Lazy Exact Deduplication

JINGWEI MA, REBECCA J. STONES, YUXIANG MA, JINGUI WANG, JUNJIE REN, GANG WANG, and XIAOGUANG LIU, College of Computer and Control Engineering, Nankai University

Deduplication aims to reduce duplicate data in storage systems by removing redundant copies of data blocks, which are compared to one another using fingerprints. However, repeated on-disk fingerprint lookups lead to high disk traffic, which results in a bottleneck.

In this article, we propose a "lazy" data deduplication method, which buffers incoming fingerprints that are used to perform on-disk lookups in batches, with the aim of improving subsequent prefetching. 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 facilitate prefetching. Furthermore, as the proportion of deduplication time spent on I/O decreases, the proportion spent on fingerprint calculation and chunking increases. Thus, we also utilize parallel approaches (utilizing multiple CPU cores and a graphics processing unit) to further improve the overall performance.

Experimental results indicate that the lazy method improves fingerprint identification performance by over 50% compared with an "eager" method with the same data layout. The GPU improves the hash calculation by a factor of 4.6 and multithreaded chunking by a factor of 4.16. Deduplication performance can be improved by over 45% on SSD and 80% on HDD in the last round on the real datasets.

CCS Concepts: • **Information systems** → **Deduplication**; *Digital libraries and archives;*

Additional Key Words and Phrases: Deduplication, lazy method, GPU, disk I/O

ACM Reference Format:
Jingwei Ma, Rebecca J. Stones, Yuxiang Ma, Jingui Wang, Junjie Ren, Gang Wang, and Xiaoguang Liu. 2017. Lazy exact deduplication. ACM Trans. Storage 13, 2, Article 11 (June 2017), 26 pages.
DOI: http://dx.doi.org/10.1145/3078837

1. 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 [Paulo and Pereira 2014; Meyer and Bolosky 2012; Quinlan and Dorward 2002; Zhu et al. 2008; Lin et al. 2015b]. Deduplication is able to reduce storage requirements by 83%, and by 68% in primary storage [Meyer and Bolosky 2011].

This article is an extended version of Ma et al. [2016], which was presented at the MSST conference in 2016. This work is partially supported by NSF of China (grant numbers: 61373018, 61602266, 11550110491), the Natural Science Foundation of Tianjin, and the PhD Candidate Research Innovation Fund of Nankai University.

Authors' addresses: J. Ma, R. J. Stones, Y. Ma, J. Wang, J. Ren, G. Wang (corresponding author), and X. Liu (corresponding author), College of Computer and Control Engineering, Nankai University. Tongyan Road 38#, Haihe Education Park, Jinnan District, Tianjin, CN, 300350; emails: {mjwtom, rebecca.stones82, mayuxiang, wangjingui, renjunjie, wgzwp, liuxg}@nbjl.nankai.edu.cn.

DOI: http://dx.doi.org/10.1145/3078837

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 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, prefetching is triggered. We will refer to this method as the *eager* method.

In this article, 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 article works better for backup flows, the lazy method is suitable for both primary workloads and backup workloads.

Computation and disk I/O are the two most time-consuming components of data deduplication. As we improve the procedure to reduce the disk I/O, the proportion of time spent on computation becomes the dominating factor. As such, we further propose a parallel approach to improve computation in lazy deduplication. The proposed approach includes pipelining and multithreading processes on both the CPU and graphics processing unit (GPU), balancing the workloads to optimize performance.

This article extends an MSST 2016 conference paper [Ma et al. 2016] by the present authors. Additional contributions include the following:

—We describe, implement, and experiment with lazy deduplication on multicore hardware, with the aim of accelerating the computational tasks in data deduplication, including a GPU and multicore CPU. We optimize and achieve load balance by distributing different tasks at different data granularities to the CPU and GPU, which we utilize through pipelining.
—We perform experiments to test the impact of the Bloom filter buffer size on its false-positive rate. We further examine the effect of the false-positive rate on the on-disk lookup time and prefetching time, as well as the total deduplication time.

The article is organized as follows: We survey related work in Section 2. Section 3 describes the overall idea of the proposed lazy method and the arising challenges. Section 4 shows the prototype implementation and how we combine the optimization methods to achieve load balance. We give experimental results in Section 5 and summarize the article and suggest future work in Section 6.

2. BACKGROUND AND RELATED WORK

Data deduplication is used to reduce storage requirements, and consequently also reduces network data transfers in storage systems. This reduces hardware costs and improves the system's online performance. However, performing deduplication is both (1) computationally intensive, due to chunking, fingerprint calculation, and compression, and (2) I/O intensive, since we are required to compare a large number of fingerprints in order to identify and eliminate redundant data.

For large-scale deduplication systems, the main memory is not large enough to hold all the fingerprints, so most fingerprints are stored on disk, creating 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 [Zhu et al. 2008], it remains a bottleneck. Our goal is to further reduce this component by combining several fingerprint lookups into

a single disk access, to the point where computation time becomes a more prominent factor.

Real-world data will often have "locality" properties, where data blocks that occur together do so repeatedly. Deduplication systems take advantage of locality properties in data streams to reduce disk accesses by using an in-memory cache [Zhu et al. 2008; Guo and Efstathopoulos 2011; Lillibridge et al. 2009; Bhagwat et al. 2009; Xia et al. 2011; Srinivasan et al. 2012; Botelho et al. 2013]. 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 [Shilane et al. 2012], optimized read [Ng and Lee 2013; Mao et al. 2014], data mining [Fu et al. 2014], separating metadata from data [Lin et al. 2015a], reducing data placement delinearization [Tan et al. 2015], and exploiting similarity and locality of data streams [Xia et al. 2015].

A *Bloom filter* [Bloom 1970; Bose et al. 2008] is a data structure that can be used to quickly probabilistically determine set membership; false positives are possible but not false negatives. It is widely used in deduplication systems to quickly filter out unique fingerprints, and we incorporate a Bloom filter into the lazy method. The Data Domain File System [Zhu et al. 2008] uses a Bloom filter, stream-informed segment layout, and locality-preserving cache, together reducing the disk I/O for index lookup by 99%. The lazy method uses similar data structures but different fingerprint identification processes.

Other work proposes improving the performance of data deduplication by accelerating some computational subtasks. A *graphics processing unit* is a commonly used many-core coprocessor, and researchers have used GPUs to improve deduplication performance. Bhatotia et al. [2012] used a GPU to accelerate the chunking process, while Li and Lilja [2009] used it to accelerate hash calculation. Ma et al. [2010] used the PadLock engine on a VIA CPU [VIA Technologies 2008] to accelerate SHA1 (fingerprint) and AES (encryption) calculation. Incremental Modulo-K was proposed by Min et al. [2011] for chunking instead of Rabin Hash [Rabin 1981]. Bhatotia et al. [2012] performed a content-based chunking process on a GPU using CUDA [NVIDIA 2013].

In this work, we accelerate SHA1 calculation (used as the chunk fingerprint) using a simple method on the GPU, which we find is sufficient for our prototype to achieve load balance. We acknowledge that more sophisticated methods could potentially improve GPU hash function calculation, which would reduce the impact of this component, but we consider this task beyond the scope of this article.

Another buffering approach was proposed by Clements et al. [2009], who presented a decentralized deduplication method for a SAN cluster. They buffer updates (primarily new writes) and apply 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.

In addition, there are a range of alternative software and/or hardware approaches to improve deduplication, of which we list some examples. IDedup [Srinivasan et al. 2012] is an example of primary deduplication (i.e., deduplication applied to primary workloads), which reduces fragments by selectively deduplicating sequences of disk blocks. I/O deduplication [Koller and Rangaswami 2010] introduces deduplication into the I/O path, with the result of reducing the amount of data written to disk, thereby reducing the number of physical disk operations.

Storing fingerprints on solid-state drives SSDs (instead of hard disk drives) can also improve fingerprint lookup throughput [Kim et al. 2011, 2012b]. Dedupv1 [Meister and

Brinkmann 2010] was designed to take advantage of the "sweet spots" of SSD technology (random reads and sequential operations). ChunkStash [Debnath et al. 2010] also was designed as an SSD-based deduplication system and uses Cuckoo Hashing [Pagh and Rodler 2001] to resolve collisions. SkimpyStash [Debnath et al. 2011] is a Key-Value Store that uses the SSD to store the Key-Value pairs. We also investigate the effect of SSDs versus HDDs in lazy deduplication. Additionally, deduplication also benefits SSDs in several aspects, such as expanding SSD lifetime, improving write performance, and reducing garbage collection overhead [Kim et al. 2012a].

