SAS-Cache: A Semantic-Aware Secondary Cache for LSM-based Key-Value Stores

Zhang Cao[1], Chang Guo[1], Ziyuan Lv[1], Anand Ananthabhotla, Zhichao Cao[1]
[1]Arizona State University

***Abstract*—LSM-based key-value stores (LSM-KV stores) are widely used in today's IT infrastructure to store unstructured data. Existing LSM-KV stores use various storage backends including HDD, SSD, and cloud storage, and rely on the in-memory block cache to improve the read performance. Due to the memory space limitations and high costs, the high-performance flash-based secondary cache was proposed as a complementary cache tier for block cache to extend the cache space with high cost-effectiveness. However, existing secondary cache designs for LSM-KV stores do not consider the LSM-specific characteristics including ignoring the secondary cache operation overhead, inserting the invalid blocks into the secondary cache, and cache block invalidation caused by compaction, which leads to low cache efficiency and even performance regression.**

To address the aforementioned critical issues of existing secondary cache designs in LSM-KV stores, we first conduct a comprehensive analysis of the design tradeoffs and limitations of existing secondary cache. Based on the insights and observations, we propose a Semantic-Aware Secondary Cache (SAS-Cache) for LSM-KV stores, which consists of three novel optimizations. First, to effectively reduce the number of unnecessary secondary cache lookups (i.e., lookups with a high probability of being missed), we propose Dynamic Adaptive Secondary Cache Bypass to quickly identify cache items that are most likely not in the secondary cache and bypass the corresponding lookups. Second, we design a Valid SST-Aware Insertion Control to prevent the invalid blocks from being inserted into the secondary cache to achieve a higher cache space utilization and longer flash lifespan. Third, we propose the Compaction-Aware Cache Replacement, which actively removes the cached blocks invalidated by compactions. We implemented the prototype of SAS-Cache based on RocksDB and CacheLib. Our evaluation shows that SAS-Cache can achieve about 5% of the overall cache hit ratio increase, 36% of the throughput improvement, and 20% of latency reduction compared with the state-of-the-art secondary cache design for LSM-KV stores. Specifically, SAS-Cache improves the secondary cache hit ratio by 40% and effectively eliminates the insertion of invalid blocks.

***Index Terms*—LSM-KV stores, Flash-based Cache, Secondary Cache.**

I. Introduction

Log-**S**tructured **M**erge-tree based **K**ey-**V**alue **stores** (**LSM-KV stores**) are widely used for their high write performance, such as LevelDB [1], HBase [2], [3], ZippyDB [4], X-Engine [5], and RocksDB [6]–[9]. In the LSM-KV store, key-value pairs (KV-pairs) are first cached in the Memtable to achieve high write performance. When Memtable is full, KV-pairs are packed into blocks and appended to the Static Sorted Table file (SST file) in the storage. One SST file is a self-contained searchable B-tree structure and KV-pairs are organized as fixed-size data blocks (e.g., 32 KB) and other metadata blocks (e.g., indexing blocks, filter blocks, and file footer). SST files are organized into different levels on the storage backend and they do not have key-range overlap in the same level (excluding Level-0). To serve a read query, the LSM-KV store may require reading multiple SST files from a higher level (e.g., L0) to the lower levels (e.g., bottom-most level) until a certain KV-pair is found. Therefore, the read performance is always a big concern of LSM-KV stores due to the SST file format and level-base designs [10]. Existing LSM-KV stores rely on the DRAM-based ***block cache*** to reduce the number of storage reads by caching frequently accessed meta blocks and data blocks. To guarantee high read and write performance, LSM-KV stores usually use directly attached SSDs as the storage backend.

In recent years, there has been a growing trend of deploying LSM-KV stores on alternative storage backends, such as cloud storage, Hard Disk Drives (HDDs), and zone-based storage devices like shingled magnetic recording (SMR) [11], [11]–[13], to achieve higher scalability and cost-effectiveness. For example, ZippyDB (a distributed LSM-KV store) is deployed in Meta private cloud storage Tectonic [14], [15]. Also, RocksDB (one of the most widely used LSM-KV store engines) is also widely deployed on the public cloud or on HDD to store the SST files for lower storage costs [16], [17]. However, these slower storage systems usually have lower bandwidth and much higher latency compared with the directly attached PCIe-based SSDs. Furthermore, the I/O performance is relatively unpredictable and unstable. These can further cause explicit read performance penalties as indicated in [15].

To effectively address the read performance regression caused by the low-performance storage backend, a secondary cache was proposed for the LSM-KV store as a cost-effectiveness caching extension of existing DRAM-based block cache [15], [18]. The secondary cache is deployed on the directly attached flash-based storage (e.g., PCIe-based SSDs) with a much larger capacity than the block cache and higher access speed than the storage backend, which bridges the performance gap between DRAM-based block caches and slow storage backend. In the existing secondary cache designs [15], [18], all the blocks being evicted from the block cache are inserted into the flash-based secondary cache. When the LSM-KV store has a block cache lookup miss, it will search the block in the secondary cache. If it is a secondary cache hit, a certain block in the secondary cache will be promoted back to the block cache. If it is a secondary cache miss, the LSM-KV store will read the corresponding block from the corresponding SST file at the storage backend and insert the block into

the block cache. We can have different secondary cache implementations with different cache management policies.

Currently, Meta [15] has implemented the secondary cache for RocksDB using CacheLib [19]. We call this version of secondary cache the **De**fault **S**econdary **Cache** (called ***De-SCache***). By comprehensively analyzing the characteristics of De-SCache, we observed that De-SCache can cause explicit read performance regression due to the ignorance of secondary cache lookup and insertion overhead and inserting invalid blocks to the secondary cache. De-SCache only benefits LSM-KV stores in limited workloads, cache settings, and storage backends. Based on our analysis, we identify three root causes that result in the inefficiency of De-SCache.

First, the cache lookup operations to De-SCache are non-selective. In current De-SCache designs, all block cache lookup misses will trigger De-SCache lookups. Due to the high lookup overhead of De-SCache, if there are a large number of misses in De-SCache, it can explicitly impact the overall performance. **Second**, there exists a large number of invalid blocks being inserted into De-SCache. In De-SCache, all evicted blocks from the block cache will be inserted into the secondary cache without identifying whether the block is still valid or invalid. Inserting invalid blocks will result in two issues: 1) The block eviction and insertion are applied by the LSM-KV store foreground read thread. Due to the relatively higher insertion latency of the secondary cache, inserting these invalid blocks to the secondary cache will lead to extra performance overhead; And 2) it will lead to a lower secondary cache efficiency and more unnecessary writes on the flash. **Third**, since the capacity of De-SCache is explicitly larger than the block cache, a large number of blocks can become invalid during their lifetime in the De-SCache caused by compaction. Most of the cache designs and caching policies do not have the application semantics to know if the cached blocks are still valid or not, and they rely on the cache eviction policy to passively evict the blocks. The secondary cache has a large capacity and its cache operations are less frequent. Therefore, the average lifetime of cached blocks can be much longer, and there will be a large portion of cold but still valid blocks. Those cold but valid blocks might be evicted earlier than the invalid blocks and can explicitly lower the cache hit ratio.

To address the aforementioned three fundamental issues of existing secondary cache designs, we propose the **S**emantic **A**ware **S**econdary **Cache** (called ***SAS-Cache***). It is a secondary cache framework optimization for LSM-KV stores. SAS-Cache is a flash-based cache and functions as an extension tier of the memory-based block cache. Importantly, different from De-SCache, SAS-Cache can selectively apply lookups and insertions based on the LSM-KV store-specific semantics to achieve explicitly higher performance, higher cache hit ratio, and better flash lifespan. However, designing such a SAS-Cache will be challenging:

- First, how to identify and avoid unnecessary secondary lookups with low operational overhead is difficult. Tracking every block in the secondary cache (e.g., using a BloomFilter) and achieving dynamic tracking synchronization as the secondary cache insertion/eviction can lead to high memory cost and long latency.
- Second, we need a strategy to prevent invalid blocks from being inserted into the secondary cache. However, it is difficult to know the status of the blocks with low memory and operational overhead since blocks are dynamically inserted, evicted, and invalidated during their lifetime in the block cache.
- Third, considering the large capacity of the secondary cache and the randomness of blocks invaliding by compaction, it is challenging to identify and handle the invalid blocks in the secondary cache. On one hand, the secondary cache is not able to know if a cached block is valid or not. On the other hand, scanning the cache and removing/replacing the invalid blocks can introduce extra overhead.

To address the aforementioned three challenges, we propose the following three novel designs to achieve a highly efficient, workload-adaptive, and flash-friendly secondary cache: 1) **Dynamic Adaptive Secondary Cache Bypass** is used to quickly filter out the unnecessary lookup operations on the secondary cache by using the LSM-based internal information, which achieves a much higher secondary cache hit ratio. We propose the LSM-Managed Cache Filter, which utilizes two synchronization methods: a lightweight cache filter synchronization for regular cache insertion/deletion and asynchronous reconstruction to synchronize different types of cache items in the cache filter with the secondary cache, minimizing the synchronization overhead; 2) **Valid SST-Aware Insertion Control** is used to identify and prevent invalid blocks from being inserted into the secondary cache. We propose to leverage the cache-key composition to quickly identify the invalid blocks. We store the file number of invalid SST files (i.e., deleted by compaction and no longer referenced by snapshots) instead of recording the cache-keys of all the blocks, which significantly reduces the number of records that need to be tracked in memory. Furthermore, we propose using a FIFO queue to track the file number of invalid SST files, minimizing the management overhead; And 3) **Compaction-Aware Cache Replacement** is used to solve the issues caused by invalidated blocks in the secondary cache. SAS-Cache asynchronously scans the secondary cache after a number of compactions and removes invalid blocks by comparing their cache-keys with the invalid SST file number. To further improve the cache hit ratio, we propose the block prefetching methods to insert some of the blocks at the hot key-ranges of the newly generated SST files into the secondary cache.

We implemented SAS-Cache based on RocksDB [6] and CacheLib [19] and open-sourced at [20]. We evaluated our prototype with db_bench in various workloads and different storage backends. Compared with De-SCache, SAS-Cache effectively improves the secondary cache hit ratio by up to 40%. Additionally, it boosts throughput by up to 36% and reduces latency by up to 20%. We also conducted a detailed breakdown analysis of each major optimization design: 1)

the LSM-managed cache filter significantly improves the secondary cache hit ratio, achieving an improvement of up to 37%. It also enhances throughput by up to 15% and reduces latency by up to 8%; 2) Valid SST-aware insertion control plays a vital role in avoiding the insertion of invalid blocks and leads to a 5% improvement in the secondary cache hit ratio. Moreover, it enhances throughput by approximately 8% and reduces latency by around 2.5%; And 3) compaction-aware cache replacement enhances the secondary cache hit ratio by up to 14% and boosts throughput by about 17%, while reducing latency by approximately 5%.

