Generic Design Patterns for Tunable and High-Performance SSD-based Indexes

Ashok Anand[†]*, Aaron Gember*, Aditya Akella**
[†]*Bell Labs-India* **University of Wisconsin-Madison*

Abstract

A number of data-intensive systems require using random hash-based indexes of various forms, e.g., hashtables, Bloom filters, and locality sensitive hash tables. In this paper, we present general SSD optimization techniques that can be used to design a variety of such indexes while ensuring higher performance and easier tunability than specialized state-of-the-art approaches.

We leverage two key SSD innovations: a) rearranging the data layout on the SSD to combine multiple read requests into one page read, and b) intelligent request reordering to exploit inherent parallelism in the architecture of SSDs. We build three different indexes using these techniques and conduct extensive studies showing their superior performance and flexibility.

1 Introduction

Data-intensive systems are being employed in a wide variety of application scenarios today. For example, key-value storage systems are employed in cloud-based applications as diverse as e-commerce and business analytics systems, and picture stores; and large object stores are used in a variety of content-based systems such as network deduplication engines, storage deduplication, logging systems and content similarity detection engines. To ensure high application performance these systems often rely on random hashing-based indexes, whose specific design may depend on the system in question. For instance, WAN optimizers [5, 6], Web caches [4, 7], and video caches [2], employ large streaming hash tables. De-duplication systems [26, 31] employ bloom filters to summarize the underlying object stores. Content similarity engines and some video proxies [11, 2] employ locality sensitive hash tables [22]. Given the volume of the underlying data, the indexes often span several 10s of GB, and they continue to grow in size.

Across these systems, the index is the most intricate in design. Heavy engineering is often devoted to ensure high index performance (low latency and high throughput) at low cost (cost of sub-components used to store the index, as well as the energy they consume). Most state-of-the-art systems [17, 23, 12, 19] advocate using SSDs to store the indexes, given flash-based media's superior density, 8X lower cost (vs. DRAM), 25X better energy efficiency (vs. DRAM or disk), and high random read performance (vs. disk) [23]. However, the commonality ends here. The conventional wisdom, which universally dictates index design, is that domain- and operations-specific SSD optimizations are necessary to meet appropriate cost-performance trade-offs. This poses two problems: (a) *Poor flexibility:* The index designs often target a specific point in the cost-performance spectrum, severely limiting the range of applications that can use them. It also makes indexes difficult to tune (e.g., use extra memory for improved performance). Finally, the indexes are designed to work best under specific workloads; minor deviations can make performance quite variable. (b) *Poor generality:* The design patterns employed apply only to the specific data structure on hand, placing a high bar on innovation. In particular, it is difficult to employ different indexes in tandem (e.g., hash tables for cache lookup alongside LSH tables for content similarity detection over the same underlying content) as they may employ conflicting techniques that result in poor SSD I/O performance.

Our paper questions the conventional wisdom. We present different indexes that all leverage a common set of novel SSD optimizations, are easy to tune to achieve optimal performance under a given cost constraint, and support widely-varying workload patterns and applications with differing resource requirements; yet, they offer better IOPS, cost less and consume lower energy than their counterparts with specialized designs.

We rely on two key innovations: (1) We develop a new technique called *slicing* where we organize data on the SSD such that related entries are located together. Slicing allows us to combine multiple reads into a single "slice read" of related items, offering high read performance. The size of a slice can be tuned to control I/O cost. (2) We leverage a unique feature of SSDs that has been overlooked by earlier proposals, namely, the internal architecture of SSDs offers parallelism at multiple levels, e.g., channel-level, package-level, die-level and plane-level. Critically, the parallelism benefits are significant only under certain I/O patterns. Our key insight lies in identifying these patterns and encapsulating regular I/O workloads into them to provide high performance.

We profile the internal parallelism behavior on a desktop-grade SSD to derive parallelism-friendly I/O patterns and understand how to configure slices (§3). We then present the design of three random-hash based indexes that leverage slicing and parallelism: a streaming

hash table called SliceHash presented in §4, large Bloom filters called SliceBloom and locality-sensitive hash tables called SliceLSH; the latter two are presented in §5.

Our index designs can be sketched as follows: We use small in-memory data structures (hash tables, Bloom filters or LSH tables, as the case may be) as *buffers* for insert operations to deal with the well-known problem of slow random writes on SSDs. When full, these are flushed to the SSD; each of these flushed data structures is called an incarnation. We organize data on the SSD such that all related entries of different incarnations are located together in a slice, thereby optimizing lookup. Finally, based on an understanding the SSD's writing policy, we appropriately reorder requests (without violating application semantics) to distribute them uniformly across different channels and extract maximal parallelism benefits. In addition to supporting high performance, the buffering and slicing primitives used in our indexes eliminate the need to maintain complex metadata to aid index I/O operations. This frees memory and compute resources for use by higher layer applications. We show that the primitives also facilitate extending the indexes to use multiple SSDs on the same machine, offering linear scaling in performance while imposing sublinear scaling in memory and CPU overhead. State-of-the-art techniques cannot be "scaled out" in a similar fashion.

We build prototype indexes using a 128GB Crucial SSD and at most 4GB of DRAM. We conduct extensive experiments under a range of realistic workloads to show that our design patterns offer high performance, flexibility and generality. Key findings from our evaluation are as follows: On a single SSD, SliceHash can provide 50K lookups/sec by intelligently exploiting parallelism, which can be 4.5X better than naively running multiple lookups in parallel. Performance is steady even with arbitrarily interleaved inserts, whereas state-of-the-art systems take a 20-30% performance hit. SliceHash can be tuned to use progressively more memory (from 0.3B/entry to 2B/entry) to scale performance (from 50K to 75K ops/s). When leveraging 3 SSDs in parallel SliceHash's throughput improves to between 150K (read-only) and 188K (read/write) ops/sec. SliceBloom performs 12-15K ops/sec mixed read/write workload, whereas state-of-the-art [20] achieves similar performance on a high-end SSD that costs 30X. SliceLSH performs 4.9K lookups/s.

2 Design Requirements

Our goal is to develop *generic* SSD design optimizations that can be applied *nearly universally* to a variety of random hash-based indexes that each have the following requirements:

Large scale: A number of data-intensive networking applications require large indexes. For example, WAN optimizer [6, 5] indexes are 16-32GB; data de-duplication indexes are 20 GB [3]. In keeping with the trend of growing data volumes, we target indexes that are an order-of-magnitude larger, i.e., a few hundred GB in size.

High performance and low cost: The index should provide high throughput and low per-operation latency, and have low overall cost, memory and energy footprint. To apply to a wide-variety of content-based systems, the index should provide good performance under both inserts/updates and reads. The state-of-the art techniques for hash tables offer 46K IOPS; those for bloom filters offer 12-15K IOPS. Our indexes should match or exceed this performance.

Flexibility: Applications that leverage these indexes require significant CPU and memory resources for their internal operations. For example, data de-duplication applications require CPU for computing SHA-1 hashes of fingerprints. Various image and video search applications require CPU resources for computing similarity metrics after they find potential matches. Caching applications may want to use memory for caching frequently or recently accessed content. To ensure that the applications can flexibly use CPU and memory and that their performance does not suffer, the index should impose low CPU and memory overhead. Unfortunately, many prior index designs ignore the high CPU overhead they impose in their singular quest for, e.g., low memory footprint and high read performance (e.g., SILT [23]), which makes application design tricky. Equally importantly, application designers should be able to easily extend the index with evolving application requirements, e.g., add memory or CPU cores at a modest additional cost to obtain commensurately better performance.