Approximate deduplication systems (as opposed to *exact* deduplication) do not search for uncached fingerprints on disk [Lillibridge et al. 2009; Bhagwat et al. 2009; Xia et al. 2011], which reduces disk I/O during deduplication but at the expense of disk space. This family of methods includes sparse indexing [Lillibridge et al. 2009] and extreme binning [Bhagwat et al. 2009] (see also SiLo [Xia et al. 2011]). 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. Approximate deduplication is most beneficial when the main memory is large enough to hold all the samples. Lazy deduplication aims at reducing on-disk fingerprint lookup, which ordinarily cannot benefit approximate deduplication. However, when the samples spill onto the disk, approximate deduplication can use the lazy method to deal with the on-disk part. It's plausible, therefore, that approximate deduplication could likewise benefit from the lazy method, although we do not explore this idea in this article.

3. LAZY DEDUPLICATION

3.1. Fingerprint Identification

A flowchart of the fingerprint identification 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. In our experiments with a 1GB Bloom filter, we found all the unique fingerprints are filtered out except for FSLHomes, which resulted in a low false-positive rate of 1.14×10^{-9} (counting a single false positive; see Table VIII for further experimental results).

Fingerprints that pass through the Bloom filter are first looked up in the cache, which we refer to as *prelookup*, 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 *postlookup*.

Prelookup exploits repeated fingerprints occurring in close proximity within the fingerprint stream, whereas postlookup exploits recurring patterns of fingerprints throughout the fingerprint stream.

3.2. Fingerprint Management

Lazy deduplication aims at decreasing disk access time by deferring and merging on-disk fingerprint lookups. Fingerprints that 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 within the hash bucket are searched for on disk. Figure 2 illustrates the underlying idea behind the 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.

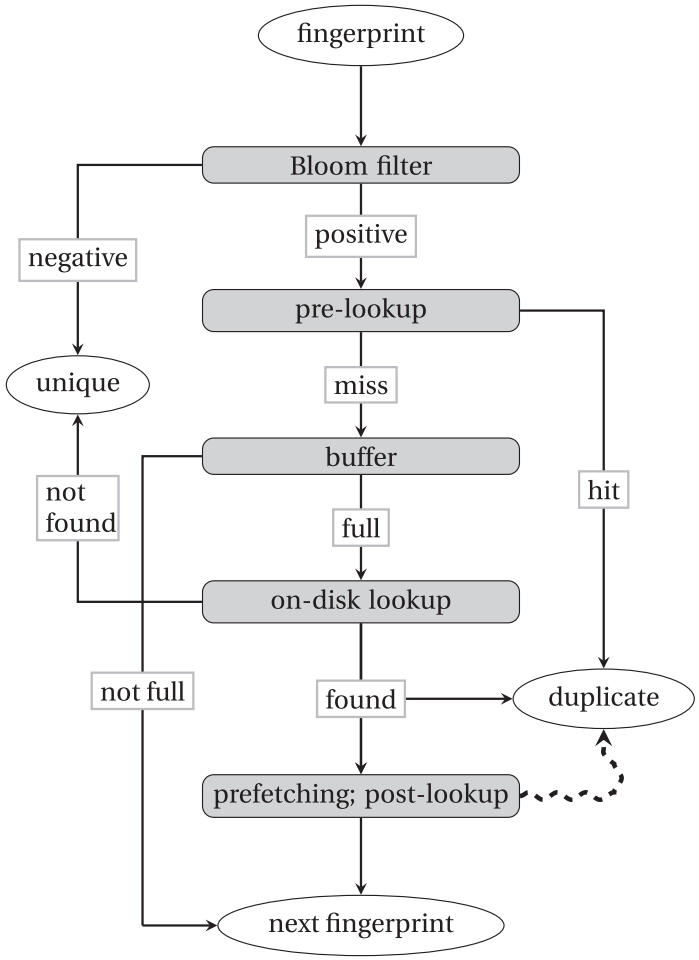


Fig. 1. Flowchart of an individual fingerprint lookup in the proposed lazy method. We either end at "unique," where we determine that the fingerprint belongs to a previously unseen chunk; "duplicate," where we find the matching on-disk fingerprint; or "next fingerprint," where we move to the next fingerprint.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

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 [Debnath et al. 2010] or BloomStore [Lu et al. 2012], as they also use a hash table to organize

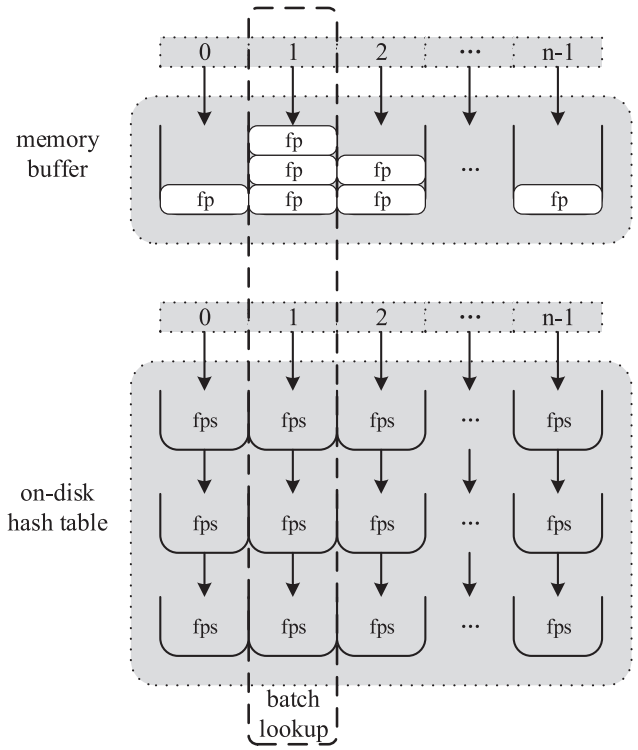


Fig. 2. Illustration of the lazy method. Three fingerprints are buffered in hash bucket 1, making it full. Together, they are searched for on disk among the fingerprints with the same bucket ID using a fingerprint-to-bucket function. Here, we use "fp" to denote an arbitrary fingerprint, and n to denote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

fingerprints. Specifically, we could similarly perform on-disk searching in batches within search spaces restricted by hash values.

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

$$\text{Space}_{\text{fp}} := nS_{\text{fp}}T,$$

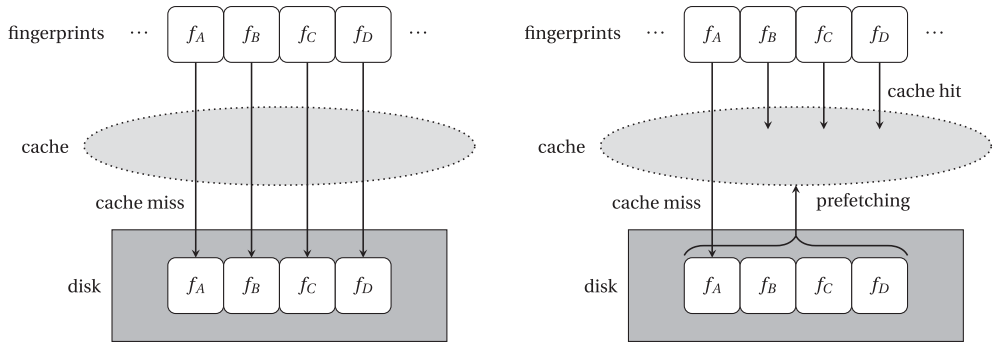
and the average memory occupied by the corresponding chunks is

$$\text{Space}_{\text{chk}} := nS_{\text{chk}}T.$$

So the extra space required is given by

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

If, for example,, 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 $n = 1{,}024$, and BFT is set to 32, then Space $\simeq$ 128MB, an acceptable cost on modern hardware.



(a) The first encounter. The fingerprints are written to disk in the order in which they arrive so that locality is preserved.

(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.

An entry in an on-disk hash bucket will have size around 40B, composed 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 $1{,}024 \times 100{,}000 \times 4\text{KB} \simeq 400\text{GB}$ of unique data. With a BFT set to 32, the buffer will have at most $32 \times 1{,}024$ fingerprints in it at a given time, for which we need to reserve at least $32 \times 1{,}024 \times 4\text{KB} = 128\text{MB}$ of memory for storing the corresponding chunks. This guarantees that 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, mGB of memory allocated to the buffer is required for a dataset with $\simeq 3{,}000m$GB 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.

