AC-Key: Adaptive Caching for LSM-based Key-Value Stores

Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H.C. Du,
University of Minnesota

https://www.usenix.org/conference/atc20/presentation/wu-fenggang

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

AC-Key: Adaptive Caching for LSM-based Key-Value Stores

Fenggang Wu Ming-Hong Yang Baoquan Zhang David H.C. Du
University of Minnesota, Twin Cities

Abstract

Read performance of LSM-tree-based Key-Value Stores suffers from serious read amplification caused by the leveled structure used to improve write performance. Caching is one of the main techniques to improve the performance of read operations. Designing an efficient caching algorithm is challenging because the leveled structure obscures the cost and benefit of caching a particular key, and the trade-off between point lookup and range query operations further complicates the cache replacement decisions.

We propose AC-Key, an Adaptive Caching enabled Key-Value Store to address these challenges. AC-Key manages three different caching components, namely key-value cache, key-pointer cache, and block cache, and adjust their sizes according to the workload. AC-Key leverages a novel caching efficiency factor to capture the heterogeneity of the caching costs and benefits of cached entries. We implement AC-Key by modifying RocksDB. The evaluation results show that the performance of AC-Key is higher than that of RocksDB in various workloads and is even better than the best offline fix-sized caching scheme in phase-change workloads.

1 Introduction

The persistent Key-Value Store (KVS) has become an indispensable storage engine in many applications [1–4] for its flexibility and scalability. Existing KVSs, e.g., LevelDB [5], RocksDB [6], Cassandra [7], etc., use a Log-Structured Merge (LSM) tree [8] to improve the performance of write operations. However, their read performance is sacrificed because of the log-structured nature of LSM trees, where finding a key by searching several levels could incur multiple storage I/Os [9–11].

Caching is one of the main techniques to improve read performance since workloads usually demonstrate certain amounts of access locality. Studies show that read operations exhibit "hot spots" in enterprise workloads on LSM-tree-based KVSs (LSM-KVS) for both point lookups [12–14] and range queries [15, 16]. In Facebook, some large-scale production use cases of RocksDB exhibit good locality [17]: fewer than 3% of the keys were accessed during a 24-hour UDB workload; in the ZippyDB workload about 1% of the KV-pairs are accountable for 50% of the total Gets.

There are two major unique challenges in designing an efficient caching scheme for LSM-KVS. First, LSM has a multi-level design where the storage I/O saved by each cached key-value pair (*caching benefit*) could be different. The deeper the level where the KV pair resides, the more storage I/Os can be saved if it is cached. Additionally, the DRAM caching size taken by one key-value pair (*caching cost*) is also different with different key and value sizes. It is challenging for the caching scheme to estimate the cost and benefit and make replacement decisions accordingly. Second, the two types of read operations, namely point lookup and range query, exhibit quite different caching requirements. Point lookup prefers caching an individual key-value pair (KV) for space efficiency [6, 7]. If the value is large, another alternative is to cache a key-pointer pair (KP, where the pointer refers to the location of the value in storage) [7]. In contrast, a range query cannot be served by caching sporadic individual keys, so people resort to caching blocks to support range queries [5, 6]. It is difficult to disentangle the trade-offs among caching KV, KP, and blocks, as each of them has certain types of favorable workloads. Additionally, designing an adaptive caching scheme that can deal with dynamic workloads is more challenging.

Existing caching schemes [5–7, 18, 19] only consider one or two types of entries to cache among KV, KP, and block, and they have a fixed allocated cache budget for one type of entry. Therefore, they cannot leverage all of their merits to cope with various workload scenarios, and cannot adjust the caching space when the workload changes. Besides, to the best of our knowledge, there is no existing work addressing the heterogeneous caching costs and benefits in the unique LSM-KVS scenario.

We comprehensively study the trade-offs among caching KV, KP, and blocks, and propose AC-Key, Adaptive Caching for LSM-based Key-Value Stores, to combine their advantages in handling different workloads. AC-Key uses one des-

ignated caching component for each type of the entries (KV, KP, and block). The size of each caching component is dynamically adjusted by the proposed *hierarchical adaptive caching* algorithm that uses *ghost caches* to guide the size adjustment. AC-Key leverages a novel *caching efficiency factor* that quantifies the different caching costs and benefits to aid the boundary adjustment among the caching components as well as the replacement decision within each caching component.

We implement AC-Key based on RocksDB [6]. Our benchmark evaluations show that the read performance of AC-Key is higher than that of the default RocksDB by up to 57.1%. In the phase-change workloads, AC-Key can achieve even better performance than the best offline fix-sized caching scheme. We also evaluate AC-Key using YCSB [15] where AC-Key outperforms the default RocksDB by up to 59.9%.

The rest of the paper is organized as follows. We first provide the background and motivation in §2 and §3, respectively. Then we present the design of AC-Key in §4 and analyze the performance of AC-Key in §5. §6 concludes the paper.

2 Background and Related Work

2.1 LSM-Tree-Based Key-Value Store

Popular implementations of LSM-tree-based Key-value stores (LSM-KVSs), such as LevelDB [5] and RocksDB [6], consist of two parts, a memory component and a storage component. The memory component, or *MemTable*, is typically implemented using in-place sorted data structures such as skip-list or B+ tree. The storage component is implemented as levels of files storing sorted runs of key-value pairs compactly. As shown in Fig. 1, one level is partitioned into multiple *sorted string table* files, or *SSTs*. Each SST has a configurable size limit, typically 2MB~64MB.

When the MemTable is full, it will be formatted into an SST and written to the storage component. This procedure is called the *flush* operation. Each level in the storage component has an exponentially increasing size limit (by default 10 times larger than the previous level). Therefore, larger levels will have more SSTs. L_0 SSTs will be merged with the L_1 SSTs when the specified size limit is reached. After that, L_1 SSTs will then be merged with L_2 SSTs, so on and so forth. This is called the *compaction* operation.

Every level is a single sorted run (except L_0) where the SSTs have disjoint key ranges. The sorted run of key-value pairs of an SST are divided into multiple *data blocks*, and the boundary keys between every two adjacent data blocks are stored in an *index block* with the corresponding data block offset within the SST. Besides, SST also contains a bloom filter block (BF block) to determine the existence of a key in this SST hence to save unnecessary storage I/Os. Block is the basic storage I/O unit in LSM-KVS.

There are two types of read operations in LSM-based key-value stores, namely *point-lookup* (or `Get`) and *range query* (or `Scan`). `Get` is to retrieve the value of a specific key in the following sequence: MemTable, every SST in L_0 from the youngest to the oldest, then L_1 to L_N. If the key is found in the MemTable, it will return with the value without any storage access. Otherwise, the key-value store will search the SSTs in the storage component. It will first check the bloom filter of one SST and skip the SST if the bloom filter indicates that the key does not exist. Otherwise, the index block will be read to locate the corresponding data block. Finally, the data block is retrieved and searched for the key. Therefore, a key-value store needs at most three storage I/Os when searching an SST for a specific key.

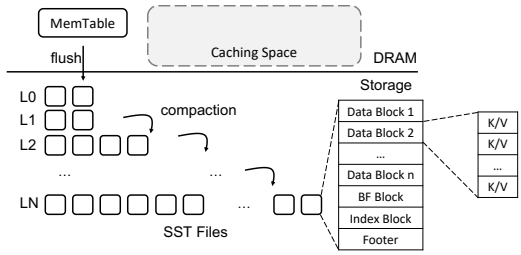


Figure 1: LSM-based Key-value Store (LSM-KVS).

