HPDedup: A Hybrid Prioritized Data Deduplication Mechanism for Primary Storage in the Cloud

Huijun Wu[†§], Chen Wang[*], Yinjin Fu[‡], Sherif Sakr[*†], Liming Zhu[*†], Kai Lu[§]
[*]*Data61, CSIRO*
[†]*The University of New South Wales, Australia*
[‡]*PLA University of Science and Technology, China*
[§]*Science and Technology on Parallel and Distributed Laboratory,*
State Key Laboratory of High Performance Computing,
State Key Laboratory of High-end Server & Storage Technology,
College of Computer, National University of Defense Technology, China

***Abstract*—Eliminating duplicate data in primary storage of clouds increases the cost-efficiency of cloud service providers as well as reduces the cost of users for using cloud services. Most existing primary deduplication techniques either use inline caching to exploit locality in primary workloads or use post-processing deduplication running in system idle time to avoid the negative impact on I/O performance. However, neither of them works well in the cloud servers running multiple services or applications for the following two reasons: Firstly, the temporal locality of duplicate data writes may not exist in some primary storage workloads thus inline caching often fails to achieve good deduplication ratio. Secondly, the post-processing deduplication allows duplicate data to be written to disks, therefore does not provide the benefit of I/O deduplication and requires high peak storage capacity. This paper presents HPDedup, a Hybrid Prioritized data Deduplication mechanism to deal with the storage system shared by applications running in co-located virtual machines or containers by fusing an inline and a post-processing process for exact deduplication. In the inline deduplication phase, HPDedup gives a fingerprint caching mechanism that estimates the temporal locality of duplicates in data streams from different VMs or applications and prioritizes the cache allocation for these streams based on the estimation. HPDedup also allows different deduplication threshold for streams based on their spatial locality to reduce the disk fragmentation. The post-processing phase removes duplicates whose fingerprints are not able to be cached due to weak temporal locality from disks. The hybrid deduplication mechanism significantly reduces the amount of redundant data written to the storage system while maintaining inline data writing performance. Our experimental results show that HPDedup clearly outperforms the state-of-the-art primary storage deduplication techniques in terms of inline cache efficiency and primary deduplication efficiency.**

***Keywords*-Data Deduplication; Cache Management; Primary Storage; Cloud Services**

I. Introduction

Data deduplication is a technique that splits data into small chunks and uses the hash fingerprints of these data chunks to identify and eliminate duplicate chunks in order to save storage space. Deduplication techniques have achieved great successes in backup storage systems [1]. However, significant challenges remain to apply deduplication techniques in primary storage systems mainly due to the low latency requirement in primary storage applications [2]. Recent studies show that duplicate data widely exists in the primary workloads [2] [3] [4]. In the cloud computing scenario, the primary workloads of the applications running on the same machine are observed having high duplicate ratio as well [5]. For a cloud datacentre, there are significant incentives to remove duplicates in its primary storage for cost-effectiveness and competitiveness.

The existing data deduplication methods for primary storage can be classified into two main categories based on when the deduplication is performed: *inline* deduplication techniques [6] [4] [5] and *post-processing* deduplication techniques [2] [7] [8]. The former performs data deduplication on the write path of I/O requests to immediately identify and eliminate data redundancy, while the latter removes duplicate data in background to avoid the performance impact on the I/O. However, challenges remain for both of these two methods.

