ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory

Biplob Debnath[*] *Sudipta Sengupta*[‡] *Jin Li*[‡]
[‡]*Microsoft Research, Redmond, WA, USA*
[*]*University of Minnesota, Twin Cities, USA*

Abstract

Storage deduplication has received recent interest in the research community. In scenarios where the backup process has to complete within short time windows, *inline* deduplication can help to achieve higher backup throughput. In such systems, the method of identifying duplicate data, using disk-based indexes on chunk hashes, can create throughput bottlenecks due to disk I/Os involved in index lookups. RAM prefetching and bloom-filter based techniques used by Zhu et al. [42] can avoid disk I/Os on close to 99% of the index lookups. Even at this reduced rate, an index lookup going to disk contributes about 0.1msec to the *average* lookup time – this is about 1000 times slower than a lookup hitting in RAM. We propose to reduce the penalty of index lookup misses in RAM by orders of magnitude by serving such lookups from a flash-based *index*, thereby, increasing inline deduplication throughput. Flash memory can reduce the huge gap between RAM and hard disk in terms of both cost and access times and is a suitable choice for this application.

To this end, we design a *flash-assisted* inline deduplication system using ChunkStash[1], a chunk metadata store on flash. ChunkStash uses one flash read per chunk lookup and works in concert with RAM prefetching strategies. It organizes chunk metadata in a log-structure on flash to exploit fast sequential writes. It uses an in-memory hash table to index them, with hash collisions resolved by a variant of cuckoo hashing. The in-memory hash table stores (2-byte) compact key signatures instead of full chunk-ids (20-byte SHA-1 hashes) so as to strike tradeoffs between RAM usage and false flash reads. Further, by indexing a small fraction of chunks per container, ChunkStash can reduce RAM usage significantly with negligible loss in deduplication quality. Evaluations using real-world enterprise backup datasets show that ChunkStash outperforms a hard disk index based inline deduplication system by 7x-60x on the metric of backup throughput (MB/sec).

1 Introduction

Deduplication is a recent trend in storage backup systems that eliminates redundancy of data across full and incremental backup data sets [30, 42]. It works by splitting files into multiple chunks using a content-aware chunking algorithm like Rabin fingerprinting and using 20-byte SHA-1 hash signatures [34] for each chunk to determine whether two chunks contain identical data [42]. In *inline* storage deduplication systems, the chunks arrive one-at-a-time at the deduplication server from client systems. The server needs to lookup each chunk hash in an index it maintains for all chunks seen so far for that storage location (dataset) instance. If there is a match, the incoming chunk contains redundant data and can be deduplicated; if not, the (new) chunk needs to be added to the system and its hash and metadata inserted into the index. The metadata contains information like chunk length and location and can be encoded in up to 44 bytes (as in [42, 30]). The 20-byte chunk hash (also referred to as *chunk-id*) is the *key* and the 44-byte metadata is the *value*, for a total *key-value pair* size of 64 bytes.

Because deduplication systems currently need to scale to tens of terabytes to petabytes of data volume, the chunk hash index is too big to fit in RAM, hence it is stored on hard disk. Index operations are thus throughput limited by expensive disk seek operations which are of the order of 10msec. Since backups need to be completed over tight windows of few hours (over nights and weekends), it is desirable to obtain high throughput in inline storage deduplication systems, hence the need for a fast index for duplicate chunk detection. The index may be used in other portions of the deduplication pipeline also. For example, a recently proposed algorithm for chunking the data stream, called *bimodal chunking* [27], requires access to the chunk index to determine whether an incoming chunk has been seen earlier or not. Thus, multiple functionalities in the deduplication pipeline can benefit from a fast chunk index.

RAM prefetching and bloom-filter based techniques used by Zhu et al. [42] can avoid disk I/Os on close to 99% of the index lookups and have been incorporated in production systems like those built by Data Domain. Even at this reduced rate, an index lookup going to disk contributes about 0.1msec to the *average* lookup time – this is about 10^3 times slower than a lookup hitting in RAM. We propose to reduce the penalty of index lookup misses in RAM by orders of magnitude by serving such lookups from a flash memory based *key-value store*, thereby, increasing inline deduplication throughput. Flash memory is a natural choice for such a store, providing persistency and 100-1000 times lower access times than hard disk. Compared to DRAM, flash access times are about 100 times slower. Flash stands in the

[1] stash: A secret place where something is hidden or stored.

middle between DRAM and disk also in terms of cost [29] – it is about 10x cheaper than DRAM, while about 10x more expensive than disk – thus, making it an ideal gap filler between DRAM and disk and a suitable choice for this application.

1.1 Estimating Index Lookup Speedups using Flash Memory

We present a back-of-the-envelope calculation for decrease in average chunk index lookup time when flash memory is used as the metadata store for chunk-id lookups. This can be viewed as a necessary sanity check that we undertook before plunging into a detailed design and implementation of a flash-assisted inline storage deduplication system. We use the fundamental equation for average access time in multi-level memory architectures. Let the hit ratio in RAM be h_r. Let the lookup times in RAM, flash, and hard disk be t_r, t_f, and t_d respectively.

In the hard disk index based deduplication system described in Zhu et al. [42], prefetching of chunk index portions into RAM is shown to achieve RAM hit ratios of close to 99% on the evaluated datasets. Lookup times in RAM can be estimated at $t_r = 1\mu$sec, as validated in our system implementation (since it involves several memory accesses, each taking about 50-100nsec). Hard disk lookup times are close to $t_d = 10msec$, composed of head seek and platter rotational latency components. Hence, the average lookup time in this case can be estimated as

$$t_r + (1 - h_r) * t_d = 1\mu\text{sec} + 0.01 * 10\text{msec} = 101\mu\text{sec}$$

Now let us estimate the average lookup time when flash is used to serve index lookups that miss in RAM. Flash access times are around $t_f = 100\mu$sec, as obtained through measurement in our system. Hence, the average lookup time in a flash index based system can be estimated as

$$t_r + (1 - h_r) * t_f = 1\mu\text{sec} + 0.01 * 100\mu\text{sec} = 2\mu\text{sec}$$

This calculation shows a *potential speedup of* 50*x using flash* for serving chunk metadata lookups vs. a system that uses hard disk for the same. *That is a speedup of more than one order of magnitude.* At 50x index lookup speedups, other parts of the system could become bottlenecks, e.g., operating system and network/disk bottlenecks for data transfer. So we do not expect the overall system speedup (in terms of backup throughput MB/sec) to be 50x in a real implementation. However, the point we want to drive home here is that flash memory technology can help to get the index lookup portion of inline storage deduplication systems far out on the scaling curve.

1.2 Flash Memory and Our Design

There are two types of popular flash devices, NOR and NAND flash. NAND flash architecture allows a denser layout and greater storage capacity per chip. As a result, NAND flash memory has been significantly cheaper than DRAM, with cost decreasing at faster speeds. NAND flash characteristics have led to an explosion in its usage in consumer electronic devices, such as MP3 players, phones, and cameras.

However, it is only recently that flash memory, in the form of Solid State Drives (SSDs), is seeing widespread adoption in desktop and server applications. For example, MySpace.com recently switched from using hard disk drives in its servers to using PCI Express (PCIe) cards loaded with solid state flash chips as primary storage for its data center operations [6]. Also very recently, Facebook announced the release of Flashcache, a simple write back persistent block cache designed to accelerate reads and writes from slower rotational media (hard disks) by caching data in SSDs [7]. These applications have different storage access patterns than typical consumer devices and pose new challenges to flash media to deliver sustained and high throughput (and low latency).

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

To this end, we present the design and evaluation of ChunkStash, a *flash-assisted* inline storage deduplication system incorporating a high performance chunk metadata store on flash. When a key-value pair (i.e., chunk-id and its metadata) is written, it is sequentially logged in flash. A specialized RAM-space efficient hash table index employing a variant of cuckoo hashing [35] and compact key signatures is used to index the chunk metadata stored in flash memory and serve chunk-id lookups using one flash read per lookup. ChunkStash works in concert with existing RAM prefetching strategies. The flash requirements of ChunkStash are well within the range of currently available SSD capacities – as an example, ChunkStash can index order of *terabytes* of unique (deduplicated) data using order of *tens of gigabytes* of flash. Further, by indexing a small fraction of chunks per container, ChunkStash can reduce RAM usage sig-

nificantly with negligible loss in deduplication quality.

