SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage

Biplob Debnath*,1 Sudipta Sengupta‡ Jin Li ‡
‡ Microsoft Research, Redmond, WA, USA
*EMC Corporation, Santa Clara, CA, USA

ABSTRACT

We present SkimpyStash, a RAM space skimpy key-value store on flash-based storage, designed for high throughput, low latency server applications. The distinguishing feature of SkimpyStash is the design goal of extremely low RAM footprint at about 1 (± 0.5) byte per key-value pair, which is more aggressive than earlier designs. SkimpyStash uses a hash table directory in RAM to index key-value pairs stored in a log-structured manner on flash. To break the barrier of a flash pointer (say, 4 bytes) worth of RAM overhead per key, it “moves" most of the pointers that locate each key-value pair from RAM to flash itself. This is realized by (i) resolving hash table collisions using linear chaining, where multiple keys that resolve (collide) to the same hash table bucket are chained in a linked list, and (ii) storing the linked lists on flash itself with a pointer in each hash table bucket in RAM pointing to the beginning record of the chain on flash, hence incurring multiple flash reads per lookup. Two further techniques are used to improve performance: (iii) two-choice based load balancing to reduce wide variation in bucket sizes (hence, chain lengths and associated lookup times), and a bloom filter in each hash table directory slot in RAM to disambiguate the choice during lookup, and (iv) compaction procedure to pack bucket chain records contiguously onto flash pages so as to reduce flash reads during lookup. The average bucket size is the critical design parameter that serves as a powerful knob for making a continuum of tradeoffs between low RAM usage and low lookup latencies. Our evaluations on commodity server platforms with real-world data center applications show that SkimpyStash provides throughputs from few 10,000s to upwards of 100,000 `get-set` operations/sec.

Categories and Subject Descriptors

H.3 Information Storage and Retrieval [**H.3.1 Content Analysis and Indexing**]: Indexing Methods

General Terms

Algorithms, Design, Experimentation, Performance.

[1]Work done while Biplob Debnath was at Microsoft Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

Keywords

Key-value store, Flash memory, Indexing, RAM space efficient index, Log-structured index.

1. INTRODUCTION

A broad range of server-side applications need an underlying, often persistent, key-value store to function. Examples include state maintenance in Internet applications like online multi-player gaming and inline storage deduplication (as described in Section 3). A high throughput persistent key-value store can help to improve the performance of such applications. Flash memory is a natural choice for such a store, providing persistency and 100-1000 times lower access times than hard disk. Compared to DRAM, flash access times are about 100 times higher. Flash stands in the middle between DRAM and disk also in terms of cost – it is 10x cheaper than DRAM, while 20x more expensive than disk – thus, making it an ideal gap filler between DRAM and disk.

It is only recently that flash memory, in the form of Solid State Drives (SSDs), is seeing widespread adoption in desktop and server applications. For example, MySpace.com recently switched from using hard disk drives in its servers to using PCI Express (PCIe) cards loaded with solid state flash chips as primary storage for its data center operations [5]. Also recently, Facebook released Flashcache, a simple write back persistent block cache designed to accelerate reads and writes from slower rotational media (hard disks) by caching data in SSDs [6]. These applications have different storage access patterns than typical consumer devices and pose new challenges to flash media to deliver sustained and high throughput (and low latency).

These challenges arising from new applications of flash are being addressed at different layers of the storage stack by flash device vendors and system builders, with the former focusing on techniques at the device driver software level and inside the device, and the latter driving innovation at the operating system and application layers. The work in this paper falls in the latter category. To get the maximum performance per dollar out of SSDs, it is necessary to use flash aware data structures and algorithms that work around constraints of flash media (e.g., avoid or reduce small random writes that not only have a higher latency but also reduce flash device lifetimes through increased page wearing). In the rest of the paper, we use NAND flash based SSDs as the architectural choice and simply refer to it as flash memory. We describe the internal architecture of SSDs in Section 2.

Recently, there are several interesting proposals to design key-value stores using flash memory [10, 11, 15, 16]. These designs use a combination RAM and flash memory – they store the full key-value pairs on flash memory and use a small amount of metadata per key-value pair in RAM to support faster insert and lookup operations. For example, FAWN [11], FlashStore [16], and ChunkStash [15] each require about six bytes of RAM space per key-value pair stored on flash memory. Thus, the amount of available RAM space limits the total number of key-value pairs that could be indexed on flash memory. As flash capacities are about an order of magnitude bigger than RAM and getting bigger, RAM size could well become the bottleneck for supporting large flash-based key-value stores. By reducing the amount of RAM bytes needed per key-value pair stored on flash down to the extreme lows of about a byte, SkimpyStash can help to scale key-value stores on flash on a lean RAM size budget when existing current designs run out.

Our Contributions

In this paper, we present the design and evaluation of SkimpyStash, a RAM space skimpy key-value store on flash-based Storage, designed for high throughput server applications. *The distinguishing feature of SkimpyStash is the design goal of extremely low RAM footprint at about* 1 *byte per key-value pair,* which is more aggressive than earlier designs like FAWN [11], BufferHash [10], ChunkStash [15], and FlashStore [16]. Our base design uses less than 1 byte in RAM per key-value pair and our enhanced design takes slightly more than 1 byte per key-value pair. By being RAM space frugal, SkimpyStash can accommodate larger flash drive capacities for storing and indexing key-value pairs.

Design Innovations: SkimpyStash uses a hash table directory in RAM to index key-value pairs stored in a log-structure on flash. To break the barrier of a flash pointer (say, 4 bytes) worth of RAM overhead per key, it "moves" most of the pointers that locate each key-value pair from RAM to flash itself. This is realized by

(i) Resolving hash table collisions using linear chaining, where multiple keys that resolve (collide) to the same hash table bucket are chained in a linked list, and

(ii) Storing the linked lists on flash itself with a pointer in each hash table bucket in RAM pointing to the beginning record of the chain on flash, hence incurring multiple flash reads per lookup.

Two further techniques are used to improve performance:

(iii) Two-choice based load balancing [12] to reduce wide variation in bucket sizes (hence, chain lengths and associated lookup times), and a bloom filter [13] in each hash table directory slot in RAM for summarizing the records in that bucket so that at most one bucket chain on flash needs to be searched during a lookup, and

(iv) Compaction procedure to pack bucket chain records contiguously onto flash pages so as to reduce flash reads during lookup.

The average bucket size is the critical design parameter that serves as a powerful knob for making a continuum of tradeoffs between low RAM usage and low lookup latencies.

Evaluation on data center server applications: SkimpyStash can be used as a high throughput persistent key-value storage layer for a broad range of server class applications. We use real-world data traces from two data center applications, namely, Xbox LIVE Primetime online multi-player game and inline storage deduplication, to drive and evaluate the design of SkimpyStash on commodity server platforms. SkimpyStash provides throughputs from few 10,000s to upwards of 100,000 `get-set` operations/sec on the evaluated applications.

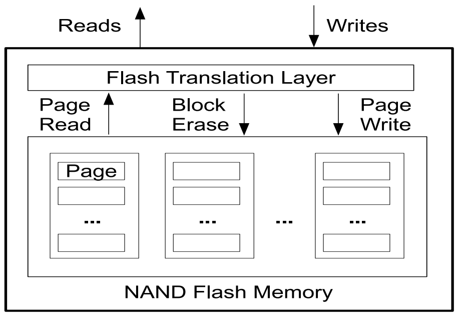


Figure 1: Internal architecture of a Solid State Drive (SSD)

The rest of the paper is organized as follows. We provide an overview of flash memory in Section 2. In Section 3, we describe two motivating real-world data center applications that can benefit from a high throughput key-value store and are used to evaluate SkimpyStash. We develop the design of SkimpyStash in Section 4. We evaluate SkimpyStash in Section 5. We review related work in Section 6. Finally, we conclude in Section 7.

