Lazy Exact Deduplication

Jingwei Ma, Rebecca J. Stones, Yuxiang Ma, Jingui Wang, Junjie Ren, Gang Wang*, Xiaoguang Liu*
College of Computer and Control Engineering, Nankai University, Tianjin, China
Email: {mjwtom, rebecca.stones82, mayuxiang, wangjingui, renjunjie, wgzwp, liuxg}@nbjl.nankai.edu.cn

***Abstract*—During data deduplication, on-disk fingerprint lookups lead to high disk traffic, resulting in a bottleneck. In this paper, we propose a "lazy" data deduplication method which buffers incoming fingerprints and performs on-disk lookups in batches, aiming to reduce the disk bottleneck. In deduplication in general, prefetching is used to improve the cache hit rate by exploiting locality within the incoming fingerprint stream. For lazy deduplication, we design a buffering strategy that preserves locality in order to similarly facilitate prefetching. Experimental results indicate that the lazy method improves fingerprint identification performance by over 50% compared with an "eager" method with the same data layout**

I. Introduction

Data deduplication is a key technology in data backup. By eliminating redundant data blocks and replacing them with references to the corresponding unique ones, we can reduce storage requirements. Many companies have backup systems utilizing deduplication [1]–[5]. In a typical deduplication procedure, a data stream is segmented into chunks and a cryptographic hash (e.g. MD5, SHA1, SHA256) is calculated as the *fingerprint* of each chunk. The system determines whether a data chunk is redundant by comparing fingerprints instead of whole chunks.

In a usual exact deduplication implementation, when encountering an uncached fingerprint, the system immediately reads the disk to search for the fingerprint. Each on-disk lookup searches for only a single fingerprint, and if discovered, prefectching is triggered. We will refer to this method as the *eager* method.

In this paper, we propose a *lazy* deduplication method, which buffers fingerprints in memory organized into hash buckets. When the number of fingerprints in a hash bucket reaches a user-defined threshold T, the system reads the disk and searches for those fingerprints together. Importantly, the lazy method performs a single on-disk lookup for T fingerprints. This reduces the disk access time for on-disk fingerprint lookups. Though the cache lookup strategy proposed in this paper works better in terms of backup flow, the lazy method is suitable for both primary workloads and backup workloads.

II. Background and related work

Data deduplication is used to reduce both storage requirements and network transfers and is both computationally and I/O intensive. It is computationally intensive due to chunking, fingerprint calculation, compression, and so on.

For large-scale deduplication systems, as the main memory is not large enough to hold all the fingerprints, most fingerprints are stored on disk. Consequently, data deduplication is also disk I/O intensive, resulting in a *disk bottleneck*, which can significantly affect throughput. The disk bottleneck has an increasing effect as the data size (and hence the number of fingerprints) grows, whereas calculation time usually remains stable. As a result, most previous work focused on eliminating the disk bottleneck in deduplication. While the disk bottleneck can be reduced by 99% in some exact deduplication systems [4], it remains a bottleneck. Our goal is to further reduce this component by combining several fingerprint lookups into a single disk access.

Deduplication systems take advantage of locality properties in data streams to reduce disk accesses by using an in-memory cache [4], [6]–[11]. When a fingerprint is found on disk, *prefetching* is invoked, whereby adjacent fingerprints stored on disk are transferred to the cache. As fingerprints frequently arrive in the same order as they arrived previously (and therefore the same order as they are stored on the disk), this prefetching strategy leads to a high cache hit rate which significantly reduces disk access time.

There are many other optimized techniques used in deduplication systems, such as delta compression [12], optimized read [13], [14], data mining [15], separating metadata from data [16], reducing data placement de-linearization [17], and exploiting similarity and locality of data streams [18].

A *Bloom filter* [19], [20] is a data structure which can be used to quickly probabilistically determine set membership; false positives are possible but not false negatives. They are widely used in deduplication systems to quickly filter out unique fingerprints, and we incorporate a Bloom filter into the lazy method.

Other work proposes improving the performance of data deduplication by accelerating some computational sub-tasks. A *graphics processing unit* (GPU) is a commonly used many-core co-processor, and researchers have used GPUs to improve deduplication performance. Debnath et al. [21] used a GPU to accelerate the chunking process while Li and Lilja [22] used it to accelerate hash calculation. Ma et al. [23] used the PadLock engine on a VIA CPU [24] to accelerate SHA1 (fingerprint) and AES (encryption) calculation. Here, we use a GPU to accelerate fingerprint calculation in our lazy deduplication prototype. Incremental Modulo-K was proposed by Jaehong et al. [25] for chunking instead of Rabin Hash [26]. Bhatotia et al. [21] performed content-based chunking process on GPU using CUDA [27].

Clements et al. [28] presented a decentralized deduplication for a SAN cluster which buffers updates and applies them "out of band" in batches. They focus on write performance rather

than the disk bottleneck, and do not include the cache lookup problem when buffering fingerprints.

Storing fingerprints on solid-state drives SSDs (instead of hard disk drives) can improve fingerprint lookup throughput [29], [30]. Dedupv1 [31] was designed to take advantage of the "sweet spots" of SSD technology (random reads and sequential operations). ChunkStash [32] also was designed as an SSD-based deduplication system and uses Cuckoo Hashing [33] to resolve collisions. SkimpyStash [34] is a Key-Value Store which uses the SSD to store the Key-Value pairs. We also investigate the effect of SSDs vs. HDDs in lazy deduplication.

Approximate deduplication systems (as opposed to *exact* deduplication) do not search for uncached fingerprints on disk [7]–[9], which reduces disk I/O during deduplication but at the expense of disk space. This family of methods includes sparse indexing [7] and extreme binning [8] (see also SiLo [9]). However, the lazy and eager methods perform exact deduplication, and consequently make on-disk lookups, so they are both slower than approximate methods. But unlike the eager method, the lazy method merges on-disk lookups.

The Data Domain File System [4] uses a Bloom filter, stream-informed segment layout, and a locality preserved cache, together reducing the disk I/O for index lookup by around 99%. The lazy method uses similar data structures but different fingerprint identification process.

The remainder of this paper is organized as follows: Section III describes the overall idea of the proposed lazy method and the arising challenges. We give experimental results in Section IV and summarize the paper and suggest future work in Section V.

III. Lazy deduplication

A flowchart of the overall process of lazy deduplication is given in Figure 1.

We use a Bloom filter in the lazy method to filter out previously unseen (unique) fingerprints for which we can bypass caching and buffering, and immediately write to disk. Fingerprints which pass through the Bloom filter are first looked up in the cache, which we refer to as *pre-lookup*, and fingerprints not in the cache are buffered. Finding a fingerprint as a result of an on-disk lookup triggers prefetching, after which some of the fingerprints in the buffer are looked up in the cache, referred to as *post-lookup*.