3.3. 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 [Bhagwat et al. 2009; Guo and Efstathopoulos 2011; Lillibridge et al. 2009; Manber 1994; Xia et al. 2011; Zhu et al. 2008].

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 3 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

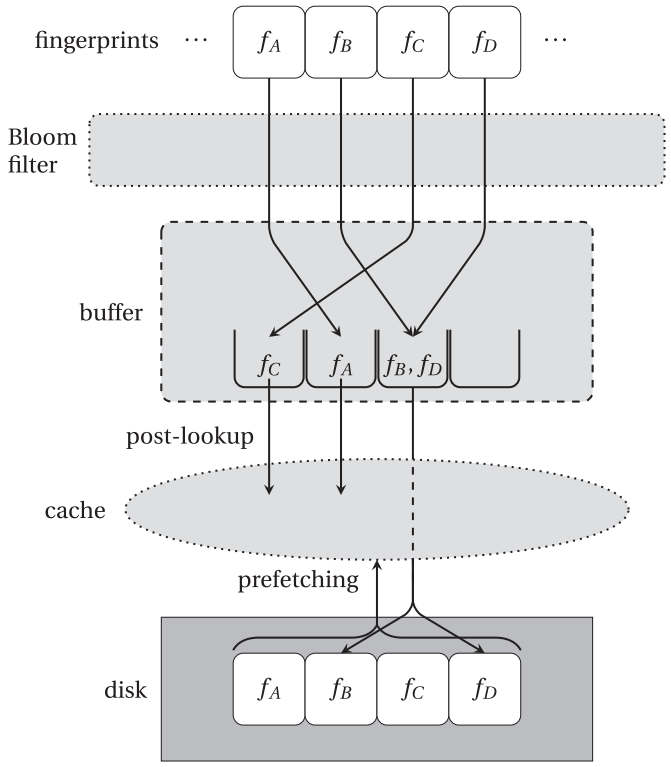


Fig. 4. Illustrating caching in the lazy method. The fingerprints f_A, f_B, f_C, and f_D are buffered, and when one of the hash buckets is full, it is looked up on disk as a batch (without checking the cache). This triggers prefetching, after which postlookup is performed and, as a result, fingerprints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

bucket is full. Subsequent incoming fingerprints will arrive before prefetching occurs, and will therefore be buffered too.

This caching strategy introduces two issues: (1) we need to decide which fingerprints should be prefetched, and (2) 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 the *rank*, which gives the order in which fingerprints arrive. 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. Algorithm 1 gives the procedure how to construct the buffer cycle.

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 $N = 2{,}048$), starting from the rth preceding 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 prelookup, we insert it into the current buffer cycle. Some (necessarily

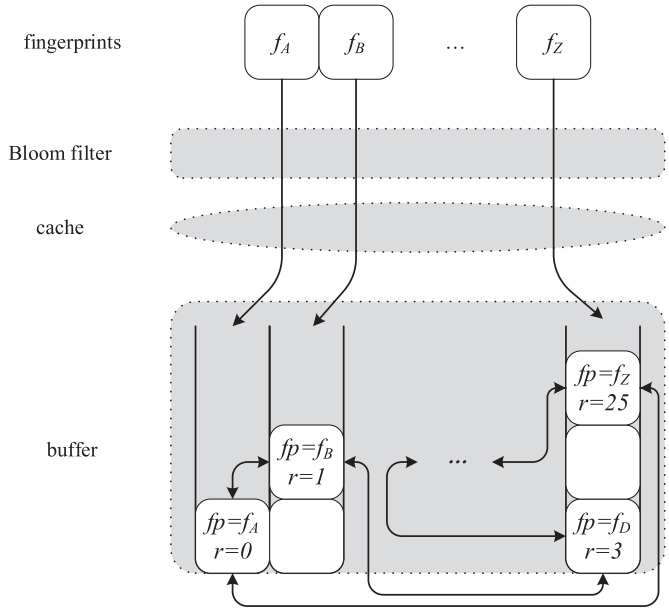


Fig. 5. An example of a buffer cycle and the fingerprint ranks r. In this example, a new buffer cycle is created when the fingerprint f_A arrives, which begins a sequence of 26 consecutive fingerprints $f_A, f_B, \ldots, f_Z$, of which all except f_C (not shown) are buffered. No fingerprint in this buffer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

ALGORITHM 1: Construction of the Buffer Cycles

Input: fingerprints $f_1, f_2, \ldots$.
MAX_UNIQUE_LENGTH=200; MAX_CYCLE_LENGTH=2048; Uniques $\leftarrow$ 0; Rank $\leftarrow$ 0;
for *each fingerprint* f_i **do**
    Test f_i using Bloom filter;
    if f_i *is found to be unique* **then**
        Uniques $\leftarrow$ Uniques + 1;
        if Uniques > MAX_UNIQUE_LENGTH **then**
            Rank $\leftarrow$ 0;
            Uniques $\leftarrow$ 0;
            Start a new buffer cycle;
        end
    else
        Rank $\leftarrow$ Rank + Uniques;
        Uniques $\leftarrow$ 0;
        Prelookup f_i in the cache;
        if f_i *is not found in the cache* **then**
            if Rank > MAX_CYCLE_LENGTH **then**
                Rank $\leftarrow$ 0;
                Start a new buffer cycle;
            end
            assign Rank to f_i and append it to the current list
        end
        Rank $\leftarrow$ Rank + 1;
    end
end

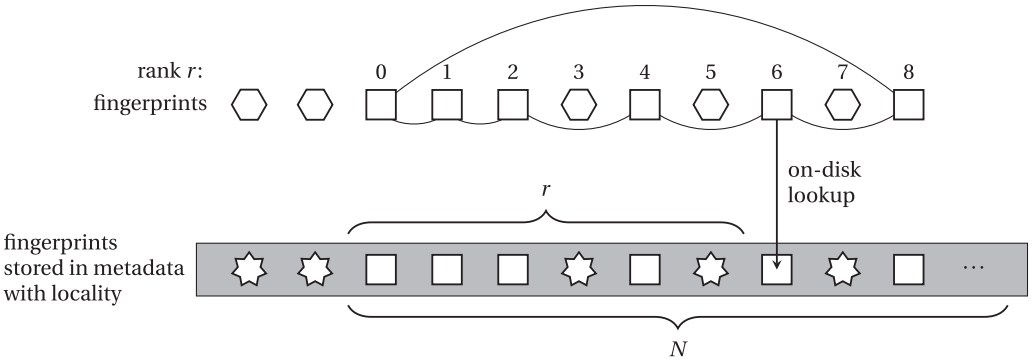


Fig. 6. Illustrating the role of a buffer cycle and the rank in prefetching. Squares represent duplicate candidates, while hexagon and stars represent unique (and therefore distinct) fingerprints in two sequences. The incoming fingerprints drawn as squares are those that are buffered (i.e., they pass through the Bloom filter and are not found during prelookup); 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 rth 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.

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 that 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 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, so all the fingerprints in the same cycle are expected to be covered by the prefetching), 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 is 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., postlookup). We use the Least Recently Used (LRU) eviction strategy to update the cache; newly added fingerprints stay longer to facilitate the flowing prelookups.

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.

3.4. Computing Optimization on a Multicore Platform

As the proportion of deduplication time spent on disk accesses decreases, the proportion of time spent on computational tasks becomes the new bottleneck (although even in the eager method, these tasks are not entirely negligible). Thus, we also aim to reduce the time spent on these computational tasks.

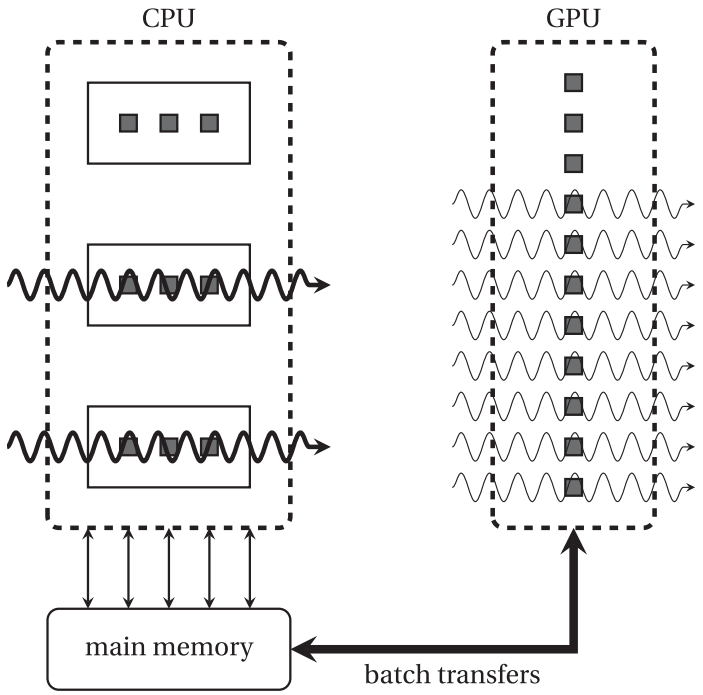


Fig. 7. Illustrating parallel processing of processing chunks on the CPU and GPU. On the CPU side, CPU threads process chunks as batches, and repeated main memory access is not a major concern. On the GPU side, individual threads are assigned individual chunks to process, and we reduce the number of main memory accesses by using batching.