2. FLASH MEMORY OVERVIEW

Figure 1 gives a block-diagram of an NAND flash based SSD. In flash memory, data is stored in an array of flash blocks. Each block spans 32-64 pages, where a page is the smallest unit of read and write operations. A distinguishing feature of flash memory is that read operations are very fast compared to magnetic disk drive. Moreover, unlike disks, random read operations are as fast as sequential read operations as there is no mechanical head movement. A major drawback of the flash memory is that it does not allow in-place updates (i.e., overwrite). Page write operations in a flash memory must be preceded by an erase operation and within a block, pages need be to written sequentially. The *in-place update* problem becomes complicated as write operations are performed in the page granularity, while erase operations are performed in the block granularity. The typical access latencies for read, write, and erase operations are 25 microseconds, 200 microseconds, and 1500 microseconds, respectively [9].

The Flash Translation layer (FTL) is an intermediate software layer inside SSD, which makes linear flash memory device act like a virtual disk. The FTL receives logical read and write commands from the applications and converts them to the internal flash memory commands. To emulate disk like in-place update operation for a logical page (L_p), the FTL writes data into a new physical page (P_p), maintains a mapping between logical pages and physical pages, and marks the previous physical location of L_p as invalid for future garbage collection. Although FTL allows current disk based application to use SSD without any modifications, it needs to internally deal with flash physical constraint of erasing a block before overwriting a page in that block. Besides the *in-place update* problem, flash memory exhibits another limitation – a flash block can only be erased for limited number of times (e.g., 10K-100K) [9]. FTL uses various wear leveling techniques to even out the erase counts of different blocks in the flash memory to increase its overall longevity [17]. Recent studies show that current FTL schemes are very effective for the workloads with sequential access write patterns. However, for the workloads with random access patterns, these schemes show very poor performance [18, 20, 22]. One of the design goals of SkimpyStash is to use flash memory in FTL-friendly manner.

3. KEY-VALUE STORE APPLICATIONS

We describe two real-world applications that can use SkimpyStash as an underlying persistent key-value store. Data traces obtained from real-world instances of these applications are used to drive and evaluate the design of SkimpyStash.

3.1 Online Multi-player Gaming

Online multi-player gaming technology allows people from geographically diverse regions around the globe to participate in the same game. The number of concurrent players in such a game could range from tens to hundreds of thousands and the number of concurrent game instances offered by a single online service could range from tens to hundreds. An important challenge in online multi-player gaming is the requirement to scale the number of users per game and the number of simultaneous game instances. At the core of this is the need to maintain server-side state so as to track player actions on each client machine and update global game states to make them visible to other players as quickly as possible. These functionalities map to `set` and `get` key operations performed by clients on server-side state. The real-time responsiveness of the game is, thus, critically dependent on the response time and throughput of these operations.

There is also the requirement to store server-side game state in a persistent manner for (at least) the following reasons: (i) resume game from interrupted state if and when crashes occur, (ii) offline analysis of game popularity, progression, and dynamics with the objective of improving the game, and (iii) verification of player actions for fairness when outcomes are associated with monetary rewards. We designed SkimpyStash to meet the high throughput and low latency requirement of such `get-set` key operations in online multi-player gaming.

3.2 Storage Deduplication

Deduplication is a recent trend in storage backup systems that eliminates redundancy of data across full and incremental backup data sets [28]. It works by splitting files into multiple chunks using a content-aware chunking algorithm like Rabin fingerprinting and using SHA-1 hash [24] signatures for each chunk to determine whether two chunks contain identical data [28]. In *inline* storage deduplication systems, the chunks (or their hashes) arrive one-at-a-time at the deduplication server from client systems. The server needs to lookup each chunk hash in an index it maintains for all chunk hashes seen so far for that backup location instance – if there is a match, the incoming chunk contains redundant data and can be deduplicated; if not, the (new) chunk hash needs to be inserted into the index.

Because storage systems currently need to scale to tens of terabytes to petabytes of data volume, the chunk hash index is too big to fit in RAM, hence it is stored on hard disk. Index operations are thus throughput limited by expensive disk seek operations. Since backups need to be completed over windows of half-a-day or so (e.g., nights and weekends), it is desirable to obtain high throughput in inline storage deduplication systems. RAM prefetching and bloom-filter based techniques used by Zhu et al. [28] can avoid disk I/Os on close to 99% of the index lookups. Even at this reduced rate, an index lookup going to disk contributes about 0.1msec to the *average* lookup time – this is about 10^3 times slower than a lookup hitting in RAM. SkimpyStash can be used as the chunk hash index for inline deduplication systems. By reducing the penalty of index lookup misses in RAM by orders of magnitude by serving such lookups from flash memory, SkimpyStash can help to increase deduplication throughput.

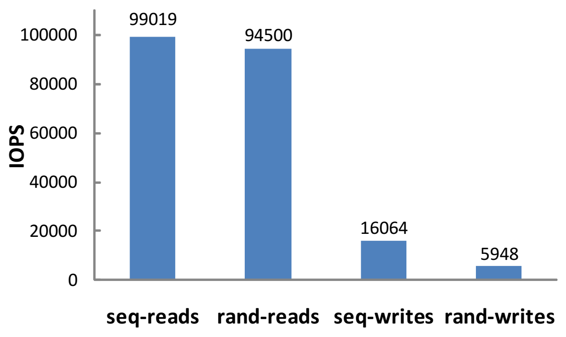


Figure 2: IOPS for sequential/random reads and writes using 4KB I/O request size on a 160GB fusionIO drive.

4. SkimpyStash DESIGN

We present the system architecture of SkimpyStash and the rationale behind some design choices in this section.

4.1 Coping with Flash Constraints

The design is driven by the need to work around two types of operations that are not efficient on flash media, namely:

1. **Random Writes:** Small random writes effectively need to update data portions within pages. Since a (physical) flash page cannot be updated in place, a new (physical) page will need to be allocated and the unmodified portion of the data on the page needs to be relocated to the new page.

2. **Writes less than flash page size:** Since a page is the smallest unit of write on flash, writing an amount less than a page renders the rest of the (physical) page wasted – any subsequent append to that partially written (logical) page will need copying of existing data and writing to a new (physical) page.

To validate the performance gap between sequential and random writes on flash, we used Iometer [3], a widely used performance evaluation tool in the storage community, on a 160GB fusionIO SSD [2] attached over PCIe bus to an Intel Core 2 Duo E6850 3GHz CPU. The number of worker threads was fixed at 8 and the number of outstanding I/Os for the drive at 64. The results for IOPS (I/O operations per sec) on 4KB I/O request sizes are summarized in Figure 2. Each test was run for 1 hour. The IOPS performance of sequential writes is about 3x that of random writes and worsens when the tests are run for longer durations (due to accumulating device garbage collection overheads). We also observe that the IOPS performance of (random/sequential) reads is about 6x that sequential writes. (The slight gap between IOPS performance of sequential and random reads is possibly due to prefetching inside the device.)

Given the above, the most efficient way to write flash is to simply use it as an append log, where an append operation involves a flash page worth of data, typically 2KB or 4KB. This is the main constraint that drives the rest of our key-value store design. Flash has been used in a log-structured manner and its benefits reported in earlier works [11, 14, 15, 16, 19, 23, 27].

4.2 Design Goals

The design of SkimpyStash is driven by the following guiding principles:

- **Support low-latency, high throughput operations.** This requirement is extracted from the needs of many server class applications that need an underlying key-value store to function. Two motivating applications that are used for evaluating SkimpyStash are described in Section 3.

- **Use flash aware data structures and algorithms.** This principle accommodates the constraints of the flash device so as to extract maximum performance out of it. Random writes and in-place updates are expensive on flash memory, hence must be reduced or avoided. Sequential writes should be used to the extent possible and the fast nature of random/sequential reads should be exploited.

- **Low RAM footprint per key independent of key-value size.** The goal here is to index all key-value pairs on flash in a RAM space efficient manner and make them accessible using a small number of flash reads per lookup. By being RAM space frugal, one can accommodate larger flash drive capacities and correspondingly larger number of key-value pairs stored in it. Key-value pairs can be arbitrarily large but the RAM footprint per key should be independent of it and small. We target a skimpy RAM usage of about 1 byte per key-value pair, a design point that is more aggressive than earlier designs like FAWN [11], BufferHash [10], ChunkStash [15], and FlashStore [16].

4.3 Architectural Components

SkimpyStash has the following main components. A base version of the design is shown in Figure 3 and an enhanced version in Figure 5. We will get to the details shortly.