Pre-lookup exploits repeated fingerprints occurring in close proximity within the fingerprint stream, whereas post-lookup exploits recurring patterns of fingerprints throughout the fingerprint stream.

A. Fingerprint management

Lazy deduplication aims at decreasing disk access time by deferring and merging on-disk fingerprint lookups. Fingerprints which need to be looked up on disk are initially stored in an in-memory hash table, the *buffer*. They are stored until the number of fingerprints in a hash bucket reaches a threshold, which we refer to as the *buffer fingerprint threshold* (BFT). When the threshold is reached, all of the fingerprints

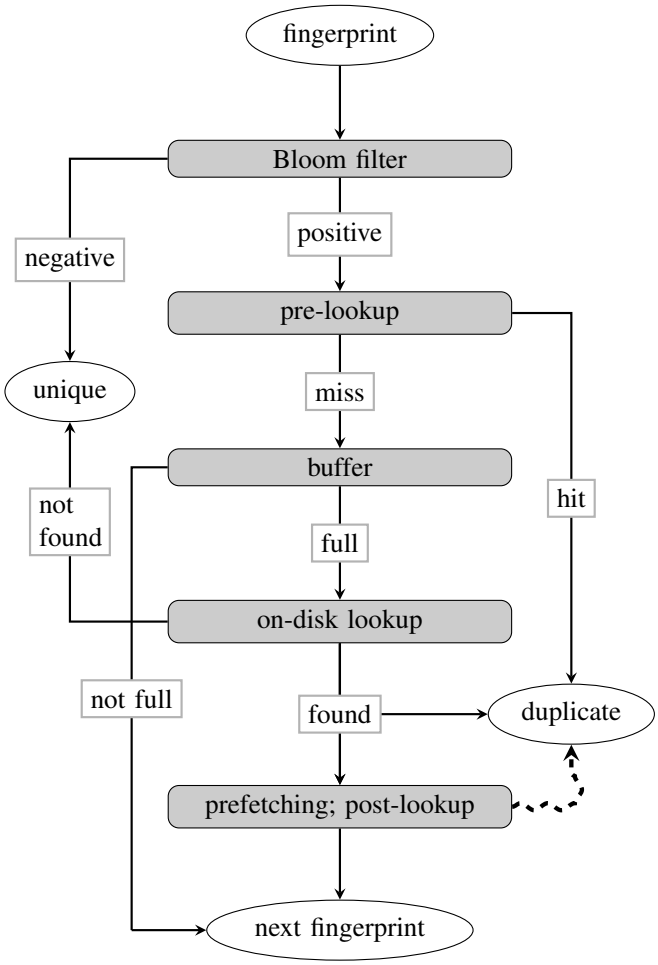


Fig. 1: Flowchart of an individual fingerprint lookup in the proposed lazy method. We either end at "unique", where we determine that the fingerprint belongs to a previously unseen chunk, "duplicate", where we find the matching on-disk fingerprint, or "next fingerprint", where we move to the next fingerprint. Prefetching and post-lookup are triggered when an on-disk lookup is performed, which might identify previously buffered duplicate fingerprints.

within the hash bucket are searched for on disk. Figure 2 illustrates the underlying idea behind lazy method. The system searches for the in-buffer bucket fingerprints among the on-disk buckets with the same bucket ID using a fingerprint-to-bucket function, proceeding bucket by bucket. Fingerprints not found are unique, which are "false positives" by the Bloom filter.

Fingerprints are stored on disk in two ways:

- Unique fingerprints are stored in an *on-disk hash table*, which is used to facilitate searching. The on-disk hash table and the buffer use the same hash function. For the on-disk hash table, we use separate chaining to resolve bucket overflow.
- Both unique and duplicate fingerprints are stored in a log-structured *metadata* array. They are stored in the order in

which they arrive, thereby preserving locality. A fingerprint in the on-disk hash table points to the corresponding metadata entry, and the neighboring metadata entries are prefetched into the cache when one fingerprint is found in the on-disk hash table.

This method could easily be adapted for systems like ChunkStash [32] or BloomStore [35], as they also use a hash table to organize fingerprints. Specifically, we could similarly perform on-disk searching in batches within search spaces restricted by hash values.

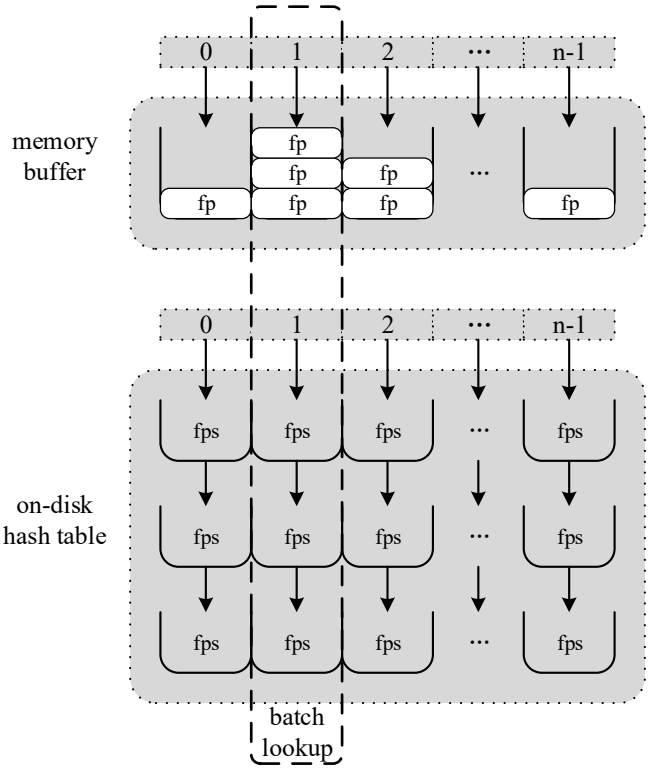


Fig. 2: Illustration of the lazy method. Three fingerprints are buffered in hash bucket 1, making it full. Together, they are searched for on disk among the fingerprints with the same bucket ID using a fingerprint-to-bucket function. Here, we use "fp" to denote an arbitrary fingerprint, and n to denote the length of the hash index.