In the rest of the paper, we use NAND flash based SSDs as the architectural choice and simply refer to it as flash memory. We describe the internal architecture of SSDs in Section 2.

1.3 Our Contributions

The contributions of this paper are summarized as follows:

- **Chunk metadata store on flash:** ChunkStash organizes key-value pairs (corresponding to chunk-id and metadata) in a log-structured manner on flash to exploit fast sequential write property of flash device. It serves lookups on chunk-ids (20-byte SHA-1 hash) using one flash read per lookup.

- **Specialized space efficient RAM hash table index:** ChunkStash uses an in-memory hash table to index key-value pairs on flash, with hash collisions resolved by a variant of cuckoo hashing. The in-memory hash table stores compact key signatures instead of full keys so as to strike tradeoffs between RAM usage and false flash reads. Further, by indexing a small fraction of chunks per container, ChunkStash can reduce RAM usage significantly with negligible loss in deduplication quality.

- **Evaluation on enterprise datasets:** We compare ChunkStash, our flash index based inline deduplication system, with a hard disk index based system as in Zhu et al. [42]. For the hard disk index based system, we use BerkeleyDB [1], an embedded key-value store application that is widely used as a comparison benchmark for its good performance. For comparison with the latter system, we also include "a hard disk replacement with SSD" for the index storage, so as to bring out the performance gain of ChunkStash in not only using flash for chunk metadata storage but also in its use of flash aware algorithms. We use three enterprise backup datasets (two full backups for each) to drive and evaluate the design of ChunkStash. Our evaluations on the metric of backup throughput (MB/sec) show that ChunkStash outperforms (i) a hard disk index based inline deduplication system by 7x-60x, and (ii) SSD index (hard disk replacement but flash unaware) based inline deduplication system by 2x-4x.

The rest of the paper is organized as follows. We provide an overview of flash-based SSD in Section 2. We develop the design of ChunkStash in Section 3. We evaluate ChunkStash on enterprise datasets and compare it with our implementation of a hard disk index based inline deduplication system in Section 4. We review related work in Section 5. Finally, we conclude in Section 6.

2 Flash-based SSD

A Solid State Drive(SSD) consists of flash chip(s) and flash translation layer (FTL). In a flash chip, data is stored in an array of flash memory blocks. Each block spans 32-64 pages, where a page is the smallest unit of read and write operations. In flash memory, unlike disks, random read operations are as fast as sequential read operations as there is no mechanical head movement. However, unlike disk, read and write operations do not exhibit symmetric behavior. This asymmetry arises as flash memory does not allow in-place update (i.e., overwrite) operations. Page write operations in a flash memory must be preceded by an erase operation and within a block pages need to be written sequentially. Read and write operations are performed in page-level, while erase operations are performed in block-level. In addition, before the erase is being done on a block, the valid (i.e., not over-written) pages from that block need to be moved to a new pre-erased blocks. Thus, a page update operation incurs lot of page read and write operations. The typical access latencies for read, write, and erase operations are 25 microseconds, 200 microseconds, and 1500 microseconds, respectively [9]. Besides the in-place update problem, flash memory exhibits another limitation – a flash block can only be erased for limited number of times (e.g., 10K-100K) [9].

The Flash Translation layer (FTL) is an intermediate software layer inside an SSD, which hides the limitations of flash memory and provides a disk like interface. FTL receives logical read and write commands from the applications and converts them to the internal flash memory commands. To emulate disk like in-place update operation for a logical page (L_p), the FTL writes data into a new physical page (P_p), maintains a mapping between logical pages and physical pages, and marks the previous physical location of L_p as invalid for future garbage collection. FTL uses various wear leveling techniques to even out the erase counts of different blocks in the flash memory to increase its overall longevity [20]. Thus, FTL allows current disk based application to use SSD without any modifications. However, it needs to internally deal with current limitations of flash memory (i.e., constraint of erasing a block before overwriting a page in that block). Recent studies show that current FTL schemes are very effective for the workloads with sequential access write patterns. However, for the workloads with random access patterns, these schemes show very poor performance [21, 23, 26, 28, 32]. One of the design goals of ChunkStash is to use flash memory in an

FTL friendly manner.

3 Flash-assisted Inline Deduplication System

We follow the overall framework of production storage deduplication systems currently in the industry [42, 30]. Data chunks coming into the system are identified by their SHA-1 hash [34] and looked up in an index of currently existing chunks in the system (for that storage location or stream). If a match is found, the metadata for the file (or, object) containing that chunk is updated to point to the location of the existing chunk. If there is no match, the new chunk is stored in the system and the metadata for the associated file is updated to point to it. (In another variation, the chunk hash is included in the file/object metadata and is translated to chunk location during read access.) Comparing data chunks for duplication by their 20-byte SHA-1 hash instead of their full content is justified by the fact that the probability of SHA-1 hash match for non-identical chunks is less by many orders of magnitude than the probability of hardware error [37]. We allocate 44 bytes for the metadata portion. The 20-byte chunk hash is the key and the 44-byte metadata is the value, for a total key-value pair size of 64 bytes.

Similar to [42] and unlike [30], our system targets *complete deduplication* and ensures that no duplicate chunks exist in the system after deduplication. However, we also provide a technique for RAM usage reduction in our system that comes at the expense of marginal loss in deduplication quality.

We summarize the main components of the system below and then delve into the details of the chunk metadata store on flash which is a new contribution of this paper.

Data chunking: We use *Rabin fingerprinting* based sliding window hash [38] on the data stream to identify chunk boundaries in a content dependent manner. A chunk boundary is declared when the lower order bits of the Rabin fingerprint match a certain pattern. The length of the pattern can be adjusted to vary the average chunk size. The average chunk size in our system is 8KB as in [42]. *Ziv-Lempel compression* [43] on individual chunks can achieve an average compression ratio of 2:1, as reported in [42] and also verified on our datasets, so that the size of the stored chunks on hard disk averages around 4KB. The SHA-1 hash of a chunk serves as its chunk-id in the system.

On-disk Container Store: The *container store* on hard disk manages the storage of chunks. Each container stores at most 1024 chunks and averages in size around 4MB. (Because of the law of averages for this large number (1024) of chunks, the deviation of container size from this average is relatively small.) As new (unique) chunks come into the system, they are appended to the current container buffered in RAM. When the current container reaches the target size of 1024 chunks, it is sealed and written to hard disk and a new (empty) container is opened for future use.

Chunk Metadata Store on Flash (ChunkStash): To eliminate hard disk accesses for chunk-id lookups, we maintain, in flash, the metadata for all chunks in the system and index them using a specialized RAM index. The chunk metadata store on flash is a new contribution of this paper and is discussed in Section 3.1.

Chunk and Container Metadata Caches in RAM: A *cache for chunk metadata* is also maintained in RAM as in [42]. The fetch (prefetch) and eviction policies for this cache are executed at the container level (i.e., metadata for all chunks in a container). To implement this container level prefetch and eviction policy, we maintain a fixed size *container metadata cache* for the containers whose chunk metadata are currently held in RAM – this cache maps a container-id to the chunk-ids it contains. The size of the chunk metadata cache is determined by the size of the container metadata cache, i.e., for a container metadata cache size of C containers, the chunk metadata cache needs to hold $1024*b$ chunks. A distinguishing feature of ChunkStash (compared to the system in [42] is that it does not need to use bloom filters to avoid secondary storage (hard disk or flash) lookups for non-existent chunks.