RAM Write Buffer: This is a fixed-size data structure maintained in RAM that buffers key-value writes so that a write to flash happens only after there is enough data to fill a flash page (which is typically 2KB or 4KB in size). To provide strict durability guarantees, writes can also happen to flash when a configurable timeout interval (e.g., 1 msec) has expired (during which period multiple key-value pairs are collected in the buffer). The client call returns only after the write buffer is flushed to flash. The RAM write buffer is sized to 2-3 times the flash page size so that key-value writes can still go through when part of the buffer is being written to flash.

RAM Hash Table (HT) Directory: The directory structure, for key-value pairs stored on flash, is maintained in RAM and is organized as a hash table with each slot containing a pointer to a chain of records on flash. Each key-value pair record on flash contains, in addition to the key and value fields, a pointer to the next record (in the order in its respective chain) on flash. The chain of records on flash pointed to by each slot comprises the bucket of records corresponding to this slot in the HT directory. This is illustrated in Figure 3. The average number of records in a bucket, k, is a configurable parameter. In summary, we resolve hash table directory collisions by linear chaining and store the chains in flash.

In an enhancement of the design, we use two-choice based load balancing to reduce wide variation in bucket sizes (hence, chain lengths and associated lookup times), and introduce a bloom filter in each hash table directory slot in RAM for summarizing the records in that bucket so that at most one bucket chain on flash needs to be searched during a lookup. These enhancements form the the core of our design and are discussed in detail in Section 4.5.

Flash store: The flash store provides persistent storage for the key-value pairs and is organized as a *circular* append log. Key-value pairs are written to flash in units of a page size to the tail of the log. When the log accumulates garbage (consisting of deleted or

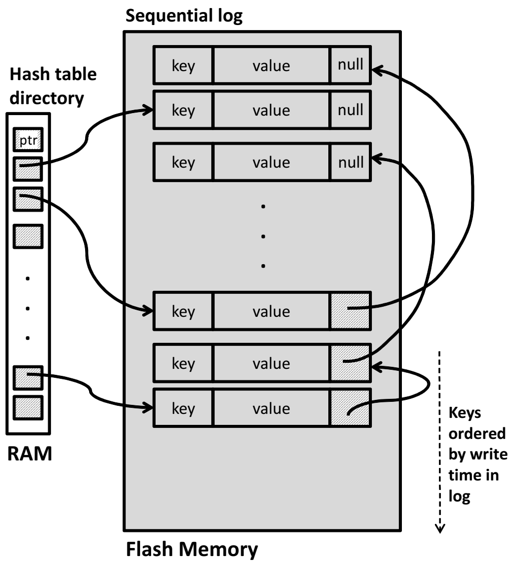


Figure 3: **SkimpyStash architecture showing the sequential log organization of key-value pair records on flash and base design for the hash table directory in RAM. (RAM write buffer is not shown.)**

older values of updated records) beyond a configurable threshold, the pages on flash from the head of the log are recycled – valid entries from the head of the log are written back to the end of the log. This also helps to place the records in a given bucket contiguously on flash and improve read performance, as we elaborate shortly. Each key-value pair record on flash contains, in addition to the key and value fields, a pointer to the next record (in the order in its HT bucket chain) on flash.

4.4 Overview of Key Lookup and Insert Operations

To understand the relationship of the different storage areas in our design, it is helpful to follow the sequence of accesses in key insert and lookup operations performed by the client application.

A key ***lookup*** operation (`get`) first looks up the RAM write buffer. Upon a miss there, it lookups up the HT directory in RAM and searches the chained key-value pair records on flash in the respective bucket.

A key ***insert*** (or, ***update***) operation (`set`) writes the key-value pair into the RAM write buffer. When there are enough key-value pairs in RAM write buffer to fill a flash page (or, a configurable timeout interval since the client call has expired, say 1 msec), these entries are written to flash and inserted into the RAM HT directory and flash.

A ***delete*** operation on a key is supported through insertion of a `null` value for that key. Eventually the *null* entry and earlier inserted values of the key on flash will be garbage collected.

When flash usage and fraction of garbage records in the flash log exceed a certain threshold, a garbage collection (and compaction) operation is initiated to reclaim storage on flash in a manner similar to that in log-structured file systems [26]. This garbage collection operation starts scanning key-value pairs from the (current) head of the log – it discards garbage (invalid or orphaned, as defined later) key-value pair records and moves valid key-value pair records from the head to the tail of the log. It stops when floor thresholds are reached for flash usage or fraction of garbage records remaining in the flash log.

The functionalities of (i) client key lookup/insert operations, (ii)

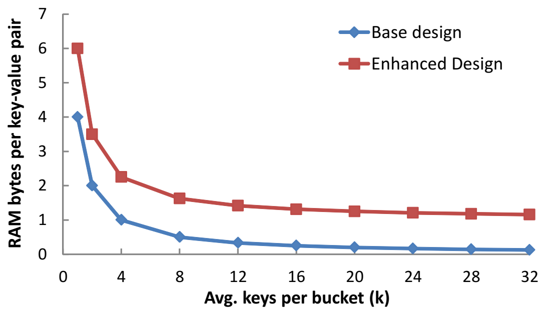


Figure 4: **RAM space usage per key-value pair in SkimpyStash for the base and enhanced designs as the average number of keys per bucket (k) is varied.**

writing key-value pairs to flash store and updating RAM HT directory, and (iii) reclaiming space on flash pages are handled by separate threads in a multi-threaded architecture. Concurrency issues with shared data structures arise in our multi-threaded design, which we address but do not describe here due to lack of space.

4.5 Hash Table Directory Design

At a high level, we use a hash table based index in RAM to index the key-value pairs on flash. Earlier designs like FAWN [11] and ChunkStash [15] dedicate one entry of the hash table to point to a single key-value pair on flash together with a checksum that helps to avoid (with high probability) following the flash pointer to compare keys for every entry searched in the hash table during a lookup. The RAM overhead in FAWN and ChunkStash is 6 bytes per key-value pair stored on flash. With such a design, we cannot get below the barrier of a flash pointer (say, 4 bytes) worth of RAM overhead per key-value pair (even if we ignore the other fields, like checksums, in the hash table entry).

Our approach, at an intuitive level, is to move most of the pointers that locate each key-value pair from RAM to flash itself. We realize this by

- Resolving hash table collisions using *linear chaining*, where multiple keys that resolve (collide) to the same hash table bucket are chained in a linked list, and
- Storing the linked lists on flash itself with a pointer in each hash table bucket in RAM pointing to the beginning record of the chain on flash. Each key-value pair record on flash contains, in addition to the key and value fields, a pointer to the next record (in the order in its respective chain) on flash.

Because we store the chain of key-value pairs in each bucket on flash, we incur multiple flash reads upon lookup of a key in the store. This is the tradeoff that we need to make with lookup times in order to be able to skimp on RAM space overhead per key-value pair. We will see that the average number of keys in a bucket (k) is the critical parameter that allows us to make a continuum of tradeoffs between these two parameters – it serves as a powerful knob for reducing RAM space usage at the expense of increase in lookup times.

We first begin with the base design of our hash table based index in RAM. Thereafter, we motivate and introduce some enhancements to the design to improve performance.

Base Design

The directory structure, for key-value pairs stored on flash, is maintained in RAM and is organized as a hash table with each slot containing a pointer to a chain of records on flash, as shown in Figure 3. Each key-value pair record on flash contains, in addition to the key and value fields, a pointer to the next record (in the order in its respective chain) on flash. The chain of records on flash pointed to by each slot comprises the bucket of records corresponding to this slot in the HT directory. A hash function h is used to map keys to slots in the HT directory. The average number of records in a bucket, k, is a configurable parameter. Then, to accommodate up to some given number n key-value pairs, the number of slots required in the HT directory is about n/k. In summary, we resolve hash table directory collisions by linear chaining and store the chains in flash. We next describe the *lookup*, *insert/update*, and *delete* operations on this data structure.

A ***lookup*** operation on a key uses the hash function h to obtain the HT directory bucket that this key belongs to. It uses the pointer stored in that slot to follow the chain of records on flash to search the key; upon finding the first record in the chain whose key matches the search key, it returns the value. The number of flash reads for such a lookup is $k/2$ on the average, and at most the size of the bucket chain in the worst case.

