BloomStore: Bloom-Filter based Memory-efficient Key-Value Store for Indexing of Data Deduplication on Flash

Guanlin Lu
EMC^2
Santa Clara, CA
Guanlin.Lu@emc.com

Young Jin Nam
Daegu University
Gyeongbuk, KOREA
yjnam@daegu.ac.kr

David H.C. Du
University of Minnesota
Minneapolis, MN
du@cs.umn.edu

***Abstract*—Due to its better scalability, Key-Value (KV) store has superseded traditional relational databases for many applications, such as data deduplication, on-line multi-player gaming, and Internet services like Amazon and Facebook. The KV store efficiently supports two operations (key lookup and KV pair insertion) through an index structure that maps keys to their associated values. The KV store is also commonly used to implement the chunk index in data deduplication, where a chunk ID (SHA1 value computed based on the chunk's content) is a key and its associative chunk metadata (e.g., physical storage location, stream ID) is the value. For a deduplication system, typically the number of chunks is too large to store the KV store solely in RAM. Thus, the KV store maintains a large (hash-table based) index structure in RAM to index all KV pairs stored on secondary storage. Hence, its available RAM space limits the maximum number of KV pairs that can be stored. Moving the index data structure from RAM to flash can possibly overcome the space limitation.**

In this paper, we propose efficient KV store on flash with a Bloom Filter based index structure called BloomStore. The unique features of the BloomStore include (1) no index structure is required to be stored in RAM so that a small RAM space can support a large number of KV pairs and (2) both index structure and KV pairs are stored compactly on flash memory to improve its performance. Compared with the state-of-the-art KV store designs, the BloomStore achieves a significantly better key lookup performance and roughly the same insertion performance with multiple times less RAM usage based on our experiments with deduplication workloads.

I. Introduction

The key-value (KV) store contains a large number of KV pairs and provides two simple operations: key lookup and KV pair insertion. These two operations heavily depend on an internal index structure that maps a key to its associated value. Recently, many applications, such as data deduplication [1], on-line multi-player gaming, and Internet services like Amazon and Facebook [2], etc., have preferred to use the KV store, rather than the traditional relational database, because of its simplicity and better scalability. In order to maximize the KV store performance, we need to carefully provide efficient index and KV pair accesses based on the characteristics of the underlying storage media containing the index structure and the KV pairs.

978-1-4673-1747-4/12/$31.00

The performance of the KV store often governs the performance of its applications. The KV store is commonly used to implement the chunk index in data deduplication, where a chunk ID (SHA1 value computed based on the chunk's content) is the key and its associative chunk metadata (e.g., physical storage location, stream ID, etc.) is the value. Chunk lookup searches a given chunk ID from the KV store, while chunk insertion adds a new chunk ID and its metadata to the KV store. Zhu et al. [1] pointed out that the key performance bottleneck for (in-line) data deduplication is its key (chunk) lookup throughput. In addition, applications that detect redundant data transfers across WANs and subsequently send their associated references are recently demanding key lookup throughputs of no less than $10,000$ operations per second [3].

To design a high-throughput KV store ($> 10,000$ key lookups/second), a typical method is to keep its index structure in RAM to rapidly map each key to its KV pair location on the secondary storage, such as flash or HDD [4]. Also, many KV store designs rely on an in-RAM large-sized hash table to index all KV pairs stored on the flash. Nevertheless, the downside of this approach is that the maximum number of KV pairs in the KV store can be constrained by the available RAM space (scalability constraint).

Provided the high throughput and low access latency requirements, the most cost-effective way to scale up the KV store is to move part of its index structure into the secondary storage. Recently, flash-memory (particularly in the form of SSD) has become one of the popular storage alternatives to the traditional HDD. The flash-memory could persistently store the index and deliver an access speed 100–$1,000$ times faster than the HDD. Compared to RAM (DRAM), however, the flash access speed is 100 times slower. As for the unit price (in terms of $/GB), flash-memory is 10 times cheaper than RAM, while it is 20 times more expensive than HDDs, thus, positioning it in the middle of these two storage devices. In our design, we use a NAND flash-memory based SSD as the secondary storage. Throughout the rest of this paper, for simplicity, we refer to the SSD as flash. Accordingly, each read/write operation on the SSD is in the unit of a flash page.

In order to break the scalability constraint (but losing performance benefits to some extent), the index structure (e.g., a large hash table) should be eventually stored in the flash-memory instead of the RAM. However, many index structures that involve intensive small random writes become challenging to be stored on the flash. More specifically, storing a hash-table based index structure on flash causes a few problems: (1) the hash table is randomly accessed (inserted); each KV insertion triggers an expensive random flash page write operation to modify only one hash table entry of several bytes, which is much smaller than the size of a flash page; (2) the hash table is not a garbage-collection friendly data structure, spreading inserted entries across all flash pages occupied by the hash table; even a small portion of invalidated entries (e.g., by update or delete) may lead to excessive in-place updates scattering a large number of pages, which severely aggravates flash write and garbage collection overheads; and (3) the hash table requires a much bigger storage space, as the load factor of an efficient hash table usually needs to be well below 50% to keep the lookup time bounded.

In this paper, we aim at designing a flash-based KV store architecture called BloomStore that not only assures an extremely low amortized RAM overhead per KV pair (the consumed RAM space divided by the total number of KV pairs) to be less than 1 byte/key, but also achieves high key lookup/insertion throughput. Each physical machine runs multiple BloomStore instances, each of which is responsible for a disjoint key range. Each BloomStore instance indexes its own key-range partition separately with a sequence (chain) of Bloom Filters (BFs). It associates a BF with a flash page of KV pairs, where the BF summarizes the keys in the flash page and has a flash pointer to the flash page. For high lookup throughput, BloomStore reduces the maximum number of flash page reads by key-range partitioning. For high insertion throughput, each BloomStore instance maintains a dedicated flash page sized KV pair buffer to temporarily buffer inserted KV pairs. For minimal RAM usage, all other BFs and associated KV pairs are stored on the flash. Under two different breeds of data dedup workloads, our experiments reveal that BloomStore outperforms the state-of-the-art KV store designs in terms of the RAM usage and key lookup throughput.

The rest of the paper is organized as follows. Section II provides a detailed survey of existing works. Section III presents the BloomStore design. Section IV gives an extensive experimental evaluation of our BloomStore design with two typical real workloads from data dedup applications. Section V summarizes our work and draw conclusions.

II. Related Work

MicroHash [5] is an index structure designed for memory-constrained embedded devices. It mainly emphasized the optimization of energy usage and the memory footprint, not access latency.

Wu et al. [6] and Nath et al. [7] proposed the on-flash B-tree and B+ tree (FlashDB) solutions, respectively. However, they are application-specific and efficient under the following limited conditions: (1) keys are distributed in a small numerical range; (2) a small number of leaf-level buckets are active at any given time; (3) access latencies are not critical.

