All-Flash Array Key-Value Cache for Large Objects

Jinhyung Koo[1], Jinwook Bae[1], Minjeong Yuk[1], Seonggyun Oh[1], Jungwoo Kim[1]
Jung-Soo Park[2], Eunji Lee[3], Bryan S. Kim[4], Sungjin Lee[1]

[1]DGIST, [2]WineSOFT, [3]Soongsil University, and [4]Syracuse University

Abstract

We present BigKV, a key-value cache specifically designed for caching large objects in an all-flash array (AFA). The design of BigKV is centered around the unique property of a cache: since it contains a copy of the data, exact bookkeeping of what is in the cache is not critical for correctness. By ignoring hash collisions, approximating metadata information, and allowing data loss from failures, BigKV significantly increases the cache hit ratio and keeps more useful objects in the system. Experiments on a real AFA show that our design increases the throughput by 3.1× on average and reduces the average and tail latency by 57% and 81%, respectively.

CCS Concepts: • **Information systems → Data management systems**; **information storage systems**.

Keywords: key-value caches, all-flash arrays, and SSDs

1 Introduction

Recently, as the quality of web content improves, the demand for caching large objects continues to increase. Fig. 1 illustrates the size distribution of objects that were collected for a week in 2021 from production content delivery network (CDN), image, and document servers. The majority of objects are 8KB–32KB in size for CDN servers, 32KB–128KB for image servers, and 1KB for document servers. If DRAM-based key-value (KV) caches were to store these large objects, the system would either be (*i*) ineffective, as large objects would quickly exhaust the DRAM's capacity [6], or (*ii*) costly, as hundreds of servers should be deployed to create a distributed DRAM cache pool. As a result, there exists a strong need to build a KV caching system specifically tailored for large objects, a niche that DRAM cannot fill.

As opposed to DRAM, NAND flash technology offers a high-performance yet cost-effective solution that makes caching large objects viable. The latest all-flash array (AFA) servers embrace many SSDs (up to 84 [12]), providing petabyte-scale capacity and tens of GB/s throughput [13]. NetApp's AFA server offers up to 1.6PB capacity with 128GB DRAM [44]. DELL-EMC's AFA provides 4PB capacity with 256GB DRAM [13]. To the best of our knowledge, however, there are no comprehensive studies on how to build an efficient KV cache over an AFA.

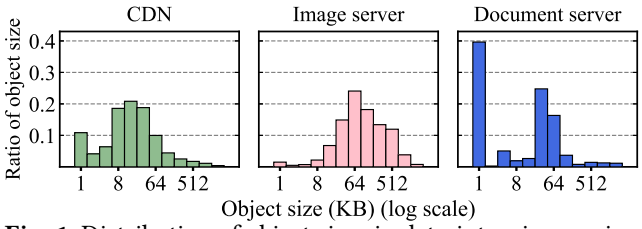


Fig. 1. Distribution of object sizes in data-intensive services

Prior studies on domain-specific AFAs for RDBMS [32, 49] and NAS/SAN servers [30, 31] mainly focus on optimizing I/O stacks to deliver high performance of SSDs [31] while ensuring data persistence and durability. These design criteria significantly differ from those for caching systems. In terms of performance, a cache hit ratio plays a significant role because a cache miss incurs very long over-the-network latency (*i.e.,* a high miss penalty). For data persistence and durability, they are not as important as in RDBMS and NAS/SAN because the copies of cached objects are safely kept elsewhere.

Unfortunately, attaining a high hit ratio in an AFA-scale KV cache is more elusive than it appears to be. First, the number of objects that can be cached in the AFA is limited by the system's DRAM. Typical flash KV caches maintain a DRAM-resident hash table, each entry of which keeps a per-object metadata to index a KV object. The size of the per-object metadata ranges from 8B to 48B [28, 33, 54, 55, 61]. Given that the AFA has 1PB SSDs with 128GB DRAM and the size of objects is 32KB on average, only at most half of the AFA capacity can be used for caching objects. This inevitably lowers the hit ratio because objects in the rest of the half cannot be indexable. Caching only hot entries in DRAM is possible, but naive hash table caching results in extra I/Os when a desired entry is not found in DRAM (see §3.1).

Second, a large amount of the AFA space is occupied by expired objects. In caching systems, an object is tagged with a Time-To-Live (TTL) that specifies the object's lifetime. TTL-expired objects must be inaccessible and properly cleaned up to reclaim free space. Unfortunately, in an AFA-scale cache where numerous objects are stored, proactively reclaiming space taken by TTL-expired objects is expensive as it requires scanning slow SSDs. On the other hand, its lazy alternative

EuroSys '23, May 9–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567509

that removes expired objects only when they are identified leads to inefficient use of cache capacity, which results in a high miss ratio (see §3.2).

Lastly, the AFA cache inevitably suffers from frequent SSD breakdowns. To provide a seamless caching service in the presence of failures, parity-based protection techniques like RAID are commonly used. RAID on SSDs, however, is detrimental in both performance and capacity since it involves extra I/Os to record parities and requires additional space that reduces effective cache capacity [3, 27, 43]. If we group 8× SSDs using RAID-6, 25% of the AFA space must be used for storing parity blocks (see §3.3).

In this study, we propose BigKV, an AFA-based KV system for caching large objects. In solving the challenges for increasing the cache hit ratio, we exploit the unique property specific to caching systems: a loss or absence of objects does not lead to catastrophic consequences as they do in RDBMS or NAS/SAN. By doing so, our design can efficiently index a large number of objects with small DRAM usage, prevent space wasted by expired objects, and provide uninterrupted caching services without relying on costly RAID. The unique features of BigKV are the following.

- We design a *collision-oblivious two-level hash table* that is organized with two levels, uses a compact 16B per-object metadata, and ignores collisions. BigKV keeps the entire hash table in the SSDs and caches only hot entries in a DRAM-resident set-associative cache. Our table design tolerates data loss due to hash collisions. This avoids extra I/Os required for collision handling, thereby greatly improving I/O throughput. Our design also bounds the number of I/Os to at most two even when object lookup misses in the DRAM cache, improving tail latency (§4.2).
- We implement *approximate-TTL management* that allocates objects to a space that already stores objects with similar TTLs. We only use 5 bits to represent a TTL, while the exact TTL needs 4B. This coarse-grained binning allows BigKV to quickly identify expired objects at near-zero cost, using the cache capacity efficiently (§4.3).
- We use a *reactive fault tolerance mechanism* across a sharded cache space, handling failures only once they manifest. By doing so, BigKV maintains high availability by isolating shards from a failure but without the overhead associated with parities. Data loss due to a shard failure is tolerated, and overall, this benefits the system by continued uptime and background recovery at no upfront cost for computing and writing parity data (§4.4).

We implement BigKV in an AFA machine that employs two Intel's Xeon Gold 6152 CPUs, PM9A3 3.84TB NVMe SSDs, and 64GB DRAM. We conduct a set of experiments using real-world workloads from YCSB [10] and Twitter traces [53, 62]. Our results show that, compared with four state-of-the-art KV caches [33, 47, 54, 55], BigKV provides 3.1× higher throughput and 57% lower read latency while offering 81% shorter tail in YCSB [10] workloads, on average. By quickly removing expired objects and not maintaining any parity blocks, BigKV can achieve the same hit ratios as those of existing techniques using half the cache capacity.

Table 1. Cost comparison of DRAM and AFA systems

$/GB: DRAM (0.42), Flash (0.01)			mW/GB: DRAM (11.72), Flash (0.73)		
Server	# of devices	Capacity	Space	Power	Cost
1 DRAM	32 DIMMs	8TB	2U	2,496 W	$15,270
161 DRAM	5,152 DIMMs	1,288TB	322U	401,856 W	$2,458,470
1 AFA	84 SSDs	1,290TB	5U	3,166 W	$67,546
×64 space, ×127 power, ×36 cost efficiency					

This paper is organized as follows: In §2, we review background and prior studies. We explain key motivations in §3 and present the design of BigKV in §4. We provide the experimental results in §5, and we conclude the study in §6.

2 Background and Related Work

In this section, we introduce all-flash array systems and explain prior work closely related to this study.

2.1 NAND Flash and All-Flash Array

Thanks to advances in non-volatile memory technologies [5, 46, 48], the capacity and performance of flash SSDs have improved rapidly. In 2013, the largest SSD was 1TB [16]; however, as of 2021, a 100TB SSD [11] is available in markets. The throughput of SSDs was 712 MB/s in 2009 [60], but the latest high-end SSDs are now able to offer higher than 7 GB/s throughputs [1, 50]. Such improvements on the storage side push industries to release high-bandwidth PCIe interconnects at a faster rate. The bandwidth of a single PCIe lane, which was 250 MB/s in 2003 (PCIe 1.0), has improved to 8 GB/s in 2021 (PCIe 6.0) [51]. At the same time, the number of PCIe lanes supported by CPUs has increased from 8 (Intel's P55) [25] to 64 (Intel's Xeon Gold 5315Y) [26].

Those innovations make it feasible to build an ultra-capacity all-flash array (AFA). Just by plugging 32× 32TB SSDs into a motherboard, one can build a single AFA machine with a petabyte capacity. By using ultra-scale AFAs as KV caches, we can speed up web services by placing large objects closer to clients. It also gives us opportunities to reduce the total cost of ownership (TCO) by consolidating many servers into a few, by reducing network facilities (*e.g.,* switches), and by lowering maintenance costs. Table 1 shows the acquisition and maintenance costs of DRAM servers [14] and an AFA server [12]. Compared with a DRAM-based solution that requires 161 servers and 2.5 million dollars and consumes 400KW of power, an AFA-based solution can build up a 1.3PB KV cache in a single machine, offering 64×, 127×, and 36× higher efficiency in terms of space, power, and cost, respectively.

Unfortunately, transforming a bare-metal AFA machine into a KV caching system requires significant effort because prior KV designs cannot be directly adopted for an AFA-scale caching system. In the following sections, we review previously proposed KV designs and their limitations.