An ***insert*** (or, ***update***) operation uses the hash function h to obtain the HT directory bucket that this key belongs to. Let a_1 be the address on flash of the first record in this chain (i.e., what the pointer in this slot points to). Then a record is created corresponding to the inserted (or, updated) key-value pair with its next-pointer field equal to a_1. This record is appended to the log on flash and its address on flash now becomes the value of the pointer in the respective slot in RAM. Effectively, this new record is inserted at the beginning of the chain corresponding to this bucket. Thus, if this insert operation corresponds to an update operation on an earlier inserted key, the most recent value of the key will be (correctly) read during a lookup operation (the old value being further down the chain and accumulating as garbage in the log).

A ***delete*** operation is same as the ***insert*** (or, ***update***) with `null` value for that key. Eventually the *null* entry on flash and old values of the key will be garbage collected in the log.

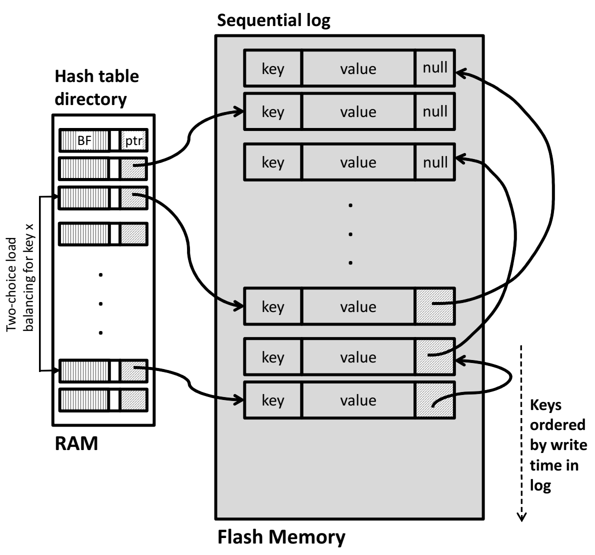
RAM Space Overhead for Base Design: Let us say that the pointer to flash in each HT directory slot is 4 bytes. (This accommodates up to 4GB of byte-addressable log. If records are of a fixed size, say 64 bytes, then this can accommodate up to 256GB of 64-byte granularity addressable log. Larger pointer sizes, up to 8 bytes, can be used according to application requirements.) Then, with a value of $k = 10$ average bucket size, the RAM space overhead is a mere $4/k = 0.4$ bytes $= 3.2$ bits per entry, independent of key-value size. At this sub-byte range, this design tests the extremes of low RAM space overhead per entry. The average number of flash reads per lookup is $k/2 = 5$; with current SSDs achieving flash read times in the range of 10μsec, this corresponds to a lookup latency of about 50 μsec. The parameter k provides a powerful knob for achieving tradeoffs between low RAM space usage and low lookup latencies. The RAM space usage per key-value pair for the base design as a function of k is shown in Figure 4.

Design Enhancements

We identify some performance inefficiencies in the base design and develop techniques to address them with only a slight increase in the RAM space overhead per key-value pair. The enhanced design is shown in Figure 5.

Load Balancing across Buckets

The hashing of keys to HT directory buckets may lead to skewed distributions in the number of keys in each bucket chain, thus creating variations in average lookup times across buckets. Thus, it



Sequential log
Hash table directory
BF
ptr
Two-choice load balancing for key x
RAM
key
value
null
Keys ordered by write time in log
Flash Memory

Figure 5: SkimpyStash architecture showing the sequential log organization of key-value pair records on flash and enhanced design for the hash table directory in RAM. (RAM write buffer is not shown.)

might be necessary to enforce fairly equal load balancing of keys across HT directory buckets in order to keep each bucket chain of about the same size. One simple way to achieve this is to use the *power of two choice idea* from [12] that has been applied to balance a distribution of balls thrown into bins. With a load balanced design for the HT directory, each key would be hashed to two candidate HT directory buckets, using two hash functions h_1 and h_2, and actually inserted into the one that has currently fewer elements. We investigate the impact of this design decision on balancing bucket sizes on our evaluation workloads in Section 5. To implement this load balancing idea, we add one byte of storage to each HT directory slot in RAM that holds the current number of keys in that bucket – this space allocation accommodates up to a maximum of $2^8 - 1 = 255$ keys per bucket.

Bloom Filter per Bucket

This design modification, in its current form, will lead to an increase in the number of flash reads during lookup. Since each key will need to be looked up in both of its candidate buckets, the worst case number of flash reads (hence lookup times) would double. To remove this latency impact on the lookup pathway, we add a *bloom filter* [13] per HT directory slot that summarizes the keys that have been inserted in the respective bucket. Note that this bloom filter in each HT directory slot can be sized to contain about k keys, since load balancing ensures that when the hash table reaches its budgeted full capacity, each bucket will contain not many more than k keys with very high probability. A standard rule of thumb for dimensioning a bloom filter to use one byte per key (which gives a false positive probability of 2%), hence the bloom filter in each HT directory slot can be of size k bytes.

The introduction of bloom filters in each HT directory slot has another desirable side effect – lookups on non-existent keys will almost always *not* involve any flash reads since the bloom filters in both candidate slots of the key will indicate that the key is not present (module false positive probabilities). (Note that in the base design, lookups on non-existent keys also lead to flash reads and involve traversing the entire chain in the respective bucket.)

In an interesting reciprocity of benefits, the bloom filters in each

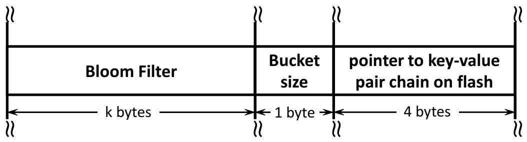


Figure 6: RAM hash table directory slot and sizes of component fields in the enhanced design of SkimpyStash. The parameter k is the average number of keys in a bucket.

bucket not only help in reducing lookup times when two-choice load balancing is used but also benefit from load balancing. Load balancing aims to keep the number of keys in each bucket upper bounded (roughly) by the parameter k. This helps to keep bloom filter false positive probabilities in that bucket bounded as per the dimensioned capacity of k keys. Without load balancing, many more than k keys could be inserted into a given bucket and this will increase the false positive rates of the respective bloom filter well beyond what it was dimensioned for.

The additional fields added to each HT directory slot in RAM in the enhanced design are shown in Figure 6.

During a ***lookup*** operation, the key is hashed to its two candidate HT directory buckets and the chain on flash is searched only if the respective bloom filter indicates that the key may be there in that bucket. Thus, accounting for bloom filter false positives, the chain on flash will be searched with no success in less than 2% of the lookups.

When an ***insert*** operation corresponds to an ***update*** of an earlier inserted key, the record is always inserted in the same bucket as the earlier one, even if the choice determined by load balancing (out of two candidate buckets) is the other bucket. If we followed the choice given by load balancing, the key may be inserted in the bloom filters of both candidate slots – this would not preserve the design goal of traversing at most one bucket chain (with high probability) on flash during lookups. Moreover, the same problem would arise with version resolution during lookups if different versions of a key are allowed to be inserted in both candidate buckets. This rule also leads to efficiencies during garbage collection operations since all the obsolete values of a key appear in the same bucket chain on flash. Note that this complication involving overriding of the load balancing based choice of insertion bucket can be avoided when the application does not perform updates to earlier inserted keys – one example of such an application is storage deduplication as described in Section 3.

In summary, in this enhancement of the base design, two-choice based load balancing strategy is used to reduce variations in the the number of keys assigned to each bucket (hence, chain lengths and associated lookup times). Each HT directory slot in RAM also contains a bloom filter summarizing the keys in the bucket and a size (count) field storing the current number of keys in that bucket.

RAM Space Overhead for Enhanced Design: With this design modification, the RAM space overhead per bucket now has three components, namely,

- Pointer to chain on flash (4 bytes),
- Bucket size (1 byte), and
- Bloom filter (k bytes).