For inline deduplication, fingerprint lookup is the main performance bottleneck due to that the size of a fingerprint table often exceeds the size of the memory. While a backup storage system may be able to tolerate the delay of disk based fingerprint lookup, the deduplication system of a primary storage system has to rely on caching to satisfy the latency requirement of applications. The state-of-the-art techniques for inline primary deduplication [6] [4] [5] exploit temporal locality of primary workloads by maintaining an in-memory fingerprint cache to perform deduplication. These deduplication mechanisms do not ensure that all duplicate chunks are eliminated. We call them *non-exact deduplication*. However, the temporal locality in primary workloads does not always exist [9] [10]. For the workloads with weak temporal locality, caching the unnecessary fingerprints not only wastes the valuable cache space but also compromises other workloads with good locality. iDedup [4] also exploits spatial locality to alleviate data

fragmentation on disks by only eliminating duplicate block sequences longer than a fixed threshold. However, when data streams from different sources have different spatial locality, a fixed threshold may fail to achieve good deduplication ratio or read performance. For primary storage in clouds, the differences of locality become a severe problem. Firstly, the weak temporal locality becomes more apparent in the cloud when multiple applications running in virtualized containers sharing the same physical primary storage system. Since deploying deduplication in each virtual machine often fails to detect the duplicates among different virtual machines, deduplication should be deployed at the host physical machine. The data streams from co-located VMs or applications may interfere with each other and destroy the temporal locality. It has significant impact on the fingerprint cache efficiency managed by existing caching policies. The stream interference problem has been addressed in backup deduplication by resorting the streams [11]. However, it is not applicable for primary storage systems because we cannot change the order of requests of primary workloads. Secondly, as shown in our experiments on real-world traces (in Section III), different workloads show quite different spatial locality. Therefore, a fixed global threshold is not optimal for alleviating the disk fragmentation in primary storage in the clouds.

For post-processing deduplication techniques [2] [7] [8]. There are two main drawbacks: Firstly, duplicate chunks are written to disks before being eliminated. This makes deduplication not effective in reducing peak storage use. For SSD based primary storage in cloud architecture like hyper-converged infrastructure, this affects the lifetime of SSD devices. Secondly, the competition between post-processing deduplication process and foreground applications on using resources such as CPU, RAM and I/O can be a problem when a large amount of duplicates has to be eliminated.

To avoid the limitations and exploit the advantages of inline and post-processing deduplication, in this paper, we fuse the two phases together and propose a hybrid data deduplication mechanism to particularly deal with deduplication in virtualized systems running multiple services or applications from different cloud tenants. The goal is to achieve a good balance between I/O efficiency and storage capacity saving in primary storage deduplication. In the inline deduplication phase, we differentiate the temporal locality of different data streams using a histogram estimation based method. The estimation method periodically assesses the temporal locality of the data streams from different services/applications. Based on the estimation, we propose a cache replacement algorithm to prioritize fingerprint cache allocation to favor data streams with good temporal locality. The mechanism significantly improves cache efficiency in inline deduplication and reduces the workload in the post-processing deduplication phase. Moreover, we adjust the threshold for different data streams dynamically to alleviate the disk fragmentation while achieving high inline deduplication ratio. The post-processing deduplication phase only deals with relatively small amount of duplicate data blocks that are missed in cache in the inline deduplication phase. Compared to systems that purely rely on post-processing deduplication, a highly efficient inline deduplication process greatly reduces the storage capacity requirement and contention in system resources.

Overall, this paper makes the following main contributions:

1) We propose a novel hybrid deduplication mechanism that fuses inline deduplication and post-processing deduplication together for primary storage systems shared by multiple applications/services. The mechanism is able to provide exact deduplication in comparison to many inline deduplication mechanisms while avoiding drawbacks of purely post-processing deduplication mechanisms.
2) We give a locality estimation based cache replacement algorithm that significantly improves the fingerprint cache efficiency in primary storage deduplication systems. The estimation method is able to exploit locality in individual data streams for cache hit rate improvement.
3) We evaluate our mechanism using traces generated from real-world applications. The result shows that the proposed mechanism outperforms the state-of-the-art inline and post-processing deduplication techniques. e.g., HPDedup improves the inline deduplication ratio by up to 39.70% compared with iDedup in our experiments. It also reduces up to 45.08% disk capacity requirement compared with the state-of-the-art post-processing deduplication mechanism in our evaluation.

