# SILT: A Memory-Efficient, High-Performance Key-Value Store

Hyeontaek Lim
Carnegie Mellon University

Bin Fan
Carnegie Mellon University

David G. Andersen
Carnegie Mellon University

Michael Kaminsky
Intel Labs

## ABSTRACT

SILT (Small Index Large Table) is a memory-efficient, high-performance key-value store system based on flash storage that scales to serve billions of key-value items on a single node. It requires only 0.7 bytes of DRAM per entry and retrieves key/value pairs using on average 1.01 flash reads each. SILT combines new algorithmic and systems techniques to balance the use of memory, storage, and computation. Our contributions include: (1) the design of three basic key-value stores each with a different emphasis on memory-efficiency and write-friendliness; (2) synthesis of the basic key-value stores to build a SILT key-value store system; and (3) an analytical model for tuning system parameters carefully to meet the needs of different workloads. SILT requires one to two orders of magnitude less memory to provide comparable throughput to current high-performance key-value systems on a commodity desktop system with flash storage.

## CATEGORIES AND SUBJECT DESCRIPTORS

D.4.2 [**Operating Systems**]: Storage Management; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance; E.1 [**Data**]: Data Structures; E.2 [**Data**]: Data Storage Representations; E.4 [**Data**]: Coding and Information Theory

## GENERAL TERMS

Algorithms, Design, Measurement, Performance

*SOSP '11*, October 23-26, 2011, Cascais, Portugal.

| Metric | 2008 → 2011 | Increase |
|---|---|---|
| CPU transistors | 731 → 1,170 M | 60 % |
| DRAM capacity | 0.062 → 0.153 GB/$ | 147 % |
| Flash capacity | 0.134 → 0.428 GB/$ | 219 % |
| Disk capacity | 4.92 → 15.1 GB/$ | 207 % |

**Table 1: From 2008 to 2011, flash and hard disk capacity increased much faster than either CPU transistor count or DRAM capacity.**

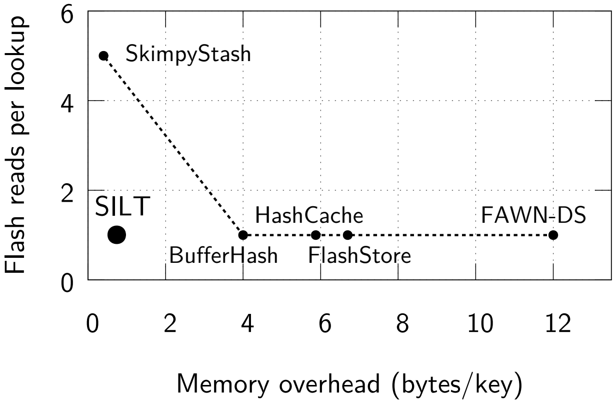

**Figure 1: The memory overhead and lookup performance of SILT and the recent key-value stores. For both axes, smaller is better.**

## KEYWORDS

Algorithms, design, flash, measurement, memory efficiency, performance

## 1. INTRODUCTION

Key-value storage systems have become a critical building block for today's large-scale, high-performance data-intensive applications. High-performance key-value stores have therefore received substantial attention in a variety of domains, both commercial and academic: e-commerce platforms [21], data deduplication [1, 19, 20], picture stores [7], web object caching [4, 30], and more.

To achieve low latency and high performance, and make best use of limited I/O resources, key-value storage systems require efficient indexes to locate data. As one example, Facebook engineers recently created a new key-value storage system that makes aggressive use of DRAM-based indexes to avoid the bottleneck caused by multiple disk operations when reading data [7]. Unfortunately, DRAM is up to 8X more expensive and uses 25X more power per bit than flash, and, as Table 1 shows, is growing more slowly than the capacity of the disk or flash that it indexes. As key-value stores scale in both size and importance, index memory efficiency is increasingly becoming one of the most important factors for the system's scalability [7] and overall cost effectiveness.

Recent proposals have started examining how to reduce per-key in-memory index overhead [1, 2, 4, 19, 20, 32, 40], but these solutions either require more than a few bytes per key-value entry in memory [1, 2, 4, 19], or compromise performance by keeping all or part of the index on flash or disk and thus require many flash reads or disk seeks to handle each key-value lookup [20, 32, 40] (see

Figure 1 for the design space). We term this latter problem *read amplification* and explicitly strive to avoid it in our design.

This paper presents a new flash-based key-value storage system, called SILT (Small Index Large Table), that significantly reduces per-key memory consumption with predictable system performance and lifetime. SILT requires approximately 0.7 bytes of DRAM per key-value entry and uses on average only 1.01 flash reads to handle lookups. Consequently, SILT can saturate the random read I/O on our experimental system, performing 46,000 lookups per second for 1024-byte key-value entries, and it can potentially scale to billions of key-value items on a single host. SILT offers several knobs to trade memory efficiency and performance to match available hardware.

This paper makes three main contributions:

- The design and implementation of three basic key-value stores (LogStore, HashStore, and SortedStore) that use new fast and compact indexing data structures (partial-key cuckoo hashing and entropy-coded tries), each of which places different emphasis on memory-efficiency and write-friendliness.
- Synthesis of these basic stores to build SILT.
- An analytic model that enables an explicit and careful balance between memory, storage, and computation to provide an accurate prediction of system performance, flash lifetime, and memory efficiency.

## 2. SILT KEY-VALUE STORAGE SYSTEM

Like other key-value systems, SILT implements a simple exact-match hash table interface including `PUT` (map a new or existing key to a value), `GET` (retrieve the value by a given key), and `DELETE` (delete the mapping of a given key).

For simplicity, we assume that keys are *uniformly distributed* 160-bit hash values (e.g., pre-hashed keys with SHA-1) and that data is fixed-length. This type of key-value system is widespread in several application domains such as data deduplication [1, 19, 20], and is applicable to block storage [18, 36], microblogging [25, 38], WAN acceleration [1], among others. In systems with lossy-compressible data, e.g., picture stores [7, 26], data can be adaptively compressed to fit in a fixed-sized slot. A key-value system may also let applications choose one of multiple key-value stores, each of which is optimized for a certain range of value sizes [21]. We discuss the relaxation of these assumptions in Section 4.

**Design Goals and Rationale** The design of SILT follows from five main goals:

1. **Low read amplification**: Issue at most $1+\varepsilon$ flash reads for a single `GET`, where $\varepsilon$ is configurable and small (e.g., 0.01).
   *Rationale*: Random reads remain the read throughput bottleneck when using flash memory. Read amplification therefore directly reduces throughput.
2. **Controllable write amplification and favoring sequential writes**: It should be possible to adjust how many times a key-value entry is rewritten to flash over its lifetime. The system should issue flash-friendly, large writes.
   *Rationale*: Flash memory can undergo only a limited number of erase cycles before it fails. Random writes smaller than the SSD log-structured page size (typically 4 KiB[1]) cause extra flash traffic.

[1] For clarity, binary prefixes (powers of 2) will include "i", while SI prefixes (powers of 10) will appear without any "i".

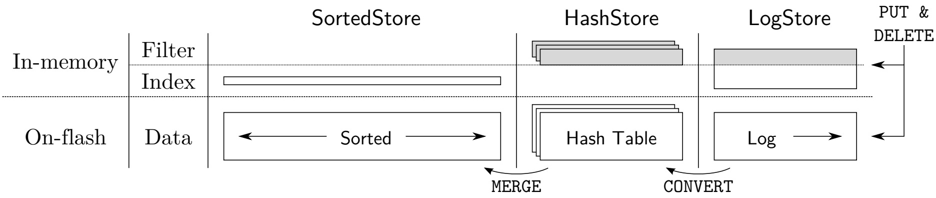

**Figure 2: Architecture of SILT.**

| | **SortedStore** (§3.3) | **HashStore** (§3.2) | **LogStore** (§3.1) |
|---|---|---|---|
| Mutability | Read-only | Read-only | Writable |
| Data ordering | Key order | Hash order | Insertion order |
| Multiplicity | 1 | $\geq 0$ | 1 |
| Typical size | $>$ 80% of total entries | $<$ 20% | $<$ 1% |
| DRAM usage | 0.4 bytes/entry | 2.2 bytes/entry | 6.5 bytes/entry |

**Table 2: Summary of basic key-value stores in SILT.**

Optimizations for memory efficiency and garbage collection often require data layout changes on flash. The system designer should be able to select an appropriate balance of flash lifetime, performance, and memory overhead.

3. **Memory-efficient indexing**: SILT should use as little memory as possible (e.g., less than one byte per key stored).
*Rationale*: DRAM is both more costly and power-hungry per gigabyte than Flash, and its capacity is growing more slowly.

4. **Computation-efficient indexing**: SILT's indexes should be fast enough to let the system saturate the flash I/O.
*Rationale*: System balance and overall performance.

5. **Effective use of flash space**: Some data layout options use the flash space more sparsely to improve lookup or insertion speed, but the total space overhead of any such choice should remain small – less than 20% or so.
*Rationale*: SSDs remain relatively expensive.