This space overhead per HT directory slot is amortized over an average of k keys (in that bucket), hence the RAM space overhead per entry can be computed as $(k + 1 + 4)/k = 1 + 5/k$ which is about 1.5 bytes for $k = 10$. The average number of flash reads per lookup is still $k/2 = 5$ (with high probability); with current SSDs achieving flash read times in the range of 10μsec, this corresponds to a lookup latency of about 50 μsec. Moreover, the variation across lookup latencies for different keys is better controlled in this design (compared to the base design) as bucket chains are about the same size due to two choice based load balancing of keys across buckets. The RAM space usage per key-value pair for the enhanced design as a function of k is shown in Figure 4.

Storing key-value pairs to flash: Key-value pairs are organized on flash in a log-structure in the order of the respective write operations coming into the system. Each slot in the HT directory contains a pointer to the beginning of the chain on flash that represents the keys in that bucket. Each key-value pair record on flash contains, in addition to the key and value fields, a pointer to the next record (in the order in its respective chain) on flash. We use a 4-byte pointer, which is a combination of a page pointer and a page offset. The all-zero pointer is reserved for the `null` pointer – in the HT directory slot, this represents an empty bucket, while on flash this indicates that the respective record has no successor in the chain.

RAM and Flash Capacity Considerations: We designed our RAM indexing scheme to use 1 byte in RAM per key-value pair so as to maximize the amount of indexable storage on flash for a given RAM usage size. Whether RAM or flash capacity becomes the bottleneck for storing key-value pairs on flash depends further on the key-value pair size. With 64-byte key-value pair records, 1GB of RAM can index about 1 billion key-value pairs on flash which occupy 64GB on flash. This flash memory capacity is well within the capacity range of SSDs shipping in the market today (from 64GB to 640GB). On the other hand, with 1024-byte key-value pair records, the same 1GB of RAM can index 1 billion key-value pairs which now occupy 1TB on flash – at currently available SSD capacities, this will require multiple flash drives to store the dataset.

4.6 Flash Storage Management

Key-value pairs are organized on flash in a log-structure in the order of the respective write operations coming into the system. When there are enough key-value pairs in the RAM write buffer to fill a flash page (or, when a pre-specified coalesce time interval is reached), they are written to flash. The pages on flash are maintained *implicitly* as a circular log. Since the Flash Translation Layer (FTL) translates logical page numbers to physical ones, this circular log can be easily implemented as a contiguous block of logical page addresses with wraparound, realized by two page number variables, one for the first valid page (oldest written) and the other for the last valid page (most recently written). We next describe two maintenance operations on flash in SkimpyStash, namely, *compaction* and *garbage collection*. Compaction is helpful in improving lookup latencies by reducing number of flash reads when searching bucket. Garbage collection is necessary to reclaim storage on flash and is a consequence of flash being used in a log-structured manner.

Compaction to Reduce Flash Reads during Lookups

In SkimpyStash , a lookup in a HT directory bucket involves following the chain of key-value records on flash. For a chain length of c records in a bucket, this involves an average of $c/2$ flash reads. Over time, as keys are inserted into a bucket and earlier inserted keys are updated, the chain length on flash for this bucket keeps

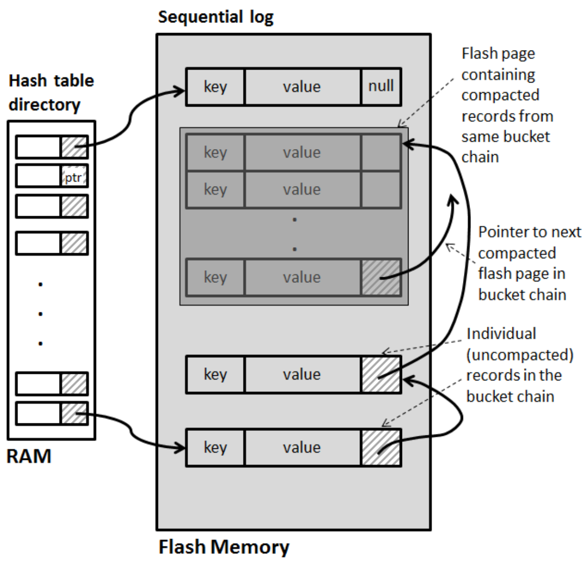

Sequential log
Hash table directory
RAM
ptr
key
value
null
Flash page containing compacted records from same bucket chain
Pointer to next compacted flash page in bucket chain
Individual (uncompacted) records in the bucket chain
Flash Memory

Figure 7: Diagram illustrating the effect of compaction procedure on the organization of a bucket chain on flash in SkimpyStash.

increasing and degrading lookup times. We address this by periodically *compacting* the chain on flash in a bucket by placing the valid keys in that chain contiguously on one or more flash pages that are appended to the tail of the log. Thus, if m key value pairs can be packed onto a single flash page (on the average), the number of flash reads required to search for a key in a bucket of k records is $k/(2m)$ on the average and at most $\lceil k/m \rceil$ in the worst case.

The compaction operations proceed over time in a bucket as follows. Initially, as key-value pairs are added at different times to a bucket, they appear on different flash pages and are chained together individually on flash. When enough valid records accumulate in a bucket to fill a flash page (say, m of them), they are compacted and appended on a new flash page at the tail of the log – the chain is now of a single flash page size and requires one flash read (instead of m) to search fully. Thereafter, as further keys are appended to a bucket, they will be chained together individually and appear before the compacted group of keys in the chain. Over time, enough new records may accumulate in the bucket to allow them to be compacted to a second flash page, and so the process repeats. At any given time, the chain on flash for each bucket now begins with a chained sequence of individual records followed by groups of compacted records (each group appearing on the same flash page). This organization of a bucket chain as a result of compaction is illustrated in Figure 7.

When a key-value pair size is relatively small, say 64 bytes as in the storage deduplication application, there may not be enough records in a bucket to fill a flash page, since this number is (roughly) upper bounded by the parameter k. In this case, we may reap the same benefits of compaction by applying the procedure to groups of chains in multiple buckets at a time.

The compaction operation, as described above, will lead to *orphaned*) (or, garbage) records on the flash log. Moreover, garbage records also accumulate in the log as a result of key update and delete operations. These need to be garbage collected on flash as we describe next.

Garbage Collection

Garbage records (holes) accumulate in the log as a result of compaction and key update/delete operations. These operations create garbage records corresponding to all previous versions of respective keys.

When a certain configurable fraction of garbage accumulates in the log (in terms of space occupied), a cleaning operation is performed to clean and compact the log. The cleaning operation considers currently used flash pages in *oldest first* order and deallocates them in a way similar to garbage collection in log-structured file systems [26]. One each page, the sequence of key-value pairs are scanned to determine whether they are valid or not. The classification of a key-value pair record on flash follows from doing a lookup on the respective key starting from the HT directory – if this record is the same as that returned by the lookup, then it is *valid*; if it appears later in the chain than a valid record for that key, then this record is *invalid* and corresponds to an obsolete version of they key; otherwise, the record is *orphaned* and cannot be reached by following pointers from the HT directory (this may happen because of the compaction procedure, for example). When an orphaned record is encountered at the head of the log, it is skipped and the head position of the log is advanced to the next record.

From the description of the key insertion procedure in Section 4.5, it follows that the first record in each bucket chain (the one pointed to from the HT directory slot) is the most recently inserted record, while the last record in the chain is the earliest inserted record in that bucket. Hence, the last record in a bucket chain will be encountered first during the garbage collection process and it may be a valid or invalid (obsolete version of the respective key) record. A valid record needs to be reinserted at the tail of the log while an invalid record can be skipped. In either case, the next pointer in its predecessor record in the chain would need to be updated. Since we want to avoid in-place updates (random writes) on flash, this requires relocating the predecessor record and so forth all the way to the first record in the chain. *This effectively leads to the design decision of garbage collecting entire bucket chains on flash at a time*.

In summary, when the last record in a bucket chain is encountered in the log during garbage collection, all valid records in that chain are compacted and relocated to the tail of the log. This garbage collection strategy has two benefits.

- First, the writing of an entire chain of records in a bucket to the tail of the log also allows them to be compacted and placed contiguously on one or more flash pages and helps to speed up the lookup operations on those keys, as explained above in the context of compaction operations, and
- Second, since garbage (orphaned) records are created further down the log between the (current) head and tail (corresponding to the locations of all records in the chain before relocation), this helps to speedup the garbage collection process for the respective pages when they are encountered later (since orphaned records can be simply discarded).

We investigate the impact of compaction and garbage collection on system throughput in Section 5.