2.2 Persistent Key-value Stores

KV stores are usually used as building blocks for managing and storing large amounts of data persistently [7]. They serve as a storage layer for RDBMS (*e.g.,* MySQL [41]), a metadata store for distributed file systems (*e.g.,* Ceph [58]), and a storage engine for distributed key-value stores (*e.g.,* ZippyDB [40]). Recent KV stores use SSDs as storage media by default for meeting performance needs of data-intensive applications [9, 39, 56].

Generally, persistent KV stores are based on LSM trees [4, 19–21, 34, 35]. The append-only property of LSM trees is suitable for the physical nature of persistent storage (*e.g.,* SSDs). Moreover, since LSM trees merge and sort objects by keys, they provide rich operations such as range queries and iterators over sorted objects, making it easier to support various applications, ranging from RDBMS to file systems.

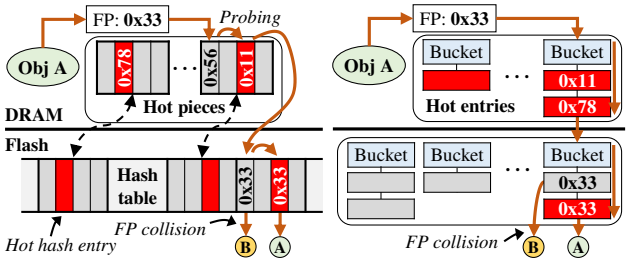
However, LSM-tree KV stores are rarely used as a back-end store for KV caching systems, except for a few cases (*e.g.,* Netflix [45]) [42]. KV stores are persistent object stores and assume that deletion happens occasionally. Under caching systems where objects are frequently deleted, KV stores suffer from severe write amplification [15, 42]. Periodic compaction that involves many I/Os also degrades caching performance. On the other hand, most requests destined for KV caches are point queries, which nullifies the necessity of costly compaction. As a result, it would not be preferred to run typical KV stores over the AFA to create a KV cache pool.

2.3 Key-value Caches for Flash

Compared with persistent KV stores, KV caches have a lean architecture. While detailed designs differ from one to the other, KV caches for flash commonly adopt *hashing* for managing metadata and *logging* for storing object data. This is reasonable because hashing is lightweight, simple, and sufficient to support point queries, and logging is suitable for flash that does not support in-place updates. Depending on the hashing strategy, existing KV caching systems can be broadly categorized into two types that use (*i*) chaining [28, 37, 42, 54, 55] or (*ii*) open addressing [33].

Fatcache [54] is the most well-known KV cache design that uses chaining. To locate objects in a log, it maintains a chained hash table in DRAM, each hash entry of which holds per-object metadata. For memory efficiency, Fatcache keeps minimal information in the metadata (*e.g.,* an object location in a log). Like other KV caches, it keeps a small-sized fingerprint (FP) of an object in the metadata and stores its full-length key with object data in the log. The per-object metadata, however, is still a very large 48B. For a 1PB AFA storing 32KB objects, the metadata size reaches 1.5TB. Fatcache also keeps a 4B TTL field in each entry but does not use it to remove expired objects proactively. This lazy reclamation causes cache space waste (10%~15%, see §3.2).

Many have attempted to improve Fatcache in several directions [28, 37, 55], but the most closely related study to ours is SlickCache [55]. SlickCache enhances Fatcache to tackle the memory pressure issue when DRAM is not sufficient to store the entire table. SlickCache addresses this issue by exploiting the locality of objects. It breaks the hash table into tiers, a hot tier in DRAM and a cold/warm tier in flash, and keeps hash entries in the right place according to their locality. Such tiering, however, results in long latency on a DRAM miss (see §3.1). For memory savings, SlickCache uses TTL fields for other purposes (*e.g.,* keeping reference bits).



(a) Open addressing hash table **(b)** Chained hash table

Fig. 2. Hash table caching strategies for existing KV caches

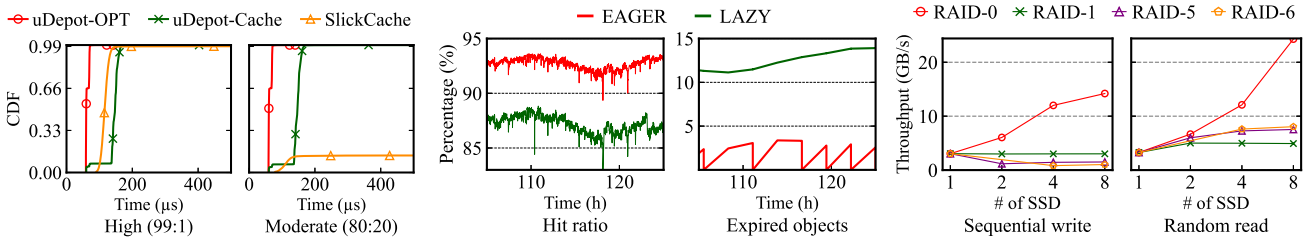
Kangaroo more aggressively reduces per-object metadata [42]. It splits storage space into log and data areas and then maintains chained hash tables only for the log area. The log area is small; it accounts for 5% of the storage space. This makes it possible to use small DRAM for its hash tables. The design of Kangaroo, however, is optimized for handling small objects. It suffers from serious read amplification when accessing large objects in the data area. This design is thus ill-suited for our use cases. Further, Kangaroo does not explicitly consider the TTLs of objects.

uDepot is designed to reap the performance of fast NVM storage (*e.g.,* Optane and NVMe SSDs) [33] by optimizing I/O stacks and task-based I/O scheduling. uDepot differs from Fatcache and its variants in that it uses open addressing to resolve hash collisions. This design reduces per-object metadata to 8B and eliminates pointers needed for chaining. However, it still requires significant DRAM; 256GB DRAM for 1PB AFA with 32KB objects. uDepot keeps the TTLs of objects in SSDs and removes expired objects when they are referenced. This design is good for memory savings but makes it difficult to identify expired objects unless they are read from the SSDs.

Finally, except for uDepot, existing KV caches target a system with few SSDs. To support fault tolerance under many SSDs, they must use a RAID-based protection technique. uDepot is designed to scale with many SSDs but relies on SPDK-RAID-0 for performance. Unless RAID is used with parity, providing a seamless caching service is difficult.

3 Motivation

In this section, we analyze the performance and present challenges of existing KV cache designs for an AFA.



(a) Lookup latency CDF of existing KV caching **(b)** Cache hit ratio and % of expired objects **(c)** Throughput of different RAID levels

Fig. 3. Motivational results

3.1 Huge Metadata and High Overhead

Existing KV caches attempt to relieve the memory pressure in several ways. Despite such efforts, the hash table is still too large to be loaded into DRAM. Further reducing per-object metadata size below 8B would be challenging; thus, it seems more reasonable to devise a way of efficiently caching hash entries by exploiting the temporal locality of objects.

We can take different approaches for chained and open-addressing hashing as shown in Fig. 2. Open-addressing hash tables (*e.g.,* uDepot) require probing on neighboring entries. To prevent flash access while hash probing, it is preferred to split the hash table into smaller pieces (*e.g.,* 16KB) and cache popular table pieces in DRAM (see Fig. 2(a)). However, this approach often fails to exploit the locality of objects. Owing to the randomness of hashing, hash entries are evenly distributed over the hash table. Even if one hash entry in the table piece is hot, the rest are likely to be cold. Even worse, costly in-flash table access is unavoidable if the probe distance goes beyond the cached table boundary.

For the chained hash table where hash entries are chained through a linked list (*e.g.,* Fatcache and its variants), we can cache only hot hash entries in DRAM (see Fig. 2(b)). This is what SlickCache does for caching hot entries [55]. The chained hash table achieves a higher hit ratio than open addressing but it suffers from long-tail latency. If the desired entry is not found in DRAM, it has to traverse the chain of the flash-resident hash table, which involves many I/Os. Implementing linked lists also requires extra DRAM, an 8B pointer per hash entry, reducing its memory efficiency.

A fingerprint (FP) collision further deteriorates the performance of both the chained and open-addressing tables. A fingerprint collision occurs when the fingerprints of objects are matched but their full-length keys differ. If a collision occurs while writing a new object, a wrong object is overwritten by the new one, which results in data loss. This problem cannot be avoided unless a lengthy full key is kept in the metadata. To prevent data loss, for every object write, we must fetch the object's header from the SSDs to see if a collision occurs. This degrades the write throughput, which is also important in KV caching systems [36, 63].

We carry out experiments using two workloads, one with high locality (99:1) and another with moderate locality (80:20), to evaluate the two caching strategies. We run the workloads on a system with a 1TB SSD and 128MB DRAM (see §5.1 for more details). We choose two KV designs, uDepot and SlickCache, which represent the open-addressing and chained hash tables, respectively. uDepot is designed to keep its entire hash table in DRAM. Since the DRAM of 128MB is not sufficiently large to load the entire table, we modify uDepot to cache popular table pieces as explained above (`uDepot-Cache`). `SlickCache` caches only hot hash entries in DRAM by design. Fig. 3(a) shows the results. `uDepot-OPT` (keeping the entire hash table in DRAM) exhibits the best performance. `uDepot-Cache` shows longer read latency. Particularly, its performance does not improve at all, even under the highly localized workload. `SlickCache` shows good read latency, but it suffers from unacceptably long tails when the locality is moderate.

The above results tell us that applying naive caching algorithms to existing KV cache designs does not work efficiently. To provide high caching performance, fundamental refactoring of existing hash tables is needed.

3.2 Space Waste by Expired Objects

TTLs are commonly used in caching systems to maintain data freshness through periodic re-computation, to comply with regulations (*e.g.,* GDPR [18]), and to delete useless objects. A TTL of an object determines how long the object should remain accessible, and a TTL-expired object must not be shown to clients. TTL-expired objects negatively affect caching performance as they unnecessarily occupy space, reducing effective cache capacity.

As explained in §2.3, existing KV caching systems remove expired objects *lazily* when they are referenced by clients. This approach does not work well in the AFA where numerous objects with diverse TTLs coexist (*i.e.,* short-lived objects and long-lived objects are mixed in the same cache space). Short-lived objects expire quickly. Until evicted, they take up space for an extended duration, early evicting other warm objects that are not very hot, yet occasionally accessed.

