache server is offline, it should not only update the data in the backend data store but also log the update operations locally or in another server, and when the cache server is recovered, the cache can be brought back to a consistent state by examining the log and replaying the update operations.

4.2 Library Level: `libssd`

As an intermediate layer, the library, `libssd`, connects the application and device layers. Unlike Liblightnvm [17], `libssd` is highly integrated with the key-value cache system. It has three main functions: (1) *Slab-to-block mapping*, which statically maps a slab to one (or multiple contiguous) flash memory block(s) in a channel. In our prototype, it is a range of blocks in a flash LUN (logic unit number). Such a mapping can be calculated through a mathematical conversion and does not require another mapping table. (2) *Operation transformation*, which converts key slab operations, namely `read`, `write`, and `erase`, to flash memory operations. This allows the key-value cache system to operate in units of slabs, rather than flash pages/blocks. (3) *Bad block management*, which maintains a list of flash blocks that are detected as "bad" and ineligible for allocation, and hides them from the key-value cache.

4.3 Hardware Level: Open-Channel SSD

Recently, there is a new trend of SSD design, called Open-Channel SSD, which directly exposes the internal channels and its low-level flash details to the host. With Open-Channel SSD, the responsibility of flash management is shared between the host software and hardware device. Compared with conventional SSD design, Open-Channel SSD has three unique features: (1) SSD internal parallelism is exposed to user applications. Open-Channel SSD exposes its internal geometry details (e.g., the layout of channels, LUNs, and flash blocks) to software applications. Applications have the flexibility of scheduling I/O tasks among different channels to fully utilize the raw flash performance. (2) Block erase command is available to applications. Open-Channel SSD exposes its low-level details to applications, thus, the applications are capable of controlling the flash GC process. (3) Open-Channel SSD enjoys a simplified I/O stack. Applications can directly operate the device hardware through the `ioctl` interface, which allows them to bypass many intermediate OS components, such as file system and the block I/O layer.

We use an Open-Channel SSD manufactured by Memblaze [37]. This hardware is similar to that used in SDF [42]. This PCIe based SSD contains 12 channels, each of which connects to two Toshiba 19nm MLC flash chips. Each chip contains two planes and has a capacity of 66GB. Unlike SDF [42], our SSD exposes several key device-level properties: first, the SSD exposes the entire flash memory space to the upper level. The SSD hardware abstracts the flash memory space in 192 LUNs, and an LUN is the smallest parallelizable unit. The LUNs are mapped to the 12 channels in a sequential manner, i.e., channel #0 contains LUNs 0-15, channel #1 contains LUNs 16-31, and so on. Therefore, we know the physical mapping of slabs on flash memory and channels. Second, unlike SDF, which presents the flash space as 44 block devices, our SSD provides direct access to raw flash memory through the `ioctl` interface. It allows us to directly operate the target flash memory pages and blocks by specifying the LUN ID and page number to compose commands added to the device command queue. Third, all FTL-level functions, such as address mapping, wear-leveling, bad block management, are bypassed. This allows us to remove the device-level redundant operations and make them completely driven by the user-level applications.

5 EVALUATION

In this section, we present evaluation results that demonstrate the benefits of the design choices of DIDACache. Specially, we seek to answer the following fundamental performance questions about DIDACache:

- Does this co-design approach result in higher SSD utilization, and how does it impact performance (throughput, latency), and device endurance?
- How does DIDACache perform with real workloads, compared to its peers?
- What is the effect of memory slab buffer on DIDACache's performance?
- What is DIDACache's garbage collection overhead with different policies?
- How does the dynamic over-provisioning space schemes perform?
- What is the CPU and memory overhead of DIDACache?

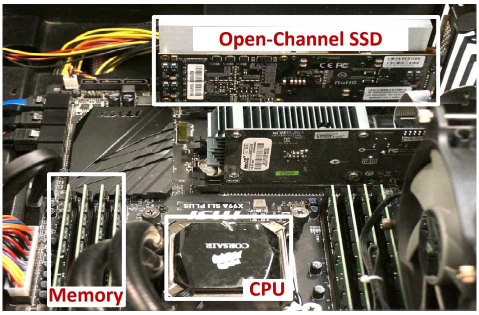


Fig. 9. Hardware platform.

5.1 Prototype System

We have prototyped the proposed key-value cache on the Open-Channel SSD hardware platform manufactured by Memblaze [37]. Our implementation of the key-value cache manager is based on Twitter's Fatcache [56]. It includes 1,640 lines of code in the stock Fatcache and 620 lines of code in the library.

In Fatcache, when a SET request arrives, if running out of in-memory slabs, it selects and flushes a memory slab to flash. If there is no free flash slab, a victim flash slab is chosen to reclaim space. During this process, incoming requests have to wait synchronously. To fairly compare with a cache system with non-blocking flush and eviction, we have enhanced the stock Fatcache by adding a drain thread and a slab eviction thread. The other part remains unchanged. We have open-sourced our asynchronous version of Fatcache for public downloading [1]. In our experiments, we denote the stock Fatcache working in the synchronous mode as "Fatcache-Sync", and the enhanced one working in the asynchronous mode as "Fatcache-Async". For each platform, we configure the slab size to 8 MB, the flash block size. The memory slab buffer is set to 128MB.

For performance comparison, we also run Fatcache-Sync and Fatcache-Async on a commercial PCI-E SSD manufactured by Memblaze. The SSD is built on the exact same hardware as our Open-Channel SSD but adopts a typical, conventional SSD architecture design. This SSD employs a page-level mapping and the page size is 16KB. Unlike the Open-Channel SSD, the commercial SSD has 2GB of DRAM on the device, which serves as a buffer for the mapping table and a write-back cache. The other typical FTL functions (e.g., wear-leveling, GC, etc.) are active on the device.

5.2 Experimental Setup

Our experiments are conducted on a workstation, which features an Intel i7-5820K 3.3GHZ processor and 16GB memory. An Open-Channel SSD introduced in Section 4.3 is used as DIDACache's underlying cache storage. (Figure 9). Since the SSD capacity is quite large (1.5TB), it would take excessively long time to fill up the entire SSD. To complete our tests in a reasonable time frame, we only use part of the flash space, and we ensure the used space is evenly spread across all the channels and flash LUNs. Note that for the commercial SSD, since we cannot control its OPS space, Fatcache running on the commercial SSD is able to use more OPS space than it should, which favors the stock Fatcache configuration as a comparison to our DIDACache.For the software, we use Ubuntu 14.04 with Linux kernel 3.17.8. Our backend database server is MySQL 5.5 with InnoDB storage engine running on a separate workstation, which features an Intel Core 2 Duo processor (3.13GHZ), 8GB memory and a 500GB hard drive. The database server and the cache server are connected in a 1-Gbps local Ethernet network. Note that in our experimental environment, network