To buffer the fingerprints, the lazy method requires additional memory space, and since we cannot know *a priori* which chunks are duplicates, the data chunks need to be buffered too. Assume a hash table with n buckets is used to buffer the fingerprints, the size of each fingerprint is S_{fp}, BFT is set to T, and the average chunk size is S_{chk}. Then the memory required to buffer the fingerprints is $\text{Space}_{\text{fp}} := nS_{\text{fp}}T$ and the average memory occupied by the corresponding chunks is $\text{Space}_{\text{chk}} := nS_{\text{chk}}T$. So the extra space required is given by

$$\text{Space} := \text{Space}_{\text{fp}} + \text{Space}_{\text{chk}} = (S_{\text{fp}} + S_{\text{chk}})nT.$$

If e.g. SHA1 is the fingerprint algorithm and we set the average chunk size to 4KB, which are typical in deduplication systems, the number of hash buckets $n = 1024$, and BFT is set to 32, then $\text{Space} \simeq 128$MB, not a huge cost on modern hardware.

An entry in an on-disk hash bucket will have size around 40B, comprising of the fingerprint itself, a pointer to the corresponding metadata entry and some other information. (The entry size will depend on the choice of cryptographic hash function and the size of the pointers.) A 4MB hash bucket can therefore contain around 100,000 entries, and, assuming there's a single 4MB hash bucket in each hash index slot, the whole on-disk hash table can support $1024 \times 100,000 \times 4\text{KB} \simeq 400$GBs of unique data. With a BFT set to 32, the buffer will have at most 32×1024 fingerprints in it at a given time, for which we need to reserve at least $32 \times 1024 \times 4\text{KB} = 128$MBs of memory for storing the corresponding chunks. This guarantees the system identifies 32 fingerprints per disk I/O.

By adjusting the number of hash buckets n, the amount of unique data supported by the on-disk hash table scales linearly with the amount of memory we need to reserve for the buffer. Thus, mGBs of memory allocated to the buffer is required for a data set with $\simeq 3000m$GBs of unique data. Duplicate fingerprints will not appear in the on-disk hash table.

Should this be a limiting factor, we can either adjust the hash bucket size or use a chain-based on-disk hash table (illustrated Figure 2), where each hash slot indexes multiple buckets. However, both of these would reduce the search performance.

B. Caching Strategy

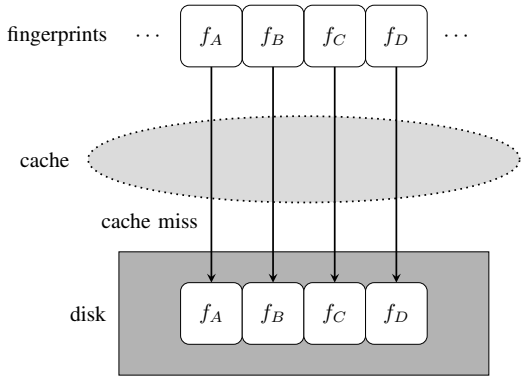
Fingerprint caching has proven to be a significant factor in data deduplication systems. Repeated patterns in backup data streams have been leveraged to design effective cache strategies to minimize disk accesses [4], [6]–[9], [36].

For the eager method, Figure 3 illustrates how locality is exploited in caching. Data chunks often arrive in a similar order to which they came previously, so when a fingerprint is found on disk, the subsequent on-disk fingerprints are prefetched into the cache. When subsequent incoming fingerprints arrive, they are often found among these prefetched fingerprints, resulting in cache hits.

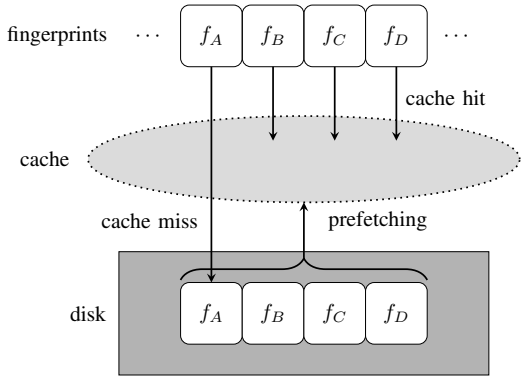
In the lazy method, fingerprints will instead be buffered, so we cannot use the same caching strategy as eager deduplication. Figure 4 modifies Figure 3b showing the caching method used in lazy deduplication. Fingerprints are buffered when processing the data stream, and will not be looked up on disk until their corresponding hash bucket is full. Subsequent incoming fingerprints will arrive before prefetching occurs, and will therefore be buffered too.

This caching strategy introduces two issues: (a) we need to decide which fingerprints should be prefetched, and (b) we need to decide which fingerprints in the buffer should be searched for in the cache after prefetching. These are addressed by using "buffer cycles" and recording a "rank".

In addition to the hash table, fingerprints that reach the buffer are inserted into a *buffer cycle*, a cyclic data structure where pointers indicate the previous and next fingerprints in the cycle. They are also stored with a number r, which we call



(a) The first encounter. The fingerprints are written to disk in order.



(b) A subsequent encounter. The fingerprint f_A is found on disk, and subsequent fingerprints are prefetched.

Fig. 3: Illustrating caching in the eager method and how data locality is exploited. The fingerprints f_A, f_B, f_C, and f_D are processed in order, and are stored on disk in that order to facilitate later prefetching.

the *rank*, which gives the order in which fingerprints arrives. The first fingerprint in a cycle has rank 0 and the subsequent fingerprints have rank $1, 2, \ldots$, including both unique and duplicate fingerprints. This is illustrated in Figure 5.

Buffer cycles and the rank are used to facilitate bidirectional prefetching: when a fingerprint with rank r is searched for on disk, we prefetch a sequence of N consecutive fingerprints (we use $N = 2048$), starting from the r-th preceeding fingerprint. The fingerprints in the buffer cycle are likely to have matching, prefetched fingerprints. So the system searches the fingerprints in the same cycle after prefetching. Figure 6 illustrates the roles of a buffer cycle and ranks.

When a fingerprint passes through the Bloom filter it is usually a duplicate, and if it's not found during pre-lookup we insert it into the current buffer cycle. Some (necessarily unique) fingerprints will be filtered out by the Bloom filter, while appearing within sequences of duplicate fingerprints. This situation often arises as the result of small modifications to a file. Fingerprints which don't make it to the buffer are not added to a buffer cycle, but we keep track of their existence using the rank.