In the rest of this section, we first explore SILT's high-level architecture, which we term a "multi-store approach", contrasting it with a simpler but less efficient single-store approach. We then briefly outline the capabilities of the individual store types that we compose to form SILT, and show how SILT handles key-value operations using these stores.

**Conventional Single-Store Approach** A common approach to building high-performance key-value stores on flash uses three components:

1. *an in-memory filter* to efficiently test whether a given key is stored in this store before accessing flash;

2. *an in-memory index* to locate the data on flash for a given key; and

3. *an on-flash data layout* to store all key-value pairs persistently.

Unfortunately, to our knowledge, no existing index data structure and on-flash layout achieve all of our goals simultaneously. For example, HashCache-Set [4] organizes on-flash keys as a hash table, eliminating the in-memory index, but incurring random writes that impair insertion speed. To avoid expensive random writes, systems such as FAWN-DS [2], FlashStore [19], and SkimpyStash [20] append new values sequentially to a log. These systems then require either an in-memory hash table to map a key to its offset in the log (often requiring 4 bytes of DRAM or more per entry) [2, 20]; or keep part of the index on flash using multiple random reads for each lookup [20].

**Multi-Store Approach** BigTable [14], Anvil [29], and BufferHash [1] chain multiple stores, each with different properties such as high write performance or inexpensive indexing.

Multi-store systems impose two challenges. First, they require effective designs and implementations of the individual stores: they must be efficient, compose well, and it must be efficient to transform data between the store types. Second, it must be efficient to query multiple stores when performing lookups. The design must keep read amplification low by not issuing flash reads to each store. A common solution uses a compact in-memory filter to test whether a given key can be found in a particular store, but this filter can be memory-intensive—e.g., BufferHash uses 4–6 bytes for each entry.

**SILT's multi-store design** uses a series of *basic key-value stores*, each optimized for a different purpose.

1. Keys are inserted into a write-optimized store, and over their lifetime flow into increasingly more memory-efficient stores.
2. Most key-value pairs are stored in the most memory-efficient basic store. Although data outside this store uses less memory-efficient indexes (e.g., to optimize writing performance), the average index cost per key remains low.
3. SILT is tuned for high worst-case performance—a lookup found in the last and largest store. As a result, SILT can avoid using an in-memory filter on this last store, allowing all lookups (successful or not) to take $1 + \varepsilon$ flash reads.

SILT's architecture and basic stores (the LogStore, HashStore, and SortedStore) are depicted in Figure 2. Table 2 summarizes these stores' characteristics.

*LogStore* is a write-friendly key-value store that handles individual `PUT`s and `DELETE`s. To achieve high performance, writes are appended to the end of a log file on flash. Because these items are ordered by their insertion time, the LogStore uses an in-memory hash table to map each key to its offset in the log file. The table doubles as an in-memory filter. SILT uses a memory-efficient, high-performance hash table based upon cuckoo hashing [34]. As described in Section 3.1, our *partial-key cuckoo hashing* achieves 93% occupancy with very low computation and memory overhead, a substantial improvement over earlier systems such as FAWN-DS and BufferHash that achieved only 50% hash table occupancy. Compared to the next two read-only store types, however, this index is still relatively memory-intensive, because it must store one 4-byte pointer for every key. SILT therefore uses only one instance of the LogStore (except during conversion to HashStore as described below), with fixed capacity to bound its memory consumption.

Once full, the LogStore is converted to an immutable *HashStore* in the background. The HashStore's data is stored as an on-flash hash table that does not require an in-memory index to locate entries. SILT uses multiple HashStores at a time before merging them into the next store type. Each HashStore therefore uses an efficient in-memory filter to reject queries for nonexistent keys.

*SortedStore* maintains key-value data in sorted key order on flash, which enables an extremely compact index representation (e.g., 0.4 bytes per key) using a novel design and implementation of *entropy-coded tries*. Because of the expense of inserting a single item into sorted data, SILT periodically merges in

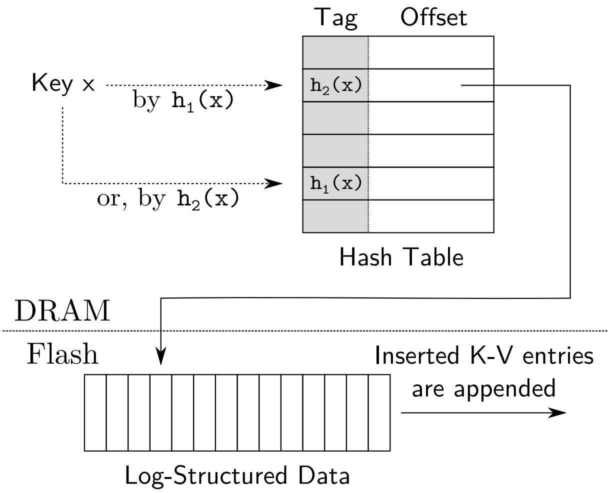

**Figure 3: Design of LogStore: an in-memory cuckoo hash table (index and filter) and an on-flash data log.**

bulk several HashStores along with an older version of a SortedStore and forms a new SortedStore, garbage collecting deleted or overwritten keys in the process.

**Key-Value Operations** Each `PUT` operation inserts a `(key,value)` pair into the LogStore, even if the key already exists. `DELETE` operations likewise append a "delete" entry into the LogStore. The space occupied by deleted or stale data is reclaimed when SILT merges HashStores into the SortedStore. These lazy deletes trade flash space for sequential write performance.

To handle `GET`, SILT searches for the key in the LogStore, HashStores, and SortedStore in sequence, returning the value found in the youngest store. If the "deleted" entry is found, SILT will stop searching and return "not found."

**Partitioning** Finally, we note that each physical node runs multiple SILT instances, responsible for disjoint key ranges, each with its own LogStore, SortedStore, and HashStore(s). This partitioning improves load-balance (e.g., virtual nodes [37]), reduces flash overhead during merge (Section 3.3), and facilitates system-wide parameter tuning (Section 5).

# 3. BASIC STORE DESIGN

## 3.1 LogStore

The LogStore writes `PUT`s and `DELETE`s sequentially to flash to achieve high write throughput. Its in-memory partial-key cuckoo hash index efficiently maps keys to their location in the flash log, as shown in Figure 3.

**Partial-Key Cuckoo Hashing** The LogStore uses a new hash table based on *cuckoo hashing* [34]. As with standard cuckoo hashing, it uses two hash functions $h_1$ and $h_2$ to map each key to two candidate buckets. On insertion of a new key, if one of the candidate buckets is empty, the key is inserted in this empty slot; if neither bucket is available, the new key "kicks out" the key that already resides in one of the two buckets, and the displaced key is then inserted to its own alternative bucket (and may kick out other keys). The insertion algorithm repeats this process until a vacant position is found, or it reaches a maximum number of displacements (e.g., 128 times in our implementation). If no vacant slot found, it indicates the hash table is almost full, so SILT freezes the LogStore and initializes a new one without expensive rehashing.

To make it compact, the hash table does not store the entire key (e.g., 160 bits in SILT), but only a "tag" of the actual key. A lookup proceeds to flash only when the given key matches the tag, which can prevent most unnecessary flash reads for non-existing keys. If the tag matches, the full key and its value are retrieved from the log on flash to verify if the key it read was indeed the correct key.

Although storing only the tags in the hash table saves memory, it presents a challenge for cuckoo hashing: moving a key to its alternative bucket requires knowing its other hash value. Here, however, the full key is stored only on flash, but reading it from flash is too expensive. Even worse, moving this key to its alternative bucket may in turn displace another key; ultimately, each displacement required by cuckoo hashing would result in additional flash reads, just to insert a single key.

To solve this costly displacement problem, our *partial-key cuckoo hashing* algorithm stores the index of the alternative bucket as the tag; in other words, partial-key cuckoo hashing uses the tag to reduce flash reads for non-existent key lookups *as well as* to indicate an alternative bucket index to perform cuckoo displacement without any flash reads. For example, if a key $x$ is assigned to bucket $h_1(x)$, the other hash value $h_2(x)$ will become its tag stored in bucket $h_1(x)$, and vice versa (see Figure 3). Therefore, when a key is displaced from the bucket $a$, SILT reads the tag (value: $b$) at this bucket, and moves the key to the bucket $b$ without needing to read from flash. Then it sets the tag at the bucket $b$ to value $a$.

To find key $x$ in the table, SILT checks if $h_1(x)$ matches the tag stored in bucket $h_2(x)$, or if $h_2(x)$ matches the tag in bucket $h_1(x)$. If the tag matches, the `(key,value)` pair is retrieved from the flash location indicated in the hash entry.

**Associativity** Standard cuckoo hashing allows 50% of the table entries to be occupied before unresolvable collisions occur. SILT improves the occupancy by increasing the associativity of the cuckoo hashing table. Each bucket of the table is of capacity four (i.e., it contains up to 4 entries). Our experiments show that using a 4-way set associative hash table improves space utilization of the table to about 93%,[2] which matches the known experimental result for various variants of cuckoo hashing [24]; moreover, 4 entries/bucket still allows each bucket to fit in a single cache line.[3]

**Hash Functions** Keys in SILT are 160-bit hash values, so the LogStore finds $h_1(x)$ and $h_2(x)$ by taking two non-overlapping slices of the low-order bits of the key $x$.