Prefetching Strategy: We use the basic idea of predictability of sequential chunk-id lookups during second and subsequent full backups exploited in [42]. Since datasets do not change much across consecutive backups, duplicate chunks in the current full backup are very likely to appear in the same order as they did in the previous backup. Hence, when the metadata for a chunk is fetched from flash (upon a miss in the chunk metadata cache in RAM), we prefetch the metadata for all chunks in that container into the chunk metadata cache in RAM and add the associated container's entry to the container metadata cache in RAM. Because of this prefetching strategy, it is quite likely that the next several hundreds or thousand of chunk lookups will hit in the RAM chunk metadata cache.

RAM Chunk Metadata Cache Eviction Strategy: The container metadata cache in RAM follows a Least Recently Used (LRU) replacement policy. When a container is evicted from this cache, the chunk-ids of all the

chunks it contains are removed from the chunk metadata cache in RAM.

3.1 ChunkStash: Chunk Metadata Store on Flash

As a new contribution of this paper, we present the architecture of ChunkStash, the on-flash chunk metadata store, and the rationale behind some design choices. The design of ChunkStash is driven by the need to work around two types of operations that are not efficient on flash media, namely:

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

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

Given the above, the most efficient way to write flash is to simply use it as an append log, where an append operation involves one or more flash pages worth of data (current flash page size is typically 2KB or 4KB). This is the main constraint that drives the rest of our key-value store design. Flash has been used in a log-structured manner and its benefits reported in earlier work ([22, 41, 33, 15, 11]. We organize chunk metadata storage on flash into logical page units of 64KB which corresponds to the metadata for all chunks in a single container. (At 1024 chunks per container and 64 bytes per chunk-id and metadata, a container's worth of chunk metadata is 64KB in size.)

ChunkStash has the following main components, as shown in Figure 1:

RAM Chunk Metadata Write Buffer: This is a fixed-size data structure maintained in RAM that buffers chunk metadata information for the currently open container. The buffer is written to flash when the current container is sealed, i.e., the buffer accumulates 1024 chunk entries and reaches a size of 64KB. The RAM write buffer is sized to 2-3 times the flash page size so that chunk metadata writes can still go through when part of the buffer is being written to flash.

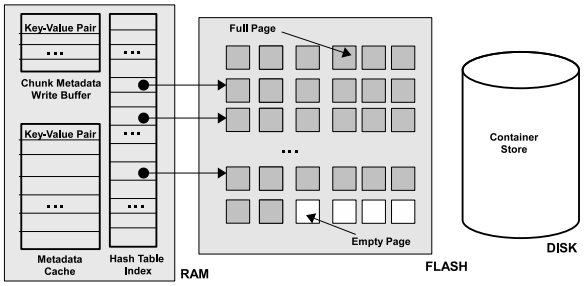


Figure 1: ChunkStash architectural overview.

RAM Hash Table (HT) Index: The index structure, for chunk metadata entries stored on flash, is maintained in RAM and is organized as a hash table with the design goal of one flash read per lookup. The index maintains pointers to the full (chunk-id, metadata) pairs stored on flash. Key features include resolving collisions using a variant of cuckoo hashing and storing compact key signatures in memory to tradeoff between RAM usage and false flash reads. We explain these aspects shortly.

On-Flash Store: The flash store provides persistent storage for chunk metadata and is organized as an append log. Chunk metadata is written (appended) to flash in units of a *logical page size* of 64KB, corresponding to the chunk metadata of a single container.

3.2 Hash Table Design for ChunkStash

We outline the salient aspects of the hash table design for ChunkStash.

Resolving hash collisions using cuckoo hashing: Hash function collisions on keys result in multiple keys mapping to the same hash table index slot – these need to be handled in any hashing scheme. Two common techniques for handling such collisions include *linear probing* and *chaining* [25]. Linear probing can increase lookup time arbitrarily due to long sequences of colliding slots. Chaining hash table entries in RAM, on the other hand, leads to increased memory usage, while chaining buckets of key-value pairs is not efficient for use with flash, since partially filled buckets will map to partially filled flash pages that need to be appended over time, which is not an efficient flash operation. Moreover, the latter will result in multiple flash page reads during key lookups and writes, which will reduce throughput.

ChunkStash structures the HT index as an array of slots and uses a variant of *cuckoo hashing* [35] to resolve collisions. Cuckoo hashing provides flexibility for each key to be in one of $n \geq 2$ candidate positions and for later inserted keys to relocate earlier inserted keys to any of their other candidate positions – this keeps the linear

probing chain sequence upper bounded at n. In fact, the value proposition of cuckoo hashing is in increasing hash table load factors while keeping lookup times bounded to a constant. A study [44] has shown that cuckoo hashing is much faster than chained hashing as hash table load factors increase. The name "cuckoo" is derived from the behavior of some species of the cuckoo bird – the cuckoo chick pushes other eggs or young out of the nest when it hatches, much like the hashing scheme kicks previously inserted items out of their location as needed.

In the variant of cuckoo hashing we use, we work with n random hash functions $h_1, h_2, \ldots, h_n$ that are used to obtain n *candidate positions* for a given key x. These candidate position indices for key x are obtained from the lower-order bit values of $h_1(x), h_2(x), \ldots, h_n(x)$ corresponding to a modulo operation. During insertion, the key is inserted in the first available candidate slot. When all slots for a given key x are occupied during insertion (say, by keys $y_1, y_2, \ldots, y_n$), room can be made for key x by relocating keys y_i in these occupied slots, since each key y_i has a choice of $(n-1)$ other locations to go to.

In the original cuckoo hashing scheme [35], a recursive strategy is used to relocate one of the keys y_i – in the worst case, this strategy could take many key relocations or get into an infinite loop, the probability for which can be shown to be very small and decreasing exponentially in n [35]. In our design, the system attempts a small number of key relocations after which it makes room by picking a key to move to an auxiliary linked list (or, hash table). In practice, by dimensioning the HT index for a certain load factor and by choosing a suitable value of n, such events can be made extremely rare, as we investigate in Section 4.4. Hence, the size of this auxiliary data structure is small. The viability of this approach has also been verified in [24], where the authors show, through analysis and simulations, that a very small constant-sized auxiliary space can dramatically reduce the insertion failure probabilities associated with cuckoo hashing. (That said, we also want to add that the design of ChunkStash is amenable to other methods of hash table collision resolution.)

The number of hash function computations during lookups can be reduced from the worst case value of n to 2 using the standard technique of *double hashing* from the hashing literature [25]. The basic idea is that two hash functions g_1 and g_2 can simulate more than two hash functions of the form $h_i(x) = g_1(x) + ig_2(x)$. In our case, i will range from 0 to $n-1$. Hence, the use of higher number of hash functions in cuckoo hashing does not incur additional hash function computation overheads but helps to achieve higher hash table load factors.

Reducing RAM usage per slot by storing compact key signatures: Traditional hash table designs store the respective key in each entry of the hash table index [25]. Depending on the application, the key size could range from few tens of bytes (e.g., 20-byte SHA-1 hash as in storage deduplication) to hundreds of bytes or more. Given that RAM size is limited (commonly in the order of few to several gigabytes in servers) and is more expensive than flash (per GB), if we store the full key in each entry of the RAM HT index, it may well become the bottleneck for the maximum number of entries on flash that can be indexed from RAM before flash storage capacity bounds kick in. On the other hand, if we do not store the key at all in the HT index, the search operation on the HT index would have to follow HT index pointers to flash to determine whether the key stored in that slot matches the search key – this would lead to many *false flash reads*, which are expensive, since flash access speeds are 2-3 orders of magnitude slower than that of RAM.

To address the goals of maximizing HT index capacity (number of entries) and minimizing false flash reads, we store a *compact key signature* (order of few bytes) in each entry of the HT index. This signature is derived from *both the key and the candidate position number that it is stored at*. In ChunkStash, when a key x is stored in its candidate position number i, the signature in the respective HT index slot is derived from the higher order bits of the hash value $h_i(x)$. During a search operation, when a key y is looked up in its candidate slot number j, the respective signature is computed from $h_j(y)$ and compared with the signature stored in that slot. Only if a match happens is the pointer to flash followed to check whether the full key matches. We investigate the percentage of false reads as a function of the compact signature size in Section 4.4.