If the number of consecutive fingerprints filtered out by the

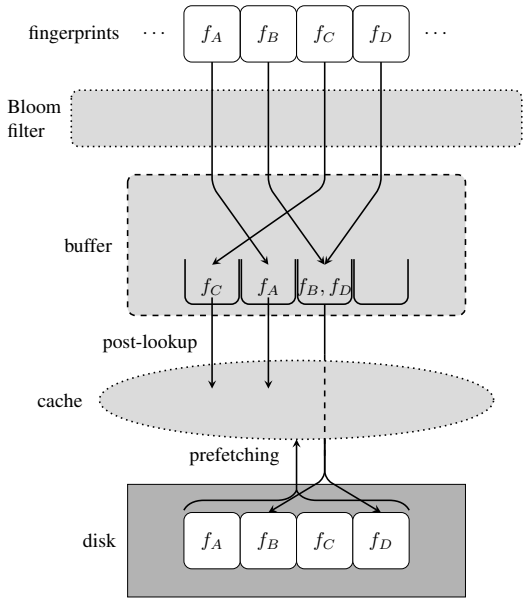


Fig. 4: Illustrating caching in the lazy method. The fingerprints f_A, f_B, f_C, and f_D are buffered, and when one of the hash buckets is full, it is looked up on disk as a batch (without checking the cache). This triggers prefetching, after which post-lookup is performed and, as a result, fingerprints f_A and f_C are found in the cache.

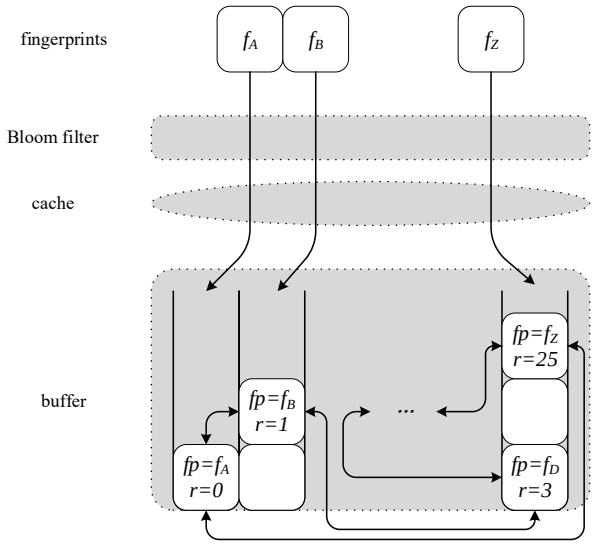


Fig. 5: An example of a buffer cycle and the fingerprint ranks r. In this example, a new buffer cycle is created when the fingerprint f_A arrives, which begins a sequence of 26 consecutive fingerprints $f_A, f_B, \ldots, f_Z$, of which all except f_C (not shown) are buffered. No fingerprint in this buffer cycle has rank 2.

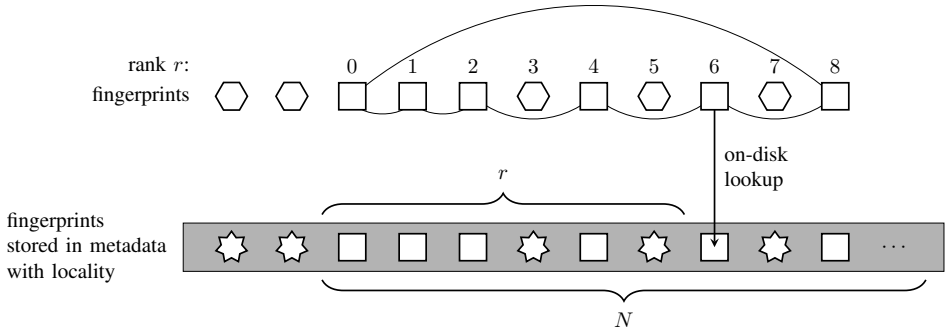


Fig. 6: Illustrating the role of a buffer cycle and the rank in prefetching. Squares represent duplicate candidates, while hexagon and stars represent unique (and therefore distinct) fingerprints in two sequences. The incoming fingerprints drawn as squares are those that are buffered (i.e., they pass through the Bloom filter and are not found during pre-lookup); one by one, they are inserted into the current buffer cycle. An on-disk lookup is performed for some fingerprint, and we prefetch N surrounding on-disk fingerprints starting from the r-th preceding fingerprint. If a similar sequence of fingerprints has occurred previously, it will be prefetched into the cache, and the buffer cycle tells us which fingerprints to look for in the cache.

Bloom filter exceeds a threshold (we use 200), we start a new cycle starting with the next duplicate fingerprint. When the length of the cycle reaches the maximum allowed length (chosen to equal the prefetching volume), we also start a new cycle and add the incoming duplicate fingerprint to the new cycle.

There will generally be many buffer cycles (one of which being the "current" buffer cycle, to which fingerprints are added), and every fingerprint in the buffer will ordinarily belong to a unique buffer cycle. When a hash bucket becomes full, the fingerprints in it will be searched for on the disk. Fingerprints not found on the disk are unique and are written to disk. Fingerprints found on the disk are duplicates, and when found, prefetching is triggered. After prefetching, the fingerprints in the same cycle are searched for in the cache (i.e., post-lookup). We use the Least Recently Used (LRU) eviction strategy to update the cache; newly added fingerprints stay longer to facilitate the flowing pre-lookups.

Some fingerprints will be searched for in the cache several times. This happens for unique fingerprints (which pass through the Bloom filter), and for fingerprints without locality with the fingerprints in the same buffer cycle. To alleviate this, we limit the number of cache lookups per fingerprint in the buffer to 10, after which the system removes the fingerprint from its buffer cycle. For fingerprints outside of buffer cycles, prefetching is not triggered after it is found on disk, avoiding disk I/O for prefetching for fingerprints without locality.

C. Disk management

The three main on-disk components are: data, metadata, and the hash table, which we describe in this section. The disk layout for metadata and the on-disk hash table is illustrated in Figure 7.

D. Prototype implementation

1) Data: Data are stored in a log-structured layout, divided into *data segments* of fixed maximum size. Incoming chunks, if found to be unique (i.e., have not been seen previously), are added to the "current" data segment, which when full, is added to the disk. Data segments are *chained*, i.e., each data segment has a pointer to the next data segment.

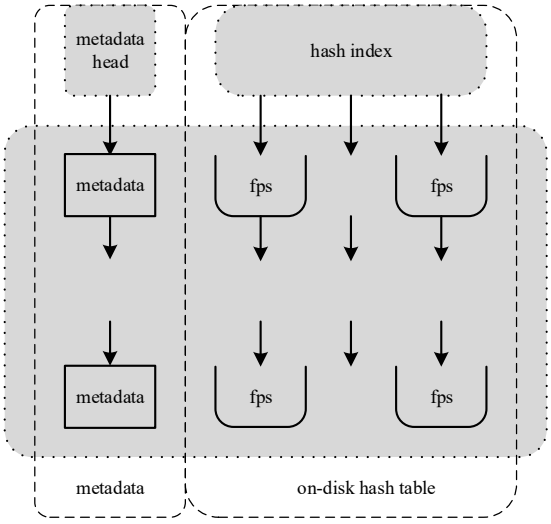


Fig. 7: Disk layout for metadata and the on-disk hash table in lazy deduplication. Here "fps" denotes a collection of fingerprints. The storage of data is not shown.