By default, SILT uses a 15-bit key fragment as the tag. Each hash table entry is 6 bytes, consisting of a 15-bit tag, a single valid bit, and a 4-byte offset pointer. The probability of a false positive retrieval is 0.024% (see Section 5 for derivation), i.e., on average 1 in 4,096 flash retrievals is unnecessary. The maximum number of hash buckets (not entries) is limited by the key fragment length. Given 15-bit key fragments, the hash table has at most $2^{15}$ buckets, or $4 \times 2^{15} = 128$ Ki entries. To store more keys in LogStore, one can increase the size of the key fragment to have more buckets, increase the associativity to pack more entries into one bucket, and/or partition the key-space to smaller regions and assign each region to one SILT instance with a LogStore. The tradeoffs associated with these decisions are presented in Section 5.

[2] Space utilization here is defined as the fraction of used entries (not used buckets) in the table, which more precisely reflects actual memory utilization.

[3] Note that, another way to increase the utilization of a cuckoo hash table is to use more hash functions (i.e., each key has more possible locations in the table). For example, FlashStore [19] applies 16 hash functions to achieve 90% occupancy. However, having more hash functions increases the number of cache lines read upon lookup and, in our case, requires more than one tag stored in each entry, increasing overhead.

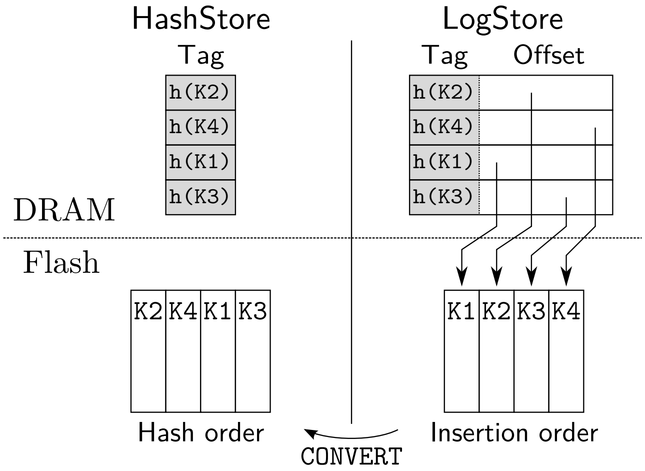

**Figure 4: Convert a LogStore to a HashStore. Four keys K1, K2, K3, and K4 are inserted to the LogStore, so the layout of the log file is the insert order; the in-memory index keeps the offset of each key on flash. In HashStore, the on-flash data forms a hash table where keys are in the same order as the in-memory filter.**

## 3.2 HashStore

Once a LogStore fills up (e.g., the insertion algorithm terminates without finding any vacant slot after a maximum number of displacements in the hash table), SILT freezes the LogStore and converts it into a more memory-efficient data structure. Directly sorting the relatively small LogStore and merging it into the much larger SortedStore requires rewriting large amounts of data, resulting in high write amplification. On the other hand, keeping a large number of LogStores around before merging could amortize the cost of rewriting, but unnecessarily incurs high memory overhead from the LogStore's index. To bridge this gap, SILT first converts the LogStore to an immutable HashStore with higher memory efficiency; once SILT accumulates a configurable number of HashStores, it performs a bulk merge to incorporate them into the SortedStore. During the LogStore to HashStore conversion, the old LogStore continues to serve lookups, and a new LogStore receives inserts.

HashStore saves memory over LogStore by eliminating the index and reordering the on-flash `(key,value)` pairs from insertion order to hash order (see Figure 4). HashStore is thus an on-flash cuckoo hash table, and has the same occupancy (93%) as the in-memory version found in LogStore. HashStores also have one in-memory component, a filter to probabilistically test whether a key is present in the store without performing a flash lookup.

**Memory-Efficient Hash Filter** Although prior approaches [1] used Bloom filters [12] for the probabilistic membership test, SILT uses a hash filter based on partial-key cuckoo hashing. Hash filters are more memory-efficient than Bloom filters at low false positive rates. Given a 15-bit tag in a 4-way set associative cuckoo hash table, the false positive rate is $f = 2^{-12} = 0.024\%$ as calculated in Section 3.1. With 93% table occupancy, the effective number of bits per key using a hash filter is $15/0.93 = 16.12$. In contrast, a standard Bloom filter that sets its number of hash functions to optimize space consumption requires at least $1.44\log_2(1/f) = 17.28$ bits of memory to achieve the same false positive rate.

HashStore's hash filter is also efficient to create: SILT simply copies the tags from the LogStore's hash table, in order, discarding the offset pointers; on the contrary, Bloom filters would have been built from scratch, hashing every item in the LogStore again.

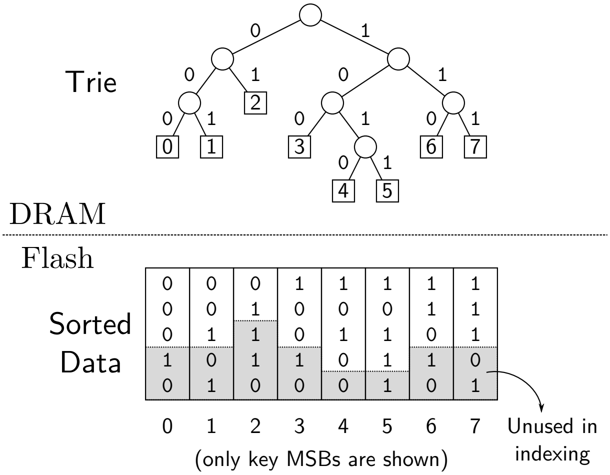

**Figure 5: Example of a trie built for indexing sorted keys. The index of each leaf node matches the index of the corresponding key in the sorted keys.**

## 3.3 SortedStore

SortedStore is a static key-value store with very low memory footprint. It stores `(key,value)` entries sorted by key on flash, indexed by a new *entropy-coded trie* data structure that is fast to construct, uses 0.4 bytes of index memory per key on average, and keeps read amplification low (exactly 1) by directly pointing to the correct location on flash.

**Using Sorted Data on Flash** Because of these desirable properties, SILT keeps most of the key-value entries in a single SortedStore. The entropy-coded trie, however, does not allow for insertions or deletions; thus, to merge HashStore entries into the SortedStore, SILT must generate a new SortedStore. The construction speed of the SortedStore is therefore a large factor in SILT's overall performance.

Sorting provides a natural way to achieve fast construction:

1. Sorting allows efficient bulk-insertion of new data. The new data can be sorted and sequentially merged into the existing sorted data.

2. Sorting is well-studied. SILT can use highly optimized and tested sorting systems such as Nsort [33].

**Indexing Sorted Data with a Trie** A trie, or a prefix tree, is a tree data structure that stores an array of keys where each leaf node represents one key in the array, and each internal node represents the longest common prefix of the keys represented by its descendants.

When fixed-length key-value entries are sorted by key on flash, a trie for the *shortest unique prefixes* of the keys serves as an index for these sorted data. The shortest unique prefix of a key is the shortest prefix of a key that enables distinguishing the key from the other keys. In such a trie, some prefix of a lookup key leads us to a leaf node with a direct index for the looked up key in sorted data on flash.

Figure 5 shows an example of using a trie to index sorted data. Key prefixes with no shading are the shortest unique prefixes which are used for indexing. The shaded parts are ignored for indexing because any value for the suffix part would not change the key location. A lookup of a key, for example,

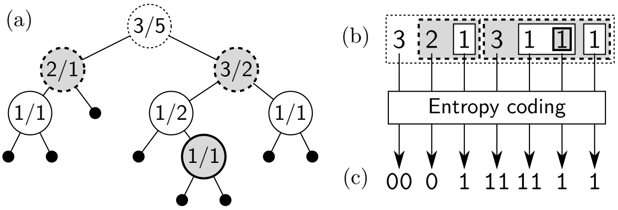

**Figure 6: (a) Alternative view of Figure 5, where a pair of numbers in each internal node denotes the number of leaf nodes in its left and right subtries. (b) A recursive form that represents the trie. (c) Its entropy-coded representation used by SortedStore.**

`10010`, follows down to the leaf node that represents `100`. As there are 3 preceding leaf nodes, the index of the key is 3. With fixed-length key-value pairs on flash, the exact offset of the data is the obtained index times the key-value pair size (see Section 4 for extensions for variable-length key-value pairs). Note that a lookup of similar keys with the same prefix of `100` (e.g., `10000`, `10011`) would return the same index even though they are not in the array; the trie guarantees a correct index lookup for stored keys, but says nothing about the presence of a lookup key.

**Representing a Trie** A typical tree data structure is unsuitable for SILT because each node would require expensive memory pointers, each 2 to 8 bytes long. Common trie representations such as level-compressed tries [3] are also inadequate if they use pointers.