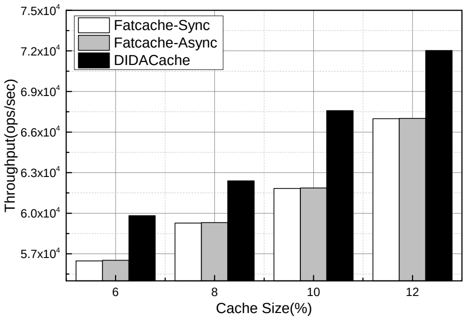


Fig. 10. Throughput vs. cache size

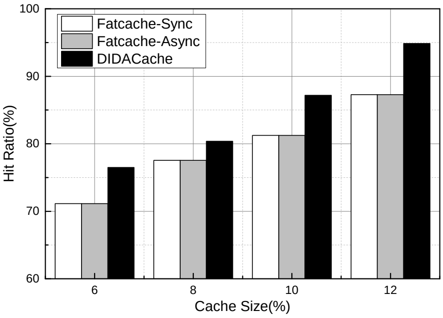


Fig. 11. Hit ratio vs. cache size.

is not the bottleneck.Fatcache-Sync and Fatcache-Async use the same system configurations, except that they run on the commercial SSD rather than the Open-Channel SSD.

5.3 Overall Performance

Our first set of experiments simulate a production data-center environment to show the overall performance. In this experiment, we have a complete system setup with a workload generator (client simulator), a key-value cache server, and a MySQL database server in the backend.

To generate key-value requests to the cache server, we adopt a workload model presented in prior work [7]. This model is built based on real Facebook workloads [5], and we use it to generate a key-value object data set and request sequences to exercise the cache server. The size distribution of key-value objects in the database follows a truncated Generalized Pareto distribution with location $\theta = 0$, scale $\psi = 214.4766$, and shape $k = 0.348238$. The object popularity, which determines the request sequence, follows a Normal distribution with mean μ_t and standard deviation σ, where μ_t is a function of time. We first generate 800 million key-value pairs (about 250GB data) to populate our database, and then use the object popularity model to generate 200 million requests. We have run experiments with various numbers of servers and clients with the above-mentioned workstation, but due to the space constraint, we only present the representative experimental results with 32 clients and 8 key-value cache servers.

We test the system performance by varying the cache size (in percentage of the data set size). Figure 10 shows the throughput, i.e., the number of operations per second (ops/sec). We can see that as the cache size increases from 5% to 12%, the throughput of all the three schemes improves significantly, due to the improved cache hit ratio. Comparing the three schemes, DIDACache outperforms Fatcache-Sync and Fatcache-Async substantially. With a cache size of 10% of the data set (about 25GB), DIDACache outperforms Fatcache-Sync and Fatcache-Async by 9.7% and 9.2%, respectively. The main reason is that the dynamic OPS management in DIDACache adaptively adjusts the reserved OPS size according to the request arrival rate. In contrast, Fatcache-Sync and Fatcache-Async statically reserve 25% flash space as OPS, which affects the cache hit ratio (see Figure 11). Another reason is the reduced overhead due to the application-driven GC. The effect of GC policies will be examined in Section 5.4.2.

We also note that Fatcache-Async only outperforms Fatcache-Sync marginally in this workload. It is because for this workload, both Fatcache-Sync and Fatcache-Async use the commercial SSD as the underlying storage and use the static OPS policy; thus, they have the same cache hit ratio. Though Fatcache-Async adopts an asynchronous drain process and GC process, they only benefit the 'set' operations, and its 'get' performance is identical to Fatcache-Sync. When the cache size varies from 5% to 12% of the workload size, the cache hit ratio can range from 71% to 87%, which is already high; thus, we cannot see much further improvement between Fatcache-Async and

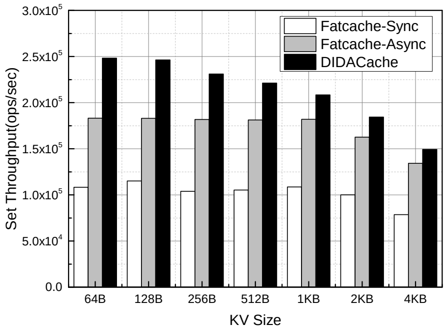


Fig. 12. SET throughput vs. KV size

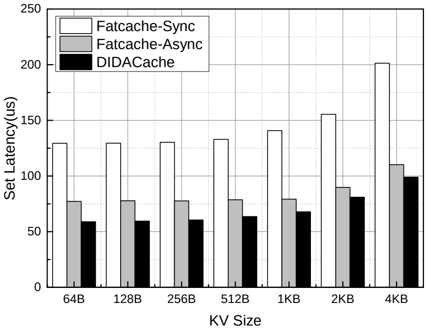


Fig. 13. SET latency vs. KV size

Fatcache-Sync. Besides, when a cache miss happens, a slow database query is needed, so the relative benefit from asynchronization is further diminished.

Figure 11 shows the hit ratios of these three cache systems. We can see that, as the cache size increases, DIDACache's hit ratio ranges from 76.5% to 94.8%, which is much higher than that of Fatcache-Sync, ranging from 71.1% to 87.3%.

5.4 Cache Server Performance

In this section we focus on studying the performance details of the cache servers. In this experiment, we directly generate SET/GET operations to the cache server. We create objects with sizes ranging from 64 bytes to 4KB and first populate the cache server up to 25GB in total. Then we generate SET and GET requests of various key-value sizes to measure the average latency and throughput. All experiments use 8 key-value cache servers and 32 clients.

5.4.1 Random SET/GET *Performance.* Figure 12 shows the throughput of SET operations. Among the three schemes, our DIDACache achieves the highest throughput and Fatcache-Sync performs the worst. With the object size of 64 bytes, the throughput of DIDACache is 2.48×10^5 ops/sec, which is 1.3 times higher than that of Fatcache-Sync and 35.5% higher than that of Fatcache-Async. The throughput gain is mainly due to our unified slab management policy and the integrated application-driven GC policy. DIDACache also selects the least loaded channel when flushing slabs to flash. Thus, the SSD's internal parallelism can be fully utilized, and with software and hardware knowledge, the GC overhead is significantly reduced. Compared with Fatcache-Async, the relative performance gain of DIDACache is smaller and decreases as the key-value object size increases. As the object size increases, the relative GC efficiency improves and the valid data copy overhead is decreased. It is worth noting that the practical systems are typically dominated by small key-value objects, on which DIDACache performs particularly well.

Figure 13 gives the average latency for SET operations with different key-value object sizes. Similarly, it can be observed that Fatcache-Sync performs the worst, and DIDACache outperforms the other two significantly. For example, for 64-byte objects, compared with Fatcache-Sync and Fatcache-Async, DIDACache reduces the average latency by 54.5% and 23.6%, respectively.