With Moore's Law "coming to an end" [Waldrop 2016], CPU frequency is developing comparatively slowly, being replaced by increases in the number of CPU cores. Additionally, GPUs are increasingly being utilized in scientific computing, some of which have thousands of cores, providing powerful parallel computing resources. However, software needs to be redesigned to meaningfully utilize the GPU's highly parallel hardware. Parallel hardware allows us to overlap computational tasks (in the case of multicore hardware) or offload them to reduce the CPU load (in the case of the GPU).

Figure 7 illustrates how the CPU and GPU threads process the data chunks in our deduplication implementation. The granularity on the CPU and GPU differ. On the GPU, we assign an individual thread to processing an individual chunk, while a CPU thread processes a batch of data. The many-core GPU hardware makes it better utilized when assigned many similar-sized tasks. However, in deduplication, this requires transfers from the main memory to the GPU memory; we reduce the number of transfers by batching. On the CPU side, assigning a batch of chunks to a CPU thread avoids problems arising from frequent thread switches.

Figure 8 illustrates the data transfer procedure. In our prototype, we use the GPU to accelerate the SHA1 calculation. The host transfers a batch of chunks to the GPU memory, after which the GPU computes the SHA1 value of each chunk. The host then transfers the calculated fingerprints from the GPU memory to the main memory.

The benefits of offloading SHA1 calculation instead of chunking to the GPU are:

—The GPU hardware is not efficient for applications with intensive branch instructions or data synchronization since it is not allowable for the threads of the same thread block to concurrently jump to different branches [Li et al. 2013].

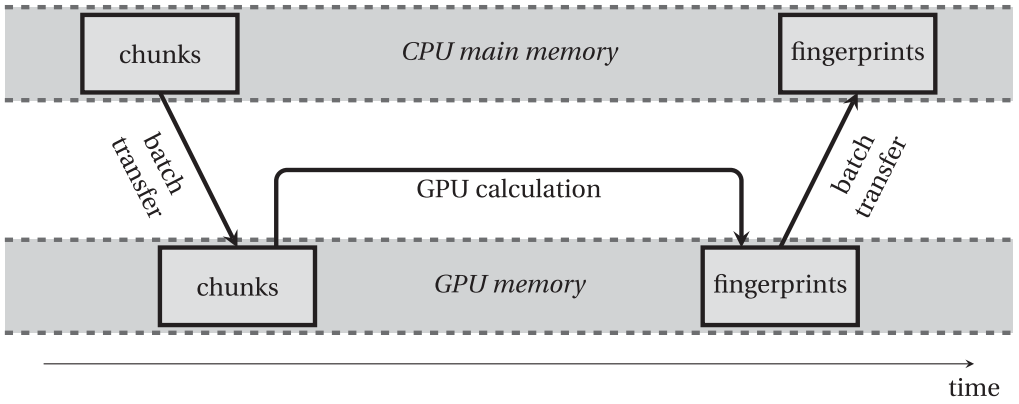


Fig. 8. Illustrating data transfers between the CPU-side main memory and the GPU memory. We use batching to reduce the number of CPU-GPU transfers.

—The input size of the chunking process is larger (chunking is the process that segments large data blocks into small, closely sized chunks). In our experimental configuration, we use a 96MB data block (as might arise in a deduplication data stream) for a chunking thread, which would be too large for a single GPU thread. Moreover, chunking is an inherently serial process, in that chunk boundaries influence one another. SHA1 calculation for the already-computed chunks is far simpler to parallelize since the chunks are of similar sizes, so the task sizes are better balanced.

3.5. Balance Between Computation and I/O

Data deduplication is both computationally intensive and I/O intensive. With lazy deduplication, we reduce the I/O component (primarily fingerprint identification) to the extent that the computational component becomes the new bottleneck (primarily computing the chunks and calculating their fingerprints). If we improve only one of these components (or even if we eliminate it entirely), Ahmdal's Law gives a theoretical limit on the benefit.

Figure 9 shows the proportion of time spent on computation in lazy deduplication as we introduce GPU fingerprinting and multithreaded chunking on the CPU. We use the FSLHomes [Tarasov et al. 2012] (which we use for experiments; see Section 5.1) as an example. We use the data from Tables VII and IX and Figure 13. This figure shows the bottleneck stage in each optimizing step.

Using the lazy method, disk I/O is reduced. As a consequence, the identification component only takes 13% of the deduplication time. Chunking and hashing consume larger proportions of the deduplication time, especially chunking, which takes over half (53%) of the entire time. So it is impossible for the overall throughput to exceed the chunking throughput, which is only 217MB/sec.

Tough fingerprinting takes less proportion of the time than chunking; it is one potential bottleneck. We offload the hash calculation to GPU, which doesn't wholly eliminate the fingerprinting component, but reduces it to 10%. Chunking remains the bottleneck, taking 74% of the deduplication time. The throughput still cannot exceed the chunking throughput. So we use multiple CPU threads to perform chunking, dropping this component to 39%. After that, the time spent on each stage is close and the system reaches a relative balance. This means that the CPU, GPU, and disk work simultaneously and the resources are relatively fully used.

The other two datasets (Vm and Src) also show a similar trend.

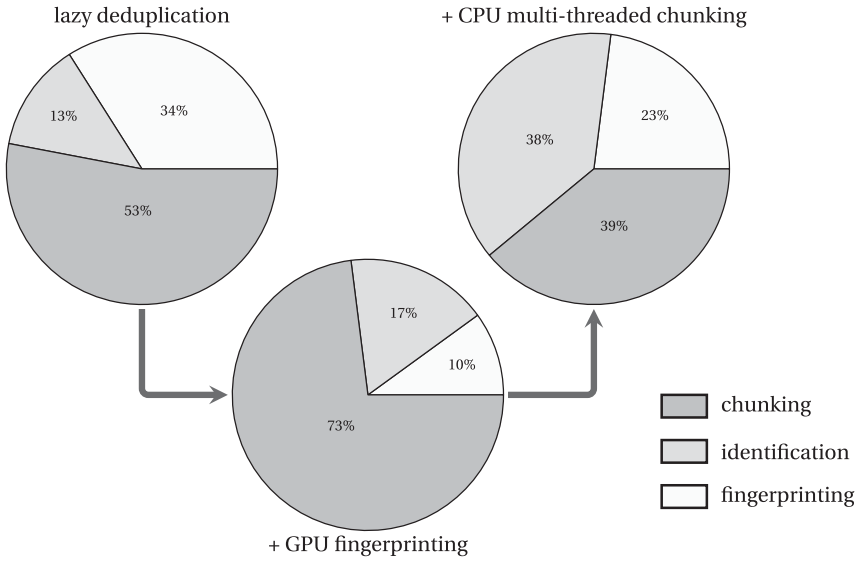


Fig. 9. How the execution time balance changes as we introduce (a) GPU fingerprinting and (b) CPU multithread chunking into lazy deduplication.

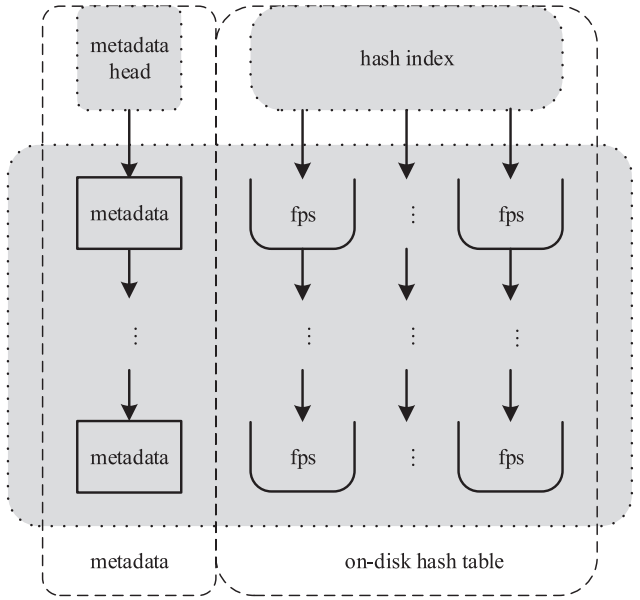


Fig. 10. Disk layout for metadata and the on-disk hash table in lazy deduplication. Here "fps" denotes a collection of fingerprints. The storage of data is not shown.