Storing key-value pairs on flash: Chunk-id and metadata pairs are organized on flash in a log-structure in the order of the respective write operations coming into the system. The HT index contains pointers to (chunk-id, metadata) pairs stored on flash. We use a 4-byte pointer, which is a combination of a logical page pointer and a page offset. With 64-byte key-value pair sizes, this is sufficient to index 256GB of chunk metadata on flash – for an average chunk size of 8KB, this corresponds to a maximum (deduplicated) storage dataset size of about 33TB. (ChunkStash reserves the all-zero pointer to indicate an empty HT index slot.)

3.3 Putting It All Together

To understand the hierarchical relationship of the different storage areas in ChunkStash, it is helpful to understand the sequence of accesses during inline deduplica-

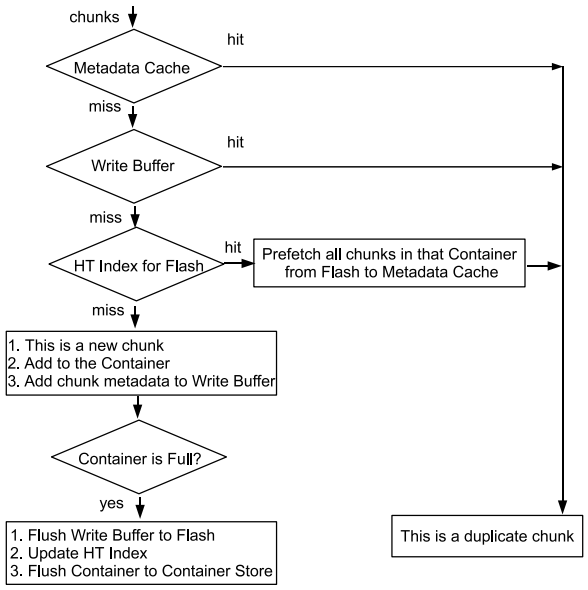


Figure 2: Flowchart of deduplication process in ChunkStash.

tion. A flowchart for this is provided in Figure 2. Recall that when a new chunk comes into the system, its SHA-1 hash is first looked up to determine if the chunk is a duplicate one. If not, the new chunk-id is inserted into the system.

A *chunk-id lookup* operation first looks up the RAM chunk metadata cache. Upon a miss there, it looks up the RAM chunk metadata write buffer. Upon a miss there, it searches the RAM HT Index in order to locate the chunk-id on flash. If the chunk-id is present on flash, its metadata, together with the metadata of all chunks in the respective container, is fetched into the RAM chunk metadata cache.

A *chunk-id insert* operation happens when the chunk coming into the system has not been seen earlier. This operation writes the chunk metadata into the RAM chunk metadata write buffer. The chunk itself is appended to the currently open container buffered in RAM. When the number of chunk entries in the RAM chunk metadata write buffer reaches the target of 1024 for the current container, the container is sealed and written to the container store on hard disk, and its chunk metadata entries are written to flash and inserted into the RAM HT index.

3.4 RAM and Flash Capacity Considerations

The indexing scheme in ChunkStash is designed to use a small number of bytes in RAM per key-value pair so as to maximize the amount of indexable storage on flash for a given RAM usage size. The RAM HT index capacity determines the number of chunk-ids stored on flash whose metadata can be accessed with one flash read. The RAM size for the HT index can be determined with application requirements in mind. With a 2-byte compact key signature and 4-byte flash pointer per entry, the RAM usage in ChunkStash is 6 bytes per entry as shown in Figure 3. For a given average chunk size, this determines the relationship among the following quantities – RAM and flash usage per storage dataset and associated storage dataset size.

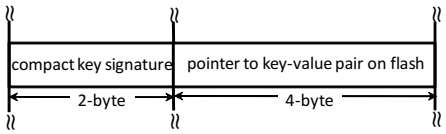


Figure 3: RAM HT Index entry and example sizes in ChunkStash. (The all-zero pointer is reserved to indicate an empty HT index slot.)

For example, a typical RAM usage of 4GB per machine for the HT index accommodates a maximum of about 716 million chunk-id entries. At an average of 8KB size per data chunk, this corresponds to about 6TB of deduplicated data, for which the chunk metadata occupies about 45GB on flash. This flash usage is well within the capacity range of SSDs shipping in the market today (from 64GB to 640GB). When there are multiple such SSDs attached to the same machine, additional RAM is needed to fully utilize their capacity for holding chunk metadata. Moreover, RAM usage by the HT index in ChunkStash can be further reduced using techniques discussed in Section 3.5.

To reap the full performance benefit of ChunkStash for speeding up inline deduplication, it is necessary for the entire chunk metadata for the (current) backup dataset to fit in flash. Otherwise, when space on flash runs out, the append log will need to be *recycled* and written from the beginning. When a page on the flash log is rewritten, the earlier one will need to be evicted and the metadata contained therein written out to a hard disk based index. Moreover, during the chunk-id lookup process, if the chunk is not found in flash, it will need to be looked up in the index on hard disk. Thus, both the chunk-id insert and lookup pathways would suffer from the same bottlenecks of disk index based systems that we sought to eliminate in the first place.

ChunkStash uses flash memory to store chunk metadata and index it from RAM. It provides flexibility for flash to serve, or not to serve, as a permanent abode for chunk metadata for a given storage location. This decision can be driven by cost considerations, for example, because of the large gap in cost between flash memory and hard disk. When flash is not the permanent abode for

chunk metadata for a given storage location, the chunk metadata log on flash can be written to hard disk in one large sequential write (single disk I/O) at the end of the backup process. At the beginning of the next full backup for this storage location, the chunk metadata log can be loaded back into flash from hard disk in one large sequential read (single disk I/O) and the containing chunks can be indexed in RAM HT index. This mode of operation amortizes the storage cost of metadata on flash across many backup datasets.

3.5 Reducing ChunkStash RAM Usage

The largest portion of RAM usage in ChunkStash comes from the HT index. This usage can be reduced by indexing in RAM only a small fraction of the chunks in each container (instead of the whole container). Flash will continue to hold metadata for *all* chunks in all containers, not just the ones indexed in RAM; hence when a chunk in the incoming data stream matches an indexed chunk, metadata for *all* chunks in that container will be prefetched in RAM. We use an uniform chunk sampling strategy, i.e., we index every i-th chunk in every container which gives a sampling rate of $1/i$.

Because only a subset of chunks stored in the system are indexed in the RAM HT index, detection of duplicate chunks will not be completely accurate, i.e., some incoming chunks that are not found in the RAM HT index may, in fact, have appeared earlier and are already stored in the system. This will lead to some loss in deduplication quality, and hence, some amount of duplicate data chunks will be stored in the system. In Section 4.6, we study the impact of this on deduplication quality (and backup throughput). We find that the loss in deduplication quality is marginal when about 1% of the chunks in each container are indexed and becomes negligibly small when about 10% of the chunks are indexed. The corresponding RAM usage reductions for the HT index are appreciable at 99% and 90% respectively. Hence, indexing chunk subsets in ChunkStash provides a powerful knob for reducing RAM usage with only marginal loss in deduplication quality.

In an earlier approach for reducing RAM usage requirements of inline deduplication systems, the method of sparse indexing [30] chops the incoming data into multiple megabyte segments, samples chunks *at random* within a segment (based on the most significant bits of the SHA-1 hash matching a pattern, e.g., all 0s), and uses these samples to find few segments seen in the recent past that share many chunks. In contrast, our sampling method is deterministic and samples chunks at uniform intervals in each container for indexing. Moreover, we are able to match incoming chunks with sampled chunks in *all* containers stored in the system, not just those seen in the recent past. In our evaluations in Section 4.6, we show that our uniform sampling strategy gives better deduplication quality than random sampling (for the same sampling rate).

4 Evaluation

We evaluate the backup throughput performance of a ChunkStash based inline deduplication system and compare it with our implementation of a disk index based system as in [42]. We use three enterprise datasets and two full backups for our evaluations.