Figures 14 and 16 show the throughput and latency for workloads with mixed SET/GET operations. We can observe that DIDACache outperforms Fatcache-Sync and Fatcache-Async across the board, but as the portion of GET operations increases, the related performance gain reduces. Although we also optimize the path of processing GET, such as removing intermediate mapping, the main performance bottleneck is the raw flash read. Thus, with the workload of 100% GET, the latency and throughput of the three schemes are nearly the same, which also indicates that the performance

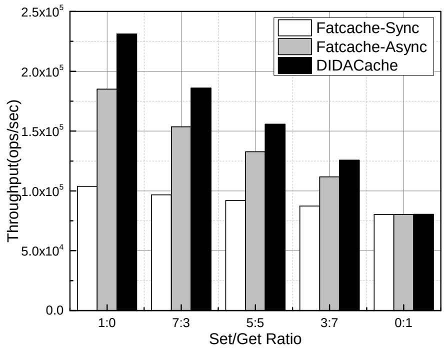


Fig. 14. Throughput vs. SET/GET ratio.

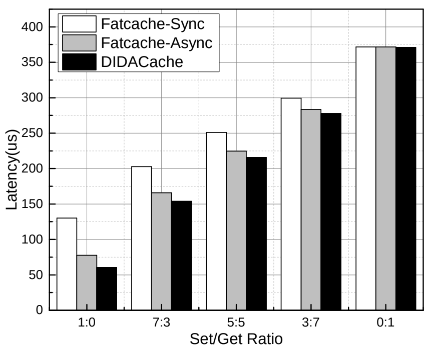
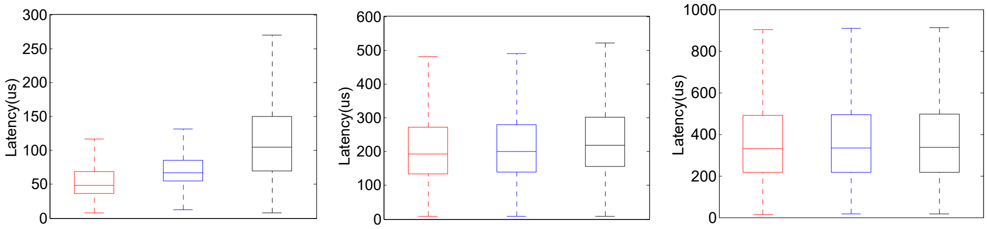


Fig. 15. Latency vs. SET/GET ratio.



(a) SET/GET(1:0).

(b) SET/GET(1:1).

(c) SET/GET(0:1).

Fig. 16. Latency (256-byte KV items) with different SET/GET ratios.

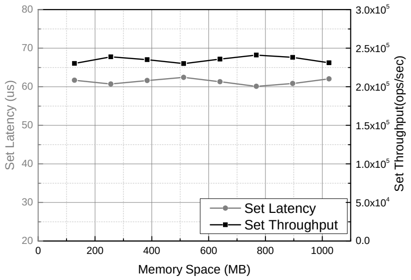


Fig. 17. Latency and Throughput for Set Operation with Different Buffer Size.

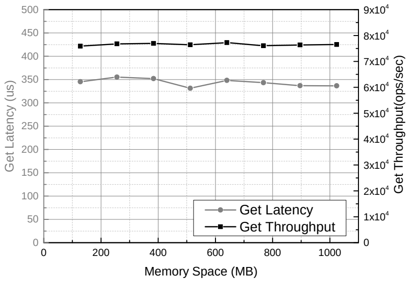


Fig. 18. Latency and Throughput for Get Operation with Different Buffer Size.

overhead (e.g., maintaining queues) introduced by our scheme is minimal. Figure 16 shows the latency distributions for key-value items of 256 bytes with different SET/GET ratios.

5.4.2 Memory Slab Buffer. Memory slab buffer enables the asynchronous operations of the drain and GC processes. To show the effect of slab buffer size, we vary the slab buffer size from 128MB to 1GB and test the average latency and throughput with the workloads generated with the truncated Generalized Pareto distribution. As shown in Figure 17 and Figure 18, for both SET and GET operations, the average latency and throughput are insensitive to the slab buffer size, indicating that a small in-memory slab buffer size (128M) is sufficient.

Table 1. Garbage collection overhead.

GC Scheme	Key-values	Flash Page	Erase
DIDACache-Space	7.48GB	N/A	4,231
DIDACache-Locality	0	N/A	3,679
DIDACache	2.05GB	N/A	3,829
Fatcache-Greedy	7.48GB	5.73GB	5,024
Fatcache-Kick	0	3.86GB	4,122
Fatcache-FIFO	15.35GB	0	5,316

5.4.3 Garbage Collection. Our cross-layer solution also effectively reduces the GC overhead, such as erase and valid page copy operations. In our cache-driven system, we can easily count erase and page copy operations in the library code. However, we cannot directly obtain these values on the commercial SSD as they are hidden at the device level. For effective comparison, we use the SSD simulator (extension to DiskSim [24]) from Microsoft Research and configure it with the same parameters of the commercial SSD. We first run the stock Fatcache on the commercial SSD and collect traces by using `blktrace` in Linux, and then replay the traces on the simulator. We compare our results with the simulator-generated results. In our experiments, we confine the available SSD size to 30GB, and preload it with 25GB data with workloads generated with the truncated Generalized Pareto distribution, and then do SET operations (80 million requests, about 30GB), following the Normal distribution.

Table 1 shows GC overhead in terms of valid data copies (key-values and flash pages) and block erases. We compare DIDACache using space-based eviction only ("DIDACache-Space"), locality-based eviction only ("DIDACache-Locality"), the adaptively selected eviction approach ("DIDACache") with the stock Fatcache using three schemes ("Fatcache-Greedy", "Fatcache-Kick", and "Fatcache-FIFO"). In Fatcache, the application-level GC has two options, copying valid key-value items from the victim slab for retaining hit ratio or directly dropping the entire slab for speed. This incurs different overheads of key-value copy operations, denoted as "Key-values". In this experiment, both Fatcache-Greedy and Fatcache-Kick use a greedy algorithm to find a victim slab, but the former performs key-value copy operations while the latter does not. Fatcache-FIFO uses a FIFO algorithm to find the victim slab and copies still-valid key-values. In the table, the flash page copy and block erase operations incurred by the device-level GC are denoted as "Flash Page" and "Erase", respectively.

Fatcache schemes show high GC overheads. For example, both Fatcache-Greedy and Fatcache-FIFO recycle valid key-value items at the application level, incurring a large volume of key-value copies. Fatcache-Kick, in contrast, aggressively drops victim slabs without any key-value copy. However, since it adopts a greedy policy (as Fatcache-Greedy) to evict the slabs with least valid key-value items, erase blocks are mixed with valid and invalid pages, which incurs flash page copies by the device-level GC. Fatcache-FIFO fills and erases all slabs in a sequential FIFO manner, thus, no device-level flash page copy is needed. All three Fatcache schemes show a large number of block erases.