4. PROTOTYPE IMPLEMENTATION

4.1. 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 10.

4.2. Data Layout

4.2.1. 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*; that is, each data segment has a pointer to the next data segment.

4.2.2. 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.

4.2.3. 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 [Policroniades and Pratt 2004] 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.

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

4.3. Deduplication Pipeline

We use multiple CPU threads and a GPU to accelerate chunking and fingerprinting. The system creates eight chunking threads (equal to the the number of logical CPU processors on our platform), each for processing data in 96MB blocks. On fingerprint calculation, the system transfers batches of 4,096 chunks to the GPU (to closely match the GPU's 3,584 cores). The GPU assigns a thread to each data chunk in a batch to calculate its fingerprint, and then transfers the batch of fingerprints from the GPU to the main memory. Identification picks out the redundant fingerprints, which are buffered for later processing. Though all the stages except fingerprinting run on the CPU, only chunking is computationally intensive. Fingerprint identification is instead disk I/O intensive.

Though multithreading and using a GPU can improve the chunking and hash calculation procedures directly, they can also perform these tasks in parallel to one another via pipelining. Figure 11 illustrates how each stage executes in parallel. In our deduplicaton implementation, we divide the process into four stages, namely chunking, fingerprinting, identification, and storage.[1] Importantly, these components are approximately time-balanced (see Figure 9), which helps pipelining to avoid idle time. By

[1]In our deduplicaton implementation, we don't actually store the chunks, so it consumes no CPU resources.

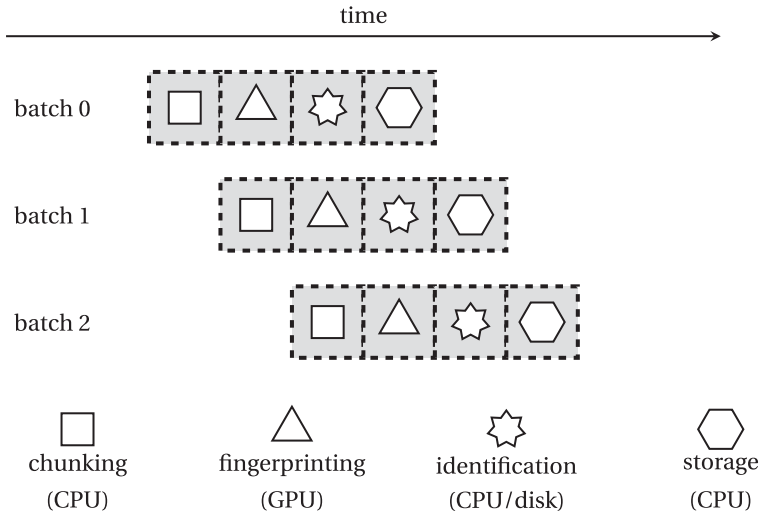


Fig. 11. Illustrating how pipelining can be used to improve computation in lazy deduplication. Batches are able to be processed in parallel using separate CPU threads.

organizing the workload onto three devices, the CPU, GPU, and disk, which can operate simultaneously, the time spent can overlap with each other.

5. PERFORMANCE EVALUATION

We measure *deduplication time,* which we define as, for a given dataset, the time it takes to classify precomputed 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 article. 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 dataset, 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 1%) and are omitted.

5.1. 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

Table I. Platform Details

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 7,200rpm
SSD-M	OCZ-AGILITY3 120GB
OS	CentOS release 6.3 (Final)
Kernel	Linux-2.6.32-279.22.1.e16.x86_64

Table II. Datasets Used for the Experiments Along with the Proportion of Duplicate Data

	Total Size	Duplication
Vm	220.85GB	35%
Src	434.88GB	19%
FSLHomes	3.58TB	91%

Table III. Deduplication Time (Sec) for Lazy Deduplication and Eager Deduplication

	Vm	Src	FSLHomes
Eager	282	476	5, 824
Lazy	151	226	3, 939

metadata and the on-disk hash table. Except for the HDD versus SSD throughput performance test, we always perform deduplication on the SSD.

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

—*Vm* refers to premade VM disk images from VMware's Virtual Appliance Marketplace,[2] which is used by Jin and Miller [2009] 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 on June 5, 2013, collected from the Linux software source server at Nankai University.
—*FSLHomes*[3] is published by the File system and Storage Lab (FSL) at Stony Brook University [Tarasov et al. 2012]. 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 is not available, we choose the closest following available date. These are combined into the FSLHomes dataset.

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

5.2. Deduplication Time

Table III gives the deduplication times for eager and lazy deduplication on the three datasets (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.

[2]http://www.thoughtpolice.co.uk/vmware/.
[3]http://tracer.filesystems.org/traces/fslhomes/2014/.

Table IV. Deduplication Time (Sec) for the Lazy Deduplication (Lazy*) and the Buffer-Exhausting Strategy (Exh.*)

Dataset	Vm		Src		FSLHomes	
Method	Lazy*	Exh.*	Lazy*	Exh.*	Lazy*	Exh.*
Cache lookup	11	83	9	50	279	5,474
On-disk lookup	41	32	58	37	19,464	18,398
Prefetching	74	60	82	69	1681	1,882
Other	74	63	106	90	2,544	2,720
Total	199	237	255	246	23,969	28,474

*Prelookup has been disabled.

Table V. Deduplication Time (Sec) with Both Prelookup and Postlookup (Lazy) and with Prelookup Disabled (Lazy*)

Dataset	Vm		Src		FSLHomes	
Method	Lazy	Lazy*	Lazy	Lazy*	Lazy	Lazy*
On-disk lookup	20	41	45	58	1,639	19,464
Prefetching	60	74	68	82	655	1,681
Prelookup	8	—	14	—	462	—
Postlookup	5	11	5	9	133	279
Other	69	95	106	124	1,049	2,544
Total	152	199	227	255	3,939	23,969

5.3. 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 prelookup, which could interfere with the strategies' effectiveness. Generally, the buffer cycle strategy has a better performance than the buffer-exhausting strategy.

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 6 to 20.
—Since Vm and FSLHomes have more redundant data, the cache lookup time occupies a greater fraction of the total time. In Src, there are relatively few duplicate chunks, so the increase of cache lookup in going from lazy to buffer exhausting does not make the performance significantly worse due to the disk access savings.
—It prefetches less often, since prefetching is triggered after a fingerprint is found on the disk.

5.4. Prelookup and Postlookup

We test the performance of lazy deduplication with both prelookup and postlookup versus with postlookup alone. The results are shown in Table V.

Fingerprints found during prelookup 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 prelookup reduces deduplication time by 24% for Vm, 11% for Src, and 84% FSLHomes (where the majority of time spent was on on-disk lookup).

Table VI lists the cache hit rates for prelookup and postlookup in lazy deduplication. (The postlookup cache hit rate is measured without disabling prelookup.) Prelookup is

Table VI. Cache Hit Rates for Prelookup and Postlookup in Lazy Deduplication

	Prelookup	Postlookup
Vm	74%	26%
Src	43%	57%
FSLHomes	45%	55%

Table VII. Disk Access Time (Sec) for Eager and Lazy Deduplication

Dataset	Vm		Src		FSLHomes	
Method	Eager	Lazy	Eager	Lazy	Eager	Lazy
On-disk lookup	176	20	325	45	4,598	1,639
Prefetching	46	60	52	68	298	655
Other	59	71	99	113	928	1,645
Total disk access	222	80	377	113	4,896	2,294
Total dedup.	282	151	476	226	5,824	3,939

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 pr-lookup cache hit rate, and Src and FSLHomes result in a higher postlookup cache hit rate. For all three datasets, both prelookup and postlookup result in a significant reduction in disk accesses.

5.5. Buffer Fingerprint Threshold

The buffer fingerprint threshold is the primary factor affecting the performance of lazy deduplication. Figure 12 plots the deduplication time of lazy deduplication as the 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 begin to increase due to the smaller cache size. There is a "sweet spot" for each dataset. In our experiments, they are all close to 32, so we choose 32 as the default BFT.

5.6. 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 64% to 89%. As a result, its proportional contribution 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.