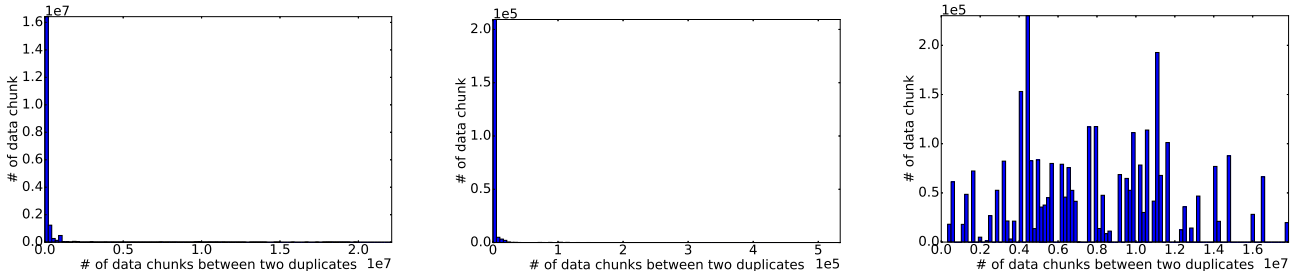
The remaining of this paper is organized as follows: Section II describes the background information and motivations for our approach; Section III presents the design of HPDedup; Section IV introduces how to differentiate the locality of data streams in deduplication; Section V presents the detailed results of our experimental evaluation; Section VI reviews related work and Section VII concludes the paper.

II. Background and Motivation

In this section, we present the background and key observations that motivate this work.

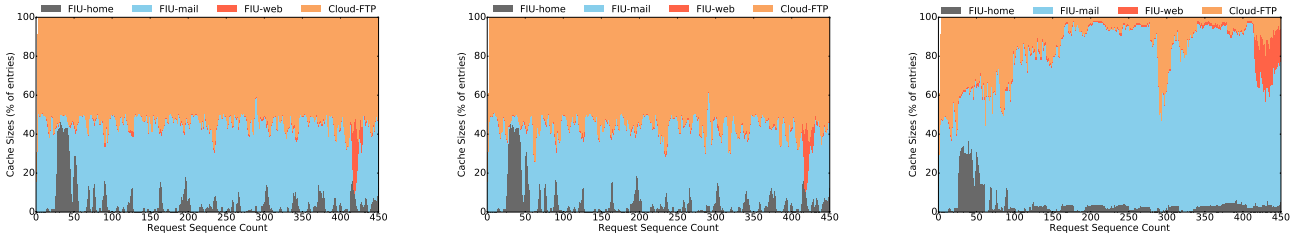
A. Deduplication for Primary Storage in Clouds

Virtualization enables a cloud provider to accommodate applications from multiple users to run on a single physical machine while providing isolation between these applications. Recently, container techniques like Docker [12] further reduce the overhead of using virtual machines to isolate user applications, thus supporting running more applications simultaneously on a physical machine.



(a) ***FIU-mail trace*** (b) ***FIU-web trace*** (c) ***Cloud-FTP trace.***

Figure 1: Temporal locality analysis for three I/O traces. The x-axis is the number of data blocks between two adjacent occurrences of the same data block. i.e., for a I/O sequence "abac", each letter represents a data block. The number of data blocks between two adjacent occurrence of "a" is 1.



(a) ***LRU*** (b) ***LFU*** (c) ***ARC***

Figure 2: The size of cache entries occupied by each data stream. The x-axis is the request sequence count while the y-axis is the percentage of cache occupied by different streams. The area of different colors indicates the cache resources used by each data stream.

Although providing much benefits in improving the resource sharing efficiency, increasing number of applications from different users sharing the same machine raises challenges to primary storage deduplication. In typical configurations, the cloud software stack such as OpenStack [13] maps the data volumes of VMs to persistent block storage devices connected by networks. It is impractical to achieve deduplication within a container or a virtual machine due to the overhead of storage device access. Moreover, it would fail to identify the duplicates among different VMs or containers. It is only feasible to detect duplication in the host's hypervisor that manages I/O requests from VMs or containers. As the file information in VMs or containers is not available in the underlying hypervisor, we design our deduplication mechanism to be based on the block layer in the hypervisor. A similar architecture has also been used in an existing post-processing primary deduplication technique [14].

