Towards Optimizing Deduplication on Persistent Memory

Yichen Li, Kewen He, Gang Wang, Xiaoguang Liu

Towards Optimizing Deduplication on Persistent Memory*

Yichen Li, Kewen He, Gang Wang ✉, and Xiaoguang Liu ✉

College of Computer Science, Nankai University, Tianjin, china
{liyc,hekw,wgzwp,liuxg}@nbjl.nankai.edu.cn

Abstract. Data deduplication is an effective method to reduce data storage requirements. In data deduplication process, fingerprint identification may cause frequent on-disk fingerprint lookups which hurt performance seriously. Some locality-aware approaches were proposed to tackle this issue. Recently, the Persistent Memory (PM) brings low latency and high bandwidth, and has become a hotspot in data storage. Deduplication systems with fingerprints stored on PM will provide extremely fast on-disk fingerprint lookup, and therefore traditional locality-aware approaches designed for slow devices are likely no longer valid.
In this paper, we model the traditional locality-aware approaches and analyze their performance on PM. Inspired by the analysis, we propose an optimized PM-based fingerprint identification scheme in which the fingerprint cache is replaced with a simple, low-cost read buffer, and the order of the Bloom filter and the read buffer is swapped. The experimental results on real PM devices show that, compared with the traditional locality-aware approaches, the proposed scheme improves the fingerprint identification throughput by 1.2-2.3 times.

Keywords: Data deduplication · Persistent Memory · fingerprint identification · Bloom filter · cache

1 Introduction

Data deduplication is a popular method for data reduction, which can greatly save storage space and network bandwidth by eliminating duplicate data. Typically, data deduplication consists of four steps: data chunking, fingerprinting, fingerprint identification, and data storing. Specifically, a deduplicated system firstly divides the data stream into data chunks (normally 4KB-128KB), and calculates their unique fingerprints using a cryptographically secure hash signature (e.g. MD5, SHA-1, SHA-256), and then identifies redundant chunks by comparing their fingerprints and finally, unique data will be stored.

Fingerprint identification is a key step in deduplication and is also an I/O intensive step. In general, DRAM is not large enough to hold all the fingerprints,

* This work is partially supported by National Science Foundation of China (U1833114, 61872201, 61702521); Science and Technology Development Plan of Tianjin (18ZXZNGX00140, 18ZXZNGX00200).

and hence we must store them on disk. Therefore, on-disk lookup for fingerprints will become a serious bottleneck of the entire system. Locality-aware approaches such as DDFS[1] were proposed to reduce the accesses to low-speed disk by using Bloom filter and cache, which can achieve a very high cache hit rate due to good temporal and/or spatial locality in data streams. This kind of approach has been widely used in deduplication systems to improve fingerprint identification performance in which fingerprints are stored on HDD or SSD.

Recently, the development of persistent memory (PM) has attracted a lot of attention. The latency of PM is several orders magnitude lower than those of SSD and HDD. Therefore, if we build deduplication systems by storing fingerprints on PM rather than SSD and HDD, looking up fingerprints will take much less time. Traditional locality-aware approaches designed for slow devices like HDD and SSD aiming to reduce on-disk lookup may be not effective for deduplication systems based on PM. In such systems, the performance bottleneck may be transferred from disk I/O to other parts.

In this paper, we model the traditional locality-aware approach and analyze its performance on PM. Inspired by the analysis, we propose two optimization strategies for deduplication with PM. The contributions of this paper are as follows.

- We model the typical locality-aware fingerprint identification approach and analyze the impact of PM on its performance.
- We find out that the overhead of fingerprint cache is two heavy for deduplication with PM and propose to replace it with a simple and low-cost read buffer to exploit the spatial locality in data streams more effectively.
- We find out that Bloom filter becomes a significant performance bottleneck of fingerprint lookup and propose to adjust the order of Bloom filter and read buffer for highly duplication data to overcome this problem.
- We conduct experiments on real-world datasets and show that the proposed optimizations improves the throughput by 1.2-2.3 times.