FAWN [8] and ChunkStash [4] store a checksum and a pointer into an index entry of an in-RAM hash table that points to a single KV pair stored in flash. The checksum is used to avoid (with high probability) triggering flash accesses to compare keys for every index entry searched in the hash table during key lookup. The amortized RAM overhead per KV pair of each design is computed as 6 bytes/key. For these designs that index all keys with a single hash table maintained in RAM, the lower-bound RAM overhead is the footprint of the flash pointers (e.g., 4 bytes/key). The KV store size in flash is constrained by the flash pointer size and its KV pair size. For example, the flash space of the KV store with a 4-byte flash pointer and a 64-byte KV pair length can be 256 GB at maximum.

In the meantime, three recent KV store designs, BufferHash [9], SkimpyStash [10] and SILT [11], have reduced the RAM overhead per key by efficiently placing their index structures over the RAM and flash. We will look into these three designs in more detail as follows.

We begin by explaining the properties of a Bloom Filter (BF) that are widely used to devise the index structures of many KV store designs.

Bloom Filters [12]: A BF supports space-efficient membership queries as follows: (1) a set $S = \{e_1, e_2, ..., e_n\}$ of n keys is represented by a vector v of m bits, initially all set to 0; (2) a set of k different hash functions $h_1, ..., h_k$, are used to set bits at $h_1(e), h_2(e), ..., h_k(e)$ positions for each e_i in set S; (3) to lookup a key e_i, bits at positions at $h_1(e_i), h_2(e_i), ..., h_k(e_i)$ are checked; (4) if any of these are 0, then e_i is not present in the set for sure; otherwise, it concludes that the key e_i is in the set (an affirmative answer); (5) false positive errors may exist; e.g., a key that is not in the set is mapped to k bit positions which are already set to 1 during insertions of other keys; and (6) a false positive probability f is affected by the BF parameters: n, m and k, where f is calculated as $(1-(\frac{1}{m})^{kn})^k \approx (1-e^{-kn/m})^k$. The right-hand-side is minimized for $k = \ln(2) \cdot \frac{m}{n}$, indicating that an optimum k exists for every choice of m and n. The false positive probability for the optimum k is $(0.5)^k = (0.6185)^{\frac{m}{n}}$.

BufferHash [9]: It divides the flash space into a number of logical partitions, as shown in Figure 1. Each partition maintains a small in-RAM hash table (HT buffer) for the KV pairs stored in the partition. The BufferHash basically employs the hash tables to store its index structure and associated KV pairs. Each hash table is implemented by using the cuckoo hashing [13] with two hash functions that help to improve the space utilization efficiency (e.g., to attain 50% load factor) at the cost of more hash table lookups per key. When the in-RAM hash table of a partition becomes full (i.e., its load factor reaches a predefined threshold), it is written to the flash. Subsequently, a new hash table of the same size is instantiated in RAM for the incoming KV pair insertions. In this way,

the multiple hash tables of a partition are being chained with the newest (chronologically) hash table residing in RAM to accommodate newly inserted KV pairs. Suppose that there are P partitions, and each partition on average contains C hash tables. Then, only $1/C$ fraction of the entire hash tables are kept in RAM. The BufferHash keeps a BF in RAM for each hash table stored in either RAM on flash. In order to look up a key in a partition, the BufferHash identifies a specific partition where the key resides by using a hash function. Next, it examines the chain of the BFs linked to the partition in the reverse order of their creation times. For each of the BFs where the key is found (note that there could be multiple BFs that have the key), the BufferHash looks up the key from the associated hash tables stored either in the HT buffer or on the flash.

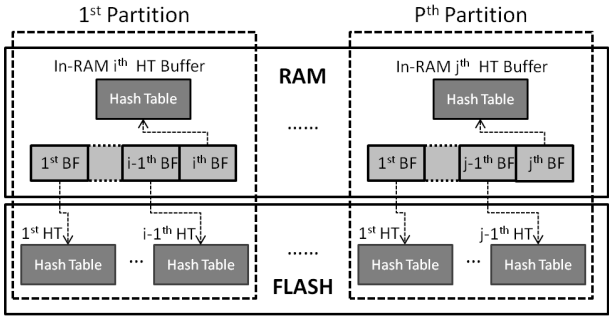

Figure 1. BufferHash architecture: multiple partitions, a hash table buffer and a chain of BFs in RAM for each partition, and a chain of corresponding hash tables in flash for each partition

However, the BufferHash consumes considerable RAM space for the following reasons: (1) hash tables have low load factors (50% recommended in the BufferHash design); and (2) all the BFs for all the partitions are kept in RAM.

SkimpyStash [10]: It stores all the KV pairs in flash, while maintaining an in-RAM hash table (called the hash table directory) to map keys to their locations in flash, as illustrated in Figure 2. Unlike the BufferHash, SkimpyStash stores a flash pointer instead of the actual KV pair in the hash table. The SkimpyStash regards the flash as an append-log. It appends the inserted KV pairs to the log sequentially. It also maintains the single in-RAM data buffer of a flash page size to temporarily hold new KV pairs. When the buffer becomes full, SkimpyStash flushes the KV pairs in the buffer to the flash through an append operation. This has been proven to be an efficient way to maximize the write performance of the flash-memory [14]. To further minimize the RAM usage, it hashes multiple keys into the same bucket in the hash table. It then resolves any collisions with the linear chaining, where the KV pairs in the same bucket are chained in a linked list and stored in the flash. The in-RAM hash table consists of a set of buckets that contains a BF and a flash pointer. Each flash pointer points to the tail (the most recently inserted KV pair) of the corresponding linked list. Each KV pair on the flash contains a flash pointer pointing to its predecessor (the previously inserted KV pair in the same bucket) in addition to its KV pair. Each bucket also keeps a BF in RAM to memorize the inserted keys in that bucket. This BF helps to decide whether the searched key exists in a bucket before blindly following the pointer to search the key from the chain of KV pairs in the flash. The use of the BF is crucial for the SkimpyStash design to reduce flash page reads for key lookups. Otherwise, it has to always traverse the entire linked list to conclude the non-existence of a searched key.

SkimpyStash may incur multiple flash page reads for a key lookup to determine the demanded key from the chained KV pairs. Suppose that the average chain length per bucket is l, then each key lookup needs to have $0.5l$ flash page reads on average. Considering the desirable uniformity of a key distribution into the buckets, the chance becomes extremely small that a series of new keys colliding with the same bucket are stored in the same flash page. Thus, the average number of flash page reads for each key lookup is proportional to the number of keys hashed into a bucket.

A formula $(1 + \frac{4}{average\ bucket\ length})$ is provided to calculate the amortized RAM overhead per key stored with the SkimpyStash design. For an average bucket length of 10, it pays a 1.4 byte RAM footprint for each KV pair stored in the flash.