B. Temporal locality Affects Efficiency of Fingerprint Cache

Existing primary storage deduplication techniques often exploit temporal locality through fingerprint cache in an attempt to detect most of duplicates in the cache. However, some recent studies reveal that the locality may be weak in the primary storage workloads [15].

We evaluate the temporal locality with real-world traces, which contain a 24-hour I/O trace from a file server running in the cloud as well as the FIU trace [16] commonly used in deduplication research. The file server is used for data sharing among a research group consisting of 20 people. We denote the file server trace as *Cloud-FTP*, and FIU mail server trace as *FIU-mail* and FIU Web server trace as *FIU-web*.

As shown in Figure 1, the average distance between two adjacent occurrences of a data block in both *FIU-mail* and *FIU-web* trace is small and highly skewed, indicating good locality. For the *Cloud-FTP* trace, the temporal locality is weak. The temporal locality of duplicates in primary storage systems varies among different applications.

We further evaluate the cache efficiency for the three different workloads when they arrive at a storage system within the same time frame. The cache replacement algorithms we use in our evaluation include LRU (Least Recently Used), LFU (Least Frequently Used) and ARC (Adaptive Replacement Cache). The three cache replacement policies exploit the recency, frequency and the combination of both of workloads, respectively.

We extract two-hour traces from the three FIU traces

Table I: Workload Statistics of the 2-hour traces.

Trace	Request number	Write request ratio	Duplicate writes
Cloud-FTP	2293424	84.15%	387140
FIU-mail	1961588	98.58%	1633424
FIU-web	116940	49.36%	30534
FIU-home	293605	91.03%	32688

(10am-12am on the first day.) and the Cloud-FTP trace we collect. The characteristics of the two-hour traces are shown in Table I. We mix these traces according to the timestamps of requests to simulate the I/O pattern of multiple applications on a cloud server. We set the cache size to 32K entries. Figure 2 illustrates the actual percentage of cache occupied by each data stream.

Table II: Duplicates detected under different cache replacement algorithms.

Cache Policy	FIU-home	FIU-mail	FIU-web	Cloud-FTP
LRU	20568	399622	16667	11977
LFU	19984	381157	16072	11072
ARC	22248	1119355	12245	467

Table II shows the number of duplicate blocks detected by each cache replacement algorithm. Under the LRU and LFU cache replacement algorithm, the cache allocated to Cloud-FTP stream is above 2/3 of the maximum cache capacity, but the number of duplicates detected in the stream is less than 2% of the total number of duplicates. Under the ARC cache replacement algorithm, the Cloud-FTP stream is allocated less portion of cache, however, only 467 duplicates are detected. This experiment shows that data streams with weak temporal locality of duplicates result in poor cache efficiency and fail to detect most of duplicates in the inline deduplication process.

When duplicates cannot be effectively detected through cache lookup, they are written to disks, which results in extra storage space requirement in capacity planning. It is therefore important to improve cache efficiency when locality of duplicates is not guaranteed.

III. The Design of HPDedup

The inline or post-processing deduplication alone is difficult to satisfy the deduplication ratio and latency requirement of a primary storage system. However, the two techniques complement each other. Fusing them together to form a hybrid deduplication system is able to achieve *exact deduplication* with satisfactory write performance. Particularly, the caching in inline deduplication not only speeds up fingerprint lookup, but also reduces the amount of data written to disks and relieves the burden of handling large amount of duplicates in the post-processing phase. On the other hand, the post-processing deduplication is able to detect duplicates missed out in the inline cache, therefore achieves *exact deduplication*. Including a post-processing phase potentially relaxes the inline cache size requirement as well.