SortedStore uses a compact recursive representation to eliminate pointers. The representation for a trie $T$ having $L$ and $R$ as its left and right subtries is defined as follows:

$$\texttt{Repr}(T) \quad := \quad |L| \ \texttt{Repr}(L) \ \texttt{Repr}(R)$$

where $|L|$ is the number of leaf nodes in the left subtrie. When $T$ is empty or a leaf node, the representation for $T$ is an empty string. (We use a special mark ($-1$) instead of the empty string for brevity in the simplified algorithm description, but the full algorithm does not require the use of the special mark.)

Figure 6 (a,b) illustrates the uncompressed recursive representation for the trie in Figure 5. As there are 3 keys starting with `0`, $|L| = 3$. In its left subtrie, $|L| = 2$ because it has 2 keys that have `0` in their second bit position, so the next number in the representation is 2. It again recurses into its left subtrie, yielding 1. Here there are no more non-leaf nodes, so it returns to the root node and then generates the representation for the right subtrie of the root, 3 1 1 1.

Algorithm 1 shows a simplified algorithm that builds a (non-entropy-coded) trie representation from sorted keys. It resembles quicksort in that it finds the partition of keys and recurses into both subsets. Index generation is fast ($\geq$ 7 M keys/sec on a modern Intel desktop CPU, Section 6).

**Looking up Keys** Key-lookup works by incrementally reading the trie representation (Algorithm 2). The function is supplied the lookup key and a trie representation string. By decoding the encoded next number, *thead*, SortedStore knows if the current node is an internal node where it can recurse into its subtrie. If the lookup key goes to the left subtrie, SortedStore recurses into the left subtrie, whose representation immediately follows in the given trie representation; otherwise, SortedStore recursively decodes and discards the entire left subtrie and then recurses into the right. SortedStore sums *thead* at every node where it recurses into a right subtrie; the sum of the *thead* values is the offset at which the lookup key is stored, if it exists.

```
# @param  T  array of sorted keys
# @return    trie representation
def construct(T):
  if len(T) == 0 or len(T) == 1:
    return [-1]
  else:
    # Partition keys according to their MSB
    L = [key[1:] for key in T if key[0] == 0]
    R = [key[1:] for key in T if key[0] == 1]
    # Recursively construct the representation
    return [len(L)] + construct(L) + construct(R)
```

**Algorithm 1: Trie representation generation in Python-like syntax. `key[0]` and `key[1:]` denote the most significant bit and the remaining bits of `key`, respectively.**

For example, to look up `10010`, SortedStore first obtains 3 from the representation. Then, as the first bit of the key is `1`, it skips the next numbers (2 1) which are for the representation of the left subtrie, and it proceeds to the right subtrie. In the right subtrie, SortedStore reads the next number (3; not a leaf node), checks the second bit of the key, and keeps recursing into its left subtrie. After reading the next number for the current subtrie (1), SortedStore arrives at a leaf node by taking the left subtrie. Until it reaches the leaf node, it takes a right subtrie only at the root node; from $n = 3$ at the root node, SortedStore knows that the offset of the data for `10010` is ($3 \times$ key-value-size) on flash.

**Compression** Although the above uncompressed representation uses up to 3 integers per key on average, for *hashed* keys, SortedStore can easily reduce the average representation size to 0.4 bytes/key by compressing each $|L|$ value using *entropy coding* (Figure 6 (c)). The value of $|L|$ tends to be close to half of $|T|$ (the number of leaf nodes in $T$) because fixed-length hashed keys are uniformly distributed over the key space, so both subtries have the same probability of storing a key. More formally, $|L| \sim \mathit{Binomial}(|T|, \frac{1}{2})$. When $|L|$ is small enough (e.g., $\leq 16$), SortedStore uses static, globally shared Huffman tables based on the binomial distributions. If $|L|$ is large, SortedStore encodes the difference between $|L|$ and its expected value (i.e., $\frac{|T|}{2}$) using Elias gamma coding [23] to avoid filling the CPU cache with large Huffman tables. With this entropy coding optimized for hashed keys, our entropy-coded trie representation is about twice as compact as the previous best recursive tree encoding [16].

When handling compressed tries, Algorithm 1 and 2 are extended to keep track of the number of leaf nodes at each recursion step. This does not require any additional information in the representation because the number of leaf nodes can be calculated recursively using $|T| = |L| + |R|$. Based on $|T|$, these algorithms choose an entropy coder for encoding `len(L)` and decoding `thead`. It is noteworthy that the special mark (`-1`) takes no space with entropy coding, as its entropy is zero.

**Ensuring Constant Time Index Lookups** As described, a lookup may have to decompress the entire trie, so that the cost of lookups would grow (large) as the number of entries in the key-value store grows.

To bound the lookup time, items are partitioned into $2^k$ buckets based on the first $k$ bits of their key. Each bucket has its own trie index. Using, e.g., $k = 10$ for a key-value store holding $2^{16}$ items, each bucket would hold in expectation $2^{16-10} = 2^6$ items. With high probability, no bucket holds more than $2^8$ items, so the time to decompress the trie for bucket is both bounded by a constant value, and small.

This bucketing requires additional information to be stored in memory: (1) the pointers to the trie representations of each bucket and (2) the number of entries in each bucket. SILT keeps the amount of this bucketing information small (less than 1 bit/key) by using a simpler version of a compact select

```
# @param  key    lookup key
# @param  trepr  trie representation
# @return        index of the key
#                in the original array
def lookup(key, trepr):
  (thead, ttail) = (trepr[0], trepr[1:])
  if thead == -1:
    return 0
  else:
    if key[0] == 0:
      # Recurse into the left subtrie
      return lookup(key[1:], ttail)
    else:
      # Skip the left subtrie
      ttail = discard_subtrie(ttail)
      # Recurse into the right subtrie
      return thead + lookup(key[1:], ttail)

# @param  trepr  trie representation
# @return        remaining trie representation
#                with the next subtrie consumed
def discard_subtrie(trepr):
  (thead, ttail) = (trepr[0], trepr[1:])
  if thead == -1:
    return ttail
  else:
    # Skip both subtries
    ttail = discard_subtrie(ttail)
    ttail = discard_subtrie(ttail)
    return ttail
```

**Algorithm 2: Key lookup on a trie representation.**

data structure, semi-direct-16 [11]. With bucketing, our trie-based indexing belongs to the class of data structures called monotone minimal perfect hashing [10, 13] (Section 7).

**Further Space Optimizations for Small Key-Value Sizes** For small key-value entries, SortedStore can reduce the trie size by applying *sparse indexing* [22]. Sparse indexing locates the *block* that contains an entry, rather than the exact offset of the entry. This technique requires scanning or binary search within the block, but it reduces the amount of indexing information. It is particularly useful when the storage media has a minimum useful block size to read; many flash devices, for instance, provide increasing I/Os per second as the block size drops, but not past a certain limit (e.g., 512 or 4096 bytes) [31, 35]. SILT uses sparse indexing when configured for key-value sizes of 64 bytes or smaller.

SortedStore obtains a sparse-indexing version of the trie by pruning some subtries in it. When a trie has subtries that have entries all in the same block, the trie can omit the representation of these subtries because the omitted data only gives in-block offset information between entries. Pruning can reduce the trie size to 1 bit per key or less if each block contains 16 key-value entries or more.

**Merging HashStores into SortedStore** SortedStore is an immutable store and cannot be changed. Accordingly, the merge process generates a new SortedStore based on the given HashStores and the existing SortedStore. Similar to the conversion from LogStore to HashStore, HashStores and the old SortedStore can serve lookups during merging.

| Comparison | "Deleted"? | Action on $K_{SS}$ | Action on $K_{HS}$ |
|---|---|---|---|
| $K_{SS} < K_{HS}$ | any | copy | – |
| $K_{SS} > K_{HS}$ | no | – | copy |
| $K_{SS} > K_{HS}$ | yes | – | drop |
| $K_{SS} = K_{HS}$ | no | drop | copy |
| $K_{SS} = K_{HS}$ | yes | drop | drop |

**Table 3: Merge rule for SortedStore. $K_{SS}$ is the current key from SortedStore, and $K_{HS}$ is the current key from the sorted data of HashStores. "Deleted" means the current entry in $K_{HS}$ is a special entry indicating a key of SortedStore has been deleted.**

The merge process consists of two steps: (1) sorting HashStores and (2) sequentially merging sorted HashStores data and SortedStore. First, SILT sorts all data in HashStores to be merged. This task is done by enumerating every entry in the HashStores and sorting these entries. Then, this sorted data from HashStores is sequentially merged with already sorted data in the SortedStore. The sequential merge chooses newest valid entries, as summarized in Table 3; either copy or drop action on a key consumes the key (i.e., by advancing the "merge pointer" in the corresponding store), while the current key remains available for the next comparison again if no action is applied to the key. After both steps have been completed, the old SortedStore is atomically replaced by the new SortedStore. During the merge process, both the old SortedStore and the new SortedStore exist on flash; however, the flash space overhead from temporarily having two SortedStores is kept small by performing the merge process on a single SILT instance at the same time.

In Section 5, we discuss how frequently HashStores should be merged into SortedStore.