5.7. Bloom Filter Size

The Bloom filter effectively filters out the unique fingerprints. However, as the data size grows, it becomes less effective in deduplication due to more false positives. Here we explore how the size of the Bloom filer impacts the deduplication performance. A smaller-sized Bloom filter will give rise to more frequent false positives, much like a

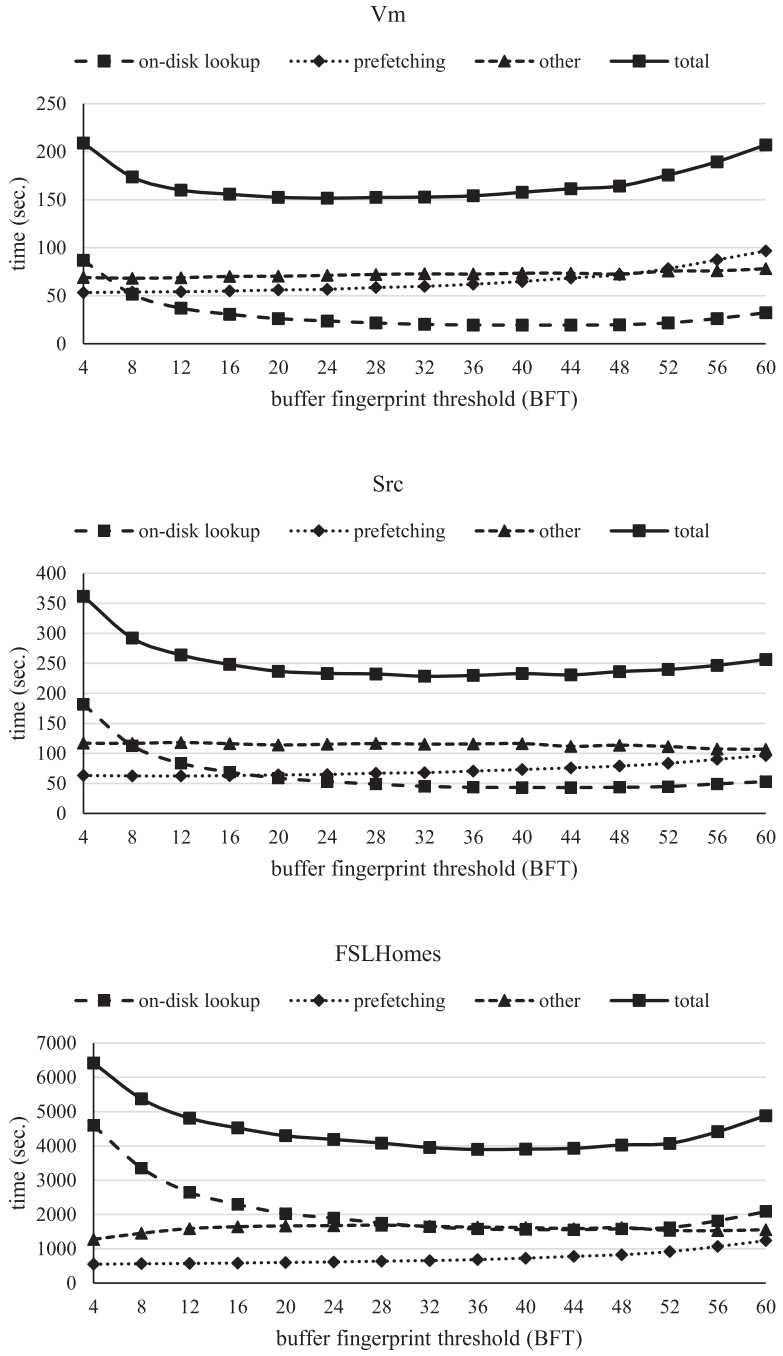


Fig. 12. Deduplication time for lazy deduplication as the buffer fingerprint threshold varies.

Table VIII. Impact of the Bloom Filter Size on False Positives (the Speedup Is for Lazy vs. Eager)

Bf Size (MB)	Datasets	Vm		Src		FSLHomes	
	Method	Eager	Lazy	Eager	Lazy	Eager	Lazy
1,024	False-positive rate	0.00		0.00		1.14×10^{-9}	
	On-disk lookup (sec)	176	20	325	45	4,598	1,639
	Prefetching (sec)	46	60	52	68	298	655
	Total dedup. (sec)	282	151	476	226	5,824	3,939
	Throughput speedup	×1.9		×2.1		×1.5	
512	False-positive rate	0.00		5.57×10^{-7}		3.08×10^{-8}	
	On-disk lookup (sec)	175	21	324	46	4,613	1,686
	Prefetching (sec)	46	61	52	69	299	667
	Total dedup. (sec)	280	152	476	228	5,824	3,968
	Throughput speedup	×1.8		×2.1		×1.5	
256	False-positive rate	0.00		3.92×10^{-5}		1.17×10^{-6}	
	On-disk lookup (sec)	175	21	335	47	4,620	1,694
	Prefetching (sec)	46	61	52	69	298	669
	Total dedup. (sec)	278	152	488	229	5,825	4,003
	Throughput speedup	×1.8		×2.1		×1.5	
128	False-positive rate	2.69×10^{-6}		1.56×10^{-3}		$4.1, 2 \times 10^{-5}$	
	On-disk lookup (sec)	176	21	736	67	5,077	1,752
	Prefetching (sec)	46	61	52	70	299	692
	Total dedup. (sec)	281	153	908	253	6,314	4,055
	Throughput speedup	×1.8		×3.6		×1.6	
64	False-positive rate	1.14×10^{-4}		3.31×10^{-2}		9.20×10^{-4}	
	On-disk lookup (sec)	185	21	9,553	660	15,421	2,953
	Prefetching (sec)	46	61	52	79	310	1,081
	Total dedup. (sec)	290	153	10,156	886	17,149	5,785
	Throughput speedup	×1.9		×11		×3.0	
32	False-positive rate	3.36×10^{-3}		2.62×10^{-1}		9.21×10^{-3}	
	On-disk lookup (sec)	410	35	100,369	6,292	113,334	10,915
	Prefetching (sec)	46	63	54	115	338	1,430
	Total dedup. (sec)	527	171	105,398	6,866	119,855	14,571
	Throughput speedup	×3.1		×15		×8.2	

dataset with more unique data. Table VIII lists runtime measurements showing the time spent on each component and the improvement in terms of speedup in deduplication time.

When the size of the Bloom filter is sufficiently large, the false-positive rate is low, so there are fewer disk accesses, so the difference between eager and lazy is narrow. With a smaller Bloom filter, we have a higher false-positive rate and consequently more disk accesses. The time spent on on-disk fingerprint lookup increases with the growth of the false-positive rate for both eager and lazy methods as both need to search for the fingerprints in the case of a cache miss. However, since the lazy method merges on-disk lookup, it takes much less time, so it can better handle situations where the Bloom filter has a large number of false positives.

Due to its small data size, a substantial false-positive rate does not appear on Vm until we cut the size of the Bloom filter to 128MB. Though FSLHomes has a much larger total size than Src, it has a similar unique data size (Src: 352GB, compared to FSLHomes: 330GB). Src has far fewer duplicate fingerprints, so Src shows a higher false-positive rate.

In the eager method, the prefetching time does not increase with the false-positive rate. This is because when a unique fingerprint is misclassified by the Bloom filter, the eager method just searches for it on the disk, which only introduces addition disk I/O for on-disk fingerprint lookup. As the fingerprint will not be found on the disk,

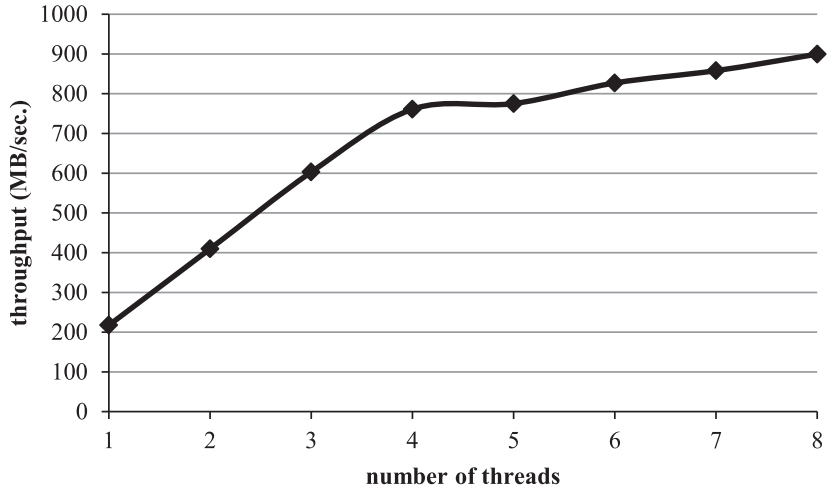


Fig. 13. Chunking throughput with different number of threads.