In another word, a hybrid deduplication system is able to achieve a balance between the I/O performance and deduplication ratio, which is essential for inline deduplication of primary storage. This motivates the architecture design of HPDedup. In the following, we first give the hybrid architecture, and then describe the inline phase and post-processing phase of HPDedup.

A. HPDedup Architecture Overview

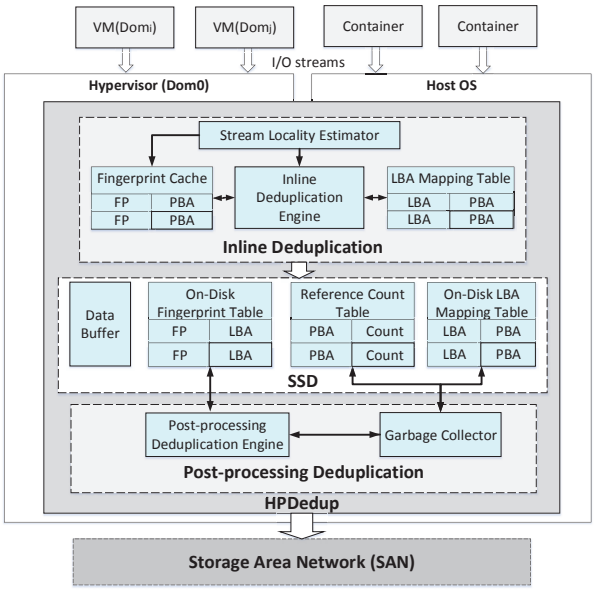


Figure 3: System architecture of HPDedup.

Virtualization is a core technology that enables the cloud computing. Running multiple virtual machines in a physical machine is a common practice in cloud datacentres. In cloud software stack such as OpenStack [13], the storage volumes of virtual machines are often mapped to persistent block storage devices. It is impractical to implement deduplication inside a virtual machine while data streams from co-located VMs are written to the same physical device. The main reason is that performing deduplication in each VM is not able to remove the duplicates across VMs. We therefore place our deduplication mechanism at the block device layer.

For the cloud scenario, the deduplication mechanism can be implemented inside the hypervisor that manages the I/O requests from VMs running on top of it. Some existing post-processing primary deduplication techniques [14] place their deduplication mechanisms at the same level.

Figure 3 illustrates the system architecture of HPDedup. A number of storage devices are connected to the server via SAN or through similar storage environments. The

hypervisor (e.g., Xen) is responsible for translating LBA (logical block address) to PBA (physical block address) for block I/O requests from VMs running on top of it. HPDedup works at the hypervisor level to eliminate duplicate data blocks. For multiple containers running on the same host, HPDedup can be deployed at the block device layer of the host machine. For simplicity, we will mainly use the hypervisor setting to describe the design of HPDedup in the rest of this paper.

B. Inline Deduplication Phase

In the inline deduplication phase, HPDedup maintains an *in-memory fingerprint cache* that stores the fingerprint and PBA mapping to avoid slow disk-based fingerprint table search, and an *LBA mapping table* that stores the mapping between LBAs and PBAs of blocks. The LBA mapping table is stored in NVRAM to avoid the data loss. The inline deduplication of data streams is performed in the *inline deduplication engine*: the fingerprint of each data block is computed by a cryptographic hash function, like MD5 or SHA-1. The deduplication engine then looks up the block fingerprint in the fingerprint cache. The *stream locality estimator* is responsible for monitoring and estimating both the temporal and spatial locality for the data streams coming from different VMs or containers. The temporal locality estimation is used for optimizing the hit rate of the fingerprint cache while the spatial locality estimation adjusts the deduplication threshold for data streams to reduce the disk fragmentation.

When the fingerprint of the incoming data block is found in the fingerprint cache, an entry of the LBA of the coming block and the corresponding PBA will be created and added into the LBA mapping table if such an entry does not exist, otherwise nothing is done because it is a duplicate write. If the block fingerprint is not found in the fingerprint cache, the data block is written to the underlying primary SAN storage. In this process, the data is staged in the data buffer in SSD for performance consideration. We use D-LRU [17] algorithm to manage the data buffer in SSD to store recently accessed data by exploiting temporal locality.