**Application of False Positive Filters** Since SILT maintains only one SortedStore per SILT instance, SortedStore does not have to use a false positive filter to reduce unnecessary I/O. However, an extension to the SILT architecture might have multiple SortedStores. In this case, the trie index can easily accommodate the false positive filter; the filter is generated by extracting the key fragments from the sorted keys. Key fragments can be stored in an in-memory array so that they have the same order as the sorted data on flash. The extended SortedStore can consult the key fragments before reading data from flash.

# 4. EXTENDING SILT FUNCTIONALITY

SILT can support an even wider range of applications and workloads than the basic design we have described. In this section, we present potential techniques to extend SILT's capabilities.

**Crash Tolerance** SILT ensures that all its in-memory data structures are backed-up to flash and/or easily re-created after failures. All updates to LogStore are appended to the on-flash log chronologically; to recover from a fail-stop crash, SILT simply replays the log file to construct the in-memory index and filter. For HashStore and SortedStore, which are static, SILT keeps a copy of their in-memory data structures on flash, which can be re-read during recovery.

SILT's current design, however, does not provide crash tolerance for *new* updates to the data store. These writes are handled asynchronously, so a key insertion/update request to SILT may complete before its data is written durably to flash. For applications that need this additional level of crash tolerance, SILT would need to support an additional synchronous write mode. For example, SILT could delay returning from write calls until it confirms that the requested write is fully flushed to the on-flash log.

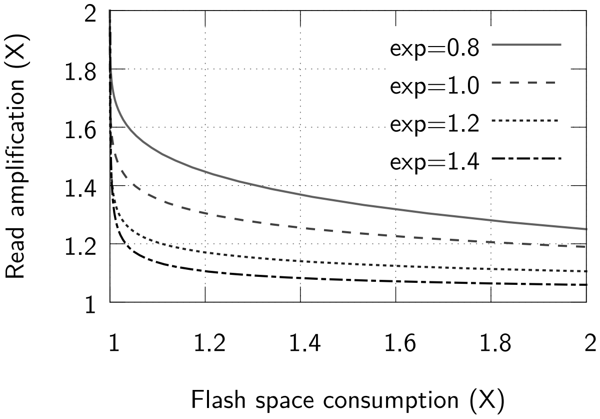

**Figure 7: Read amplification as a function of flash space consumption when inlining is applied to key-values whose sizes follow a Zipf distribution. "exp" is the exponent part of the distribution.**

**Variable-Length Key-Values** For simplicity, the design we presented so far focuses on fixed-length key-value data. In fact, SILT can easily support variable-length key-value data by using *indirection with inlining*. This scheme follows the existing SILT design with fixed-sized slots, but stores `(offset, first part of (key, value))` pairs instead of the actual `(key, value)` in HashStores and SortedStores (LogStores can handle variable-length data natively). These offsets point to the remaining part of the key-value data stored elsewhere (e.g., a second flash device). If a whole item is small enough to fit in a fixed-length slot, indirection can be avoided; consequently, large data requires an extra flash read (or write), but small data incurs no additional I/O cost. Figure 7 plots an analytic result on the tradeoff of this scheme with different slot sizes. It uses key-value pairs whose sizes range between 4 B and 1 MiB and follow a Zipf distribution, assuming a 4-byte header (for key-value lengths), a 4-byte offset pointer, and an uniform access pattern.

For specific applications, SILT can alternatively use segregated stores for further efficiency. Similar to the idea of simple segregated storage [39], the system could instantiate several SILT instances for different fixed key-value size combinations. The application may choose an instance with the most appropriate key-value size as done in Dynamo [21], or SILT can choose the best instance for a new key and return an opaque key containing the instance ID to the application. Since each instance can optimize flash space overheads and additional flash reads for its own dataset, using segregated stores can reduce the cost of supporting variable-length key-values close to the level of fixed-length key-values.

In the subsequent sections, we will discuss SILT with fixed-length key-value pairs only.

**Fail-Graceful Indexing** Under high memory pressure, SILT may temporarily operate in a degraded indexing mode by allowing higher read amplification (e.g., more than 2 flash reads per lookup) to avoid halting or crashing because of insufficient memory.

(1) Dropping in-memory indexes and filters. HashStore's filters and SortedStore's indexes are stored on flash for crash tolerance, allowing SILT to drop them from memory. This option saves memory at the cost of one additional flash read for the SortedStore, or two for the HashStore.

(2) Binary search on SortedStore. The SortedStore can be searched without an index, so the in-memory trie can be dropped even without storing a copy on flash, at the cost of $\log(n)$ additional reads from flash.

These techniques also help speed SILT's startup. By memory-mapping on-flash index files or performing binary search, SILT can begin serving requests before it has loaded its indexes into memory in the background.

# 5. ANALYSIS

Compared to single key-value store approaches, the multi-store design of SILT has more system parameters, such as the size of a single LogStore and HashStore, the total number of HashStores, the frequency to merge data into SortedStore, and so on. Having a much larger design space, it is preferable to have a systematic way to do parameter selection.

This section provides a simple model of the tradeoffs between write amplification (WA), read amplification (RA), and memory overhead (MO) in SILT, with an eye towards being able to set the system parameters properly to achieve the design goals from Section 2.

$$\mathrm{WA} = \frac{\text{data written to flash}}{\text{data written by application}}, \tag{1}$$

$$\mathrm{RA} = \frac{\text{data read from flash}}{\text{data read by application}}, \tag{2}$$

$$\mathrm{MO} = \frac{\text{total memory consumed}}{\text{number of items}}. \tag{3}$$

**Model** A SILT system has a flash drive of size $F$ bytes with a lifetime of $E$ erase cycles. The system runs $P$ SILT instances locally, each of which handles one disjoint range of keys using one LogStore, one SortedStore, and multiple HashStores. Once an instance has $d$ keys in total in its HashStores, it merges these keys into its SortedStore.

We focus here on a workload where the total amount of data stored in the system remains constant (e.g., only applying updates to existing keys). We omit for space the similar results when the data volume is growing (e.g., new keys are inserted to the system) and additional nodes are being added to provide capacity over time. Table 4 presents the notation used in the analysis.

**Write Amplification** An update first writes one record to the LogStore. Subsequently converting that LogStore to a HashStore incurs $1/0.93 = 1.075$ writes per key, because the space occupancy of the hash table is 93%. Finally, $d$ total entries (across multiple HashStores of one SILT instance) are merged into the existing SortedStore, creating a new SortedStore with $N/P$ entries. The total write amplification is therefore

$$\mathrm{WA} = 2.075 + \frac{N}{d \cdot P}. \tag{4}$$

**Read Amplification** The false positive rate of flash reads from a 4-way set associative hash table using $k$-bit tags is $f = 8/2^k$ because there are eight possible locations for a given key—two possible buckets and four items per bucket.

This 4-way set associative cuckoo hash table with $k$-bit tags can store $2^{k+2}$ entries, so at 93% occupancy, each LogStore and HashStore holds $0.93 \cdot 2^{k+2}$ keys. In one SILT instance, the number of items stored in HashStores ranges from 0 (after merging) to $d$, with an average size of $d/2$, so the average number of HashStores is

$$H = \frac{d/2}{0.93 \cdot 2^{k+2}} = 0.134\,\frac{d}{2^k}. \tag{5}$$

| Symbol | Meaning | Example |
|---|---|---|
| | **SILT design parameters** | |
| $d$ | maximum number of entries to merge | 7.5 M |
| $k$ | tag size in bit | 15 bits |
| $P$ | number of SILT instances | 4 |
| $H$ | number of HashStores per instance | 31 |
| $f$ | false positive rate per store | $2^{-12}$ |
| | **Workload characteristics** | |
| $c$ | key-value entry size | 1024 B |
| $N$ | total number of distinct keys | 100 M |
| $U$ | update rate | 5 K/sec |
| | **Storage constraints** | |
| $F$ | total flash size | 256 GB |
| $E$ | maximum flash erase cycle | 10,000 |

**Table 4: Notation.**

In the *worst-case* of a lookup, the system reads once from flash at the SortedStore, after $1+H$ failed retrievals at the LogStore and $H$ HashStores. Note that each LogStore or HashStore rejects all but an $f$ fraction of false positive retrievals; therefore, the expected total number of reads per lookup (read amplification) is:

$$\mathrm{RA} = (1+H)f + 1 = \frac{8}{2^k} + 1.07\frac{d}{4^k} + 1. \tag{6}$$

By picking $d$ and $k$ to ensure $1.07d/4^k + 8/2^k < \varepsilon$, SILT can achieve the design goal of read amplification $1+\varepsilon$.

