Computer Sciences Department

Building Cheap and Large CAMs Using BufferHash

Ashok Anand
Steven Kappes
Aditya Akella
Suman Nath

Technical Report #1651

February 2009

Building Cheap and Large CAMs Using BufferHash

Ashok Anand*, Steven Kappes*, Aditya Akella* and Suman Nath†
*UW-Madison, †Microsoft Research

ABSTRACT

We show how to build cheap and large CAMs, or *CLAMs*, using flash memory. These CLAMs are targeted at an emerging class of networking applications that require massive indexes running into a hundred GB or more, with items been inserted, updated and looked up at a rapid rate. Examples of such applications include WAN optimizers, data de-duplication, network monitoring, and traffic analyzers. For such applications, using DRAM-based indexes is quite expensive, while on-disk approaches are too slow. In contrast, our flash memory-based CLAMs cost nearly the same as using existing on-disk approaches but offer orders of magnitude better performance.

While flash memory inherently offers efficient random reads required for fast lookups, it does not support efficient small random writes required for inserts and updates. To address this, we design an efficient data-structure called BufferHash that significantly lowers the amortized cost of all write operations. Our design of BufferHash also incorporates efficient and flexible eviction policies.

We build CLAMs using BufferHash on SSDs and disks. We find that the SSD-based CLAMs can offer average insert and lookup latencies of 0.02ms and 0.06ms (for 40% lookup success rate), respectively. We show that using such a CLAM in a WAN optimization application can offer 3X better throughput improvement than current designs.

1. INTRODUCTION

In recent years, a number of networked systems and architectures have emerged that use indexes as large as tens to a few hundred gigabytes in size. These indexes are hash tables of random "fingerprints" constructed over large streaming data sets. Examples include WAN optimization [2, 8, 7, 1] (index is 16-32GB), data deduplication and backup systems [3, 38] (index is >20GB [38]), monitoring and traffic analysis systems for fine-grained traffic engineering and anomaly detection [29] (index is >100GB) and data-oriented networks [36, 32, 27]. In all these applications the index is looked-up and updated frequently. For instance, a WAN optimizer connected to a 1Gbps link that is operating under medium-to-heavy load may require roughly 10,000 index lookups and insertions each per second.

Conventionally, CAMs (i.e., content-addressable memories) measuring a few hundred KB have been used to provide low latency support for lookups, insertions and updates in high performance networking scenarios such as forwarding. In this paper, we show how to build CAMs that are very large and inexpensive, and are fast enough to meet the latency demands of the above applications. We call these *CLAMs*, for cheap and large CAMs.

Today, there are two possible choices for providing CAM-like functionalities at large scales in the aforementioned systems. The first is to maintain a large index in DRAM. However, at the sizes we are targeting (a few hundred GB), the memory sub-system cost could place the system in question at a very unattractive price point. Furthermore, memory systems with such large amounts of DRAM also consume a lot of power, further increasing operating cost.

A popular alternative is to maintain database indexes, such as Berkeley-DB [6], designed for magnetic disks. Such indexes can be very large, but the throughputs of the above operations are roughly two orders of magnitude lower than what is needed optimally. Unfortunately, this can severely undermine the effectiveness of the systems in practice. For example, bandwidth savings from using Berkeley-DB in WAN optimization could be less than half the best possible savings when the WAN optimizer is under heavy load. Note that fast stream databases such as GigaScope [19] and others [10, 11] and wire-speed data collection systems such as Endace DAG [21] and CoMo [24] are not suitable as CLAMs as they do not include any archiving and indexing mechanism.

In this paper we investigate a new approach that offers an attractive middleground in the cost-performance tradeoff posed by the above two choices. In terms of cost-per-GB, our desired CLAM should be significantly, e.g., an order of magnitude, cheaper than DRAM (but can be slightly more expensive than disks). In terms of performance, the CLAM should be at least 2 orders of magnitude faster than conventional disk-based approaches (but can be 2 or more orders of magnitude slower than DRAM-only solutions). Moreover, the CLAM can be very large, e.g., up to a few hundred GB, which may be infeasible in DRAM-only systems. Such CLAMs, in addition to significantly boosting the performance of the above applications, may suggest alternate network architectures that are cost-effective and efficient. For instance, high-throughput centralized mechanisms for resolving content-based names in data-oriented architectures [36, 27] can be designed at low cost using CLAMs.

Our CLAM design uses a commodity two-level storage/memory hierarchy consisting of a small amount of DRAM and a much larger amount of *flash-based storage* (i.e. flash memory chips or solid state disks (SSDs)). Flash is rapidly supplanting magnetic disks in many systems because of many superior

aspects such as higher I/O per second per dollar, greater reliability, and better power efficiency. Because of popularity, newer generation of SSDs are getting bigger and cheaper. Configuring a CLAM with 4GB of memory and 32GB of flash, for instance, costs as little as $300 using current hardware, and the cost is likely to fall as flash becomes cheaper [18].

Flash is attractive for designing CLAMs because it supports fast random reads required for lookup operations. However, some operations for streaming applications, such as random inserts and updates are very expensive on flash as they may require erasing entire flash blocks (details in Section 4). Moreover, the size of an I/O operation on flash is several orders of magnitude bigger than that of an individual CLAM operation. Because of these unique properties of flash, unless designed carefully, a flash-based CLAM could perform poorer than a disk-based index! (See Section 2)

To support high performance in light of the above properties of flash, we introduce a new data structure called BufferHash. A key idea behind BufferHash is that instead of performing individual random insertions directly on flash, we can leverage faster memory (DRAM) to buffer multiple operations and write them to flash all at once in a *batch*. This shares the cost of a flash I/O operation across multiple CLAM operations, resulting in a better amortized cost per operation. Like a log-structured file system [33], batches are written to flash sequentially, the most efficient write pattern for flash. The idea of batching operations to amortize I/O costs has been used before in the context of many systems, including tree-based index [12], and group commit in DBMS and file systems [23]. However using it for a flash-based hash table is novel, and it poses several challenges.

Fast lookup. A CLAM must support very fast lookup of a given key. However, with batched write, a given key may reside in any prior batch, depending on time it was written out to flash. A naive lookup algorithm would examine all batches for the key, which would incur high and potentially unacceptable I/O costs. To reduce the overhead of examining on-flash batches, BufferHash (i) partitions the key space to limit the lookup in one partition, instead of the entire flash, and (ii) uses in-memory Bloom filters (like Hyperion [20]) to efficiently determine a small set of batches that may contain the key. To further reduce the cost of Bloom filter lookup and update, BufferHash organizes the Bloom filters with a technique called *windowed bit-slicing*.

Flash properties. The unique I/O properties of flash demand careful choice of various CLAM design parameters such as the amount of DRAM to use, the sizes of batches and Bloom filters, etc. Suboptimal choice of these parameters may result in poor performance of a CLAM on flash. An important contribution of this paper is to analytically show the impact of these parameters on latencies of different CLAM operations. Our analysis also provides optimal values of various design parameters.

Limited flash. In many of the applications identified earlier in this section the index holds streaming data (e.g. WAN optimization and network monitoring). Thus, under steady state, insertion of new keys into a CLAM may require creating space by evicting old keys. Without effective support, eviction could be time consuming and impact the performance of other operations such as insertions and lookups. BufferHash uses an age-based internal organization of index information that naturally supports bulk evictions of old keys in an I/O-efficient manner. BufferHash also supports other flexible eviction policies (e.g. priority-based removal) to match different application needs, albeit at additional performance cost. Existing indexing systems for archived streaming data do not address such eviction.

Supporting updates. Since flash does not support update or deletion efficiently, modifying existing $(key, value)$ mappings *in situ* in a CLAM is expensive. To support good update latencies, we adopt a *lazy update* approach where *all* value mappings, including deleted or updated ones, are temporarily left on flash and later deleted in batch during eviction. However, this does not affect the semantics of the CLAM operations; i.e., during lookup, BufferHash returns the most recent value. Such lazy updates have been previously used in other contexts, such as in tree-based index [12] and lazy garbage collection in log structured file systems [33].

We prototype CLAMs using BufferHash on SSDs from two different vendors. While designed for flash, we show that BufferHash could enable CLAM-like functionality on disks, but with poor performance than flash-based CLAMs. Using extensive analysis based on a variety of workloads, we study the latencies supported in each case and compare the CLAMs against popular approaches such as using Berkeley-DB on disk. Finally, we study the benefits of using CLAMs in a WAN optimization application using a variety of synthetic and real traffic traces. Our key observations are:

(1) Our Intel SSD-based CLAM offers average insert latency of 0.02ms compared to 7ms from using Berkeley-DB on disk. For a workload with 40% hit rate, the average lookup latency is 0.06ms for our CLAM, but 7ms for Berkeley-DB.

(2) The other two CLAMs are cheaper than the Intel-based CLAM. They are each an order of magnitude worse in terms of latencies but still much better than Berkeley-DB on disk. They could replace disk-based Berkeley-DB index in low-end equipment such as WAN optimizers for slow links.