2) Metadata: Pointers (8-byte offset; 4-byte length) to the data chunks together with the fingerprints are stored in the metadata in a log-structured layout, divided into *metadata segments* of fixed maximum size (for simplicity, we choose the same maximum size as for data). For each incoming fingerprint, whether unique or not, an entry is stored to the location of the corresponding chunk in the data. We use metadata to keep track of the temporal order in which chunks arrive. The metadata stores the information about which chunks a file consists of; they are small and there are duplicate metadata

entries.

3) On-disk hash table: When one of the hash buckets in the buffer is full, the fingerprints in it are looked up as a batch on disk. The on-disk fingerprints are stored in the on-disk hash table. Fingerprints are stored together with a pointer to a corresponding metadata entry. On-disk hash buckets are chained together to facilitate on-disk lookups of fingerprints. For each unique chunk, after its metadata entry is inserted, one hash table entry is added to the corresponding bucket. An entry consists of the fingerprint, an 8-byte pointer to show the metadata entry position, and a pair (8-byte offset, 4-byte length) giving the chunk information. Entries in the bucket are stored one by one until the bucket is full.

We implement a lazy deduplication prototype using the CDC chunking method [37] with a 4KB target. The Rabin Hash algorithm is used to calculate the signature in the sliding window. We use SHA1 to calculate the fingerprints.

We use multiple CPU threads and a GPU to accelerate chunking and fingerprinting. The system creates 16 chunking threads, each for processing data in 64MB blocks. On fingerprint calculation, the system transfers batches of 4096 chunks to the GPU. The GPU assigns a thread to each data chunk in a batch to calculate its fingerprint. The system then transfers the batch of fingerprints from the GPU to the main memory.

In our implementation, the cache is organized into a hash table with collision lists and a LRU eviction policy. We bypass the file system cache to avoid its impact on the experimental results.

IV. Performance evaluation

We measure *deduplication time* which we define as, for a given data set, the time it takes to classify pre-computed fingerprints as "unique" or "duplicate".

When measuring deduplication time, the process is simulated, in that we do not include the time for chunking, fingerprinting, and writing data chunks to disk. In this way, we focus on fingerprint lookup performance. Reading the data from disk and writing the unique data chunks to disk will affect deduplication performance, but disk storage methods are instead chosen to optimize the system's online performance, and go beyond the scope of this paper. We compare the deduplication time of the lazy method to the eager method.

We also investigate *deduplication throughput*, where we include chunking and fingerprint calculation time (except for the FSLHomes data set, where we only have access to fingerprints). Note that the GPU and CPU parallelism is used to speed up chunking and fingerprint calculation (except for FSLHomes).

Each experiment runs 10 times and we give the average results. The errors encountered were consistently negligible (typically around 0.3%) and are omitted.

A. Experimental details

To compare eager and lazy deduplication as fairly as possible, they are both assigned a fixed 1GB Bloom filter, and we allocate them the same amount of memory (256MB in two experiments). For eager deduplication, the memory is fully allocated to the cache. For lazy deduplication, half of the memory is reserved for the buffer (storing both fingerprints and their corresponding chunks), and the remainder is allocated to the cache. The lazy method always has BFT set to 32 except for the test to evaluate the influence of BFT.

Table I lists the platform details. The operating system was installed on one HDD (HDD-OS). SSD-M and HDD-M respectively refer to the SSD and HDD used to store metadata and the on-disk hash table. Except for the HDD vs. SSD throughput performance test, we always perform deduplication on the SSD.

CPU	Intel(R) Core(TM) i7-3770 @3.40GHz
Memory	4× (CORSAIR) Vengeance DDR3 1600 8GB
GPU	GeForce Titan (NVIDIA Corporation Device 1005 (rev. a1))
HDD-OS	WDC WD20EARX-07PASB0 2TB 64MB IntelliPower
HDD-M	WDC WD5000AADS-00S9B0 500GB 16MB 7200rpm
SSD-M	OCZ-AGILITY3 120GB
OS	CentOS release 6.3 (Final)
Kernel	Linux-2.6.32-279.22.1.e16.x86_64

TABLE I: Platform details.

Table II lists the details of the three data sets we used in our experiments:

- *Vm* refers to pre-made VM disk images from VMware's Virtual Appliance Marketplace[1], which is used by Jin [38] to explore the effectiveness of deduplication on virtual machine disk images.
- *Src* refers to the software sources of ArchLinux, CentOS, Fedora, Gentoo, Linux Mint, and Ubuntu at 5 June 2013, collected from the Linux software source server at Nankai University.
- *FSLHomes*[2] is published by the File system and Storage Lab (FSL) at Stony Brook University [39]. It contains snapshots of students' home directories. The files consist of source code, binaries, office documents, and virtual machine images. We collect the data in 7-day intervals from the year 2014, simulating weekly backups. If the data on one date are not available, we choose the closest following available date. These are combined into the FSLHomes data set.

Unlike Vm and Src, FSLHomes directly gives the fingerprints, so chunking cannot be performed. FSLHomes has a large amount of redundant data.

	total size	duplication
Vm	220.85GB	35%
Src	434.88GB	19%
FSLHomes	3.58TB	91%

TABLE II: Data sets used for the experiments along with the proportion of duplicate data.

[1] http://www.thoughtpolice.co.uk/vmware/

[2] http://tracer.filesystems.org/traces/fslhomes/2014/

B. Deduplication time

Table III gives the deduplication times for eager and lazy deduplication on the three data sets (Vm, Src, and FSLHomes). With the lazy method, deduplication time is reduced by 46%, 53%, and 32% on Vm, Src, and FSLHomes, respectively. This experiment consistently shows that lazy deduplication is faster than eager deduplication.

	Vm	Src	FSLHomes
eager	282	476	5824
lazy	151	226	3939

TABLE III: Deduplication time (sec.) for lazy deduplication and eager deduplication.

C. Buffer cycle effectiveness

We test the effectiveness of the lazy fingerprint buffer strategy (which utilizes buffer cycles and ranks) by comparing it with a *buffer-exhausting* strategy, which instead compares all the fingerprints in the buffer area with the prefetched ones to find as many duplicate fingerprints as possible. The results are shown in Table IV. During the test, we disable pre-lookup, which could interfere with the strategies' effectiveness.