LSM-KVS performs a range query using a starting key and an ending key or a number specifying how many key-value pairs to return. To execute a range query, the KVS uses a `Seek()` function to construct a merging iterator that can iterate through the MemTable, all SSTs in L_0, and one SST in each of the larger levels at the same time. Then, the KVS will call a `Next()` function to return the next larger key. When the key returned by the `Next()` function is larger than the ending key, or the number of the returned key-value pairs has reached the specified number, the range query will be terminated.

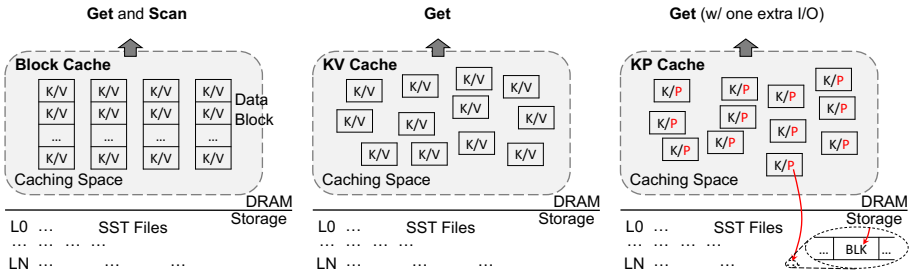
2.2 Related Work

2.2.1 Caching Schemes in LSM-KVS

There are three type of entries that can be cached in LSM-KVS: block, KV, and KP (Fig. 2).

LevelDB [5] only adopts the Block Cache (Fig. 2a), where the blocks could be data block, index block, or Bloom Filter (BF) block. The blocks in the Block Cache are indexed using the SST file ID and the block offset (`<SstID|BlockOffset>`). Block Cache can be beneficial for both point lookup and range query operations. Storage I/Os can be saved as long as the target block is cached. However, Block Cache is not space-efficient to serve point lookup, as the whole block has to be cached even though only a small portion of the keys in the block are accessed frequently.

RocksDB [6] has both Block Cache (Fig. 2a) and KV Cache (Fig. 2b). The KV Cache stores KV pairs that can serve point lookup. However, the sizes of the Block and KV Caches in RocksDB are predefined and fixed. When the KV cache is enabled, point lookup will first consult the KV cache if the write buffer does not contain the key. If the key was not cached in the KV Cache, RocksDB will follow the normal read process using Block Cache as LevelDB does, and insert



(a) Block Cache **(b)** KV Cache **(c)** KP Cache

Figure 2: Three Types of Entries to Cache in LSM-KVS.

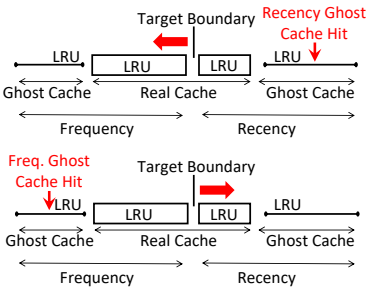


Figure 3: ARC Algorithm.

the KV pair into the KV cache afterwards. Range queries are supported by the Block Cache.

Cassandra [7] does not have a Block Cache but has both KV Cache (Fig. 2b) and KP Cache (Fig. 2c. In the KP Cache, the locations of the values in the storage are cached in memory as pointers). On a hit in the KP Cache, one point lookup can be served with only one storage I/O, skipping all the shallower levels in the storage component. Compared with KV Cache, KP Cache is more space-efficient for large value sizes at a price of one extra storage I/O. Similar to KV Cache, KP Cache cannot handle range queries. When promoting a key from KP Cache to KV Cache, Cassandra does not remove the KP entry.

zExpander [18] is a KVS caching scheme which caches key-value pairs only. It partitions caching space into a compressed zone (Z-zone) and an uncompressed zone (N-zone) and can adapt the boundary between them. LSbM [19] has a small on-disk *compaction buffer* that keeps old, but hot data from being deleted during compaction to reduce the block cache invalidation due to compaction. It needs extra storage space for compaction buffer, and does not take advantage of the more efficient KV or KP Cache to support point lookup.

2.2.2 General Caching Algorithms

The page cache replacement problem has been studied for decades [20]. Adaptive Replacement Cache (ARC) [21] is a dynamic page replacement algorithm designed for managing the page cache in DRAM. As shown in Fig. 3, ARC divides the caching space into two parts, the *recency cache* and the *frequency cache*, each of which is an LRU cache. A page is brought into the recency cache when first encountered. If the page gets a second access before being evicted, it is considered a frequently accessed page and will be migrated to the frequency cache.

The space distribution between the recency cache and the frequency cache is dynamic. ARC uses two *ghost caches* to store metadata of evicted pages (page number) from the recency and frequency cache respectively. The pages stored in the ghost cache will be a future reference for adjusting the allocated space for each part. Compared with the *real cache*, i.e., the cache stores the actual page contents, the size of the ghost cache is negligible since they only store page numbers.

A hit on the recency ghost cache indicates the recency cache should have been larger, so the target boundary will be moved leftwards (top figure in Fig. 3) and vice versa (bottom figure in Fig. 3). As a result, the size of the corresponding real cache will be increased or decreased according to the workload.

CAR [22] also maintains a dynamic partition between the recency and frequency cache using ghost caches, but it uses CLOCK instead of LRU to manage each caching component to reduce overhead. H-ARC [23] uses Non-Volatile Memory (NVM) to cache both clean and dirty pages and propose a hierarchical algorithm to adaptively handle the page replacement.

Such page-based caching algorithms (e.g. ARC [21], CAR [22], H-ARC [23]) do not fit LSM-KVS well because they are based on identical page sizes and caching benefits (i.e. storage I/Os saved by caching an entry), whereas the entry sizes and caching benefits in LSM-KVS are no longer uniform. Similarly, pure frequency-based cache eviction algorithms (e.g. WLFU [24]) and admission policies (e.g. TinyLFU [25]) assume homogeneous entry size and caching benefit too and make replacement decisions solely based on the access frequency. However, in LSM-KVS, the difference in entry size and caching benefit should also be considered along with frequency. Web caching algorithms often take into account the entry size information [26]. In LSM-KVS, however, besides entry size, the caching benefit for different entries is also diverse. For example, caching a KV from a deeper level will save more storage I/Os than caching one from a shallower level. Such special knowledge of the leveled structure of LSM-tree should be exploited to aid caching decisions in LSM-KVS.

3 Motivation

3.1 Unique Challenges in Caching for LSM

Comparing with the page cache replacement problem [20–23], where the pages have identical sizes, keys and values in the LSM tree do not necessarily have the same size. The caching algorithm in LSM should also take the size difference into consideration when designing the replacement algorithm.

Hash-based KVS [27, 28] does not support range queries. In contrast, LSM-KVS has two distinct read operations, point

Table 1: LSM-KVS Caching Scheme Comparison

Cached Entry	Space Efficiency	Extra I/O	Get Existing	Get Missing	Scan	Compaction	Flush	Favorite Workload
Block	Low	No	Helpful	Helpful (BF block)	Helpful	Affected	Not Affected	Scan / Get (missing)
KV	High	No	Helpful	Not Helpful	Not Helpful	Not Affected	Affected	Get (hot/small value)
KP	High	Yes	Helpful	Not Helpful	Not Helpful	Affected	Affected	Get (warm/large value)

lookup and range query, that exhibit quite different caching requirements, and brings additional challenges in the design of LSM caching algorithms.