(3) Using an Intel-SSD based CLAM, the throughput benefits from a WAN optimizer under heavy load can be improved 3X compared to using Berkeley-DB on disk.

2. MOTIVATING APPLICATIONS

Our goal is to design CLAMs that support random lookup and insert latencies of under 0.1ms and can hold hash tables that are several tens of GB in size. We now describe three networking and systems applications that employ indexes with these requirements. We also describe a data oriented architecture that could leverage such CLAMs.

WAN Optimization. WAN optimizers [2, 1, 8, 7] leverage the fact that network transfers carry redundant information and that it may be faster to look for redundancies locally and

transmit compressed data than to transmit full data. These products are widely deployed by enterprises and data centers to lower their WAN usage costs. A WAN optimizer computes fingerprints of each arriving data object. These are looked up in an index constructed over all prior content seen. A matching fingerprint indicates a certain degree of similarity between the current object and a prior object. Overlapping content is removed, and the "compressed object" (augmented with some meta-data) is transmitted to the destination, where it gets reconstructed. Fingerprints for the original object are inserted into the index to aid in future matches.

The caches holding prior content are $\sim$10TB in size [9]. The fingerprint are 16-32B hashes computed over 10KB data chunks; thus the index could be 16-32GB. Consider a WAN optimizer connected to a 1Gbps link operating at 100% utilization. Assuming an average object size of 100KB, about 10,000 fingerprints are created per second. Fingerprint lookup latencies could add to the total transfer time of each object; hence it should be low enough so that tangible compression benefits are obtained. Depending on the implementation, three scenarios may arise during insertion under heavy load situations: (1) lookups for upcoming objects are held-up until inserts for prior objects complete (2) instead of waiting, upcoming objects are transmitted without fingerprinting and look up (3) insertions are aborted mid-way and upcoming objects looked up against an "incomplete index". Fast support for insertions can improve all three situations and help identify more content redundancy in situations of heavy load. We studied the suitability of an on-disk index like Berkeley-DB for this application use a real trace (§8). We found that due to poor support for insertions and lookups of streams random keys, the throughput improvements are less than a third of the optimal assuming a 100Mbps link (this could be much worse for higher speed links). In contrast, using an Intel-based CLAM offers optimal benefits.

Data deduplication and Backup. Data de-duplication [3] is the process of removing redundant content from enterprise data leaving only one copy of the data to be stored for archival. As users generate greater amounts of data, the demand for these products has risen sharply. Prior work suggests that the data sets could be roughly 8-10TB and employ 20GB indexes [3, 38]. A time-consuming activity in deduplication is merging data sets and the corresponding indexes. Reducing the time taken in this operation is crucial to ensuring high availability of the deduplication service. To merge a smaller index into a larger one, fingerprints from the latter dataset needs to be looked up, and the larger index updated with any new information. Merging a million fingerprints into a larger index using Berkeley-DB could take as long as 2 hours. In contrast, using CLAMs based on BufferHash, the merge finishes in under 2 minutes.

Without going into the details, we note that a similar set of challenges arise in online backup services [4] which allow users to constantly, and in an online fashion, update a central repository with "diffs" of the files they are editing, and to retrieve changes from any remote location on demand.

Monitoring. An important task in traffic engineering is to monitor the performance experienced by packets at they traverse a set of links for traffic engineering and debugging performance problems. One approach to do this is obtaining header traces, say for a customer's traffic, from multiple network locations (e.g. entry/exit links into a PoP) over several minutes to an hour. Packet records from one location are loaded into a large hash table. Records from adjacent links are then looked up. As lookups are performed, record could get updated with performance information (e.g. PoP-by-PoP delay). Assuming a customer's traffic makes up 1% of total traffic, capturing this on a 50% utilized OC768 link over a one hour interval could create a 100GB index. Sprint's IPMon employed such an approach [29] but relied on an on-disk index. This needs fast random inserts, lookups and updates to improve the speed of responding to anomalies.

A Data Oriented Network Architecture. Network users today are interested in content, but not who is serving it. To simplify content access in the modern age, recent proposals argue for a separate and robust resolution infrastructure for content names [27, 36, 32]. The names are hashes computed over chunks of content inside data objects. The resolution infrastructure provides a mapping of the locations of data chunks. As new sources of data arise or as old sources leave the network, the resolution infrastructure should be updated accordingly. To support scalability, the architectures have conventionally relied on a distributed resolution mechanism based on DHTs [27, 36, 32]. However, in some deployment scenarios (e.g. a large corporation), the resolution may have to be provided by a trusted central entity. To ensure high availability and throughput for a large user-base, the centralized deployment should support fast inserts and efficient lookups of the mappings. CLAMs can support such an architecture effectively.

3. RELATED WORK

Hash table is used as one of the fundamental modules in several network processing algorithms and applications and therefore building fast hash tables has been an active research area. Broder et al. [16] demonstrated a fast lookup technique based on multiple hashing, which has recently been optimized for hardware implementation [26]. Song et al. [34] proposed a novel and fast hash table data structure based on multiple-access Bloom filter. The common goals of these work is to build hash tables on hardware or on extremely fast but small, byte-addressable memory and to optimize for lookups only (i.e., optimizing insertions is typically not the goal). In contrast, a CLAM is much larger and cheaper, but slower, than these solutions. Moreover, unlike these existing solutions, CLAM is designed for streaming applications that require, in addition to looking up items, inserting items to a hash table at a high rate and discarding old items if needed.

Like BufferHash, Hyperion [20] enables archival, indexing, and on-line retrieval of high-volume data streams. Although it has a more general set of functionalities than Buffer-

Hash (e.g., it can support rank and range queries), BufferHash is more optimized for CAM-like functionalities in a streaming scenario. For example, to lookup a key, Hyperion may need to examine prohibitively high volume of data (it does not use partition, and individual batch of data is not organized as hash tables), resulting in a high latency. Second, it does not consider using flash storage, and hence does not aim to optimize design parameters for flash. Third, it does not focus on efficient eviction of indexed data. Finally, it does not support updating or deleting already indexed data.

The GigaScope [19] network monitoring system is able to process full-speed network monitoring streams and provides a SQL-based query language. Queries can, however, be made on incoming data streams only; there is no mechanism in GigaScope to index and query past data. The same is true for many other existing datastream systems [10, 11]. StreamBase [35], another general purpose streaming database, supports archiving data and processing query over past data; but the data is archived in conventional hash or B-Tree-indexed tables, which are slow and are suitable only for offline queries. Endace DAG [21] and CoMo [24] are designed for wire-speed data collection and archiving; but they provide no mechanism to index and query the archived data. Existing DBMSs can support CAM-like functionalities. However, they are designed neither for high update and lookup rates (see [11]) nor for flash storage (see [31]).

Recent research has shown how to design efficient data structures on flash memory. Examples include MicroHash [37], a hash table and FlashDB [31], a B-Tree index, both designed for flash memory. Unlike BufferHash, these data structures are designed for memory-constrained embedded devices where the design goal is to optimize energy usage and minimize memory footprint—latency is typically not a design goal; e.g., in MicroHash, a lookup operation may need to follow multiple pointers to locate the desired key in a chain of flash blocks.

4. FLASH MEDIA AND HASH TABLES

Flash storage media comes in two different flavors: raw flash chips and portable flash packages.

Flash Chips. The most common type of flash chip used for storage is NAND flash. Its high storage capacity (currently up to 32GB in a single chip) is suitable for storing large amounts of data. The key properties of NAND flash that directly influence storage design are related to the method in which the media can be read or written, and are discussed in [28, 31]. In summary, all read and write operations happen at page granularity (or for some chips down to $\frac{1}{8}$th of a page granularity), where a page is typically 512–2048 bytes. Pages are organized into blocks, typically of 32 or 64 pages. A page can be written only after erasing the entire block to which the page belongs. However, once a block is erased, all the pages in the block can be written once with no further erasing. Thus, for an in-place update, before the erase and write can proceed, any useful data residing in other pages in the same block must be copied to a new block; this *internal copying* incurs a considerable overhead. Because there is no mechanical latency involved, random read/write is as fast as sequential read/write (assuming the writes are for erased pages).

Device	Operation	Latency (ms)	Linear model (ms)
Flash chip (FujiFilm XD-card flash chip, 2GB)	Read	0.24/page	$0.15 + 0.05x$
	Write	0.28/page	$0.15 + 0.07x$
	Erase	3.31/block	$0.5 + 0.022x$
Intel SSD (Model: X18-M) 80 GB	Seq. Read	0.16	$0.036 + 0.004x$
	Seq. Write	0.49	$0.095 + 0.012x$
	Rnd. Read	0.31	$0.143 + 0.005x$
	Rnd. Write	0.83	$0.284 + 0.019x$
Samsung SSD (MCBQE32G5MPP) 32 GB	Seq. Read	0.5	$0.114 + 0.01x$
	Seq. Write	0.6	$0.086 + 0.014x$
	Rnd. Read	0.5	$0.117 + 0.01x$
	Rnd.Write	18	$12.772 + 0.13x$
MTron SSD (SATA7035-016) 16GB	Seq. Read	0.4	$0.058 + 0.012x$
	Seq. Write	0.4	$0.035 + 0.012x$
	Rnd. Read	0.5	$0.07 + 0.012x$
	Rnd. Write	9	$8.79 + 0.006x$

Table 1: I/O latency for different flash devices. The I/O size for SSDs in the Latency column is 32 KB. The symbol x in the models represents the I/O size in KiloBytes.