In the rest of this section, we survey other related hash-based systems that employ flash storage. As stated earlier, none of these studies use techniques that are all generally applicable across different random hash-based indexes. Even ignoring this issue, all prior designs fall short on one or more of the above requirements.

2.1 Trade-Offs in Example Flash Indexes

In this section, we review a specific class of indexes, namely those based on hash tables, to highlight the design choices made and the restrictions those impose. We discuss prior work on optimizing I/O patterns on SSDs in §3 and §4. We discuss work related to other forms of indexes in §5.

Many recent research proposals [18, 19, 12, 17, 23] have proposed SSD-based indexes for large key-value stores. While these systems are designed to deal with slow random writes of flash storage, they are not designed for exploiting intrinsic parallelism of SSDs. As we show in §3, SSD lookup performance can be 4.5X better if underlying parallelism is exploited opti-

Metrics	FlashStore	SkimpyStash	BufferHash	SILT
Lookup (#page read)	∼1	∼**5**	∼1	∼1
Memory (# bytes/entry)	∼**6**	∼1	∼4	∼0.7
CPU overhead	Low	Low	Low	**High**

Table 1: Comparison of different Flash-based Hashtables under different metrics.

mally. Furthermore, these systems do not simultaneously optimize for different performance metrics (i.e., high throughput, low latency, low memory footprint and low computation overhead). As Table 1 shows, they optimize for some of the metrics, but at the expense of significantly impacting others.

FlashStore[18] stores key-value pairs in a log-structured fashion on flash storage and uses an in-memory hashtable to index them. It requires one flash read/lookup but has a high memory footprint (∼6 bytes/key). *SkimpyStash* [19] uses 1 byte/key by maintaining a hashtable with linear chaining on flash; however, it requires 5 page reads/lookup on average.

BufferHash [12] buffers all insertions in the memory, and writes them in a batch on flash. It maintains in-memory Bloom filters [8] to avoid spurious lookups to any batch on flash. BufferHash requires ∼1 page read per lookup on average, but may need to scan multiple pages in the worst case due to false positives of Bloom filters. Furthermore, BufferHash requires ∼4 bytes/key.

SILT [23] comes close to meeting the design requirements outlined above. SILT achieves a low memory footprint (0.7 bytes/entry) and requires a single page lookup on average. However, SILT uses a much more complex design than other systems discussed above. It employs three data structures, where one of them is highly optimized for a low memory footprint, and the others are more write-optimized but require more memory. SILT continuously moves data from the write-optimized data structures to the memory-efficient data structure. In doing so, SILT has to continuously sort new data written and merge it with old data, thereby increasing the computation overhead. These background operations also affect the performance of SILT under continuous inserts and lookups. For example, the lookup performance drops by 21% for a 50% lookup-50% insert workload on 64B key-value pairs. The authors also acknowledge that sorting becomes performance bottleneck.

Other recent works, *MicroHash* [21] and *FlashDB* [28], also maintain hashtables on SSDs to reduce the memory footprint. However, these systems are designed for memory constrained devices with the goal of optimizing the memory footprint and energy. Unfortunately, they end up significantly increasing the latency of lookups. For example, MicroHash requires looking up multiple pages to locate the desired key.

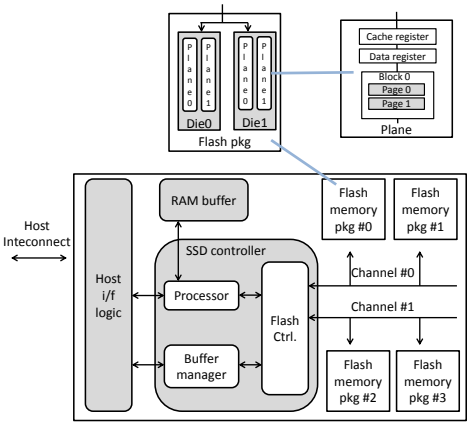


Figure 1: SSD internal architecture

3 Exploiting Novel Properties of SSDs

We present key properties of flash SSDs that influence the designs of large random hash-based indexes. We start by revisiting known properties of flash SSDs. We describe how best to leverage them in designing various indexes. We then describe flash SSDs' inherently parallel architecture and techniques to leverage parallelism.

Page-level reads: In flash SSDs, a page (2KB-4KB) is the smallest unit of read or write operations. This implies that reading a 16B entry (e.g., a key-value pair in a hashtable) is as costly as reading a page. Thus, if we can organize data in such a way that multiple entries to be read can reside on the same page, we can reduce the number of page reads.

Avoiding random writes: Pages are organized into blocks, each typically spanning 32 or 64 pages. It is now well known that the performance of random page reads is comparable to that of sequential page reads. However, random page writes (or in-place updates) are slow. Earlier works have shown poor performance under high random page writes [29]. Even the random read performance is affected in a mixed workload of continuous reads and writes [12]. So, we must avoid random page writes as much as possible.

3.1 Inherent Parallelism in Flash SSDs

Flash SSDs have a highly parallel internal architecture. This intrinsic parallelism can be a great source for high performance, but it has not been carefully exploited in prior works. We first provide an overview of the parallelism before discussing guidelines for systematically leveraging it for random hash-based indexes.

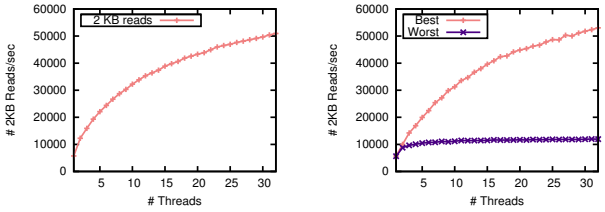
Architecture: Figure 1 shows an illustration of a SATA-based SSD architecture. All I/O requests are processed by *an SSD controller*. The controller receives I/O requests from the host via an interface connection (i.e., the SATA interface). It uses the FTL to translate logical pages of incoming requests to physical pages, and it issues commands to *flash packages* via *flash memory con-*

trollers. The flash memory controller connects to flash packages via *multiple channels* (generally 2-10). Each package has two or more *dies* or chips. Each die is composed of two or more *planes*. Each plane has a data register to temporarily store the data page during reads or writes. For a write command, the controller first transfers data to a *data register* on a channel, and then the data is written from the data register to the corresponding physical page. For a read command, the data is first read from the physical page to the data register, and then it is transferred from the data register to the controller on a channel. This architecture incorporates varying degrees and levels of parallelism, as discussed next.

Forms of parallelism available: Each of an SSD's channels can operate in parallel and independently of each other. Thus, SSDs inherently have *channel-level parallelism*. Typically, the data transfers from/to the multiple packages on the same channel get serialized. However, data transfers can be interleaved with other operations (e.g., reading data from a page to the data register) on other packages sharing the same channel [10, 27]. This interleaving provides *package-level parallelism*. The FTL stripes consecutive logical pages across a gang of different packages on the same channel [10] to expoit package-level parallelism. The command issued to a die can be executed independently of the others on the same package. This provides *die-level parallelism*.