**Memory Overhead** Each entry in LogStore uses $(k+1)/8+4$ bytes (k bits for the tag, one valid bit, and 4 bytes for the pointer). Each HashStore filter entry uses $k/8$ bytes for the tag. Each SortedStore entry consumes only 0.4 bytes. Using one LogStore, one SortedStore, and $H$ HashStores, SILT's memory overhead is:

$$\begin{aligned}\mathrm{MO} &= \frac{\left(\left(\frac{k+1}{8}+4\right)\cdot 2^{k+2} + \frac{k}{8}\cdot 2^{k+2}\cdot H + 0.4\cdot\frac{N}{P}\right)\cdot P}{N}\\ &= \left((16.5+0.5k)2^k + 0.067\,kd\right)\frac{P}{N} + 0.4.\end{aligned} \tag{7}$$

**Tradeoffs** Improving either write amplification, read amplification, or memory amplification comes at the cost of one of the other two metrics. For example, using larger tags (i.e., increasing $k$) reduces read amplification by reducing both $f$ the false positive rate per store and $H$ the number of HashStores. However, the HashStores then consume more DRAM due to the larger tags, increasing memory overhead. Similarly, by increasing $d$, SILT can merge HashStores into the SortedStore less frequently to reduce the write amplification, but doing so increases the amount of DRAM consumed by the HashStore filters. Figure 8 illustrates how write and read amplification change as a function of memory overhead when the maximum number of HashStore entries, $d$, is varied.

**Update Rate vs. Flash Life Time** The designer of a SILT instance handling $U$ updates per second wishes to ensure that the flash lasts for at least $T$ seconds. Assuming the flash device has perfect wear-leveling when being sent a series of large sequential writes [15], the total number of writes, multiplied by the write amplification WA, must not exceed the flash device size times its erase cycle budget. This creates a relationship between the lifetime, device size, update rate, and memory overhead:

$$U\cdot c\cdot \mathrm{WA}\cdot T \quad\leq\quad F\cdot E. \tag{8}$$

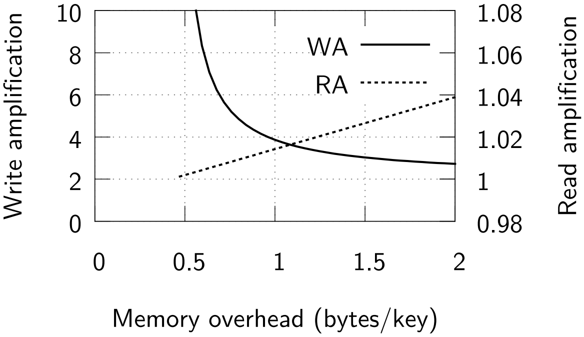

**Figure 8: WA and RA as a function of MO when *N*=100 *M*, *P*=4, and *k*=15, while *d* is varied.**

**Example** Assume a SILT system is built with a 256 GB MLC flash drive supporting 10,000 erase cycles [5] ($E = 10000$, $F = 256 \times 2^{30}$). It is serving $N = 100$ million items with $P = 4$ SILT instances, and $d = 7.5$ million. Its workload is 1 KiB entries, 5,000 updates per second ($U = 5000$).

By Eq. (4) the write amplification, WA, is 5.4. That is, each key-value update incurs 5.4 writes/entry. On average the number of HashStores is 31 according to Eq. (5). The read amplification, however, is very close to 1. Eq. (6) shows that when choosing 15 bits for the key fragment size, a `GET` incurs on average 1.008 of flash reads even when all stores must be consulted. Finally, we can see how the SILT design achieves its design goal of memory efficiency: indexing a total of 102.4 GB of data, where each key-value pair takes 1 KiB, requires only 73 MB in total or 0.73 bytes per entry (Eq. (7)). With the write amplification of 5.4 from above, this device will last 3 years.

## 6. EVALUATION

Using macro- and micro-benchmarks, we evaluate SILT's overall performance and explore how its system design and algorithms contribute to meeting its goals. We specifically examine (1) an end-to-end evaluation of SILT's throughput, memory overhead, and latency; (2) the performance of SILT's in-memory indexing data structures in isolation; and (3) the individual performance of each data store type, including flash I/O.

**Implementation** SILT is implemented in 15 K lines of C++ using a modular architecture similar to Anvil [29]. Each component of the system exports the same, basic key-value interface. For example, the classes which implement each of the three stores (LogStore, HashStore, and SortedStore) export this interface but themselves call into classes which implement the in-memory and on-disk data structures using that same interface. The SILT system, in turn, unifies the three stores and provides this key-value API to applications. (SILT also has components for background conversion and merging.)

**Evaluation System** We evaluate SILT on Linux using a desktop equipped with:

| | |
|---|---|
| CPU | Intel Core i7 860 @ 2.80 Ghz (4 cores) |
| DRAM | DDR SDRAM / 8 GiB |
| SSD-L | Crucial RealSSD C300 / 256 GB |
| SSD-S | Intel X25-E / 32 GB |

The 256 GB SSD-L stores the key-value data, and the SSD-S is used as scratch space for sorting HashStores using Nsort [33]. The drives connect using SATA and are formatted with the ext4 filesystem

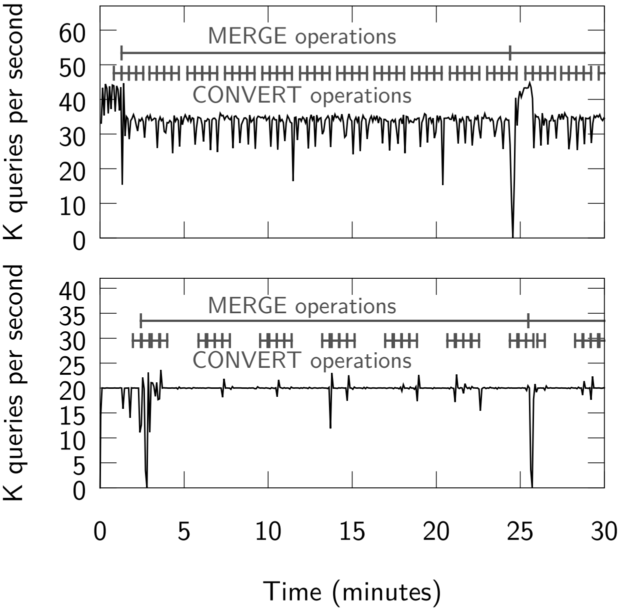

**Figure 9: `GET` throughput under high (upper) and low (lower) loads.**

using the `discard` mount option (TRIM support) to enable the flash device to free blocks from deleted files. The baseline performance of the data SSD is:

| | |
|---|---|
| Random Reads (1024 B) | 48 K reads/sec |
| Sequential Reads | 256 MB/sec |
| Sequential Writes | 233 MB/sec |

## 6.1 Full System Benchmark

**Workload Generation** We use YCSB [17] to generate a key-value workload. By default, we use a 10% PUT / 90% GET workload for 20-byte keys and 1000-byte values, and we also use a 50% PUT / 50% GET workload for 64-byte key-value pairs in throughput and memory overhead benchmarks. To avoid the high cost of the Java-based workload generator, we use a lightweight SILT client to replay a captured trace file of queries made by YCSB. The experiments use four SILT instances ($P = 4$), with 16 client threads concurrently issuing requests. When applicable, we limit the rate at which SILT converts entries from LogStores to HashStores to 10 K entries/second, and from HashStores to the SortedStore to 20 K entries/second in order to prevent these background operations from exhausting I/O resources.

*Throughput:* SILT can sustain an average insert rate of 3,000 1 KiB key-value pairs per second, while simultaneously supporting 33,000 queries/second, or 69% of the SSD's random read capacity. With no inserts, SILT supports 46 K queries per second (**96**% of the drive's raw capacity), and with no queries, can sustain an insert rate of approximately 23 K inserts per second. On a deduplication-like workload with 50% writes and 50% reads of 64 byte records, SILT handles 72,000 requests/second.

SILT's performance under insert workloads is limited by the time needed to convert and merge data into HashStores and SortedStores. These *background operations* compete for flash I/O resources,

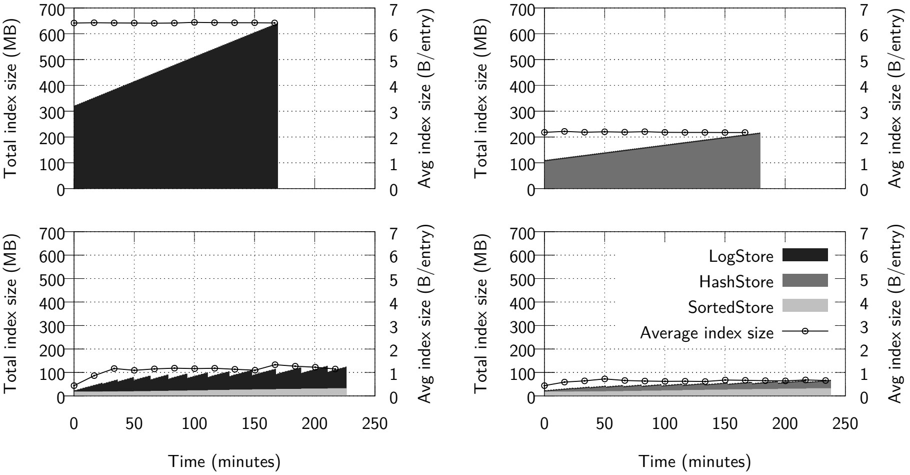

**Figure 10: Index size changes for four different store combinations while inserting new 50 M entries.**

resulting in a tradeoff between query latency and throughput. Figure 9 shows the sustainable query rate under both high query load (approx. 33 K queries/second) and low query load (22.2 K queries/second) for 1 KiB key-value pairs. SILT is capable of providing predictable, low latency, or can be tuned for higher overall throughput. The middle line shows when SILT converts LogStores into HashStores (periodically, in small bursts). The top line shows that at nearly all times, SILT is busy merging HashStores into the SortedStore in order to optimize its index size.[4] In Section 6.3, we evaluate in more detail the speed of the individual stores and conversion processes.