II. Background

A. LSM-KV Store Preliminary

LSM-KV stores are extensively used in today's IT infrastructure due to their high write performance. Notable examples include LevelDB [1], HBase [2], ZippyDB [4], X-Engine [5] and RocksDB [6]. The generic architecture of an LSM-KV store is shown in Figure 1, which contains two key components: a memory-resident segment and a disk-resident segment. New KV-pairs are inserted into active Memtables located in the main memory. When an active Memtable reaches its capacity, it is transformed into an immutable Memtable and scheduled to be flushed to the storage system as a Static Sorted Table (SST) file. SST files are organized into multiple levels and each level (excluding Level-0) maintains a number of SST files with non-overlapped key-ranges. Compaction is applied to merge one SST file at Level-i with multiple SST files at Level-$(i+1)$ into new SST files at Level-$(i+1)$ and remove records marked for deletion or updated.

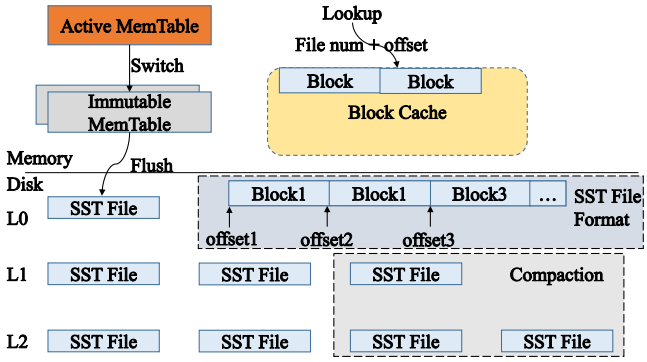


Fig. 1. Architecture of LSM-KV Stores and SST File Format

The aforementioned write flow is optimized for high write throughput but comes at the expense of read performance. On one hand, a certain KV-pair may exist in multiple Memtables and in multiple SST files at different levels. LSM-KV store needs to search from a higher level (e.g., L0) to the lower levels (e.g., bottom-most level) until a certain KV-pair is found (or confirm the KV-pair does not exist). On the other hand, to locate a KV-pair in an SST file, multiple blocks including index blocks, filter blocks, and data blocks are needed for the in-memory binary search. Therefore, the read performance is a big concern of LSM-KV stores [10]. Existing LSM-KV stores rely on the DRAM-based block cache by caching frequently accessed blocks in memory to reduce the reads from the underlying storage system. *Block Cache* is a key component of LSM-KV stores to improve the performance of different types of read queries, such as Get, Multi-Get, and Range queries.

The SST file format is also shown in Figure 1. Since compaction deletes the old SST files, the blocks belonging to these deleted SST files are no longer valid and will not be accessed anymore if there is no snapshots pinpointing the blocks. These invalid blocks in the block cache will be passively evicted by the cache policies (e.g., LRU). Each block is uniquely identified in the block cache by a combination of (*db-unique id, SST file id, block offset*), which is also known as the ***cache-key***.

B. Storage Backend for LSM-KV Stores

LSM-KV stores are widely deployed on SSDs due to their low latency and high throughput. Some LSM-KV stores, like RocksDB, are well-optimized for SSDs [10]. However, there is a growing trend of deploying LSM-KV stores on alternative storage backends, such as cloud storage and HDDs (Hard Disk Drives), to achieve higher scalability and better cost-effectiveness. Cloud storage services or disaggregated storage like AWS S3 [21], Microsoft Azure Blob Storage [22], and private cloud storage services like Tectonic at Meta [14] have become increasingly popular due to their high cost-effectiveness and better reliability. However, accessing data from cloud storage typically involves higher latency and less predictable performance compared to directly attached PCIe SSDs [23]. HDD is advantageous for cost-effective high-capacity storage, making it suitable for scenarios with large datasets or long-term data persistent storage demands. However, HDDs have much worse random access performance, explicitly lower throughput, and limited I/O operations compared with SSDs, which can hinder overall performance, particularly for applications demanding low-latency, high throughput. Therefore, when LSM-KV stores are deployed on those slower storage backends, they face serious performance challenges, especially for read queries.

C. Existing Secondary Cache Designs

To bridge the performance gap between DRAM-based block caches and slow storage backend (e.g., cloud storage or HDD), the secondary cache was proposed [15] as an extension of the DRAM-based block cache for LSM-KV stores. The secondary cache is designed based on the high-performance SSD or Optane Memory to achieve significantly larger capacity than the block cache and much higher access performance than the slow storage backend. De-SCache, the current state-of-the-art secondary cache for LSM-KV stores, is proposed by Meta [15] and applied in RocksDB production [6]. Figure 2 shows the basic workflow of the LSM-KV store with De-SCache.

Typically, there are multiple LSM-KV instances launched on the server, and the local high-performance SSDs serve as the De-SCache. The LSM-KV instances in the same process share the same block cache and De-SCache via a shared pointer. When the LSM-KV store receives read queries, it will look up the block cache first for a certain block (e.g., a metadata block or a data block). If it is a block cache miss,

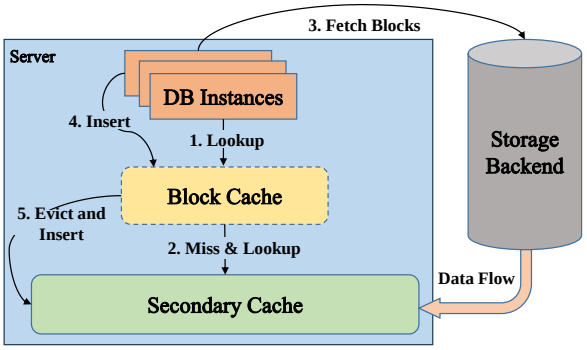


Fig. 2. Workflow of LSM-KV Stores with Secondary Cache.

the LSM-KV store will search the same block in De-SCache. If the target block cannot be found in the secondary cache, the LSM-KV store instance will read the corresponding block from the SST file at the storage system and then insert this block into the block cache directly to search inside the block. When a block is evicted from the block cache, it will be inserted into the De-SCache. Additionally, if the LSM-KV store finds the block in De-SCache (i.e., a secondary cache hit), the block will be promoted back into the block cache for future expedited retrieval. It is a typical tiered cache design and has been explored in other scenarios [19], [24].

III. Motivations

A. Secondary Cache Performance Analysis

1) Operations Characteristics of De-SCache.

To comprehensively analyze the characteristics of De-SCache, we use RocksDB to evaluate the average latency of flash-based secondary cache lookup/insertion, and storage I/Os on different types of storage systems. The results are presented in Table I and we have the following observations: The storage read latency applied by LSM-KV stores on different types of storage devices/systems is about 12-95 times that of flash-based secondary cache. Therefore, the overhead of De-SCache lookup cannot be easily ignored. Similar to the lookup, we observe a non-significant difference between storage insertion and secondary cache insertion (less than 77 times the write latency), indicating that the insertion overhead of De-SCache should be considered in the design. Thus, ***The lookup and insertion overhead of De-SCache cannot be ignored when compared with the storage I/Os.***

TABLE I
Latency of Each Components in LSM_KV Stores

Type	Read Latency (us)	Write Latency (us)
Secondary Cache	67.70	158.75
Storage(NVMe SSD)	70.50	92.10
Storage(HDFS)	606.75	651.32
Storage(HDD)	6,580.60	7012.40

2) Performance Tradeoffs Analysis of De-SCache.

Considering the unignorable lookup and insertion overhead, we provide the following theoretical estimations for the tradeoffs in the De-SCache design. For simplification, our analysis is on a one-thread scenario, but it can also show the performance tradeoffs. For the secondary cache, it has a lookup latency of T_{sec}. Reading a block from the storage system has the latency of T_{stg}. For LSM-KV stores with block cache, its storage access account is N_{stg}^{nsc} (*nsc* is an abbreviation for *n*o *s*econdary *c*ache) and the total latency of responding queries is T_{total}^{nsc}. The total latency here represents the sum of the latency for all lookups and cache insertions. For LSM-KV stores with secondary cache, its secondary cache access account is N_{sec} and its storage read account is N_{stg}^{dsc} (*dsc* is an abbreviation for *d*efault *s*econdary *c*ache) and the total latency of responding queries is T_{total}^{dsc}. We assume that two LSM-KV stores run with the same workload, configurations, and hardware (same storage access latency and block cache access latency) to ensure that we have the same number of block cache hits and misses. Thus, the total latency of all the block cache hits in the two scenarios is the same, we donate it as T_{hit}, and the total latency of all the block cache insertion in the two scenarios is the same, we donate it as T_{bins}. We also can get: $N_{sec} = N_{stg}^{nsc} = block\ cache\ miss\ numbers$. We donate the insertion overhead introduced by the secondary cache as T_{sins}. Then, we can get:

$$T_{total}^{nsc} = N_{stg}^{nsc} * T_{stg} + T_{hit} + T_{bins} \quad (1)$$

$$T_{total}^{dsc} = N_{stg}^{dsc} * T_{stg} + N_{sec} * T_{sec} + T_{hit} + T_{bins} + T_{sins} \quad (2)$$

The latency gap between T_{total}^{nsc} and T_{total}^{dsc} is ΔT. $\Delta T = T_{total}^{nsc} - T_{total}^{dsc}$. Further, we can get:

$$\Delta T = N_{stg}^{nsc} * T_{stg} - N_{stg}^{dsc} * T_{stg} - N_{sec} * T_{sec} - T_{sins} \quad (3)$$

We then replace N_{stg}^{nsc} with N_{sec}, and get:

$$\Delta T = N_{sec} * T_{stg} - N_{stg}^{dsc} * T_{stg} - N_{sec} * T_{sec} - T_{sins} \quad (4)$$

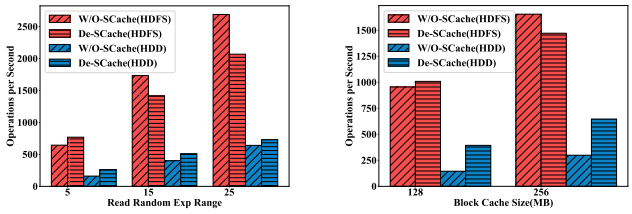
$$= (N_{sec} - N_{stg}^{dsc}) * T_{stg} - N_{sec} * T_{sec} - T_{sins} \quad (5)$$

The first part of the formula 5, $(N_{sec} - N_{stg}^{dsc}) * T_{stg}$, represents the benefits introduced by the secondary cache. The second part, $N_{sec} * T_{sec}$, signifies the lookup overhead of the secondary cache, and the third part T_{sins} indicates the insertion overhead of the secondary cache. If $\Delta T > 0$, it indicates that we can improve the overall performance with a secondary cache. Conversely, if $\Delta T < 0$, the using secondary cache even causes performance regression.

3) Performance Evaluations of De-SCache.