Fig. 3(b) shows our experimental results using real-world traces from Twitter [53, 62]. We compare two systems: one that eagerly removes expired objects (`EAGER`) and another that lazily removes them (`LAZY`). The traces were collected

over 7 days. A total of 12.5 billion objects are read; 0.5 billion objects are newly written or updated. Short-lived objects whose TTLs are shorter than 24 hours account for 65%. 7% of the objects last longer than 14 days. Compared with EAGER, LAZY shows a 5% lower cache hit ratio. According to our analysis, approximately 10-15% of the cache space is wasted by expired objects.

It is relatively easy to quickly identify and delete expired objects [47, 63] if TTLs always reside in DRAM. In our case, identifying expired objects is not easy as the metadata mostly stays in slow SSDs. Keeping all TTLs in DRAM is possible but makes the memory pressure more severe.

3.3 Fault Tolerance vs Parity Cost

In existing KV caches, providing robust persistence of objects in the event of disk or system failures is not a primary design concern since they cache objects temporarily. Thus, unless persistence is strongly needed by clients, existing KV caches do not explicitly cope with storage failures. However, simply neglecting failures leads to serious problems in the AFA-scale KV cache. The AFA employs a large number of SSDs. The more SSDs employed, the higher the probability of failures occurring. Without the proper handling of failures, the caching service is frequently interrupted.

Using RAID is a viable option to provide fault tolerance, but it comes at the expense of extra I/Os and space waste. We conduct experiments with three RAID setups to understand how significantly RAID affects performance. Using FIO [2], we measure the performance of random reads and sequential writes, which are dominant I/O patterns in KV caches. Fig. 3(c) shows the results. Only RAID-0, which does not provide fault tolerance, exhibits scalable performance with multiple SSDs. RAID-1/5/6 guarantee fault tolerance but perform poorly; their performance does not scale well with the number of SSDs because of extra I/Os and computation for parity blocks. The overhead associated with parity increases as the performance of SSDs improves. When we repeat the same experiments with slower SSDs, RAID-5/6 exhibit scalable performance (even though they are still slower than RAID-0). The same observation is also reported by [22, 29].

The three problems mentioned above also occur in a single-SSD caching system. However, their negative impacts become more severe in an AFA-scale caching system. First, while the maximum DRAM capacity is limited by the DIMM slots available in the CPUs, there is no limit on the number of SSDs that can be connected to the CPUs through PCIe switches. The gap between DRAM and SSD capacities becomes much larger in the AFA, which makes DRAM a more precious resource. Second, as the SSD capacity increases, KV objects with diverse lifespans are more likely to accumulate in the cache. Scanning the cache to identify TTL-expired objects takes longer in proportion to the number of SSDs attached. Lastly, it is obvious that device failure frequently occurs in the AFA. NetApp reports that a larger RAID group has a higher probability of experiencing device failures [38]. Compared with a RAID group with 4 SSDs, a RAID group with 24 SSDs suffers from 5× higher device replacements.

4 Design and Implementation of BigKV

BigKV is a new AFA-scale KV caching system. BigKV is designed to index billions of large objects over a large-scale AFA with minimal DRAM usage and maximize the cache hit ratio through efficient space management. Based on our observations in §3, we develop BigKV with the following three key techniques.

- **Collision-oblivious two-level hash table**. We design a two-level hash table that takes advantage of chaining and open-addressing and overcomes their drawbacks. BigKV has 16B per-object metadata. BigKV keeps only hot metadata in a DRAM-resident set-associative cache. This enables BigKV to achieve a high hit ratio without any pointer overhead. The entire metadata of BigKV is split and stored in small-sized (16KB) hash tables in flash, each of which is built using open addressing. On a cache miss, BigKV reads in the 16KB hash table and performs hash probing on it. The probe distance is limited within the table boundary. Thus, at most one extra I/O is needed upon a cache miss. BigKV ignores fingerprint collisions, which results in the loss of collided objects. Instead, BigKV minimizes its negative impact by making the collision rate extremely low (10^{-8}). This improves write throughput by removing the collision resolving step in the write path.
- **Proactive expired object removal**. BigKV facilitates fast removal of expired objects. It approximates TTLs of objects using 5 bits, which is small but is sufficient to identify expired objects accurately. For efficient space reclamation, when writing KV objects, BigKV allocates objects with similar TTLs in the same allocation group such that they are evicted and removed together. Keeping track of such large allocation groups requires little memory.
- **Reactive fault tolerance**. BigKV does not employ parity-based techniques, but it provides excellent fault tolerance through its reactive fault tolerance mechanism. BigKV isolates one or few SSDs within a separate shard, called a KV-Shard. Once a failed SSD is detected, BigKV disables the shard associated with the failed SSD, discarding the associated objects. Individual shards operate independently; thus, turning off the problematic shard (until the failed SSD is recovered) does not affect healthy shards. This approach reduces the cache capacity upon SSD failures. However, in normal cases, BigKV utilizes the full capacity of the AFA without any space overhead, thereby offering a higher hit ratio. No extra I/Os to write parity blocks are needed.

4.1 Overall Architecture of BigKV

The BigKV system embraces the above three techniques in a manner that provides scalable performance. BigKV employs a sharding-based design, as shown in Fig. 4. For each NIC,

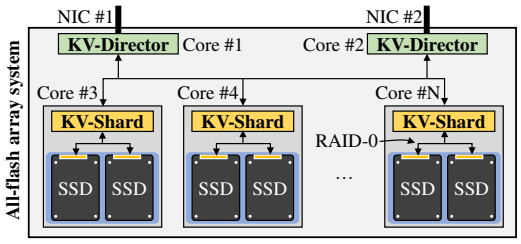


Fig. 4. Overall architecture of BigKV

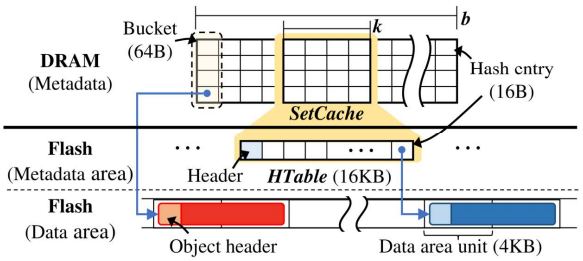


Fig. 5. Two-level metadata indexing

BigKV assigns a core, forming a KV-Director. One or few SSDs are grouped as a KV-Shard managed by a dedicated core. Each KV-Shard shares nothing with other KV-Shards. This design allows BigKV performance to scale with the number of SSDs and cores.

A KV-Director receives KV requests over the network and distributes them across KV-Shards. BigKV uses a strong yet efficient 128-bit hash function [24] to evenly distribute objects over shards. A KV-Director is also responsible for detecting and disabling a failed KV-Shard and for recovering objects through the reactive fault tolerance mechanism.

A KV-Shard maintains a two-level hash table, handles KV requests forwarded from KV-Directors, and manages any underlying SSD(s). A KV-Shard abstracts an SSD(s) belonging to it as a log. If more than one SSD is assigned, it aggregates them using RAID-0 to maximize performance. All the changes made on objects and metadata are appended to the log. A KV-Shard regularly performs cleaning to reclaim the space occupied by obsolete and expired objects by taking into account the TTLs.

4.2 Object Indexing with Two-level Metadata

We start by describing in-memory and in-flash data structures to index KV objects and explain how BigKV manipulates them while serving `LOOKUP()` and `SET()` requests.

4.2.1 Organization of two-level indexing

Fig. 5 illustrates the organization of the two-level metadata. BigKV divides flash space into two areas, a data area and a metadata area. KV objects and their headers containing properties (*e.g.*, a full-length key and an exact TTL) are stored in the data area. BigKV uses a fixed-size unit—4KB by default—to allocate space for KV objects. This fixed-size allocation leads to internal fragmentation, but its negative impact is insignificant since 4KB is sufficiently small to accommodate large objects (see Fig. 1).

In the metadata area, BigKV keeps flash-resident metadata that indexes KV objects. It is split into h hash tables (*HTables*), each of which is managed by open addressing hashing with a linear probing scheme. An HTable size is 16KB, which is aligned with a typical NAND page size. Each hash table entry has 16B per-object metadata associated with a KV object it points to. The per-object metadata holds essential information: a 64-bit fingerprint of a KV object, a 38-bit pointer to the object on the data area, an 18-bit object size, and other flags (8-bit). Including a 16B table header, each HTable can hold 1,023 hash entries.

BigKV caches hash entries that hold the metadata of hot objects in DRAM. KV objects (data and headers) are never cached in DRAM. Keeping many small hash entries in DRAM, rather than a few large objects, is better to provide consistent object access. The DRAM-resident metadata is organized as a set-associative cache that is divided into b *buckets*. Each bucket holds four 16B hash entries such that it fits a 64B CPU cache-line. A group of k consecutive buckets forms a *SetCache* that is dedicated to an HTable. In other words, each HTable has its own SetCache with k buckets. Each SetCache has a pointer to keep track of the up-to-date HTable, which is used later for metadata cleaning (see §4.3.1).

BigKV initially creates a large number of HTables to index all KV objects even if they are (the smallest) 4KB objects. HTables account for 0.4% of the total AFA space, and thus, it does not lead to serious cache space waste. Given a device capacity for a KV-Shard, the number h of HTables is statically decided (*e.g.*, $h = 2^{18}$ if the device capacity is 1TB). Similarly, given a DRAM size for a KV-Shard, the number b of buckets in SetCache is statically obtained (*e.g.*, $b = 2^{21}$ if DRAM is 128MB). In the above example, $k = 2^3$, and 8 buckets (= 32 hash entries) are dedicated to caching the same HTable. Considering that each HTable has 1,023 hash entries, the top 3.128% of hot entries can be cached in DRAM. This property enables system designers to configure an optimistic DRAM-to-SSD ratio for their workloads.

