Optimizing Flash-based Key-value Cache Systems

Zhaoyan Shen[†] *Feng Chen*[‡] *Yichen Jia*[‡] *Zili Shao*[†]
[†]*Department of Computing* [‡]*Computer Science & Engineering*
Hong Kong Polytechnic University *Louisiana State University*

Abstract

Flash-based key-value cache systems, such as Facebook's McDipper [1] and Twitter's Fatcache [2], provide a cost-efficient solution for high-speed key-value caching. These cache solutions typically take commercial SSDs and adopt a Memcached-like scheme to store and manage key-value pairs in flash. Such a practice, though simple, is inefficient. We advocate to reconsider the hardware/software architecture design by directly opening device-level details to key-value cache systems. This co-design approach can effectively bridge the semantic gap and closely connect the two layers together. Leveraging the domain knowledge of key-value caches and the unique device-level properties, we can maximize the efficiency of a key-value cache system on flash devices while minimizing its weakness. We are implementing a prototype based on the Open-channel SSD hardware platform. Our preliminary experiments show very promising results.

1 Introduction

High-speed key-value caches, such as Memcached and Redis, are the "first line of defense" in today's low-latency Internet services. Traditionally, these in-memory key-value caches heavily rely on large amount of expensive and power-hungry DRAM. In order to lower the Total Cost of Ownership (TCO), a more cost-efficient alternative, *flash-based key-value cache*, has recently raised a high interest in the industry [1, 2]. Facebook, for example, deploys a flash-based Memcached-compatible key-value cache system, called McDipper [1]. 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 [3]. Twitter also has a similar flash-based key-value cache solution, called Fatcache [2].

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, such as organizing key-values into slabs of different size classes, and using in-memory hash table to maintain the key-to-value mapping, etc. Such a design, though simple, disregards an important fact – The key-value cache system and the underlying flash storage both have very *unique* properties. Simply treating the flash SSD 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 problems, namely *redundant mapping*, *double garbage collection*, and *over-overprovisioning*. In this study, we advocate to reconsider the current software/hardware architecture for designing an efficient key-value cache system, highly optimized for flash.

2 Background and Motivation

2.1 Flash-based key-value caches

The existing flash-based key-value cache system design is fairly similar to its in-memory counterpart – both use a slab-based space management. Here we use Twitter's Fatcache [2] as an example for explanation:

The flash SSD space is first segmented into *slabs*. Each slab is often of several Megabytes and further divided into an array of *slots* (a.k.a. chunks) of equal size. Each slot stores a "value" item. Slabs are logically organized into different *slab classes* based on the slot sizes. An incoming value item is stored in a slab whose slot size is the best fit of its size. For quick accesses, a *hash mapping table* is maintained in memory to map the keys to the slabs that contain the corresponding 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 to a new location and updating the mapping table entry. Deleting a key-value pair (`delete`) simply removes the mapping from the hash table. The deleted or obsolete value items are left for garbage collection (GC) later. The current design has three critical problems, which have motivated us to perform this study.

2.2 Critical Issues

• **Problem 1: Redundant mapping**. Modern flash SSDs implement a complex Flash Translation Layer (FTL) in the firmware. A key function of FTL is to translate Logical Block Addresses (LBA) to Physical Flash Memory Pages. Although a variety of mapping schemes exist [8], for performance reasons, high-end SSDs often adopt *page-level mapping* for a fine-grained logical-to-physical address translation. 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 DRAM on device raises not only production cost and also reliability concerns upon power failures. In the meantime, at the application level, the key-value cache system also manages another mapping structure, the in-memory hash table, which translates a hashed key to the corresponding slab block. These two mapping structures co-exist at the two levels simultaneously, which unnecessarily doubles the memory consumption and also incurs high complexity and other issues. The problem essentially stems from the fact that the generic page-level mapping is designed for general-purpose file systems rather than key-value cache systems. In a typical key-value cache, for example, the slab size is typically in Megabytes, which is 100-1,000 times larger than flash page size. It means that the fine-grained page-level mapping scheme adopted in the FTL of high-end flash SSDs is simply an *expensive over-kill*. If we could directly map the hashed keys to the physical locations in flash, we can completely remove this redundant and highly inefficient mapping for lower cost, simpler design, and improved performance.