When a data block is written to the underlying primary storage, the corresponding metadata associated with this data block including its fingerprint, LBA and PBA mapping as well as the reference count is updated in three tables in the SSD: on-disk fingerprint table, on-disk LBA mapping table and reference count table. The duplicates whose fingerprints are not cached in fingerprint cache will be eliminated in the post-processing phase.

C. Post-Processing Deduplication Phase

In the post-processing deduplication phase, the post-processing deduplication engine scans the on-disk fingerprint table and identifies duplicates. Note that duplicates identified in this phase are not in the fingerprint cache. Therefore, they are not processed by inline deduplication phase. The entries with duplicate fingerprints are then removed while the corresponding LBAs are mapped to the same PBA in LBA mapping table. The reference count to the original PBAs containing the same data is decremented and the disk space is further claimed by the garbage collector. After post-processing, the unique data blocks in data buffer of SSD are organized into fixed-sized coarse-grained objects and flushed to the underlying persistent store.

IV. Differentiate Data Stream Locality in Deduplication

In this section we describe how to differentiate the temporal and spatial locality among different data streams to improve the efficiency of primary deduplication in the cloud. Both temporal and spatial locality estimation are performed in the stream locality estimator of the inline deduplication module. Specifically, the temporal locality of duplicates in a data stream is used to guide the allocation of fingerprint cache to the data stream in order to achieve higher inline deduplication ratio. The spatial locality of a stream is used to achieve a balance between the inline deduplication ratio and the read performance. We first describe how to measure and estimate the temporal locality of duplicates in data streams in IV-A, and then describe how to manage the fingerprint cache based on the temporal locality measurement in IV-B. Thirdly, we discuss how to handle the disk fragmentation based on the difference of spatial locality among data streams in IV-C.

A. Temporal Locality Estimation for data streams

The *temporal locality of duplicates* characterizes how soon duplicates of a data block may arrive in the system in a data stream. A good temporal locality indicates duplicate data blocks generally are close to each other while a weak locality indicates that duplicate blocks are often far away from each other or there are few duplicates in the data stream.

To measure the temporal locality of duplicates, we introduce a metric called *Local Duplicate Set Size (LDSS)*. *LDSS* of a stream is defined as the number of duplicate fingerprints in last n contiguous data blocks arriving before a given time. Here, we call n *estimation interval*.

To use *LDSS* to guide fingerprint cache allocation, we need to predict the *LDSS* of the future arrivals of data blocks of the data stream. A common approach is to use the historical *LDSS* values to predict the future *LDSS* of the data stream. To obtain a historical *LDSS* value from a data stream, a naive way is to count all distinct fingerprints for each data stream and their occurrences within an *estimation interval*. However, this incurs high memory overhead which is close to the cache capacity in the worst case because all the fingerprints need to be recorded. To address the problem, the *stream locality estimator* uses the reservoir sampling

algorithm [18] to sample fingerprints from a data stream, and then estimate *LDSS* from these samples using the unseen estimation algorithm [19].

Reservoir sampling algorithm assumes an unknown number of fingerprints in a data stream and guarantees that each fingerprint in the data stream has an equal chance to be sampled. In our implementation, each element in the *sampling buffer* is a pair containing a fingerprint and its occurrence count.

The unseen estimation algorithm is able to estimate the unseen data distribution based on the histogram of the samples of observed data. For HPDedup, the unseen estimation algorithm is used to estimate *LDSS* of data streams based the sampled fingerprints from these streams. We refer readers to [19] for more details about the theoretical aspect of estimation algorithms. Here, we give a high-level description of using the unseen estimation algorithm to estimate the *LDSS* values for a data stream.