B+ tree-based KVS [29] supports both point lookup and range query. LSM-KVS is different as it has a leveled structure that diversifies the caching benefit of KV pairs on different levels. The deeper level the KV resides, the more storage I/O can be saved by caching this KV pair. Besides, native operations in LSM-KVS such as compaction and flush do not exist in B+ tree-based KVS, but they will invalidate cached entries and need special treatment in the design.

To motivate our design, we discuss two key question about the LSM-KVS caching: *what to cache*, and *how to perform replacement*, while summarizing the lesson learned.

3.2 What to Cache in LSM-KVS

For point lookup which retrieves the value for a given key, it's natural to directly cache the hot KV pairs in the DRAM cache. However, when the value size is large, and/or the access is less frequent, another alternative is to cache a pointer referring to the location of the value on the storage component. When looking for this key, the value can be fetched using one storage I/O instead of performing multiple I/Os along every SST from L_0 down to the level where the KV resides. By storing a smaller pointer instead of the original larger value, KP Cache can hold more entries. The hit ratio of KP Cache will be higher than that of a KV Cache and can potentially save more I/Os. Comparing KV with KP entry, a hit on a KV entry can save more storage I/Os since a KP hit still needs one storage I/O to get the value. On the other hand, caching KP entries is more space-efficient in the case of relatively large value sizes.

Lesson 1: The merits of caching KV and KP entries should be combined to efficiently serve point lookups.

Unfortunately, cached KV and KP entries cannot help with range queries. Given the staring key of a range query, the next larger key cannot be determined by only examining the sporadic cached KV or KP entries. Therefore, some LSM-KVS implementations such as LevelDB [5] and RocksDB [6] cache data blocks for range queries.

Cached data blocks can serve point lookup too. However, retaining a whole block for point lookup is not space-efficient as it keeps a whole block in the DRAM even only a few keys are frequently looked up. Beside data blocks, frequently accessed index blocks and bloom filter blocks (BF blocks) are also cached in Block Cache.

Lesson 2: Cached blocks and KV/KP entries each have their advantage to support range query and point lookup.

3.3 How to Perform Replacement

From the discussion above, we reach to the conclusion that caching KV, KP, and block each has its own favorable workload scenario. We have a comprehensive comparison summarized in Table 1. However, designing the replacement algorithm for a cache that consists of all these three different entries is challenging.

A straightforward approach is to treat all the cached entries (KP entries, KP entries, or blocks) equally and borrow existing replacing schemes, such as LRU (Least Recently Used) or LFU (Least Frequently Used), to manage the cache. Each entry is inserted or evicted according to the corresponding eviction policy. Therefore, the number of cached entries among the cached KV entries, KP entries, and blocks are solely determined by the access pattern. However, this "unified" caching approach is too simplified and cannot distinguish the differences between these cached entries. First, different cached entries have different sizes. For example, one cached block may take up the DRAM space which can hold tens or hundreds of KV/KP entries. Second, different cached entries have different numbers of saved storage I/Os. For example, one cached block saves one I/O if hit, but one cached KV entry could save multiple storage I/Os for all the SSTs to be accessed for a point lookup. Besides, KV/KP entries on different levels will have different numbers of storage I/O saved too. General cost-aware caching schemes [30, 31] does not have special analysis on the caching cost and benefit in this LSM-KVS scenario either. Third, if there is a single-pass large range query, the fetched blocks will evict useful entries out of cache without bringing any benefit.

Lesson 3: The caching algorithm should consider the difference of DRAM size taken and number of storage I/O saved among different cached entries, respecting the unique leveled structure of LSM-KVS.

Another approach is to have designated fix-sized KV Cache, KP Cache, and Block Cache for the corresponding entries, and perform independent eviction decisions based on popular caching algorithms, such as LRU or LFU. In this case, the cache is resilient to the large single-pass range query (as mentioned before) because the each caching component has a bounded capacity and will not influence the other caching components. However, this fix-sized approach has some problems. First, it is difficult to get the size distribution right in the first place. Second, even if we could have a favorable fix-size configuration at the beginning, it might not be suitable later on as the access pattern of workloads could change over time. For example, a workload has a phase change from range query

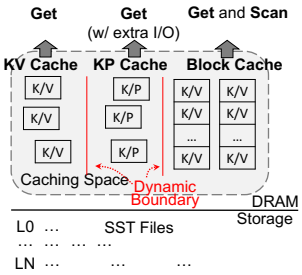


Figure 4: AC-Key Caching Components

dominant into point lookup dominant. If the fixed configuration is good for the range query at first, i.e., majority of the cache space is used as Block Cache, in the point lookup phase, it is clearly not efficient.

Lesson 4: The caching algorithm should be adaptive to the workload changes.

4 AC-Key Design

AC-Key (Fig. 4) caches all three types of entries – KV, KP, and block – with designated caching components for each of the three. Different from the fix-sized scheme described in the previous section, AC-Key has *dynamic* sizes for each of the caching components. The sizes are adjusted by the proposed *hierarchical adaptive caching* algorithm. Considering the heterogeneous costs and benefits of different cached entries and the unique leveled structure of LSM-KVS, AC-Key uses the proposed *caching efficiency factor* to quantitatively guide the size adjustment among the caching components as well as the replacement policy within each caching component.

4.1 AC-Key Caching Components

The AC-Key system architecture is depicted in Fig. 4. The storage component is identical to popular LSM implementations (§2). The DRAM caching space has three components: the *KV Cache*, the *KP Cache*, and the *Block Cache*. The Block, KP, and KV Caches are managed by E-LRU, an improved LRU with cache efficiency factor based eviction (see §4.2).

KV cache stores the key-value pairs directly. KP Cache holds keys with pointers, where the pointers are in a format of `<SstID|BlockOffset>`. When a KP Cache entry is hit, it takes only one storage I/O for the KVS to fetch the data block which contains the target key-value pair. Block Cache stores frequently accessed blocks, which can be either a data block, an index block, or a Bloom Filter (BF) block.

Remarks on the KV and KP Cache. In a point lookup operation, if a *lookup-key* is accessed for the first time, it will be brought to the KP Cache. A key cached in KP cache is called a *warm key*. If a warm key in KP cache is hit again, we consider the key as a *hot key*. We anticipate that it has a higher probability to be accessed again in the future. Therefore, we "promote" the key to KV Cache to potentially save more I/Os from the future accesses. Different from the existing solution in [7] that still keeps key in the KP Cache, AC-Key removes the key from KP Cache to avoid the duplicity and achieve better space-efficiency.

In our current design, we will not "demote" a hot key from KV Cache to KP Cache since we no longer have the pointer information. Another design alternative is to have both the pointer and value stored in KV Cache. However, the accompanying pointer will occupy extra cache space of the KV Cache hence we do not take this approach. As an optimization, if the value size of a KV pair is smaller than the pointer size (implementation-dependent, 24 B in our case), we will cache the *value* with the key into the KP Cache instead of the *pointer* to save the extra I/O without paying more DRAM caching space. This entry still needs another hit to be promoted to the KV Cache. AC-Key does not directly insert this KV entry to the KV cache. The reason is that if this key is not "hot" enough, i.e., having less chance to be hit again, it will evict other "hot" entries from the KV Cache. For example, the workload at a certain time starts to access through a substantial number of keys with small values without a second hit, those keys will occupy the whole KV cache, kicking out useful KV pairs without bringing any benefit.