The GC process in our scheme is directly driven by the key-value cache. It performs a fine-grained, single-level, key-value item-based reclamation, and no flash page copy is needed (denoted as "N/A" in Table 1). The locality-based eviction policy enjoys the minimum data copy overhead, since it aggressively evicts the LRU slab without copying any valid key-value items. The space-based eviction policy needs to copy 7.48 GB key-value items and incurs 4,231 erase operations. DIDACache dynamically chooses the most appropriate policy at runtime, so it incurs a GC overhead between the above two (2.05 GB data copy and 3,829 erases). Compared to Fatcache schemes, the overheads are much lower (e.g., 28% lower than Fatcache-FIFO).

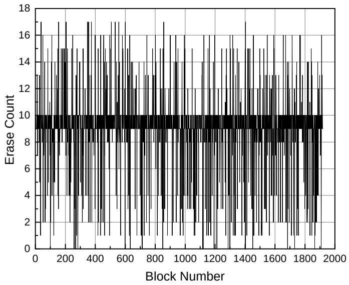


Fig. 19. Wear distribution among blocks without wear-leveling.

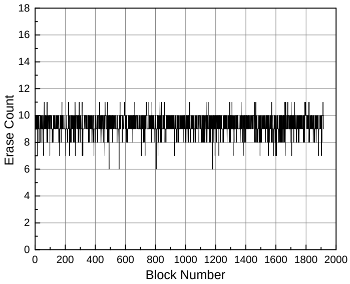


Fig. 20. Wear distribution among blocks with wear-leveling.

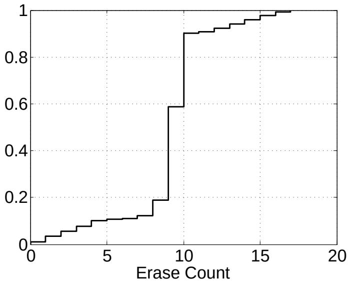


Fig. 21. CDF of blocks' erase count without wear-leveling.

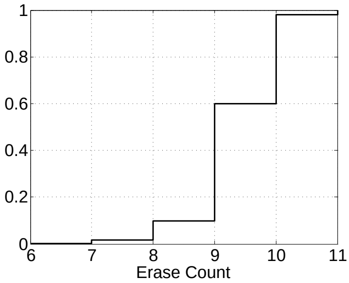


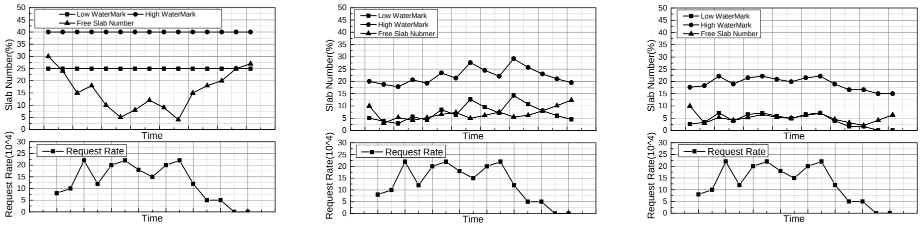
Fig. 22. CDF of blocks' erase count with wear-leveling.

Table 2. Wear-leveling overhead.

	GC		Wear-leveling		**Flash Page**
	Data copy	Erase	Data copy	Erase	
Wear-leveling	13.48GB	16,542	6.34GB	1,323	N/A
No Wear-leveling	15.57GB	17,285	N/A	N/A	N/A

5.4.4 Wear-leveling. To investigate the block aging status in DIDACache, we carry out experiments by keeping issuing SET and GET operations to DIDACache and collect the distribution of block erase operations in our library layer. In this experiment, to control the experimental time, we further confine the available SSD size to 15GB, and preload it with 10GB data with workloads generated with the truncated Generalized Pareto distribution, and then do SET and GET operations with workloads (480 million requests, about 240GB, SET/GET ratio is 1:1) that follow the Normal distribution. During the experiment, we count the number of GC operations, and our wear-leveling policy is periodically triggered when the total GC time comes up to two times of the total number of flash blocks. When the wear-leveling is triggered, we mark those blocks whose erase count is less than half of the average block erase count as victim blocks, and then reclaim these flash blocks with the GC process.

Figure 19 and Figure 20 show the block wear out distribution before and after we apply our wear-leveling policy, respectively. It can be observed that after applying our wear-leveling policy, flash blocks in the system are worn out much more evenly. Without wear-leveling, the minimum block erase count is 0, and the maximum block erase count is 17. With our wear-leveling policy, the flash block erase counts vary between 6 and 11. The maximum gap is only 5, which is much smaller than 17 in the former case. Figure 21 and Figure 22 give the CDF graphs of block erase

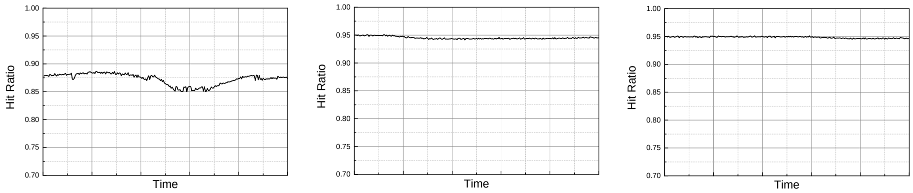


(a) Static policy.

(b) Heuristic policy.

(c) Queuing theory policy.

Fig. 23. Over-provisioning space with different policies.

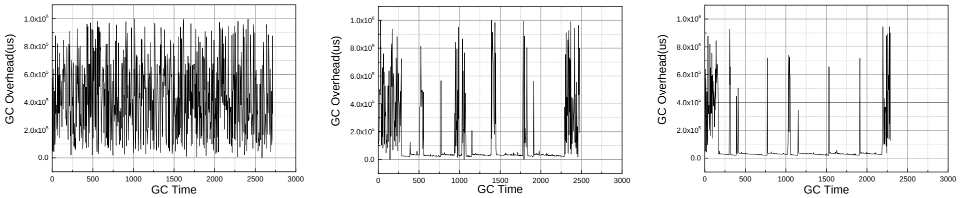


(a) Static policy.

(b) Heuristic policy.

(c) Queuing theory policy.

Fig. 24. Hit ratio with different OPS policies.

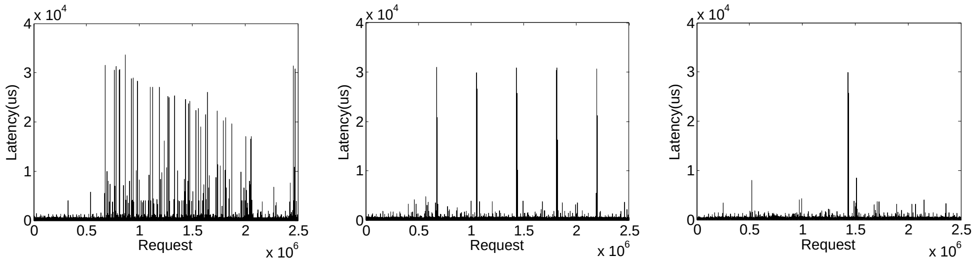


(a) Static policy.