The remainder of this paper is organized as follows. Section 2 surveys the related work and Section 3 presents our motivation. Section 4 models the fingerprint identification and analyze the performance on PM. Section 6 presents the experimental results of different fingerprint identification algorithms. Section 7 concludes the paper.

2 Background and Related Work

2.1 Data deduplication

Since data deduplication is generally applied to where a huge amount of data is involved, we have to store all fingerprints on disk rather than in DRAM. Therefore, fingerprint identification may cause frequent accesses to low-speed disks, which becomes a severe bottleneck of data deduplication. There are many

researches focusing on this problem [1,2,3,4,5,6], in which a number of locality-aware approaches are proposed. It is found that redundant data tend to reappear as similar sequences in real-world workload, which is called the spatial locality. Data Domain Deduplication File System (DDFS) [1] is one of the earliest deduplication systems exploiting the spatial locality in workload. DDFS uses Bloom filters and Locality Preserved Caching (LPC) to reduce disk I/O. Since traditional cache algorithms do not work well for caching fingerprints, LPC is designed to exploit the spatial locality in workload to eliminate the disk I/O bottleneck in deduplication. By combining Bloom filter and LPC, DDFS reduces 99% of disk accesses in fingerprint identification.

Lazy Exact Deduplication [6] is another locality-aware approach. Deduplication systems such as DDFS look up fingerprints on disk whenever a cache miss occurs, so they are called "eager" deduplication in [6]. The lazy deduplication, however, buffers the incoming fingerprints and then searches for them on disk in batch. The basic idea is to combine multiple fingerprint lookup into one singer on-disk lookup. The lazy strategy can significantly reduce the number of on-disk lookups but increases the overhead of in-memory cache operations. In general, because most data streams to be deduplicated have excellent locality, locality-aware approaches are widely used in deduplication [1][6][8].

2.2 Persistent Memory

Persistent Memory (PM) provides low latency, high bandwidth, persistence and byte-addressability, and thus has attracted a lot of attention to exploring its performance. In recent years, there have been a lot of studies on programming models [9], file systems [10], data structures [11], and key-value store [12] on PM. Table 1 shows the performance of PM devices. Recent search [14] shows that real PM device, Intel Optane DC Persistent Memory Module[13], exhibits 100∼300ns random or sequential read latency which is 2∼4x longer than DRAM's. Considering the limited write endurance and capacity of persistent memory, data deduplication may become an important part of PM in the future.

Table 1. Storage device latency and bandwidth

Device	Latency	bandwidth
DRAM	$60ns$	64GB/s
PM	300ns - 1μs	5 - 10GB/s
SSD	50μs - 80μs	250MB/s
HDD	$10ms$	2.6MB/s

NV-dedup [15] implements an in-line deduplication system based on the Persistent Memory File System (PMFS). It proposes a fine-grained metadata table and lightweight consistency strategy for metadata store in persistent memory. Regardless of the DRAM capacity, NV-dedup indexing all fingerprints in DRAM, and therefore the spatial locality in data streams is not considered in this work. However, with the help of the traditional locality-aware approaches, we can improve the performance of fingerprint identification on a huge amount of data.

3 Motivation

For deduplication systems storing fingerprints on PM, due to extremely low access latency, the traditional locality-aware approaches designed for slow devices may be no longer suitable now. To better understand the effect of the traditional approaches on PM, we conducted experiments on Intel's Optane PMM, as the details are shown in Section 6. We have the following observations.

Fingerprint lookup on PM is very fast. Compared with HDD, the overall fingerprint identification time is reduced by more than 90% with PM. Moreover, it is very fast to look up fingerprints on PM, which is only 3% of the total time. Even if we directly look up fingerprints on PM without any filtering and caching, the performance is much better than that on HDD

Traditional locality-aware approaches become less effective. With the traditional locality-aware approaches, the fingerprint identification time is reduced at most by 40% on PM. However, for HDD, it can improve performance by hundreds of times [1]. In other words, the traditional locality-aware approaches still work with PM, but far less effective than with HDD.