• **Problem 2: Double garbage collection**. Garbage collection (GC) is a well-known performance bottleneck of flash devices [7]. In flash memory, the smallest read/write unit is a page (e.g., 4KB), and a page cannot be overwritten in place until the entire *erase block* (e.g., 256 pages) is erased. As so, upon a write, the FTL simply marks the obsolete pages as "invalid" and writes the data to another physical location. At a later time, a GC procedure is scheduled to recycle the invalidated pages and maintain 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 could take as much as dozens to hundreds of milliseconds to complete, especially for large erase blocks. The key-value cache, similarly, also has a GC procedure, which recycles the slab space occupied by obsolete or deleted key-value pairs, whose size could be much smaller than flash pages.

These two independent GC processes are redundant and could interfere with each other. For example, from the device FTL's perspective, it is unaware of the semantic meaning of page content. Even a flash page contains no valid key-value pairs, the entire flash page appears to be "valid" at the device level, as long as the page is not trimmed explicitly. During the FTL-level GC, this page has to be copied and moved, though unnecessarily. Also, since the FTL-level GC has to assume all valid pages contain useful content, it cannot selectively recycle the semantically valid entries, not to mention aggressively invalidating certain semantically unimportant data. For example, even if a page contains only one valid key-value pair, the entire page is still regarded as valid and cannot be erased, although the key-value cache may consider it as a removable item. Also note that `TRIM` command [4] cannot address this issue. If we could merge the two-level GCs and control the GC process based on the semantic knowledge of the key-value cache, we can completely remove all the abovesaid inefficient operations and create new optimization opportunities.

• **Problem 3: Over-over-provisioning**. In order to minimize the performance impact of GC to foreground I/Os, the device FTL typically reserves a portion of flash memory space, 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, however, such a 20-30% OPS space is nothing but an expensive unusable space. We should note that the static factory setting of OPS in SSDs is mostly based on a conservative estimation for being prepared to handle the worst case (i.e., highly intensive write traffic). Unfortunately, key-value cache systems are often read-intensive [6]. Reserving such a large portion of flash memory space is a significant resource waste, considering the high cost of flash memory. In the meantime, as key-value cache systems possess rich knowledge about the I/O patterns and are capable of accurately estimating the incoming write intensity, and based on such estimation, a reasonable OPS could be determined during runtime for maximizing the usable flash space. Considering the importance of cache size to the cache hit ratio, if we could leverage the domain knowledge of the key-value cache systems to determine the optimal (minimized) OPS size, such a 20-30% extra space could be saved to significantly improve cache hit ratio and system performance.

Essentially, all the above-said issues stem from a fundamental problem in the current architectural design: The key-value cache manager runs at the application level and views the storage abstraction simply as a sequence of sectors; the flash memory manager (i.e., the FTL) runs at the device layer and views incoming requests simply as a sequence of individual I/Os. This, unfortunately, creates a huge *semantic gap* between the key-value cache and the underlying flash storage. In the next section, we will describe our cohesive design approach to bridge this semantic gap and build a highly optimized flash-based key-value cache system.

3 Design

3.1 Overview

Our design consists of three main layers (see Figure 1): (1) *An enhanced flash-aware key-value cache*, which is highly optimized for flash memory storage, runs at the application level, and directly drives the flash management; (2) *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`). (3) *A*

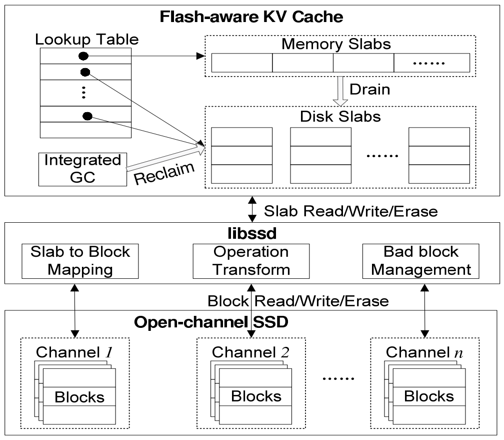

Figure 1: The Architecture Overview.

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.

3.2 Application level: Key-value Cache

The key-value cache runs at the application level. For space limit, here we only discuss three key aspects that differentiate our design from the conventional one.

• **Unified Direct Mapping**. In order to address the double mapping issue, a key change made in our design 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. This is feasible, since the underlying library layer exposes the flash memory space in an abstraction of slabs to the upper-level applications. An incoming request simply looks up the hash table, retrieves the mapping information, and sends `read` command via the library layer to load the data from flash.