4.1 C# Implementation

We have prototyped ChunkStash in approximately 8000 lines of C# code. MurmurHash [5] is used to realize the the hash functions used in our variant of cuckoo hashing to compute hash table indices and compact signatures for keys; two different seeds are used to generate two different hash functions in this family for use with the double hashing based simulation of n hash functions, as described in Section 3.2. In our implementation, writes to the on-disk container store are performed in a non-blocking manner using a small pool of file writer worker threads. The metadata store on flash is maintained as a log file in the file system and is created/opened in non-buffered mode so that *there are no buffering/caching/prefetching effects in RAM from within the operating system*.

4.2 Comparison with Hard Disk Index based Inline Deduplication

We compare ChunkStash, our flash index based inline deduplication system, with a hard disk index based system as in Zhu et al. [42]. The index used in [42] appears to be proprietary and no details are provided in the paper. Hence, for purposes of comparative evaluation, we have built a hard disk index based system incorporating the ideas in [42] with the hard disk index implemented by BerkeleyDB [1], an embedded key-value database that is widely used as a comparison benchmark for its good performance. For comparison with the latter system, we also include a "hard disk replacement with SSD" for the index storage, so as to bring out the performance gain of ChunkStash in not only using flash for chunk metadata storage but also in its use of flash aware algorithms.

BerkeleyDB does not use flash aware algorithms but we used the parameter settings recommended in [3] to improve its performance with flash. We use BerkeleyDB in its non-transactional concurrent data store mode that supports a single writer and multiple readers [40]. This mode does *not* support a transactional data store with the

Trace	Size (GB)	Total Chunks	#Full Backups
Dataset 1	8GB	1.1 million	2
Dataset 2	32GB	4.1 million	2
Dataset 3	126GB	15.4 million	2

Table 1: Properties of the three traces used in the performance evaluation of ChunkStash. The average chunk size is 8KB.

ACID properties, hence provides a fair comparison with ChunkStash . BerkeleyDB provides a choice of B-tree and hash table data structures for building indexes – we use the hash table version which we found to run faster. We use the C++ implementation of BerkeleyDB with C# API wrappers [2].

4.3 Evaluation Platform and Datasets

We use a standard server configuration to evaluate the inline deduplication performance of ChunkStash and compare it with the disk index based system that uses BerkeleyDB. The server runs Windows Server 2008 R2 and uses an Intel Core 2 Duo E6850 3GHz CPU, 4GB RAM, and fusionIO 160GB flash drive [4] attached over PCIe interface. Containers are written to a RAID4 system using five 500GB 7200rpm hard disks. A separate hard disk is used for storing disk based indexes. For the fusionIO drive, *write buffering inside the device is disabled* and cannot be turned on through operating system settings. The hard drives used support write buffering inside the device by default and this setting was left on. This clearly gives some advantage to the hard disks for the evaluations but makes our comparisons of flash against hard disk more conservative.

To obtain traces from backup datasets, we have built a storage deduplication analysis tool that can crawl a root directory, generate the sequence of chunk hashes for a given average chunk size, and compute the number of deduplicated chunks and storage bytes. The enterprise data backup traces we use for evaluations in this paper were obtained by our storage deduplication analysis tool using 8KB (average) chunk sizes (this is also the chunk size used in [30]). We obtained two full backups for three different storage locations, indicated as Datasets 1, 2, and 3 in Table 1. The number of chunks in each dataset (for each full backup) are about 1 million, 4 million, and 15 million respectively.

We compare the throughput (MB/sec processed from the input data stream) on the three traces described in Table 1 for the following four inline deduplication systems:

- Disk based index (BerkeleyDB) and RAM bloom filter [42] (***Zhu08-BDB-HDD***),
- Zhu08-BDB system with SSD replacement for BerkeleyDB index storage (***Zhu08-BDB-SSD***),
- Flash based index using ChunkStash (***ChunkStash-SSD***),
- ChunkStash with the SSD replaced by hard disk (***ChunkStash-HDD***).

Some of the design decisions in ChunkStash also work well when the underlying storage is hard disk and not flash (e.g., log structured data organization and sequential writes). Hence, we have included Chunkstash running on hard disk as a comparison point so as to bring out the impact of log structured organization for a store on hard disk for the storage deduplication application. (Note that BerkeleyDB does not use a log structured storage organization.)

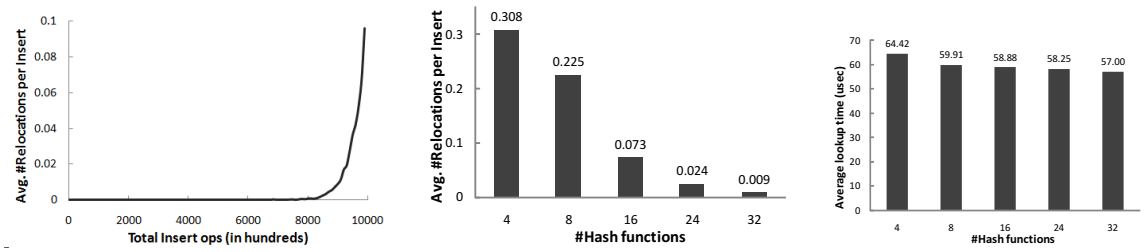
All four systems use RAM prefetching techniques for the chunk index as described in [42]. When a chunk hash lookup misses in RAM (but hits the index on hard disk or flash), metadata for all chunks in the associated container are prefetched into RAM. The RAM chunk metadata cache size for holding chunk metadata is fixed at 20 containers, which corresponds to a total of 20,480 chunks. In order to implement the prefetching of container metadata in a single disk I/O for Zhu08-BDB, we maintain, in addition to the BerkeleyDB store, an auxiliary sequential log of chunk metadata that is appended with container metadata whenever a new container is written to disk.

Note that unlike in [42], our evaluation platform is not a production quality storage deduplication system but rather a research prototype. Hence, our throughput numbers should be interpreted in a relative sense among the four systems above, and not used for absolute comparison with other storage deduplication systems.

4.4 Tuning Hash Table Parameters

Before we make performance comparisons of ChunkStash with the disk index based system, we need to tune two parameters in our hash table design from Section 3.2, namely, (a) number of hash functions used in our variant of cuckoo hashing, and (b) size of compact key signature. These affect the throughput of read key and write key operations in different ways that we discuss shortly. For this set of experiments, we use one of the storage deduplication traces in a modified way so as to study the performance of hash table insert and lookup operations separately. We pre-process the trace to extract a set of 1 million unique keys, then insert all the key-value pairs, and then read all these key-value pairs in random order.

As has been explained earlier, the value proposition of cuckoo hashing is in accommodating higher hash table load factors without increasing lookup time. It has been



(a) (b) (c)

Figure 4: Tuning hash table parameters in ChunkStash: (a) Average number of relocations per insert as keys are inserted into hash table; (b) Average number of relocations per insert, and (c) Average lookup time (μsec), vs. number of hash functions (n), averaged between 75%-90% load factors.

shown mathematically that with 3 or more hash functions and with load factors up to 91%, insertion operations succeed in expected constant time [35]. With this prior evidence in hand, we target a maximum load factor of 90% for our cuckoo hashing implementation. Hence, for the dedup trace used in this section with 1 million keys, we fix the number of hash table slots to 1.1 million.

Number of Hash Functions. When the number of hash functions n is small, the performance of insert operations can be expected to degrade in two ways as the hash table loads up. First, an insert operation will find all its n slots occupied by other keys and the number of cascaded key relocations required to complete this insertion will be high. Since each key relocation involves a flash read (to read the full key from flash and compute its candidate positions), the insert operation will take more time to complete. Second, with an upper bound on the number of allowed key relocations (which we set to 100 for this set of experiments), the insert operation could lead to a key being moved to the auxiliary linked list – this increases the RAM space usage of the linked list as well as its average search time. On the other hand, as the number of hash functions increase, lookup times will increase because of increasing number of hash table positions searched. However, the latter undesirable effect is not expected to be as degrading as those for inserts, since a lookup in memory takes orders of magnitude less time than a flash read. We study these effects to determine a suitable number of hash functions for our design. (Note that because of our use of double hashing, the number of hash function computations per lookup does not increase with n.)