Table IX. GPU SHA1 Speedup

Device	CPU (serial)	GPU
Throughput (MB/sec)	338	1,563

prefetching is not triggered, so the prefeching time does not change. However, in the lazy method, the prefetching time increases with the false-positive rate. Since we buffer the fingerprints, if there are many unique fingerprints in the buffer, the true duplicates have to wait for slots. As a result, duplicate fingerprints are more sparsely spread throughout the buffer cycles, and fewer duplicate fingerprints are found after prefetching is triggered. This results in prefetching being called more often and a greater prefetching time.

Nevertheless, even with an increasing prefetching time, the lazy method is still beneficial due to the reduction of costly on-disk fingerprint lookups. For example, when the size of the Bloom filter is set to 32MB, the lazy method can get a $\times 15$ improvement compared with the eager method on Src (despite prefetching time doubling in going from eager to lazy).

5.8. Parallel Computation

We vary the number of threads from one to eight and test the throughput of chunking. Figure 13 shows the speedup. The speed grows approximately linearly as the number of threads increases from one to four. However, only a minor improvement in throughput is noticeable as the number of threads increases from four to eight, due to the CPU having four physical cores (supporting eight logical processors). We still see an improvement from four threads to eight threads, so we use eight threads when testing the overall throughput.

We use a GPU to accelerate fingerprinting (SHA1 hash calculation), and Table IX shows the improvement. We find that SHA1 on the GPU is 4.6 times faster than a single thread on CPU.

5.9. Throughput

Here we compare the throughput of lazy deduplication and eager deduplication on our SSD and HDD. 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 consists of 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 datasets.

Figure 14 shows the experimental results on our SSD. We see that lazy deduplication gives an improvement in the final throughput over eager deduplication of 80%, 65%, and 46% for Vm, Src, and FSLHomes, respectively. The lazy method achieves a greater throughput improvement versus 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% higher than the eager method on the SSD. 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.

The results on our HDD are shown in Figure 15. The improvements in the final throughput over eager are 150%, 119%, and 79% for Vm, Src, and FSLHomes, respectively. On HDD, the throughput is initially limited by chunking, 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 versus the SSD. This is due to the HDD having a much higher latency than the SSD. For FSLHomes, since there is no chunking or fingerprint calculation, the overall performance is limited by the disk I/O. The lazy method achieves 76% higher throughput than the eager method, which is greater than that on SSD due to higher disk access latency.

6. CONCLUDING REMARKS

In this article, 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 space. We also design a caching strategy that 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 tradeoff in the context of lazy deduplication.

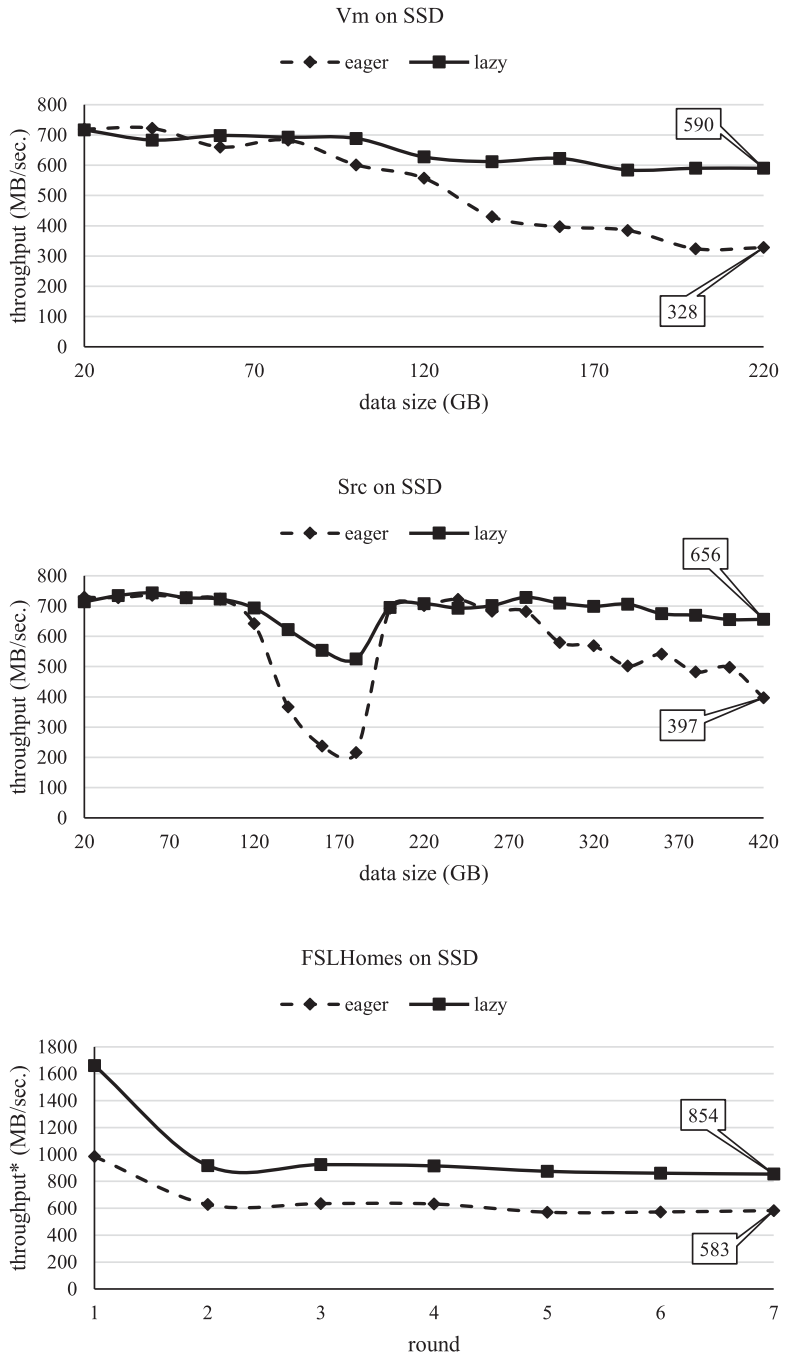


Fig. 14. Deduplication throughput on an SSD. *Throughput for FSLHomes does not include chunking and fingerprint calculation time.