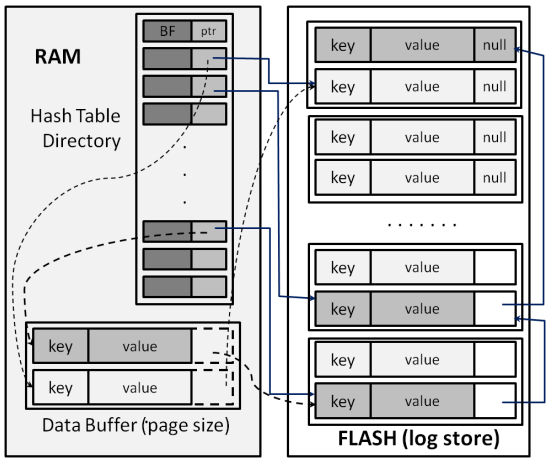

Figure 2. SkimpyStash architecture: a hash table directory and a single data buffer in RAM, and linked-listed KV pairs in flash

SILT [11]: it is constituted by a series of basic KV stores, each of which is optimized for a different purpose. KV pairs are inserted into a write-optimized store, called LogStore, and gradually migrated to increasingly more memory-efficient stores. The LogStore sequentially writes the inserted KV pairs into on-flash data log. The KV pairs are ordered by their insertion time. The LogStore indexes the data log by an in-RAM hash table. To improve the RAM space utilization, the hash table is built with cuckoo hashing [13]. To make it more compact, the hash table does not store the full key but only a tag of the actual key. Moreover, SILT boosts the hash table occupancy to about 93% by increasing the associativity of the hash table (i.e., having more candidate victims that could be "kicked out" once neither bucket is available). Each hash table entry consumes 6 bytes, consisting of a 15-bit

tag, a single valid bit, and a 4-byte offset pointer. Once a LogStore fills up, SILT freezes the LogStore and begins to convert it into a more memory-efficient, static data structure, called SortedStore. The SortedStore maintains KV pairs in a sorted key order on flash and indexes them with a very compact in-RAM index representation, called *entropy-coded tries*. In contrast to the 6 bytes/key RAM indexing overhead of LogStore, SortedStore achieves sub-byte range (e.g., 0.4 bytes/key) RAM indexing overhead. The entropy-coded tries, however, does not allow insertions or deletions. Therefore, to merge LogStore entries into the SortedStore, SILT must create a new SortedStore. Hence, directly sorting a relatively small LogStore and merging it into the much larger SortedStore requires rewriting the entire SortedStore. To amortize the cost of rewriting the entire SortedStore, SILT first converts the LogStore to an intermediate KV store called HashStore with higher memory efficiency (4 bytes/key RAM indexing overhead). Once SILT accumulates a sufficient number of HashStores, it performs merge operation in bacth mode to incorporate these HashStores into the SortedStore. In this way, the immutable HashStores essentially serves as the input buffer for SortedStore. Most KV pairs are stored in SortedStore (e.g., $> 80\%$) to make the average index cost per key low.

Although SILT is suitable for lookup-intensive workloads (i.e., the major portion of workloads are key lookups), it pays even higher insertion overhead than either BufferHash or SkimpyStash. There are a few significant disadvantages of SILT design, preventing it from being a suitable candidate KV store for inline data deduplication systems. Firstly, the SILT design involves complicated and repeated conversion and merge operations running in backgrond. Each conversion operation requires reordering KV pairs of the LogStore on flash. Each merge operation requires sorting all merging keys and rewriting the entire SortedStore. Both background operations compete for a significant amount of I/O resources, making it unable to meet the high key insertion throughput demanded by typical in-line data deduplication systems. For example, the middle-end EMC Inline Deduplication System [15] provides near 10 TB/hour data backup throughput, demanding a stable key insertion rate over $300,000$ requests/second, far beyond the $36,000$ requests/second achieved by SILT [11]. Secondly, indexing chunks using SILT would require sorting a major fraction (e.g., $> 80\%$) of all chunk-ids, competing for a significantly large amount of computational resources. Within typical real deduplication systems like EMC Inline Deduplication Storage Systems series, there is aleady a number of computational intensive tasks, such as chunking, garbage collection, and even replication, running. Sorting a major fraction of all chunk-ids (i.e., the keys) would add a prohibitively expensive computational overhead for a deduplicaiton system of billions of unique data chunks. Thirdly, SILT might fail to work (e.g., overflowing the flash space) once the number of update/delete operations increases beyond even a fairly low threshold, for the following reason: each update/delete operation to an immutable HashStore will be translated into an insert operation to a new HashStore instance. Owing to the significantly high merging overhead, the invoking frequency of the merge operation has to be limited to avoid significant foreground lookup/insert throughput degradation. Within a certain period of time, if the number of update/delete operation is high, more HashStore instances would be created than the number of instances merged to the new SortedStore. Thus, these pending-to-merge HashStore instances would eventually overflow the flash space. On the other hand, if we allow only a limited number of HashStores instances to be created at the same time (e.g., 31 HashStores per SILT instance, as presented in the paper), SILT may not be able to cope with the desired troughput.

III. BloomStore Design

Our KV store design is driven by the goal to deliver high key lookup/insertion throughput to meet the demands of the recent KV store applications at the minimum RAM space usage. We use the following performance metrics to evaluate the effectiveness of our design:

- **Amortized RAM overhead per KV pair**: It measures how frugal the RAM space usage is that a KV store design can achieve. It is defined as the consumed RAM space (by the KV store) divided by the total number of KV pairs in the store. This metric can be further decomposed into two parts: (1) the amortized RAM overhead to index the key; and (2) the amortized RAM overhead to buffer the key in the data buffer. The rationale behind this metric is as follows. The RAM size imposes a big scalability challenge to the existing KV store designs. As the growth rate of the flash-memory capacity is nonlinearly faster than that of the RAM capacity (e.g., while the RAM capacity is still in the ten or tens of GBs level, a $1,024$GB SSD is already in market), it becomes impractical for a KV store design to use an amount of RAM space proportional to the overall flash space in the store. Our KV store design overcomes the limitation by partitioning the overall key-range and placing the index structures corresponding to each key-range partition on the flash. With this design choice, our design uses extremely frugal RAM space (e.g., < 1 byte/key) for the index and data buffers. As a byproduct of this design, when an unexpected event like a power failure occurs, our design is expected to experience a much shorter service outage than those designs that store index structures in the RAM. This is because the latter ones need to scan and reprocess all stored KV pairs for reconstructing the index structure in the RAM.
- **KV lookup/insertion throughput**: These two metrics measure the performance of the basic KV store operations, crucial in order to meet the demands of the recent KV store applications. Our design should be able to deliver high lookup and insertion throughputs to be comparable to the state-of-the-art design (SkimpStash) with the same amount of RAM space.

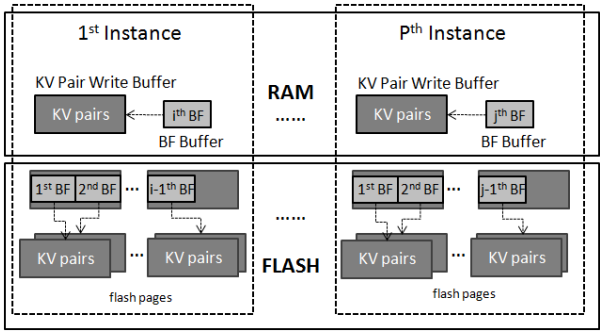

Figure 3. BloomStore architecture: P BloomStore instances, corresponding the same number of key-range partitions, where each instance contains a KV pair write buffer (flash page size), a BF buffer, a chain of BFs, and their associated data (KV pairs) pages in flash

A. Overall Architecture

Figure 3 shows the overall architecture of our KV store instance called BloomStore. Each BloomStore instance consists of the following four components: a KV pair write buffer, a BF buffer, a BF chain, and a number of data pages. To minimize the amortized RAM space overhead per KV pair, BloomStore only maintains a flash-page sized data (KV pairs) buffer and a very small sized BF buffer in RAM. To achieve high lookup/insertion throughput, BloomStore reduces the number of flash page reads and writes in the following ways: (1) it runs multiple BloomStore instances, each of which is responsible for a disjoint key range. This key-range partitioning scheme would reduce the maximum number of flash page reads for key lookups; (2) It treats the flash-memory as an append-log. For the KV pair insertion, it maintains a dedicated KV pair write buffer to temporarily buffer the inserted KV pairs and appends the KV pairs in a single flash page sequentially to the end of the log on flash.