4.2.2 Object update with collision-oblivious hashing

Fig. 6 illustrates how BigKV deals with a `SET()` request. Given an object to write, KV-Director computes a 128-bit hash value from the object key using the hash function [24]. The highest $\log_2(s)$ bits of the hash value is used as a *shard number*, where s is the number of shards in the system. The request is then forwarded to the designated KV-Shard. The KV-Shard writes the object header and data to the data area first. Then, the KV-Shard uses the next $\log_2(b)$ bits of the hash value as a *bucket number* to select a target bucket where the new entry is added. The lowest 64 bits of the hash value are used as the object's *fingerprint*.

If the hash entry with the same fingerprint (FP) exists in the bucket, BigKV updates the existing entry such that it points to the up-to-date object in the data area. There is a

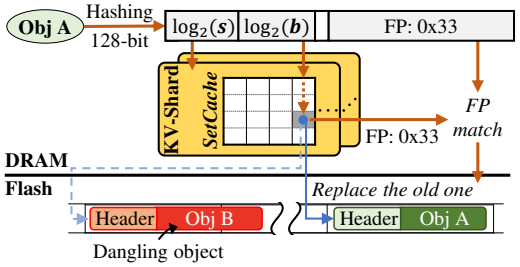


Fig. 6. Handling of SET() with collision-oblivious hashing

possibility that a fingerprint collision occurs if the new and old entries point to objects with different full keys (Obj A and Obj B in Fig. 6). BigKV simply ignores a possible fingerprint collision, allowing the new entry to replace the old one.

If no matched entry exists, BigKV adds the new hash entry to a free slot in the bucket (if there is a room). The old hash entry for the object may exist in the corresponding HTable. The old entry is eventually replaced with the new one when the bucket is evicted to the table. Finally, BigKV sets a 1-bit dirty bit in the entry and notifies the client of completion.

By ignoring hash collisions, BigKV involves no metadata I/Os while serving SET(). This, however, inevitably creates *dangling objects*. If a hash collision happens, a former (previously written) hash entry is overwritten by a new one, and thus its KV object in the data area becomes a dangling object that is not reachable (*e.g.,* Obj B in Fig. 6). Dangling objects not only occupy the cache space uselessly, but cause cache misses. Assuming that a dangling object is hot, future LOOKUP()s destined for it are missed all the time.

The negative impacts of dangling objects are not significant. Unlike other KV designs that rely on a tiny fingerprint (*e.g.,* 8-bit in μDepot), the two-level hash organization with a large 64-bit fingerprint exhibits an extremely low collision rate. In BigKV, a fingerprint collision occurs only when different objects are assigned to (*i*) the same shard and (*ii*) the same SetCache and (*iii*) they have the same fingerprint. According to our analysis, the collision rate is around 10^{-8}; only 5 cache misses out of 414M requests happen by collisions.

Some might think that even though dangling objects are rarely created, they may accumulate in the SSDs over time. This is not the case as dangling objects are regularly removed by the cleaning process (see §4.3). For performance, BigKV caches up-to-date hash entries in SetCaches, instead of immediately flushing them to HTables. Therefore, the old entries in the HTables still point to dangling objects. If system failures happen, BigKV may lose the up-to-date hash entries in DRAM that point to the latest objects. BigKV addresses this problem by taking a log-structured design that ensures metadata persistence from the failures (see §4.4).

4.2.3 Eviction of SetCache to HTable

If no free slots are left in a bucket, BigKV should evict a cold entry. To identify cold hash entries, BigKV maintains a 2-bit LRU field for each entry that is managed by a modified CLOCK algorithm. When a hash entry is referenced, BigKV sets its LRU field to 0 while increasing the others in the same bucket by 1. BigKV selects one that has the largest LRU among the four as a victim and evicts it.

If the victim was modified before (*e.g.,* the dirty field is 1), it must be written back to its HTable. BigKV evicts all dirty entries of k buckets in the SetCache to its HTable at a time. It is not only suitable for flash whose I/O unit is large (16KB) but amortizes future eviction I/Os to the same HTable. When evicting dirty entries, BigKV reads in the HTable to DRAM and inserts new entries into the table through linear probing. BigKV finally writes the up-to-date HTable back to flash.

Inserting a new entry to an HTable may fail if the table utilization is too high. Typical hashing performs resizing or rehashing to expand the table size or to move some entries elsewhere. This is, however, unnecessary in BigKV because BigKV initially creates sufficient HTables to index the smallest objects. If empty slots in HTables are nearly exhausted, it means that BigKV is almost full with objects and cannot accommodate new ones. Instead of resizing or rehashing the table, BigKV creates room in an HTable by evicting cold entries. Cold entries of an HTable are detected in a similar manner as in a SetCache. BigKV does not delete KV objects of evicted hash entries right away, so they become dangling objects in the data area. These objects are cleaned later.

4.2.4 Bounded object lookup

Given a KV object to retrieve, BigKV computes the bucket number for the object as explained in §4.2.2. If there exists a hash entry with the same fingerprint in the bucket, BigKV reads the object header and data pointed to by the entry from flash. Except for reading data, no extra I/Os occur.

If no matched entry is found, BigKV has to fetch a corresponding HTable from flash to DRAM, which involves one flash read. BigKV then sees if the desired entry is in the table using linear probing. If the matched one is found, BigKV retrieves the object data and its header from flash and inserts the entry into the bucket. Otherwise, BigKV immediately returns a cache miss. It is unnecessary to keep searching neighboring HTables. BigKV does not have to perform resizing or rehashing. Hash entries assigned to one HTable never move to adjacent HTables. This property limits the probe distance within a table and, consequently, guarantees that LOOKUP() is served by at most one extra flash I/O.

The matched hash entry may point to a wrong KV object owing to a fingerprint collision. If a full key from an object header is not identical to that from a LOOKUP() request, BigKV notifies a client of a cache miss.

4.3 TTL-aware Space Management

In this subsection, we present how BigKV manages the AFA space, allocating and cleaning objects. BigKV splits the metadata and data areas into 2MB segments, each of which is a unit of device-space allocation and cleaning. We take a

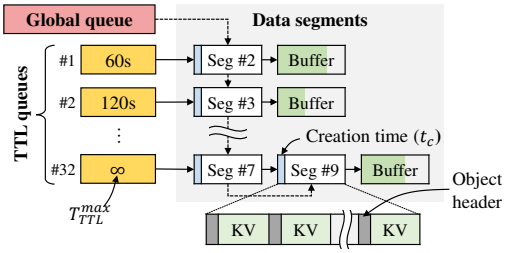


Fig. 7. TTL-aware segment allocation

log-structure approach to write metadata and object data to SSDs. BigKV keeps pending HTables and KV objects in separate 2MB write buffers. Once a buffer is full, it flushes it to its designated area, forming a new 2MB segment in either the metadata (for HTables) or the data area (for KV objects and headers). Old HTables and KV objects are left unchanged in other segments. If no free space is available, BigKV triggers cleaning to reclaim the space taken by old HTables and objects. To minimize cleaning costs and proactively remove expired KV objects, BigKV employs a TTL-aware allocation and cleaning policy.

4.3.1 TTL-aware segment allocation and cleaning

We first explain how BigKV manages metadata segments and then data segments because the allocation and cleaning of metadata segments are simpler.

Metadata segment. BigKV uses a FIFO queue to keep track of metadata segments. After writing pending HTables to a new metadata segment, BigKV pushes its segment number to the queue. When space is needed, BigKV pops the oldest segment from the queue as a victim for cleaning. This is reasonable as old HTables (written a long time ago) are likely to have more invalid entries than recent ones.

The cleaning process is simple. BigKV reads in all HTables from the victim. A 2MB metadata segment contains 64 HTables. For each HTable, BigKV checks whether the SetCache points to the table in the victim segment. If it does, that table contains up-to-date entries, so it must be moved elsewhere. Otherwise, the HTable is outdated, so it can be discarded.

Data segment. There are studies to place data with similar lifetimes together in SSDs to minimize cleaning costs [8, 23, 57]. Our approach is similar to these but takes TTLs into consideration. TTLs of objects let us know when the objects become obsolete. BigKV places KV objects with similar TTLs in the same segment such that they expire at a similar time and thus removed together. BigKV categorizes data segments into 32 types, each of which covers a different range of TTLs (*e.g.,* [0, 60s), [60s, 120s), ..., [5,184,000s, ∞) in Fig. 7). BigKV maintains a separate 2MB write buffer for each segment type. Each buffers pending KV objects whose TTLs fit in the TTL range of a designated segment type. Once a buffer becomes full, BigKV flushes out pending objects, creating a new data segment for that type.

To manage data segments, BigKV uses two types of FIFO queues. The first is a global FIFO queue. A newly created data segment is pushed to the global queue, regardless of its TTL type. The second is 32 TTL queues, each of which keeps a list of data segments with the same TTL type. Each TTL queue header has the maximum TTL T_{TTL}^{max} it can cover. In Fig. 7, the TTL queue #2 contains data segments that store objects whose TTLs are longer than 60 seconds but do not exceed 120 seconds. The TTL queue header records the creation time t_c of each segment (*i.e.,* the time at which the segment is written). Since KV objects in the segment are all expired before T_{TTL}^{max}, its expiration time can be estimated as $t_c + T_{TTL}^{max}$.

For data segment cleaning, BigKV scans the first (oldest) segments in the TTL queues and sees if there exists an expired segment whose estimated expiration time is smaller than the current time. If so, that segment is selected as a victim. Since all objects in the victim are expired, they are discarded and the victim is put into a free data segment list. No extra I/Os are necessary for cleaning.

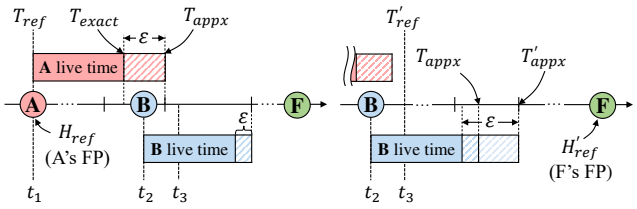
If no expired segments exist, BigKV evicts cold objects by picking up the oldest segment from the global queue. To prevent a hot object from being evicted, BigKV reads the full key in the object's header and sees if any SetCache caches its metadata. If it does, the object is hot, and thus, it should not be removed. It is simply rewritten to BigKV. Otherwise, the object is discarded. After moving all hot objects, the victim segment is discarded and is put into a free list.