In Figure 4(a), we plot the average number of key relocations (hence, flash reads) per insert operation as keys are inserted into the hash table (for $n = 24$ hash functions). We see that the performance of insert operations degrades as the hash table loads up, as expected because of the impact of the above effect. Accordingly, for the following plots in this section, we present average numbers between 75% and 90% load factors as the insert operations are performed.

In Figure 4(b), we plot the average number of key relocations per insert operation as the number of hash functions n is varied, $n = 4, 8, 16, 24, 32$. Beyond $n = 16$ hash functions, the hash table incurs less than 0.1 key relocations (hence, flash reads) per insert operation. Figure 4(c) shows that there is no appreciable increase in average lookup time as the number of hash functions increase. (The slight decrease in average lookup time with increasing number of hash functions can be attributed to faster search times in the auxiliary linked list, whose size decreases as number of hash functions increases.)

Based on these effects, we choose $n = 24$ hash functions in the RAM HT Index for our ChunkStash implementation. Note that during a key lookup, all n hash values on the key need not be computed, since the lookup stops at the candidate position number the key is found in. Moreover, because of the use of double hashing, at most two hash function computations are incurred even when all candidate positions are searched for a key. We want to add that using fewer hash functions may be acceptable depending on overall performance requirements, but we do not recommend a number below $n = 8$. Also, with $n = 24$ hash functions, we observed that it is sufficient to set the maximum number of allowed key relocations (during inserts) to 5-10 to keep the number of inserts that go to the linked list very small.

Compact Key Signature Size. As explained in Section 3.1, we store compact key signatures in the HT index (instead of full keys) to reduce RAM usage. However, shorter signatures lead to more *false flash reads* during lookups (i.e., when a pointer into flash is followed because the signature in HT index slot matches, but the full key on flash does not match with the key being looked up). Since flash reads are expensive compared to RAM reads, the design should strike a balance between reducing false flash reads and RAM usage. We observe that the fraction of false reads is about 0.01% when the number of signature bytes is fixed at 2, and we use this signature

Trace	RAM Hit Rate	
	1st Full Backup	2nd Full Backup
Dataset 1	20%	97%
Dataset 2	2%	88%
Dataset 3	23%	80%

Table 2: RAM hit rates for chunk hash lookups on the three traces for each full backup.

size in our implementation. Even a 1-byte signature size may be acceptable given that only 0.6% of the flash reads are false in that case.

4.5 Backup Throughput

We ran the chunk traces from the the three datasets outlined in Table 1 on ChunkStash and our implementation of the system in [42] using BerkeleyDB as the chunk index on either hard disk or flash. We log the backup throughput (MB of data backed up per second) at a period of every 10,000 input chunks during each run and then take the overall average over a run to obtain throughput numbers shown in Figures 5, 6, and 7.

ChunkStash achieves average throughputs of about 190 MB/sec to 265 MB/sec on the first full backups of the three datasets. The throughputs are about 60%-140% more for the second full backup compared to the first full backup for the datasets – this reflects the effect of prefetching chunk metadata to exploit sequential predictability of chunk lookups during second full backup. The speedup of ChunkStash over Zhu08-BDB-HDD is about 30x-60x for the first full backup and 7x-40x for the second full backup. Compared to the Zhu08-BDB-SSD in which the hard disk is replaced by SSD for index storage, the speedup of ChunkStash is about 3x-4x for the first full backup and about 2x-4x for the second full backup. The latter reflects the relative speedup of ChunkStash due to the use of flash-aware data structures and algorithms over BerkeleyDB which is not optimized for flash device properties.

We also run ChunkStash on hard disk (instead of flash) to bring out the performance impact of a log-structured organization on hard disk. Sequential writes of container metadata to the end of the metadata log is a good design for and benefits hard disks also. On the other hand, lookups in the log from the in-memory index may involve random reads in the log which are expensive on hard disks due to seeks – however, container metadata prefetching helps to reduce the number of such random reads to the log. We observe that the throughput of ChunkStash-SSD is more than that of ChunkStash-HDD by about 50%-80% for the first full backup and by about 10%-20% for the second full backup for the three datasets.

In Table 2, we show the RAM hit rates for chunk hash

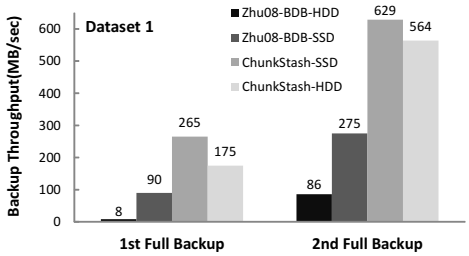


Figure 5: Dataset 1: Comparative throughput (backup MB/sec) evaluation of ChunkStash and BerkeleyDB based indexes for inline storage deduplication.

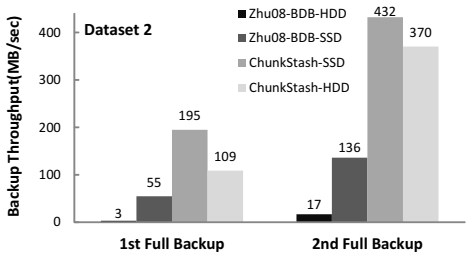


Figure 6: Dataset 2: Comparative throughput (backup MB/sec) evaluation of ChunkStash and BerkeleyDB based indexes for inline storage deduplication.

lookups on the three traces for each full backup. The RAM hit rate for the first full backup is indicative of the redundancy (duplicate chunks) *within* the dataset – this is higher for Datasets 1 and 3 and is responsible for their higher backup throughputs. The RAM hit rate for the second full backup is indicative of its similarity with the first full backup – this manifests itself during the second full backup through fewer containers written and through sequential predictability of chunk hash lookups (which is exploited by the RAM prefetching strategy).

When compared to ChunkStash, the relatively worse performance of a BerkeleyDB based chunk metadata index can be attributed to the increased number of disk I/Os (random reads and writes). We measured the number of disk I/Os for Zhu08-BDB and ChunkStash systems using Windows Performance Analysis Tools (`xperf`) [8]. In Figure 8, we observe that the number of read I/Os in BerkeleyDB is 3x-7x that of ChunkStash and the number of write I/Os is about 600-1000x that of ChunkStash. Moreover, these writes I/Os in BerkeleyDB are all random I/Os, while ChunkStash is designed to use only sequential writes (appends) to the chunk metadata log. Because there is no locality in the key space in an application like storage deduplication, container metadata writes to a BerkeleyDB based index lead to in-place updates (random writes) to many different pages (on disk or flash) and appears to be one of the main reasons for its worse performance.

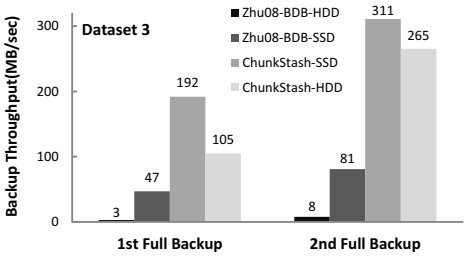


Figure 7: Dataset 3: Comparative throughput (backup MB/sec) evaluation of ChunkStash and BerkeleyDB based indexes for inline storage deduplication.

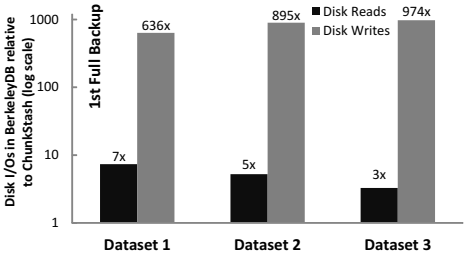


Figure 8: Disk I/Os (reads and writes) in BerkeleyDB relative to ChunkStash on the first full backup of the three datasets. (Note that the y-axis is log scale.)