(b) Heuristic policy.

(c) Queuing theory policy.

Fig. 25. Garbage collection overhead with different OPS policies.



(a) Static policy.

(b) Heuristic policy.

(c) Queuing theory policy.

Fig. 26. Request latency with different OPS policies.

counts accordingly. From them, we can see that with our wear-leveling policy, more than 90% flash blocks are erased 9 or 10 times. For the scheme without wear-leveling, although the majority of flash blocks are also erased by 9 or 10 times, but the percentage is much smaller, and the variance range is also much larger.

Table 3. Effect of different OPS policies.

GC Scheme	Hit Ratio	GC	Latency	Throughput
Static	87.7 %	2716	79.95	198,076
Heuristic	94.1 %	2480	64.24	223,146
Queuing	94.8 %	2288	62.41	229,956

The experimental results show that the our wear-leveling policy can effectively balance wears across flash blocks. However, since the wear-leveling policy incurs more GC operations, it also introduces some overhead. To illustrate the overhead of this mechanism, we compare the GC overheads of the systems with and without the wear-leveling policy (denoted as No wear-leveling and Wear-leveling) in Table 2. In this table, “Data copy” and “Erase” under column “GC” represent valid data copy and block erase operations caused by the GC process. Similarly, “Data copy” and “Erase” under column “Wear-Leveling” represent valid data copy and block erase operations caused by the wear-leveling process.

During the experiment, wear-leveling is triggered 4 times, and incurs 1323 block erase operations and 6.34GB data copy. Additionally, after applying our wear-leveling policy, the overhead of the GC procedure is less than that without our wear-leveling policy. The reason behind this is that we have integrated our wear-leveling procedure with the GC process. When wear-leveling happens, instead of swapping cold blocks with hot blocks, we mark those cold blocks as victim blocks. When reclaiming these victim blocks, we only copy those valid key-value items. If a victim block is not frequently accessed, we would directly erase the flash block without coping data. These measures, to some extent, can ease the pressure of the GC process. In all, with our wear-leveling, 580 more block erase and 4.25GB more data copy operations are introduced. We can further mitigate this overhead using a longer interval for wear-leveling, if needed.

5.4.5 Dynamic Over-Provisioning Space. To illustrate the effect of our dynamic OPS management, we run DIDACache on our testbed that simulates the data center environment in Section 5.3. We use the same data set containing 800 million key-value pairs (about 250GB), and the request sequence generated with the Normal distribution model. We set the cache size as 12% (around 30GB) of the data set size. In the experiment, we first warm up the cache server with the generated data, and then change the request coming rates to test our dynamic OPS policies.

Figure 23 shows the dynamic OPS and the number of free slabs with the varying request incoming rates for three different policies. The static policy reserves 25% of flash space as OPS to simulate the conventional SSD. For the heuristic policy, we set the initial W_{low} with 5%. For the queuing theory policy, we use the model built in Equation 3 to determine the value of W_{low} at runtime. We set W_{high} 15% higher than W_{low}. The GC is triggered when the number of free slabs drops below W_{high}.

As shown in Figure 23(a), the static policy reserves a portion of flash space for over-provisioning. The number of free slabs fluctuates, responding to the incoming request rate. In Figure 23(b), our heuristic policy dynamically changes the two watermarks. When the arrival rate of requests increases, the low watermark, W_{low}, increases to aggressively generate free slabs by using *quick clean*. The number of free slabs approximately follows the trend of the low watermark, but we can also see a lag-behind effect. Our queuing policy in Figure 23(c) performs even better, and it can be observed that the free slab curve almost overlaps with the low watermark curve. Compared with the static policy, both heuristic and queuing theory policies enable a much larger flash space for caching. Accordingly, we can see in Figure 24 that the two dynamic OPS policies are able to maintain a hit ratio close to 95%, which is 7% to 10% higher than the static policy. Figure 25 shows the GC cost, and we can find that the two dynamic policies incur lower overhead than the static policy. In fact, compared with the static policy and the heuristic policy, the queuing theory policy

Table 4. CPU utilization of different schemes.

Scheme	SET	GET	SET/GET (1:1)
DIDACache	47.7%	20.5 %	37.4 %
Fatcache-Async	42.3 %	20 %	33.8 %
Fatcache-Sync	40.1 %	20 %	31.3 %

erases 15.7% and 8% less flash blocks, respectively. Correspondingly, in Figure 26, it can be observed that the queuing policy can most effectively reduce the number of requests with high latencies.

To further study the difference of these three policies, we also compared their runtime throughput in Table 3. We can see that the static policy has the lowest throughput (198,076 ops/sec). The heuristic and queuing theory policies can deliver higher throughput, 223,146 and 229,956 ops/sec, respectively.

5.5 Overhead Analysis

DIDACache is highly optimized for key-value caching and moves certain device-level functions up to the application level. This could raise consumption of host-side resources, especially memory and CPU.

Memory Utilization: In DIDACache, memory is mainly used for three purposes. (1) In-memory hash table. DIDACache maintains a host-side hash table with 44-byte mapping entries (`<md, sid, offset>`), which is identical to the stock Fatcache. (2) Slab buffer. DIDACache performance is insensitive to the slab buffer size. We use a 128MB memory for slab buffer, which is also identical to the stock Fatcache. (3) Slab metadata. For slab allocation and GC, DIDACache introduces two additional queues (*Free Slab Queue* and *Full Slab Queue*) for each channel. Each queue entry is 8 bytes, corresponding to a slab. Each slab also maintains an erase count and a valid data ratio, each requiring 4 bytes. Thus, in total, DIDACache adds 16-byte metadata for each slab. For a 1TB SSD with a regular slab size of 8MB, it consumes at most 2MB memory. In our experiments, we found that the memory consumptions of DIDACache and Fatcache are almost identical during runtime. Also note that the device-side demand for memory is significantly decreased, such as the removed FTL-level mapping table.

CPU utilization: DIDACache is multi-threaded. In particular, we maintain 12 threads for monitoring the load of each channel, one global thread for garbage collection, and one load-monitoring thread for determining the OPS size. To show the related computational cost, we compare the CPU utilization of DIDACache, Fatcache-Async, and Fatcache-Sync in Table 4. It can be observed that DIDACache only incurs marginal increase of the host-side CPU utilization. In the worst case (100% SET), DIDACache only consumes extra 7.6% and 5.4% CPU resources over Fatcache-Sync (40.1%) and Fatcache-Async (42.3%), respectively. Finally it is worth noting that DIDACache removes much device-level processing, such as GC, which simplifies device hardware.