Bloom filter and cache become the main bottlenecks. The Bloom filter and cache can avoid 99% on-disk lookup, which is very important for deduplication systems using HDD. However, they take over 90% of the overall time when storing fingerprints on PM, which become the new bottlenecks in deduplication.

4 Modeling and Analysis

In this section, we focus on locality-aware approaches for deduplication, which adopts Bloom filter and locality-preserved cache. Specifically, those kinds of approaches first check whether the fingerprint is duplicate by using a Bloom filter. A negative answer means that the fingerprint is *definitely* not in the fingerprint set, and a positive answer indicates that it is *very likely* in the set so that a cache lookup needs to be performed next. However, false positives exist, which will cause unnecessary cache misses and disk accesses, just to find that the fingerprints are unique. Cache missing will cause a fingerprint lookup on disk. If the fingerprint exists, some of its neighbor fingerprints will be prefetched to the cache together to increase the cache hit rate. According to the results of Bloom filter lookup, fingerprint identification can be divided into three cases.

a) Negative: A negative answer given by the Bloom filter means that the data chunk is unique and no further search in cache and on disk is needed. Therefore we have Eq.(1) that shows the negative search time T_n. N_n is the number of negative answers and L_B is the latency of a single lookup in Bloom filter. Bloom filter can greatly reduce access to fingerprints index on disk by identifying most unique data chunks.

$$T_n = N_n \times L_B. \tag{1}$$

b) True positive: True positives indicate that the data chunks are actually duplicate. The time T_{tp} of this case is shown in Eq.(2). $N_{tp}(= N_d)$ is both the

number of false positive answers and the number of duplicate fingerprints. L_C and L_D are the latency of a single lookup in cache and on disk respectively. R_{miss} refers to the cache miss rate. Therefore, the last two terms of Eq.(2) denote the time taken by looking up fingerprints further in cache and on disk respectively. We will analyze the impact of cache on performance later in this section.

$$T_{tp} = N_{tp} \times L_B + N_d \times L_C + N_d \times R_{miss} \times L_D. \tag{2}$$

c) False positive: Here, the data blocks are unique but judged to be duplicate. Therefore, we still need to search their fingerprints further in cache and on disk although it is impossible to find them. False positives waste a lot of time and hence should be reduced by increasing the size of Bloom filter. The lookup time T_{fp} of false positives is given by,

$$T_{fp} = N_{fp} \times (L_B + L_C + L_D) = N_u \times R_{fp} \times (L_B + L_C + L_D), \tag{3}$$

where N_{fp} $(= N_u \times R_{fp})$ refers to the false positive rate of Bloom filter, N_u is the number of unqiue fingerprints. Therefore, if the total fingerprints is N $(= N_u + N_d)$, the total time for fingerprint identification is,

$$\begin{aligned} T =& T_n + T_{tp} + T_{fp} \\ =& N \times L_B + (N_u \times R_{fp} + N_d) \times L_C + (N_u \times R_{fp} + N_d \times R_{miss}) \times L_D. \end{aligned} \tag{4}$$

We can see from Eq.(4) that the main effect of Bloom filter on the fingerprint identification time is the lookup latency L_B and the false positive rate R_{fp}. The larger the two factors are, the longer the identification time is. L_B is determined by the hash computation used in the Bloom filter, and R_{fp} mainly depends on the relative size of the Bloom filter to the data. For the systems storing fingerprints on HDD and SSD, since L_B and L_C is much lower than the disk access latency L_D, so previous work concerning fingerprint identification optimization mainly focused on decreasing R_{fp}, that is, decreasing the number of disk accesses. However, we can't infinitely increase the size of the Bloom filter to decrease R_{fp}.

When we store fingerprints on PM, L_D becomes very close to L_B and L_C, and thus Eq.(4) is no longer dominated by the third term. To optimize fingerprint identification, we need to find a good trade-off between the time taken by the Bloom filter itself (the first term) and the time saved by its low false positive rate (the last two terms). A big and complex Bloom filter will decrease the false positive rate effectively and then reduce in-cache and on-disk lookup time, but the improvement may be offset by its own cost. According to our experimental results in Section 6, as the size of Bloom filter grows, the fingerprint identification time doesn't change appreciably. This indicates that a smaller Bloom filter is enough for deduplication with PM.