Portable flash packages. Portable flash packages such as solid state disks (SSDs), compact flash (CF) cards, secure digital (SD) cards, and USB sticks provide a disk-like ATA bus interface on top of flash chips. Typically, the unit of I/O operations is a sector of 512 bytes. The disk-like interface is provided through a Flash Translation Layer (FTL) [25], which is implemented within the micro-controller of the device (or in software, such as Windows Mobile). FTL emulates disk-like in-place update for a (logical) address by writing the new data to a different physical location, maintaining a mapping between each logical address and its current physical address, and marking the old data as invalid for later garbage collection. Thus, although FTL enables disk-based applications to use flash without any modification, it needs to internally deal with flash characteristics (e.g., erasing an entire block before writing to a page). Many recent studies have shown that FTL-equipped flash devices, although a great convenience, suffer many performance problems (in particular for random writes and in-place updates [13, 15]).

Performance of flash devices. Table 1 shows I/O costs of several flash devices. The flash chip and Intel SSD costs are based on our experiments with the uFlip Benchmark [14]. The Samsung and MTron costs are reported by the authors of the uFlip benchmark in [15] and in [14]. As shown in Table 1, also mentioned in previous work [15, 28, 31], different I/O costs on flash can be modeled well with linear equations.

Because of unique characteristics of flash media, applications designed for flash should follow a few key well-known design principles. First, applications should avoid random writes, as they are significantly more expensive than other I/Os, as shown in Table 1. Second, applications should avoid in-place updates and sub-block deletions. As shown in [30], such operations are over two orders of magnitude slower than

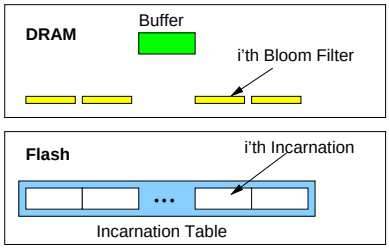


Figure 1: A Super Table

out-of-place updates and block deletions on flash devices (with or without an FTL), because they require internal copying. Third, since reads and writes happen at the granularity of a flash page (or an SSD sector), an I/O of size smaller than a page costs at least as much as a full-page I/O. Thus, applications should avoid small I/Os if possible. Finally, the high fixed initialization cost of an I/O (the component a in Table 1) can be amortized with a large I/O size. Thus, applications should batch I/Os whenever possible. In designing flash-based CLAMs using BufferHash, we follow these design principles.

A conventional hash table on flash. Before going into the details of our BufferHash design, it might be useful to see why a conventional hashtable on flash is likely to suffer from poor performance. Successive keys inserted into a hashtable are likely to hash to random locations in the hashtable; therefore, values written to those hashed locations will result in random writes, violating the first design principle above. Updates and deletions are immediately applied to a conventional hashtable, resulting in in-place updates and sub-block deletions (since each hashed value is typically much smaller than a flash block), and violation of the second principle above. Since each hashed value is much smaller than a flash page (or an SSD sector), inserting a single key in an in-flash hashtable violates the third and the fourth principles above. Violation of these design principles results in a poor performance of a conventional hashtable on flash, as we demonstrate in §7.

5. THE BufferHash DATA STRUCTURE

The BufferHash is a flash-friendly data structure that supports hashtable-like operations on $(key, value)$ pairs. The key idea underlying BufferHash is that instead of performing individual insertions/deletions one at a time to the hash table on flash, we can perform multiple operations all at once. This way, the cost of a flash I/O operation can be shared among multiple insertions, resulting in a better amortized cost for each operation (similar to buffer trees [12] and group commits in DBMS and file systems [23]). For simplicity, we consider only insertion and lookup operations for now; we will discuss updates and deletions later. To allow multiple insertions to be performed all at once, a BufferHash operates in a lazy batched manner: it accumulates insertions in an in-memory buffer, without actually performing the insertions on flash. When the buffer fills up, all inserted items are pushed in a batch to in-flash hash tables. For I/O efficiency, items pushed from buffer to flash are sequentially written as a new hash table, instead of performing expensive update to existing in-flash hash tables. During lookup, a set of Bloom filters is used to determine which in-flash hash tables may contain the desired key, and only those tables are retrieved from flash. At a high level, the efficiency of this organization comes from batch I/O and sequential writes during insertions. Successful lookup operations still need random reads, however, random reads are almost as efficient as sequential reads in flash.

5.1 A Super Table

A BufferHash consists of multiple *super tables*. In this section we describe the structure of a super table; the overall structure of BufferHash will be described in the next section. Each super table has three main components; a buffer, an incarnation table, and a set of Bloom filters. These components are organized in two levels of hierarchy, as shown in Figure 1. Components in the higher level are maintained in fast memory such as DRAM, while that in the lower level are maintained in large (but potentially slow) memory, such as flash.

Buffer. This is an in-memory hash table where all newly inserted hash values are stored. The hash table can be built using existing fast algorithms such as multiple-choice hashing [16, 26]. A buffer has a fixed capacity of maximum number of items, determined by its size and the desired upper bound of hash collisions. When the number of items in the buffer reaches its capacity, the entire buffer is flushed to flash, after which the buffer is re-initialized for inserting new keys. The buffers flushed to flash are called *incarnations*.

Incarnation Table. This is an in-flash table that contains old and flushed incarnations of the in-memory buffer. The table contains k incarnations, where k denotes the ratio of the size of the incarnation table and the buffer. The table is organized as a circular list, where a new incarnation is sequentially written at the list-head. To make space for a new incarnation, the oldest incarnation, at the tail of the circular list, is evicted from the table. Depending on how an application configures a BufferHash, some items in an evicted incarnation may need to be retained and are re-inserted into the buffer (details in §5.1.2).

Bloom Filters. Since the incarnation table contains a sequence of incarnations, the value for a given hash key may reside in any of the incarnations, depending on its insertion time. A naive lookup algorithm for an item would examine all incarnations, which would require reading all incarnations from flash. To avoid this excessive I/O cost, a super table maintains a set of in-memory Bloom filters [17], one per incarnation. The Bloom filter for an incarnation is a compact signature built on the hash keys in that incarnation. To search for a particular hash key, we first test the Bloom filters for all incarnations; if any Bloom filter matches, then the corresponding incarnation is retrieved from flash and looked up for the desired key. Bloom filters do not produce any false negative, and hence no hash keys stored in any incarnation

will be missed. However, Bloom filters may have some probability of a false positive, where it can indicate a match when there is none. This may result in unnecessary flash I/O. Bloom filter lookup poses a trade-off between the filter size and I/O overhead due to false positives. We examine the tradeoff in §6.4.

The Bloom filters are maintained as follows. When a buffer is initialized after a flush, a Bloom filter is created for it. When items are inserted into the buffer, the Bloom filter is updated with the corresponding key. When the buffer is flushed as an incarnation, the Bloom filter is saved in memory as the Bloom filter for that incarnation. Finally, when an incarnation is evicted, it's Bloom filter is discarded from memory.

5.1.1 *Super Table Operations*

A super table supports all standard hash table operations.

Insert. To insert a $(key, value)$ pair, the value is inserted in the buffer (which is a hash table). If the buffer does not have space to accommodate the key, the buffer is flushed and written as a new incarnation in the incarnation table. The incarnation table may need to evict an old incarnation to make space for this new incarnation.

Lookup. To lookup a key, it is first looked up in the buffer. If it is found, the corresponding value is returned. Otherwise, in-flash incarnations are examined in the order of their age until the key is found. To examine an incarnation, first its Bloom filter is checked to see if the incarnation might include the key. If the Bloom filter matches, the incarnation is read from flash, and checked if it really contains the key. Note that since each incarnation is in fact a hash table, to lookup a key in an incarnation, only the relevant part of the incarnation (e.g., a flash page) can be read directly.

Update/Delete. As mentioned before, flash does not support small updates/deletions efficiently; hence, we support them in a lazy manner. Suppose a super table contains an item (k, v), and later, the item needs to be updated with the item (k, v'). In a traditional hash table, the item (k, v) is immediately replaced with (k, v'). If (k, v) is still in the buffer when (k, v') is inserted, we do the same. However, if (k, v) has already been written to flash, replacing (k, v) will be expensive. Hence, we simply insert (k, v') without doing anything to (k, v). Since the incarnations are examined in order of their age during lookup, if the same key is inserted with multiple updated values, the latest value (in this example, v') is returned by a lookup. Similarly, for deleting a key k, a super table does not delete the corresponding item unless it is still in the buffer; rather the deleted key is kept in a separate list (or, a small in-memory hash table), which is consulted before lookup—if the key is in the delete list, it is assumed to be deleted even though its present in some incarnation. Lazy update wastes space on flash, as outdated items are left on flash; the space is reclaimed during incarnation eviction.