Cost implications: DIDACache is cost efficient. As an application-driven design, the device hardware can be greatly simplified for lower cost. For example, the DRAM required for the on-device mapping table can be removed and the reserved flash space for OPS can be saved. At the same time, our results also show that the host-side overhead, as well as the additional utilization of the host-side resources are minor.

6 DISCUSSION ON EXTREME CONDITIONS

Due to hardware constraint, some extreme cases are not triggered in our experiment. In this section, we will discuss the cache performance on some extreme conditions. We model the working procedure and analyze the performance of SET and GET operations, which are the two typical operations for key-value cache system. We breakdown and compare request latency for both the conventional key-value caching design and DIDACache.

• **SET Operation**: In both DIDACache and the conventional key-value caching, SET operations are served in an asynchronous way. When a SET operation comes, it will be firstly served by a memory slab. If the key-value item is stored in memory slab, the request can be returned, and the full memory slabs are flushed to flash in background as described in algorithm 4.1. So, in the best case, one key-value item SET operation only consists of one hash index build operation and one memory store operation. The request latency can be presented as:

$$t_{SET} = t_{hash} + t_{wmem}. \tag{4}$$

In here, t_{hash} and t_{wmem} stand for the hash index build time and memory store time, respectively.

However, in the worst case, the memory slab buffer and flash slabs are consumed very fast, which may cause incoming requests wait for the flash write and GC process synchronously. For DIDACache, in the worst case, the incoming SET request needs to wait for one flash block write operation and one integrated GC process. DIDACache adopts the *quick clean* scheme which directly erases the victim block without copying data; so when the system is starving for space, the integrated GC process only incurs one flash block erase operation. In the worst case, the request latency for SET operation can be denoted as:

$$t_{SET} = t_{hash} + t_{wmem} + t_{fwrite} + t_{erase}. \tag{5}$$

In here, t_{fwrite} is the time for one flash block write operation, t_{erase} is the time consumed by erasing one flash block.

In contrast, for conventional key-value caching, when the worst case happens, the serving process of one SET operation can be separated into software part and hardware part. From the software aspect, the request needs to wait for one software level GC process to reclaim cache space. From the hardware aspect, the request needs to wait for one slab flush operation and one hardware GC process to reclaim flash blocks. For the slab flush operation, the conventional SSD will slice one slab into stripes and flush the data to all its channels in parallel. Suppose the SSD contains N channels, and the time for one slab flush operation is t_{fwrite}/N. In the worst case, when hardware GC happens, each flash block contains one invalid flash page. If each block has m flash pages, to reclaim one flash block, the SSD needs to copy $m(m-1)$ flash pages, and erase m flash blocks. So the latency for one hardware GC process can be $t_{fwrite} \times (m-1)/N + t_{erase} \times m/N$. Thus, in the worst case, the request latency for SET operation is:

$$t_{SET} = t_{hash} + t_{wmem} + t_{sgc} + t_{fwrite} \times (m-1)/N + t_{erase} \times m/N. \tag{6}$$

Here, t_{sgc} is the time consumed by software level GC process. In the worst case, the software level GC process needs to copy $S_{slab}/S_{KV} - 1$ key-value items.

• **GET Operation**: Basically, the working procedure for GET operations of DIDACache and the conventional key-value caching are the same. For a GET operation, the caching system will firstly look up its in-memory index. If the corresponding key-value item is in memory, the data can be returned by a memory load operation. Otherwise, the system needs to read the data from SSD flash. The difference is that in DIDACache, when reading data from SSD flash, it does not need to use address mapping model to translate the logical disk slab number to flash pages. The time consumption for one GET operation in the conventional key-value caching is:

$$t_{GET} = \begin{cases} t_{rhash} + t_{rmem} & \text{if } \textit{the KV item is in memory slab} \\ t_{rhash} + t_{mapping} + t_{fread} & \text{if } \textit{the KV item is in disk slab} \end{cases} \tag{7}$$

Table 5. Key-value (256Bytes) request latency on extreme conditions.

Key-value Caching	Best Case		Worst Case	
	SET Latency	GET Latency	SET Latency	GET Latency
DIDACache	1 *us*	1 *us*	0.363 *s*	370 *us*
Conventional	1 *us*	1 *us*	15.492 *s*	370 *us*

Here, t_{rhash} represents the time consumed by searching the in-memory hash table. $t_{mapping}$ denotes the time consumed by FTL address mapping model, and t_{fread} is the time for flash page read operation. When the key-value item is in memory slab, it can be returned by just one hash table search and one memory load operation. Otherwise, if the key-value item is in disk slab, the latency is composed of a hash table search operation, an SSD address mapping search operation, and a flash page read operation.

For DIDACache, the time consumption for one GET operation is:

$$t'_{GET} = \begin{cases} t_{rhash} + t_{rmem} & \text{if } the\ KV\ item\ is\ in\ memory\ slab \\ t_{rhash} + t_{fread} & \text{if } the\ KV\ item\ is\ in\ disk\ slab \end{cases} \tag{8}$$

Similar to the conventional key-value caching, if the key-value item is in memory slab buffer, the latency for GET request is also $t_{rhash} + t_{rmem}$. But if the key-value item is in disk slab, the latency just include one hash table search and a flash page read operation. To conclude, for both best case and worst case, the latency for GET operation of DIDACache and the conventional key-value caching are basically the same.

Table 5 shows an example of latencies for SET and GET request on two extreme cases with our experimental hardware configuration. Due to space constraint, we only show the results with key-value item size of 256 bytes. Key-value items of other sizes have the same trend. For a SET request, in the best case, its latency only includes one index build operation and one memory store operation. In our experiments, for both DIDACache and the conventional key-value caching, the shortest latency is around 1 *us*. In our experiment, the conventional SSD contains 12 channels and each block has 512 pages, and each slab is 8MB. The time for writing and erasing one block are 0.356s and 7 *ms* respectively. The time granularity for t_{hash} and t_{wmem} are in *us*, which can be ignored in comparison. With equations 5 and 6, we get that the worst latency for one SET request in DIDACache is 0.363 *s*, and the worst latency for one SET request of conventional key-value caching is 15.492 *s*. For a GET request, DIDACache and the conventional key-value caching have quite similar working procedure. In our experiment, the shortest latency for both DIDACache and the conventional key-value caching is 1 *us*. In the worst case, the main bottleneck for the GET request latency is the raw flash read performance, and it is about 370 *us*.

7 OTHER RELATED WORK

Both flash memory [3, 8–10, 12, 18, 26, 28, 33, 36, 44, 50, 53, 59] and key-value systems [4, 5, 11, 16, 30, 32, 58, 61] are extensively researched. This section discusses prior studies most related to this paper.