Multiple operations of the same type (read/write/erase) can happen simultaneously on different planes in the same die. Currently, a *two plane command* is widely used for executing two operations of the same type on two different planes simultaneously [25]. This provides *plane-level parallelism*. Furthermore, the data transfers to/from the physical page can be *pipelined* for consecutive commands of the same type. This is achieved using a cache register in the plane. For consecutive write commands, the cache register stores the data temporarily until the previous data is written from the data register to the physical page. The cache register is similarly used for pipelining read commands.

Leveraging parallelism: While the above forms of parallelism have existed in most SSD designs, support for concurrent I/O operations was not available [14]. This is important for SSDs since the parallel architecture can be leveraged only when there are parallel I/O requests. Recently, flash SSDs have begun to support native command queueing (NCQ). With NCQ, multiple I/O operations can execute concurrently and leverage the inherent parallelism. In the Crucial M4 SSD, NCQ allows up to 32 I/O requests to run in parallel.



(a) Random lookups (b) Best vs. Worst

Figure 2: Concurrent lookup performance

3.2 Extracting the Benefits of Parallelism

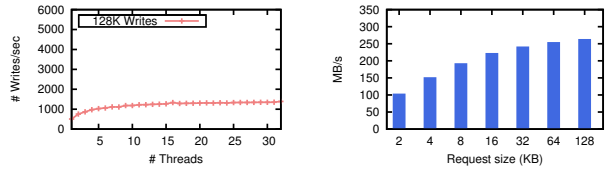
3.2.1 Channel-level Parallelism Benefits

Reading data from a physical page to the data register typically takes $\sim 25\mu$s. Data transfers on the channel take roughly 100 μs [10]. Thus, transfer time on the channel is the primary bottleneck for page reads. If we can leverage channel-level parallelism, the throughput of page reads can be significantly improved.

A naive approach: A naive way to extract the benefits of parallelism is to simply use multiple threads issuing requests in parallel. Figure 2 (a) shows that by issuing multiple requests in parallel and increasing the depth of the I/O queue, the overall throughput is improved by up to 9X compared to single thread performance.

Intelligent request scheduling: However, is just issuing multiple requests in parallel enough? Note that the benefits of parallelism would depend on how page read requests get distributed over channels. For example, if all requests go to the same channel, parallel lookups may not provide any benefits. If requests are uniformly distributed over different channels, we can exploit channel-level parallelism maximally. A related question is, how far apart are the best and worst case throughput? This determines the extent to which we need to control how reads are performed to exploit channel-level parallelism.

To issue requests in a manner that ideally exploits parallelism, we need to know the mapping between pages and channels. However, this is often internal to flash SSDs and not exposed by vendors. Recently, in [16], the authors devised a method to reverse engineer the mapping. As discussed earlier, the FTL stripes a group of consecutive logical pages across different packages on the same channel. The authors discuss a technique to determine the size of the group that gets contiguously allocated within a channel; they call this logical unit of data a *chunk*. They show how to determine the chunk size and the number of channels in the SSD. Using this, they also show how to derive the mapping policy. In particular, they discuss techniques for deriving two common mapping policies: (a) *write-order mapping*, where the i^{th} chunk write is assigned the channel $i \% N$, assuming N is the number of channels, and (b) *LBA-based mapping*, where the logical block address (LBA) is mapped to a



(a) Concurrent large writes (b) Effect of large reads

Figure 3: Concurrent large I/O performance

channel based on *LBA % N*.

Using their technique, we estimate the chunk size and number of channels to be 8KB and 32, respectively for the Crucial SSD. We further find that the Crucial SSD follows write-order mapping. Using knowledge of the order of writes to the SSD, we can determine the channel corresponding to a page. This enables us to determine how to schedule requests to spread them across channels.

Figure 2(b) shows the gap between the best and worst cases for different numbers of threads based on our estimates of the flash channel count and mapping policies, illustrating the benefits of intelligently exploiting parallelism. To study the worst case, we force requests to go to the same channel. We find that the gap is nearly 4.5X. The big performance gap indicates that for exploiting intrinsic parallelism maximally, we should be careful about how requests are issued. We can reorder requests such that those issued concurrently are uniformly distributed over channels. Given accurate knowledge of the order of writes, the potential benefit from such reordering can be as high as 4.5X. For example, if we have a set of 1024 requests such that the first 32 requests go to first channel, the second set of 32 requests go to second channel and so on, we can reorder them by picking one request from each set and issuing them in parallel.

Benefits of parallelism for writes: We further investigate if there is any benefit from issuing concurrent large writes. Figure 3 (a) shows that there is no significant performance benefits by issuing concurrent large writes. The reason is that the write-order-based mapping assigns consecutive chunks to different channels, and so, by default, writes larger than the chunk size are distributed over multiple channels.

3.2.2 Other parallelism benefits

A chunk is distributed across multiple packages on the same channel. Thus, we can extract the benefits of package-level parallelism by issuing chunk-sized or larger reads. We observe that we get higher read bandwidth (Figure 3(b)) by issuing larger reads.

Earlier works [16, 27] have shown that if reads and writes are intermingled, they affect plane-level parallelism and there is a performance drop of up to 1.3*X* in comparison to issuing consecutive reads followed by consecutive writes. In our context, we would be issuing large writes (since small random writes are expensive) and small reads (small page reads for the lookup requests). A large 128KB write already corresponds to multiple consecutive writes (since flash access granularity is at the page level, a large write is converted into multiple sequential page writes). Thus, in our context, reads and writes are not intermingled by default and there is no performance impact.

3.3 Design Guidelines

Based on the above unique properties of flash SSDs, we identify the following guidelines in designing large hash table-based data structures:

- **Avoiding random page writes:** The system should avoid random page writes and issue few large writes.
- **Combining multiple reads:** Arrange data in such a way that the multiple lookups can be confined to a single page or a small number of pages.
- **Intelligent request reordering:** To maximally explore the underlying parallelism in flash systems, we must devise mechanisms to reorder requests so that they are distributed over channels uniformly.

4 Streaming Hash Tables: SliceHash

In this section, we discuss how, using the guidelines listed above, we can build high-performance large streaming hashtables where <key, value> pairs can be looked up, inserted and updated over time. We call our index SliceHash. We first describe the approach for combining multiple reads and dealing with random writes. Then we discuss how we add concurrency to SliceHash to exploit the intrinsic parallelism of SSDs. We end with a simple analysis of SliceHash's performance as a function of its configuration.

4.1 Basic SliceHash

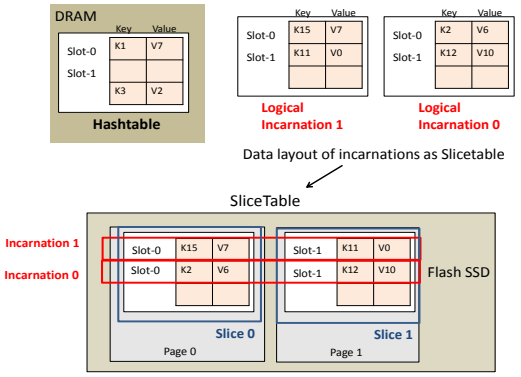


Figure 4: SliceHash structure

Our design is influenced by BufferHash, but it differs in certain key aspects. Like BufferHash, we deal with slow random writes by maintaining an in-memory hashtable where all inserts happen. Writes to the SSD

occur only when the in-memory hashtable is full. We refer to the writes of the in-memory hashtable to the SSD as *incarnations*.