5.1.2 *Incarnation Eviction*

A BufferHash with limited flash memory may require to evict old in-flash items to make space for new items. For example, in a streaming application, many more items may be inserted into a BufferHash than it can store in its limited flash. In such case, a BufferHash must select items to discard. However, the choice of particular items to discard depends on the policy set by the application.

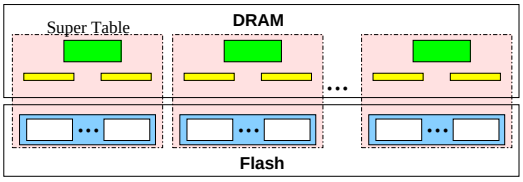


Figure 2: A BufferHash with multiple super tables

For I/O efficiency, BufferHash evicts items in granularity of an incarnation. Since each incarnation is an independent hash table, discarding a part of it may require expensive reorganization of the table and expensive I/O to write it back to flash. Since items with similar ages (i.e., items that are flushed together from the buffer) are clustered in the same incarnation, BufferHash naturally supports discarding items based on their ages. Thus, if an application uses a policy of discarding oldest items and indexing recent items only, a BufferHash can easily support such a policy by a *full discard* mechanism by entirely discarding the oldest incarnation, deallocating (and erasing) its flash blocks, and later allocating them for future incarnations.

Supporting discard policies that are not based on age are, however, a bit expensive. Such policies can be desirable when, for example, an application wants to discard lower priority data, irrespective of their ages. Such a policy is also useful when the workload contains a lot of update or delete operations—in such a case, in-flash incarnations may contain many out-of-date items that have already been (logically) deleted of updated, but haven't been physically removed from flash yet. During eviction of the oldest incarnation, the BufferHash should discard only the items that have already been deleted or updated, and retain the other items.

BufferHash supports a *partial discard* mechanism to support the above policies. This mechanism scans through all the items in an incarnation to be discarded, selects the items to be retained, and re-inserts them (into the buffer). An application can configure BufferHash with different policies to determine if an item should be discarded during incarnation eviction. For example, in a priority-based policy, an item is discarded if its priority is less than a threshold (the threshold can change over time, as in [30]). For a workload with many updates and deletes, the BufferHash discards an item if it has been deleted or updated. The former can be efficiently determined by examining the in-memory delete list, while the latter can be determined by checking the in-memory Bloom filters. Note that partial discard is expensive as it requires processing all items in the victim incarnation; moreover, since some items are re-inserted into the buffer, buffers fill up more frequently, resulting in more frequent flush operations.

5.2 Partitioned Super Tables

A super table, although simple, has a drawback. Since only a single buffer is maintained in a super table, it can be very large (e.g., as permitted by the available DRAM). Since the entire buffer is flushed at once, the flushing operation can take a long time. Since flash I/Os are blocking operations, lookup operations that go to flash during this long flushing period will block (insertions can still happen as they go to in-memory buffer). Moreover, an entire incarnation from the incarnation table is evicted at a time, increasing the cost of eviction with partial discard.

BufferHash avoids this problem by partitioning the hash key space and maintaining one super table for each partition (Figure 2). More specifically, suppose each hash key has $k = k_1 + k_2$ bits; then, a BufferHash maintains 2^{k_1} super tables. The first k_1 bits of a hash key represents the index of the super table containing the hash key, while the last k_2 bits are used as the hash key within the particular super table.

Partitioning enables using small buffers in super tables, thus avoiding the problems caused by a large buffer. However, we show in §6.4 that too many partitions (i.e., very small buffers) can also adversely affect performance. We show how to choose the number of partitions for good performance in practice. For example, we show for flash chips that the number of partitions should be such that the size of a buffer matches the size of a flash block.

A BufferHash with multiple super tables can be implemented on a flash device as follows. To implement on a flash chip, the chip can be statically partitioned and each partition can be allocated to a super table. A super table writes its incarnations in its partition in a circular way—after the last block of the partition is written, the first block of the partition is erased and the corresponding incarnation is evicted. Such an implementation, however, may not be optimal on an SSD. Even though writes within a single partition are sequential, writes from different super tables to different partitions may interleave with each other, resulting in a performance worse than a single sequential write (see [15] for empirical results). To deal with that, BufferHash uses the entire SSD as a single circular list and writes incarnations from different super tables sequentially, in the order they are flushed to the flash. (This is in contrast to the log rotation approach of Hyperion [20] that provides FIFO semantics for each partition, instead of the entire key space.) Partitioning also naturally supports using multiple SSDs in parallel, by distributing partitions to different SSDs. This scheme, however, spreads the incarnations of a super table all over the SSD. To be able to locate incarnations for a given super table, we maintain their flash addresses along with their Bloom filters and use the addresses during lookup. Note that lookup operations now may require random reads, but random reads are cheap on SSDs.

5.3 Bit-slicing with a Sliding Window

To support efficient Bloom filter lookup, we organize the Bloom filters for all incarnations in a super table in bit-sliced fashion [22]. Suppose a super table contains k incarnations, and the Bloom filter for each incarnation has m bits. We store all k Bloom filters as m k-bit slices, where the i'th slice is constructed by concatenating bit i from each of the k Bloom filters (Fig 3(b)). Then, if a Bloom filter uses h hash functions, we need to check only h bit-slices to check which incarnations may contain a key x. That is, we first apply h hash functions on the key x to get h bit positions in a Bloom filter, retrieve h bit slices at those positions, compute bit-wise AND of those slices. Then, the positions of 1-bits in this aggregated slice, which can be looked up from a pre-computed table, represent the incarnations that may contain the key x.

Symbol	Meaning
N	Total number of items inserted
M	Total memory size
B	Total size of buffers
b	Total size of Bloom filters
k	Number of incarnations in a super table
F	Total flash size
s	Average size taken by a hash entry
h	Number of hash functions
B'	Size of a single buffer (=B/n)
S_p	Size of a flash page/sector
S_b	Size of a flash block

Table 2: Notations used in BufferHash analysis

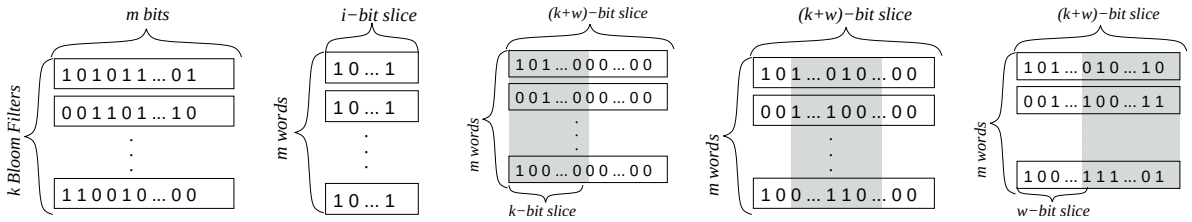
As new incarnations are added and old ones are evicted from an incarnation table, bit slices need to be updated accordingly. A naive approach would reset the left-most bits of all m bit-slices on every eviction, further increasing the cost of an eviction operation. To avoid this, we append w extra bits with every bit-slice, where w is the size of a word that can be reset to 0 with one memory operation. Within each $(k+w)$-bit-slice, a window of k bits represents the Bloom filter bits of k current incarnations (Figure 3(c)), and only these bits are used during lookup. After an incarnation is evicted, the window is shifted one bit right (Figure 3(d)). Since the bit falling off the window is no longer used for lookup, it can be left unchanged. When the window has shifted w bits (Figure 3(e)), entire w-bit words are reset to zero at once, resulting in a small amortized cost. The window wraps around after it reaches the end of a bit-slice.

Note that this is an in-memory optimization; therefore, it will be useful when the workload is mostly memory bound. For example, for a workload with very little redundancy, lookup operations will mostly be unsuccessfully returned from memory; for such a workload, the above optimization can improve the overall throughput.

6. COST ANALYSIS OF BUFFERHASH

In this section, we first analyze the I/O costs of insertion and lookup operations in using BufferHash for flash-based storage, and then use the analytical results to determine optimal values of two important parameters of a BufferHash. We use the notations in Table 5.3 for our analysis.

6.1 Insertion Cost



(a) k Bloom filters (b) Bit-slicing (c) Sliding window (initial state) (d) Sliding window (after 2 evictions) (e) Sliding window (after w evictions)

Figure 3: Bit-slicing with Sliding Window

We now analyze the amortized and the worst case cost of an insertion operation on BufferHash. We assume that the BufferHash is maintained in a flash chip without an FTL; later we show how the results can be trivially extended to SSDs with FTLs. As the measurements in Table 1 show, we use linear cost functions for flash I/Os—reading, writing, and erasing x bits, at appropriate granularities, cost $a_r + b_r x$, $a_w + b_w x$, and $a_e + b_e x$ respectively.