4.6 Impact of Indexing Chunk Subsets

In Section 3.5, we saw that RAM usage in ChunkStash can be reduced by indexing a small fraction of the chunks in each container. In this section, we study the impact of this on deduplication quality and backup throughput. Because only a subset of the chunks are indexed in the RAM HT index, detection of duplicate chunks will not be completely accurate, i.e., some incoming chunks that are not found in the RAM HT index may have appeared earlier and are already stored in the system. Hence, some amount of duplicate data chunks will be stored in the system. We compare two chunk sampling strategies – uniform chunk sampling strategy (i.e., index every i-th chunk) and random sampling (based on the most significant bits of the chunk SHA-1 hash matching a pattern, e.g., all 0s), the latter being used as part of a sparse indexing scheme in [30].

In Figure 9, we plot the fraction of chunks that are declared as new by the system during the second full backup of Dataset 2 as a percentage of the total number of chunks in the second full backup. The lower the value of this fraction, the better is the deduplication quality (the baseline for comparison being the case when all chunks are indexed). The fraction of chunks indexed in each container is varied as $1.563\% = 1/64$, $6.25\% = 1/16$, $12.5\% = 1/8$, and 100% (when all chunks are indexed). We choose sampling rates that are reciprocals of powers of 2 because those are the types of sampling rates

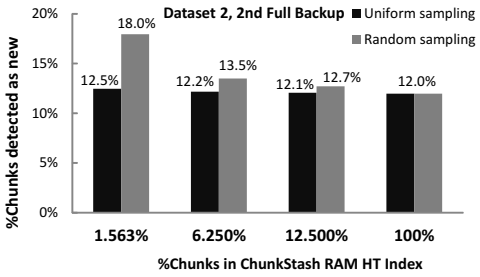


Figure 9: Dataset 2: Number of chunks detected as new as a fraction of the total number of chunks (indicating deduplication quality) vs. fraction of chunks indexed in ChunkStash RAM HT index in second full backup. (When 100% of the chunks are indexed, all duplicate chunks are detected accurately.) The x-axis fractions correspond to sampling rates of $1/64$, $1/16$, and $1/8$. For a sampling rate of $1/2^n$, uniform sampling indexes every 2^n-th chunk in a container, whereas random sampling indexes chunks with first n bits of SHA-1 hash are all 0s.

possible in the random sampling scheme – when n most significant bits of the SHA-1 hash are matched to be all 0s, the sampling rate is $1/2^n$.

We observe that when 1.563% of the chunks are indexed, the uniform chunk sampling strategy results in only a 0.5% increase in the number of chunks detected as new (as a fraction of the whole dataset). This loss in deduplication quality could be viewed as an acceptable tradeoff in exchange for a 98.437% reduction in RAM usage of ChunkStash HT index. When 12.5% of the chunks are indexed, the loss in deduplication quality is almost negligible at 0.1% (as a fraction of the whole dataset), but the reduction in RAM usage of ChunkStash HT index is still substantial at 87.5%. Hence, indexing chunk subsets provides a powerful knob for reducing RAM usage in ChunkStash with only marginal loss in deduplication quality.

We also note that the loss in deduplication quality with random sampling is worse than that with uniform sampling, especially at lower sampling rates. An intuitive explanation for this is that uniform sampling gives better coverage of the input stream at regular intervals of number of chunks (hence, data size intervals), whereas random sampling (based on some bits in the chunk SHA-1 hash matching a pattern) could lead to large gaps between two successive chunk samples in the input stream.

An interesting side-effect of indexing chunk subsets is the increase in backup throughput – this is shown in Figure 10 for the first and second full backups for Dataset 2. The effects on first and second full backups can be explained separately. When a fraction of the chunks are indexed, chunk hash keys are inserted into the RAM HT index at a lower rate during the first full backup, hence the throughput increases. (Note, however, that writes to

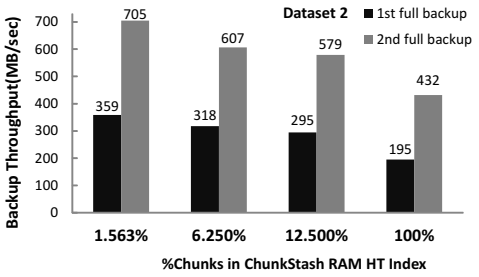


Figure 10: Dataset 2: Backup throughput (MB/sec) vs. fraction of chunks indexed in ChunkStash RAM HT index in first and second full backups.

the metadata log on flash are still about the same, but possibly slightly higher due to less accurate detection of duplicate chunks within the dataset.) During the second full backup, fewer chunks from the incoming stream are found in the ChunkStash RAM HT index, hence the number of flash reads during the backup process are reduced, leading to higher throughput. Both of these benefits drop gradually as more chunks are indexed but still remain substantial at a sampling rate of 12.5% – with the loss in deduplication quality being negligible at this point, the tradeoff is more of a win-win situation than a compromise involving the three parameters of RAM usage (low), deduplication throughput (high), and loss is deduplication quality (negligible).

4.7 Flash Memory Cost Considerations

Because flash memory is more expensive per GB than hard disk, we undertake a performance/dollar analysis in an effort to mitigate cost concerns about a flash-assisted inline deduplication system like ChunkStash. In our system, we use 8KB (average) chunk sizes and store them compressed on hard disk – with an average compression ratio of 2:1 (which we verified for our datasets), the space occupied by a data chunk on hard disk is about 4KB. With chunk metadata size of 64 bytes, the metadata portion is about $64/(4 * 1024) = 1/64$ fraction of the space occupied by chunk data on hard disk. With flash being currently 10 times more expensive per GB than hard disk, the cost of metadata storage on flash is about $10/64 = 16\%$ that of data storage on HDD. Hence, the overall increase in storage cost is about 1.16x.

Using a ballpark improvement of 25x in backup throughput for ChunkStash over disk based indexes for inline deduplication (taken from our evaluations reported in Section 4.5), *the gain in performance/dollar for ChunkStash over disk based indexes is about* $25/1.16 = 22x$. We believe that this justifies the additional capital investment in flash for inline deduplication systems that are hard-pressed to meet short backup window deadlines.

Moreover, the metadata storage cost on flash can be amortized across many backup datasets by storing a dataset's chunk metadata on hard disk and loading to flash just before the start of the backup process for the respective dataset – this reduces the flash memory investment in the system and makes the performance-cost economics even more compelling.

5 Related Work

We review related work that falls into two categories, namely, storage deduplication and key-value store on flash. The use of flash memory to speed up inline deduplication is a unique contribution of our work – their is no prior research that overlaps both of these areas. We also make new contributions in the design of ChunkStash, the chunk metadata store on flash which can be used as a key-value store for other applications as well.

5.1 Storage Deduplication

Zhu et al.'s work [42] is among the earliest research in the inline storage deduplication area and provides a nice description of the innovations in Data Domain's production storage deduplication system. They present two techniques that aim to reduce lookups on the disk-based chunk index. First, a bloom filter [13] is used to track the chunks seem by the system so that disk lookups are not made for non-existing chunks. Second, upon a chunk lookup miss in RAM, portions of the disk-based chunk index (corresponding to all chunks in the associated container) are prefetched to RAM. The first technique is effective for new data (e.g., first full backup) while the second technique is effective for little or moderately changed data (e.g., subsequent full backups). Their system provides perfect deduplication quality. Our work aims to reduce the penalty of index lookup misses in RAM that go to hard disk by orders of magnitude by designing a flash-based index for storing chunk metadata.

Lillibridge et al. [30] use the technique of sparse indexing to reduce the in-memory index size for chunks in the system at the cost of sacrificing deduplication quality. The system chunks the data into multiple megabyte segments, which are then lightly sampled (at random based on the chunk SHA-1 hash matching a pattern), and the samples are used to find a few segments seen in the recent past that share many chunks. Obtaining good deduplication quality depends on the chunk locality property of the dataset – whether duplicate chunks tend to appear again together with the same chunks. When little or no chunk locality is present, the authors recommend an approach based on *file similarity* [12] that achieves significantly better deduplication quality. In our work, the memory usage of ChunkStash can be reduced by indexing only a subset of the chunk metadata on flash (using an uniform

sampling strategy, which we found gives better deduplication quality than random sampling).