Data set	Vm		Src		FSLHomes	
Method	lazy*	exh.*	lazy*	exh.*	lazy*	exh.*
cache lookup	11	83	9	50	279	5474
on-disk lookup	41	32	58	37	19464	18398
prefetching	74	60	82	69	1681	1882
other	74	63	106	90	2544	2720
total	199	237	255	246	23969	28474

TABLE IV: Deduplication time (sec.) for the lazy deduplication (lazy*) and the buffer-exhausting strategy (exh.*). *Pre-lookup has been disabled.

Due to its design, the buffer-exhausting strategy has the following properties (compared with the lazy buffering strategy):

- It finds more fingerprints in the cache, but has a low cache hit rate. As a result, the buffer-exhausting method saves 5% to 36% of the time spent on on-disk lookup, while the time spent on cache lookups increases by a factor of 7 to 20.
- It prefetches less often, since prefetching is triggered after a fingerprint is found on the disk.

Generally, the buffer cycle has a better performance than the buffer-exhausting strategy.

D. Pre-lookup and post-lookup

We test the performance of lazy deduplication with both pre-lookup and post-lookup vs. with post-lookup alone. The results are shown in Table V.

Fingerprints found during pre-lookup are not searched for on disk and so prefetching is not triggered. Thus, we observe that the time spent on on-disk fingerprint lookup and prefetching is reduced. Using pre-lookup reduces deduplication time by 24% for Vm, 11% for Src, and 84% FSLHomes (where the majority of time spent was on on-disk lookup).

Data set	Vm		Src		FSLHomes	
Method	lazy	lazy*	lazy	lazy*	lazy	lazy*
on-disk lookup	20	41	45	58	1639	19464
prefetching	60	74	68	82	655	1681
pre-lookup	8	—	14	—	462	—
post-lookup	5	11	5	9	133	279
other	69	95	106	124	1049	2544
total	152	199	227	255	3939	23969

TABLE V: Deduplication time (sec.) with both pre-lookup and post-lookup (lazy) and with pre-lookup disabled (lazy*).

Table VI lists the cache hit rates for pre-lookup and post-lookup in lazy deduplication. (The post-lookup cache hit rate is measured without disabling pre-lookup.) Pre-lookup is used on the fingerprints that pass through the Bloom filter, identifying a significant proportion of such fingerprints. We see that Vm results in a higher pre-lookup cache hit rate and Src and FSLHomes result in a higher post-lookup cache hit rate. For all three data sets, both pre-lookup and post-lookup result a significant reduction in disk accesses.

	pre-lookup	post-lookup
Vm	74%	26%
Src	43%	57%
FSLHomes	45%	55%

TABLE VI: Cache hit rates for pre-lookup and post-lookup in lazy deduplication.

E. Buffer fingerprint threshold

Figure 8 plots the deduplication time of lazy deduplication as the buffer fingerprint threshold (BFT) varies from 4 to 60. During the test, the total memory size of the buffer and the cache is set to 256MB, so if the buffer needs more memory due to a larger BFT, there will be less memory for the cache. When limiting the memory size, 64 is the largest BFT the system can reach. Leaving a small part of memory for the cache, we set the maximum BFT as 60 in the experiment.

Experimental results show a significant impact of BFT on deduplication time. When BFT is small, the on-disk fingerprint lookup time dominates the overall time. As BFT increases, the on-disk lookup time drops quickly, and the deduplication time decreases. However, when BFT becomes large, both the on-disk lookup time and prefetching time begins to increase due to the smaller cache size.

F. Disk Access Time

Here we test the on-disk fingerprint lookup time and fingerprint prefetching time, together with the deduplication time, for eager and lazy deduplication, the results of which are shown in Table VII.

The time consumed by on-disk lookups is reduced by around 64% to 89%. As a result, its proportional contribution

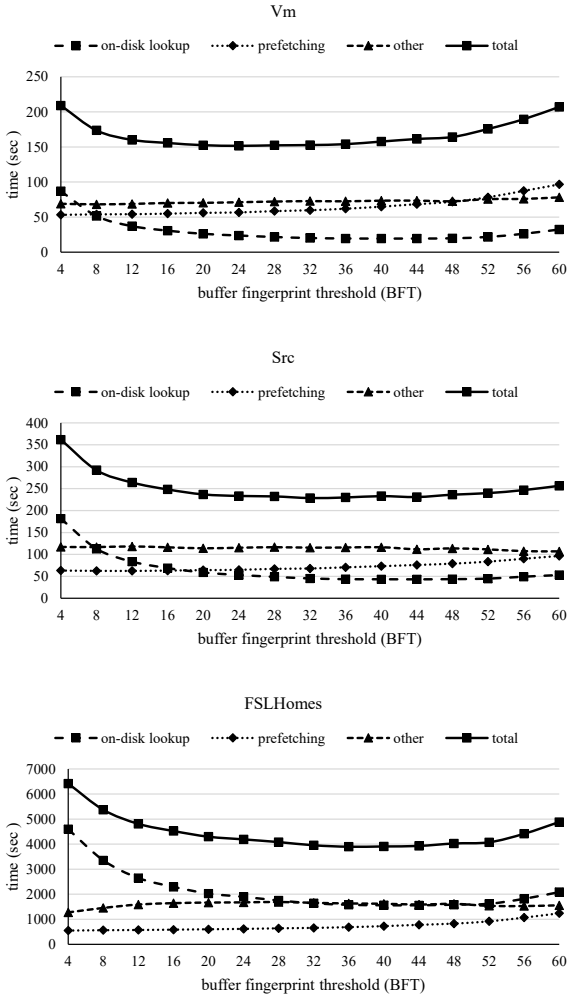


Fig. 8: Deduplication time for lazy deduplication as the buffer fingerprint threshold varies.

Data set	Vm		Src		FSLHomes	
Method	eager	lazy	eager	lazy	eager	lazy
on-disk lookup	176	20	325	45	4598	1639
prefetching	46	60	52	68	298	655
other	59	71	99	113	928	1645
total disk access	222	80	377	113	4896	2294
total dedup.	282	151	476	226	5824	3939

TABLE VII: Disk access time (sec.) for eager and lazy deduplication.

to the total time is also significantly reduced: in eager deduplication, on-disk fingerprint lookup alone takes over 62% to 84% of the total time, which drops to 13% to 42% using the lazy method. This is precisely what lazy deduplication was designed to achieve.