Hereafter, we will describe each of the BloomStore components in more detail. To begin with, each BF in BloomStore consists of a flash pointer (4 bytes) and a bit vector for the normal BF operation. The flash pointer points to the address (in the form of a LBA, short for "Logical-Block-Address", which is commonly supported by the block interface of most flash-memory devices) of its associated page of KV pairs on flash.

KV Pair Write Buffer: Each BloomStore instance has a dedicated fixed-sized in-RAM data buffer to log incoming KV pair insertions (called a KV pair write buffer). Its size is equal to the underlying flash page size. This buffer is flushed (written) into the flash only when it is filled up with the inserted KV pairs. Data pages written by different BloomStore instances would be interleaved in the append-log on flash. Thus, the number of flash page writes can be reduced. If more stringent durability is required, a configurable timeout interval (e.g., 1 ms) can be used to ensure that a flash write occurs in the interval.

BF Buffer: Each BloomStore instance only keeps a very small sized active BF in the BF buffer while storing all the other BFs in the flash. The BloomStore uses a single BF to summarize up to the maximum number of KV pairs stored in a flash page. **The active BF** in the BF buffer represents the membership of the keys stored in the KV pair write buffer. Thus, when the KV pair write buffer is written into the flash, the active BF (in the BF buffer) is also flushed into the flash. At this time, the flash pointer of the active BF is updated with a valid LBA of the flash page. This BF buffer flush operation involves a number of flash page accesses: (1) read the flash pages containing a chain of other BFs (so called the **remainder of the BF chain**) in this instance into an temporary in-RAM buffer; (2) move the active BF into this temporary buffer; (3) append the buffer into the flash; and (4) update the in-RAM pointer with the beginning address of the flash pages storing the newly written remainder of the BF chain. Each BloomStore instance maintains an in-RAM pointer (not shown in Figure 3) pointing to the valid beginning address of the flash pages storing the remainder of the BF chain.

The minimum RAM usage of a BloomStore instance includes a BF size of a BF buffer and a flash page size of a KV pair write buffer. Then, **the minimum RAM usage** can be computed by multiplying the per-instance minimum RAM usage with the number of allocated BloomStore instances.

BF Chain: The BloomStore instance indexes a number of data pages independently with a sequence of BFs (briefly a BF chain), each of which memorizes the inserted KV values in one flash page. The associated BF chain is decomposed into two parts: (1) the active BF that is most recently instantiated in the chain and is always stored in RAM (the BF buffer); and (2) the remainder of the BF chain that consists of the rest of BFs in the chain and is stored in the flash. For example, in the first BloomStore instance of Figure 3, the active BF is the i^{th} BF. The remainder of the BF chain is a list of BFs including the 1^{st} BF through the $i-1^{th}$ BF. The **BF chain length** is defined as the number of BFs in the BF chain.

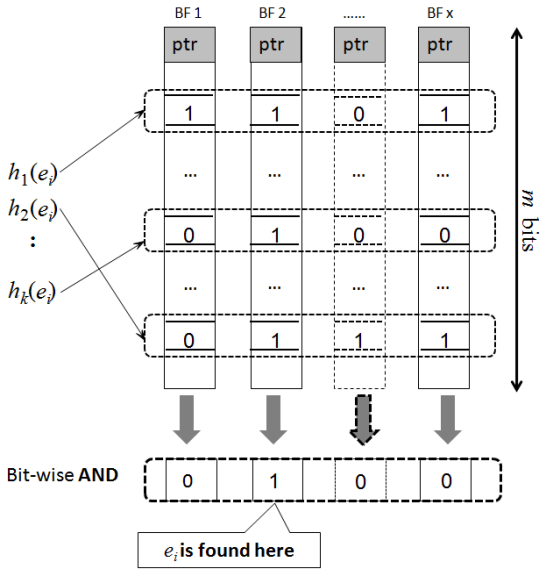

Figure 4. Illustration of checking multiple bloom filters in parallel

Bloom Filter Parallel Lookup: To look up a key e_i, our BloomStore design first uses a hash function to locate an

associated key-range partition, followed by checking the key e_i in its BF chain. Initially, a BF chain contains only an active BF in the BF buffer, i.e., no BFs are stored in the flash. Thus, simply one BF lookup operation with the active BF is enough to check the key. Assume that x flash pages of KV pairs in the instance have been written to the flash. It implies that there is a BF chain of x BFs. With the minimum BF buffer size, only the active BF is in the BF buffer, while the rest of the BF chain (containing other $x-1$ BFs) is in the flash. Thus, we need to read the $(x-1)$ BFs from the flash to an (temporary) in-RAM buffer and then check the key in each of the x BFs. Note that the temporary buffer to hold the BFs can be shared by all the BloomStore instances (causing negligible RAM space overhead). Simply, each of x BFs should be checked separately, requiring x separate BF lookup operations.

However, BloomStore design adopts the parallel BF checking scheme proposed in the SegmentedHash [16] to check the buffered BFs in parallel. Figure 4 illustrates an example of checking x BFs in parallel. Each m-bit BF (represented as a column) consists of a bit vector and a flash pointer pointing to a flash page of KV pairs. All x BFs use the same group of k different hash functions. Thus, for a given key e_i, the same bit positions are checked in each BF. As such, a row of bits can be composed of one bit from each of the x BFs, as illustrated in the Figure 4. To look up a key e_i in x BFs, k different bit positions are selected from a column using k different hash functions h_1, ..., h_k. For each of the k bit positions selected, it accesses a row of the length of x bits and executes bit-wise AND operations on k rows. The 1's position in the resultant row of bits indicates the BF where the key was found. If the resultant row of bits is all zero, according to the BF property, the key e_i is not present in any of the x BFs. However, if the resultant row has a single 1, the flash page associated with the BF of the 1's position should be read to search the key from the set of KV pairs contained in the flash page. As an example, Figure 4 shows that the key e_i is found in BF 2. Still, a chance exists that no KV pairs have the key (called false positive error). Possibly, the resultant row can have multiple 1's, requiring to read and check more than one flash page (from the largest BF label, implying the most recently written BFs) to find the key. Note that as the BF chain length increases, the number of false positive errors increases as well.

It is worth noting that the key lookup operation bearing such parallel lookup scheme is bottlenecked by the read throughput of the SSD, as the entire remainder of a BF chain (e.g., $x-1$ BFs of a BF chain containing x BFs) may need to be read from flash-memory for a parallel lookup. If the value x goes beyond the number of internal data channels/buses of the SSD, one possible way to boost the key lookup performance is to pipeline the BF chain retrieval process with the parallel lookup process. In the future, we plan to expand our BloomStore design with this feature.