Similarly, the cache algorithms also take time themselves, especially the complicated ones. When HDD/SSD is used to store fingerprints, the on-disk lookup time saved by cache is much higher than that spent in cache lookup, and therefore a cache is necessary for fingerprint identification. However, when PM is

introduced, we must carefully trade off between the two parts of time (the second and the third terms in Eq.(4)). An effective but high-cost cache algorithm may not suitable for deduplication with PM. For example, in [6], a highly effective cache algorithm is proposed. However, our experimental results show that, compared with another simpler but faster cache algorithm, it causes performance degradation in PM-based systems because of its high overhead. We found that even a simple cache algorithm like LRU is too heavy for fingerprint identification with PM. This inspires us to turn to a lightweight approach to exploit spatial locality in data streams, which may slightly increase the third term in Eq.(4) but significantly decrease the second term.

5 Design Choices

According to previous analyses, we can draw the following conclusions helpful to design an optimized fingerprint identification method with PM.

- Lookup in Bloom filter takes much time, and thus reducing the amount of work performed on Bloom filter may improve the overall performance.
- Cache lookup and cache prefetching latency are important now. Cache algorithms such as the one used in lazy exact deduplication are too heavy for deduplication with PM.

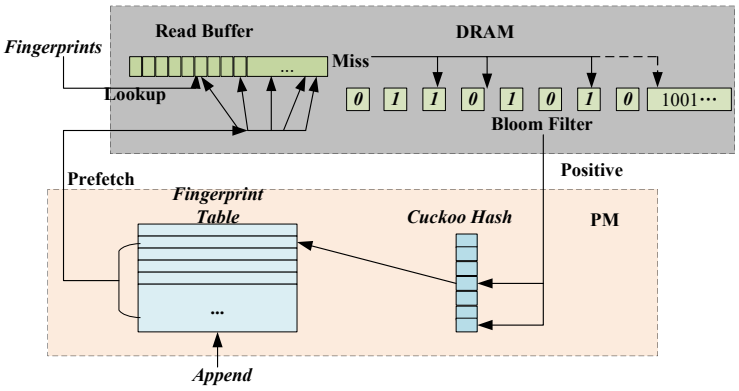


Fig. 1. Design with PM

Inspired by these conclusions, we proposed a fingerprints identification framework as Fig. 1 shows, in which we adopt the following design choices.

a) Read buffer: We replace the relatively heavy fingerprint cache with a simple, low-cost read buffer to exploit the spatial locality in workload. We use a simple hash table to implement the read buffer. When we find a duplicate fingerprint on PM, we load its subsequent fingerprints together to DRAM (in our experiments, 50 neighbor fingerprints are prefetched). If the buffer is full, the oldest fingerprints in the buffer will be replaced without using any cache eviction algorithm. So a prefetching is quite light, only involves one *memcpy* from PM

to DRAM. No other operations and data structures are involved to increase the hit rate. A buffer lookup needs only a simple hash table search.

In general, traditional cache algorithms such as LRU are designed under the consideration of the temporal locality, which needs dedicated data structures and operations to record the least recently used information to determine which old item should be replaced. But a simple read buffer doesn't have these overhead. Despite sacrificing a certain hit rate, it still improves performance by saving the above costs. Our experimental results show that read buffer can still obtain a hit rate of about 90% which indicates effective exploitation of the spatial locality in data streams. Moreover, its own cost is much lower than the traditional cache, and then the overall performance is improved.

b) Log-structured fingerprints table: To preserve the spatial locality of the data layout, we use a log-structured fingerprints store on PM. A cuckoo hash table indexed by fingerprints is also deployed on PM. A newly arrived fingerprint is appended to the fingerprints store. Therefore, for adjacent data chunks in the data stream, their fingerprints are also neighbors in the log-structured store. We can just prefetch consecutive fingerprints from PM to DRAM to exploit the spatial locality, and only a single *memcpy* operation needs to be performed, which is very fast. The cuckoo hash provides fast fingerprint lookups.