DEDE [16] is a decentralized deduplication system designed for SAN clustered file systems that supports a virtualization environment via a shared storage substrate. Each host maintains a write-log that contains the hashes of the blocks it has written. Periodically, each host queries and updates a shared index for the hashes in its own write-log to identify and reclaim storage for duplicate blocks. Unlike inline deduplication systems, the deduplication process is done out-of-band so as to minimize its impact on file system performance. In this paper, we focus on inline storage deduplication systems.

HYDRAstor [17] discusses architecture and implementation of a commercial secondary storage system, which is content addressable and implements a global data deduplication policy. Recently, a new file system, called HydraFS [39], has been designed for HYDRAstor. In order to reduce the disk accesses, HYDRAstor uses bloom filter [13] in RAM. In contrast, we aim to eliminate disk seek/access (and miss) overheads by using a flash-based chunk metadata store.

Deduplication systems differ in the granularity at which they detect duplicate data. EMC's Centera [18] uses file level duplication, LBFS [31] uses variable-sized data chunks obtained using Rabin fingerprinting, and Venti [36] uses individual fixed size disk blocks. Among content-dependent data chunking methods, Two-Threshold Two-Divisor (TTTD) [19] and bimodal chunking algorithm [27] produce variable-sized chunks.

5.2 Key-Value Store on Flash

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

MicroHash [41] designs a memory-constrained index structure for flash-based sensor devices with the goal of optimizing energy usage and minimizing memory footprint. This work does not target low latency operations as a design goal – in fact, a lookup operation may need to follow chains of index pages on flash to locate a key, hence involving multiple flash reads.

FlashDB [33] is a self-tuning B^+-tree based index that dynamically adapts to the mix of reads and writes in the workload. Like MicroHash, this design also targets memory and energy constrained sensor network devices. Because a B^+-tree needs to maintain partially filled leaf-level buckets on flash, appending of new keys to these buckets involves random writes, which is not an efficient flash operation. Hence, an adaptive mechanism is also provided to switch between disk and log-based modes. The system leverages the fact that key values in sensor applications have a small range and that at any given time, a small number of these leaf-level buckets are active. Minimizing latency is not an explicit design goal.

The benefits of using flash in a log-like manner have been exploited in FlashLogging [15] for synchronous logging. This system uses multiple inexpensive USB drives and achieves performance comparable to flash SSDs but with much lower price. Flashlogging assumes sequential workloads.

Gordon [14] uses low power processors and flash memory to build fast power-efficient clusters for data-intensive applications. It uses a flash translation layer design tailored to data-intensive workloads. In contrast, ChunkStash builds a persistent key-value store using existing flash devices (and their FTLs) with throughput maximization as the main design goal.

FAWN [11] uses an array of embedded processors equipped with small amounts of flash to build a power-efficient cluster architecture for data-intensive computing. Like ChunkStash, FAWN also uses an in-memory hash table to index key-value pairs on flash. The differentiating aspects of ChunkStash include its adaptation for the specific server-class application of inline storage deduplication and in its use of a specialized in-memory hash table structure with cuckoo hashing to achieve high hash table load factors (while keeping lookup times bounded) and reduce RAM usage.

BufferHash [10] builds a content addressable memory (CAM) system using flash storage for networking applications like WAN optimizers. It buffers key-value pairs in RAM, organized as a hash table, and flushes the hash table to flash when the buffer is full. Past copies of hash tables on flash are searched using a time series of Bloom filters maintained in RAM and searching keys on a given copy involve multiple flash reads. Moreover, the storage of key-value pairs in hash tables on flash wastes space on flash, since hash table load factors need to be well below 100% to keep lookup times bounded. In contrast, ChunkStash is designed to access any key using one flash read, leveraging cuckoo hashing and compact key signatures to minimize RAM usage of a customized in-memory hash table index.

6 Conclusion

We designed ChunkStash to be used as a high throughput persistent key-value storage layer for chunk metadata for inline storage deduplication systems. To this end, we

incorporated flash aware data structures and algorithms into ChunkStash to get the maximum performance benefit from using SSDs. We used enterprise backup datasets to drive and evaluate the design of ChunkStash . Our evaluations on the metric of backup throughput (MB/sec) show that ChunkStash outperforms (i) a hard disk index based inline deduplication system by 7x-60x, and (ii) SSD index (hard disk replacement but flash unaware) based inline deduplication system by 2x-4x. Building on the base design, we also show that the RAM usage of ChunkStash can be reduced by 90-99% with only a marginal loss in deduplication quality.

References

[1] BerkeleyDB. `http://www.oracle.com/technology/products/berkeley-db/index.html`.

[2] BerkeleyDB for .NET. `http://sourceforge.net/projects/libdb-dotnet/`.

[3] BerkeleyDB Memory-only or Flash configurations. `http://www.oracle.com/technology/documentation/berkeley-db/db/ref/program/ram.html`.

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

[5] MurmurHash Function. `http://en.wikipedia.org/wiki/MurmurHash`.

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

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

[8] Windows Performance Analysis Tools (xperf). `http://msdn.microsoft.com/en-us/performance/cc825801.aspx`.

[9] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX*, 2008.

[10] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. In *NSDI*, 2010.

[11] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*, Oct. 2009.

[12] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In *MASCOTS*, 2009.

[13] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*, 2002.

[14] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using Flash Memory to Build Fast, Power-Efficient Clusters for Data-Intensive Applications. In *ASPLOS*, 2009.

[15] S. Chen. Flashlogging: Exploiting flash devices for synchronous logging performance. In *SIGMOD*, 2009.

[16] A. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized Deduplication in SAN Cluster File Systems. In *USENIX*, 2009.

[17] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, , and M. Welnicki. HYDRAstor: a Scalable Secondary Storage. In *FAST*, 2009.

[18] EMC Corporation. EMC Centera: Content Addresses Storage System, Data Sheet, April 2002.

[19] K. Eshghi. A framework for analyzing and improving content-based chunking algorithms. *HP Labs Technical Report HPL-2005-30 (R.1)*, 2005.

[20] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. In *ACM Computing Surveys*, volume 37, 2005.

[21] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *ASPLOS*, 2009.

[22] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-Memory Based File System. In *USENIX*, 1995.

[23] H. Kim and S. Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *FAST*, 2008.

[24] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2009.

[25] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching (Volume 3)*. Addison-Wesley, Reading, MA, 1998.

[26] I. Koltsidas and S. Viglas. Flashing Up the Storage Layer. In *VLDB*, 2008.

[27] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal Content Defined Chunking for Backup Streams. In *FAST*, 2010.

[28] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song. A Log Buffer-Based Flash Translation Layer Using Fully-Associate Sector Translation. In *ACM TECS*, volume 6, 2007.

[29] A. Leventhal. Flash Storage Memory. *Communications of the ACM*, 51(7):47 – 51, July 2008.

[30] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *FAST*, 2009.

[31] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, 2001.

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

[33] S. Nath and A. Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *IPSN*, 2007.

[34] National Institute of Standards and Technology, FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce, 1995.

[35] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[36] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Data Storage. In *FAST*, 2002.

[37] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *FAST*, 2002.

[38] M. O. Rabin. Fingerprinting by Random Polynomials. *Harvard University Technical Report TR-15-81*, 1981.

[39] C. Ungureanu, B. Atkin, A. Aranya, S. R. Salil Gokhale, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: a High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In *FAST*, 2010.

[40] H. Yadava. *The Berkeley DB Book*. Apress, 2007.

[41] D. Zeinalipour-yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: An Efficient Index Structure for Flash-based Sensor Devices. In *FAST*, 2005.

[42] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *FAST*, 2008.

[43] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337 – 343, May 1977.

[44] M. Zukowski, S. Heman, and P. Boncz. Architecture-Conscious Hashing. In *DAMON*, 2006.