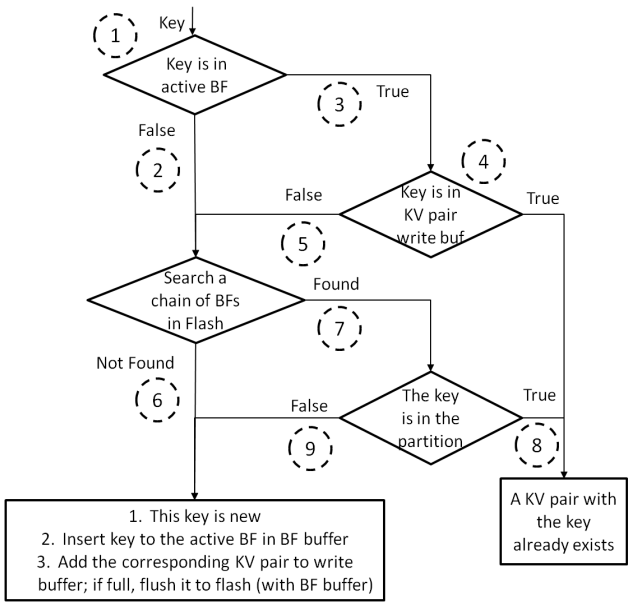

Figure 5. Flowchart of BloomStore operations: key lookup and KV pair insertion

B. KV Store Operations

To help to understand the relationship of different components in BloomStore, we demonstrate the operations of key lookup and KV pair insertion for a single BloomStore instance (see Figure 5). In the flowchart, we assume the same data deduplication application logic as in our KV store application; i.e., if a key (e.g., a chunk ID) lookup is answered negatively, a successive KV insertion will be triggered to add a corresponding KV pair (e.g., chunk metadata) to the store.

Key Lookup: To look up a key, (1) BloomStore instance looks up the key at the active BF (held in the BF buffer) in the associated BF chain; (2) if the key is not found in the active BF, it follows the in-RAM flash pointer to retrieve the remainder of the BF chain and performs the BFs parallel lookup; (3) if the key is found in the active BF, BloomStore proceeds to check the key in the KV pair write buffer; (4) if the key is in the write buffer, the key lookup operation returns an affirmative value; (5) if the key is not found in the write buffer, BloomStore retrieves the remainder of the BF chain and performs the BFs parallel lookup on all fetched BFs; (6) if a key is not found in any of the BFs in the BF chain, the key is considered new; (7) for each BF in the BF chain where the key was found, a flash pointer will be extracted and followed to search the key in its corresponding flash page containing the KV pairs; in the case where more than one flash page needs to be searched, BloomStore searches the pages by the reverse order of their write times (from the largest BF label); BloomStore then stops its lookup and returns affirmatively upon finding the first KV pair whose key matches the searched one; (8) if the key is found, the lookup operation returns an affirmative value; and (9) if the key is not found, the lookup operation returns a negative value.

KV Pair Insertion: This operation inserts/updates a KV pair into its KV pair write buffer and inserts its key to the

active BF associated with a BloomStore instance. When the KV pair write buffer becomes full, all KV pairs in the buffer are written to flash with a flash page write. In addition, the BloomStore instance updates its BF chain with aforementioned BF buffer flush operation to move (append) the currently active BF to the remainder of the BF chain and instantiate a new active BF in the BF buffer. This update operation requires the BloomStore instance to fetch the remainder of the BF chain from flash into RAM, append the active BF to the remainder, and write back the updated remainder to the flash. Moreover, if this insertion corresponds to an update operation on an earlier inserted key, the most recent value of the key will be (correctly) retrieved during a key lookup operation as the older value was stored closer to the head in the append-log.

KV Pair Deletion: A delete operation on a KV pair is supported by inserting a *null* value for the key (becoming a garbage KV pair). When the flash usage or a fraction of garbage KV pairs in the flash exceeds a predefined threshold, a garbage collection procedure (different from the garbage collection inside the flash-memory) begins to reclaim the flash space in a way similar to that in the log-structured file systems [17]. The garbage collection starts scanning the KV pairs from the earliest written flash pages (the head of the flash append-log). It discards the garbage KV pairs as well as the BFs and copy-forwards the valid ones (from the head to the tail of the flash log); the garbage collection stops when the fraction of garbage KV pairs decreases under a certain threshold value.

C. Design Enhancements:

Having enough RAM space larger than the minimum RAM usage, our design employs the additional RAM space (the available RAM space minus the minimum RAM usage) to enhance the efficiency of the BloomStore design.

Multi-BF buffering: For each BloomStore instance, its BF buffer can hold the active BF plus a number of BFs whose data flash pages of KV pairs have been already written into the flash. By buffering more than one BF in RAM, we can expect performance benefits for key lookup and insertion operations by reducing the number of flash page reads to fetch the associated BFs into the RAM and by decreasing the number of the BF buffer flushes into the flash.

Prefilter: A key lookup operation, as indicated in the flowchart of Figure 5, may generate unnecessary flash page reads caused by loading a BF chain and its associated KV pair data pages from flash when searching the keys that do not exist in the KV store (briefly non-existent keys). Upon a key lookup miss in the BF buffer, the remainder of the BF chain will be read into RAM to check for the key. Many real applications frequently issue lookup operations on the non-existent keys. For example, primary file system deduplicaiton (refer to the property of the *Vx* workload in Section IV) usually finds many new chunk IDs through the deduplication process. Microsoft LIVE Primetime on-line multi-player game [18] frequently looks up non-existent keys to implement the game logic [19]. These cause many unnecessary flash page reads because BloomStore reads the remainder of the BF chain and its associated data KV pair pages from the flash to look up the non-existent keys.

Therefore, for workloads that search many non-existent keys from the KV store, BloomStore maintains a fixed sized prefilter in RAM that may filter out many lookups for the non-existent keys before reading a BF chain from the flash. The rationale behind using a Bloom Filter structure as our prefilter is as follows: (1) the BF has a unique characteristic that it is free of false negative errors (i.e., if a key is not found in a BF, it is for sure a non-existent one), regardless of the size of the BF. Hence, we could freely adjust the size of the pre-filter according to the available RAM space without worrying about missing any keys; and (2) with a fairly small RAM footprint (e.g., 4 bits/key), the BF is able to identify and filter out a significant amount of the non-existent keys observed in the key lookup and insertion process.

Obviously, employing a bigger prefilter will filter out a larger portion of lookups for the non-existent keys, owing to the less incurred false positive errors.

More RAM space consumed by the prefilter implies that less RAM space can be used for the Multi-BF buffering. This will make potentially more BF chains be retrieved. Therefore, BloomStore has to choose the prefilter size carefully in order to optimize the lookup throughput.

IV. Performance Evaluation

We compare our BloomStore with BufferHash and SkimpyStash by using realistic workloads obtained from the data deduplication applications in terms of the following performance metrics: (1) amortized RAM overhead per KV pair and (2) key lookup/insertion throughput.