*Memory overhead*: SILT meets its goal of providing high throughput with low memory overhead. We measured the time and memory required to insert 50 million new 1 KiB entries into a table with 50 million existing entries, while simultaneously handling a high query rate. SILT used at most 69 MB of DRAM, or 0.69 bytes per entry. (This workload is worst-case because it is never allowed time for SILT to compact all of its HashStores into the SortedStore.) For the 50% PUT / 50% GET workload with 64-byte key-value pairs, SILT required at most 60 MB of DRAM for 100 million entries, or 0.60 bytes per entry.

The drastic improvement in memory overhead from SILT's three-store architecture is shown in Figure 10. The figure shows the memory consumption during the insertion run over time, using four different configurations of basic store types and 1 KiB key-value entries. The bottom right graph shows the memory consumed using the full SILT system. The bottom left configuration omits the intermediate HashStore, thus requiring twice as much memory as the full SILT configuration. The upper right configuration instead omits the SortedStore, and consumes four times as much memory. Finally, the upper left configuration uses only the basic LogStore, which requires nearly 10x as much memory as SILT. To make this comparison fair, the test generates unique new items so that garbage collection of old entries cannot help the SortedStore run faster.

The figures also help understand the modest cost of SILT's memory efficiency. The LogStore-only system processes the 50 million inserts (500 million total operations) in under 170 minutes, whereas

[4] In both workloads, when merge operations complete (e.g., at 25 minutes), there is a momentary drop in query speed. This is due to bursty TRIMming by the ext4 filesystem implementation (`discard`) used in the experiment when the previous multi-gigabyte SortedStore file is deleted from flash.

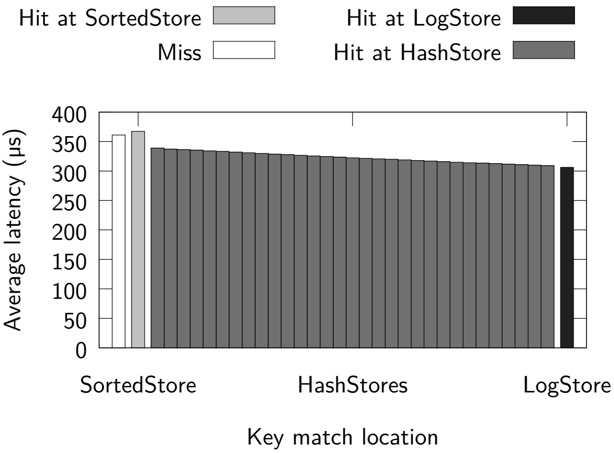

**Figure 11: `GET` query latency when served from different store locations.**

the full SILT system takes only 40% longer–about 238 minutes–to incorporate the records, but achieves an order of magnitude better memory efficiency.

*Latency:* SILT is fast, processing queries in 367 $\mu$s on average, as shown in Figure 11 for 100% `GET` queries for 1 KiB key-value entries. GET responses are fastest when served by the LogStore (309 $\mu$s), and slightly slower when they must be served by the SortedStore. The relatively small latency increase when querying the later stores shows the effectiveness (reducing the number of extra flash reads to $\varepsilon < 0.01$) and speed of SILT's in-memory filters used in the Log and HashStores.

In the remaining sections, we evaluate the performance of SILT's individual in-memory indexing techniques, and the performance of the individual stores (in-memory indexes plus on-flash data structures).

## 6.2 Index Microbenchmark

The high random read speed of flash drives means that the CPU budget available for each index operation is relatively limited. This microbenchmark demonstrates that SILT's indexes meet their design goal of computation-efficient indexing.

**Experiment Design** This experiment measures insertion and lookup speed of SILT's in-memory partial-key cuckoo hash and entropy-coded trie indexes. The benchmark inserts 126 million total entries and looks up a subset of 10 million random 20-byte keys.

This microbenchmark involves memory only, no flash I/O. Although the SILT system uses multiple CPU cores to access multiple indexes concurrently, access to individual indexes in this benchmark is single-threaded. Note that inserting into the cuckoo hash table (LogStore) proceeds key-by-key, whereas the trie (SortedStore) is constructed en mass using bulk insertion. Table 5 summarizes the measurement results.

**Individual Insertion Speed (Cuckoo Hashing)** SILT's cuckoo hash index implementation can handle 10.18 M 20-byte key insertions (`PUT`s or `DELETE`s) per second. Even at a relatively small, higher overhead key-value entry size of 32-byte (i.e., 12-byte data), the index would support 326 MB/s

| Type | Cuckoo hashing<br>(K keys/s) | Trie<br>(K keys/s) |
|---|---|---|
| Individual insertion | 10182 | – |
| Bulk insertion | – | 7603 |
| Lookup | 1840 | 208 |

**Table 5: In-memory performance of index data structures in SILT on a single CPU core.**

| Type | Speed (K keys/s) |
|---|---|
| LogStore (by `PUT`) | 204.6 |
| HashStore (by `CONVERT`) | 67.96 |
| SortedStore (by `MERGE`) | 26.76 |

**Table 6: Construction performance for basic stores. The construction method is shown in the parentheses.**

of incoming key-value data on one CPU core. This rate exceeds the typical sequential write speed of a single flash drive: inserting keys into our cuckoo hashing is unlikely to become a bottleneck in SILT given current trends.

**Bulk Insertion Speed (Trie)** Building the trie index over 126 million pre-sorted keys required approximately 17 seconds, or 7.6 M keys/second.

**Key Lookup Speed** Each SILT `GET` operation requires a lookup in the LogStore and potentially in one or more HashStores and the SortedStore. A single CPU core can perform 1.84 million cuckoo hash lookups per second. If a SILT instance has 1 LogStore and 31 HashStores, each of which needs to be consulted, then one core can handle about 57.5 K `GET`s/sec. Trie lookups are approximately 8.8 times slower than cuckoo hashing lookups, but a `GET` triggers a lookup in the trie only after SILT cannot find the key in the LogStore and HashStores. When combined, the SILT indexes can handle about $1/(1/57.5\text{ K} + 1/208\text{ K}) \approx 45$ K `GET`s/sec with one CPU core.

Insertions are faster than lookups in cuckoo hashing because insertions happen to only a few tables at the same time and thus benefit from the CPU's L2 cache; lookups, however, can occur to any table in memory, making CPU cache less effective.

**Operation on Multiple Cores** Using four cores, SILT indexes handle 180 K `GET`s/sec in memory. At this speed, the indexes are unlikely to become a bottleneck: their overhead is on-par or lower than the operating system overhead for actually performing that many 1024-byte reads per second from flash. As we see in the next section, SILT's overall performance is limited by sorting, but its index CPU use is high enough that adding many more flash drives would require more CPU cores. Fortunately, SILT offers many opportunities for parallel execution: Each SILT node runs multiple, completely independent instances of SILT to handle partitioning, and each of these instances can query many stores.

## 6.3 Individual Store Microbenchmark

Here we measure the performance of each SILT store type in its entirety (in-memory indexing plus on-flash I/O). The first experiment builds multiple instances of each basic store type with 100 M key-value pairs (20-byte key, 1000-byte value). The second experiment queries each store for 10 M random keys.

| Type | SortedStore<br>(K ops/s) | HashStore<br>(K ops/s) | LogStore<br>(K ops/s) |
|---|---|---|---|
| GET (hit) | 46.57 | 44.93 | 46.79 |
| GET (miss) | 46.61 | 7264 | 7086 |

**Table 7: Query performance for basic stores that include in-memory and on-flash data structures.**

Table 6 shows the construction performance for all three stores; the construction method is shown in parentheses. LogStore construction, built through entry-by-entry insertion using PUT, can use 90% of sequential write bandwidth of the flash drive. Thus, SILT is well-suited to handle bursty inserts. The conversion from LogStores to HashStores is about three times slower than LogStore construction because it involves bulk data reads and writes from/to the same flash drive. SortedStore construction is slowest, as it involves an external sort for the entire group of 31 HashStores to make one SortedStore (assuming no previous SortedStore). If constructing the SortedStore involved merging the new data with an existing SortedStore, the performance would be worse. The large time required to create a SortedStore was one of the motivations for introducing HashStores rather than keeping un-merged data in LogStores.

Table 7 shows that the minimum GET performance across all three stores is 44.93 K ops/s. Note that LogStores and HashStores are particularly fast at GET for non-existent keys (more than 7 M ops/s). This extremely low miss penalty explains why there was only a small variance in the average GET latency in Figure 11 where bad cases looked up 32 Log and HashStores and failed to find a matching item in any of them.