G. Throughput

Here we compare the throughput of lazy deduplication and eager deduplication on our HDD and SSD. For Vm and Src, we calculate the throughput (from the start) at 20GB intervals throughout the deduplication process. For FSLHomes, we calculate the throughput at the end of each "round", where a round comprises the data from one weekly backup. Since we only have the fingerprints for FSLHomes, we estimate the throughput where each fingerprint represents a 4KB chunk. Both eager and lazy deduplication utilize the GPU and CPU parallelism in the same way, but this is only relevant for the Vm and Src data sets. The results are given in Figure 9.

We see that lazy deduplication gives an improvement in the final throughput over eager deduplication of 80%, 65%, and 46% on our SSD and 150%, 119%, and 79% on our HDD, for Vm, Src, and FSLHomes, respectively. The lazy method achieves a greater throughput improvement vs. the eager method on Vm than Src since Src has less duplication, resulting in fewer on-disk lookups.

For Vm and Src, in the early stages, there are few fingerprints stored on disk, so looking up fingerprints does not require much disk I/O and throughput is limited by chunking. As more duplicate chunks arrive, the throughput drops as the system needs more disk I/O to find these duplicate chunks.

On FSLHomes, we see the overall throughput of the lazy method is 52% and 76% higher than the eager method on the SSD and HDD, respectively. In the first round, as there are initially few duplicate chunks, the deduplication does not make many disk accesses, resulting in higher throughput than the other rounds.

For Vm and Src, duplicate data arise in various places in the data stream, which results in unstable throughput. We see a drop in throughput when there are many duplicates in the data stream as this results in more on-disk lookups. For FSLHomes, as the duplicate chunks distribute evenly in each round of backup, we only see a slight change in throughput between adjacent backup rounds.

On the HDD, throughput is initially limited by chunking and fingerprint calculation, but as the procedure goes on, on-disk fingerprint lookup becomes the bottleneck. For Vm and Src, the overall throughput on the HDD is limited by disk accesses, and we see the lazy method shows a greater advantage over the eager method on the HDD vs. the SSD. This is due to the HDD having a much higher latency than the SSD.

V. Concluding remarks

In this paper, we describe a "lazy" method for data deduplication. It buffers incoming fingerprints until the number of fingerprints in a hash bucket reaches a threshold, after which they are jointly searched for on disk within a restricted search

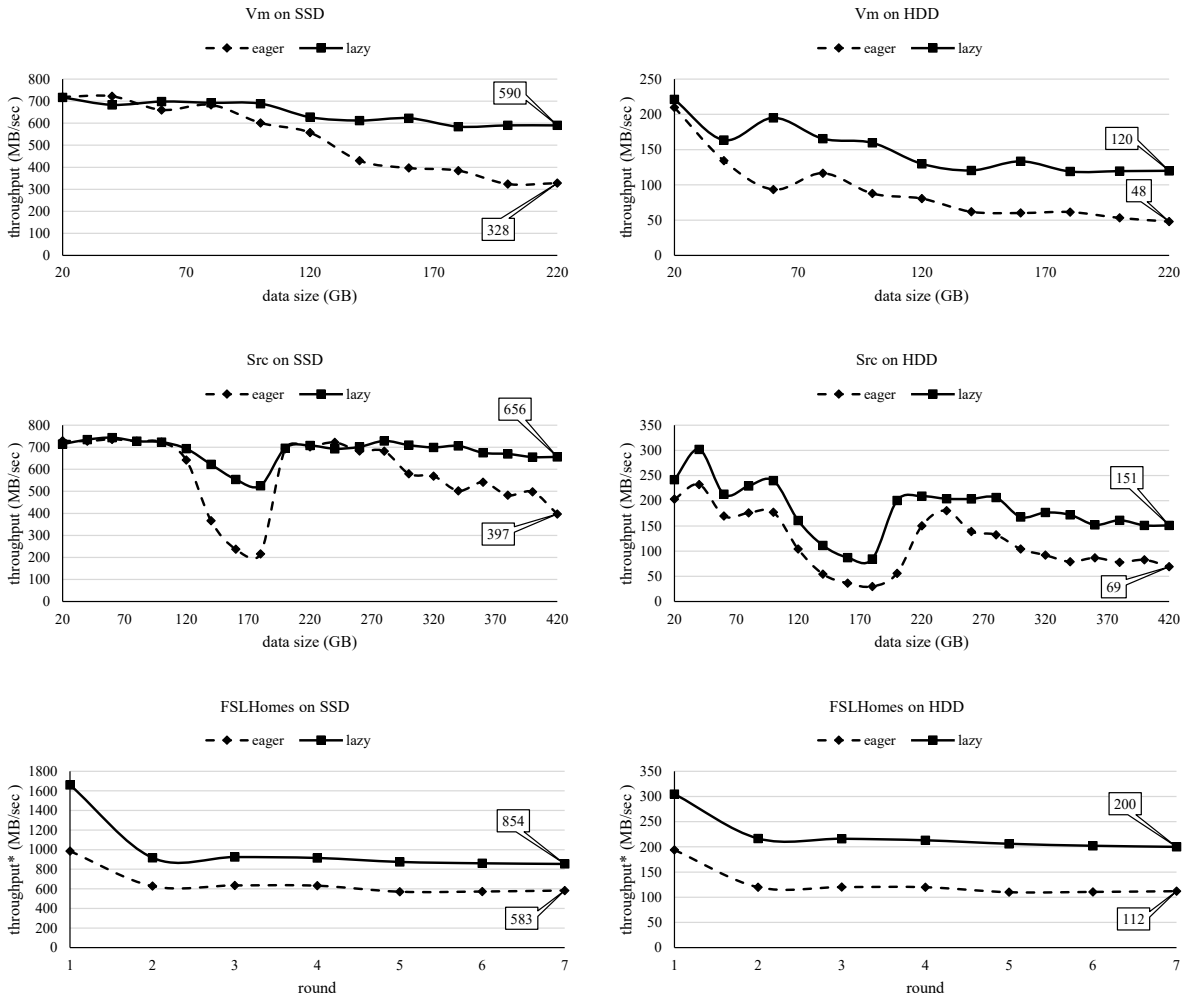


Fig. 9: Deduplication throughput on an SSD (left) and a HDD (right). *Throughput for FSLHomes does not include chunking and fingerprint calculation time.

space. We also design a caching strategy which reaches a high cache hit rate and avoids unnecessary cache lookups for fingerprints in the buffer area. Experimental results indicate that this method can be used to significantly reduce the time for on-disk fingerprint lookup, by up to 70% on SSDs and over 85% on HDDs.