We conducted experiments to validate our aforementioned analysis. We evaluate De-SCache under a read-only and write-intensive workload and measure the overall performance. To be precise, for the read-only workload, we conducted experiments by running De-SCache integrated RocksDB with different workload skewness on different storage backends (HDFS and HDD). Here, different storage backends are used to verify how the secondary cache works when its overhead can cause varying levels of performance degradation to LSM-KV stores. For HDD, the overhead brought by the secondary cache is relatively smaller due to the large latency gap between the secondary cache and HDD. However, for HDFS, the overhead brought by the secondary cache is relatively larger due to the

small latency gap between the secondary cache and HDFS. To control the data skewness, we selected the ReadRandom workload from db_bench [25] and followed the default settings described in Section V-B. Initially, we used the fillrandom workload to insert 1 million key-value pairs, and subsequently, we employed the readrandom workload to retrieve 1 million keys. We also configured RocksDB with the same block cache size but without a secondary cache (called **W/O-SCache**) as the baseline. For the write-intensive workload, we run the default workload while maintaining the settings described in Section V-B, with different block cache sizes. The results are presented in Figure 3.



(a) Read-Only Workload (b) Write-Intensive Workload

Fig. 3. Overall Throughput on HDFS and HDD

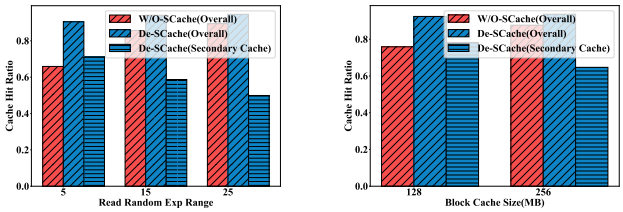
In Figure 3(a), as the Exp Range increases (i.e., the workload is more skewed), the performance gap (the performance of De-SCache minus W/O-SCache) between De-SCache and W/O-SCache becomes smaller (from positive to negative), which is the same for both HDFS and HDD. Furthermore, due to the limited latency gap between the secondary cache and HDFS which amplifies the overhead introduced by the secondary cache. We observe that when the Exp Range is set to 15 and 25, the throughput of the De-SCache is even lower than that of the W/O-SCache (i.e., $N_{sec} * T_{sec}$ and T_{sins} dominate the total latency). As shown in Figure 3(b), with HDD, the performance of De-SCache is higher than W/O-SCache no matter which block cache size we use. However, with HDFS, the performance of De-SCache is higher than W/O-SCache when the block cache size is 128 MB and lower than W/O-SCache when the block cache size is 256 MB. This is because when the block cache size is smaller than 128MB, the block cache is not enough to contain all the working set (or hot) blocks and RocksDB can get benefits from the secondary cache. However, with a 256 MB size of the block cache, it can hold most of the working set items (i.e., almost no block cache misses), and inserting evicted blocks to the block cache only introduced extra overhead.

From the above analysis and experiments, we can conclude that ***the current state-of-the-art secondary cache design (i.e., De-SCache) can cause explicit read performance regression due to the ignorance of lookup and insertion overhead and only benefits LSM-KV stores in limited workloads, cache settings, and storage backends.***

B. Why De-SCache is Inefficient?

1) Non-Selective Lookups into De-SCache.

In current De-SCache designs, all block cache lookup misses will trigger De-SCache lookup. Due to the high lookup overhead of De-SCache, if there are a large number of misses in De-SCache, it can explicitly impact the overall performance. We use **Lookup Hit Ratio (LHR)** to measure the lookup efficiency of De-SCache. It is defined as the cache lookup hit number divided by the total lookup number, commonly referred to as the hit ratio in other studies [19], [26]. A higher LHR usually indicates better overall performance. We collect the statistics of the overall hit ratio (i.e., including block cache hits and misses) and the secondary cache LHR (i.e., including secondary cache hits and misses) for the read-only and write-intensive workloads used in Section III-A3, and the results are shown in Figure 4. As shown in Figure 4(a), with the range of read random exp range increases (i.e., the workload is more skewed and less random), the overall hit ratio difference between W/O-SCache and De-SCache becomes smaller, corresponding to a decrease in the performance gap as described in Section III-A3. Additionally, the LHR also decreases, indicating that the benefits of De-SCache become smaller and its overhead becomes more significant. This is part of the reason that the performance of De-SCache can be even lower than W/O-SCache, as described in Section III-A3.



(a) Read-Only Workload (b) Write-Intensive Workload

Fig. 4. Overall Hit Ratio and Secondary Cache LHR.

2) Insertion invalid items into De-SCache.

In the De-SCache design, all evicted blocks from the block cache will be inserted into the secondary cache without identifying whether the block is still valid (i.e., the corresponding SST file is maintained by LSM-KV store or pinpointed by snapshots) or invalid (i.e., the corresponding SST file is already deleted by compaction and no longer referenced by snapshots). If we insert invalid blocks to the secondary cache, this will result in two issues: 1) The block eviction and insertion are applied by the LSM-KV store foreground read thread. Due to the relatively higher insertion latency of the secondary cache, inserting these invalid blocks into the secondary cache will lead to extra performance overhead; And 2) it will lead to a lower secondary cache efficiency (i.e., valid blocks in the secondary cache are forced to be evicted) and more unnecessary writes on the flash (i.e., impact the SSD lifespan). Here, we introduce **Invalid Blocks Insertion Ratio (IBIR)** to measure how these invalid blocks being inserted impact the effectiveness of De-SCache. It is defined as the ratio of invalid blocks among all the inserted blocks. If there is a large number of invalid cache items being inserted into the secondary cache (i.e., a high IBIR), it indicates a low efficient cache insertion policy and a potential overall performance regression.

To precisely analyze the existence and impact of inserting invalid blocks into De-SCache, we record the IBIR of RockDB with the write-intensive workload as described in Section

III-A3. To monitor the IBIR, we collected the secondary cache insertion trace and statistics from RocksDB and filtered out the inserted blocks that belong to the invalid SST files, such that we can calculate the overall IBIR. The result is shown in Figure 5. In Figure 5, we can see a significant portion of invalid blocks is indeed inserted into the secondary cache (i.e., even about 60% when the block cache is 512 MB). Furthermore, as the block cache's size increases, the number of invalid blocks and IBIR increases significantly.

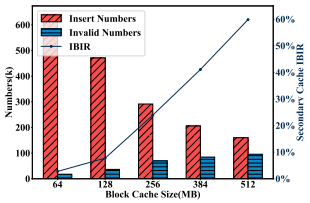


Fig. 5. Secondary Cache IBIR

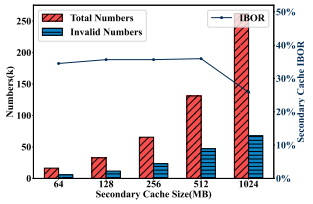


Fig. 6. Secondary Cache IBOR

3) Slow eviction of invalid blocks retained in De-SCache.

Compaction can also directly invalidate the valid blocks in the De-SCache. Most of the cache designs do not have the application semantics to know if the cached blocks are still valid or not, and they rely on the cache eviction policy to passively evict the cold blocks. If the cache capacity is relatively small and has highly frequent lookups/insertions (e.g., the LRU-based block cache in LSM-KV stores), invalid blocks will be quickly moved to the LRU-end and evicted. Invalid blocks will not cause explicit cache efficiency issues. However, if the cache capacity is much larger and the cache operations are less frequent (e.g., the flash-based secondary cache design in LSM-KV stores), the average lifetime of cached blocks can be much longer and there will be a large portion of cold but still valid blocks. Those cold but valid blocks might be evicted earlier than the invalid blocks and can explicitly lower the cache hit ratio. Here, we introduce **Invalid Blocks Occupancy Ratio (IBOR)** to measure the impact of these invalid blocks retained in De-SCache. It is defined as the invalid block numbers divided by the total block numbers in the secondary cache. It is dynamically changing during the running time. If there is a significant number of invalid blocks, it indicates a low cache space efficiency.

To apply quantitative analysis, we collect the IBOR of RocksDB with De-SCache under the write-intensive workload as described in Section III-A3. To track the invalid items retained in De-SCache, we record all the deleted SST file names (excluding the one pinpointed by snapshots) from each compaction and iterate through the secondary cache to find all the blocks that belong to those deleted SST files. This allows us to calculate the amount of the retained invalid blocks when each compaction occurs. We also record the total number of blocks in the secondary cache to calculate the overall IBOR. As shown in Figure 6, there is a large number of invalid blocks retained in the secondary cache. Specifically, when the secondary cache size is about 256 and 512 MB, the IBOR can reach approximately 30%, which indicates that 30% of the cache space is wasted. Furthermore, we can observe that an increase in cache capacity is directly associated with a higher number of invalid blocks retained in the cache.

Therefore, motivated by the aforementioned analysis, we are aiming to design and implement a novel secondary cache that can explicitly achieve better overall performance for LSM-KV stores with the improvement of LHR, reducing IBIR, and reducing IBOR with low memory and operational overhead and can adapt the secondary cache to more workloads, storage backends, and cache configurations.

IV. Semantic-Aware Secondary Cache

A. Challenges

How to Dynamically Bypass Secondary with Low Overhead. Due to the non-selectivity of the secondary cache lookup, the LHR of the secondary will change as the LSM-KV store workloads fluctuate over time, causing performance regression. To consistently maintain the secondary cache LHR at a high level we need to determine whether it's worth accessing the secondary cache or directly bypassing it (i.e., there is a high probability of a secondary cache miss) to read the block from storage. One simple solution to achieve such a cache selectivity is to track all the cache-keys of blocks in the secondary cache and use a data structure (referred to as a *cache filter*) to store them in memory, checking their existence before issuing lookup queries to the secondary cache.

Kangaroo [27] uses the Bloom filter to serve as the cache filter. In Kangaroo, the flash-based cache is divided into many *sets*, each of which has a size of about 4KB. For each set, Kangaroo maintains a small Bloom filter in DRAM built from all the keys in the set to reduce unnecessary flash reads. Whenever a set is written, the Bloom filter is reconstructed to reflect the set's contents. However, this method poses two problems if we directly apply the filter in our scenario. First, in Kangaroo, the cache size (each *set*) corresponding to each Bloom filter is small. However, in our design, we need to track all the blocks in the cache with the cache filter, making the cost of reconstruction more expensive due to the larger cache size. Second, in Kangaroo, any change to the cache only triggers one of the small Bloom filter reconstructions. However, in our case, it will trigger the entire cache filter construction, which increases the reconstruction frequency significantly.

From the above analysis, the cache filter should possess the following essential attribute: *Support for Efficient Synchronization with Secondary Cache Content*. As the cache filter is used to dynamically track the cache-keys of blocks in the secondary cache, it needs to maintain synchronized information with secondary cache content and the synchronization overhead should be small. In LSM-KV stores, two types of cache items need to be synchronized with the cache filter. The first type is regular cache items. When items are inserted into the secondary cache or evicted from it, the corresponding cache-keys in the cache filter should be updated synchronously. The second type is the invalidated cache items caused by compaction. When an SST file becomes invalid, the blocks of a certain SST file in the secondary cache should also be updated in the cache filter, which can be applied asynchronously.