4.1.1 `Get` Handling

`Get` Existing Key. Denote K as the key to search. First, the MemTable is searched for K as it potentially has the latest version of the value. If not found, then the KV and KP Cache is searched for the key. One of the following cases will happen.

- *Case I*: **Hit in KV Cache**. The value is returned without any I/O incurred.
- *Case II*: **Miss in KV Cache but hit in KP Cache**. Using the cached pointer (`<SstID|BlockOffset>`) as block handle, AC-Key will check whether the data block is already cached in the Block Cache. If not, AC-Key will load the data block into the Block Cache. Using binary search, AC-Key locates the KV pair in the data block and then serves the `Get` request. Besides, the key will be migrated to the KV Cache, promoting the cached entry from key-pointer format to key-value format.
- *Case III*: **Miss both in KV Cache and KP Cache**. AC-Key will examine every sorted run level by level, identifying each SST that has a key range overlaps K, from the youngest to the oldest. Once the key is found in a certain SST, AC-Key will stop searching and return the result directly. Besides, the location of the KV pair, i.e., the pointer, will be recorded and cached in the KP Cache indexed by the key.

In case III, when searching one SST, AC-Key first consults the BF block of this SST. If the BF block is not in the Block Cache, it will be fetched from the storage and inserted into the Block Cache. If the BF block indicates the key is not in the SST, AC-Key will skip this SST and proceed to the

next SST. Otherwise, AC-Key will access the index block to narrow down the scope and pinpoint which data block to search. The index block and data block will be fetched from storage and inserted to the Block Cache if not already cached. The BF block, index block, and data block share the Block Cache similar to RocksDB [6].

`Get` Missing Key. When looking up a missing key K in AC-Key, Case I and II of §4.1.1 will not happen as there will be cache miss in both KV and KP Caches.

Similarly, in Case III, the Block Cache will be searched for the corresponding BF block and one storage I/O will be saved if hit. Getting missing keys is the "worst" case where all the overlapping SSTs will be checked. However, by using the cached BF blocks in the Block Cache, multiple storage I/Os can be saved.

4.1.2 `Flush` Handling

`Flush` will dump the MemTable that contains the latest version of the values into the storage component as an L_0 SST. It is possible that a key inserted to the MemTable is already cached in either KV or KP cache. Insertions (also called `Put` operations) to the MemTable will obsolete the corresponding cached entries. As long as the new version of value is still in MemTable, obsolete entries in the caches do not matter because point lookup operation will always consult the MemTable first. However, the cached KV and KP entries must be synced before the keys is flushed to the storage to avoid returning stale results.

We have two alternatives of the timing of the sync: during `Put` or during `Flush`. If during `Put`, the KV and KP Caches will be checked for potential obsolete entries in each `Put` and update the KV entry or delete the KP entry accordingly. Note that during `Put`, AC-Key cannot update the KP entry since the latest value of such key has not been written into an SST yet, hence there is no way to know where the pointer should point to. This approach introduces significant performance overhead because of the extra checking during *every* `Put` operation. Besides, if a key gets multiple updates before flushed from MemTable, we have to repeatedly check both KV and KP Cache for it, which further increases the sync overhead.

Therefore, AC-Key takes another alternative to sync the caches only during `Flush` time. It can accumulate multiple updates to the same key and only sync the KV or KP cache once for each key. Besides, during `Flush` time, AC-Key can figure out the new pointers of keys in the KP Cache and update them accordingly.

4.1.3 `Compaction` Handling

Compaction will affect KP and Block Caches since it creates new SSTs and delete old ones. The old SSTs deleted during compaction may contain blocks that are already cached in the Block Cache. Such deleted SSTs may also contain data blocks being referenced by the pointers cached in the KP Cache. However, it will not affect the entries in KV Cache because compaction reorders and consolidates old KV pairs instead of inserting new ones. AC-Key updates KP and block Caches when compaction affects any of cached KP entries or blocks.

For the KP Cache, during the compaction, AC-Key identifies affected KP entries in the KP-Cache and update the pointers to point to the new data blocks that contain the keys. For the Block Cache, if one cached data block is to be invalidated during compaction, AC-Key will replace the invalidated data block with one new data block generated by compaction to avoid the invalidated block wasting the Block Cache's capacity. The key range of the newly generated data block may not be exactly the same as the old one. AC-Key chooses the block that has the largest overlap with the old cached block and repopulates it back to the Block Cache. Similarly, invalidated cached BF blocks and index blocks are also replaced by the new ones with the most overlapping key ranges. Such block replacement does not incur extra I/O, as the new blocks are generated in memory during compaction.

4.2 Caching Efficiency Factor

To quantitatively analyze the trade-off between the costs and benefits of the cache entries, we propose the novel *caching efficiency factor* that takes into account the unique level structure of LSM-KVS. Using this caching efficiency factor, AC-Key improves LRU into *E-LRU* to manage cache evictions within each caching component, and modifies ARC into *E-ARC* to adjust the size of each caching component. We introduce the caching efficiency factor and E-LRU in this section, and discuss E-ARC in the next section.

We define the *caching efficiency factor* E (E standing for Efficiency) for one cached entry as the following equation. The meaning of this caching efficiency factor is "the number of saved I/O per byte of DRAM caching space".

$$E = \frac{b}{s}, \tag{1}$$

where: E = caching efficiency factor of one cached entry,
b = number of saved storage I/O if cached,
s = caching space taken by this entry.

For example, one typical cached KV entry will take one or several hundreds of bytes, one KP entry will normally take less than one hundred bytes, and one cached block will take 4~16KB. b denotes the number of I/O being saved if this entry is cached. It is given by:

$$b = \begin{cases} 1 & \text{if block,} \\ f(m) & \text{if KV entry,} \\ f(m)-1 & \text{if KP entry.} \end{cases} \tag{2}$$

where: m = number of SSTs to search for the key,
$f(m)$ = number I/Os to get a key. It is a function of m.

The function $f(m)$ depends on the LSM-KVS implementation. Typically, $f(m) = m + 2$, where we have to read m bloom filters each from one SST along the searching path, as well as one index block and one data block in the SST that contains the lookup-key. The number of SSTs to search, m, is estimated by the level l where the key resides:

$$m = \begin{cases} n_0/2 & \text{if } l = 0, \\ l + n_0 & \text{if } l >= 1. \end{cases} \quad (3)$$

where: n_0 = max number of SSTs L_0 can hold,
l = the level where the key resides.

If $l = 0$, the key is in L_0. AC-Key assumes $m = n_0/2$ as an estimate of the average number of SST to search for a key in L_0. If the lookup key resides in levels greater than L_0, potentially every SST in level L_0 will be checked. So, AC-Key uses n_0, the max number of files in L_0, to estimate m.