Consider a workload of inserting N keys into a BufferHash. Most insertions are consumed in buffers, and hence do not need any I/O. However, expensive flash I/O occurs when a buffer fills and is flushed to flash. Each flush operation involves three different types of I/O costs. First, each flush requires writing $n_i = \lceil B'/S_p \rceil$ pages, where B' is the size of a buffer in a super tale, and S_p is the size of a flash page (or an SSD sector). This results in a write cost of

$$C_1 = a_w + b_w n_i S_p$$

Second, each flush operation requires evicting an old incarnation from the incarnation table. For simplicity, we consider full discard policy for an evicted incarnation. Note that each incarnation occupies $n_i = \lceil B'/S_p \rceil$ flash pages, and each flash block has $n_b = S_b/S_p$ pages, where S_b is the size of a flash block. If $n_i \geq n_b$, every flush will require erasing flash blocks; otherwise, only n_i/n_b fraction of the flushes will require erasing blocks. Finally, during each erase, we need to erase $\lceil n_i/n_b \rceil$ flash blocks. Putting all together, we get the erase cost of a single flush operation as

$$C_2 = \text{Min}(1, n_i/n_b)(a_e + b_e \lceil n_i/n_b \rceil S_b)$$

Finally, a flash block to be erased may contain valid pages (from other incarnations), which must be backed up before erase and copied back after erase. This can happen because flash can be erased only at the granularity of a block and an incarnations to be evicted may occupy only part a block. In this case, $p' = (n_b - n_i) \mod n_b$ pages must be read and written during each flush. This results in a copying cost of

$$C_3 = a_r + p' b_r S_p + a_w + p' b_w S_p$$

Amortized cost. Consider insertion of N keys into a BufferHash. If each hash entry occupies s space in the BufferHash, each buffer can hold B'/s entries, and hence buffers will be flushed to flash total $n_f = Ns/B'$ times. Thus, the amortized insertion cost is

$$C_{amortized} = n_f(C_1 + C_2 + C_3)/N = (C_1 + C_2 + C_3)s/B'$$

Note that the cost is independent of N and inversely proportional to the buffer size B'.

Worst case cost. An insert operation experiences the worst-case performance when the buffer for the key is full, and hence must be flushed. Thus, the worst case cost of an insert operation is

$$C_{worst} = C_1 + C_2 + C_3$$

SSD. The above analysis can trivially be extended for SSDs. Since the costs C_2 and C_3 in an SSD are handled by its FTL, the overheads of erasing blocks and copying valid pages are reflected in its write cost parameters a_w and b_w. Hence, for an SSD, we can ignore the cost of C_2 and C_3. This results in an amortized cost of insertion is given by $C_{amortized} = C_1 s/B'$ and $C_{worst} = C_1$.

6.2 Lookup Cost

A lookup operation in a super table involves first checking the buffer for the key, checking the Bloom filters to determine which incarnations may contain the key, and reading a flash page for each of those incarnations to actually lookup the key. Since a Bloom filter may produce false positives, some of these incarnations may not contain the key, and hence some of the I/Os may redundant.

Suppose the BufferHash contains n_t super tables. Then, each super table will have $B' = B/n_t$ bits for its buffer, and $b' = b/n_t$ bits for Bloom filters. In steady state, each super table will contain $k = (F/n_t)/(B/n_t) = F/B$ incarnations. Each incarnation contains $n' = B'/s$ entries, and a Bloom filter for an incarnation will have $m' = b'/k$ bits. For a given m' and n', the false positive rate of a Bloom filter is minimized with $h = m' \ln 2/n'$ hash functions [17]. Thus, the probability that a Bloom filter will return a hit (i.e., indicating the presence of a given key) is given by $p = (1/2)^h$. For each hit, we need to read a flash page. Since there are c incarnations, the expected flash I/O cost is given by

$$\begin{aligned} C_{lookup} &= kpc_r = k(1/2)^h c_r \\ &= F/B(1/2)^{bs \ln 2/F} c_r \end{aligned}$$

where c_r is the cost of reading a single flash page from a flash chip, or a single sector from an SSD.

6.3 Discussion

The above analysis can provide insights into benefits and overheads of various BufferHash components not used in

traditional hash tables. Consider a traditional hash table stored on an SSD; without any buffer, each insertion operation would require one random sector write. Suppose, sequentially writing a buffer of size B' is α times more expensive than randomly writing one sector of an SSD. α is typically small even for a buffer significantly bigger than a sector, mainly due to two reasons. First, sequential writes are significantly cheaper than random writes in most existing SSDs. Second, writing multiple consecutive sectors in a batch has better per sector latency. In fact, for many existing SSDs, the value of α is less than 1 even for a buffer size of $256KB$ (e.g., 0.39 and 0.36 for Samsung and MTron SSDs respectively). For Intel SSD, the gap between sequential and random writes is small; still the value of α is less than 10 due to I/O batching.

Clearly, the worst case insertion cost of a BufferHash is α times more expensive than that of a traditional hash table without buffer—a traditional hash table requires writing a random sector, while BufferHash requires sequentially writing the entire buffer. As discussed above the value of α is small for existing SSDs, and for many existing SSDs, BufferHash provides better worst case cost. On the other hand, our previous analysis shows that the amortized insertion cost of BufferHash is at least $\frac{B'}{\alpha s}$ times less than a traditional hash table, even if we assume random writes required by traditional hash table are as cheap as sequential writes required by BufferHash. In practice, random writes are more expensive, and therefore, the amortized insertion cost of a BufferHash is even more cheaper than that of a traditional hash table.

Similarly, a traditional hashtable on flash will need one read operation for each lookup operation, even for the unsuccessful ones. In contrast, the use of Bloom filter can significantly reduce the number of flash reads for unsuccessful lookups. More precisely, if the Bloom filters are configured to provide a false positive rate of p (as shown before), use of Bloom filter can reduce the cost of an unsuccessful lookup by a factor of $1/p$. Note that the same benefit can be realized by using Bloom filters with a traditional hash table as well. Even though BufferHash maintains multiple Bloom filters over different partitions and incarnations, the total size of all Bloom filters will be the same as the size of a single Bloom filter computed over all items. This is because for a given false positive rate, the size of a Bloom filter is proportional to the number of unique items in the filter,

6.4 BufferHash Parameters

Tuning a BufferHash for performance requires carefully setting two key parameters. First, one needs to decide how much DRAM to use, and if a large enough DRAM is available, how much of it is to allocate for buffer and how much to allocate for Bloom filters. Second, once the total size of in-memory buffers is decided, one need to decide how many super tables to use. We now use the previous cost analysis to address these two questions.

Buffer Size. Assume that the total memory size is M bits, of which B bits are allocated for (all) buffers (in all super tables) and $b = M - B$ bits are allocated for Bloom filters. Our previous analysis shows that the value of B does not directly affect insertion cost; however, it affects lookup cost. So, we would like to find the optimal value of B, in the number of bits, that minimizes the expected lookup cost.

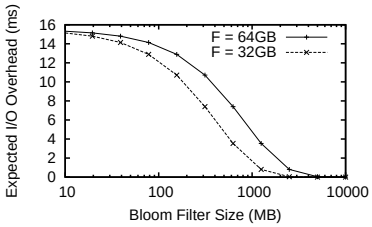


Figure 4: Expected I/O overhead vs Bloom filter size

Intuitively, a buffer size poses a tradeoff between the number of total incarnations and the probability of an incarnation to be read from flash during lookup. As our previous analysis showed, the I/O cost is proportional to the product of the number of incarnations and the hit rate of Bloom filters. In one hand, reducing buffer size increases the number of incarnations, increasing the cost. On the other hand, increasing buffer size leaves less memory for Bloom filters, which increases its false positive rate and I/O cost.

We can use our previous analysis to find a sweet spot within this tradeoff. The analysis shows that the lookup cost is given by $C = F/B \cdot (1/2)^{(M-B)s\ln 2/F} \cdot c_r$. The cost C is minimized when $dC/dB = 0$, or, equivalently $d(log_2(C))/dB = 0$. Solving this equation gives the optimal value of B as,

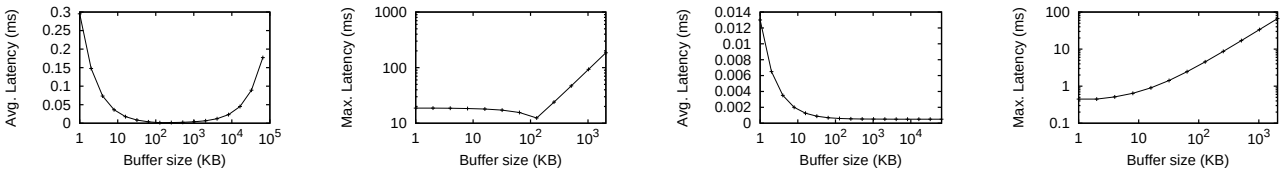
$$B_{opt} = \frac{F}{s(\ln 2)^2} \approx \frac{2F}{s}$$

Interestingly, this optimal value of B does not depend on M; rather, it depends only on the total size F of flash and the average space s taken by each hashed item. Thus, given some memory of size $M > B$, we should use $\approx 2F/s$ bits for buffers, and the remaining for Bloom filters. If additional memory is available, that should be used only for Bloom filters, not for the buffers.