How to Prevent Invalid Blocks from Being Inserted into Secondary Cache with Low Overhead. To lower the IBIR, we need a strategy to prevent invalid blocks from being inserted into the secondary cache with low operational overhead. One approach is to maintain a list of cache-keys for these invalid blocks in the block cache and use a specific list to track them. When an SST file becomes invalid (i.e., deleted and no longer referenced by snapshots), we can scan the block cache, identify the blocks from a certain SST file, and insert the corresponding cache-key into the list. Before inserting evicted blocks into the secondary cache, we check their cache-keys against this list to determine their validity. If the cache-key is invalid, it is removed from the tracking list without inserting it into the secondary cache. Additionally, this list should be kept in memory to minimize the operational overhead (e.g., insertion, removing, and lookup), making memory efficiency crucial. However, how to efficiently identify the invalid blocks and track their information during the running time is very challenging.

How to Address the Large Number of Invalid Blocks Retained in Secondary Cache. To lower the IBOR, one naive method is to collect all the cache-keys of the invalid blocks during compaction and utilize this information to evict the corresponding blocks from the secondary cache. However, this method poses several problems. First, how to collect the cache-keys of invalid blocks without influencing the compaction process is challenging. Second, we need to store these cache-keys with minimal memory usage, which can be difficult. Third, since the secondary cache is large and contains millions of blocks, scanning the cache and identifying the cache-keys of invalid blocks is time-consuming and it can influence the cache performance. Additionally, as the invalid blocks are evicted, a large amount of SSD space becomes available. How to utilize this space efficiently and recover the decreased LHR caused by the invalidation needs to be explored.

B. Architecture Overview of SAS-Cache

To address the aforementioned challenges, we propose a **S**emantic **A**ware **S**econdary **SAS-Cache**, which is flash-based and functions as a second-tier cache alongside the memory-based block cache. SAS-Cache is a secondary cache optimization framework for the LSM-KV store and is independent of the implementation of the secondary cache. That is, all existing cache policies (e.g., LRU, FIFO, or Clock) or flash-based cache implementations (e.g., CacheLib [19] or Flashield [28]) can be directly integrated with SAS-Cache. We propose three important components in SAS-Cache to effectively address the aforementioned challenges: **1) Dynamic Adaptive Secondary Cache Bypass**, abbreviated as Adaptive Bypass, is designed to adaptively bypass the secondary cache lookups that have a high probability of cache misses before we issue the lookup queries to the secondary cache. We propose an LSM-Managed Cache Filter (abbreviated as Cache Filter) that possesses fast lookup operations with high memory efficiency, and it can sync with the cache content with low operational overhead. **2) Valid SST-Aware Insertion Control**, abbreviated as Insertion Control, efficiently identifies invalid blocks evicted from the block cache and prevents their insertion into the secondary cache. It uses an Insertion Control Queue and a novel cache-key composition to effectively track the invalid block. **3) Compaction-Aware Replacement**, abbreviated as Compaction Replacement, leverages low-overhead runtime compaction information from LSM-KV stores and applies compaction replacement techniques to replace the invalid blocks with newly generated blocks (i.e., evict invalid blocks, prefetch and insert new valid blocks) in the secondary cache.

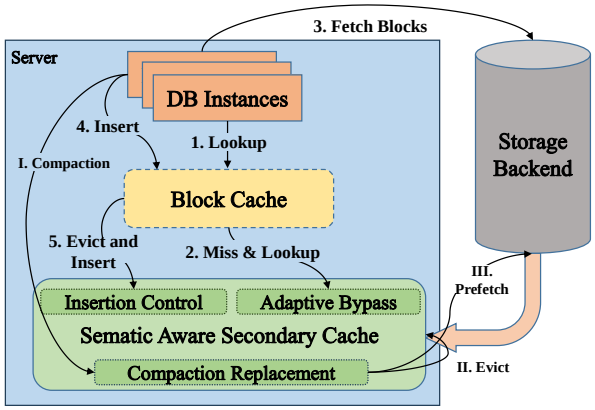


Fig. 7. Architecture of SAS-Cache.

The overall architecture of SAS-Cache is shown in Figure 7. When an LSM-KV instance receives a read query request (e.g., Get, Multi-Get, or Scan) and fails to find the request block in the block cache, it will first check the cache-key of a certain block in the Cache Filter. If the cache-key is found in the Cache Filter, it will access the secondary cache. Otherwise, it will bypass the secondary cache, directly retrieve the block from storage, and insert the block into the block cache (improve LHR). When the block cache becomes full, blocks will be evicted from the block cache. Those evicted blocks will be inspected by the Insertion Control Queue to determine if they are invalid or not. The valid blocks will be inserted into the secondary cache, while the invalid ones will be discarded (reduce IBIR and IBOR). For each compaction execution, the LSM-KV store instance gathers information of deleted SST files and newly generated SST files. SAS-Cache utilizes such compaction information to execute Compaction Replacement, aiming to mitigate the impact of compaction on the SAS-Cache LHR. This replacement process includes evicting invalid blocks and prefetching some of the useful blocks in the newly generated SST files (reduce IBOR).

C. Dynamic Adaptive Secondary Cache Bypass

To efficiently bypass the secondary cache when a block cache miss happens, we propose the Dynamic Adaptive Secondary Cache Bypass, which relies on a key structure: LSM-Managed Cache Filter. As discussed in Section IV-A, the most widely used BloomFilter cannot sync with the cache content changes. The LSM-Managed Cache Filter uses two key designs to efficiently achieve this synchronization: employing a lightweight synchronization method for regular cache items

and asynchronous reconstruction to synchronize invalid cache items. These designs are based on two key observations. First, invalid item synchronization occurs less frequently than regular item synchronization since the frequency of compaction is much smaller than cache insertion/eviction. Second, although the number of invalid items is large, they are generated in a batch fashion during each compaction. Therefore, for regular items, a lightweight synchronization method (direct insertion and deletion) is needed due to its high operation frequency. For invalid items, we employ asynchronous reconstruction instead of directly evicting and inserting the invalid items one by one to mitigate the influence on other cache filter operations, such as lookups and regular item synchronization.

The LSM-Managed Cache Filter is designed based on the cuckoo filter [29], ensuring fast lookup/insertion/deletion operations and high memory efficiency. We leverage the insertion/deletion operations of the cuckoo filter for lightweight synchronization. For regular item synchronization, it employs insertion and deletion (eviction) to synchronize the cache-keys with the secondary cache insertion and eviction. For invalid item synchronization, we use asynchronous reconstruction to construct a new LSM-Managed Cache Filter that does not include the invalid items and replace it with the current cuckoo filter. The workflow of asynchronous reconstruction is shown in Figure 8. Note that, this workflow of asynchronous reconstruction is also closely related to Compaction Replacement presented in Section IV-E. In Compaction Replacement, the Replacement Executor has two operational steps: the first is to evict the invalid items from the secondary cache, and the second is to prefetch new valid items and insert them into the secondary cache. For asynchronous reconstruction, it has two phases: prefill and update phase. In the prefill phase, the reconstruction executor creates an empty cache filter to insert the newly prefetched cache items and synchronize it with the prefetch process. The end of the prefill phase is when the prefetch process ends. When the prefetch process finishes, the eviction process has already finished because the prefetch (insertion) process is usually slower than the eviction process, which is detailed in Section IV-E. Then the reconstruction executor is able to collect the invalid item information from the replacement executor and begins to merge the prefilled cache filter with the original cache filter.

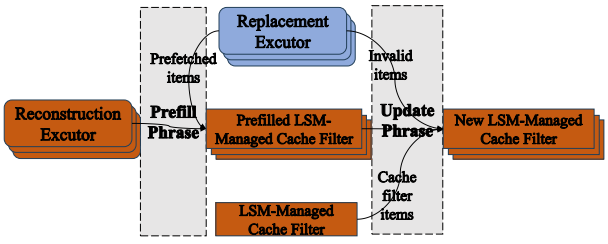


Fig. 8. The workflow of asynchronous reconstruction.

When using the cuckoo filter, there are two important parameters we need to determine: 1) total items in the Cache Filter, which is determined by the size of the secondary cache and also the size of each block. Assuming the total number of cache items is denoted as T, the secondary cache size is S, and the size of each block is B, they are related by the following formula: $T = S/B$; and 2) the size of the fingerprints, which is determined by the desired target false positive rate ϵ. Smaller values of ϵ necessitate longer fingerprints to reject more false queries. In our design, since the LSM-Managed Cache Filter is used in the filtering step to improve the secondary cache hit ratio, we can tolerate a high false positive ratio (e.g., 10%). Therefore, we employ small fingerprints in our evaluations, such as 16 bits. Compared to the original cache-key size (16 bytes), the memory overhead is minimized by about eight times.

D. Valid SST-Aware Insertion Control

In Section IV-A, we highlighted a potential issue: tracking all the invalid blocks in the block cache could lead to high memory consumption due to the considerable number of blocks. To address this issue, we propose Valid SST-Aware Insertion Control, as shown in Figure 9. First, we propose to leverage the cache-key composition to quickly identify the invalid blocks. Since the cache-key is constructed based on the SST file number (it is a unique ID generated by the LSM-KV store), we can identify whether the block evicted from the block cache belongs to the invalid SST files by extracting and comparing the SST file number of the evicted block. Therefore, instead of recording the cache-keys of all the blocks, the Insertion Control stores the file number of invalid SST files. This significantly reduces the number of records that need to be stored in memory. Given that the default block size is 4 KB, while the typical size of SST files is several dozens to hundreds of megabytes (e.g., typically 64 MB in RocksDB), the difference is on the order of tens of thousands.

Although recording invalid file numbers can significantly reduce memory consumption, another problem needs to be addressed: we need to track the invalid SST file numbers in a certain data structure until their corresponding blocks are fully evicted from the block cache. However, since SST files are randomly invalidated by compaction, it is challenging to determine precisely if the blocks belonging to a certain invalid SST file are entirely evicted from the block cache unless we incur additional efforts to scan the block cache. To address this problem, we propose to use a FIFO queue to track the file number of invalid SST files, and the queue is updated during each compaction and snapshot update. The main observation behind this idea is as follows: although it is challenging to determine when blocks belonging to certain invalid SST files are entirely evicted from the block cache, all associated invalid blocks are evicted when the cache iterates through one round because these invalid blocks are no longer accessed. When the cache iterates one round, they will be evicted. Therefore, we can keep the file number in the queue until the cache completes one round of iteration. Additionally, for those SST files invalidated by the same compaction, they are inserted into the FIFO queue at the same time. Thus, we can batch them and evict them all together as the cache iterates through one round.