This unified direct mapping not only removes the time overhead for the intermediate mapping but also dramatically reduces the demand for on-device DRAM buffer – Multiple mapping tables are collapsed into one single must-have in-memory hash table, and the large FTL-level mapping table can be completely removed from the device. By removing a page-level mapping, we could save hundreds of Megabytes to even Gigabytes on-device DRAM space, which could either reduce production cost or allow a better use of on-device DRAM, such as on-device caching/buffering.

• **Slab Management**. Similar to Memcached, our key-value cache system also adopts a slab-based space management scheme. In our design, the flash space is divided into equal-sized slabs. Each slab is statically mapped to a flash memory erase block (8MB). When mapping the slab to flash blocks, we have considered two possible mappings, *per-channel mapping* and *cross-channel mapping*. The former maps a slab to one channel, while the latter maps a slab to multiple channels in a round-robin way. Although the latter may yield better bandwidth for one single slab access, it causes the multi-block pollution problem upon updates. Considering that a typical key-value cache has sufficient slab-level parallelism, we choose the simpler per-channel mapping, which also allows us to easily infer the relationship between slabs (in different channels or not) and to support multiple key-value cache instances by separating channels.

To handle the "no in-place overwrite" constraint, we maintain an in-memory slab buffer for each slab class. An incoming `PUT` of a key-vale pair is first stored in an in-memory slab, according to its value size, and when the in-memory slab buffer is full, the entire slab is flushed to an in-flash slab for persistent storage. For cross-channel load balance, we maintain a queue of free slabs for each channel and select the queue for allocation in a round-robin manner. On each queue the slab with the smallest erase count is first selected for wear-leveling purposes. When the system runs out free slabs, the key-value cache starts the GC procedure to reproduce clean slabs. Thus, from the device's perspective, all I/Os seen at the device level are in large-size slabs, which completely removes the need for a generic GC at the FTL level.

• **Garbage Collection**. Garbage collection (GC) in our design is application driven and there is no device-level GC. When the system runs out free slabs, the key-value cache system will start a GC procedure. Given the semantic knowledge about the validity of slots, we can perform a more fine-grained GC in one single procedure. We have two strategies to identify a target slab for cleaning: (1) *Locality-based cleaning*, which selects the most seldomly accessed slab based on the Least Recently Used (LRU) order, and (2) *Space-based cleaning*, which selects the slab containing the largest number of obsolete value items. We apply the two policies depending on the runtime system condition: When the system is under significant pressure (e.g., when incoming requests are busy waiting for a clean slab), we use the former approach to quickly release the LRU slab space for fast response. An aggressive measure, called *quick clean*, can be applied by simply dropping the entire slab, including all valid slots. It is feasible, as our application is a cache – Clients are required to write key-values to the backend store first, so it is safe to aggressively drop any key-values. When the system is under light pressure, we use the space-based cleaning and try to retain all valid key-values in the cache and recycle as much invalidated slot space as possible. Once a victim slab is identified, we scan the slots and migrate valid ones to another slab and put the cleaned slab back to the queue.

• **Over-Provisioning Space Management**. OPS in our design is not of a fixed size. Using the flash space for caching allows us to dynamically adjust the usable cache space. We desire to maintain an OPS size just enough to handle incoming writes. In our current prototype, we adopt a self-tuning feedback-based solution. We set two watermarks, *low* (W_L) and *high* (W_H). Ideally, the size of the clean block pool should be between the two watermarks, which means that the speed of cleaning slabs is roughly equal to that of consuming them. Our self-tuning solution works as follows: When the low watermark is hit, we lift the low/high watermarks by doubling the value to quickly respond to increasing writes; when the high watermark is hit, we linearly drop the low/high watermarks to give more cache space back. In this way, we can keep the OPS space automatically adaptive to the incoming traffic. Another mathematical model based solution is under development.