A. Experiment Setup

Table I
Properties of two data deduplication workloads

Workload name	Lookup & insert operations #	Lookup:insert ratio	Key/value size (byte)
Linux	$12,427,697$	$4.1:1$	$20/44$
vx	$14,628,873$	$1.6:1$	$20/44$

The BufferHash is excluded from throughput comparisons, because the RAM overhead analysis shows that its amortized RAM overhead per KV pair is ten times higher than the others. We also rule out SILT from our comparison, since it is not a suitable KV store design to meet the high insertion/update throughput of in-line data deduplication.

We implement BloomStore and SkimpyStash by using Python. MurmurHash [20] is adopted to realize the hash functions used in our BF implementation (with different seeds) and to compute a hash table index entry in SkimpyStash. Both BloomStore and SkimpyStash are built on top of a raw block device interface, implying there are no file-system related effects, such as buffering, caching, and prefetch.

We run the implemented BloomStore and SkimpyStash on a typical server having Intel Xeon L5530 Quad Core 2.4GHz

and Linux kernel 2.6.32 (ubuntu 10.04). The server is also attached with a prototype Micro 1TB NAND flash based SSD through a high-speed PCIe interface. The physical flash page size of the SSD is 4KB. In addition, to study the performance of our design on a SATA interfaced SSD, we run both BloomStore and SkimpyStash with an INTEL X25E 32GB SATA interfaced NAND flash based SSD.

We use two real-world data deduplication workloads: *Linux* and *Vx* workloads. *Linux* is a typical data dedup workload collected from a data backup environment. This workload was obtained from the Linux kernel source backups which consist of 20 different versions of the Linux kernel source distributions (Linux-2.6.30.1 – Linux-2.6.30.10, Linux-2.6.35.1 – Linux-2.6.35.8, Linux-2.6.36.1, and Linux-2.6.36.2). As with typical data backup streams, this workload contains many duplicates that will exhibit a higher lookup/insertion ratio. The *Vx* stands for another breed of data dedup workload that was obtained from a primary file system environment. We investigate BloomStore performance on *Vx*, because the data deduplication on the primary file systems has recently drawn increasing attention from the data storage community. For example, Meyer et al. [21] studied the data dedup performance on a collection of 857 file systems running Windows at Microsoft. The *Vx* workload was collected by crawling a networked primary file system shared by a group of software engineers. Below are the statistics for the two workloads: (1) the *Linux* workload contains $10,000,000$ total chunks and $2,427,697$ unique chunks, while the *Vx* workload contains $9,000,000$ total chunks and $5,628,873$ unique chunks; and (2) the ratios of the key lookup operations to KV pair insertion operations in the *Linux* and *Vx* workloads are 4.1:1 and 1.6:1, respectively. The *Linux* workload has a higher key lookup/insertion ratio than the *Vx* workload. In addition, the *Vx* workload looks up *many non-existent keys* in the KV store, while the *Linux* workload searches *many already existent keys* due to its high data duplication. The properties of the two workloads are summarized in Table I. In these dedup applications, each key is a 20-byte SHA-1 hash value of the corresponding data chunk, while the value is a 44-byte metadata for the chunk. Thus, the size of each KV pair for the two workloads is 64 bytes.

B. Amortized RAM Overhead

We analyze the amortized RAM overhead per KV pair for BufferHash, SkimpyStash, and BloomStore. For this analysis, we assume that the flash page size is 4KB and adopt the commonly used 64-bytes KV pair length for data deduplication applications [4], [10], [22]. The 64-byte KV pair combines a 20-byte key (a SHA1 hash as a chunk ID) and a 44-byte value (chunk metadata for encoding chunk location, chunk size and chunk offset, etc.).

BufferHash (10 bytes/key): With a 16-byte KV pair length and a maximum number of 16 hash tables per partition, BufferHash consumes a 4-byte amortized RAM overhead per KV pair [9]. The amortized RAM overhead grows linearly as the KV pair length increases; i.e., the 64-byte KV pair length increases the amortized RAM overhead per KV to 10 bytes/key.

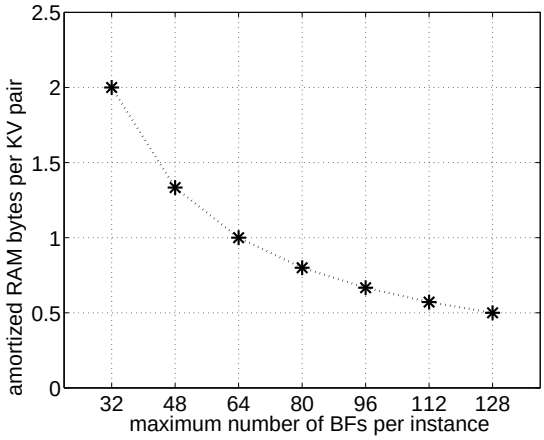

Figure 6. Amortized RAM overhead per KV pair for BloomStore as the number of BFs per key-range partition (BF chain length) increases

SkimpyStash (practical range: $(1, 4]$ bytes/key): To mitigate the impact of false positive errors on key lookup throughput, SkimpyStash pays a relatively high RAM overhead; i.e., 1-byte RAM footprint per key represented in a BF. The amortized RAM overhead per KV pair can be computed as follows: $1 + \frac{pointer\ \ size}{average\, bucket\, length}$, where the pointer size is set to 4 bytes [10]. Its smallest amortized RAM overhead per KV pair in theory approaches 1 byte/key, as the average bucket length becomes extremely large. In our experiments, the smallest amortized RAM overhead was 1.2 bytes/key.

BloomStore (practical range: $[0.5, 1]$): BloomStore design utilizes the RAM space for two purposes: (1) buffering the inserted KV pairs and (2) buffering the BFs for the key insertions. The amortized RAM overhead for the first is usually several times bigger than that for the second. With the (maximum) BF chain length per BloomStore instance of 96, each BloomStore instance could store up to $\frac{4,096}{64} \times 96 = 6144$ KV pairs. Then, the amortized RAM space overhead of the write buffer per KV pair becomes $4,096/6,144 = 0.667$ byte/key in a fully filled BloomStore instance. The amortized RAM overhead of the BF buffer per KV pair can be safely omitted (assuming the minimum size case, where there is only one active BF buffered in RAM for each instance and the BF size is of 64 bytes) because it is tens of times smaller than that of the KV pair write buffer (e.g., assuming a minimum BF size of 64 bytes and only one active BF buffered in RAM for each instance: $\frac{64}{64\times 96} = 0.01$ byte/key vs. 0.667 byte/key). Figure 6 illustrates that as the H (BF chain length) increases, the amortized RAM space overhead per KV pair decreases non-linearly. Notice that with $H = 128$, the amortized RAM space overhead becomes 0.5 bytes/key.

A larger BF chain length H increases the number of KV pairs stored in each instance. This result in a smaller number of BloomStore instances. It eventually helps to reduce the RAM usage consumed by the KV pair write buffers for all instances. However, as H increases linearly, the number of false positive errors increases dramatically. Each of false

positive errors may trigger an extra flash data page read. In our experiments, we vary the BF chain length H to 64, 96, and 128 BFs to study its effect on the performance. Based on the results of both workloads *Linux* and *Vx*, we observe that BloomStore configured with $H = 96$ yields the highest key lookup throughput.