Due to batched writes, a lookup would have to scan all incarnations for finding a key, and thus incur multiple flash reads (one flash read per incarnation, corresponding to reading an entry at a given slot on the incarnation). BufferHash uses in-memory Bloom filters [8] to avoid multiple spurious flash reads, but has to incur substantial memory overhead ($\sim$4 byte/entry). Furthermore, due to false positives of Bloom filters, key lookup in the worst case can require as many flash reads as the number of incarnations. SliceHash does not impoe the extra memory overhead, yet it avoids multiple flash reads and, in addition, bounds the worst-case lookup cost.

Our basic idea is to combine multiple reads; i.e., we arrange data on the SSD such that entries from all incarnations for the same slot are stored on the same page, and we can just read them together by a single flash page read. We refer to this approach of arranging data as *slicing*, and to this particular arrangement on SSD as a *slicetable*.

Figure 4 shows a hashtable, its *logical* incarnations, and how they are laid out on the SSD in the form of a slicetable. The slicetable contains slices; each slice contains a given entry from all incarnations. For example, slice-0 contains the entries in slot-0 from all incarnations together (incarnation 0: <K15,V7>, and incarnation 1: <K2,V6> in this example), slice-1 contains the entries in slot-1 from all incarnations together and so on. In order to look up a key, we just read the corresponding slice from flash and then search for the key in the entries from all incarnations. The size of a slice can be limited to a page, and thus it would require only one page read[1]. A slicetable contains as many slices as the number of slots in the in-memory hashtable. In case of a 2-function cuckoo hashtable, a key lookup would need to read the second slot if the key is not found in the first; thus, the number of page reads is bound by 2.

We analyze the memory overhead and average lookup latency in §4.5 and show that SliceHash can achieve low memory overhead (0.6 bytes/entry) along with low average latency (a page lookup).

Maintaining such a data layout requires a certain way of writing full hashtables into the SSD: We would need to place the entries corresponding to the latest incarnation in the slicetable. For example, in Figure 4, we would need to replace the entries in the red-bounded area with the entries in the in-memory hashtable. However, this additional cost is amortized over multiple insert operations, since it happens only when the in-memory hashtable becomes full.

[1]For a 16B key-value pair, one slice can contain as many as 128 incarnations

4.1.1 SliceHash Operations

We now discuss the basic SliceHash operations in detail.

Inserts. We insert a key into the in-memory hashtable. If the in-memory hashtable becomes full, we first read the corresponding slicetable from flash. We then replace the entries for the corresponding incarnation for each slot or slice with the entry of the in-memory hashtable. Then, we write back the modified slicetable to flash SSD. The in-memory hashtable is cleared, and the current incarnation count is incremented. Subsequent insertions happen in a similar way. Once all incarnations are exhausted on the flash SSD, the incarnation count is reset to zero. Thus, SliceHash supports a default FIFO eviction policy.

Updates. Our approach is similar to BufferHash [12]. If the key is in the in-memory hashtable, we just update it with the new value. If the key lies on the flash, directly updating the corresponding key-value pair on flash would cause random page writes and affect performance. Instead, we insert the new key-value pair into the in-memory hashtable.

Lookups. We first look up the key in the in-memory hashtable. If not found, we look up the corresponding slicetable and read the slice from the SSD. We scan the entries for all incarnations in the order of from the latest to the oldest incarnation. This ensures that the lookup does not return stale values.

4.1.2 Partitioning SliceHash

Based on the first few bits of keys, we partition the in-memory hashtable into multiple small in-memory hashtables, and, for each in-memory hashtable, we maintain a corresponding small-sized slicetable on flash. Thus, if an in-memory partition becomes full, we only need to update the corresponding slicetable on the SSD. In this way, we can control the size of slicetables on flash and the worst case insertion latency.

4.1.3 Leveraging available memory

An issue is that even if a key is not present on the SSD, SliceHash has to read the corresponding slice from the SSD. If additional memory is available, we can reduce such spurious lookups using in-memory Bloom filters. All the lookups are first checked in these Bloom filters. If the Bloom filters indicate that the key is present in flash, only then do we issue an SSD lookup. Further, SliceHash enables using memory opportunistically: e.g., we can maintain bloom filters for only some partitions, for example, those that are accessed frequently. This gives SliceHash the ability to adapt to memory needs, while ensuring that in the absence of such additional memory application performance targets are still met.

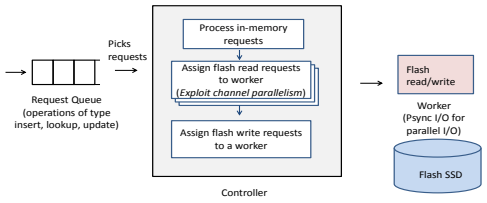


Figure 5: Adding concurrency to SliceHash

4.2 Adding concurrency to SliceHash

In order to leverage the parallelism inherent to SSDs, I/O requests should be issued in parallel. Instead of using a multithreaded programming model to achieve this, we use more lightweight method – psync I/O [27] – that can issue multiple concurrent I/O requests to the SSD and retrieve the results. Internally, psync I/O uses multiple asynchronous I/O calls, and waits till all I/Os are completed. Similar to [27], we found that the performance of psync I/O is comparable to the performance with multiple threads. The added advantage of psync I/O is that it allows using threads for other tasks, e.g., computations, that may be needed by the networked applications running atop the SSD.

We next describe how concurrency is added to SliceHash through two main components: 1) a *Controller* for request selection and 2)a *Worker* for flash reads/writes.

The controller processes requests in batches. It first process all requests that can be instantly served in memory. Then it processes lookup requests which need reading from the SSD. To leverage channel-level parallelism maximally, the controller should pick requests that go to different channels. Based on the measurements in [16], we have developed a *channel-estimator* (described in the next section) to estimate the mapping between read requests and channels. Using these estimates, we find a set of K requests (we choose K as the size of the SSD's NCQ) such that the number of requests picked for any channel is minimized. The idea is that while we want to use as much concurrency as NCQ can provide, we also want to use it carefully to optimally exploit channel parallelism.

Request-selection algorithm: The algorithm underlying request selection works as follows. We maintain a "depth" for each channel, which estimates the number of selected requests for a channel. We take multiple passes over the request queue until we have selected K requests. In each pass, we select requests that would increase the depth of any channel by at most 1. In this manner, we first find the set of read requests to be issued.

The controller then asks the worker to process these read requests in parallel using psync I/O. While the worker is waiting for flash reads to complete, the controller also determines the next batch of read requests to be issued to the worker. After the flash page reads are complete, the worker searches the entries of all incarnations on the corresponding flash page for the given key.

After processing lookups, the controller assigns SSD insert requests to the worker. Note that these occur when an in-memory hashtable is full and needs to be flushed onto the SSD. The worker processes these SSD insert requests, and accordingly reads/writes slicetables from the SSD.

Note that there may be consistency issues with reordering reads and writes. The controller handles such corner cases explicitly.

Next, we describe our channel estimator for determining channels that correspond to specific I/O requests.