3.3 Library Level: `libssd`

As an intermediate layer, the `libssd` library glues together the application and device layers. It provides an abstraction of flash memory space in the form of slabs and an API interface to directly drive the SSD operations. In particular, `libssd` has three main functions: (1) *Slab-to-block mapping*, which statically maps a slab to one (or multiple contiguous) flash memory blocks in a channel. Such a mapping can be calculated through a mathematical conversion and does not need another mapping table. (2) *Operation transformation*, which converts key slab operations, namely `read`, `write`, and `erase`, to flash memory operations, which allow the key-value cache system to operate in units of slabs, rather than flash memory pages/blocks. (3) *Bad block management*, which maintains a list of flash memory blocks that have been detected to be "bad" and ineligible for allocation, and hides them from the key-value cache.

3.4 Hardware Level: Open-Channel SSD

We use a customized SSD hardware similar to the Open-Channel SSD used in SDF [11]. This PCI-E 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, which exposes the flash space as 44 host-visible block devices, our SSD abstracts the flash memory space in 192 host-invisible LUNs and statically maps them over the channels. An `ioctl` interface is provided by the driver to directly operate flash memory pages by specifying the target LUN ID and the page number. Most FTL-level functions, such as address mapping, wear-leveling, and 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. Other core functions, such as error handling and flash control, are still handled by the device.

4 Preliminary Results

We implement a prototype of the proposed flash-based key-value cache system on the Open-Channel SSD hardware, denoted as "Open-Channel". Our implementation of flash-aware key-value cache layer is based on Twitter's Fatcache [2]. Here we present some preliminary results to illustrate system basic performance and compare the results with running the stock Fatcache on two commercial SSDs, a 128GB Samsung 850 Pro and a 120GB KingSpec PCIE-2u SSD.

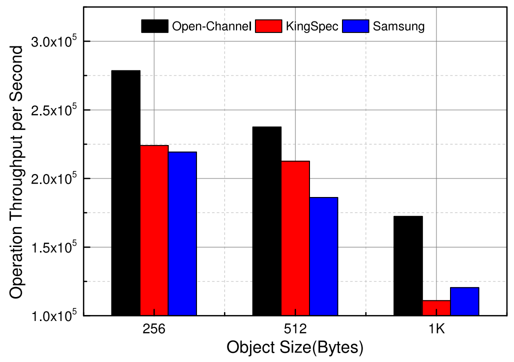

Figure 2: Set Throughput (Operations per Second)

We first measure the throughput of `set` and `get` key-value items of varied sizes. We use the *twemperf* benchmark [13] from Twitter to issue set operations with key-value item sizes of 256 bytes, 512 bytes and 1 KB to the key-value cache. Figure 2 shows the throughput (operations per second). We can see that with 1KB items our cross-layer solution can effectively achieve a throughput as high as 172,335 ops/sec. Compared to the conventional solution, our cross-layer solution is 55% and 43% higher than running the stock Fatcache on KingSpec and Samsung. For get operations, we find that its throughput is mostly bounded by the read bandwidth of the SSDs, so we omitted it here due to space constraint.

Table 1: Block Erase Count

SSDs	**KV Cache Instance**				
	1	**2**	**4**	**6**	**8**
Open-Channel	2492	2465	2440	1952	1938
SSDSim	2884	2861	2842	2815	2797

The second experiment is to compare erase counts of our cross-layer solution and the conventional solution. In our cache-driven system, erase operations can be easily counted in the library code. However, we cannot directly obtain erase counts on the two commercial SSDs. So, we use the SSD simulator from Microsoft Research [5] and configure it with the same parameters of the Open-Channel SSD. Then we first repeat the test by running stock Fatcache on the Samsung SSD and collect the trace by using the *blktrace* tool and then run the simulation with the trace. So, the simulator simulates a conventional solution and we compare it with our solution. To limit the test duration, we confine the available SSD size to 30GB, and preload it with 14GB data of 1KB items. The workload has 30 million records (30GB data) to overwrite all the items based on a truncated Normal distribution.

Table 1 shows block erase counts. As we can see, GC in our design is application driven and can directly exploit available semantic knowledge. By removing the double GC problem and working at a fine granularity, our GC is much more efficient. Compared to the conventional solution, our design can get about 30% reduction of erase counts. To test with a high I/O pressure, we increase the number of cache instances from 1 to 8, and interestingly, we find that as the instance number increases, erase count decreases. This is because when the system is under significant pressure, our aggressive GC policy, *quick clean*, kicks in at the 6-instance case. As no extra slot copies are needed, the block erase count is reduced from 2,440 to 1,952, showing its effectiveness.