BigKV finishes the cleaning process without removing hash entries of evicted objects from HTables. To remove them, BigKV has to undergo many HTable updates. A 2MB segment contains up to 512 4KB objects, and their hash entries are evenly distributed over HTables. In the worst case, 512 16KB HTables need to be updated, which involves up to 8MB (= 512×16KB) data reads and writes. This is too costly considering that 2MB of free space is reclaimed after segment cleaning. For performance, BigKV skips updating HTables, but it leaves *zombie hash entries* in HTables that point to the locations of evicted objects in data segments.

4.3.2 Zombie cleaning

Zombie entries are inevitably created as a result of data segment cleaning. They accumulate over time and eventually take up space in HTables. Once an HTable becomes full, BigKV removes unpopular entries from HTables using LRU (see §4.2.3). This approach, however, cannot efficiently remove zombie entries for TTL-expired objects, in particular, when their TTLs are relatively short. Short-lived objects expire quickly after being written. BigKV promptly removes expired KV objects from data segments, as explained in §4.3.1. However, their hash entries in HTables are still considered hot since they have been referenced recently. Therefore, other hash entries pointing to valid KV objects are likely to be chosen and removed by the LRU policy.

To quickly remove zombie hash entries, BigKV leverages their TTLs. To make per-object metadata compact, BigKV reserves only a 5-bit TTL field to approximate the expiration



(a) Initializing <T_{ref}, H_{ref}> **(b)** Refreshing <T_{ref}, H_{ref}>

Fig. 8. TTL approximation heuristic

time of a hash entry (not a KV object). A 5-bit TTL represents one of 32 TTL types in §4.3.1. When cleaning up metadata segments (§4.3.1) or evicting dirty entries from a SetCache to an HTable (§4.2.3), BigKV reads an HTable from flash. Combined with these processes, BigKV checks TTL fields and removes expired entries. Doing so can clean up zombies without any extra I/Os.

TTL approximation. We use a simple yet accurate heuristic to approximate a TTL using 5 bits, which is illustrated in Fig. 8. For each HTable, we use two parameters <T_{ref}, H_{ref}>. T_{ref} is a 64-bit reference time and is used to estimate expiration times of hash entries. H_{ref} is a 64-bit fingerprint of a reference hash entry and is used to decide when to refresh T_{ref}. They are stored in the HTable header.

T_{ref} and H_{ref} are not initialized when an HTable is empty. When the first hash entry is added to the table at t_1 (Ⓐ in Fig. 8(a)), T_{ref} is set to t_1. The first hash entry is chosen as a reference entry, and H_{ref} is set to have its fingerprint. Then, using the exact TTL T_{exact} of the first hash entry, BigKV estimates its 5-bit approximate TTL, T_{appx}. Among the 32 TTL candidates in §4.3.1, BigKV chooses the shortest one that is longer than T_{exact}. Here, $T_{appx} - T_{exact}$ is the approximation error ε. The exact TTL is stored in its object header, while the approximate TTL is kept in the per-object metadata of the HTable. For the next hash entry written at t_2 (Ⓑ in Fig. 8(a)), T_{appx} is chosen to be the shortest one that is longer than $(t_2 - T_{ref}) + T_{exact}$ among the candidates.

The identification of expired hash entries is straightforward. Supposing that BigKV attempts to remove expired entries at t_3, hash entries whose $T_{appx} < t_3$ are expired ones. In the example of Fig. 8(a), the hash entry Ⓐ is expired and can be removed from the HTable.

If the reference entry that has the same fingerprint as H_{ref} is expired or evicted, we should decide the new <T'_{ref}, H'_{ref}> pair. In Fig. 8(b), the reference entry Ⓐ is deleted at t_3. The new T'_{ref} is set to t_3. The new reference hash entry is chosen to be the one that has the longest T_{appx}. In the example, Ⓕ is the new reference entry, and thus, its fingerprint becomes H'_{ref}. Since the reference time is updated, we have to update the approximate TTLs of the hash entries accordingly. We do not keep exact TTLs in the HTable. Thus, we estimate new approximate TTLs T'_{appx} using old ones T_{appx}. T'_{appx} is the shortest one that is longer than $T_{appx} - T'_{ref}$. Since $T_{appx} - T'_{ref} > 0$, the approximation error ε becomes larger than before. However, ε no longer increases as all the entries in the HTable expire before the new reference entry Ⓕ.

4.4 Reactive Fault Tolerance Mechanism

In the event of SSD failures, BigKV disables problematic KV-Shards, abandoning associated objects. This policy exploits the fact that data loss is tolerable in a caching system. To make BigKV truly fault-tolerable with this policy, however, two technical issues must be addressed.

The first issue is how to redirect incoming KV requests to healthy shards upon failure. A shard number is statically decided by the hash function with an object key. An object destined for a failed shard can no longer be cached, and clients suffer from repeated cache misses until a failed SSD is recovered. BigKV addresses this by using additional hash functions. For an object mapped to a failed shard, BigKV computes its shard number using a different hash function and sends it to a healthy shard. Reading a redirected KV object is performed similarly through the new hash function.

The second issue is the efficient migration of redirected objects. After a failed KV-Shard is recovered, BigKV may migrate KV objects redirected to other healthy shards to the recovered one. There may exist two extreme solutions, full migration and no migration. The full-migration scans all the healthy shards and migrates redirected objects to the repaired shard. After all objects are moved, BigKV activates the recovered shard such that it starts to service client requests. Compared with RAID, full-migration requires smaller I/Os since it does not need to scan all SSDs in the system and assemble failed blocks. However, it still causes non-trivial I/Os, taking a long time until activation. The no-migration discards all KV objects temporarily stored in healthy shards and immediately activates the repaired shard. This requires no I/Os for migration but suffers from high miss ratios until the recovered shard is sufficiently warmed up .

BigKV employs a complementary approach. Once a failed KV-Shard is recovered, BigKV initially keeps it deactivated, and incoming KV requests are serviced by healthy shards. If objects read by clients belong to the recovered one, BigKV forwards them to the original shard. Those objects are then directly served from the recovered shard. After the original shard contains a sufficient number of KV objects, servicing almost all requests to hot objects, BigKV activates the recovered shard, discarding the remaining KV objects in other shards. This approach not only minimizes migration costs but also quickly warms up the recovered shard.

Metadata persistence. BigKV keeps dirty hash entries in SetCaches until they are flushed out to HTables. If a sudden system or power failure occurs, dirty entries are inevitably lost. Thanks to its log-structured design, BigKV can achieve excellent metadata persistence. BigKV appends KV objects to data segments, together with their headers containing full properties. All the data segments are linked to another,

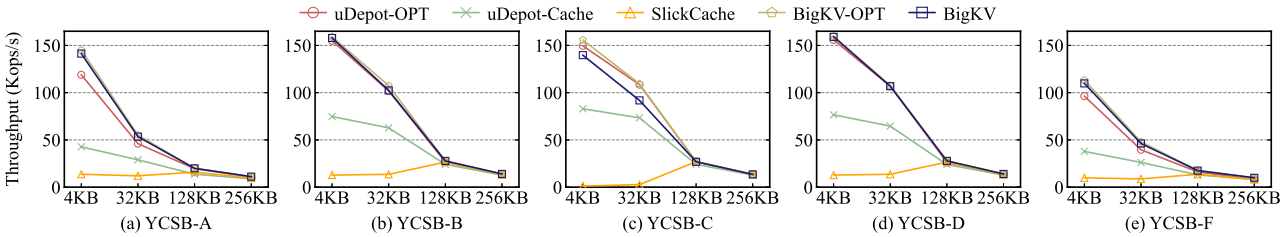


Fig. 9. Throughput of YCSB workloads with high locality

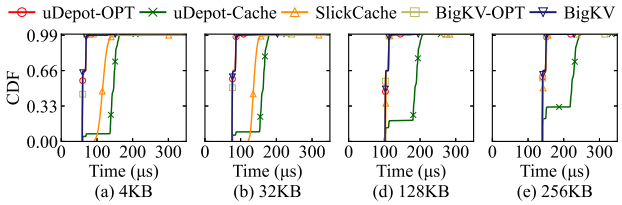


Fig. 10. Latency of YCSB-B with high locality

forming a huge log. Even when system failure occurs, BigKV can recover up-to-date HTables by scanning object headers recorded in the log. To avoid making the log too long, BigKV may regularly flush out dirty entries to HTables akin to checkpointing in other log-structured systems.

5 Experiments

We implement a proof-of-concept prototype of BigKV in a real AFA machine and evaluate it using various workloads. We compare BigKV with four state-of-the-art KV caching systems: uDepot, SlickCache, Fatcache, and Memcached. All the experiments are conducted locally to analyze cache performance and hit ratios without network interference.

5.1 Experimental Setup

Our AFA machine has two Xeon Gold 6152 CPUs (44 cores running at 2.1 GHz) and 64GB DRAM. It can accommodate 32× U.2 PCIe SSDs. We use 8× 3.84TB NVMe SSD (Samsung's PM9A3 [17]), each of which provides 3.4GB/s and 3.1GB/s sequential read and write throughputs, respectively. For fast evaluation, we intentionally scale down our AFA system so that it uses 1TB SSD space and 128MB DRAM. This DRAM-to-SSD ratio is equivalent to that of an AFA with 1PB SSD space and 128GB DRAM (which is similar to NetApp's AFA products [44]). Three key parameters, h, b, and k, of BigKV are set to 2^{18}, 2^{21}, and 2^3, respectively (see §4.2.1).

We compare BigKV with uDepot [33] and SlickCache [55], which represent open-addressing and chaining KV caching systems, respectively. Unlike BigKV and SlickCache that operate under small DRAM by caching hot hash entries, uDepot is designed to keep the entire hash table in DRAM. Therefore, we implement a modified version of uDepot, `uDepot-Cache`, that runs with limited DRAM by caching popular table pieces as explained in §3.1. We also evaluate the original uDepot, `uDepot-OPT`, which assumes to have sufficient DRAM to hold the entire hash table. To show how BigKV behaves with sufficient memory, we compare `BigKV-OPT` keeping all hash tables in DRAM. `uDepot-OPT` and `BigKV-OPT` are unrealistic but are helpful for performance analysis.