A recent research interest in flash memory is to investigate the interaction between applications and underlying flash storage devices. Yang et al. investigate the interactions between log-structured applications and the underlying flash devices [60]. Differentiated Storage Services [39] proposes to optimize storage management with semantic hints from applications. Nameless Writes [62] is a de-indirection scheme to allow writing only data into the device and let the device choose the physical location. Similarly, FSDV [63] removes the FTL level mapping by directly storing physical flash addresses in the file systems. Multi-stream SSD [25] maintains multiple write streams with different expected lifetime for SSD. Applications write to different streams according to data lifetime. This design aims to make the NAND capacity unfragmented and handle the GC without costly data movement. Although sharing a similar principle of leveraging application semantics for efficient device management, DIDACache aims to bridge the semantic gaps between application and the underlying hardware and is specific for key-value cache systems. For example, DIDACache leverages the properties of key-value cache for aggressive quick-clean without incurring a problem.Willow [49] exploits on-device programmability to move certain computation from the host to the device. FlashTier [48] uses a customized flash translation layer optimized for caching rather than storage. OP-FCL dynamically manages OPS on SSD to balance the space needs for GC and for caching [41]. Some commercial SSDs allow users to define their own OPS space, such as Samsung 840 Pro [47]. However, these SSDs only allow applications to adjust the OPS space statically, and the OPS space cannot be dynamically adjusted according to the applications' runtime patterns. Our DIDACache dynamically determines the minimum reserved space for OPS purpose and maximizes the usable cache space during the runtime according to the application workload pattern. RIPQ [55] optimizes the photo caching in Facebook particularly for flash by reshaping the small random writes to a flash-friendly workload. FlashBlox [22] proposes to utilize flash parallelism to improve isolation between applications by running them on dedicated channels and dies, and balance wear within and across different applications. LightNVM [6] is an open-channel SSD subsystem in the Linux kernel, which introduces a new physical page address I/O interface that exposes SSD parallelism and storage media characteristics. Our solution shares a similar principle of removing unnecessary intermediate layers and collapsing multi-layer mapping into only one, but we particularly focus on tightly connecting key-value cache systems and the underlying flash SSD hardware.

Key-value cache systems recently show their practical importance in Internet services [5, 16, 32, 61]. A report from Facebook discusses their efforts of scaling Memcached to handle the huge amount of Internet I/O traffic [40]. McDipper [14] is their latest effort on flash-based key-value caching. Several prior research studies specifically optimize key-value store/cache for flash. Ouyang et al. propose an SSD-assisted hybrid memory for Memcached in high performance network [43]. This solution essentially takes flash as a swapping device. Flashield [13] is also a hybrid key-value cache which uses DRAM as a "filter" to minimize writes to flash. NVMKV [34, 35] gives an optimized key-value store based on flash devices with several new designs, such as dynamic mapping, transactional support, and parallelization. Unlike NVMKV, our system is a key-value cache, which allows us to aggressively integrate the two layers together and exploit some unique opportunities. For example, we can invalidate all slots and erase an entire flash block, since we are dealing with a cache rather than storage.

Some prior work also leverages Open-Channel SSDs for domain optimizations. Our prior study [51] outlines the key issues and a preliminary design of flash-based key-value caching. Ouyang et al. present SDF [42] for web-scale storage. Wang et al. further present a design of LSM-tree based key-value store on the same platform, called LOCS [57]. KAML [23] presents a key-addressable multi-log SSD which exposes a key-value interface to enable applications to make use of internal parallelism of flash channels through using Open-Channel SSD. This paper and its earlier version [52] present DIDACache, an Open-Channel based solution that deeply integrates device- and application-level semantics. We share the common principle of bridging the semantic gap and aim to deeply integrate device and key-value cache management.

8 CONCLUSIONS

Key-value cache systems are crucial to low-latency high-throughput data processing. In this paper, we present a co-design approach to deeply integrate the key-value cache system design with the flash hardware. Our solution enables three key benefits, namely a single-level direct mapping from keys to physical flash memory locations, a cache-driven fine-grained garbage collection, and an adaptive over-provisioning scheme. We implemented a prototype on real Open-Channel SSD hardware platform. Our experimental results show that we can significantly increase the throughput by 35.5%, reduce the latency by 23.6%, and remove unnecessary erase operations by 28%.

Although this paper focuses on key-value caching, such an integrated approach can be generalized and applied to other semantic-rich applications. For example, for file systems and databases, which have complex mapping structures in different levels, our unified direct mapping scheme can also be applied. For read-intensive applications with varying patterns, our dynamic OPS approach would be highly beneficial. Various applications may benefit from different policies or different degrees of integration with our schemes. As our future work, we plan to further generalize some functionality to provide fine-grained control on flash operations and allow applications to flexibly select suitable schemes and reduce development overheads.

ACKNOWLEDGMENTS

We thank our shepherd, Gala Yadgar, and the anonymous reviewers for their constructive comments. This work is partially supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 152223/15E, GRF 152736/16E, GRF 152066/17E), Louisiana Board of Regents LEQSF(2014-17)-RD-A-01, and U.S. National Science Foundation (CCF-1453705, CCF-1629291).

REFERENCES

[1] *Fatcache-Async.* https://github.com/polyu-szy/Fatcache-Async-2017.

[2] *Whitepaper: memcached total cost of ownership (TCO).* https://goo.gl/SD2rZe.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. 2008. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference (ATC 08).*

[4] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. 2010. Cheap and large CAMs for high performance data-intensive networked systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 10).*

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review (SIGMETRICS 12).*

[6] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *USENIX Conference on File and Storage Technologies (FAST 17).*

[7] Damiano Carra and Pietro Michiardi. 2014. Memory partitioning in Memcached: an experimental performance analysis. In *International Conference on Communications (ICC 14).*

[8] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 09).*

[9] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011a. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *International Symposium on High Performance Computer Architecture (HPCA 11).*

[10] F. Chen, T. Luo, and X. Zhang. 2011b. CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *USENIX Conference on File and Storage Technologies (FAST'11).*

[11] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 11).*

[12] C. Dirik and B. Jacob. 2009. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, architecture, and system organization. In *International Symposium on Computer Architecture (ISCA 09).*

[13] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2017. Flashield: a key-value cache that minimizes writes to flash. *arXiv preprint arXiv:1702.02588* (2017).

[14] Facebook. *McDipper: a key-value cache for flash storage.* https://goo.gl/ZaavWa.

[15] E. Gal and S. Toledo. 2005. Algorithms and data structures for flash memories. In *ACM Computing Survey (CSUR)*, Vol. 37:2.

[16] Salil Gokhale, Nitin Agrawal, Sean Noonan, and Cristian Ungureanu. 2010. KVZone and the search for a write-optimized key-value store.. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 10)*.

[17] Javier González, Matias Bjørling, Seongno Lee, Charlie Dong, and Yiren Ronnie Huang. 2016. Application-driven flash translation layers on Open-Channel SSDs. (2016).

[18] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. 2009. Characterizing flash memory: anomalies, observations, and applications. In *International Symposium on Microarchitecture (Micro 09)*.