5 Other Related Work

Both flash memory and key-value systems are hot research topics. Due to space constraint, we only present the most related work here. The effect of redundant operations in software layers has been studied before. For example, Yang et al. have pointed out that redundant structures and functions in multiple layers of logs could result in negative effects on flash devices, such as increased write pressure, and should be carefully handled [14]. Nameless Writes [15] presents a new device interface to allow the device to directly control block allocation. FSDV [16] attempts to remove the mapping redundancy by directly storing physical flash addresses in the file systems. Our solution shares the same *de-indirection* principle as these prior studies. Another set of related work is flash-based key-value stores. For example, SILT [9] optimizes the key-value store with three basic stores for high memory efficiency. NVMKV [10] is a light-weight key-value store that builds upon an enhanced FTL. NVMKV extends FTL primitives for simplifying the key-value store design. Our approach is different. We aim to simplify the device FTL design and expand the capabilities of the key-value cache, such as directly driving GC. Another key distinction is that our work is on a key-value cache rather than a persistent store, which offers unique optimization opportunities, such as quick clean. Some other prior research has also studied hybrid memory solution for key-value caches. For example, Ouyang et al. propose to adopt commercial flash SSDs to expand RAM space for Memcached in HPC systems [12]. Our design is a cohesive whole-system solution.

6 Conclusions

Building a highly efficient flash-based cache system is challenging, because of the huge semantic gap between the applications and the devices. In this paper, we present a holistic design, which enables three key benefits, namely a unified single-level direct mapping, a cache-driven fine-grained garbage collection, and an adaptive over-provisioning scheme. We are implementing a prototype on the Open-Channel SSD hardware and our preliminary results show that it is highly promising.

Acknowledgments

We thank anonymous reviewers for their constructive comments and feedback. This work is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 152138/14E and GRF 15222315/15E), National Natural Science Foundation of China (Project 61272103 and Project 61373049), National 863 Program 2013AA013202, Hong Kong Polytechnic University (4-ZZD7,G-YK24, G-YM10 and G-YN36), Louisiana Board of Regents LEQSF(2014-17)-RD-A-01, and U.S. National Science Foundation (CCF-1453705).

References

[1] https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920.

[2] https://github.com/twitter/fatcache.

[3] http://www.maximumpc.com/facebook-ditches-dram-flaunts-flash-based-mcdipper.

[4] http://t13.org/Documents/MinutesDefault.aspx?keyword=trim.

[5] http://research.microsoft.com/en-us/downloads/b41019e2-1d2b-44d8-b512-ba35ab814cd4/.

[6] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload Analysis of a Large-scale Key-value Store. In *ACM SIGMETRICS Performance Evaluation Review* (2012), ACM.

[7] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. In *Proceedings of SIGMETRICS'09* (Seattle, WA, June 2009).

[8] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proc. of ASPLOS'09* (2009).

[9] LIM, H., FAN, B., ANDERSON, D. G., AND KAMINSKY, M. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (Cascais, Portugal, October 23-26 2011).

[10] MÁRMOL, L., SUNDARARAMAN, S., TALAGALA, N., AND RANGASWAMI, R. NVMKV: A Scalable and Lightweight, FTL-aware Key-Value Store. In *Prceedings of USENIX ATC'15* (2015).

[11] OUYANG, J., LIN, S., JIANG, S., HOU, Z., WANG, Y., AND WANG, Y. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *Proceedings of ASPLOS'2014* (Salt Lake City, UT, March 1-5 2014).

[12] OUYANG, X., ISLAM, N. S., RAJACHANDRASEKAR, R., JOSE, J., LUO, M., WANG, H., AND PANDA, D. K. SSD-Assisted Hybrid Memory to Accelerate Memcached over High Performance Networks. In *Prceedings of ICPP'12* (Pittsburgh, PA, Sept 10-13 2012).

[13] TWITTER. Twemperf. https://github.com/twitter/twemperf.

[14] YANG, J., PLASSON, N., GILLIS, G., TALAGALA, N., AND SUNDARARAMAN, S. Don't Stack Your Log on My Log. In *Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)* (Broomfield, CO, October 5 2014).

[15] ZHANG, Y., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of FAST'12* (2012).

[16] ZHANG, Y., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Removing the Costs and Retaining the Benefits of Flash-Based SSD Virtualization with FSDV. In *Proceedings of MSST'15* (Santa Clara, CA, June 2015).