c) Adjusting the order of operations: After replacing the heavy cache with a lightweight read buffer, we found that the time spent in this part becomes much shorter than that taken by Bloom filter. This observation inspires us to swap the order of the read buffer and the Bloom filter. Although this brings more lookups in the read buffer, the additional time contributes little to the overall time. Meanwhile, the work on Bloom filter is cut down drastically, which may reduce the overall time,

$$T^{'} = N \times L_C + (N_u + N_d \times R_{miss}) \times L_B + (N_u \times R_{fp} + N_d \times R_{miss}) \times L_D, \quad (5)$$

where we ignore the difference between the miss rates of the two procedures here for simplicity. Compared with Eq.(4), we can see that the performance is improved, that is $T^{'} < T$, if and only if the following inequality holds,

$$N \times L_C + (N_u + N_d \times R_{miss}) \times L_B < N \times L_B + (N_u \times R_{fp} + N_d) \times L_C, \quad (6)$$

that is,

$$\frac{L_C}{L_B} < \frac{N_d \times (1 - R_{miss})}{N_u \times (1 - R_{fp})} \approx \frac{N_d \times (1 - R_{miss})}{N_u} \approx \frac{R_d \times (1 - R_{miss})}{1 - R_d}, \quad (7)$$

where R_d ($\approx \frac{N_d}{N}$) is the duplication rate of the dataset. So, if the duplication rate R_d meets:

$$R_d > \frac{L_C}{L_B \times (1 - R_{miss}) + L_C}, \quad (8)$$

adjusting the order of the read buffer and Bloom filter can effectively improve the performance. The larger R_d is, the more effective the new order is.

6 Experimental Results

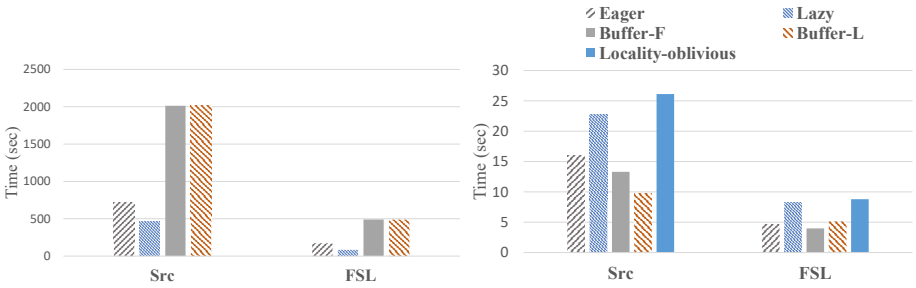
6.1 Experiment Settings

We conducted our experiments on a machine equipped with two 18-core Intel Xeon Gold 5220 processors run at 2.2GHz. The machine has 4 Intel Optane DC persistent memory (128GB each) installed in socket 0. The ext4-DAX file system is mounted on the PM devices. A Seagate 4TB HDD is also used for comparison. All the code is compiled using GCC 4.8.5. We use PMDK to map files into virtual address space and use `clflush` and `mfence` to persist fingerprints to PM.

We used two datasets to evaluate our deduplication methods. *Src* refers to the source code of CentOS, Fedora, Ubuntu, etc., which is collected from a server at Nankai University[1]. The total size is 272.12GB and the duplication rate is 53.12%. *FSL* refers to the snapshots of students' home directories published by the File system and Storage Lab (FSL) at Stony Brook University [16]. We randomly select a part of data within 7-day intervals from the year 2011 and 2014[2]. The total size is 101.037GB and the duplication rate is 20.69%.

We implemented the traditional lazy and eager methods for comparison. We implemented our new design for PM based on the eager method but replace the fingerprint cache with a simple read buffer and swap the order of the Bloom filter and the read buffer. We call it the buffer method. We only measure and compare the fingerprint identification time.