4.3 Channel Estimation

We build on the technique used in [16] for reverse-engineering write-order mapping to predict the channel corresponding to a request. As discussed in §3.2, chunk writes alternate across channels, i.e., the first write goes to the first channel, the second write goes to the second channel, and so on. Knowing this write order can help us determine the channel for any chunk. One approach is to maintain an index that keeps track of the assignment of each chunk to a channel; whenever a chunk is written, we estimate its channel as i % N for the i^{th} write and update the index. We estimate the size of index to be around 160 MB (for 4KB chunk, 128 GB SSD, and assuming 4 bytes for the chunk identifier, and 1 byte for the channel in the index).

Model-based approach: We consider an approach that does not require any index management. We configure the size of the slicetable to be a multiple of $N \times$ *ChunkSize*, where N is the number of channels; this simplifies determination of the channel. Whenever a slicetable is written to the SSD, there will be N chunk writes, and the i^{th} chunk write would go to the i^{th} channel. The subsequent slicetable write would also follow the same pattern; after the N^{th} channel, the first chunk write would go to the first channel, the second chunk write would go to the second channel, and so on. In other words, once we determine the relative chunk identifier (first, or second, or N^{th}) for an offset in the slicetable, we can determine the channel. The relative chunk identifier can be determined as the offset modulo chunk size.

We estimate the accuracy of this technique based on the performance gap between the best and the worst case using our estimated channels (similar to Figure 2 (b)) since we don't know the ground truth. We observe a similar trend as in Figure 2 (b) (omitted for brevity), showing that this simple technique works as good as a much more accurate write-order-tracking technique.

4.4 Leveraging multiple SSDs

Due to its simplistic design and low resource footprint, SliceHash can easily leverage multiple SSDs attached to a single machine.

SliceHash can benefit from multiple SSDs in two ways: (a) Higher parallelism: The key space is partitioned across multiple SSDs. We maintain one controller-worker combination for each SSD. Lookup/insert requests then get distributed across multiple SSDs, and each controller handles these requests in parallel. (b) Lower memory footprint: For each in-memory hashtable, we maintain one slicetable per SSD. For lookups, we issue concurrent lookup requests to all SSDs, in effect requiring an average latency of one page lookup. For insertions, we insert into a slicetable on one SSD, and as it becomes full, we move to next SSD. Once all SSDs' slicetables get full, we return to inserting into the slicetable on the first SSD. We show in our analysis (§4.5) that this scheme can reduce the memory footprint from 0.6 bytes/entry to 0.15 bytes/entry for 4 SSDs, while maintaining the same latency and throughput. Other systems, e.g., SILT and BufferHash, do not support such scaling out and ease of tuning.

In practice, depending on the specific requirements of throughput and memory footprint, we can use a combination of above two techniques to tune the system accordingly. Thus, SliceHash allows us to leverage multiple SSDs in many different ways.

4.5 Analysis

In this section, we analyze the latency and the memory overhead of SliceHash, and compare with SILT and BufferHash. We evaluate the throughput of SliceHash in §6 and show that it is better than these systems. We also estimate the number of writes to SSD per unit time, and use that to estimate the lifetime of SSD. Our aim is to both analyze the I/O and overall costs of SliceHash but also to show the knobs it offers to easily control cost-performance trade-offs, an aspect missing from virtually all prior designs.

Table 2 summarizes the notation used.

Memory overhead per entry: We estimate the memory overhead per entry. The total number of entries in an in-memory hashtable is H/s_{eff}, where H is the size of a single hashtable and s_{eff} is the effective average space taken by a hash entry (actual size (s)/utilization (u)). The total number of entries overall in SliceHash for a given size F of flash is: $(\frac{F+M}{H}) \times \frac{H}{s_{eff}} = \frac{F+M}{s_{eff}}$

Here, M is the total memory size. Hence, the memory overhead per entry is, $\frac{M}{\#entries}$, i.e., $\frac{M}{F+M} \times s_{eff}$, or $\frac{1}{k+1} \times s_{eff}$, where k is the number of incarnations.

For $s = 16$B (key 8 bytes, value 8 bytes), $u = 80\%$, $M = 1$GB, and $F = 32$GB, the memory overhead per entry is *0.6 bytes/entry*. In contrast, SILT and BufferHash have memory overheads of *0.7 bytes/entry* and *4 bytes/entry*, respectively.

Symbol	Meaning
M	Total memory size
N	Number of SSDs
n	number of partitions
H	Size of a single hashtable ($= M/n$)
s	Size taken by a hash entry
u	Utilization of the hashtable
s_{eff}	Effective average space taken by a hash entry ($= s/u$)
k	Number of incarnations ($= F/M$)
F	Total flash size
S	Size of slicetable ($= H \times k$)
P	Size of a flash page/sector
B	Size of a flash block
r_p	Page read latency
r_b	Block read latency
w_b	Block write latency

Table 2: Notations used in cost analysis.

By using N SSDs, we can reduce the memory overhead to even lower, $\frac{1}{k \times N+1} \times s_{eff}$ using the scheme outlined in §4.4. For the above configuration with $N = 4$ SSDs, this amounts to *0.15 bytes/entry*.

Insertion cost: We estimate the average time taken for insert operations. We first calculate the time taken to read a slicetable and then write it back. This is given by: $(\frac{S}{B} \times r_b + \frac{S}{B} \times w_b)$, where S is the size of the slicetable, B is the size of a flash block, and r_b and w_b are the read and write latencies per block, respectively. This happens after H/s_{eff} entries are inserted to the hashtable; all insertions up to this point are made in memory. Hence, the average insertion cost is

$\left(\frac{S}{B} \times r_b + \frac{S}{B} \times w_b\right) \times \frac{s_{eff}}{H}$

Replacing S by $H * k$, we get $\frac{(r_b+w_b) \times s_{eff} \times k}{B}$, which is independent of the size of the hashtable.

For a typical block read latency of 0.31ms, a block write latency of 0.83ms, $s = 16$B, $M = 1$GB, $F = 32$GB, and $u = 80\%$, the average insertion cost is $\sim 5.7\mu$s, and thus still small. In contrast, BufferHash has average insertion latency of $\sim 0.2\mu$s.

Similarly, the worst case insertion cost of SliceHash is $(0.31 + 0.83) \times \frac{S}{B}$ms. By configuring S to be same size as B, we can control the worse case insertion cost to be $(0.31 + 0.83) = 1.14ms$, slightly higher than the worst case insertion cost (0.83 ms) of BufferHash.

Thus, the average and the worst case insertion latency of SliceHash are higher than those of BufferHash, but we believe that this is an acceptable tradeoff for much lower memory footprint.

Lookup cost: We consider a Cuckoo hashing based hashtable implementation with 2 hash functions. Suppose the success probability of the first lookup is p. For each lookup, corresponding slice is read. We configure H, the size of an in-memory hash table, such that size of

a slice is not more than a page. With this, the average lookup cost is $r_p + (1-p) \times r_p$ or $(2-p) \times r_p$, assuming that almost all of the lookups go to SSD and only few requests are served by in-memory hashtables. For $p = 0.9, r_p = 0.15$ ms, the average lookup cost is 0.16 ms. SILT and BufferHash, both have similar average lookup cost.

The worst case happens when we have to read both pages corresponding to the two hash functions. So, the worst case lookup latency is $2 \times r_p$. For $r_p = 0.15$ ms, this cost is 0.3 ms. In contrast, BufferHash may have very high worst case lookup latency; in the worst case, it may have to scan all incarnations. For $k = 32$, this cost would be 4.8 ms.