The size of the queue is pivotal to the effectiveness of the Insertion Control. Two key parameters are involved: the time that invalid file numbers from each compaction stay in the queue (denoted as T_c), and the average stay time of items in the block cache (denoted as T_b). To ensure the Insertion Control works effectively, the following formula needs to hold: $T_c > T_b$. The value of T_b is influenced by the workload and block cache size, while T_c is affected by the frequency of compaction and the queue size. Out of these four parameters, we can only determine the size of the queue. A larger queue size enhances the effectiveness of the Insertion Control, preventing more invalid blocks from entering the secondary cache. Conversely, a smaller queue size may allow some invalid blocks into the secondary cache. In our experiments, when the queue size is set to 10, its effectiveness is significant. More analysis and experiment results of the impact of the queue size are shown in Section V-E2.

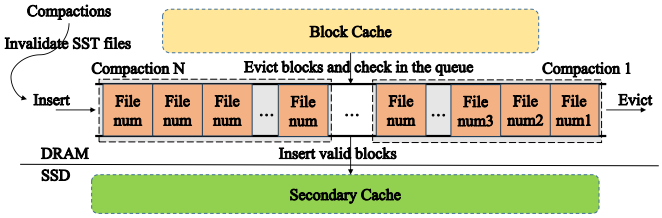


Fig. 9. Workflow of Insertion Control.

E. Compaction-Aware Cache Replacement

In SAS-Cache, we introduce Compaction-Aware Replacement to aggressively remove invalid blocks with low overhead and asynchronously prefetch newly generated blocks to maintain a high LHR. The frequency of the compaction replacement is related to the compaction. In our setting, we employ compaction replacement when each compaction happens. Instead of tracking all block cache-keys, SAS-Cache collects the file numbers of SST files being deleted by compaction and no longer referenced by snapshots, which reduces the influence on the compaction process and also lowers memory usage. The secondary cache is scanned by applying a 10-bit prefix match of cache-keys (hash value of the SST file number) to quickly identify and evict the data blocks belonging to the deleted and non-referenced SST files. Note that this eviction operation is applied asynchronously as a background job to minimize the performance impact. Considering the key-range hotness and KV-pair access localities [7], the KV-pairs in the deleted SST files are reorganized in the blocks of newly generated SST files during compaction and those newly generated blocks might be accessed shortly. It can easily cause a high number of cache misses shortly after compaction. To maintain a high LHR, we propose the multi-level hints-based replacement scheme to identify and prefetch blocks to the secondary cache to mitigate the cache warm-up issues.

Specifically, we utilize the secondary instance of RocksDB [30] to concurrently prefetch the target blocks. Due to the latency of opening a secondary instance and fetching the blocks from storage [31], the prefetch process is usually slower than the eviction process. Moreover, we introduce four levels of hints for prefetching: 1) block type (i.e., data blocks, metadata blocks, and filter blocks), 2) key-range, 3) SST file levels, and 4) SST files. SAS-Cache can use one or combine multiple levels of hints together to precisely identify the blocks to be prefetched. For example, LSM-KV store can specify a key-range with data block type in the first 3 levels (L0 to L2), SAS-Cache prefetches data blocks that overlap with the specified key-range from the corresponding levels. By default, our primary prefetching approach is to prefetch all the blocks of the newly generated files, which is also used in Incremental Warmup Algorithm [32].

F. Discussion

One big limitation of SAS-Cache is: it may perform poorly when LSM-KV stores use a lot of snapshots during reads, which will force most of the blocks belonging to the compacted SST files to be cached. It will explicitly lower the cache efficiency. There are also several limitations that should be considered in the three main optimizations of SAS-Cache. First, The Cache Filter's effectiveness is closely related to the secondary cache hit ratio. Cache Filter is introduced to improve the secondary cache hit ratio by reducing unused secondary cache lookups. However, it still introduces a slight overhead. If the secondary cache hit ratio is already high (e.g., higher than 95% before using the filter), checking the Cache Filter will lead to extra latency. Therefore, the Cache Filter plays a more significant role when the secondary cache hit ratio is relatively low (e.g., lower than 80%). Second, Insertion Control prevents invalid blocks from entering the secondary cache, which brings two improvements: It reduces the insertion overhead and it frees up cache space that was previously occupied by invalid blocks, which can increase the hit ratio of the secondary cache. The effectiveness of the hit ratio improvement is more pronounced when the capacity of the secondary cache is relatively smaller. Third, Compaction Replacement evicts invalid blocks in the secondary cache and also prefetches useful blocks. During the block prefetching, some of the valid blocks might be evicted. Therefore, its effectiveness is related to how useful the prefetched blocks are and the free space of the cache after aggressive evictions. As described in Section IV-E, we prefetch the new-generated blocks. If the cache space is more ample, the negative impact of the evicted blocks will be small and will not influence performance.

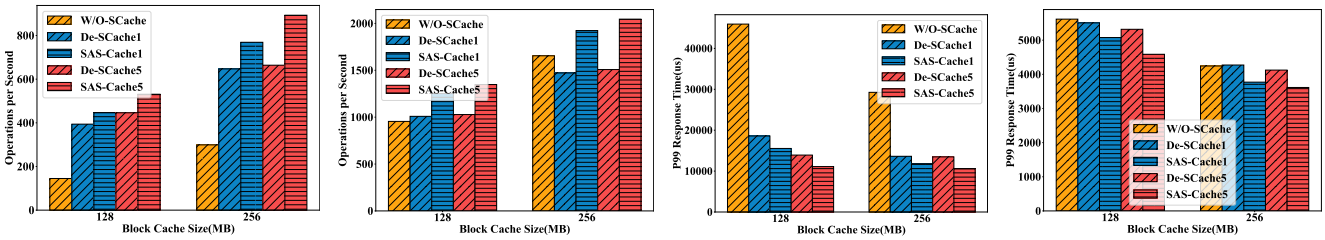
V. Implementation and Evaluations

A. Implementation

We implement the SAS-Cache prototype based on RocksDB [6] version 7.7.3 and CacheLib v2022.10.17.00 [19]. For Cuckoo Filter implementation, we use the open-source code available on Github [33]. The source code of SAS-Cache is available in a Github repository [20].

B. Experimental Setup

We conducted all experiments on Dell EdgePower 650 servers equipped with an Intel(R) Xeon(R) Silver 4,310 CPU, 64 GB of memory, and 480 GB of SSD storage, running Ubuntu Linux release 20.04 LTS. We use both HDD



(a) Overall Throughput on HDD (b) Overall Throughput on HDFS (c) Read Latency(P99) on HDD (d) Read Latency(P99) on HDFS

Fig. 10. Write-Intensive Workload Overall Throughput and Read Latency Comparison.

(TOSHIBA MG06ACA8 [34]) and HDFS as the storage backend for RocksDB. For HDFS deployment, we used a cluster with three machines in the same rack, connected to a 10 Gbps network switch. Each machine was equipped with an Intel(R) Xeon(R) Silver 4,310 CPU, 256 GB of memory, and 7 TB of HDD storage. We configured HDFS with a local cache (each node accessing HDFS will use the local page cache), which is a common configuration [35]. We use Samsung 980 PRO (7,000 MB/s read, 5,100 MB/s write) [36] as the backend for the secondary cache, which has explicitly higher performance than either HDD or HDFS storage backend.

Without explicit explanation, We conducted the native RocksDB benchmark, db_bench, with a ReadRandomWriteRandom workload, which has read operations and write operations that trigger compaction when reading. We used the default setting of ReadRandomWriteRandom, with a read-to-write ratio of 90%. The block cache size used in our experiments is set to 256 MB, with the index blocks caching enabled in the block cache. We set block size as the default 4KB and key size as 48 bytes, value size as 10240 bytes. We also utilize direct I/O for read, flush, and compaction operations. For other RocksDB configurations, we follow the default settings. The backend of SAS-Cache is CacheLib and is configured with 8 MB DRAM and 5 GB flash storage by default. We run the workload under different storage backends (HDD and HDFS). Here, different storage backends are used to verify how the secondary cache works when its overhead can cause varying levels of performance degradation to LSM-KV stores. For HDD, the overhead brought by the secondary cache is relatively smaller due to the large latency gap between the secondary cache and HDD. However, for HDFS, the overhead brought by the secondary cache is relatively larger due to the small latency gap between the secondary cache and HDFS. We compare SAS-Cache to De-SCache and W/O-SCache. For Cache Filter, we set the fingerprint size to 16 bits by default and the estimated false positive ratio is about 10%. The total item number is the same as the block item number in the secondary cache. For the Insertion Control queue size, we set the number to 10, which matches with workloads and cache size settings.

C. Overall Performance Comparison

1) Write-Intensive Workload.

For the write-intensive workload, we use the default settings as detailed in Section V-B. To show that SAS-Cache can perform well under different cache settings, We adjusted the block cache size to 128 MB and 256 MB, while the secondary cache size was set to 5 GB and 1 GB. We compared five configurations: W/O-SCache, De-SCache1 (De-SCache with 1 GB SSD), De-SCache5 (De-SCache with 5GB SSD), SAS-Cache1 (SAS-Cache with 1 GB SSD space), and SAS-Cache5 (SAS-Cache with 5 GB SSD space).

Overall Throughput Analysis. The overall throughput comparison is shown in Figure 10(a) and 10(b). With HDD storage backend, De-SCache1 and De-SCache5 outperform W/O-SCache, achieving up to 3 times of throughput improvement when the block cache size is set to 128 MB and 256 MB. The working set size (WSS) is much larger than the block cache capacity and thus secondary cache avoids most of the storage HDD reads, leading to a significant throughput improvement. SAS-Cache1 and SAS-Cache5 exhibit even higher throughput compared to De-SCache1 and De-SCache5, with around 34% improvement compared to De-SCache and approximately a 3.6 times throughput improvement compared with W/O-SCache. This demonstrates the effectiveness of the optimizations we proposed in SAS-Cache. When using HDFS (HDFS has higher throughput and lower latency than HDD due to the in-parallel access and page cache at data nodes), De-SCache1 and De-SCache5 have even lower throughput than W/O-SCache when the block cache size is set to 256 MB (e.g., secondary cache cause performance regression). In this scenario, due to the relatively large block cache, fewer blocks belonging to the working set are in the secondary cache. The secondary cache miss ratio is explicitly higher, which causes extra read overhead as explained in Section III-A. However, when the block cache size is set to 128 MB, De-SCache outperforms W/O-SCache by approximately 7%. With SAS-Cache, we can achieve about 36% throughput improvement compared to De-SCache and 41% improvement compared to W/O-SCache, indicating the success of secondary cache hit ratio improvement.