6.2 Overall Performance

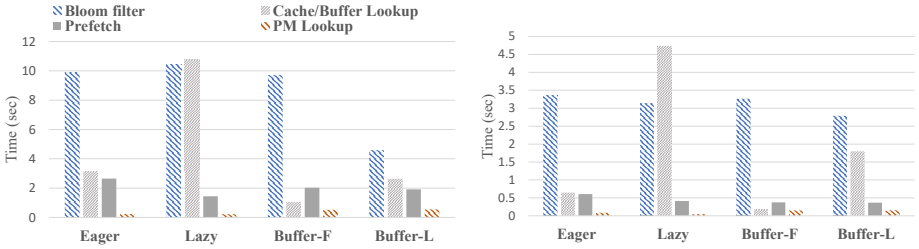


(a) Finger identification time on HDD (b) Finger identification time with PM

Fig. 2. Fingerprint Identification time

We first evaluate the fingerprint identification time on HHD and PM. To ensure a very low false positive rate, the Bloom filter uses 6 hash functions and has a size of 1GB. Fig. 2 shows the results. ***Locality-oblivious*** refers to the method that searches the fingerprints directly on PM. ***Buffer-F*** refers to the method that adopts a lightweight read buffer. ***Buffer-L*** is based on Buffer-F and further swaps the order of the read buffer and Bloom filter.

[1] http://ftp.nankai.edu.cn/
[2] http://tracer.filesystems.org/



(a) Src (b) FSL

Fig. 3. Breakdown of Fingerprint Identification Time with PM

From Fig. 2(a) we can see that, when fingerprints are stored on HDD, both locality-aware methods perform better than the buffer method because of a much higher cache hit rate and the lazy method performs the best because it significantly reduces disk I/O through batch lookup. The buffer method is 3-4 times slower than the locality-aware methods. When we store fingerprints PM, the overall fingerprint identification time is reduced by up to 97.23%, which mainly due to the extremely low latency of PM. The locality-aware method still works with PM and reduces at most 32% of time compared with ***Locality-oblivious***. However, the gap is much narrower than that with HDD. For both datasets on PM, the buffer method performs best and the lazy method performs worst (excluding Local-oblivious), which is consistent with our analysis. For dataset *Src* with a higher duplication rate, ***Buffer-L*** performs best and is 1.7-2.3 times faster than the locality-aware methods and 2.7 faster than Local-oblivious. However, for *FSL* dataset with a low duplication rate, ***Buffer-L*** is not so effective and even slower than the eager method. ***Buffer-F*** performs best, which is 1.2-2.1 times faster than the locality-aware methods. It is worth noting that the lazy method takes nearly the same time as ***Locality-oblivious***, which is mainly due to the high cost of the cache algorithm.

Fig. 3 shows the breakdown of fingerprint identification time. Different from fingerprint identification on HDD, lookup on PM only accounts for 2% of the overall time. Lookup in Bloom filter and cache takes up 52.8% and 44.8% of the total time respectively in ***Eager***. With a read buffer, lookup on PM takes a little more time but the time spent in cache is effectively reduced. Although a certain hit rate is sacrificed, the low latency of the read buffer significantly reduces the overall time. For dataset *Src*, adjusting the order of the Bloom filter and read buffer can effectively reduce the time cost in the Bloom filter but increases a small amount of in-cache lookup time. As a result, the overall time is effectively reduced. However, for the dataset *FSL*, adjusting the order may cause performance degradation. For data with a low duplication rate, the Bloom filter is much more important than the read buffer. Adjusting the order reduces few lookups in the Bloom filter but increases much more lookups in read buffer.

To summarize, with fast PM, the fingerprint identification process takes much less time. Traditional locality-aware approaches still work but the performance bottleneck is transferred from disk I/O to in-memory operations. To further optimize fingerprint identification, a low-cost read buffer should be used in dedu-

plication systems. Other methods such as adjustment of the order between the Bloom filter and read buffer are effective for some specific workloads.