Total Memory Size. We can also determine how much total memory to use for a BufferHash. Intuitively, increasing more memory improves lookup performance, as this allows using larger Bloom filters and lowering false positive rates. Suppose, we want to limit the I/O overhead that happen due to false positives to C_{target}. Then, we can determine b', the required size of Bloom filters as follows.

$$\begin{aligned} C_{target} &\geq \frac{F}{B}\left(\frac{1}{2}\right)^{b's\ln 2/F} \cdot c_r \\ b' &\geq \frac{F}{s(\ln 2)^2}\ln\left(\frac{s(\ln 2)^2 c_r}{C_{target}}\right) \end{aligned}$$

Figure 4 shows required size of a Bloom filter for different expected I/O overheads. As the graph shows, the benefit of using large Bloom filter diminishes after a certain size. For example, for a BufferHash with 32GB flash, allocating 1GB for all Bloom filters is sufficient to limit the expected I/O



(a) Avg. latency (chip) (b) Max. latency (chip) (c) Avg. latency (SSD) (d) Max. latency (SSD)

Figure 5: Amortized and worst-case insertion cost on a flash chip and an Intel SSD. Only flash I/O costs are shown.

overhead C_{target} below 1ms.

In summary, to limit I/O overhead during lookup to C_{target}, a BufferHash requires $(B_{opt} + b')$ bits of memory, of which B_{opt} is used for buffers and the rest for Bloom filters.

Number of Super Tables. Given a fixed memory size B for all buffers, the number of super table determines the size B' of a buffer within a super table. As our analysis shows, B' does not affect the lookup cost; rather, it affects the amortized and worst case cost of insertion. So one should use a suitable value of B' that minimizes the insertion cost.

Figure 5 shows the insertion cost of a BufferHash, based on our previous analysis, in two flash media. (The SSD performs better because it uses multiple flash chips in parallel.). For the flash chip, both amortized and worst-case cost minimize when the buffer size B' matches the flash block size. Thus, buffer size should match a block size for flash chip. The situation is slightly different for SSDs; as Figure 5(b) and (c) show, a large buffer reduces average latency but increases worst case latency. An application should use its tolerance for average- and worst-case latencies and our analytical results to determine the desired size of B' and the number of super tables B/B'.

7. EVALUATION

In this section, we evaluate several BufferHash-based CLAM prototypes with different secondary storage media and compare them with an existing disk-based index.

BufferHash Prototype. We have prototyped BufferHash in around 3000 lines of C++ code and run it on a Linux machine. The hash table in a buffer is implemented with Google Sparsehash library [5]. For simplicity of implementation, different incarnations are written as separate files. A new incarnation is written by overwriting the file corresponding to the oldest incarnation in its super table. Thus the performance numbers we report include small overheads imposed by the `ext3` file system. One can achieve better performance by writing directly to the disk as a raw device, bypassing the file system. We run the prototype on three storage devices: an Intel SSD (model: X18-M, which represents a new generation SSD), a Transcend SSD (model: TS32GSSD25, which represent a relatively old generation but cheaper SSD), and a magnetic disk (Hitachi Deskstar 7K80 drive).

Workload. We use several workloads in our comparative study. Each workload consists of a sequence of lookups and insertions of keys. The keys are generated using random distribution with varying range; the range effects the lookup success rate (LSR) of a key. The workloads are motivated by the WAN optimization application (§8).

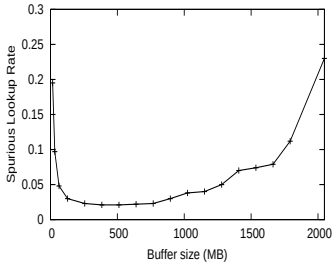


Figure 6: Spurious rate vs. memory allocated to buffers in BufferHash with 4 GB RAM, 32 GB Flash

# of flash I/O	Probability		Latency (ms)	
	0% LSR	40% LSR	Flash chip	Intel SSD
0	0.9899	0.6032	0	0
1	0.0094	0.3894	0.24	0.31
2	0.0005	0.0073	0.48	0.62
3	0.00005	0.00003	0.72	0.93

Table 3: Lookup latency probability distribution

7.1 BufferHash Parameters

As mentioned in §2, our key motivating applications like WAN optimization and deduplication employ 16-32GB indexes. In our evaluation, we use a similar index: specifically, we use 32GB of slow storage (flash, SSD or disk) and 4GB of DRAM. The buffer size of a super table is set to 128 KB, as suggested by our analysis in §6.4. Each hash entry takes 16 bytes of space. However, we limit the utilization of the hash table in a buffer to 80%—a higher utilization increases hash collision and lookup latency. Thus, each buffer (and each incarnation) contains around 6500 hash entries.

According to the analysis in §6.4, the optimal size of buffers for the above configuration is 416MB. We now experimentally validate the value. Figure 6 shows the variation of false positive rates as the memory allocated to buffers is varied from 16 MB to 2048 MB in our prototype. As our analysis indicated, allocating a very small memory for all buffers gives a high false positive rate (e.g., 0.2 for 16MB) and this imposes significant lookup overhead. Increasing this memory to 384MB reduces spurious lookup rate (due to false positives) to 0.02, which is optimal for our setup. Further increasing the size of buffers increases the spurious rate (e.g., 0.23 for 2048 MB). The trend is similar to that shown by our analysis in §6.4. Moreover, the experimental optimal buffer size (384MB) is close to our analytical optimal (416MB); the small difference is because our analysis allows the optimal number of hash functions to be an any positive real number.

7.2 I/O Rates of BufferHash Operations

BufferHash consists of two levels of memory/storage hierarchy. The storage layer is relatively slow, but many of the operations are performed on memory. Table 3 shows the distribution of flash I/Os required by a lookup operation in our BufferHash prototype, under two different lookup success ratio (LSR). As shown, most of the lookups are answered from memory. Moreover, $\approx 99\%$ lookups require at most only one flash read. Table 3 also shows distributions of lookup latencies for two different flash devices, based on their measured latencies as shown in Table 1. As shown, more than 99.99% of the lookups are answered within 1 ms.

Since BufferHash buffers writes in memory before writing to flash, most of the insert operations are done in memory. Since a buffer holds around 6500 items, only 1 out of 6500 insertions on average requires writing to flash. So, the average and median insert latencies for BufferHash are minimal (average ≈ 0.02ms for Intel SSD). The worst case insert latency, when a buffer is flushed to the Intel SSD, is 0.83 ms.

7.3 Performance of SSD-based CLAMs

We now evaluate two CLAMs: 1) **BH+SSD**: our BufferHash prototype running on an SSD and 2) **BH+Disk**: BufferHash running on a magnetic disk, and report the measured latencies of different BufferHash operations. This helps us understand how much performance benefit of a CLAM comes from using SSD. We use a workload with 40% look-up success rate over random keys with interleaved inserts and lookups.

In Figure 7(a), we show the distribution of latencies for lookup operations on the **BH+SSD** CLAM with an Intel and a Transcend SSD. Around 63% of the time, the in-memory bloom filter saves the lookups from going to the slow media; of course, in the rare case of false positives some additional latency is incurred, but we see very negligible impact in practice (Recall that BufferHash is configured for < 0.02 false positive rate). 99.8% of the lookup times are less than 0.176 ms for the Intel SSD. For Transcend SSD, 90% of the lookup times are under 0.6ms and the maximum is 1ms.

In Figure 7(b), we show the latencies for insert operations on different CLAMs. As shown, most of the operations involve only the main memory, so the average insert cost is very small (0.02ms and 0.032ms for Intel and Transcend SSD). The worst case latency for insert is 0.9ms and 20ms for Intel and Transcend SSDs respectively.

Figures 7(a) and (b) also show the latencies for lookup and inserts in **BH+Disk**. Lookup latencies range from 0.1 to 12 ms, an order or magnitude higher than the SSD prototypes due to the high seek latencies in disks. The average insert cost is very small and the worst case insert cost is 12 ms, corresponding to high seek latency.

The results show using different storage media gives rise to systems with different performance. Note that the prices of these different storage media are different as well. Thus application and system designers with different cost-performance constraints could select a suitable CLAM from the above set.