Frequency of SSD writes, and trade-offs: We estimate the ratio of the number of insertions to the number of block writes to the SSD; let this ratio be r_{write}. A hashtable becomes full after every H/s_{eff} inserts, after which the corresponding slicetable on flash is modified. The number of blocks occupied by a slicetable is S/B or $k \times H/B$. Thus,

$$r_{write} = \frac{H}{s_{eff}} \times \frac{B}{k \times H} = \frac{B}{k \times s_{eff}}$$

Thus, by increasing the number of incarnations k, the frequency of writes to SSD (which is inversely proportional to r_{write}) also increases. This in turn affects the overall performance. Note, however, that increasing the number of incarnations also decreases the memory overhead as shown earlier. Thus, our design imposes a trade-off of between memory overhead and performance. We investigate this dependency in more detail in §6.3 and find that our design provides a smooth trade-off allowing designers the flexibility to pick a point in the design space that fits their specific cost-performance profile.

Effect on SSD lifetime: SliceHash increases the number of writes to the SSD which may impact its overall lifetime. We now estimate the lifetime of an SSD as follows. For a given insert rate of R, the number of block writes to the SSD per second is $\frac{R}{r_{writes}}$ or the average time interval between block writes is $\frac{r_{writes}}{R}$. Say the SSD supports E erase cycles. Also let say that the wear leveling scheme for flash is perfect, then the lifetime (T) of the SSD could be approximately estimated as number of blocks, $\frac{F}{B}$ times erase cycles E, times average time interval between block writes, $\frac{r_{writes}}{R}$, i.e., $T = \frac{F \times E \times r_{writes}}{R \times B}$

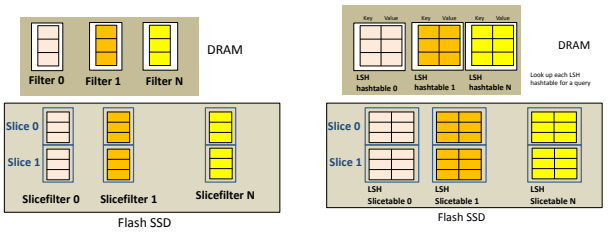
Consider a 256GB MLC flash drive that supports 10000 erase cycles [13]. We use SliceHash on this SSD with $M = 4$GB of DRAM, i.e., $k = 64$. With a 16B entry size and utilization of 80%, the ratio r_{write} would be 102.4. Even with $R = 10K$ inserts/sec (required, e.g., for a WAN optimizer connected to 500 Mbps link), the SSD would last 6.8 years. Thus, despite an increase in the writes to SSD, the lifetime of SSD would still be reasonably long.

Summary: Thus, our analysis shows that using the design guidelines, SliceHash can reduce the memory overhead to 0.6 bytes/entry and limit the lookups to 1 page read on average, without affecting the average insert performance or lifetime of SSDs significantly. A simple knob, the number of incarnations, helps control the performance-cost trade-off in a fine-grained fashion. We empirically study the performance and flexibility benefits of SliceHash in §6.

Next, we discuss how our key techniques can also be applied to other forms of hashtables.

5 Generality

In this section, we discuss how the techniques discussed in § 3, and used toward hashtables in § 4, can be easily extended to other hash-based data structures. In particular, we discuss large Bloom filters and locality sensitive hashing-based indexes.



(a) SliceBloom (b) SliceLSH

Figure 6: Extension to other hash-based systems

Bloom Filters: Bloom filters are traditionally used as in-memory data structures [8]. As some recent studies have observed [20, 15], with storage costs falling and data volumes growing into the peta- and exa-bytes, space requirements for Bloomfilters constructed over such datasets are also growing commensurately. In limited memory environments, there is a need to maintain large Bloom filters on secondary storage. We show how we can apply our techniques for supporting Bloom filters on flash storage efficiently.

Figure 6(a) shows the overview of our SliceBloom design. Similar to SliceHash, we maintain several in-memory small Bloom filters and corresponding slicefilters on flash (these are similar to slicetables in SliceHash). The in-memory Bloom filters are written to flash as incarnations. Each slot in a slicefilter contains the bits from all incarnations taken together.

In traditional Bloom filters, a key lookup requires computing multiple hash functions and reading entries corresponding to the bit positions computed by the hash functions. In our case, we first look up the corresponding in-memory Bloom filter partition and then lookup the corresponding slicefilter on the flash storage for each hash function. The number of hash functions would determine the number of page lookups, which could limit the throughput.

Since flash storage is much cheaper than DRAM, we can use more space per entry on flash – i.e., large m/n where m and n are the Bloom filter size and number of unique elements, respectively – and reduce the number of hash functions (k) while maintaining a similar overall false positive rate [1]. For example, for a target false positive rate of 0.0008, instead of using $m/n = 15$ and $k = 8$, we can use $m/n = 32$ and $k = 3$. By reducing k, we can reduce the number of page lookups and improve performance.

Our design techniques enable us to reduce the effective memory footprint per key while achieving high performance. For example, choosing $m/n = 32$, we can use a combination of 256MB DRAM and a 64GB SSD (leading to 256 incarnations per Bloom filter) to store Bloom filters, resulting in an effective memory overhead of 0.1 bits per entry and causing block writes to flash every 128 key insertions. Our evaluations in § 6.4 shows that we achieve good performance with this configuration.

LSH hashtables: Locality sensitive hashing [22] is a technique used in the multimedia community([24, 9]) for finding duplicate videos and images at large scale. These systems use multiple hashtables. For each key, the corresponding bucket in each hashtable is looked up. Then, all entries in the buckets are compared with the key to find the nearest neighbor based on a certain metric (e.g., the Hamming distance or an L_2 norm). We discuss how we can apply our set of techniques to build large LSH hashtables efficiently on flash storage.

Figure 6(b) shows the overview of SliceLSH design. Each of the LSH hashtable is designed as SliceHash; when a query comes, it goes to all SliceHash instances. We further optimize for LSH to exploit SSD-intrinsic parallelism. When we write in-memory LSH hashtable partitions to flash, we arrange them on the flash such that each LSH slicetable partition belongs to one channel and the hashtables are uniformly distributed over multiple channels (details omitted for brevity). This would ensure that multiple hashtable lookups would be uniformly distributed over multiple channels, and we would be able to maximally leverage the intrinsic parallelism of flash SSDs.

6 Evaluation

In this section, we measure the effectiveness of our design patterns as applied to the three different indexes described above. For simplicity, a majority of our evaluation focuses on SliceHash.

Our goal is to answer the following key questions:

- **Performance:** What is the lookup and insert performance of SliceHash? To what extent can SliceHash leverage the benefits of the intrinsic parallelism in flash storage? How does SliceHash perform under different mixes of read and write workloads? How does it compare with state-of-the-art in these respects?
- **Flexibility:** How much flexibility does SliceHash provide in terms of meeting different memory footprint (which we use as a proxy for cost) vs. performance trade-offs? How effectively can SliceHash can leverage the scale offered by multiple SSDs without sacrificing index or application performance?
- **Generality:** How do our design choices improve the performance of other indexes?

6.1 Implementation and Configuration

We have implemented SliceHash in C++ in roughly 3000 lines of code. The concurrency of I/O is implemented using the psync I/O library [27]. The hashtable is implemented using Cuckoo hashing [30] with 2 hash functions and 3 entries per bucket, which corresponds to 86% space utilization. The size of each hashtable partition is 128KB, and it can hold ∼7K items for 16B key-value entries.