6.3 Quantitative analysis

Table 2. performance details of different methods

Dataset	Src		FSL	
Method	Eager	Buffer	Eager	Buffer
R_{fp}	1.5%		1.1%	
R_{hit}	98.65%	94.89%	99.17%	95.16%
L_B	$0.1531\mu s$	$0.1517\mu s$	$0.139\mu s$	$0.141\mu s$
L_C	$0.14241\mu s$	$0.0366\mu s$	$0.13457\mu s$	$0.0378\mu s$
L_D	$0.38901\mu s$ (PM)			

We tested the values of the parameters involved in our analytical model described in Section 4 with two datasets on our experimental platform. Table 2 shows the results. For convenience, we rewrite Eq. (4) as,

$$T = N \times L_B + N_d \times L_C + N_u \times R_{fp} \times (L_C + L_D) + N_d \times R_{miss} \times L_D. \quad (9)$$

With fixed L_D, the first and second terms respectively refer to the contribution of the lookup latency in the Bloom filter and cache on the overall time. The third and fourth terms refer to the effect of the cache hit rate and false positive rate respectively. We substitute the parameter values of the buffer and eager methods with dataset *src* into Eq. (9), and we obtain,

$$T_{PM}^{Eager} = 0.15 \times N + 0.14 \times N_d + 0.0079 \times N_u + 0.00525 \times N_d, \quad (10)$$

$$T_{PM}^{Buffer} = 0.15 \times N + 0.03 \times N_d + 0.00615 \times N_u + 0.019 \times N_d. \quad (11)$$

The coefficients of the third and fourth terms in Eq.(10) is much smaller than those of the first and second terms, which means that the lookup latency in the Bloom filter and Cache becomes the main factor affecting the overall fingerprint identification time. In contrast, in Eq. (11), although a certain amount of extra PM accesses is introduced (that is, the fourth term is increased), the lookup time in cache/buffer is greatly reduced (that is, the second term is effectively reduced). As a result, identification for duplicate fingerprints takes much less time. This conclusion is also true for the dataset *FSL*, what's more, for datasets with a higher duplication rate, the read buffer is more effective.

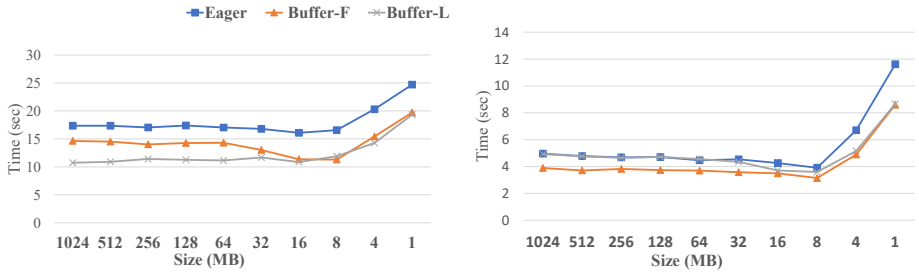
It worth noting that reordering the Bloom filter of read buffer is effective for dataset *Src* but less effective for dataset *FSL*, we substitute the values of L_B , L_C and R_{hit} into Eq.(8),

$$\frac{L_C}{L_B \times (1 - R_{miss}) + L_C} = \frac{0.0378}{0.141 \times 0.9516 + 0.0378} = 21.9\%. \quad (12)$$

which means that the reordering strategy improves the performance only when the duplicate rate of the dataset is higher than 21.9%. However, the duplicate

rate of *FSL* is smaller than 21.9%, therefore reordering does not work here. Since the hit rate and lookup latency may change with different datasets, swapping the order between the Bloom filter and read buffer is more suitable for datasets with high duplication rate.

6.4 Bloom Filter Size



(a) Src (b) FSL

Fig. 4. Impact on Bloom filter size

As we analyzed in section IV, the Bloom filter can effectively filter out the unique data chunks. However, a low false positive rate may not be essential for deduplication systems with PM. We explore how the size of the Bloom filter impacts the performance. We can see from Fig. 4 that ***Buffer-L*** performs best on *Src* and ***Buffer-F*** performs best on *FSL* at all Bloom filter sizes. When the Bloom filter is large enough (32MB-1024MB), the false-positive rate is low. Therefore the performance gap between different methods is narrow, and the main factor that affects the overall performance is the read buffer or cache.