4.7 Crash Recovery

SkimpyStash 's persistency guarantee enables it to recover from system crashes due to power failure or other reasons. Because the system logs all key-value write operations to flash, it is straightforward to rebuild the HT directory in RAM by scanning all valid flash pages on flash. Recovery using this method can take some time, however, depending on the total size of valid flash pages that need to be scanned and the read throughput of flash memory.

If crash recovery needs to be executed faster so as to support “near" real-time recovery, then it is necessary to checkpoint the RAM HT directory periodically into flash (in a separate area from the key-value pair logs). Then, recovery involves reading the last written HT directory checkpoint from flash and scanning key-value pair logged flash pages with timestamps after that and inserting them into the restored HT directory. During the operation of checkpointing the HT directory, all insert operations into it will need to be suspended (but the read operations can continue). We use a temporary, small in-RAM hash table to provide index for the interim items and log them to flash. After the checkpointing operation completes, key-value pairs from the flash pages written in the interim are inserted into the HT directory. Key lookup operations, upon missing in the HT directory, will need to check in these flash pages (via the small additional hash table) until the later insertions into HT directory are complete. The flash garbage collection thread is suspended during the HT directory checkpointing operation, since the HT directory entries cannot be modified during this time.

Trace	**Total `get-set` ops**	**`get:set` ratio**	**Avg. size (bytes)**	
			Key	**Value**
Xbox	5.5 millions	7.5:1	92	1200
Dedup	40 millions	2.2:1	20	44

Table 1: Properties of the two traces used in the performance evaluation of SkimpyStash.

5. EVALUATION

We evaluate SkimpyStash on real-world traces obtained from the two applications described in Section 3.

5.1 C# Implementation

We have prototyped SkimpyStash in approximately 3000 lines of C# code. MurmurHash [4] is used to realize the hash functions used in our implementation to compute hash table directory indices and bloom filter lookup positions; different seeds are used to generate different hash functions in this family. The metadata store on flash is maintained as a file in the file system and is created/opened in non-buffered mode so that *there are no buffering/caching/prefetching effects in RAM from within the operating system*. The `ReaderWriterLockSlim` and `Monitor` classes in .NET 3.0 framework [1] are used to implement the concurrency control solution for multi-threading.

5.2 Evaluation Platform and Datasets

We use a standard server configuration to evaluate the performance of SkimpyStash. The server runs Windows Server 2008 R2 and uses an Intel Core2 Duo E6850 3GHz CPU, 4GB RAM, and fusionIO 160GB flash drive [2] attached over PCIe interface. We compare the throughput (`get-set` operations per sec) on the two traces described in Table 1.

We described two real-world applications in Section 3 that can use SkimpyStash as an underlying persistent key-value store. Data traces obtained from real-world instances of these applications are used to drive and evaluate the design of SkimpyStash.

Xbox LIVE Primetime trace

We evaluate the performance of SkimpyStash on a large trace of `get-set` key operations obtained from real-world instances of the Microsoft Xbox LIVE Primetime online multiplayer game [7]. In this application, the key is a dot-separated sequence of strings with total length averaging 94 characters and the value averages around 1200 bytes. The ratio of `get` operations to set `set` operations is about 7.5:1.

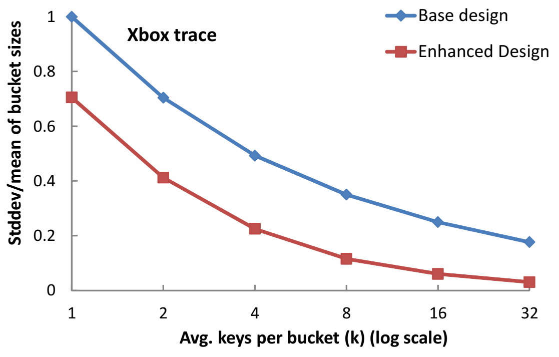


Figure 8: **Xbox trace: Relative variation in bucket sizes (standard-deviation/mean) for different values of average bucket size (k) for the base and enhanced designs. (Behavior on dedup trace is similar.)**

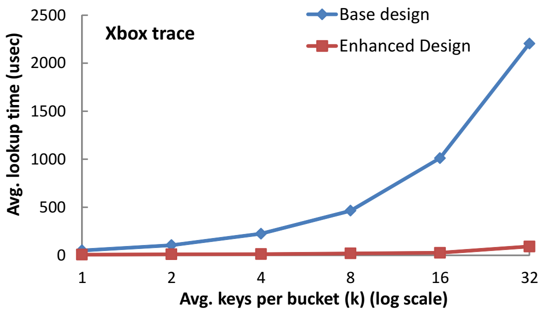


Figure 9: **Xbox trace: Average lookup (`get`) time for different values of average bucket size (k) for the base and enhanced designs. The enhanced design reduces lookup times by factors of 6x-24x as k increases.**

Storage Deduplication trace We have built a storage deduplication analysis tool that can crawl a root directory, generate the sequence of chunk hashes for a given average chunk hash size, and compute the number of deduplicated chunks and storage bytes. The enterprise data backup trace we use for evaluations in this paper was obtained by our storage deduplication analysis tool using 4KB chunk sizes. The trace contains 27,748,824 total chunks and 12,082,492 unique chunks. Using this, we obtain the ratio of `get` operations to `set` operations in the trace as 2.2:1. In this application, the key is a 20-byte SHA-1 hash of the corresponding chunk and the value is the meta-data for the chunk, with key-value pair total size upper bounded by 64 bytes.

The properties of the two traces are summarized in Table 1. Both traces include `get` operations on keys that have not been `set` earlier in the trace. (Such `get` operations will return `null`.) This is an inherent property of the nature of the application, hence we play the traces "as-is" to evaluate throughput in operations per second.

5.3 Performance Evaluation

We evaluate the performance impact of our design decisions and obtain ballpark ranges on lookup times and throughput (`get-set` operations/sec) for SkimpyStash on the Xbox LIVE Primetime online multi-player game and storage deduplication application traces (from Table 1). We disable the log compaction procedure for all but the last set of experiments. In the last set of experiments, we investigate the impact of garbage collection (which also does compaction) on system throughput.

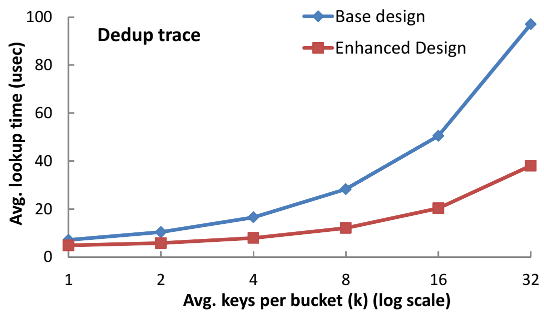


Figure 10: **Dedup trace: Average lookup (`get`) time for different values of average bucket size (k) for the base and enhanced designs. The enhanced design reduces lookup times by factors of 1.5x-2.5x as k increases.**

Impact of load balancing on bucket sizes: In the base design, hashing keys to single buckets can lead to wide variations in the chain length on flash in each bucket, and this translates to wide variations in lookup times. We investigate how two-choice based load balancing of keys to buckets can help reduce variance among bucket sizes. In Figure 8, we plot the relative variation in the bucket sizes (standard-deviation/mean of bucket sizes) for different values of $k = 1, 2, 4, 8, 16, 32$ on the Xbox trace (the behavior on dedup trace is similar). This metric is about 1.4x-6x times better for the enhanced design than for the based design on both traces. The gap starts at about 1.4x for the case $k = 1$ and increases to about 6x for the case $k = 32$. This provides conclusive evidence that two choice based load balancing is an effective strategy for reducing variations in bucket sizes.

Key lookup latencies: The two ideas of load balancing across buckets and using a bloom filter per bucket help to significantly reduce average key lookup times in the enhanced design, with the gains increasing as the average bucket size parameter k increases. At a value of $k = 8$, the average lookup time in the enhanced design is 20 μsec for the Xbox trace and 12 μsec for the dedup trace. In Figures 9 and 10, we plot the average lookup time for different values of $k = 1, 2, 4, 8, 16, 32$ on the two traces respectively. As the value of k increases, the gains in the enhanced design increase from 6x to 24x for the Xbox trace and from 1.5x to 2.5x for the dedup trace. The gains are more for the Xbox trace since that trace has many updates to earlier inserted keys (while the dedup trace has none), hence chains accumulate garbage records and get longer over time and bloom filters help even more to speedup lookups on non-existing keys (by avoiding searching the entire chain on flash).