E-LRU. Traditional LRU only considers the access pattern without taking care of the different benefits and costs of the cached entry. We develop E-LRU, the efficiency-based LRU, to address this issue. E-LRU checks the least used a cached entries and evict one with the least caching efficiency E. The value of a depends on the variance of the caching efficiency factor E of the cached entries. It is given by $a = e^v$, where v is the standard deviation of the caching efficiency factor E of sampled entries in the caching component. When $v = 0$, meaning cached entries has identical efficiency, then $a = 1$, and E-LRU degenerates to the original LRU algorithm that evicts the last one entry from the list. The larger v, the more variance of the efficiency E, the greater a should be used to select a better candidate to evict. In the current implementation, we have a cap on a to avoid AC-Key checking too many entries when making eviction decision. We use E-LRU for simplicity, yet other cost-aware caching schemes [30, 31] can be adapted here using our proposed caching efficiency factor.

4.3 HAC: Hierarchical Adaptive Caching

Hierarchical Adaptive Caching (HAC) has a two-level hierarchy to manage different caching components (Fig. 5). On the upper level, the cache is divided into two components: the Point Cache and the Block Cache. The boundary between the Point Cache and the Block Cache is dynamically adjusted. On the lower level, the Point Cache is further divided into KV Cache and KP Cache with an adjustable boundary too. HAC maintains *ghost caches* to keep a record of the evicted entries from the KV, KP, and Block Cache. On the upper level, there are two ghost caches for the Point Cache and the Block Cache respectively, while in the lower level, there are two ghost caches each for the KV and KP Cache. Opposed to ghost caches, the original KV, KP, and Block Caches are called *real caches*. Here the KV and KP Real Caches collectively make up the Point Real Cache. The ghost caches do not hold the real entry, but only metadata of the evicted entries. A hit in the ghost cache means it could have been a real cache hit if the corresponding real cache was larger. By using ghost cache with the caching efficiency factor, we design E-ARC (caching Efficiency enabled ARC) to adjust the size of the corresponding real cache.

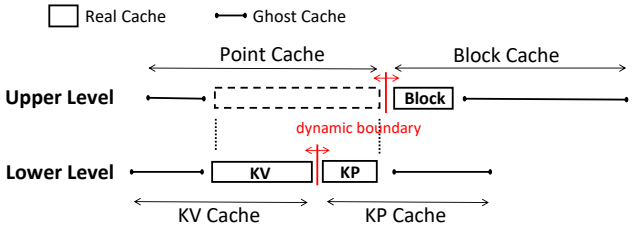


Figure 5: Hierarchical Adaptive Caching (HAC) Algorithm.

4.3.1 Lower-Level HAC

AC-Key uses E-ARC, or efficiency based ARC, to manage the lower-level HAC. On the lower-level HAC, the Point Cache is divided into the KV Real Cache (R_{kv}) and KP Real Cache (R_{kp}). That is: $|R_{kv}| + |R_{kp}| = S_{point}$, where $|\cdot|$ means size, and S_{point} denotes the Point Cache size. AC-Key maintains the KV Ghost Cache as if the KV Real Cache R_{kv} plus the KV Ghost Cache G_{kv} equals to the total size of the Point Cache. Note that the KV Ghost Cache only holds the metadata of the keys evicted from the KV Cache. Denote $|G_{kv}|$ as the size if they were storing the whole KV pair, then $|R_{kv}| + |G_{kv}| = S_{point}$. Similarly, with G_{kp} denoting the KP Ghost Cache, we have $|R_{kp}| + |G_{kp}| = S_{point}$.

Therefore, the following equation is always maintained:

$$S_{point} = |R_{kv}| + |R_{kp}| = |R_{kv}| + |G_{kv}| = |R_{kp}| + |G_{kp}|. \quad (4)$$

We show how E-ARC handles cache hits and misses in the following cases:

- Case I: **Real Cache Hit**. Cache hits on R_{kv} or R_{kp}. Move the hit entry to the MRU end of R_{kv}. Especially, if the hit happens on R_{kp}, AC-Key needs one storage I/O to get the value. Then the key and value is inserted into R_{kv} (promotion).

- Case II: **KV Ghost Cache Hit**. Cache hits on G_{kv} means the size of R_{kv} should have been larger. Shift the target boundary towards the KP Cache end by $\delta = kE$, where E is the caching efficiency factor of the hit entry on G_{kv}. Here k is a configurable learning rate. After fetching from storage, insert the fetched KV to the MRU end of R_{kv}. To make room for this KV entry, evict from R_{kv} (resp. R_{kp}) if the target boundary is within R_{kv} (resp. R_{kp}), meaning that the target size of R_{kv} (resp. R_{kp}) is smaller than its actual size.

- Case III: **KP Ghost Cache Hit**. Cache hits on G_{kp} means the size of R_{kp} should have been larger. Shift the target boundary towards the KV Cache end by $\delta = kE$.

E is the caching efficiency factor of the hit entry on G_{kp}. After fetching from storage, insert the fetched KV to the MRU end of R_{kv}. To make room for this KV entry, evict from R_{kv} (resp. R_{kp}) if the target boundary is within R_{kv} (resp. R_{kp}), similar to Case II.

- Case IV: **Cache Miss**. Retrieve the entry from storage and cache to R_{kp} in KP format. To make room for this KP entry, if the target boundary is within KV cache, evict from R_{kv}. Otherwise evict from R_{kp}.

The target boundary between the KV Real Cache R_{kv} and KP Real Cache R_{kp} indicates the direction the actual boundary should move. The actual boundary will normally lag behind the target boundary. The high level sequence of operations is as follows: 1) ghost hit adjusts the target boundary; 2) entry insertion or promotion shifts the actual boundary towards the target boundary, and as a result, real cache sizes $|R_{kv}|$ and $|R_{kp}|$ are updated; 3) ghost cache sizes are adjusted based on the new real cache sizes using the Eqn. 4; and 4) real and ghost caches perform eviction if necessary to fit the updated sizes using E-LRU (§4.2).

Remarks on E-ARC. Although we follow a similar logic of the canonical ARC algorithm, the original ARC does not have the size and cost differences. In the original ARC, the saved block access b and space cost s for each entry are always the same. E-ARC's definition of $\delta = kE = k\frac{b}{s}$ is a generalization of that of the canonical ARC, and the ARC's definition of the adjustment is a special case of E-ARC's formula where $\frac{b}{s} = 1$.

4.3.2 Upper-Level HAC

On the upper level of HAC, we re-apply the E-ARC scheme to adjust the boundary between Point Cache and Block Cache. Block Cache and Point Cache each has a real cache (R_{block} and R_{point}) and a ghost cache (G_{block} and G_{point}). R_{kv} and R_{kp} collectively forms R_{point}. Blocks evicted from R_{block} enter G_{block}. On the other hand, entries evicted from R_{point} (R_{kv} or R_{kp}) will be inserted to G_{point}. Note that the evicted entry will also be inserted to the corresponding KV or KP Ghost Caches (G_{kv} or G_{kp}) in the lower level (§4.3.1). Similarly to the low level of HAC, the sum of the virtual "size" of the real cache and ghost cache of the Block Cache (resp. Point Cache) will be the total available cache size:

$$\begin{aligned} S_{total} = &\quad |R_{block}| + |R_{point}| \\ = &\quad |R_{block}| + |G_{block}| = \quad |R_{point}| + |G_{point}|. \end{aligned} \tag{5}$$

Target Boundary Adjustment. A ghost hit on G_{block} will move the target boundary between R_{point} and R_{block} toward R_{point} by $\Delta = kE$, where E is the caching efficiency factor of the entry on G_{block} being hit, and k is the learning rate as defined earlier. As a result, the target size of R_{point} will be reduced by Δ. Then, in the lower level, the amount of the adjustment will be distributed between the target size of R_{kv} and R_{kp} proportionally to their current target size ratio. In this example, target size of R_{point} is shrinking by Δ. Denoting the current target size of R_{kv} as $|R^*_{kv}|$ and the target size R_{kp} as $|R^*_{kp}|$, they will be updated as follows: $|R^*_{kv}| \leftarrow |R^*_{kv}| - \Delta\frac{|R^*_{kv}|}{|R^*_{kv}|+|R^*_{kp}|}$, and $|R^*_{kp}| \leftarrow |R^*_{kp}| - \Delta\frac{|R^*_{kp}|}{|R^*_{kv}|+|R^*_{kp}|}$.