However, when the Bloom filter is between 32MB and 8MB, the overall fingerprint identification time ***Eager*** and ***Buffer-F*** is reduced as the size of the Bloom filter becomes smaller. We found that the Bloom filter smaller than 32 MB achieves a lower lookup latency, it may benefit from CPU L3 Cache. That is, a smaller Bloom filter leads to less CPU cache misses. Although on-PM lookup time increases, due to a large number of lookup on the Bloom filter becomes faster, the overall performance gets much improvement. As a result, With a size of 8MB, ***Buffer-F*** and ***Buffer-L*** perform the same with an extremely low lookup latency in Bloom filter. When the Bloom filter is much smaller (8MB-1MB), due to the extremely high false positive rate, it needs much more PM accesses to identifies the fingerprints, resulting in severe performance degradation.

In summary, the fingerprint identification with PM is not sensitive to the size of the Bloom filter, which is consistent with our modeling. A small and fast bloom filter may perform better in deduplication systems.

7 Conclusion

In this paper, we model the typical locality-aware fingerprint identification approach to better understand the performance of the deduplication system with

PM. With PM, the identification time becomes extremely low, the hit rate of cache and the positive rate of the Bloom filter is less important for deduplication. We propose a simple and low-cost read buffer which obtains better performance by exploiting the spatial locality of data and adjusting the order of Bloom filter and read buffer to improve the performance of deduplication. Experiment results show that the read buffer based method effectively saves the fingerprint identification time. It is 1.2x-2.3x faster than the traditional locality-aware approach.

References

1. Zhu, B., Li, K., & Patterson, R. H. (2008, February). Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In Fast (Vol. 8, pp. 1-14).
2. Xia, W., Jiang, H., Feng, D., & Hua, Y. (2011, June). SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In USENIX ATC (pp. 26-30).
3. Bhagwat, D., Eshghi, K., Long, D. D., & Lillibridge, M. (2009, September). Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In Proc. MASCOTS 2009 (pp. 1-9). IEEE.
4. Debnath, B. K., Sengupta, S., & Li, J. (2010, June). ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In USENIX ATC (pp. 1-16).
5. Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezis, G., & Camble, P. (2009, February). Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In Fast (Vol. 9, pp. 111-123).
6. Ma, J., Stones, R. J., Ma, Y., Wang, J., Ren, J., Wang, G., & Liu, X. (2017). Lazy exact deduplication. ACM Transactions on Storage (TOS), 13(2), 1-26.
7. Meister, D., Kaiser, J., & Brinkmann, A. (2013, June). Block locality caching for data deduplication. In Proc. Fast. (pp. 1-12).
8. Yang, Jian, et al. "An empirical guide to the behavior and use of scalable persistent memory." In Proc. FAST. (2020)
9. Rudoff, A. (2017). Persistent memory programming. Login: The Usenix Magazine, 42(2), 34-40.
10. Xu, J., & Swanson, S. (2016). NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In 14th USENIX Conference on File and Storage Technologies (FAST 16) (pp. 323-338).
11. Nam, M., Cha, H., Choi, Y. R., Noh, S. H., & Nam, B. (2019). Write-optimized dynamic hashing for persistent memory. In 17th USENIX Conference on File and Storage Technologies (FAST 19) (pp. 31-44).
12. Lepers, B., Balmau, O., Gupta, K., & Zwaenepoel, W. (2019, October). KVell: the design and implementation of a fast persistent key-value store. In Proceedings of the 27th ACM SOSP (pp. 447-461).
13. Beeler, B. (2019). Intel optane dc persistent memory module (pmm).
14. Yang, Jian, et al. "An empirical guide to the behavior and use of scalable persistent memory." In Proc. Fast. (2020).
15. Wang, Chundong, et al. "Nv-dedup: High-performance inline deduplication for non-volatile memory." IEEE Transactions on Computers 67.5 (2017): 658-671.
16. Tarasov, V., Mudrankit, A., Buik, W., Shilane, P., Kuenning, G., & Zadok, E. (2012). Generating realistic datasets for deduplication analysis. In USENIX ATC (pp. 261-272).