Read Latency Analysis. The tail latency of RocksDB read queries is shown in Figure 10(c) and 10(d). When using HDD, the secondary cache significantly reduces latency by more than 50%, primarily due to the substantial performance gap between HDD and NVMe SSD. Compared to De-SCache, SAS-Cache further reduces read latency by about 20%. When using HDFS, the read latency of De-SCache is slightly lower than that of W/O-SCache. Even when the block cache size is set to 256 MB, the read latency of W/O-SCache and De-SCache is nearly identical. Compared with De-SCache,

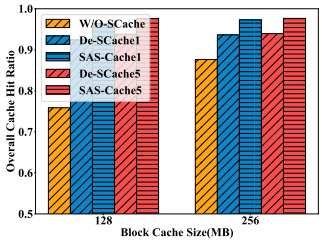


Fig. 11. Overall Cache Hit Ratio

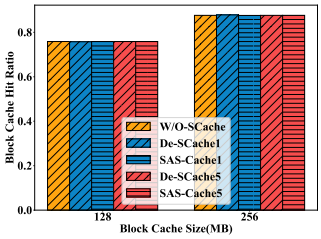


Fig. 12. Block Cache Hit Ratio

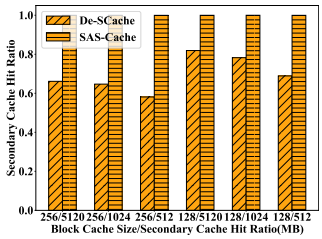


Fig. 13. Secondary Cache Hit Ratio

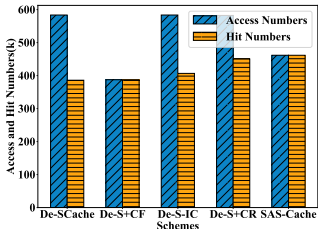


Fig. 14. Secondary Cache Hit and Access Numbers

SAS-Cache reduces read latency, yielding an improvement of approximately 14% when the block cache size is set to 128 MB.

Cache Hit Ratio Analysis. We implemented the cache statistics in RocksDB and SAS-Cache, and collected three types of cache hit ratios: overall, block cache, and secondary cache. The overall cache hit ratio accounts for both the block cache and secondary cache hits and directly affects throughput and read latency. The results of the overall cache hit ratio are shown in Figure 11. RocksDB with De-SCache1 and De-SCache5 show a higher overall cache hit ratio than RocksDB with W/O-SCache, with a more significant improvement when the block cache size is set to 128 MB. Additionally, the overall hit ratio of RocksDB with SAS-Cache1 and SAS-Cache5 is higher than that of RocksDB with De-SCache1 and De-SCache5, with an improvement of approximately 5%. The Block Cache Hit Ratio statistics are shown in Figure 12. The block cache hit ratio remains nearly the same among the five setups, indicating that the three designs of SAS-Cache do not impact the block cache effectiveness. The major performance improvement is the result of avoiding the storage I/Os. Finally, the secondary cache hit ratio is shown in Figure 13. SAS-Cache can always achieve up to 40% of the hit ratio improvement compared with that of De-SCache in all scenarios.

Cost Analysis. The memory cost of SAS-Cache primarily comprises two perspectives: 1) maintaining the cache filter, and 2) the queue of Insertion Control. For Cache Filter, we define its fingerprint size as F bits and its total count as N. Consequently, the total number of blocks in the secondary cache is also denoted as N and the block cache size as B. The ratio of the total cache filter size to the secondary cache size, denoted as σ, is calculated as the formula 6:

$$\sigma = N * F/N * B = F/B \tag{6}$$

In our specific setup, the block cache size is 4K bits, and we have set the fingerprint size to 16 bits, resulting in $N = 257.5$. For instance, when the cache size is 1 GB, the Cache Filter occupies 3.95 MB, and for a cache size of 5 GB, the Cache Filter's size increases to 19.85 MB. Regarding the queue size of Insertion Control, it comprises two member variables: one to record the maximum queue size and the other to maintain the queue itself. In our evaluation, we've configured the queue size to be 10, resulting in a total size of 80 Bytes. Therefore, the memory overhead of the queue for SST file numbers can be ignored.

TABLE II
PERFORMANCE OF READ-ONLY WORKLOAD ON HDFS. **ER** STANDS FOR READ RANDOM EXP RANGE. THE LARGER THE ER, THE MORE SKEWED THE DATA IS. **OHR** STANDS FOR OVERALL HIT RATIO. **LHR** STANDS FOR (SECONDARY CACHE) LOOKUP HIT RATIO.

Scheme	ER	OPS	P99 Latency	OHR	LHR
W/O-SCache		311	9536.66	0.399	-
De-SCache	5	365	8847.31	0.790	**0.546**
SAS-Cache		421	8049.36	0.790	**0.993**
W/O-SCache		517	6420.61	0.732	-
De-SCache	15	536	6204.96	0.896	**0.718**
SAS-Cache		643	5637.44	0.896	**0.997**
W/O-SCache		860	5917.22	0.794	-
De-SCache	25	801	6079.45	0.906	**0.759**
SAS-Cache		965	5446.95	0.906	**0.998**

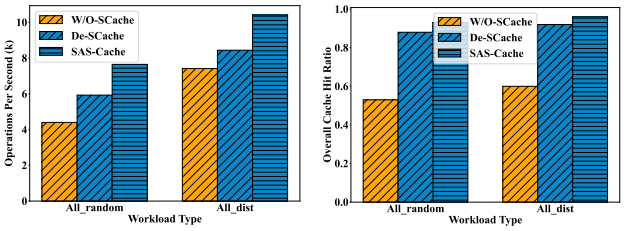
2) Read-Only Workload.

For Read-Only workload, we selected the ReadRandom workload from db_bench [25] and use ReadRandom Exp Range (abbreviated as ER) to control the data skewness. For cache and RocksDB settings, we followed the default configurations described in Section V-B choosing HDFS as the storage backend. Initially, we used the fillrandom workload to insert 10 million key-value pairs, and subsequently, we employed the readrandom workload to retrieve 1 million keys. The result is shown in Table II.

Throughput and Read Latency Analysis. In Table II, as the ER value increases (i.e., the workload becomes more skewed), the performance gap between De-SCache and W/O-SCache decreases. Furthermore, due to the limited latency gap between the secondary cache and HDFS, when the Exp Range is set to 25, the throughput of De-SCache is even lower than that of W/O-SCache, and the latency of De-SCache is higher than that of W/O-SCache. What's more, SAS-Cache shows higher throughput and lower latency under all ER values compared with W/O-SCache and De-SCache. When the ER value is set to 5, the improvement is maximized, with a throughput increase of about 35% and a latency reduction of about 15%.

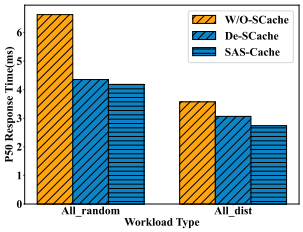
Cache Hit Ratio Analysis. We primarily record the overall hit ratio and LHR to reflect the improvement of SAS-Cache under a read-only workload. In a read-only workload, there is no compaction, only the Adaptive Bypass design influences the performance. As show in Table II, the overall hit ratio of De-SCache and SAS-Cache is much higher than that of W/O-SCache, and the overall hit ratio of De-SCache and SAS-Cache is almost the same. This is because Adaptive Bypass only improves the LHR and doesn't contribute to the overall hit ratio. Moreover, the LHR of SAS-Cache is much higher

than that of De-SCache, with an improvement of up to 45%.

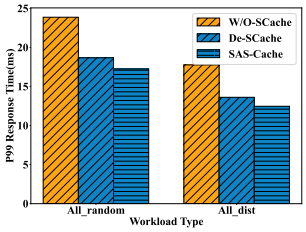


(a) Overall Throughput

(b) Overall Cache Hit Ratio



(c) Average Read Latency (P50)



(d) Tail Read Latency (P99)

Fig. 15. Real world Workload (Mixgraph Workload) Overall Throughput, Cache Hit Ratio and Read Latency Comparison.

3) Real-World Workload.

For the real-world workload, we utilize the mix-graph workload [7]. More specifically, we utilize two types of workloads: the all random and all dist workloads. The all dist workload has a more skewed data distribution than the all random workload. In order to be closer to the real workload, we set the key size to 48 bytes and the value size to 72 bytes, and set the number of threads to 32 [7]. For W/O-Cache, we choose the default block cache size; for De-SCache and SAS-Cache, we set the block cache size to 8 MB and set the flash-based secondary cache to 10 GB. For other settings, we followed the default configurations described in Section V-B choosing HDFS as the storage backend. Initially, we used the fillrandom workload to insert 300 million key-value pairs, and subsequently, we employed the mixgraph workload to operate on 5 million keys (a mix of get, put, and seek operations). The result is shown in Figure 15.

Throughput and Read Latency Analysis. In Figure 15(a), De-SCache has higher throughput than W/O-SCache no matter whether under All_random or All_dist workload. This is because of the significant hit ratio improvement as shown in Figure 15(b). Moreover, with SAS-Cache, we can achieve about up to 29% throughput improvement compared to De-SCache and 74% improvement compared to W/O-SCache. The average latency of RocksDB read queries is shown in Figure 15(c). The average read latency of De-SCache is much lower than that of W/O-SCache and its up to 34%. This is also because of the significant hit ratio improvement. Compared with De-SCache, SAS-Cache reduces average read latency, yielding an improvement of approximately 10% under the All_random workload. The tail latency results are shown in Figure 15(d). The result is similar to the average latency; the tail latency of De-SCache is much lower than W/O-SCache, and SAS-Cache has even lower tail latency than De-SCache.

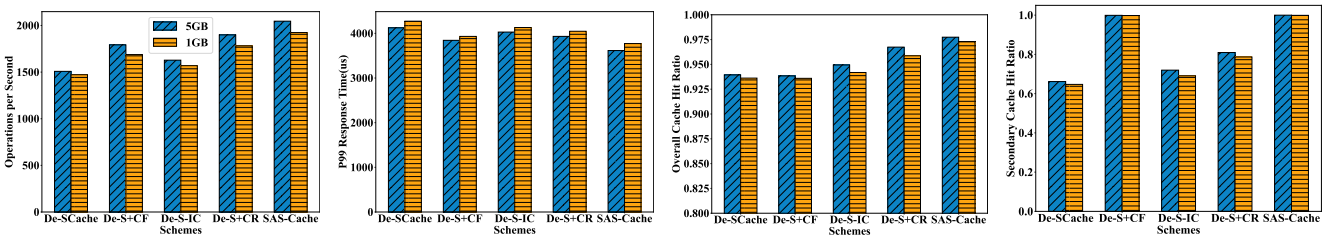
Cache Hit Ratio Analysis. The result of the overall cache hit ratio is shown in Figure 15(b). De-SCache shows a much higher hit ratio compared to W/O-SCache with an up to 45% improvement. This is because of the larger cache capacity using SSDs. Additionally, the overall hit ratio of SAS-Cache is higher than that of De-SCache, with an improvement of approximately 5%.

D. Performance Breakdown Analysis

In this section, we conducted a breakdown evaluation to demonstrate the effectiveness of three major optimizations of SAS-Cache and their impact on overall performance metrics including throughput, latency, and cache hit ratio. We configured the secondary cache size to be either 1 GB or 5 GB, while the block cache size remained the default 256 MB. We use 5 different setups: De-SCache, De-S+CF (De-SCache with cache filter), De-S+IC (De-SCache with insertion control), De-S+CR (De-SCache with compaction replacement), and SAS-Cache (De-SCache with all three optimizations). The results are presented in Figure 14 and 16. The results in Figure 14 pertain to a secondary cache size of 1 GB capacity.