# 7. RELATED WORK

**Hashing** *Cuckoo hashing* [34] is an open-addressing scheme to resolve hash collisions efficiently with high space occupancy. Our partial-key cuckoo hashing—storing only a small part of the key in memory without fetching the entire keys from slow storage on collisions—makes cuckoo hashing more memory-efficient while ensuring high performance.

*Minimal perfect hashing* is a family of collision-free hash functions that map $n$ distinct keys to $n$ consecutive integers $0 \ldots n-1$, and is widely used for memory-efficient indexing. In theory, any minimal perfect hash scheme requires at least 1.44 bits/key [27]; in practice, the state-of-the-art schemes can index any static data set with 2.07 bits/key [10]. Our entropy-coded trie achieves 3.1 bits/key, but it also preserves the lexicographical order of the keys to facilitate data merging. Thus, it belongs to the family of monotone minimal perfect hashing (MMPH). Compared to other proposals for MMPH [8, 9], our trie-based index is simple, lightweight to generate, and has very small CPU/memory overhead.

**External-Memory Index on Flash** Recent work such as MicroHash [40] and FlashDB [32] minimizes memory consumption by having indexes on flash. MicroHash uses a hash table chained by pointers on flash. FlashDB proposes a self-tuning $B^+$-tree index that dynamically adapts the node representation according to the workload. Both systems are optimized for memory and energy consumption of sensor devices, but not for latency as lookups in both systems require reading multiple flash pages. In contrast, SILT achieves very low memory footprint while still supporting high throughput.

**Key-Value Stores** HashCache [4] proposes several policies to combine hash table-based in-memory indexes and on-disk data layout for caching web objects. FAWN-DS [2] consists of an on-flash data log and in-memory hash table index built using relatively slow CPUs with a limited amount of memory. SILT dramatically reduces DRAM consumption compared to these systems by combining more memory-efficient data stores with minimal performance impact. FlashStore [19] also uses a single hash table to index all keys on flash similar to FAWN-DS. The flash storage, however, is used as a cache of a hard disk-backed database. Thus, the cache hierarchy and eviction algorithm is orthogonal to SILT. To achieve low memory footprint (about 1 byte/key), SkimpyStash [20] moves its indexing hash table to flash with linear chaining. However, it requires on average 5 flash reads per lookup, while SILT only needs $1+\varepsilon$ per lookup.

More closely related to our design is BufferHash [1], which keeps keys in multiple equal-sized hash tables—one in memory and the others on flash. The on-flash tables are guarded by in-memory Bloom filters to reduce unnecessary flash reads. In contrast, SILT data stores have different sizes and types. The largest store (SortedStore), for example, does not have a filter and is accessed at most once per lookup, which saves memory while keeping the read amplification low. In addition, writes in SILT are appended to a log stored on flash for crash recovery, whereas inserted keys in BufferHash do not persist until flushed to flash in batch.

Several key-value storage libraries rely on caching to compensate for their high read amplifications [6, 28], making query performance depend greatly on whether the working set fits in the in-memory cache. In contrast, SILT provides uniform and predictably high performance regardless of the working set size and query patterns.

**Distributed Key-Value Systems** Distributed key-value storage clusters such as BigTable [14], Dynamo [21], and FAWN-KV [2] all try to achieve high scalability and availability using a cluster of key-value store nodes. SILT focuses on how to use flash memory-efficiently with novel data structures, and is complementary to the techniques used in these other systems aimed at managing failover and consistency.

**Modular Storage Systems** BigTable [14] and Anvil [29] both provide a modular architecture for chaining specialized stores to benefit from combining different optimizations. SILT borrows its design philosophy from these systems; we believe and hope that the techniques we developed for SILT could also be used within these frameworks.

# 8. CONCLUSION

SILT combines new algorithmic and systems techniques to balance the use of memory, storage, and computation to craft a memory-efficient, high-performance flash-based key value store. It uses two new in-memory index structures—partial-key cuckoo hashing and entropy-coded tries—to reduce drastically the amount of memory needed compared to prior systems. SILT chains the right combination of basic key-value stores together to create a system that provides high write speed, high read throughput, and uses little memory, attributes that no single store can achieve alone. SILT uses in total only 0.7 bytes of memory per entry it stores, and makes only 1.01 flash reads to service a lookup, doing so in under 400 microseconds. Our hope is that SILT, and the techniques described herein, can form an efficient building block for a new generation of fast data-intensive services.

# ACKNOWLEDGMENTS

This work was supported by funding from National Science Foundation award CCF-0964474, Google, the Intel Science and Technology Center for Cloud Computing, by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. Hyeontaek Lim is supported in part by the Korea Foundation for Advanced Studies. We thank the SOSP reviewers, Phillip B. Gibbons, Vijay Vasudevan, and Amar Phanishayee for their feedback, Guy Blelloch and Rasmus Pagh for pointing out several algorithmic possibilities, and Robert Morris for shepherding this paper.

# REFERENCES

[1] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *NSDI'10: Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 29–29, Berkeley, CA, USA, 2010. USENIX Association.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. SOSP*, Big Sky, MT, Oct. 2009.

[3] A. Andersson and S. Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46(6):295–300, 1993.

[4] A. Badam, K. Park, V. S. Pai, and L. L. Peterson. HashCache: Cache storage for the next billion. In *Proc. 6th USENIX NSDI*, Boston, MA, Apr. 2009.

[5] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi. Differential RAID: Rethinking RAID for SSD reliability. In *Proc. European Conference on Computer Systems (Eurosys)*, Paris, France, 2010.

[6] Berkeley DB. http://www.oracle.com/technetwork/database/berkeleydb/, 2011.

[7] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, Oct. 2010.

[8] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: searching a sorted table with O(1) accesses. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 785–794, 2009.

[9] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Theory and practise of monotone minimal perfect hashing. In *Proc. 11th Workshop on Algorithm Engineering and Experiments*, ALENEX '09, 2009.

[10] D. Belazzougui, F. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. In *Proceedings of the 17th European Symposium on Algorithms*, ESA '09, pages 682–693, 2009.

[11] D. K. Blandford, G. E. Blelloch, and I. A. Kash. An experimental analysis of a compact graph representation. In *Proc. 6th Workshop on Algorithm Engineering and Experiments*, ALENEX '04, 2004.

[12] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[13] F. C. Botelho, A. Lacerda, G. V. Menezes, and N. Ziviani. Minimal perfect hashing: A competitive method for indexing internal memory. *Information Sciences*, 181:2608–2625, 2011.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. 7th USENIX OSDI*, Seattle, WA, Nov. 2006.

[15] L.-P. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing (SAC '07)*, Mar. 2007.

[16] D. R. Clark. *Compact PAT trees*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1998.

[17] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, Indianapolis, IN, June 2010.

[18] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[19] B. Debnath, S. Sengupta, and J. Li. FlashStore: High throughput persistent key-value store. *Proc. VLDB Endowment*, 3:1414–1425, September 2010.

[20] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proc. International Conference on Management of Data*, ACM SIGMOD '11, pages 25–36, New York, NY, USA, 2011.

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[22] J. Dong and R. Hull. Applying approximate order dependency to reduce indexing space. In *Proc. ACM SIGMOD International Conference on Management of data*, SIGMOD '82, pages 119–127, 1982.

[23] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, Mar. 1975.

[24] Ú. Erlingsson, M. Manasse, and F. Mcsherry. A cool and practical alternative to traditional hash tables. In *Proc. 7th Workshop on Distributed Data and Structures (WDAS'06)*, CA, USA, 2006.

[25] Facebook. `http://www.facebook.com/`, 2011.

[26] Flickr. `http://www.flickr.com/`, 2011.

[27] E. A. Fox, L. S. Heath, Q. F. Chen, and A. M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35:105–121, Jan. 1992.

[28] S. Ghemawat and J. Dean. LevelDB. `https://code.google.com/p/leveldb/`, 2011.

[29] M. Mammarella, S. Hovsepian, and E. Kohler. Modular data storage with Anvil. In *Proc. SOSP*, Big Sky, MT, Oct. 2009.

[30] Memcached: A distributed memory object caching system. `http://www.danga.com/memcached/`, 2011.

[31] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. In *Proc. VLDB*, Auckland, New Zealand, Aug. 2008.

[32] S. Nath and A. Kansal. FlashDB: Dynamic self-tuning database for NAND flash. In *Proc. ACM/IEEE International Conference on Information Processing in Sensor Networks*, Cambridge, MA, Apr. 2007.

[33] C. Nyberg and C. Koester. Ordinal Technology - NSort. `http://www.ordinal.com/`, 2011.

[34] R. Pagh and F. Rodler. Cuckoo hashing. *Journal of Algorithms*, (2):122–144, May 2004.

[35] M. Polte, J. Simsa, and G. Gibson. Enabling enterprise solid state disks performance. In *Proc. Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, Washington, DC, Mar. 2009.

[36] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, CA, Jan. 2002.

[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, San Diego, CA, Aug. 2001.

[38] Twitter. http://www.twitter.com/, 2011.

[39] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. *Lecture Notes in Computer Science*, 1995.

[40] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. MicroHash: An efficient index structure for flash-based sensor devices. In *Proc. 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, Dec. 2005.