Our code uses direct I/O to access flash SSDs. We use the simplest noop scheduler (FIFO) for leveraging the intrinsic parallelism of SSDs. Slicetables corresponding to different partitions are arranged on continuous logical block addresses on SSDs.

We evaluate SliceHash on a 128GB Crucial M4 SSD using a quad-core Intel Xeon processor. We use a batch size of 32 for concurrent lookups using psync I/O. The size of the NCQ is 32. Unless otherwise specified, the size of each slicetable is 4096 KB and the slicetable contains 32 incarnations of an in-memory hashtable partition. This amounts to using 4GB DRAM for the 128GB Crucial SSD.

6.2 SliceHash Performance

Workloads: We consider three types of workloads: *a) Random:* The keys are generated randomly, so distribution of requests among channels is also random. *b) Skewed:* The channel distribution is skewed, i.e., certain number of requests (configured by skew parameter *S*) go to the same channel, while remaining requests are evenly distributed across channels, and *c) Ordered:* The requests are uniformly distributed across channels, however their ordering is such that the first K requests go to first channel, the second K requests go to second channel and the i^{th} set of K requests go to channel $i \bmod N$ (where N is the number of channels; $N = 32$ for Crucial SSD). Essentially, if $K = 1$, even a FIFO scheme would have all 32 requests going to different channels (the best case), while if $K = 32$, it would result in flash page read requests going to the same channel (the worst case).

We first evaluate lookup and insert performance of SliceHash under the above workloads. We compare performance of SliceHash with BufferHash and SILT.

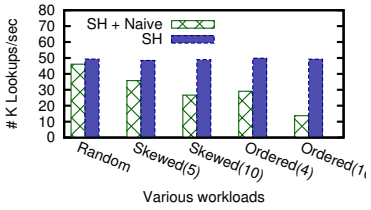
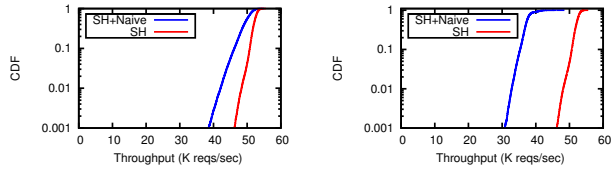


Figure 7: Average SliceHash performance for channel-awareness, compared to naive FIFO scheme



(a) Random (b) Skewed (S=5)

Figure 8: CDF of throughputs for SliceHash compared to naive FIFO, under random and skewed workloads

Lookup performance: Our particular focus is on examining how lookup performance in SliceHash benefits from intelligent request reordering. To study this, we compare SliceHash with a scheme that, like SliceHash, uses slicing but, unlike SliceHash, employs FIFO instead of channel-aware request selection. We consider lookup-only versions of the above workloads.

Figure 7 shows the performance difference between SliceHash (labeled as SH) and SliceHash without channel-awareness (labeled as SH+Naive). We see that SliceHash can consistently provide 49-51K lookups/sec by appropriately reordering requests and distributing them across channels uniformly. In contrast, being oblivious to channels, the performance of SH+Naive can drop by almost 4X (Ordered (K=16)). Even under a small skew of 5 requests (S=5), the performance drops by 30%; larger skew (S=10) deteriorates performance by almost 2X. In the case of a random workload, where keys are likely to be evenly distributed across channels, the performance difference between SH and SH+Naive is 7%.

We further investigate the distribution of instantaneous throughputs (measured using time taken to process each batch of 32 requests) for SH and SH+Naive. Figures 8 (a) and (b) shows the CDF of instantaneous throughputs for SH and SH+Naive under random and skewed (S=5) workloads respectively. The difference between 99.9% performance of SH and SH+Naive is 18% for random workload, and 36% for skewed workload. These results indicate that how channel-awareness is crucial to high performance.

SliceHash processes lookup requests in batches. With channel-aware parallelism, the average read latency for batch of 32 page read requests is 0.6 ms under various workloads. In contrast, SH+Naive has average read latency of 0.7 ms, 0.91 ms and 1.14 ms for random, skewed (S=5) and ordered (K=4) workloads, respectively.

Both, BufferHash and SILT are oblivious to channel parallelism. Hence, we can expect that their performance would be poor under certain workloads such as the ones presented above, and we find that this indeed is the case (results omitted for brevity). Thus, SliceHash provides much higher performance than BufferHash and SILT by intelligent request selection.

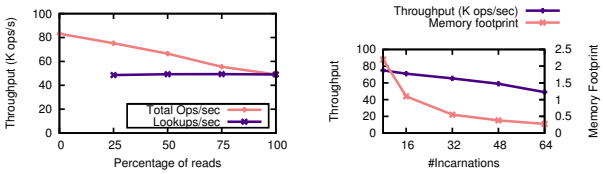
In the rest of the evaluation, we focus on the random key workload.

Insert performance: We now study insert throughput of SliceHash. We observe that for a continuous insert-only workload (random keys), SliceHash can achieve 83K inserts/sec. Most of the requests are served in memory, so the average insert latency is very small (0.012 ms). In contrast, BufferHash can achieve almost 130K inserts/sec for the same configuration (i.e., 128 KB in-memory hashtable).

Recall that each hashtable partition in SliceHash holds ∼7K items. Flushing to flash happens once the hashtable becomes full. At that time, SliceHash reads the corresponding slicetable from flash, modifies it and then writes it back to flash. The total time for this operation is 43 ms. However, this cost gets amortized over multiple insertions; further, this can be scheduled in the background, so it does not effect insert performance significantly. In comparison with SliceHash, BufferHash is a highly write-optimized data structure; because Bufferhash has to just write the buffer to flash storage when its gets full, a much higher write throughput is possible.

We believe that this is an acceptable trade-off for the low memory overhead and better/more consistent lookup performance offered by SliceHash. Moreover, SliceHash can be augmented with a small write-optimized table (using a BufferHash-like data structure) for handling bursts of writes; this table can be written back to SliceHash during a low-activity period. SILT uses a similar idea; it uses a write-optimized data structure for handling writes, which is later merged into SILT's read-optimized data structures. However, merging in SILT is far more compute-intensive (needs sorting) than writing a hashtable back to a slicetable with SliceHash, which just requires copying entries to appropriate positions.

Finally, we investigate how SliceHash performs under a continuous workload of lookups and inserts where we vary the proportion of inserts to lookups. Figure 9(a) shows the overall performance of SliceHash in this mixed workload setting; we also separately monitor the lookup performance by measuring the amount of time to process batches of 32 lookup requests each (also shown in figure). We observe that SliceHash provides 49K-83K ops/sec under the different workloads. Considering lookups, even under a 50% lookup-50% insert workload, the lookup performance of SliceHash is not



(a) Mixed Workload (b) Memory vs. Performance

Figure 9: Performance evaluation under (a) mix of lookups/inserts and (b) increase in number of incarnations

# SSDs	100% Lookups (ops/sec)	50% Lookup/50% Inserts (ops/sec)
1	51K	64K
2	101K	127K
3	148K	188K

Table 3: Performance with multiple SSDs

visibly affected.In contrast, SILT's lookup performance drops significantly under this workload (20-30%) due to its background sorting and merging operations [23].