We propose some future research directions:

- The lazy method would improve the performance of garbage collection in deduplication, since we can batch check the fingerprints to determine whether or not chunks are valid. It would be interesting to explore how much of an effect the lazy method has on garbage collection.
- Many key-value stores and object-oriented storage systems use a "key" to track the data blocks or objects. In this setting, we can sacrifice response time to improve throughput, and it would be interesting to investigate this trade-off in the context of lazy deduplication.
- It would also be worthwhile exploring the compatibility of lazy deduplication with commonly used data storage methods, to see when it is most effective.

Also, as the lazy method buffers both fingerprints and chunks, there is a problem in guaranteeing persistence. Buffering the chunks and fingerprints in NVRAM would be a possible way to solve this.

ACKNOWLEDGMENTS

This work is partially supported by NSF of China (grant numbers: 61373018, 11301288, 11550110491, 61379146, 61272483), Program for New Century Excellent Talents in University (grant number: NCET130301), the Fundamental Research Funds for the Central Universities (grant number: 65141021) and the Ph.D. Candidate Research Innovation Fund of Nankai University.

REFERENCES

[1] J. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, pp. 11:1–11:30, 2014.

[2] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Transaction on Storage (TOS)*, vol. 7, no. 4, pp. 14:1–14:20, 2012.

[3] S. Quinlan and S. Dorward, "Venti: A new approach to archival data storage," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, vol. 4, 2002, pp. 89–102.

[4] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2008, pp. 269–282.

[5] X. Lin, M. Hibler, E. Eide, and R. Ricci, "Using deduplicating storage for efficient disk image deployment," *EAI Endorsed Transactions on Scalable Information Systems*, vol. 15, no. 6.

[6] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2011, pp. 25–25.

[7] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2009, pp. 111–123.

[8] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *Proceedings of IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2009, pp. 1–9.

[9] W. Xia, H. Jiang, D. Feng, and H. Yu, "SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2011, pp. 26–28.

[10] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "iDedup: Latency-aware, inline data deduplication for primary storage," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2012, pp. 24–24.

[11] F. C. Botelho, P. Shilane, N. Garg, and W. Hsu, "Memory efficient sanitization of a deduplicated storage system," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2013, pp. 81–94.

[12] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "WAN-optimized replication of backup datasets using stream-informed delta compression," *ACM Transaction on Storage (TOS)*, vol. 8, no. 4, pp. 13:1–13:26, 2012.

[13] C.-H. Ng and P. P. C. Lee, "RevDedup: A reverse deduplication storage system optimized for reads to latest backups," in *Proceedings of ACM Asia-Pacific Workshop on Systems*, vol. 15, 2013.

[14] B. Mao, H. Jiang, S. Wu, Y. Fu, and L. Tian, "Read-performance optimization for deduplication-based storage systems in the cloud," *ACM Transaction on Storage (TOS)*, vol. 10, no. 2, pp. 6:1–6:22, 2014.

[15] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu, "Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2014, pp. 181–192.

[16] X. Lin, F. Douglis, J. Li, X. Li, R. Ricci, S. Smaldone, and G. Wallace, "Metadata considered harmful ... to deduplication," in *Proceedings of USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2015, pp. 11–16.

[17] Y. Tan, Z. Yan, D. Feng, X. He, Q. Zou, and L. Yang, "De-frag: An efficient scheme to improve deduplication performance via reducing data placement de-linearization," *Cluster Computing*, vol. 18, no. 1, pp. 79–92, 2015.

[18] W. Xia, H. Jiang, D. Feng, and Y. Hua, "Similarity and locality based indexing for high performance data deduplication," *IEEE Transaction on Computers*, vol. 64, no. 4, pp. 1162–1176, 2015.

[19] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[20] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang, "On the false-positive rate of Bloom filters," *Information Processing Letters*, vol. 108, no. 4, pp. 210–213, 2008.

[21] P. Bhatotia, R. Rodrigues, and A. Verma, "Shredder: GPU-accelerated incremental storage and computation," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2012, pp. 1–15.

[22] X. Li and D. Lilja, "A highly parallel GPU-based hash accelerator for a data deduplication system," in *Proceedings of IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, 2009, pp. 268–275.

[23] L. Ma, C. Zhen, B. Zhao, J. Ma, G. Wang, and X. Liu, "Towards fast de-duplication using low energy coprocessor," in *Proceedings of IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2010, pp. 395–402.

[24] VIA Technologies, "VIA Nano processor," *http://www.viatech.com.cn/cn/downloads/whitepapers/processors/WP080529VIA_Nano.pdf*, 2008.

[25] J. Min, D. Yoon, and Y. Won, "Efficient deduplication techniques for modern backup operation," *IEEE Transactions on Computers*, vol. 60, no. 6, pp. 824–840, 2011.

[26] M. O. Rabin, *Fingerprinting by random polynomials*. Center for Research in Computing Technology, Aiken Computation Laboratory, 1981.

[27] NVIDIA, "NVIDIA CUDA," *https://developer.nvidia.com/cuda-downloads*, Jul. 2013.

[28] A. T. Clements, I. Ahmad, M. Vilayannur, J. Li *et al.*, "Decentralized deduplication in san cluster file systems." in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2009, pp. 101–114.

[29] C. Kim, K.-W. Park, K. Park, and K. H. Park, "Rethinking deduplication in cloud: From data profiling to blueprint," in *Proceedings of IEEE International Conference on Networked Computing and Advanced Information Management (NCM)*, 2011, pp. 101–104.

[30] J. Kim, C. Lee, S. Lee, I. Son, J. Choi, S. Yoon, H. ung Lee, S. Kang, Y. Won, and J. Cha, "Deduplication in SSDs: Model and quantitative analysis," in *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2012, pp. 1–12.

[31] D. Meister and A. Brinkmann, "dedupv1: Improving deduplication throughput using solid state drives (ssd)," in *Proceedings of IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–6.

[32] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding up inline storage deduplication using flash memory," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2010.

[33] R. Pagh and F. F. Rodler, *Cuckoo hashing*. Springer, 2001.

[34] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM space skimpy key-value store on flash-based storage," in *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2011, pp. 25–36.

[35] G. Lu, Y. J. Nam, and D. H. C. Du, "BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash," in *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2012, pp. 1–11.

[36] U. Manber *et al.*, "Finding similar files in a large file system." in *Usenix Winter*, vol. 94, 1994, pp. 1–10.

[37] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2004, pp. 73–86.

[38] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proceedings of ACM Israeli Experimental Systems Conference (SYSTOR)*, vol. 7, 2009.

[39] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok, "Generating realistic datasets for deduplication analysis," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2012, pp. 261–272.