Throughput (`get-set` operations/sec): The enhanced design achieves throughputs in the range of 10,000-69,000 ops/sec on the Xbox trace and 34,000-165,000 ops/sec on the dedup trace, with throughputs decreasing (as expected) with increasing values of k. This is shown in Figures 11 and 12. The throughput gains for the enhanced design over the base design are in the range of 3.5x-20x on the Xbox trace and 1.3x-2.5x on the dedup trace, with the gains increasing as the parameter k increases in value. These trends are in line with that of average lookup times.

The higher throughput of SkimpyStash on the dedup trace can be explained as follows. The write mix per unit operation (`get` or `set`) in the dedup trace is about 2.65 times that of the Xbox trace. However, since the key-value pair size is about 20 times smaller for the dedup trace, the number of syncs to stable storage per write

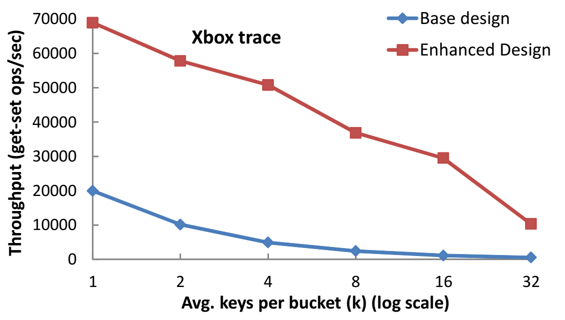


Figure 11: Xbox trace: Throughput (`get-set` ops/sec) for different values of average bucket size (k) for the base and enhanced designs. The enhanced design improves throughput by factors of 3.5x-20x as k increases.

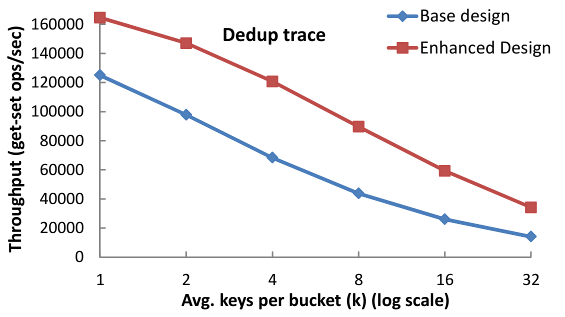


Figure 12: Dedup trace: Throughput (`get-set` ops/sec) for different values of average bucket size (k) for the base and enhanced designs. The enhanced design improves throughput by factors of 1.3x-2.5x as k increases.

operation is about 20 times less. Overall, the number of syncs to stable storage per unit operation is about 7.6 times less for the dedup trace. Moreover, bucket chains get longer in the Xbox trace due to invalid (garbage) records accumulating from key updates. For these reasons, SkimpyStash obtains higher throughputs on the dedup trace. In addition, since the dedup trace has no key update operations, the lookup operation can be avoided during key inserts for the dedup trace; this also contributes to the higher throughput of dedup trace over the Xbox trace. Garbage collection can help to improve performance on the Xbox trace, as we discuss next.

Impact of garbage collection activity: We study the impact of garbage collection (GC) activity on system throughput (ops/sec) and lookup times in SkimpyStash. The storage deduplication application does not involve updates to chunk metadata, hence we evaluate the impact of garbage collection on the Xbox trace. We fix an average bucket size of $k = 8$ for this set of experiments.

The aggressiveness of the garbage collection activity is controlled by a parameter g which is the interval of number of key *update* operations after which the garbage collector is invoked. Note that update operations are only those insert operations that update an earlier value of an already inserted key. Hence, the *garbage collector is invoked after every g key-value pair records worth of garbage accumulate in the log.* When the garbage collector is invoked, it starts scanning from the (current) head of the log and skips over orphaned records until it encounters the first valid or invalid record (that is part of some bucket chain). Then, it garbage collects that entire bucket chain, compacts and writes the valid records in that chain to the tail of the log, and returns.

In Figure 13, we plot the throughput (ops/sec) obtained on the

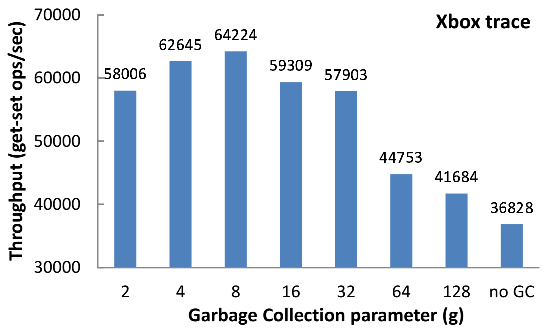


Figure 13: Xbox trace: Throughput (`get-set` ops/sec) for different values of the garbage collection parameter g. (The average bucket size parameter is fixed at $k = 8$.)

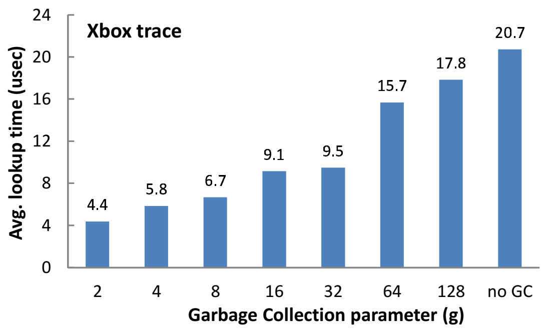


Figure 14: Xbox trace: Average lookup (`get`) time for different values of the garbage collection parameter g. (The average bucket size parameter is fixed at $k = 8$.)

Xbox trace as the garbage collection parameter g is varied. (The average bucket size parameter is fixed at $k = 8$.) We see that as g is increased from $g = 2$ onwards, the throughput first increases, peaks at $g = 8$, and then drops off all the the way to $g = 128$, at which point the throughput is close to that without garbage collection. This trend can be explained as follows. For small values of g, the high frequency of garbage collection is an overhead on system performance. As g is increased, the frequency of garbage collection activity decreases while lookup times continue to benefit (but less and less) from the compaction procedure involved in garbage collection (as explained in Section 4.6). Thus, the parameter g pulls system throughput in two different directions due to different effects, with the sweet spot for optimal performance occurring at $g_{opt} = 8$. We also reran the experiments for an average bucket size parameter of $k = 4$ and found the optimal value of g to be $g_{opt} = 8$ in that case also. It appears that the value of g_{opt} is a property of the trace and depends on the rate at which the application is generating garbage records.

Finally, we study the impact of the compaction procedure (as part of garbage collection activity) on average lookup times. As the compaction procedure helps to remove invalid records from bucket chains and reduce their lengths as well as place them contiguously on one or more flash pages, it helps to reduce search times in a bucket, as explained in Section 4.6. The impact can be seen in Figure 14, where lookup times steadily *decrease* as the aggressiveness of the garbage collection activity is *increased* (through *decrease* of the parameter g).

Comparison with earlier key-value store design: We compare SkimpyStash with FlashStore [16], a flash-based key-value store

Trace	FlashStore (ops/sec)	SkimpyStash ($k = 1$) (ops/sec)
Xbox	58,600	69,000
Dedup	75,100	165,000

Table 2: FlashStore [16] vs. SkimpyStash ($k = 1$).

which uses a variant of cuckoo hashing-based collision resolution policy [25] and compact key signatures to store metadata per key-value pair in the RAM. We use $n = 16$ hash functions to implement cuckoo hashing as suggested in [16]. Since FlashStore keeps metadata for only one key-value pair per hash table bucket, we compare it with the SkimpyStash design with the average bucket size parameter (k) set to 1, so as to keep the comparisons fair. Table 2 gives the summary of the throughput performance on Xbox and Dedup traces. SkimpyStash shows superior performance due to the fewer number of RAM accesses when non-existing keys are looked up – in such cases, SkimpyStash needs only two memory accesses, while FlashStore needs as many as 16. In addition, due to the nature of cuckoo hashing, FlashStore also needs to lookup an auxiliary hash table when items are not found in the main (cuckoo) hash table, which contributes to its reduced throughput. Since the ratio of the insert (set) to the lookup (get) operations on the Dedupe trace is higher than the Xbox trace (as explained in Section 5.2), SkimpyStash shows relatively better performance for the Dedupe trace as it has less overheads during the insertion of non-existing keys.