We also compare BigKV with two real-world KV caching systems, Memcached and Fatcache (see §5.2.4). Unless otherwise states, all the systems use a single SSD. This is because slow RAID makes it difficult to interpret the impact of individual techniques objectively. The impact of RAID on performance and hit ratios is evaluated separately in §5.2.3.

As benchmarks, we use five workloads from YCSB [10], except for YCSB-E, which is used to evaluate range-query performance. The hotspot distribution is used by default, but we configure parameters to generate two types of workloads with different localities: one with high locality (99:1) and another with moderate locality (80:20). The original YCSB does not have any TTL information. To evaluate the impact of the TTL-aware space management, we use Twitter traces that record TTLs for every object. Twitter's traces [53, 62] were collected from in-memory KV caches where objects were small. To reflect realistic object sizes, we adjust the size distribution of objects based on the statistic of real-world caching systems as shown in Fig. 1.

5.2 Experimental Results

5.2.1 Performance analysis with YCSB

We compare the performance of BigKV with uDepot [33] and SlickCache [55] using YCSB. We focus on measuring throughput (KV operations per second) and lookup latency. To understand the impact of object sizes, we conduct experiments while varying object sizes 4KB, 32KB, 128KB, and 256KB. We fill up the AFA space with objects during a load phase and run 10M KV operations over the created KV pool.

Fig. 9 shows the throughputs of the KV caching systems under the highly-localized YCSB workload (99:1). When average object sizes are 4KB and 32KB, `BigKV` exhibits throughputs comparable to `uDepot-OPT` and outperforms `uDepot-Cache` and `SlickCache` by 2.0× and 8.8×, on average, respectively. By taking advantage of highly-localized object access patterns, `BigKV` accomplishes a 99.3% metadata hit ratio. `BigKV` caches the top of 3.128% of the hot entries in the SetCache, which is sufficiently large to hold the 1% hot entries of the highly-localized workload. Thanks to the high hit ratio, `BigKV` achieves similar throughput to `BigKV-OPT`.

`uDepot-Cache` fails to exploit the high locality of objects, showing a low metadata hit ratio, 8.9%. This almost doubles

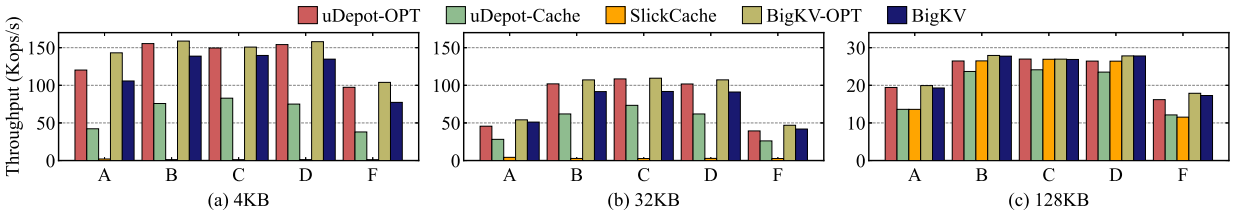


Fig. 11. Throughput of YCSB workloads with moderate locality

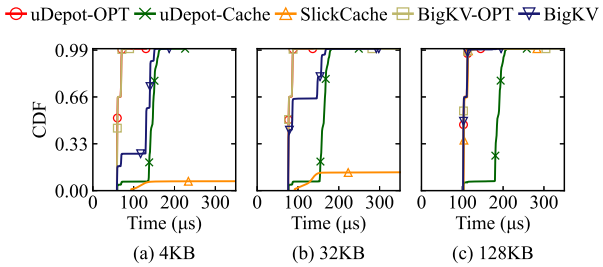


Fig. 12. Latency of YCSB-B with moderate locality

I/O traffic to the SSD as `uDepot-Cache` has to look up in-flash hash tables before serving user requests.

`SlickCache` also caches hot entries in DRAM, achieving a high hit ratio. Nonetheless, `SlickCache` suffers from severe throughput drops across all workloads. This is due to the high penalty when a cache miss occurs. As explained in §2, on a cache miss, `SlickCache` has to traverse in-flash hash tables, which causes many I/Os. `SlickCache` also has to perform merge operations regularly. For performance, `SlickCache` flushes out many dirty entries to the log in the SSD (a warm tier) at once and then merges them with in-flash hash tables (a cold tier) later. Merging warm and cold tiers, however, incurs many I/Os, making `SlickCache` slow.

Another noticeable observation is that, for YCSB-A (Fig. 9(a)) and F (Fig. 9(e)), where updates are dominant (50% for A and F), `BigKV` even outperforms `uDepot-OPT`. While `BigKV` ignores fingerprint collisions, `uDepot` has to read object headers and verify whether collisions occur for every `SET()`. This results in moderate performance penalties.

When the average object size is 128KB or larger, all the systems exhibit similar throughputs. There are two reasons. First, as the object size becomes larger, the number of objects to index decreases. Given the same DRAM, a larger proportion of the hash table can be cached, which leads to a higher metadata hit ratio. Even `uDepot-Cache` brings many portions of the hash table to DRAM, thus achieving better throughput. The merge cost of `SlickCache` is also lowered since most dirty entries are cached in DRAM and not evicted to the SSD. This makes `SlickCache` faster than when the object size is 32KB or under. Second, the negative impact of a metadata miss decreases; instead, reading large objects (128KB and 256KB) from the SSD becomes a dominant part.

Fig. 10 illustrates the CDF of lookup latency of YCSB-B where `LOOKUP()` is dominant (95%), while others have a similar latency trend. `BigKV` offers almost the same lookup latency as `uDepot-OPT`, thanks to its high metadata hit ratio. `SlickCache` also achieves a high hit ratio, but it suffers from long latency. This is because internal I/Os (required to scan in-flash tables and merge warm and cold tiers) frequently delay client requests. When the object size increases to 128KB and 256KB, such interference disappears. `uDepot-Cache` shows longer latency than `BigKV` and `SlickCache` because of its high metadata miss ratio.

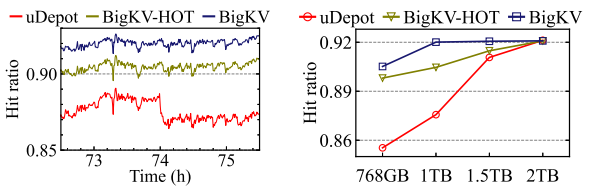
Fig. 11 and Fig. 12 show the throughput and latency of YCSB with moderate locality. The results with 256KB objects are omitted because all the systems exhibit high performance as that in Fig. 9. `BigKV` exhibits fairly good throughput across all workloads. As shown in Fig. 12, even when a DRAM cache miss occurs, `BigKV` can bound a miss penalty by only one I/O. `uDepot-Cache` shows similar throughputs as those under the highly-localized workload. This is owing to its architectural limit that cannot exploit the locality of objects. `SlickCache` shows the worst performance in most cases because of its high miss penalty. Since `BigKV-OPT` and `uDepot-OPT` keep all the metadata in DRAM, the performance gap between them shows the impact of the collision-oblivious hashing. As `BigKV-OPT` does not require lookups before updating objects by ignoring hash collisions, it outperforms `uDepot-OPT` on update-dominant workloads (YCSB-A and F).

Finally, Table 2 compares the tail latency of the systems. Both `uDepot-Cache` and `SlickCache` suffer from long tails. `BigKV` exhibits 4.7× and 12.2× shorter tails than `uDepot-Cache` and `SlickCache` at the 99.99^{th} percentile.

Some might think that BigKV is only effective in a limited object size range (*e.g.*, 4KB–128KB). This is not the case. According to our observation, even for objects smaller than 4KB, BigKV works better than other systems by better utilizing limited DRAM space. As objects get smaller, BigKV suffers from internal fragmentation because of its 4KB allocation unit. This problem, however, commonly happens in flash-based caching systems and can be mitigated by making the allocation unit smaller. Meanwhile, if object sizes are very large (over 256KB) and thus, the entire metadata can

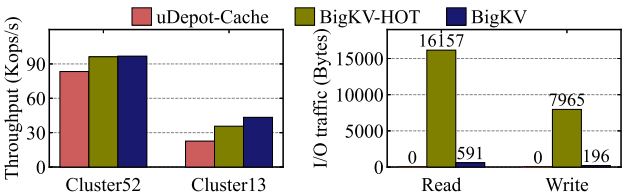
Table 2. Tail latency of YCSB-B with 32KB objects

Latency unit: us		uDepot -OPT	uDepot -Cache	SlickCache	BigKV -OPT	BigKV
99:1	99%	88	183	201	88	96
	99.9%	118	247	1,655	95	167
	99.99%	208	1,115	1,993	208	204
80:20	99%	88	183	2,639	88	168
	99.9%	123	231	4,019	99	189
	99.99%	218	1,220	4,602	215	301



(a) Hit ratio trend over time **(b)** Hit ratios over various cache sizes

Fig. 13. Impact of TTL-aware cleaning on hit ratios



(a) Throughput (b) I/O traffic for segment cleaning

Fig. 14. Throughput and cleaning I/O of Twitter benchmark

be loaded into DRAM, BigKV outperforms other systems by ignoring hash collisions as depicted in Fig. 11.

5.2.2 Impact of TTL-aware space cleaning

We analyze the impact of TTL-aware space management on the hit ratios. We choose cluster52 from the Twitter traces, which is composed of 95% object lookups and 5% object updates. The cluster52 trace is a read-heavy workload and is thus suitable for examining hit ratios of AFA caches when the working set is huge. The distribution of TTLs ranges from 12 hours to two weeks. The trace was gathered for one week and it recorded 130 billion KV requests. We modify the socket front-ends of the caches to replay the trace files.

We compare hit ratios of three techniques, `uDepot`, `BigKV-HOT`, and `BigKV`. `uDepot` selects victim objects using LRU and discards all of them from the AFA cache. `BigKV-HOT` chooses a cold segment from the global queue and moves only hot objects to a free segment (see §4.3.1). It does not explicitly remove TTL-expired objects. In addition to hot object migration, `BigKV` proactively evicts expired objects through TTL approximation.