Impact of Cache Filter. Cache Filter is primarily used to reduce unnecessary secondary cache lookups. Therefore, it has an explicit influence on the secondary cache hit ratio, which can significantly improve the LSM-KV store performance. Figure 14 shows that De-S+CF has significantly decreased secondary cache access numbers compared to De-SCache. Also, the number of secondary cache accesses in De-S+CF is nearly equivalent to the number of cache hits, indicating that the cache filter effectively filters out the secondary cache lookups that have a high probability to be cache misses. Figure 16(a) indicates that De-S+CF achieves about 15% higher throughput than that of De-SCache, which is the result of the secondary cache hit ratio improvement. Figure 16(b) shows that with the cache filter, latency decreases by about 8%. The overall hit ratio, as shown in Figure 16(c), remains unaffected by the cache filter, and De-SCache and De-S+CF exhibit almost identical overall hit ratios. Figure 16(d) demonstrates that the cache filter substantially improves the secondary cache hit ratio up to 37%.

Impact of Insertion Control. The Insertion Control is designed to prevent invalid blocks from being inserted into the secondary cache. This can improve the secondary cache effectiveness, SSD lifetime and the cache hit ratio while reducing insertion overhead. In Figure 14, with the same number of secondary cache lookup, De-S+IC has a higher number of secondary cache hits than De-SCache. This effectiveness is further highlighted in Figures 16(d) and 16(c), De-S+IC can achieve about 5% of cache hit ratio improvement than De-SCache. In terms of performance, Figure 16(a) demonstrates that De-S+IC achieves about 8% of throughput than improvement. In addition, Insertion Control slightly reduces the overall latency (about 2.5%) as shown in Figure 16(b). In Figure 19(b), it is shown that the insertion control can efficiently reduce unnecessary SSD writes by up to 24%, significantly prolonging the SSD lifetime.



(a) Throughput (b) P99 Response Time(us)) (c) Overall Cache Hit Ratio (d) Secondary Cache Hit Ratio

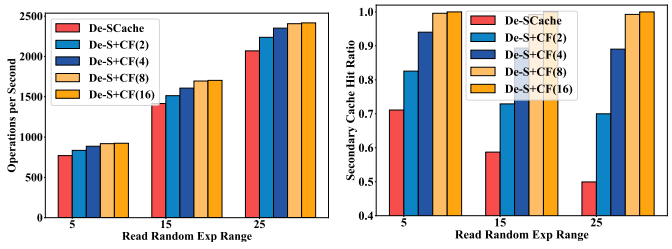
Fig. 16. Impact of Each Design on Performance (Throughput, Latency and Cache Hit Ratio).

Impact of Compaction Replacement. Compaction Replacement is designed to aggressively remove the invalid blocks caused by compaction and replace them with blocks from the newly generated SST files, which aims to improve the secondary cache and overall hit ratios. As shown in Figure 14, De-S+CR improves the total secondary cache hits by about 17% compared with that of De-SCache. Figures 16(c) and 16(d) show that with Compaction Replacement, we can achieve 14% of secondary cache hit ratio improvement and 3% of overall cache hit ratio improvement, compared with De-SCache. The performance comparison is shown in Figure 16(a) and Figure 16(b). De-S+CR can achieve about 21% of throughput improvement and 5% of latency reduction than that of De-SCache.

E. Sensitivity and Characteristic of Three Optimizations

1) Cache Filter Analysis.

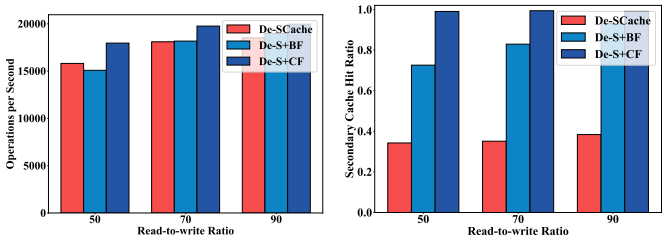
As discussed in Section IV-F, the effectiveness of the Cache Filter is closely tied to the secondary cache hit ratio and its impact becomes more pronounced when the cache hit ratio is relatively low. Furthermore, the size of the cache filter directly affects the false positive ratio, thereby influencing the filter's overall effectiveness. We use ReadRandom workload in the benchmarking with the same configurations described in Section V-B. Additionally, we varied the fingerprint size from 2 to 16 and compared De-SCache with De-S+CF(n), where n indicates the fingerprint size. The results are shown in Figure 17. When Exp Range is equal to 25 (the workload is highly skewed), we can achieve the best cache hit ratio and throughput improvement. De-S+CF(16) can achieve about hit ratio 50% improvements and 17% of throughput improvement. As the fingerprint size increases from 2 to 16 bits, the secondary cache hit ratio and throughput improvements increase.



(a) Throughput(ops) (b) Secondary Hit Ratio

Fig. 17. Cache Filter Analysis

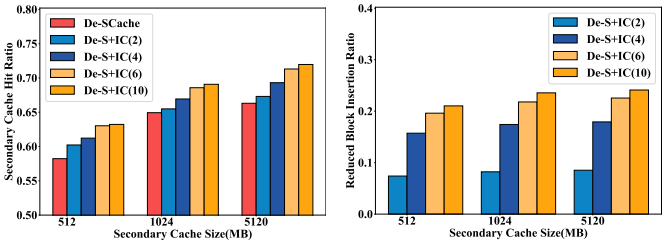
To further verify the effectiveness of our cache filter designs, we conduct comprehensive experiments comparing our cache filter with the bloom filter, which is used in Kangaroo [27].



(a) Throughput(ops) (b) Secondary Hit Ratio

Fig. 18. Comparison between Our Cache Filter and Bloom Filter.

In this experiment, we set the key size to 48 bytes and the value size to 72 bytes, and set the number of threads to 32. For cache size settings, we set the flash-based secondary cache size to 256MB. Moreover, we change the read-to-write ratio from 90% to 50% for more write-intensive workload evaluations. For other settings, we followed the default configurations described in Section V-B, choosing HDFS as the storage backend. For bloom filter settings, we set its capacity to the same as cache filter and false positive to 95%. The result is shown in Figure 18. For a bloom filter, it works well when the read-to-write ratio equals 90%, which largely improves the secondary cache hit ratio and also improves the throughput. However, as the read-to-write ratio decreases (more write-intensive), its effectiveness decreases and can even bring performance degradation when the ratio equals 50%. This is mainly because the bloom filter cannot update with the cache content instantly and causes false judgments. Compared with the bloom filter, the cache filter can work well regardless of the read-to-write ratio, and the maximum throughput improvement is about 14%.



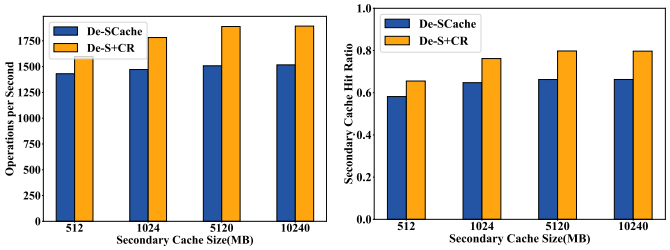
(a) Secondary Cache Hit Ratio (b) Reduced Block Insertion Ratio

Fig. 19. Insertion Control Analysis

2) Insertion Control Analysis.

As described in Section IV-F, The improvement of the Insertion Control is more pronounced when the cache is relatively smaller. What's more, the size of the queue largely influences its effectiveness. We run the default workload with

default configurations as described in Section V-B. We change the queue size from 2 to 10 and compare De-SCache with De-S+IC(n), where n indicates the queue size. The results are shown in Figure 19. Figure 19(a) that with a smaller secondary cache size, Insertion Control can achieve higher cache hit ratio improvement. With a larger queue size, the cache hit ratio will also be explicitly improved, When the queue size is configured as 10, we can achieve about 5% of cache hit ratio improvement. Also, with a larger queue size, more invalid data blocks can be identified. We reject about 24% of the total secondary cache insertions when the queue size is 10, shown in Figure 19(b).



(a) Throughput(ops) (b) Secondary Hit Ratio

Fig. 20. Compaction Replacement Analysis

3) Compaction Replacement Analysis.

As discussed in Section IV-F, the improvement of Compaction Replacement is more pronounced when the available cache space is substantial. We use default workload with default setting as described in Section V-B. We changed the secondary cache size from 512 MB to 5,120 MB. We compared De-SCache with De-S+CR, and the results are shown in Figure 20. Figure 20(b) shows that as the secondary cache size increases, De-S+CR can achieve higher secondary cache hit ratio improvement compared with De-SCache. When the secondary cache size is set to 10,240 MB, De-S+CR can improve about 15% of the secondary cache hit ratio. Figure 20(a) shows that as the secondary cache size increases, the throughput improvement of De-S+CR over De-SCache also increases. When the secondary cache size is 10,240 MB, De-S+CR can achieve 22% higher throughput than De-SCache.

VI. Related Work

Cache optimizations for LSM-KV Stores. There are several works related to the cache optimizations for LSM-KV Stores [32], [37]–[41]. These papers [32], [37]–[39] try to solve the problem of compaction-caused invalid blocks retained in the DRAM-based block cache. For example, LSbM-tree [38] combines the SM-tree [39] and bLSM [42] to maintain the connection between cache and disk during compaction. Unlike those previous works, the aim of this paper is to provide thorough guidelines on optimizing the flash-based secondary cache for LSM-KV stores. AC-Key [40] aims at LSM cache mechanisms in the memory, hybrids different kinds of cache objects(key-value pair, key-pointer pair, and block), and dynamically adjusts their sizes to improve cache efficiency.

Optimizations of LSM-KV Stores on Different Storage Backends. Recently, some of the studies optimized LSM-tree on different storage types: like cloud storage [15], [43], hybrid storage [44], [45] and ZNS SSDs [46], [47]. Calcspar [43] is designed to reduce latency spikes, adapt to changing workloads, and efficiently manage internal operation contentions when LSM-KV stores interact with Amazon's Elastic Block Store (EBS), which is a type of cloud storage [48] offered by Amazon Web Services (AWS). The proposed solution combines fluctuation-aware caching, congestion-aware IOPS allocation, and opportunistic compaction to improve the performance of LSM-KV stores specifically on EBS volumes. RocksDB for disaggregated infrastructure [15] tries to optimize LSM-KV stores on disaggregated storage (Tectonic File System [14] from Meta) involving latency issues, managing fault tolerance, ensuring data integrity during failovers, and adapting the RocksDB [6] to remote I/O behaviors. It proposes Metadata Cache, Local Flash Cache (Secondary Cache), Parallel IO, and Compaction Tuning to solve performance issues.