On the other hand, a ghost hit on G_{point} will move the target boundary toward R_{block}, i.e., making the target size of R_{block} smaller and that of R_{point} larger. The adjustment amount is also $\Delta = kE$. The caching efficiency factor E of an entry in G_{point} will normally be larger that that in G_{block}, as a Point Cache entry (either KV or KP entry) takes less DRAM caching space and save a greater number of storage I/Os than one Block entry. While there is a ghost hit on G_{point} on the upper level, normally there will be a ghost hit in the lower level too, either in G_{kv} or G_{kp}. In this case, the lower level target boundary will be adjusted first, then that of the upper level. For example, if the ghost hit happens both in the KV Ghost Cache G_{kv} and the Point Ghost Cache G_{point}, the target size of R_{kv} and R_{kp} will first be updated as described in §4.3.1), then increment of Δ is distributed proportionally to the new target size of R_{kv} and R_{kp} accordingly.

Actual Boundary Adjustment. On a block miss when this block needs to be inserted to the Block Cache, if the current Block Cache size plus the new block is greater than the target block size, the Block Cache will not grow. It will evict one block from itself to make space for the new-coming block. Otherwise, if the current Block Cache size plus the new block is within the target Block Cache size, the Block Cache will expand by inserting the new block, and the Point Cache will shrink to make room for the growth of the Block Cache. Typically, more than one KV and KP entries will be evicted because a block is normally larger than cached KV and KP entries. The number of KV or KP entries to be evicted will be based on current target size of the KV and KP Real Caches.

On the other hand, when the Point Cache demands more capacity, (i.e., when new KP entry is inserted to the KP Cache, or when a KP entry is promoted to KV Cache and takes more space), HAC estimates the new Point Cache size after the growth and compares it with the target Point Cache. If the estimated new Point Cache size is within the target Point Cache, one block from the Block Cache will be evicted to make the room for the Point Cache to grow. On the other hand, if the estimated new Point Cache size is above the target Point Cache size, Point Cache will not expand, and the eviction will happen within the Point Cache, following the current target KV and KP Real Cache sizes. One problem is that increasing the size of Point Cache by small amount may result in a whole block evicted from the Block Cache. To address this, not until the target boundary is within the Block Cache for at least one block's size (typically 4KB or 16KB) does HAC evict blocks from the Block Cache. In other words, the size of the Point Cache will grow at the expense of evicting from the Block Cache only when the target Point Cache size is greater by the current actual Point Cache by a whole block size.

4.3.3 Reduce Ghost Cache Size

In the ARC design [21], when a page is evicted from the real cache, the page content is dropped, and only the page number is retained in the ghost cache. The size of the page number is negligible compared to the page content. Similarly, in AC-Key, the ghost cache for the Block Cache G_{block} only stores the block handle in the format of `<SstID|BlockOffset>` (24 B in our implementation) which is also negligible compared to the cache block contents (typically 4~16 KB). However, the ghost caches of G_{point}, G_{kv}, and G_{kp} have significant overhead that can no longer be ignored. For example, assuming a key size of 16 B and the value size of 100 B, one real KV entry takes 116 B. When this entry is evicted from the real cache, the value is dropped, and the key is inserted into the KV ghost cache which still takes 16 B. As the key size is no longer negligible when comparing to the value size and pointer size, a ghost cache potentially occupies a substantial portion of the limited caching space, impair the caching efficiency.

We propose two ways to reduce the space overhead of the ghost caches. First, instead of using the original key of the evicted KV or KP entries, AC-Key only stores a hash value of the evicted key as a "fingerprint". In this way, AC-Key reduces the ghost cache size overhead while sacrificing the ghost cache hit accuracy. Hash collision will cause false-positive ghost hits, resulting in imprecise adjustment decisions. In our implementation, we found a hash value of 4 B shows a good trade-off between the ghost cache overhead and accuracy.

Although using a hash-based fingerprint to replace the key in ghost cache reduces the overhead, a hash value (for example, 4 B) is still taking significant space compared with the real KV and KP Caches. We further propose another optimization to eliminate such ghost cache overhead by disabling ghost cache when the adaptive scheme settles to a favorable capacity distribution among the KV, KP, and Block Cache. If the changes of hit ratios of all the caching components remains within a threshold θ (called *ghost cache turnoff threshold*, 5% by default), the ghost cache mechanism will be turned off, and the space taken by the ghost caches will be reclaimed for real caches. The size of the real caches will be proportionally scaled up to use all the available cache capacity when the ghost cache's space is released. Later, when the access pattern switches and the current size distribution is not favorable, the change in the hit ratio will flag the phase transition. When the hit ratio fluctuates beyond the threshold of θ, the AC-Key will turn the ghost cache mechanism back on until the hit ratio converges again (fluctuating within the threshold θ).

5 Evaluation

5.1 Implementation and Setup

We implement AC-Key based on RocksDB `version 6.2` with roughly 5.6K lines of change in `C++` code. We carry out our experiment on a Dell PowerEdge R430 1U Server. It has two six-core Intel Xeon E5-2620 v3 @ 2.40 GHz processors and 64 GB of DDR3 memory. The operating system is Ubuntu LTS 18.04 with Linux kernel version 4.15.0. The storage device is a 372 GB Intel DC P3700 PCIe SSD formatted as `xfs`.

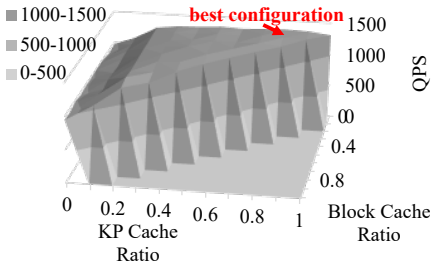


Figure 6: The `offline` scheme tries out on different configurations (at a smallest granularity of 1/10 of the cache size) and select the fix-sized configuration with the best result.

We load a 100GB database with randomly generated keys [9, 11]. The default key size is 16 B and value size is 100 B, which is the default value of RocksDB [32]. The length of range scans is set as 100, which is close to the length used in literature [15, 33–35]. The default point lookup to range query ratio is set to 1:1. We use the existing exponential function based workload generator in RocksDB [32] to generate workloads with different skewness of hot keys (point lookup key and scan starting key). In our experiments, we take the default skewness as the hottest 1% keys taking up the 99% of the accesses (including both the key for point lookup and the starting key for range query). The default learning rate k is 100K (see §4.3) and default ghost cache turnoff threshold θ is 5%. Data compression is disabled to rule out unrelated performance interference and simplify the analysis as in literature [9, 36]. We specify identical cache sizes in the configuration files of RocksDB and YCSB to ensure the same amount of caching budget is used in competing schemes. Page-based direct I/O [37] is enabled to rule out the interference of the OS buffer cache, similar to [38, 39].