With the obtained BF chain length ($H = 96$), we further investigate the impact of BF size m. We find out that the best BF size is 64 bytes for *Vx* workload and 128 bytes for *Linux* workload.

C. Effectiveness of Prefilter

Table II
Linux WORKLOAD: RAM USAGE (KB) DECOMPOSITION FOR DIFFERENT CONFIGURATIONS

RAM usage decomposition	BF buffer	prefilter overhead	KV pair write buffer
base	1,302	0	1,648
base+prefilter	807	495	1,648

Table III
Vx WORKLOAD: RAM USAGE (KB) DECOMPOSITION FOR DIFFERENT CONFIGURATIONS

RAM usage decomposition	BF buffer	prefilter overhead	KV pair write buffer
base	3,066	0	3,840
base+prefilter	186	2,880	3,840

Either *base* or *base+prefilter* configuration enables multi-BF buffering. For fairness in comparing the BloomFilter (*base*) and the BloomFilter with the prefilter (*base+prefilter*), the *base+prefilter* configuration is given a smaller amount of BF buffer space to compensate for the prefilter RAM usage. Table II and III present the RAM usage decomposition of different components (BF buffer and prefilter) with respect to the two different configurations for both workloads. For the two workloads, the amortized RAM overhead per KV pair is 1.2 bytes. Figure 7 and 8 illustrate the key lookup throughputs for the *Vx* and *Linux* workloads, respectively. In the figures, each group of three bars represents the number of the BF chain reads that represents how many times the remainder of the BF chain in the flash is fetched into RAM (the lower the better), the number of the (KV pair) data page reads (the lower the better), and the key lookup throughput (the higher the better). We omit presenting the insertion throughput, because the prefilter enhancement is only for improving the key lookup throughput. Notice that the improvement contributed by the prefilter is remarkable in the *Vx* workload. The reason is that a major fraction of the lookup operations is used to check non-existent keys in the workload, which could be avoided by the prefilter. On the other hand, for the *Linux* workload, the prefilter does not help significantly because a major portion of the lookup operations is spent to check those already-existent inserted keys.

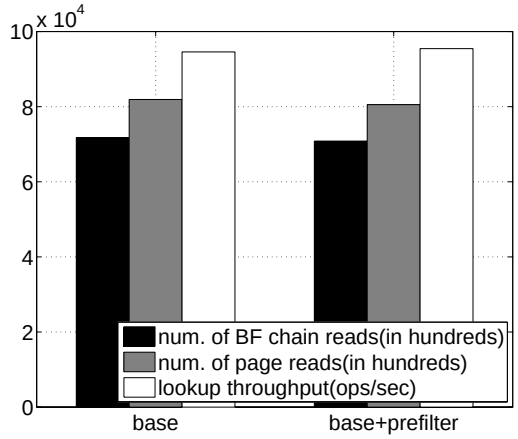

Figure 7. *Linux* workload: Impact of prefiltering on key lookup throughput (ops/sec)

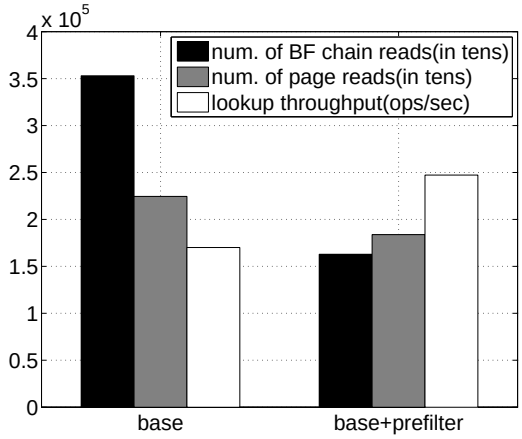

Figure 8. *Vx* workload: Impact of pre-filtering on key lookup throughput (ops/sec)

D. Key Lookup & Insertion Throughput

Figure 9 and 10 present the throughput comparisons for BloomStore and SkimpyStash over the *Linux* and *Vx* workloads, respectively. The available RAM space is the same for both designs. For fairness, the additional RAM space required by the extra feature of BloomStore (prefilter) is consumed in the given RAM space. In particular, Figure 9 illustrates the results on two different types of SSDs: the upper two curves plot the results obtained from the Micro PCIe SSD, while the lower two curves plot the results obtained from the INTEL SATA SSD. We omit the results of both designs with the INTEL 32GB X25E on the *Vx* workload because they follow the same trend. The BloomStore is configured for the two different workloads as follows: (1) for the *Linux* workload, $H = 96$, $m = 128$, and no prefilter (the unused space is used for more BF buffering); (2) for the *Vx* workload, $H = 96$, $m = 64$, and with a prefilter.

The following observations are made from the figures. First, both BloomStore and SkimpyStash achieve significantly higher key lookup throughput on the *Vx* workload than on the *Linux* workload. One reason behind this is that the lookup ratio in the *Linux* is much higher than that in the *Vx* (4.1 vs. 1.6, a factor of 2.56 times higher). Secondly, BloomStore delivers

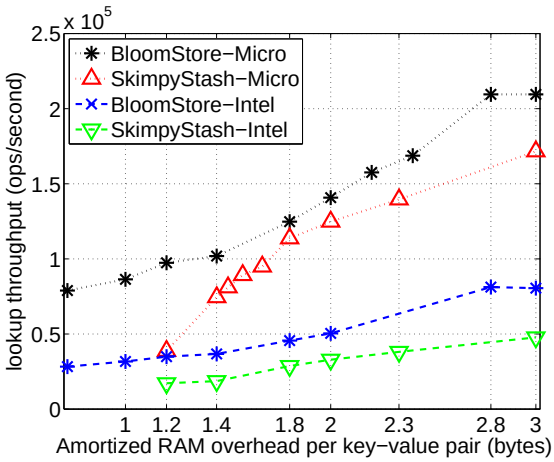

Figure 9. *Linux* workload: Key lookup throughput (ops/sec) comparisons between BloomStore and SkimpyStash as the amortized RAM overhead per KV pair varies