6.3 Flexibility in SliceHash

SliceHash is highly flexible and can be tuned to match application requirements. SliceHash has a very small memory footprint (0.6 bytes/entry), and it can leverage additional memory to speed up lookups, e.g., by using bloom filters (§4.1.3). It also has a small CPU footprint – it can provide high performance using just one core. The remaining cores can be used for other application-specific compute tasks. In contrast, BufferHash has high memory footprint, and SILT imposes high CPU overhead due to continuous sorting; these aspects limit their suitability to a range of important applications.

In addition, SliceHash provides the flexibility to tune the memory footprint at the cost of performance, and, it can scale to multiple SSDs without usurping memory/CPU, as we show below.

Memory footprint vs. Performance: As discussed in §4.5, by increasing the number of incarnations, we can reduce the memory footprint of SliceHash. The side effect is that the number of block writes to flash SSDs is higher. Figure 9(b) shows this trade-off for a 50% lookup/50% insert workload. SliceHash provides a throughput between 75K-49K operations/sec with memory footprint ranging from 2 bytes/entry to 0.275 bytes/entry.

Scaling using multiple SSDs: We evaluate SliceHash on our quad-core Intel Xeon machine using upto 3 SSDs; we find that SliceHash can provide linear scaling in throughput performance; see Table 3. With 3 SSDs, SliceHash offers 150 K ops/sec for a full lookup workload, and 188K ops/sec for a 50% lookup/50% inserts workload. Because of its low CPU and memory footprint, SliceHash can easily leverage the multiple SSDs on a single physical machine to match higher data volumes and provide higher overall throughput without usurping the machine's resources. Neither SILT nor BufferHash can scale in this fashion the former due to high CPU overhead and the latter due to high memory overhead.

6.4 Generality: SliceBloom and SliceLSH

We now show how our general design patterns improve the performance of other indexes.

We evaluate SliceBloom on the 128GB Crucial SSD using 512 MB DRAM. We use $m/n = 32$ and $k = 3$ hash functions with a memory overhead of 0.1 bits/entry. Under a continuous 50% insert-50% lookup workload, our system can perform 12-15K ops/sec. With naive parallelism, the system performance can drop to 4-5K ops/sec. In contrast, BloomFlash [20] achieves similar performance for 50% insert-50% lookup workload, but on a high-end fusionIO SSD (100,000 4KB I/Os per sec) that costs 30X more ($6K vs. $200). Furthermore, on a low-end Samsung drive, BloomFlash only provides 4-5K lookups/sec.

We also evaluate SliceLSH on the Crucial SSD. We use 10 hashtables, where each hashtable uses 256MB in memory, and the corresponding slicetable occupies 8GB on flash. SliceLSH can perform 4.9K lookups/sec, as it has to look up each hashtable. By design, SliceLSH can intrinsically exploit channel parallelism. Hence, our system consistently offers similar performance under various workload patterns (results omitted for brevity).

7 Conclusion

A key impediment in the design of emerging high-performance data-intensive systems is the design of large hash-based indexes that offer good throughput and latency under specific workloads and at specific cost points. Prior works have explored point solutions using flash-based SSDs that are each suited to a narrow setting and crucially lack flexibility and generalizability.

In this paper, we develop a set of general techniques for building large, efficient and flexible hash-based systems by carefully leveraging the unique properties of flash SSDs. Using these techniques, we first build a large streaming hash table, called SliceHash, that provides higher performance, while imposing low computation overhead and low memory overhead, compared to the state-of-the-art. Developers can easily tune SliceHash to meet performance goals under tight memory constraints and satisfy the diverse requirements of various data-intensive applications. We illustrate the generality of our ideas by showing that they can be applied to building other efficient and flexible hash-based indexes.

While not conclusive, our work shows the promise of adopting general patterns centered around the primitives we advocate to design SSD-based indexes.

References

[1] Bloom filter maths. http://pages.cs.wisc.edu/ cao/papers/summary-cache/node8.html.

[2] BlueCoat video caching appliance. http://www.bluecoat.com/company/press-releases/blue-coat-introduces-carrier-caching-appliance-large-scale-bandwidth-savings.

[3] Disk Backup and deduplication with DataDomain. http://www.datadomain.com.

[4] Memcached: A distributed memory object aching system. http://www.danga.com/memcached.

[5] Peribit Networks (Acquired by Juniper in 2005): WAN Optimization Solution. http://www.juniper.net/.

[6] Riverbed Networks: WAN Optimization. http://www.riverbed.com/solutions/optimize/.

[7] A. Badam, K. Park, V. S. Pai and L. Peterson. HashCache: Cache Storage for the Next Billion. In *NSDI 2009*.

[8] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics, 2005*, 1(4):485–509.

[9] A. Torralba, R. Fergus, and Y. Weiss. Small code and large image databases for recognition. In *Proc. CVPR*, 2008.

[10] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, pages 57–70, 2008.

[11] A. Anand, A. Akella, V. Sekar, and S. Seshan. A case for information-bound referencing. In *HotNets*, page 4, 2010.

[12] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella and Suman Nath. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. In *NSDI 2010*.

[13] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi. Differential raid: rethinking raid for ssd reliability. In *EuroSys*, pages 15–26, 2010.

[14] M. Bjørling, L. L. Folgoc, A. Mseddi, P. Bonnet, L. Bouganim, and B. T. Jónsson. Performing sound flash device measurements: some lessons from uflip. In *SIGMOD Conference*, pages 1219–1222, 2010.

[15] M. Canim, G. A. Mihaila, B. Bhattacharjee, C. A. Lang, and K. A. Ross. Buffered bloom filters on solid state storage. In *ADMS*, 2010.

[16] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA*, pages 266–277, 2011.

[17] David Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP 2009*.

[18] B. K. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. *PVLDB*, 3(2):1414–1425, 2010.

[19] B. K. Debnath, S. Sengupta, and J. Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *SIGMOD Conference*, pages 25–36, 2011.

[20] B. K. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H.-C. Du. Bloomflash: Bloom filter on flash-based storage. In *ICDCS*, pages 635–644, 2011.

[21] Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos and Walid A. Najjar. Microhash: an efficient index structure for fash-based sensor devices. In *USENIX FAST 2005*.

[22] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. VLDB*, 1999.

[23] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. Silt: a memory-efficient, high-performance key-value store. In *SOSP*, pages 1–13, 2011.

[24] Q. Lv, M. Charikar, and K. Li. Image Similarity Search with Compact Data Structures. In *Proc. CIKM*, 2004.

[25] S.-Y. Park, E. Seo, J.-Y. Shin, S. Maeng, and J. Lee. Exploiting internal parallelism of flash-based ssds. *Computer Architecture Letters*, 9(1), 2010.

[26] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST*, pages 89–101, 2002.

[27] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *PVLDB*, 5, 2011.

[28] Suman Nath and Aman Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *ACM/IEEE IPSN 2007*.

[29] Suman Nath and Phillp B. Gibbons. Online Maintenance of Very Large Random Samples on Flash Storage . In *VLDB 2008*.

[30] Ulfar Erlingsson, Mark Manasse and Frank McSherry. A cool and practical alternative to traditional hash tables. In *Proceedings of the 7th Workshop on Distributed Data and Structures (WDAS 2006)*.

[31] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, 2008.