The following schemes are compared.

- `pure-kv`: The whole caching space is used as KV Cache.
- `pure-kp`: The whole caching space is used as KP Cache.
- `rocksdb`: Off-the-shelf RocksDB with default setting. Note that RocksDB disables KV cache by default and the whole caching space is used as Block Cache.
- `offline`: We try combinations of different component size with the granularity of 1/10 of the cache size and select the best fix-sized configuration. Note that such best configuration is determined offline, and is not applicable in a real-time caching system (Fig. 6).
- `ac-key`: Our AC-Key scheme.

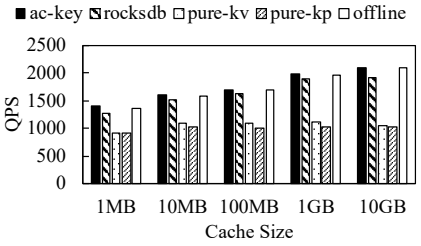


Figure 7: Varying Cache Size.

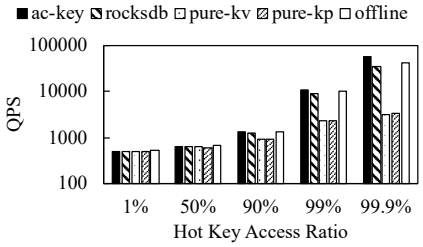


Figure 8: Varying Skewness.

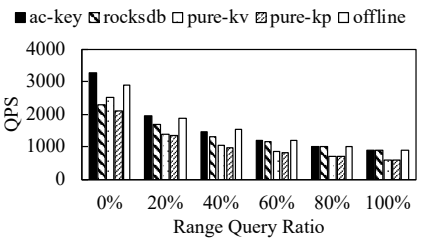


Figure 9: Varying Range Query Ratio.

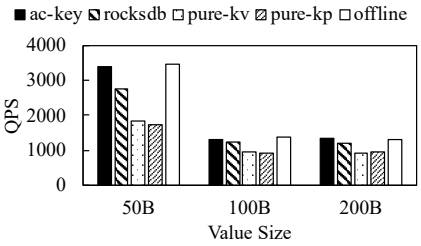


Figure 10: Varying Value Size.

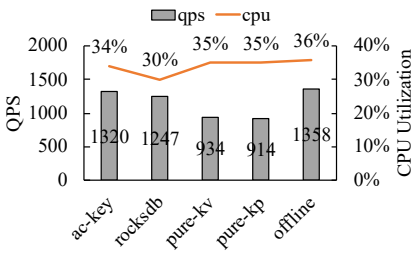


Figure 11: CPU Utilization Comparison.

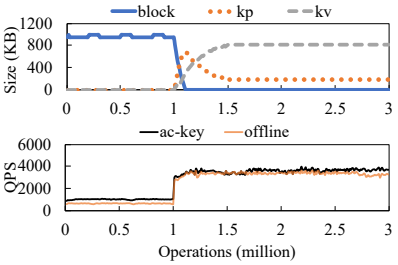


Figure 12: Adaptive Adjustment in AC-Key.

5.2 Micro-benchmark

Varying cache size. We vary the cache size from 1 MB to 10 GB, collect the Query Per Second (QPS) and plot them in Fig. 7. As the cache size increases, the QPS also increases for `ac-key`, `rocksdb`, and `offline`, as larger cache can potentially cache more entries, hence results in less cache misses. In contrast, `pure-kv` and `pure-kp` do not improve much because they do not support range queries, and therefore the range queries will cause many storage I/Os. `ac-key` outperforms other caching schemes that have only a single type of caching component (better than `rocksdb` by 5.0%~9.1%, `pure-kv` by 47.1%~97.7%, and `pure-kp` by 52.8%~104.5%) since they cannot serve both point lookups and range queries efficiently. Comparing with `offline`, `ac-key` performs close to or even better than `offline` because of the ability to adaptively configure the size of each caching component. The reason for the better performance of `ac-key` in some cases is because `offline` cannot exhaust all the possibilities of configurations as `offline` only tries out the cache size at a granularity of one-tenth of the total cache size (Fig. 6).

Varying skewness. We vary the access skewness of the point lookup and the range query using RocksDB's native benchmark tool [32]. In Fig. 8, x-axis shows the access ratio of the hottest 1% ranges from 1% (no skew, uniform distribution) to 99.9% (very skewed). Note that the y-axis is set to log-scale to enhance readability. We can see that when the access is uniformly distributed (left-most cluster of bars) over all the keys, the QPS is similarly low for all the caching schemes. This is because in a uniform-distributed workload, every caching scheme has hardly any hit. In contrast, as the access becomes increasingly skewed, the performance of all the schemes rises, and `ac-key` outperforms `rocksdb` by 3.6%~57.1%, `pure-kv` by 5%~17.6×, and `pure-kp` by 7%~16.5×.

Varying range query ratio. We change the ratio of the point lookup and range query in the workload, ranging the range query ratio from 0% (pure point lookup) to 100% (pure range query) to create Fig. 9. In this figure, we see that as the range query ratio increases, the QPS decreases. This is because a range query has more potential storage I/Os and occupies more caching space. `ac-key` has similar result as `offline` and is better than `rocksdb` (by 1.1%~42.6%), `pure-kv` (by 30.4%~54.7%), and `pure-kp` (by 43.0%~52.8%).

When increasing the range query ratio from 0% to 20%, the performance of `rocksdb` becomes better and exceeds `pure-kv`. This is because KV Cache cannot serve range queries. In contrast, Block Cache supports both point lookups and range queries. However, the space efficiency of Block Cache is low which misses the opportunity to cache more useful entries. `ac-key` adaptively combines Block Cache, KV Cache, and KP Cache and delivers the best performance.

Varying value size. We try different value sizes ranging from 50B to 200B (Fig. 10). As the value size increases, the performance of all caching schemes decreases. This is because larger value size incurs more storage I/O overhead per each key being read. `pure-kv` performs better than `pure-kp` at small value but is exceeded by `pure-kp` as the value size increases. This is because the total number of keys can be cached for `pure-kv` decreases as the value size increases, hence the performance of `pure-kv` becomes not as good as that of `pure-kp`. `rocksdb` still performs better than `pure-kv` and `pure-kp`, as half of the requests are range queries that cannot be served by the KV or KP Caches. `ac-key` performs close to `offline`, and constantly better than `rocksdb` (by 5.9%~22.4%), `pure-kv` (by 41.4%~83.6%), and `pure-kp` (by 44.5%~96.0%) because of the use of the hierarchical adaptive

caching in adjusting the size of each component.

CPU Utilization Comparison. With the default settings (§5.1), we record the CPU utilization by the Linux native `time` command and plot them in Fig. 11. The QPS is also plotted to better illustrate the trade-off. We can see that `rocksdb` consumes less CPU resource than the other schemes (`ac-key`, `pure-kv`, `pure-kp`, and `offline`) because they all use Point Cache and thus need to calculate the hash value of *every* key inserted to check if they were cached. We can also observe that although `ac-key` has extra operations to adapt the size of caching components, the CPU utilization is not increased significantly comparing with `pure-kv`, `pure-kp`, and `offline`. Besides, `ac-key` performs close to `offline` and is better than `rocksdb`, `pure-kv`, and `pure-kp`.