[19] Yong Guan, Guohui Wang, Yi Wang, Renhai Chen, and Zili Shao. 2013. BLog: block-level log-block management for NAND flash memorystorage systems. In *Annual ACM SIGPLAN / SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 17)*.

[20] A. Gupta, Y. Kim, and B. Urgaonkar. 2009. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 09)*.

[21] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Cheng Ding, Song Jiang, and Zhenlin Wang. 2015. LAMA: optimized locality-aware memory allocation for key-value cache. In *USENIX Annual Technical Conference (ATC 15)*.

[22] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K Qureshi. 2017. FlashBlox: achieving both performance isolation and uniform lifetime for virtualized SSDs. In *USENIX Conference on File and Storage Technologies (FAST 17)*.

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

[24] Steve Schlosser Greg Ganger John Bucy, Jiri Schindler. *DiskSim 4.0.* http://www.pdl.cmu.edu/DiskSim/.

[25] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The Multi-streamed solid-state drive.. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*.

[26] Ana Klimovic, Christos Kozyrakis, Eno Thereksa, Binu John, and Sanjeev Kumar. 2016. Flash storage disaggregation. In *The Eleventh European Conference on Computer Systems (EuroSys 16)*.

[27] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *ACM Association for Computing Machinery Special Interest Group on Computer Architecture (SIGARCH 09)*.

[28] Adam Leventhal. 2008. Flash storage memory. In *Communications of the ACM*, Vol. 51(7). 47–51.

[29] Paul Lilly. 2013. *Facebook ditches DRAM, flaunts flash-based McDipper.* http://www.maximumpc.com/facebook-ditches-dram-flaunts-flash-based-mcdipper.

[30] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. 2011. SILT: a memory-efficient, high-performance key-value store. In *ACM Symposium on Operating Systems Principles (SOSP 11)*.

[31] Duo Liu, Tianzheng Wang, Yi Wang, Zhiwei Qin, and Zili Shao. 2011. PCM-FTL: A write-activity-aware NAND flash memory management scheme for PCM-based embedded systems. In *IEEE Real-Time Systems Symposium (RTSS 11)*.

[32] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. WiscKey: separating keys from values in SSD-conscious storage. In *USENIX Conference on File and Storage Technologies (FAST 16)*.

[33] Fabio Margaglia, Gala Yadgar, Eitan Yaakobi, Yue Li, Assaf Schuster, and André Brinkmann. 2016. The devil is in the details: implementing flash page reuse with WOM codes. In *USENIX Conference on File and Storage Technologies (FAST 16)*.

[34] Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015a. NVMKV: a scalable and lightweight, FTL-aware key-value store. In *USENIX Annual Technical Conference (ATC 15)*.

[35] Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. 2015b. NVMKV: a scalable and lightweight flash aware key-value store. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*.

[36] B. Marsh, F. Douglis, and P. Krishnan. 1994. Flash memory file caching for mobile computers. In *Hawaii Conference on Systems Science*.

[37] Memblaze. *Memblaze.* http://www.memblaze.com/en/.

[38] Memcached. *Memcached: a distributed memory object caching system.* http://www.memcached.org.

[39] Michael P. Mesnier, Jason Akers, Feng Chen, and Tian Luo. 2011. Differentiated storage services. In *ACM Symposium on Operating System Principles (SOSP 11)*.

[40] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling

memcache at facebook. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*.

[41] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. 2012. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems.. In *USENIX Conference on File and Storage Technologies (FAST 12)*.

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

[43] Xiangyong Ouyang, Nusrat S. Islam, Raghunath Rajachandrasekar, Jithin Jose, Miao Luo, Hao Wang, and Dhabaleswar K. Panda. 2012. SSD-assisted hybrid memory to accelerate memcached over high performance networks. In *International Conference for Parallel Processing (ICPP 12)*.

[44] Zhiwei Qin, Yi Wang, Duo Liu, Zili Shao, and Yong Guan. 2011. MNFTL: An efficient flash translation layer for MLC NAND flash memory storage systems. In *Proceedings of the 48th Design Automation Conference (DAC 11)*.

[45] Redis. http://redis.io/.

[46] M. Rosenblum and J. K. Ousterhout. 1992. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems (TC 92)*, Vol. 10(1):26-52.

[47] SamSung. *Samsung 840 Pro*. https://www.cnet.com/products/samsung-840-pro-ssd/.

[48] Mohit Saxena, Michael M Swift, and Yiying Zhang. 2012. Flashtier: a lightweight, consistent and durable storage cache. In *The European Conference on Computer Systems (EuroSys 12)*.

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

[50] Mansour Shafaei, Peter Desnoyers, and Jim Fitzpatrick. 2016. Write amplification reduction in flash-based SSDs through extent-based temperature identification. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*.

[51] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. 2016. Optimizing flash-based key-value cache systems. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*.

[52] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. 2017. DIDACache: A Deep Integration of Device and Application for Flash Based Key-Value Caching. In *USENIX Conference on File and Storage Technologies (FAST 17)*.

[53] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. 2010. Extending SSD lifetimes with disk-based write caches. In *USENIX Conference on File and Storage Technologies (FAST 10)*.

[54] T13. *T13 documents referring to TRIM*. https://goo.gl/5oYarv.

[55] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. RIPQ: advanced photo caching on flash for facebook. In *USENIX Conference on File and Storage Technologies (FAST 15)*.

[56] Twitter. *Fatcache*. https://github.com/twitter/fatcache.

[57] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2015. An efficient design and implementation of LSM-tree based key-value store on Open-Channel SSD. In *The European Conference on Computer Systems (EuroSys 15)*.

[58] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: an LSM-tree-based ultra-large key-value store for small data items. In *USENIX Annual Technical Conference (ATC 15)*.

[59] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Hao Tong, Swaminatahan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. 2017. Tiny-tail flash: near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *USENIX Conference on File and Storage Technologies (FAST 17)*.

[60] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. 2014. Don't stack your log on my log. In *Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*.

[61] Heng Zhang, Mingkai Dong, and Haibo Chen. 2016. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *USENIX Conference on File and Storage Technologies (FAST 16)*.

[62] Yiying Zhang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. De-indirection for flash-based SSDs with nameless writes. In *USENIX Conference on File and Storage Technologies (FAST 12)*.

[63] Yiying Zhang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. Removing the costs and retaining the benefits of flash-based SSD virtualization with FSDV. In *International Conference on Massive Storage Systems and Technology (MSST 15)*.

[64] Yiying Zhang, Gokul Soundararajan, Mark W Storer, Lakshmi N Bairavasundaram, Sethuraman Subbiah, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2013. Warming up storage-level caches with bonfire. In *USENIX Conference on File and Storage Technologies (FAST 13)*.

[65] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S Yang, Bill W Zhao, and Shashank Singh. 2014. Torturing databases for fun and profit. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*.

[66] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. 2013. Understanding the robustness of SSDs under power fault. In *USENIX Conference on File and Storage Technologies (FAST 13)*.