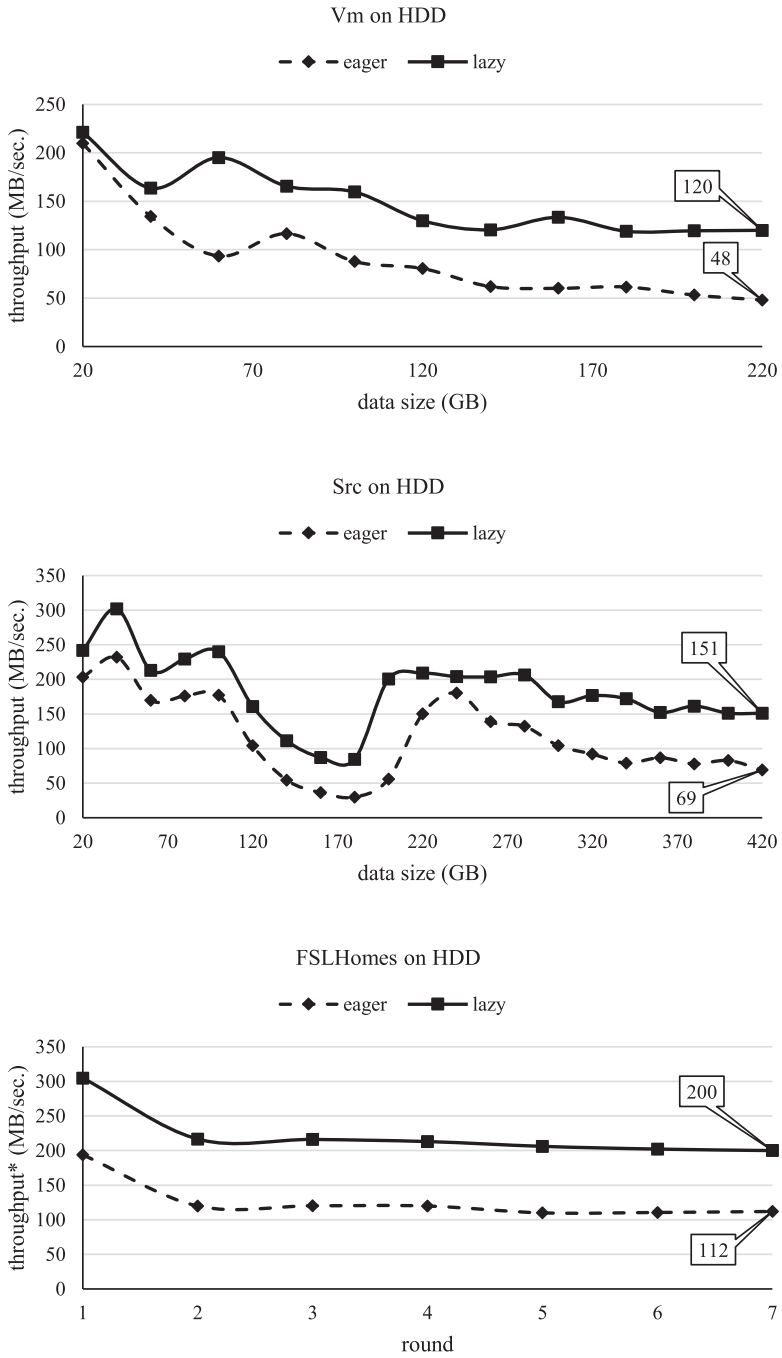


Fig. 15. Deduplication throughput on an HDD. *Throughput for FSLHomes does not include chunking and fingerprint calculation time.

—It would also be worthwhile to explore the compatibility of lazy deduplication with commonly used data storage methods (e.g., RAID), to see when it is most effective.
—The computation and I/O components are not completely balanced, so there is room for improvement via more sophisticated load-balancing methods, for example, by moving part of chunking computation to the GPU.

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.

REFERENCES

D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. 2009. 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'09)*. 1–9.

P. Bhatotia, R. Rodrigues, and A. Verma. 2012. Shredder: GPU-accelerated incremental storage and computation. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'12)*. 1–15.

B. H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang. 2008. On the false-positive rate of bloom filters. *Inform. Process. Lett.* 108, 4 (2008), 210–213.

F. C. Botelho, P. Shilane, N. Garg, and W. Hsu. 2013. Memory efficient sanitization of a deduplicated storage system. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'13)*. 81–94.

A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li. 2009. Decentralized deduplication in SAN cluster file systems. In *Proceedings of USENIX Annual Technical Conference (ATC'09)*. 101–114.

B. Debnath, S. Sengupta, and J. Li. 2010. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proceedings of USENIX Annual Technical Conference (ATC'10)*.

B. Debnath, S. Sengupta, and J. Li. 2011. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of ACM International Conference on Management of Data (SIGMOD'11)*. 25–36.

M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu. 2014. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *Proceedings of USENIX Annual Technical Conference (ATC'14)*. 181–192.

F. Guo and P. Efstathopoulos. 2011. Building a high-performance deduplication system. In *Proceedings of USENIX Annual Technical Conference (ATC'11)*. 25–25.

K. Jin and E. L. Miller. 2009. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of ACM Israeli Experimental Systems Conference (SYSTOR'09)*, Vol. 7.

C. Kim, K.-W. Park, K. Park, and K. H. Park. 2011. Rethinking deduplication in cloud: From data profiling to blueprint. In *Proceedings of IEEE International Conference on Networked Computing and Advanced Information Management (NCM'11)*. 101–104.

J. Kim, C. Lee, S. Lee, I. Son, J. Choi, S. Yoon, H. u. Lee, S. Kang, Y. Won, and J. Cha. 2012a. Deduplication in SSDs: Model and quantitative analysis. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST'12)*. 1–12. DOI:http://dx.doi.org/10.1109/MSST.2012.6232379

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

R. Koller and R. Rangaswami. 2010. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Trans. Storage* 6, 3 (2010), 13.

H. Li, T. Liang, and J. Chiu. 2013. A compound OpenMP/MPI program development toolkit for hybrid CPU/GPU clusters. *J. Supercomput.* 66, 1 (2013), 381–405.

X. Li and D. J. Lilja. 2009. 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'09)*. 268–275.

M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. 2009. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'09)*. 111–123.

X. Lin, F. Douglis, J. Li, X. Li, R. Ricci, S. Smaldone, and G. Wallace. 2015a. Metadata considered harmful... to deduplication. In *Proceedings of USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'15)*. 11–16.

X. Lin, M. Hibler, E. Eide, and R. Ricci. 2015b. Using deduplicating storage for efficient disk image deployment. *EAI Endorsed Trans. Scalable Information Systems* 2, 6 (2015), e1.

G. Lu, Y. J. Nam, and D. H. C. Du. 2012. 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'12)*. 1–11.

J. Ma, R. J. Stones, Y. Ma, J. Wang, J. Ren, G. Wang, and X. Liu. 2016. Lazy exact deduplication. In *Proceedings of IEEE 32th Symposium on Mass Storage Systems and Technologies (MSST'16)*. 1–10.

L. Ma, C. Zhen, B. Zhao, J. Ma, G. Wang, and X. Liu. 2010. Towards fast de-duplication using low energy coprocessor. In *Proceedings of IEEE International Conference on Networking, Architecture and Storage (NAS'10)*. 395–402.

U. Manber. 1994. Finding similar files in a large file system. In *Usenix Winter*, Vol. 94. 1–10.

B. Mao, H. Jiang, S. Wu, Y. Fu, and L. Tian. 2014. Read-performance optimization for deduplication-based storage systems in the cloud. *ACM Trans. Storage (TOS)* 10, 2 (2014), 6:1–6:22.

D. Meister and A. Brinkmann. 2010. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *Proceedings of IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST'10)*. 1–6.

D. T. Meyer and W. J. Bolosky. 2011. A study of practical deduplication. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'11)* (2011), 1–13.

D. T. Meyer and W. J. Bolosky. 2012. A study of practical deduplication. *ACM Trans. Storage (TOS)* 7, 4 (2012), 14:1–14:20.

J. Min, D. Yoon, and Y. Won. 2011. Efficient deduplication techniques for modern backup operation. *IEEE Trans. Comput.* 60, 6 (2011), 824–840.

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

NVIDIA. 2013. NVIDIA CUDA. Retrieved from https://developer.nvidia.com/cuda-downloads (July 2013).

R. Pagh and F. F. Rodler. 2001. *Cuckoo Hashing*. Springer.

J. Paulo and J. Pereira. 2014. A survey and classification of storage deduplication systems. *ACM Comput. Surv. (CSUR)* 47, 1 (2014), 11:1–11:30.

C. Policroniades and I. Pratt. 2004. Alternatives for detecting redundancy in storage systems data. In *Proceedings of USENIX Annual Technical Conference (ATC'04)*. 73–86.

S. Quinlan and S. Dorward. 2002. Venti: A new approach to archival data storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'02)*, Vol. 4. 89–102.

M. O. Rabin. 1981. *Fingerprinting by Random Polynomials*. Center for Research in Computing Technology, Aiken Computation Laboratory.

P. Shilane, M. Huang, G. Wallace, and W. Hsu. 2012. WAN-optimized replication of backup datasets using stream-informed delta compression. *ACM Trans. Storage (TOS)* 8, 4 (2012), 13:1–13:26.

K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. 2012. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'12)*. 24–24.

Y. Tan, Z. Yan, D. Feng, X. He, Q. Zou, and L. Yang. 2015. De-Frag: An efficient scheme to improve deduplication performance via reducing data placement de-linearization. *Cluster Comput.* 18, 1 (2015), 79–92.

V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok. 2012. Generating realistic datasets for deduplication analysis. In *Proceedings of USENIX Annual Technical Conference (ATC'12)*. 261–272.

VIA Technologies. 2008. VIA nano processor. Retrieved from http://www.viatech.com.cn/cn/downloads/white papers/processors/WP080529VIA_Nano.pdf.

M. M. Waldrop. 2016. The chips are down for moores law. *Nature* 530 (2016), 144–147.

W. Xia, H. Jiang, D. Feng, and Y. Hua. 2015. Similarity and locality based indexing for high performance data deduplication. *IEEE Trans. Comput.* 64, 4 (2015), 1162–1176.

W. Xia, H. Jiang, D. Feng, and H. Yu. 2011. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proceedings of USENIX Annual Technical Conference (ATC'11)*. 26–28.

B. Zhu, K. Li, and H. Patterson. 2008. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'08)*. 269–282.

Received November 2016; revised February 2017; accepted March 2017