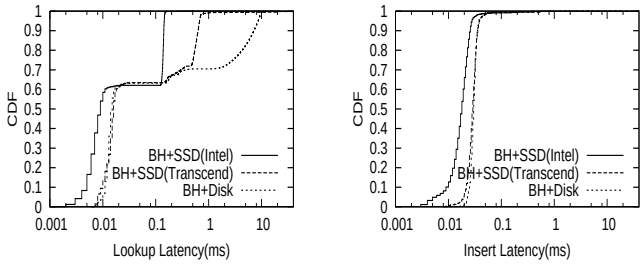


Figure 7: BufferHash **latencies on different media**

Intel SSD is the most expensive of them all, but it offers the best latencies for lookups and inserts, and we expect high-end systems like WAN optimizers for 1Gbps or faster links to leverage these. CLAMs based on the lower-end Transcend SSDs are less expensive but they offer an order or magnitude worse worst-case latencies for both operations. Finally, disk-based indexes are the cheapest but the worst-case latencies they offer are a further order of magnitude worse. WAN optimizers designed for lower speed network links could employ the latter two variants, for instance.

7.4 Comparison with DB-Indexes

We now compare our BufferHash prototype against the hash table structure in Berkeley-DB [6], a popular database index, with the same workload above. (We also considered the B-Tree index of the same database, but the performance was worse than the hash table.) We consider the following systems: a) **DB+SSD**: Berkeley-DB running on an SSD, and b) **DB+Disk**: Berkeley-Db running on a magnetic disk.

Figure 8 (a) and (b) show the look-up and the insert latencies for the two systems. More than 60% of the lookups and more than 40% of the inserts have latencies greater than 5 ms for **DB+Disk**. Surprisingly, for the Intel SSD, around 40% of lookups and 40% inserts have latencies greater than 5ms! This is counterintuitive given that Intel SSD has significantly faster random I/O latency (0.15 ms) than magnetic disks. This is explained by the fact that the low latency of an SSD is achieved only when the write load on the SSD is "low"; i.e., there are sufficient pauses between bursts of writes so that the SSD gets enough time to clean dirty blocks to produce erased blocks for new writes [15]. Under a high write rate, the SSD quickly uses up its pool of erased blocks and then I/Os block until it has reclaimed enough space from dirty blocks by performing garbage collection.

This result shows that existing disk based solutions that send all I/O requests to disks are not likely to perform well on SSDs, even if SSDs are significantly faster than disks (for workloads that give SSDs sufficient time for garbage collection). In other words, these solutions are not likely to exploit the performance benefit of SSDs under "high" write load. In contrast, since BufferHash writes to flash only when the buffer fills up, it poses a relatively "light" load on the SSD, resulting in faster reads.

Comparing Figure 7 and Figure 8 shows that **BH+Disk**

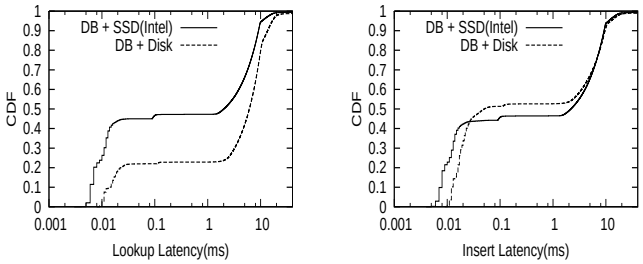


Figure 8: Berkeley-DB Latencies

Buffer size (KB)	Priority-discard	Update-discard
16	0.38 ms	0.43 ms
32	0.67 ms	0.84 ms
64	1.39 ms	1.73 ms
128	2.73 ms	3.41 ms

Table 4: Increase in worst-case insert latency (ms) for different eviction policies

performs better than **DB+SSD**, implying that the benefit due to using BufferHash dominates the benefit due to using SSD.

We do note that it is possible to supplement the Berkeley-DB index with an in-memory Bloom filter to improve lookups. For disks, we note that lookup performance will become comparable to that achieved using BufferHash (Figure 7 (a)). For Intel-SSD, the lookup performance will improve but not become comparable to BufferHash, as the lookups going to SSD will still be effected by the garbage collection overhead imposed by fast insertions in the workload.

7.5 Eviction Support in BufferHash

Our experiments so far are based on the default eviction policy of full discard for BufferHash. Here, we consider two partial discard policies discussed in § 5.1.2: the *update-based policy* where only the stale entries are discarded, and the *priority-based policy* where entries with priority lower than a threshold are discarded. Note that these policies increase the worst-case insertion cost as they require reading entries from the oldest incarnation and re-inserting some of them to the buffer. We evaluate the two policies for the BufferHash configuration of 32 incarnations. Priority-based discard policy increases the worst-case insertion cost by 2.73 ms, while update-based policy increases it by 3.41 ms. Update-based discard is more expensive as it needs to search Bloom filters to see if an entry has already been updated.

The overhead of partial discard depends on the buffer size (which determines the number of items in the evicted incarnation). Table 4 shows the increase in worst-case insertion cost for these two policies for different buffer sizes with a configuration of 32 incarnations per super table. An application using partial discard can choose the buffer size according to how much worst-case insert latency it can tolerate.

8. WAN OPTIMIZATION

In this section, we perform a front-to-end evaluation of employing a CLAM designed using BufferHash in a WAN optimization application. We use a variety of traces, both synthetic and real, to examine the benefits and trade-offs of the CLAM under different settings.

Real traces. We collect packet traces at a large US university's access link to the Internet and at the access link of a high volume Web server in the university. We then construct an object-level trace by grouping packets with the same connection 4-tuple into a single object.

Synthetic traces. Synthetic workloads allow us to flexibly study the benefits of CLAMs and traditional approaches under different settings. We construct synthetic traces based on key properties of real traces. A synthetic trace has 3 main parameters: (1) object size distribution, (2) overall redundancy and (3) content match length distribution.

(1) We choose object sizes from a Zipf distribution with $\alpha = 1$. (2) We assume that fingerprints (i.e., a random hashes) are computed for 1KB chunks of content in each object. Depending on the overall redundancy in the trace, a fraction of fingerprints may observe a match, implying there is overlap in the object's content and the content in prior objects. (3) When a match is observed, the match length distribution determines the amount of redundant content "in the vicinity" of the fingerprinted region. This models how WAN optimizers identify content overlap, namely, by "growing" the data chunk corresponding to a matching fingerprint until a maximal match region is found. We use a simple match length distribution derived from real traces.

Emulator. To evaluate the application, we built *a WAN optimizer emulator* to emulate a scenario where two branch offices of an enterprise, connected by a link of certain capacity, wish to employ WAN optimization to improve effective link utilization and speed up transfers. Each end has an *object cache* that holds items sent earlier to the counterpart campus (we assume that the object cache resides on a disk). New objects are fingerprinted and matched against the content in the object cache to identify redundancy. A *fingerprint index* holds information on the mapping between fingerprints and the on-disk locations of the objects in which the corresponding chunks were found. We study how the benefits of the WAN optimizer depend on the architecture of the index.

There are two differences between our emulator and a real WAN optimizer. (1) The emulator approximates the latencies of reading objects from, and writing objects, to an object cache. For reading/writing X bytes of content, we assume that the latency needed is X/D, where D is the disk's sequential read/write throughput. (2) The network and storage subsystems are assumed to have no interaction, which may not be true in practice. In our emulator, a compressed object being transmitted does not "block" an arriving yet-to-be compressed object from getting processed (i.e., fingerprints getting computed and then getting looked up).

We replay synthetic and real workloads on this emulator using our Intel-SSD CLAM prototype based on BufferHash and on-disk index based on Berkeley-DB. Our CLAM implements the full BufferHash functionality, including eviction based on full discard and windowed bit slicing. The CLAM

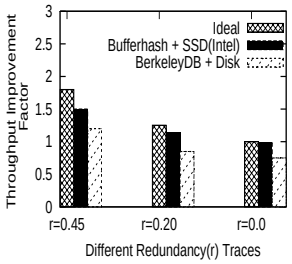


Figure 9: End-to-end throughput improvement for Berkeley-DB on disk and Intel SSD-based CLAM on traces with different levels of redundancy.

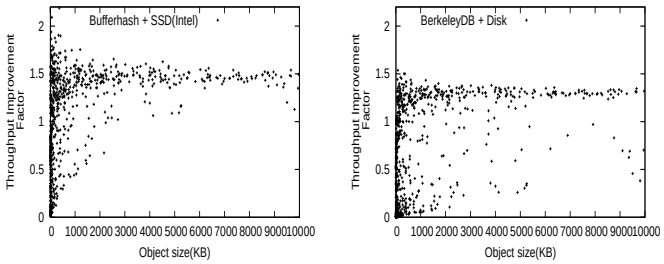


Figure 10: Heavy Load: Throughput improvement per object for (a) BufferHash-based CLAM using Intel SSD and (b) Berkeley-DB on disk

is configured with 4GB RAM and 32GB of SSD. Berkeley-DB holds 32GB index on disk.