6. RELATED WORK

Flash memory has received lots of recent interest as a stable storage media that can overcome the access bottlenecks of hard disks. Researchers have considered modifying existing applications to improve performance on flash as well as providing operating system support for inserting flash as another layer in the storage hierarchy. In this section, we briefly review work that is related to SkimpyStash and point out its differentiating aspects.

MicroHash [27] designs a memory-constrained index structure for flash-based sensor devices with the goal of optimizing energy usage and minimizing memory footprint. Keys are assigned to buckets in a RAM directory based on range and flushed to flash when RAM buffer page is full. The directory needs to be repartitioned periodically based on bucket utilizations. SkimpyStash addresses this problem by hashing keys to hash table directory buckets and using two-choice load balancing across them, but gives up the ability to serve range queries. It also uses a bloom filter in each directory slot in RAM to disambiguate the two-choice during lookups. Moreover, MicroHash needs to write partially filled (index) pages to flash corresponding to keys in the same bucket, while the writes to flash in SkimpyStash always write full pages of key-value pair records.

FlashDB [23] is a self-tuning B^+-tree based index that dynamically adapts to the mix of reads and writes in the workload. Like MicroHash, it also targets memory and energy constrained sensor network devices. Because a B^+-tree needs to maintain partially filled leaf-level buckets on flash, appending of new keys to these buckets involves random writes, which is not an efficient flash operation. Hence, an adaptive mechanism is also provided to switch between disk and log-based modes. The system leverages the fact that key values in sensor applications have a small range and that at any given time, a small number of these leaf-level buckets are active. Minimizing latency is not an explicit design goal.

Tree-based index structures optimized for the flash memory are proposed in [8, 21]. These works do not focus on the RAM space reduction. In contrast, here we are mainly focusing on the hash-based index structure with low RAM footprint.

The benefits of using flash in a log-like manner have been exploited in FlashLogging [14] for synchronous logging. This system uses multiple inexpensive USB drives and achieves performance comparable to flash SSDs but with much lower price. Flashlogging assumes sequential workloads. In contrast, SkimpyStash works with arbitrary key access workloads.

FAWN [11] uses an array of embedded processors equipped with small amounts of flash to build a power-efficient cluster architecture for data-intensive computing. The differentiating aspect of SkimpyStash includes the aggressive design goal of extremely low RAM usage – SkimpyStash incurs about 1 byte RAM overhead per key-value pair stored on flash while FAWN uses 6 bytes. This requires SkimpyStash to use some techniques that are different from FAWN. Moreover, we investigate the use of SkimpyStash in data center server-class applications that need to achieve high throughput on read-write mixed workloads.

BufferHash [10] builds a content addressable memory (CAM) system using flash storage for networking applications like WAN optimizers. It buffers key-value pairs in RAM, organized as a hash table, and flushes the hash table to flash when the buffer is full. Past copies of hash tables on flash are searched using a time series of Bloom filters maintained in RAM and searching keys on a given copy involve multiple flash reads. Thus, a key uses as many bytes in RAM (across all bloom filters) as the number of times it is updated. Moreover, the storage of key-value pairs in hash tables on flash wastes space on flash, since hash table load factors need to be well below 100% to keep lookup times bounded.

FlashStore [16] is a high throughput persistent key-value store that uses flash memory as a non-volatile *cache* between RAM and hard disk. It is designed to store the working set of key-value pairs on flash and use one flash read per key lookup. As the working set changes over time, space is made for the current working set by destaging recently unused key-value pairs to hard disk and recycling pages in the flash store. FlashStore organizes key-value pairs in a log-structure on flash to exploit faster sequential write performance. It uses an in-memory hash table to index them, with hash collisions resolved by a variant of cuckoo hashing [25]. The in-memory hash table stores compact key signatures instead of full keys so as to strike tradeoffs between RAM usage and false flash read operations. The RAM usage is 6 bytes per key-value stored on flash.

ChunkStash [15] is a key-value store on flash that is designed for speeding up inline storage deduplication. It indexes data chunks (with SHA-1 hash as key) for identifying duplicate data. ChunkStash uses one flash read per chunk lookup and organizes chunk metadata in a log-structure on flash to exploit fast sequential writes. It builds an in-memory hash table to index them using techniques similar to FlashStore [16] to reduce RAM usage.

7. CONCLUSION

We designed SkimpyStash to be used as a high throughput persistent key-value storage layer for a broad range of server class applications. The distinguishing feature of SkimpyStash is the design goal of extremely low RAM footprint at about 1 byte per key-value pair, which is more aggressive than earlier designs. We used real-world data traces from two data center applications, namely, Xbox LIVE Primetime online multi-player game and inline storage deduplication, to drive and evaluate the design of SkimpyStash on commodity server platforms. SkimpyStash provides throughputs from few 10,000s to upwards of 100,000 `get-set` operations/sec on the evaluated applications.

8. REFERENCES

[1] C# `System.Threading`. `http://msdn.microsoft.com/en-us/library/system.threading.aspx`.

[2] Fusion-IO Drive Datasheet . `http://www.fusionio.com/PDFs/Data_Sheet_ioDrive_2.pdf`.

[3] Iometer. `http://www.iometer.org/`.

[4] MurmurHash Fuction. `http://en.wikipedia.org/wiki/MurmurHash`.

[5] MySpace Uses Fusion Powered I/O to Drive Greener and Better Data Centers. `http://www.fusionio.com/case-studies/myspace-case-study.pdf`.

[6] Releasing Flashcache. `http://www.facebook.com/note.php?note_id=388112370932`.

[7] Xbox LIVE Primetime game. `http://www.xboxprimetime.com/`.

[8] AGRAWAL, D., GANESAN, D., SITARAMAN, R., DIAO, Y., AND SINGH, S. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. In *VLDB* (2009).

[9] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J., MANASSE, M., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *USENIX* (2008).

[10] ANAND, A., MUTHUKRISHNAN, C., KAPPES, S., AKELLA, A., AND NATH, S. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. In *NSDI* (2010).

[11] ANDERSEN, D., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. FAWN: A Fast Array of Wimpy Nodes. In *SOSP* (2009).

[12] AZAR, Y., BRODER, A., KARLIN, A., AND UPFAL, E. Balanced Allocations. In *SIAM Journal on Computing* (1994).

[13] BRODER, A., AND MITZENMACHER, M. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics* (2002).

[14] CHEN, S. FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance. In *SIGMOD* (2009).

[15] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory. In *USENIX* (2010).

[16] DEBNATH, B., SENGUPTA, S., AND LI, J. FlashStore: High Throughput Persistent Key-Value Store. In *VLDB* (2010).

[17] GAL, E., AND TOLEDO, S. Algorithms and Data Structures for Flash Memories. In *ACM Computing Surveys* (2005), vol. 37.

[18] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *ASPLOS* (2009).

[19] KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. A Flash-Memory Based File System. In *USENIX* (1995).

[20] KOLTSIDAS, I., AND VIGLAS, S. Flashing Up the Storage Layer. In *VLDB* (2008).

[21] LI, Y., HE, B., 0001, J. Y., LUO, Q., AND YI, K. Tree Indexing on Solid State Drives. *PVLDB 3*, 1 (2010).

[22] NATH, S., AND GIBBONS, P. Online Maintenance of Very Large Random Samples on Flash Storage. In *VLDB* (2008).

[23] NATH, S., AND KANSAL, A. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *IPSN* (2007).

[24] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce, 1995.

[25] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (May 2004).

[26] ROSENBLUM, M., AND OUSTERHOUT, J. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems 10* (1991).

[27] ZEINALIPOUR-YAZTI, D., LIN, S., KALOGERAKI, V., GUNOPULOS, D., AND NAJJAR, W. A. Microhash: An Efficient Index Structure for Flash-based Sensor Devices. In *FAST* (2005).

[28] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *FAST* (2008).