Flash-based Cache Optimizations. Recently, due to the higher cost-effectiveness and explicitly larger capacity of Flash-based SSD compared with DRAM, there are several studies designed and optimized Flash-based Cache [19], [27], [28], [49], [50]. CacheLib [19] uses a Large Object Cache (LOC) and Small Object Cache (SOC) to optimize flash caching for different object sizes. The LOC indexes large objects using efficient data structures with low DRAM overhead and the SOC uses approximate indexing with Bloom filters to cache small objects on flash. Flashield [28] uses DRAM as a filter to avoid writing non-flash-worthy objects to flash, based on predicting "flashiness" with lightweight machine learning classifiers. It writes predicted flash-worthy objects to flash sequentially in large chunks to minimize device-level write amplification and features a highly efficient in-memory index for variable-sized flash objects using less than 4 bytes per object. Kangaroo [27] combines a small log-structured cache (KLog) to reduce flash writes with a large set-associative cache (KSet) to minimize DRAM overhead. It introduces threshold admission and RRIParoo eviction to further reduce flash writes and improve hit ratio.

VII. Conclusion

In this paper, we first conduct a comprehensive analysis of the existing secondary cache designs and explore their limitations. Then, we propose SAS-cache for LSM-KV stores, which improves the existing secondary cache designs with three major optimizations. LSM-Managed Cache Filter is proposed to effectively reduce the number of unnecessary secondary cache lookups. Additionally, Valid SST-aware insertion control is proposed to prevent the blocks belonging to the obsoleted SST files from being inserted into the secondary cache. Moreover, compaction-aware cache replacement actively removes the cache items invalidated by compactions. Our evaluation shows that SAS-Cache can achieve about 40% of the secondary cache hit ratio improvement, 36% of the throughput improvement, and 20% of latency reduction compared with the state-of-the-art secondary cache. In our future work, we will explore the multi-tenancy and workload adaptiveness of SAS-Cache.

ACKNOWLEDGEMENTS

We would like to thank our shepherd, Xiaodong Zhang, and all the anonymous reviewers for their valuable feedback. We thank all the members of ASU-IDI Lab for providing useful comments. This work was partially funded by the Arizona State University startup fund.

REFERENCES

[1] Google, "Leveldb. https://github.com/google/leveldb," 2023. Accessed March 25, 2023.

[2] Apache, "Hbase. https://hbase.apache.org/," 2023. Accessed 10 Jan, 2023.

[3] Z. Cao, H. Dong, Y. Wei, S. Liu, and D. H. Du, "Is-hbase: An in-storage computing optimized hbase with i/o offloading and self-adaptive caching in compute-storage disaggregated infrastructure," *ACM Transactions on Storage (TOS)*, vol. 18, no. 2, pp. 1–42, 2022.

[4] Meta, "Zippydb. https://www.zippydb.com/," 2023. Accessed 10 Jan, 2023.

[5] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li, "X-engine: An optimized storage engine for large-scale e-commerce transaction processing," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 651–665, 2019.

[6] FaceBook, "Rocksdb. https://github.com/facebook/rocksdb," 2023. Accessed March 25, 2023.

[7] Z. Cao and S. Dong, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *18th USENIX Conference on File and Storage Technologies (FAST'20)*, 2020.

[8] H. T. Kassa, J. Akers, M. Ghosh, Z. Cao, V. Gogte, and R. Dreslinski, "Power-optimized deployment of key-value stores using storage class memory," *ACM Transactions on Storage (TOS)*, vol. 18, no. 2, pp. 1–26, 2022.

[9] H. T. Kassa, J. Akers, M. Ghosh, Z. Cao, V. Gogte, and R. G. Dreslinski, "Improving performance of flash based key-value stores using storage class memory as a volatile memory extension.," in *USENIX Annual Technical Conference*, pp. 821–837, 2021.

[10] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb.," in *CIDR*, vol. 3, p. 3, 2017.

[11] F. Wu, B. Li, Z. Cao, B. Zhang, M.-H. Yang, H. Wen, and D. H. Du, "Zonealloy: Elastic data and space management for hybrid smr drives," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[12] F. Wu, B. Li, B. Zhang, Z. Cao, J. Diehl, H. Wen, and D. H. Du, "Tracklace: Data management for interlaced magnetic recording," *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 347–358, 2020.

[13] Z. Cao, H. Wen, F. Wu, and D. H. Du, "Smrts: A performance and cost-effectiveness optimized ssd-smr tiered file system with data deduplication," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*, pp. 275–282, IEEE, 2023.

[14] S. Pan, T. Stavrinos, Y. Zhang, A. Sikaria, P. Zakharov, A. Sharma, M. Shuey, R. Wareing, M. Gangapuram, G. Cao, *et al.*, "Facebook's tectonic filesystem: Efficiency from exascale," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pp. 217–231, 2021.

[15] S. Dong, S. S. P, S. Pan, A. Ananthabhotla, D. Ekambaram, A. Sharma, S. Dayal, N. V. Parikh, Y. Jin, A. Kim, *et al.*, "Disaggregating rocksdb: A production experience," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–24, 2023.

[16] P. Xu, N. Zhao, J. Wan, W. Liu, S. Chen, Y. Zhou, H. Albahar, H. Liu, L. Tang, and Z. Tan, "Building a fast and efficient lsm-tree store by integrating local storage with cloud storage," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 3, pp. 1–26, 2022.

[17] RockSet, "Rocksdb-cloud. https://github.com/rockset/rocksdb-cloud," 2023. Accessed March 25, 2023.

[18] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "Evolution of development priorities in key-value stores serving large-scale applications: The rocksdb experience.," in *FAST*, pp. 33–49, 2021.

[19] B. Berg, D. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, *et al.*, "The cachelib caching engine: Design and experiences at scale," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*, 2020.

[20] Github, "Sas-cache. https://github.com/asu-idi/SAS-Cache," 2023. Accessed March 25, 2023.

[21] Amazon, "Amazon s3. https://aws.amazon.com/s3/," 2023. Accessed March 25, 2023.

[22] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, *et al.*, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 143–157, 2011.

[23] Z. Chen, X. Yang, F. Li, X. Cheng, Q. Hu, Z. Miao, R. Xie, X. Wu, K. Wang, Z. Song, *et al.*, "Cloudjump: optimizing cloud databases for cloud storages," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3432–3444, 2022.

[24] T. Estro, P. Bhandari, A. Wildani, and E. Zadok, "Desperately seeking... optimal {Multi-Tier} cache configurations," in *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[25] FaceBook, "dbbench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools/," 2023. Accessed March 25, 2023.

[26] N. Beckmann, H. Chen, and A. Cidon, "{LHD}: Improving cache hit rate by maximizing hit density," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pp. 389–403, 2018.

[27] S. McAllister, B. Berg, J. Tutuncu-Macias, J. Yang, S. Gunasekar, J. Lu, D. S. Berger, N. Beckmann, and G. R. Ganger, "Kangaroo: Caching billions of tiny objects on flash," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 243–262, 2021.

[28] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti, "Flashield: a hybrid key-value cache that controls flash write amplification," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 65–78, 2019.

[29] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pp. 75–88, 2014.

[30] Facebook, "Rocksdb secondary instance. https://github.com/facebook/rocksdb/wiki/Read-only-and-Secondary-instances," 2023. Accessed March 25, 2023.

[31] J. Z. V. T. J. W. Qiaolin Yu, Chang Guo and Z. Cao, "Caas-lsm: Compaction-as-a-service for lsm-based key-value stores in storage disaggregated infrastructure.," *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–26, 2024.

[32] M. Y. Ahmad and B. Kemme, "Compaction management in distributed key-value datastores," *Proceedings of the VLDB Endowment*, vol. 8, no. 8, pp. 850–861, 2015.

[33] Github, "Cuckoofilter. https://github.com/efficient/cuckoofilter," 2023. Accessed March 25, 2023.

[34] TOSHIBA, "Toshiba mg06aca8. https://storage.toshiba.com/enterprise-hdd/enterprise-capacity/mg06-series," 2023. Accessed March 25, 2023.

[35] HDFS, "Centralized cache management in hdfs. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html," 2023. Accessed March 25, 2023.

[36] Samsung, "Samsung 980pro. https://semiconductor.samsung.com/consumer-storage/internal-ssd/980pro/," 2023. Accessed March 25, 2023.

[37] L. Yang, H. Wu, T. Zhang, X. Cheng, F. Li, L. Zou, Y. Wang, R. Chen, J. Wang, and G. Huang, "Leaper: A learned prefetcher for cache invalidation in lsm-tree based storage engines," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 1976–1989, 2020.

[38] D. Teng, L. Guo, R. Lee, F. Chen, S. Ma, Y. Zhang, and X. Zhang, "Lsbm-tree: Re-enabling buffer caching in data management for mixed reads and writes," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 68–79, IEEE, 2017.

[39] H. Jagadish, P. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, "Incremental organization for data recording and warehousing," in *VLDB*, pp. 16–25, 1997.

[40] F. Wu, M.-H. Yang, B. Zhang, and D. H. Du, "{AC-Key}: Adaptive caching for {LSM-based}{Key-Value} stores," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 603–615, 2020.

[41] X. Wu, L. Zhang, Y. Wang, Y. Ren, M. Hack, and S. Jiang, "zexpander: A key-value cache with both high performance and fewer misses," in *Proceedings of the Eleventh European Conference on Computer Systems*, pp. 1–15, 2016.

[42] R. Sears and R. Ramakrishnan, "blsm: a general purpose log structured merge tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 217–228, 2012.

[43] Y. Zhou, J. Zhou, S. Chen, P. Xu, P. Wu, Y. Wang, X. Liu, L. Zhan, and J. Wan, "Calcspar: A {Contract-Aware}{LSM} store for cloud storage with low latency spikes," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 451–465, 2023.

[44] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "{SpanDB}: A fast,{Cost-Effective}{LSM-tree} based {KV} store on hybrid storage," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pp. 17–32, 2021.

[45] J. Li, Q. Wang, and P. P. Lee, "Efficient lsm-tree key-value data management on hybrid ssd/hdd zoned storage," *arXiv preprint arXiv:2205.11753*, 2022.

[46] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *Proceedings of the Ninth European Conference on Computer Systems*, pp. 1–14, 2014.

[47] J. Lee, D. Kim, and J. W. Lee, "Waltz: Leveraging zone append to tighten the tail latency of lsm tree on zns ssd," *Proceedings of the VLDB Endowment*, vol. 16, no. 11, pp. 2884–2896, 2023.

[48] Amazon, "Cloud storage from aws. https://aws.amazon.com/what-is-cloud-storage/," 2023. Accessed March 25, 2023.

[49] Netflix, "Netflix technology blog. application data caching using ssds. https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef," 2023. Accessed March 25, 2023.

[50] Netflix, "Netflix technology blog. evolution of application data caching : From ram to ssd https://netflixtechblog.com/evolution-of-application-data-caching-from-ram-to-ssd-a33d6fa7a690," 2023. Accessed March 25, 2023.