5.3 Adaptive Adjustment in AC-Key

To verify the adaptive adjustment process of AC-Key, we construct a workload with two phases: 1 million range queries with random starting key, then 2 million random point lookups. We plot the size of the caching components – Block, KV, and KP Caches – to show the direct evidence of the adaptation process (Fig. 12 top figure). Besides, we also compare the real-time QPS of `ac-key` and `offline` (Fig. 12 bottom figure).

We can see from the top figure of Fig. 12 that during the first 1 million queries, the sizes of the KV and KP Cache are reduced to zero to maximize the Block Cache since they cannot serve range queries. When the workload changes from pure range query to pure point lookup after the first 1 million queries, the Block Cache shrinks sharply, and the KV and KP Cache start to grow. This is because Point Cache (KV and KP Cache) are more space-efficient in caching point lookups, so the HAC algorithm adjusts in favor of the Point Cache (including both the KV and KP Caches). The KP Cache grows faster than the KV Cache at the beginning because every entry is first cached in the KP Cache, and only a small amount of "hot" KVs with a second access will be migrated to the KV Cache. At the beginning of the second phase, both KV and KP Cache grow as they are "stealing" space from the Block Cache. After the size of the Block Cache declines to almost zero, the KV and KP Cache start to compete with each other for space, and it is when KP Cache starts to shrink. The size of the KP and KV Cache finally converges to 19% and 80% of the total cache size. In different workload settings, the Block, KP, and KV Cache will stabilize into different ratio.

`ac-key` is adaptive and can adjust the size of the caching components based on the workload. We also run the `offline` scheme, which tries to find the best fix-sized configuration. However, such one-size-fit-all configuration is not tailored for the special workload of each phase, thus has inferior performance than `ac-key` in each phase: `ac-key` is 32%~66% better in the range query phase, and 2%~20% better in the point lookup phase after convergence (bottom figure in Fig. 12).

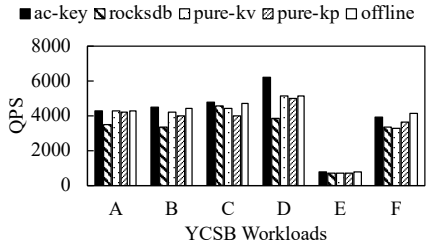


Figure 13: Macro-benchmark YCSB Evaluation.

Table 2: Average Write Latency (μs) and Regression.

Workload	A	B	C	D	E	F
RocksDB	242.8	238.0	N/A	21.7	32.8	28.6
AC-Key	241.1	241.4	N/A	22.7	33.5	26.7
Regression	-0.7%	1.4%	N/A	4.9%	2.0%	-6.8%

5.4 Macro-benchmark YCSB Evaluation

We also use six workloads in YCSB [15] to further verify the performance of our design in near-production workloads. As can be seen in Fig. 13, the overall QPS of `ac-key` is higher than `rocksdb` (by 3.6%~59.9%), `pure-kv` (by 0.1%~20.7%), and `pure-kp` (by 1.2%~25.2%).

We also measure the regression of write performance of AC-Key from RocksDB (Table 2) and find the regression is below 5%. This shows that `ac-key` does not incur significant overhead on the write operations while improving the overall performance.

5.5 Sensitivity on Parameters

Using the default setting in §5.1, we range the learning rate k in the adaptive algorithm (see §4.3) from 1K to 10M, i.e. (1K, 10K, 100K, 1M, 10M). The fluctuation of the QPS is within 3.9%, ranging from 1461 to 1520 operation/s. We also test the ghost cache turnoff threshold θ (§4.3.3) from 0%, 5% , $\cdots$, 30%, and the QPS stays nearly constant (1418~1459, varying within 2.8%). As the two parameters do not have a significant impact on the result, we set their default values as $k = 100K$ and $\theta = 5\%$.

6 Conclusion

Caching is one of the essential techniques to improve the read performance of LSM-tree-based key-value stores. We investigate three different types of entries being cached, namely block, KV, and KP, and incorporate them into one integrated cache. We propose a Hierarchical Adaptive Caching (HAC) scheme to dynamically adjust the size of the block, KV, and KP caching components. To deal with the heterogeneous costs and benefits of the cached entries, we leverage a novel caching efficiency factor to aid the size adjustment among caching components and the eviction decisions within each caching component. We implement the proposed AC-Key by modifying RocksDB. Evaluations show that AC-Key improves the read performance of default RocksDB by up to 57.1% without significant impact on write performance.

Acknowledgments

We thank the anonymous ATC reviewers and our anonymous shepherds for their feedback. This work was partially supported by NSF I/UCRC Center Research in Intelligent Storage and the following NSF awards 1439622, 1525617, and 1812537.

References

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[3] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, volume 10, pages 1–8, 2010.

[4] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.

[5] Google. Leveldb. https://leveldb.org/.

[6] Facebook. Rocksdb. https://rocksdb.org/.

[7] Apache. Cassandra. http://cassandra.apache.org/.

[8] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[9] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, 2016. USENIX Association.

[10] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 537–550, 2016.

[11] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. Elasticbf: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 739–752, 2019.

[12] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[13] John Liang, James Luo, Mark Drayton, Rajesh Nishtala, Richard Liu, Nick Hammer, Jason Taylor, and Bill Jia. Storage and performance optimization of long tail key access in a social network. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, pages 1–6. ACM, 2013.

[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[15] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[16] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. Evendb: optimizing key-value storage for spatial locality. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[17] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[18] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. zexpander: a key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 14. ACM, 2016.

[19] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Yanfeng Zhang, Siyuan Ma, and Xiaodong Zhang. A low-cost disk solution enabling lsm-tree to achieve high performance for mixed read/write workloads. *ACM Transactions on Storage (TOS)*, 14(2):15, 2018.

[20] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[21] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[22] Sorav Bansal and Dharmendra S Modha. Car: Clock with adaptive replacement. In *FAST*, volume 4, pages 187–200, 2004.

[23] Ziqi Fan, David HC Du, and Doug Voigt. H-arc: A non-volatile memory based cache policy for solid state drives. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2014.

[24] George Karakostas and Dimitrios N Serpanos. Exploitation of different types of locality for web caches. In *Proceedings ISCC 2002 Seventh International Symposium on Computers and Communications*, pages 207–212. IEEE, 2002.

[25] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[26] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Edward A. Fox, and Stephen Williams. Removal policies in network caches for world-wide web documents. page 293–305, 1996.

[27] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large cams for high performance data-intensive networked systems. In *NSDI*, volume 10, pages 29–29, 2010.

[28] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13. ACM, 2011.

[29] MySQL 8.0 Reference Manual. The innodb storage engine. https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html.

[30] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, volume 12, pages 193–206, 1997.

[31] Jaeheon Jeong and Michel Dubois. Cost-sensitive cache replacement algorithms. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 327–337. IEEE, 2003.

[32] Facebook. Rocksdb – benchmarking tools. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[33] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196. ACM, 2013.

[34] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf: Practical range query filtering with fast succinct stries. In *Proceedings of the 2018 International Conference on Management of Data*, pages 323–336. ACM, 2018.

[35] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

[36] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514. ACM, 2017.

[37] Jonathan Corbet. Page-based direct i/o. https://lwn.net/Articles/348719/, 2009.

[38] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 79–94. ACM, 2017.

[39] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, pages 505–520. ACM, 2018.