Fig. 13(a) shows the hit ratio trend of `uDepot`, `BigKV-HOT`, and `BigKV` over time when the AFA capacity is 1TB. Overall, `BigKV` provides 1.6% and 4.5% higher hit ratios than `BigKV-HOT` and `uDepot`, respectively. For them to achieve the same hit ratio as `BigKV`, much larger cache space is needed. We plot the cache hit ratios while increasing the AFA capacity from 768GB to 2TB in Fig. 13(b). When the AFA capacity is 1TB, `BigKV` accomplishes the maximum hit ratio, 92.1%, that we can achieve on the trace. Both `BigKV-HOT` and `uDepot` offer this hit ratio when we double the AFA capacity to 2TB. In production environments, caching systems should have sufficient cache space to satisfy the target miss ratio [63]. The above results show that BigKV can achieve a specified miss ratio with a smaller AFA capacity.

We also measure the throughputs and cleaning costs of the three policies in the 1TB device capacity by using two traces, cluster52 and cluster13, from Twitter. The cluster52 trace is a read-intensive workload as explained above. The cluster13 trace is a write-intensive workload that comprises 50% lookups and 50% updates. As shown in Fig. 14(a), `BigKV` exhibits 16% and 1.91×% higher throughputs than `uDepot-Cache` on the cluster52 and cluster13 traces, respectively. This performance improvement stems from the high metadata hit ratio and the collision-oblivious hashing. Additionally, compared with `BigKV-HOT`, the TTL-aware space cleaning of `BigKV` improves throughput by 21% in the cluster13 workload by lowering cleaning costs. Fig. 14(b) represents the average amount of data moved while cleaning a 2MB data segment. `uDepot` requires zero extra I/Os as it discards all the objects without any hot object migration. `BigKV-HOT` must read object headers to identify hot objects and copy them to a free segment. To create a free segment, `BigKV-HOT` incurs 24KB of extra I/O traffic. `BigKV` selects expired segments first and then chooses cold ones. No extra I/Os are needed for expired segments; thus, `BigKV` issues much less I/O traffic when cleaning segments.

5.2.3 Impact of reactive fault tolerance

Next, we explain how reliably BigKV operates under SSD failures. Our system has eight SSDs. We compare two systems: `BigKV` and `BigKV-RAID`. `BigKV` runs 8 KV-Shards, each of which is coupled with one SSD, and manages them using the reactive fault tolerance mechanism. `BigKV-RAID` runs 8 KV-Shards over a volume created by grouping eight SSDs with RAID-6. `BigKV-RAID` uses 25% of the capacity for parity blocks. We run YCSB-A with an average object size of 32KB.

Fig. 15 illustrates the hit ratios and throughputs of `BigKV` and `BigKV-RAID` (*i*) when an SSD fails, (*ii*) during its recovery phase and (*iii*) after the failed SSD is replaced with a new one. The throughput is normalized to when no SSD fault occurs, denoted by `BigKV(NoFault)`. Initially, `BigKV` exhibits a higher hit ratio (83%) than `BigKV-RAID` (79%) because it has a larger cache capacity. The throughput of `BigKV` is 3× higher than that of `BigKV-RAID`.

To emulate a device failure, we intentionally disable one SSD at 7,000 seconds. Since `BigKV` does not protect any SSDs, it results in the loss of objects. `BigKV` experiences a sharp hit ratio drop from 83% to 73%, and the overall throughput of `BigKV` degrades by 5–6%. The hit ratio and throughput gradually increase because KV-Directors redirect incoming KV objects to other KV-Shards. At around 7,400 seconds, we replace the failed SSD, and `BigKV` starts a recovery process. No throughput drop occurs during the recovery because `BigKV` sends hot objects to the recovered shard in the background while other KV-Shards keep servicing requests. In our setting, after 12.5% of the SetCache is filled with hot objects, `BigKV` activates the recovered KV-Shard to start caching services. There is a small hit ratio drop at around 8,400 seconds.

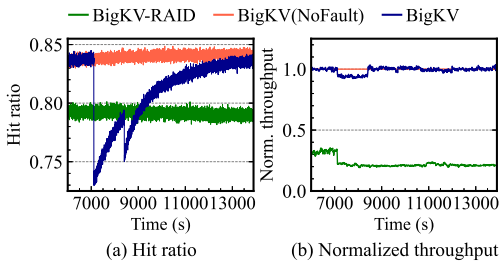


Fig. 15. Comparison of reactive fault tolerance and RAID

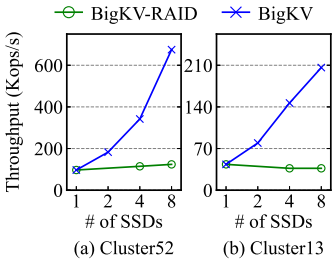


Fig. 16. Performance scalability on Twitter traces

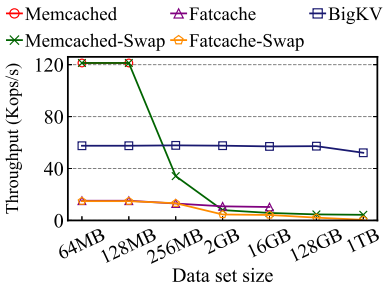


Fig. 17. Comparison with Memcached and Fatcache

This is because other KV-Shards opt out, and thus requests to warm objects stored on them result in cache misses.

In contrast to BigKV, the hit ratio of BigKV-RAID is not affected by the SSD failure because there is no loss in capacity. However, after the failed SSD is detected, BigKV-RAID enters a degraded mode [59] and experiences a 36% performance drop because of online reconstruction of damaged data with parities. Even after repairing the failed SSD, BigKV-RAID performs slowly until the RAID rebuilding process is completed, which takes longer than 300 minutes.

Fig. 16 compares the throughputs while increasing the number of SSDs. We use the Twitter traces used in §5.2.2 to measure the performance scalability. BigKV provides 5.4× and 5.8× higher throughputs than BigKV-RAID with 8 SSDs. The performances of BigKV scales linearly with the number of SSDs. The KV-Director of BigKV distributes incoming requests across the KV-Shards evenly, while each KV-Shard handles them simultaneously. BigKV-RAID fails to deliver scalable performance due to extra I/Os and calculations for parity blocks. Note that BigKV-RAID with 2 SSDs does not work because RAID-6 requires at least 4 SSDs.

5.2.4 Comparison with real-world caching systems

Finally, we compare the performance of BigKV with two real-world caching systems, Memcached [47] and Fatcache [54]. Our server uses 128MB DRAM and a 1TB SSD the same as the default setup. We run YCSB-A over the three systems while increasing its data set size from 64MB to 1TB. The average object size is set to 32KB. The original Memcached is designed to keep the entire data set in DRAM, not persistent storage like SSDs. Thus, it is unable to accommodate more than 128MB data sets. For comparison, we configure Memcached to use the swap area if no DRAM space is available for caching objects (Memcached-Swap). Fatcache improves its caching capacity by using SSDs. However, it still assumes that the system has sufficient DRAM to keep the entire metadata in the main memory. Similar to the case in Memcached, we configure Fatcache to use the swap area if there is no DRAM space for keeping the metadata (Fatcache-Swap).

Fig. 17 shows our experimental results. Memcached exhibits the best performance. However, after the data set size grows beyond 256MB, it cannot store the entire data due to insufficient DRAM space. Memcached-Swap still operates but its performance drops severely due to frequent I/Os to the swap area. Fatcache stops working after the data set size increases to around 85GB. Because of its huge per-object metadata size, 48B, Fatcache requires more than 128MB of DRAM for its metadata when the data set size is about 85GB. Similar to Memcached-Swap, Fatcache-Swap operates but suffers from significant performance penalties. It is worth noting that both Fatcache and Fatcache-Swap do not show good performance even when the entire metadata is kept in DRAM. As discussed in other literature [52, 61], Fatcache handles flush and cleaning synchronously, which often blocks client requests. BigKV performs worse than Memcached for small data sets, but it outperforms all the systems when the data set is huge. Particularly, BigKV exhibits consistent throughput, regardless of the input data set size.

6 Conclusion

In this paper, we presented a new AFA-scale KV caching system, called BigKV, for large objects. We addressed three technical challenges that arose when transforming an AFA into a big KV cache: (*i*) high indexing overhead caused by huge metadata, (*ii*) space waste by expired objects, and (*iii*) service interruption by frequent SSD failures. To tackle these problems and improve the cache hit ratio, we proposed three techniques: collision-oblivious hash table, approximate-TTL management, and reactive fault tolerance mechanism. Our experimental results using real AFA machine showed that BigKV provided 3.1× higher throughput and 57% lower latency, on average, for the YCSB workloads. For future work, we plan to enhance BigKV such that it is efficiently integrated with emerging network architectures (*e.g.,* NVMe-oF) and storage devices (*e.g.,* ZNS SSD).

Acknowledgments

We thank our shepherd, Sam H. Noh, and the anonymous reviewers for all their helpful comments. This work was supported by SNU-SK Hynix Solution Research Center (S3RC), the National Research Foundation of Korea (NRF-2018R1A5A1060031 and NRF-2020R1A4A4079859), the National Science Foundation of the USA (CNS-2008453), and the MOTIE (Ministry of Trade, Industry & Energy) (1415181081) and KSRC (Korea Semiconductor Research Consortium) (20019402). (Corresponding author: Sungjin Lee)

References

[1] Inc. Adata. 2022. Adata XPG Gammix S70. https://www.xpg.com/us/xpg/685

[2] Jens Axboe. 2022. FIO: Flexible I/O Tester Synthetic Benchmark. https://github.com/axboe/fio

[3] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. 2010. Differential RAID: Rethinking RAID for SSD Reliability. In *Proceedings of the European conference on Computer systems*. 15–26.

[4] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference*. 753–766.

[5] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. 2003. Introduction to Flash Memory. *Proc. IEEE* 91, 4 (2003), 489–502.

[6] Michaela Blott, Ling Liu, Kimon Karras, and Kees Vissers. 2015. Scaling out to a Single-node 80gbps Memcached Server with 40Terabytes of Memory. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*.