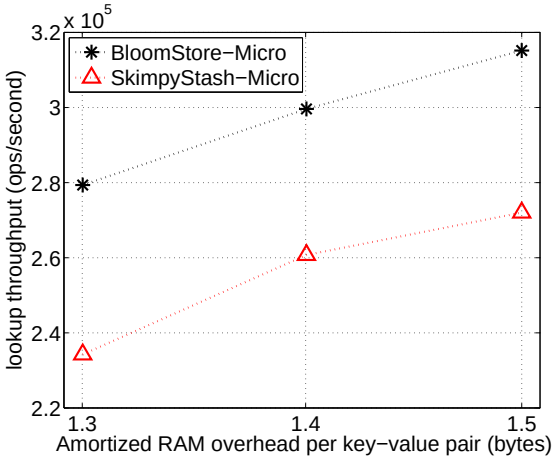

Figure 10. *Vx* workload: Key lookup throughput (ops/sec) comparisons between BloomStore and SkimpyStash as the amortized RAM overhead per KV pair varies

a higher key lookup throughput than SkimpyStash. Thirdly, as the available RAM space declines, the lookup throughput drops in both designs. However, the lookup throughput of SkimpyStash drops slightly faster than that of BloomStore. In particular, on the *Linux* workload, as the amortized RAM overhead per KV pair decreases from 1.8 bytes to 1.2 bytes, the lookup throughput of SkimpyStash descends by 2.9 times, which is much steeper than BloomStore. It is worth noting that as the amortized RAM overhead per KV pair grows, the lookup throughput for BloomStore increases monotonically and saturates after reaching a 2.8-byte amortized RAM overhead per KV pair (see Figure 9). The reason behind this is that from this point, all the BF chains are completely buffered in the RAM. Only the number of data page reads affects the lookup throughput. In fact, for the *Linux* workload, if more RAM space is allowed beyond the 2.8 bytes/key, one approach to further improve the lookup throughput is to increase the BF size, so as to minimize the wasted flash page reads. To minimize the impact of false positive errors on the key lookup throughput, SkimpyStash pays relatively high RAM overhead for maintaining BFs for each bucket in RAM, e.g., 1-byte RAM footprint per key. Therefore, there are no results obtained from SkimpyStash corresponding to the ≤ 1 amortized RAM overhead per KV pair. On the contrary, BloomStore could easily achieve reasonably a good lookup throughput with the sub-byte range RAM usage per KV pair. As shown in Figure 9, even with 0.72-byte amortized RAM overhead per KV pair, BloomStore is still able to deliver $78,879$ ops/second key lookup throughput, only 22.5% lower than what is achieved by the doubled (1.44 bytes) amortized RAM overhead.

As for the insertion performance with the two workloads, BloomStore performs slightly lower (1.5% and 1.4% for *Linux* and *Vx* than SkimpyStash, which can be explained by the extra overhead of the periodical BF buffer flush operations (i.e., updating the respective chain of BFs on the flash by appending all buffered BFs to the existing BF chain in flash) performed in BloomStore.

Similarly, with the INTEL SATA SSD, BloomStore achieves a uniformly higher key lookup throughput than SkimpyStash for all amortized RAM overheads per KV pair. The lookup throughput results for both BloomStore and SkimpyStash are universally higher on the Micro PCIe SSD than on the INTEL SATA SSD. For example, on the *Linux* workload, BloomStore achieves approximately 2.6–2.8 times higher lookup throughput, while SkimpyStash accomplishes 2.25–4 times higher lookup throughput on the Micro PCIe SSD than on the INTEL SATA SSD.

V. Conclusions

In this paper, we designed a flash-based KV store architecture called BloomStore that not only assures an extremely low amortized RAM overhead per KV pair (by keeping a flash-page sized data buffer and a very small sized BF buffer per BloomStore instance in RAM), but also achieves a high lookup/insertion throughput (by reducing the maximum number of flash page reads with key-range partitioning; by buffering multiple BFs per BloomStore instance in RAM to reduce the BF-containing flash page reads and writes). BloomStore design also employs a prefilter to avoid many unnecessary flash page reads for looking up the non-existent keys. Throughput comparisons on the two real-world workloads in data deduplication applications illustrated that BloomStore design achieved a significantly better key lookup throughput and roughly the same insertion performance with much lower RAM usage – BloomStore provides the same lookup throughput ($78,879$ ops/second in the *Linux* workload), while consuming 22.5% lower amortized RAM overhead per KV pair, as compared with SkimpyStash.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments. This work was partially supported by grants from NSF (NSF Awards: 0960833, 0934396 and 1115471) and by the Ministry of Knowledge Economy, Korea, under the ITRC (Information Technology Research

Center) support program (NIPA-2012-H0301-12-3002) supervised by the National IT IndustryPromotion Agency.

REFERENCES

[1] B. Zhu, K. Li, and P. Hugo, "Avoiding the disk bottleneck in the data domain deduplication file system," in *In Proceedings of the 6th USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2008.

[2] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[3] "Riverbed SteelHead Product Family: http://www.riverbed.com."

[4] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: speeding up inline storage deduplication using flash memory," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. USENIX Association, 2010, pp. 16–16.

[5] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar, "MicroHash: An efficient index structure for flash-based sensor devices," in *FAST*. USENIX, 2005.

[6] C.-H. Wu, T.-W. Kuo, and L.-P. Chang, "An efficient b-tree layer implementation for flash-memory storage systems," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, 2007.

[7] S. Nath and A. Kansal, "FlashDB: dynamic self-tuning database for nand flash," in *IPSN*, T. F. Abdelzaher, L. J. Guibas, and M. Welsh, Eds. ACM, 2007, pp. 410–419.

[8] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: a fast array of wimpy nodes," in *SOSP*, J. N. Matthews and T. E. Anderson, Eds. ACM, 2009, pp. 1–14.

[9] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath, "Cheap and large cams for high performance data-intensive networked systems," in *NSDI*. USENIX Association, 2010, pp. 433–448.

[10] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM space skimpy key-value store on flash-based storage," in *SIGMOD Conference*, T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, Eds. ACM, 2011, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989327

[11] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: a memory-efficient, high-performance key-value store," in *SOSP*, T. Wobber and P. Druschel, Eds. ACM, 2011, pp. 1–13.

[12] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, pp. 422–426, July 1970. [Online]. Available: http://doi.acm.org/10.1145/362686.362692

[13] Pagh and Rodler, "Cuckoo Hashing," *ALGORITHMS: Journal of Algorithms*, vol. 51, 2004.

[14] S. Chen, "FlashLogging: exploiting flash devices for synchronous logging performance," in *SIGMOD Conference*. ACM, 2009, pp. 73–86. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559855

[15] "EMC Data Domain DD800 Series: http://www.datadomain.com."

[16] S. Kumar and P. Crowley, "Segmented Hash: an efficient hash table implementation for high performance networking subsystems," in *ANCS*, A. D. Berenbaum, K. Li, and J. S. Turner, Eds. ACM, 2005, pp. 91–103. [Online]. Available: http://doi.acm.org/10.1145/1095890.1095904

[17] M. Rosenblum and J. Ousterhout, "The design and implementation of a log-structured file system," in *sosp*, Oct. 1991, pp. 1–15.

[18] "Xbox LIVE 1 vs 100 game: http://www.xbox.com." [Online]. Available: http://en.wikipedia.org/wiki/1_vs._100_(Xbox_360)

[19] B. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H. Du, "BloomFlash: Bloom filter on flash-based storage," in *Proceedings of the 31th International Conference on Distributed Computing Systems*, ser. ICDCS 2011, 2011.

[20] "MurmurHash Function: http://en.wikipedia.org/wiki/murmurhash." [Online]. Available: http://en.wikipedia.org/wiki/MurmurHash

[21] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," in *FAST*, G. R. Ganger and J. Wilkes, Eds. USENIX, 2011, pp. 1–13. [Online]. Available: http://www.usenix.org/events/fast11/tech/techAbstracts.html#Meyer

[22] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High Throughput Persistent Key-Value Store," in *VLDB*, 2010.