Performance metrics. Our key metrics are the end-to-end and per-object throughput improvements from employing WAN optimization. Typically, WAN optimizers transmit bytes from all compressed objects in a single or a small number of TCP connections to avoid TCP start-up effects. Thus, the throughput of an object compressed to a certain size S, where S can be small or large, can be approximated by $S/\Delta T$, where ΔT is the difference between the time when the object arrived till the time when the last byte of the compressed object was sent. Note that ΔT includes the time to fingerprint the object, look for matches and compress the object. In addition, it may include delays from actions that the WAN optimizer was taking on earlier objects when the object in question arrived (e.g., updating the index with fingerprints for the earlier object).

8.1 Analysis Using Synthetic Traces

Unless otherwise specified, we assume that the link between the WAN optimizers is heavily loaded, i.e., new objects arrive at one of the WAN optimizers before fingerprints for earlier objects are inserted into the index. We also studied other "light" and "medium" load situations and our observations were qualitatively similar — we omit these results for brevity. The link capacity is assumed to be 10Mbps.

We study how different approaches used for holding the index – i.e. CLAMs vs Berkeley-DB – impact the benefits of the WAN optimizer. As a baseline, we compare the approaches against the ideal achievable benefits which depends simply on the amount of redundancy in all content.

Figure 9 shows the end-to-end throughput improvement from WAN optimization on synthetic workloads with three different levels of redundancy. For the highly redundant trace, using the Intel-based CAM to hold the index allows the WAN optimizer to achieve a throughput improvement, averaged across all objects, of 1.51X, compared to the optimal of 1.8X. There are many fingerprint matches in the highly redundant trace, and the CLAM-based optimizer incurs substantial overhead in reading objects from disk for each match. There is no overhead due to the index itself because of the optimized lookup and insert performance of BufferHash.

In contrast, using Berkeley-DB, the WAN optimizer achieves just 1.2X average improvement. The additional overhead compared to the CLAM-based optimizer is due to the high lookup and insert latencies for the on-disk index. When there is no redundancy in network traffic, the Berkeley-DB based optimizer has a very large overhead of 30%, while the CLAM-based optimizer imposes little or no overhead.

We now take a closer look at the improvements for the high redundancy workload. Figures 10 (a) and (b) show the relative throughput improvement on an object-by-object basis. We see that Berkeley-DB has a negative effect on the throughputs of a large number of objects (compared to ideal), especially objects 500KB or smaller; their throughput is worsened by a factor of two or more due to the high costs of lookups and inserts (the latter for fingerprints of prior objects). When a small object observes a match, the accompanying disk reads also impose an overhead. Our CLAM also imposes overhead on some of these objects, but this happens on far fewer occasions and it is significantly lower. As before, the overhead when using the CLAM arises only due to disk reads for matching objects.

8.2 Analysis Using Real Traces

We repeat the above analysis for real object traces. The university-wide trace had a redundancy of 5% and the trace from the Web server had a redundancy of 38%. We evaluate a heavy load situation with a link speed of 100Mbps.

We note from Figure 10(a) that the Berkeley-DB optimizer achieves an improvement of 1.2X compared to the optimal 1.62X; thus Berkeley-DB based on-disk index offers less than a *third* of the optimal benefits. In contrast, the BufferHash-based CLAM performs *very close to optimal*.

Compared to the synthetic trace (Figure 9), we see that the overhead incurred by the CLAM-based optimizer is small, even though the redundancy is very high. This is because there were few fingerprint matches per object in this trace but each match resulted in the WAN optimizer identifying a large region of overlap with a prior object. As a result the amortized overheads due to accessing the SSD in order to read matching fingerprint information and the overhead of reading matching objects from the object cache are both low. In case of the less-redundant trace, our CLAM imposes a negligible overhead as most lookups do not go to the SSD.

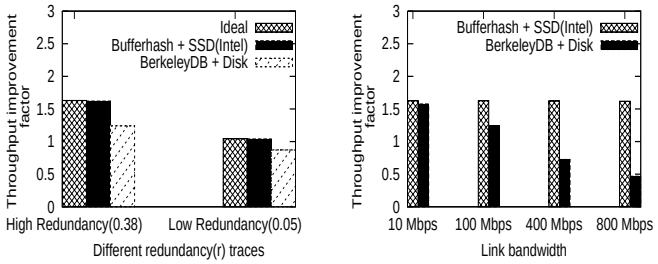


Figure 11: (a) End-to-end throughput improvement for high and low redundancy traces (link speed of 100Mbps) (b) Effect of link speeds on throughput improvement.

Next, we study the effect of link speed on the overall benefits. The greater the speed, the greater the demand on the WAN optimizer to support low latency index operations. Indexes with poor insert/lookup latencies may perform well at low link speeds, but their peak-time performance will diminish rapidly at higher link rates.

A comparison of CLAMs against Berkeley-DB for four different link speeds is shown in Figure 11. We see that the performance of the Berkeley-DB based index degrades rapidly with link speed. In contrast, the benefits from our Intel-SSD based CLAM are sustained benefits even until 800Mbps.

9. CONCLUSIONS

Recent years have seen the emergence of applications that maintain indexes up to 100GB or more in size and require constants lookups into and updates of the index. Current alternative to maintaining such indexes are either too expensive (e.g. using DRAM) or too constraining (e.g. using disk-based indexes). To support such application we propose and design CLAMs, or cheap and large CAMs. Our CLAMs use flash-based storage and cost slightly more than disk-based indexes, but are two orders of magnitude faster, and could hold a few hundred GB worth of index information. We introduce a new data structure, called BufferHash, to facilitate rapid random writes and lookups on flash-based storage.

Our analytical study and evaluation of prototypes based on high-end SSDs show that our CLAMs offer random update and lookup latencies of 0.02 and 0.06ms on average, implying that they can support 10,000 or more insertions and lookups per second on average. We show how such CLAMs can improve the performance of networked systems, using the example of WAN optimization. Our CLAMs can improve the bandwidth savings from such WAN optimizers by nearly three-fold in situations of high load.

We note that alternate designs are possible for CLAMs. In particular, BufferHash is just one of a family of data structures that enable flash storage to provider CLAM-like functionality. CLAMs can also be designed for traditional storage systems and our evaluations show that BufferHash is ideally suited for such designs as well.

10. REFERENCES

[1] BlueCoat: WAN Optimization. `http://www.bluecoat.com/`.

[2] Cisco Wide Area Application Acceleration Services. `http://www.cisco.com/en/US/products/ps5680/Products_Sub_Category_Home.html`.

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

[4] Dropbox. `http://www.getdropbox.com`.

[5] Google sparse hash library. `http://code.google.com/p/google-sparsehash/`.

[6] Oracle Berkeley-DB. `http://www.oracle.com/technology/products/berkeley-db/index.html`.

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

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

[9] WAN Optimization Design. Private communication with a major vendor.

[10] D. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[11] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26, 2003.

[12] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms. In *4th International Workshop on Algorithms and Data Structures (WADS)*, 1995.

[13] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2), 2007.

[14] L. Bouganim, B. Jónsson, and P. Bonnet. The uFlip Benchmark. `http://www.uflip.org`, 2009.

[15] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *CIDR*, 2009.

[16] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP lookups. In *IEEE INFOCOM*, 2001.

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

[18] A. Caulfield, L. Grupp, and S. Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS*, 2009.

[19] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD*, 2003.

[20] P. J. Desnoyers and P. Shenoy. Hyperion: high volume stream archival for retrospective querying. In *USENIX*, 2007.

[21] Endace Inc. Endace DAG3.4GE network monitoring card. `http://www.endace.com/`, 2009.

[22] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems*, 2(4):267–288, 1984.

[23] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *ACM SOSP*, 1987.

[24] G. Iannaccone, C. Diot, D. McAuley, A. Moore, I. Pratt, and L. Rizzo. The CoMo white paper. Technical Report IRC-TR-04-17, Intel Research, 2004.

[25] Intel-Corporation. Understanding the Flash Translation Layer (FTL) specification. www.embeddedfreebsd.org/Documents/Intel-FTL.pdf, 1998.

[26] A. Kirsch and M. Mitzenmacher. The power of one move: Hashing schemes for hardware. In *IEEE INFOCOM*, 2008.

[27] T. Koponen, M. Chawla, B. Chun, A. Ermolinskiy, K. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *ACM SIGCOMM*, 2006.

[28] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *ACM SenSys*, 2006.

[29] S. Moon and T. Roscoe. Metadata management of terabyte datasets from an ip backbone network: Experience and challenges. In *ACM SIGMOD Workshop on Network-Related Data Management*, 2001.

[30] S. Nath and P. B. Gibbons. Online Maintenance of Very Large Random Samples on Flash Storage . In *VLDB*, 2008.

[31] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *ACM/IEEE IPSN*, 2007.

[32] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *NSDI*, 2007.

[33] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM SOSP*, 1991.

[34] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *In ACM SIGCOMM*, 2005.

[35] StreamBase Inc. Streambase: Real-time low latency data processing with a stream processing engine. `http://www.streambase.com/`, 2009.

[36] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[37] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: an efficient index structure for fash-based sensor devices. In *USENIX FAST*, 2005.

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