[7] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 209–223.

[8] Chandranil Chakraborttii and Heiner Litz. 2021. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proceedings of the ACM International Conference on Systems and Storage*. 1–12.

[9] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast,Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 17–32.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM symposium on Cloud computing*. 143–154.

[11] Nimbus Data. 2020. ExaDrive DC. https://nimbusdata.com/docs/ExaDrive-DC-Datasheet.pdf

[12] Inc. Dell EMC. 2018. Dell EMC PowerVault ME4 Series Storage Specification Sheet. https://www.delltechnologies.com/asset/en-us/products/storage/technical-support/h17384-powervault-me4-series-ss.pdf

[13] Inc. Dell EMC. 2019. DELL EMC SC ALL-FLASH STORAGE ARRAYS. https://i.dell.com/sites/csdocuments/Shared-Content_data-Sheets_Documents/en/SC-All-Flash-spec-sheet.pdf

[14] Inc. Dell EMC. 2021. Dell EMC PowerEdge R750 Specification Sheet. https://i.dell.com/sites/csdocuments/Product_Docs/en/poweredge-R750-spec-sheet.pdf

[15] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*. 65–78.

[16] Samsung Electronics. 2013. Samsung 840 EVO. https://www.samsung.com/us/system/consumer/product/mz/7t/e5/mz7te500lw/Sam_1303_SSD-840-EVO-Spec-Sheet_v9.pdf

[17] Samsung Electronics. 2020. Samsung PM9A3 NVMe PCIe SSD. https://www.samsung.com/semiconductor/global.semi.static/PM9A3_SSD_Whitepaper.pdf

[18] EU. 2016. General Data Protection Regulation (EU GDPR). https://gdpr-info.eu/

[19] Inc. Facebook. 2022. RocksDB: A Persistent Key-value Store for Fast Storage Environments. https://rocksdb.org

[20] Apache Software Foundation. 2022. hbase. https://github.com/apache/hbase

[21] Inc. Google. 2022. LevelDB. https://github.com/google/leveldb

[22] Jaehyun Han, Guangyu Zhu, Eunseo Lee, Sangmook Lee, and Yongseok Son. 2021. An Empirical Evaluation and Analysis of Performance of Multiple Optane SSDs. In *Proceedings of the International Conference on Information Networking*. 541–545.

[23] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. The unwritten contract of solid state drives. In *Proceedings of the twelfth European conference on computer systems*. 127–144.

[24] Google Inc. 2022. Google's City Hash functions. https://github.com/jeremybarnes/cityhash

[25] Inc. Intel. 2022. Intel P55 Express Chipset. https://ark.intel.com/content/www/us/en/ark/products/42690/intel-p55-express-chipset.html

[26] Inc. Intel. 2022. Intel Xeon Gold 5315Y Processor. https://ark.intel.com/content/www/us/en/ark/products/215286/intel-xeon-gold-5315y-processor-12m-cache-3-20-ghz.html

[27] Nikolaus Jeremic, Gero Mühl, Anselm Busse, and Jan Richling. 2011. The pitfalls of deploying solid-state drive RAIDs. In *Proceedings of the ACM International Conference on Systems and Storage*. 1–13.

[28] Yichen Jia, Zili Shao, and Feng Chen. 2018. SlimCache: Exploiting Data Compression Opportunities in Flash-based Key-value Caching. In *Proceedings of the IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 209–222.

[29] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. 2021. FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 355–370.

[30] Gaurav Kaul, Zeeshan Ali Shah, and Mohamed Abouelhoda. 2017. A High Performance Storage Appliance for Genomic Data. In *Proceedings of the International Conference on Bioinformatics and Biomedical Engineering*. 480–488.

[31] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H Noh. 2019. Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays. In *Proceedings of the USENIX Annual Technical Conference*. 799–812.

[32] Sungjoon Koh, Jie Zhang, Miryeong Kwon, Jungyeon Yoon, David Donofrio, Nam Sung Kim, and Myoungsoo Jung. 2018. Exploring Fault-tolerant Erasure Codes for Scalable All-flash Array Clusters. *IEEE Transactions on Parallel and Distributed Systems* 30, 6 (2018), 1312–1330.

[33] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the Performance of Fast NVM Storage with Udepot. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 1–15.

[34] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[35] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*. 447–461.

[36] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*. 137–152.

[37] Jian Liu, Kefei Wang, and Feng Chen. 2021. TSCache: an Efficient Flash-based Caching Scheme for Time-series Data Workloads. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3253–3266.

[38] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. 2020. A Study of SSD Reliability in Large Scale Enterprise Storage Deployments. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 137–149.

[39] Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. 2014. NVMKV: A Scalable and Lightweight Flash Aware Key-value Store. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems*.

[40] Sarang Masti. 2021. How we built a general purpose key value store for Facebook with ZippyDB. https://engineering.fb.com/2021/08/06/core-data/zippydb

[41] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-tree database storage engine serving Facebook's Social Graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.

[42] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*. 243–262.

[43] Sangwhan Moon and AL Narasimha Reddy. 2016. Does RAID improve lifetime of SSD arrays? *ACM Transactions on Storage* 12, 3 (2016), 1–29.

[44] Inc. NetApp. 2016. NetApp All Flash FAS. https://www.netapp.com/media/19763-ds-3829.pdf

[45] Netflix. 2018. Evolution of Application Data Caching:From RAM to SSD. https://netflixtechblog.com/evolution-of-application-data-caching-from-ram-to-ssd-a33d6fa7a690

[46] Yoshio Nishi and Blanka Magyari-Kope. 2019. *Advances in Non-volatile Memory and Storage Technology*. Woodhead Publishing.

[47] 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 *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*. 385–398.

[48] Ki-Tae Park, Sangwan Nam, Daehan Kim, Pansuk Kwak, Doosub Lee, Yoon-He Choi, Myung-Hoon Choi, Dong-Hun Kwak, Doo-Hyun Kim, Min-Su Kim, et al. 2014. Three-dimensional 128 Gb MLC Vertical NAND Flash Memory with 24-WL Stacked Layers and 50 MB/s High-speed Programming. *IEEE Journal of Solid-State Circuits* 50, 1 (2014), 204–213.

[49] Yifan Qiao, Xubin Chen, Jingpeng Hao, Jiangpeng Li, Qi Wu, Jingqiang Wang, Yang Liu, and Tong Zhang. 2021. Improving Relational Database Upon the Arrival of Storage Hardware with Built-in Transparent Compression. In *Proceedings of the IEEE International Conference on Networking, Architecture and Storage*. 1–9.

[50] Inc. Seagate. 2022. Seagate FireCuda 530. https://www.seagate.com/products/gaming-drives/pc-gaming/firecuda-530-ssd/

[51] Debendra Das Sharma. 2020. PCI Express® 6.0 Specification at 64.0 GT/s with PAM-4 Signaling: a Low Latency, High Bandwidth, High Reliability and Cost-effective Interconnect. In *Proceedings of the IEEE Symposium on High-Performance Interconnects*. 1–8.

[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 *Proceedings of the USENIX Conference on File and Storage Technologies*. 391–405.

[53] Inc. Twitter. 2022. Anonymized Cache Request Traces from Twitter Production. https://github.com/twitter/cache-trace

[54] Inc. Twitter. 2022. Fatcache: Memcache on SSD. https://github.com/twitter/fatcache

[55] Kefei Wang and Feng Chen. 2018. Cascade mapping: Optimizing Memory Efficiency for Flash-based Key-value Caching. In *Proceedings of the ACM Symposium on Cloud Computing*. 464–476.

[56] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the European Conference on Computer Systems*. 1–14.

[57] Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. 2022. Separating Data via Block Invalidation Time Inference for Write Amplification Reduction in Log-Structured Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 429–444.

[58] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 307–320.

[59] Wikipedia. 2022. Degraded mode. https://en.wikipedia.org/wiki/Degraded_mode

[60] Wikipedia. 2022. *Solid-state drive*. https://en.wikipedia.org/wiki/Solid-state_drive#cite_note-2Xelp-48

[61] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, et al. 2016. Bluecache: A Scalable Distributed Flash-based Key-value Store. *Proceedings of the VLDB Endowment* 10, 4 (2016), 301–312.

[62] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 191–208.

[63] Juncheng Yang, Yao Yue, and Rashmi Vinayak. 2021. Segcache: a Memory-efficient and Scalable In-memory Key-value Cache for Small Objects. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*. 503–518.

A Artifact Appendix

A.1 Abstract

BigKV is a key-value cache specifically designed for caching large objects in an all-flash array. For artifact evaluation, we provide our source code and scripts for benchmarks. Please refer to the README file at https://github.com/dgist-datalab/bigkv.

A.2 Scope

The artifact includes all the necessary source code required to run BigKV as well as the benchmarks used in this study. As it takes several weeks to run all the benchmarks used in this study, we also provide an exemplary benchmark suite with tuned parameters.

A.3 Description & Requirements

We provide a Git repository related to BigKV. The repository includes four caching systems (BigKV, `uDepot-OPT`, `uDepot-Cache`, and `SlickCache`. You may compare the performances of the cache systems by using the YCSB benchmark with a script in this repository. We also provide detailed instructions to run a cache.

Hardware requirements. For the evaluation, you must prepare a separate block device in addition to the block device on which the root file system is mounted. We also recommend that your server has 4GB of DRAM and a CPU with 8 cores at least.

Software requirements. BigKV uses several libraries and third parties. They include the open-source CityHash algorithm, io_uring, libaio, etc. The README file in the repository describes detailed instructions for the installation.

Hosting. We use the following Git URL and the commit hash for the repository during artifact evaluation.

- BigKV
  https://github.com/dgist-datalab/bigkv
  `eef715b7c94b8fe42223eca118a785df17a0816b`

A.4 Set-up & Evaluation Workflow

We recommend evaluators follow the steps on the README file. You can refer to the "Prerequisites" section to install the software that BigKV uses. The "Installation" section describes how to build our caches. Finally, the "Execution" section provides detailed instructions to install the YCSB benchmarks, run the test script, and see the results of the evaluation.