Consider the storage system handles *M* data streams denoted by S_1,S_2,...,S_M from *M* VMs and the *estimation interval* size is *n*, the goal of the temporal locality estimation is to collect k fingerprint samples from the last n write requests of each stream and compute the *LDSS* values for these streams based on fingerprint samples.

Specifically, after sampling, we denote the number of sampled fingerprints coming from stream *i* by N_i. By using unseen estimation algorithm, we can accurately estimate the number of unique writes (denoted by u_i) in stream S_i among last n write requests in the mixed stream. Then the estimated *LDSS* for stream i can be denoted by $LDSS_i$ as below:

$$LDSS_i = N_i - u_i$$

whereas N_i is the total number of write requests for stream i in the *estimation interval*.

The estimation of u_i as well as the calculation of $LDSS_i$ is shown in Algorithm 1. Before discussing the algorithm, we introduce a concept named *Fingerprint Frequency Histogram (FFH)*. A *FFH* of a set of fingerprints F is a histogram $f = \{f_1, f_2...\}$ where f_j is the number of distinct fingerprints that appear exactly j times in F.

We derive the *FFH* from the *sampling buffer* to estimate the $LDSS_i$ of data stream i. We use H_s to denote the *FFH* of the samples and H to denote the *FFH* of the whole estimation interval for stream i. According to the unseen estimation algorithm, we then compute the transformation matrix T by a combination of binomial probabilities about the chances an item is drawn a certain times. The expected histogram $H_s^{'}$ for sampled data blocks can be computed by $H_s^{'} = T \cdot H$. To solve the equation and get H, we minimize the distance between H_s and $H_s^{'}$ which are the observed histogram and expected histogram of sampled data blocks, respectively. Once having obtained H, we are able to compute the $LDSS_i$ for the data stream.

Algorithm 1: Temporal Locality Estimation Algorithm

Input : H_s – The *FFH* for samples in stream i;
N_i – the number of write requests of stream i in the estimation interval.

Output: Estimated $LDSS_i$ of data stream i

Compute matrix T by binomial probabilities.
$H_s^{'}$ for samples is computed by $H_s^{'} = T \cdot H$
Linear programming:
  Objective function: $min(\Delta(H_s, H_s^{'}))$, in which $\Delta(H_s, H_s^{'}) = \Sigma_i \frac{1}{\sqrt{H_s[i]+1}} |H_s[i] - (T \cdot H)[i]|$.
  under constraints:
    $\Sigma_i H[i] = N$
    $\forall i \quad H[i] > 0$
  return $LDSS_i = N_i - \Sigma_i H[i]$

For some streams which have few write requests during the *estimation interval*, it is not necessary to run unseen estimation algorithm to estimate the *LDSS*. The *LDSS* of these streams are set to a small value for simplicity.

B. LDSS Estimation Based Fingerprint Cache Management

We use the *LDSS* of different streams to guide the cache allocation for these streams. The fingerprints from a data stream with higher predicted *LDSS* is more likely to be kept in the cache than those from a data stream with lower *LDSS*. As mentioned earlier, we use the historical *LDSS* values which are accurately estimated by the unseen algorithm to predict the *LDSS* of streams. We use a self-tuned double exponential smoothing method to predict the *LDSS* values. Using the estimated $LDSS(w-2), LDSS(w-1), ...$, we can predict $LDSS(w)$ where w is the next *estimation interval*. The predicted *LDSS* is used to guide the fingerprint cache management as follows.

Firstly, we propose a cache admission policy that the fingerprints from streams with very low *LDSS* would not be cached if there exists streams with much higher *LDSS*. This strategy can avoid caching the fingerprints of data streams containing compressed data or other forms of compact data. The cache of each stream can be managed by any cache replacement policies.

Secondly, for the fingerprints already cached, we assign an evict priority value p_i to data stream i, denoted by:

$$p_i = \frac{1}{LDSS_i(w)}$$

The evict priorities are mapped to adjacent non-overlapping segments in a segment tree. Specifically, stream i is represented by the segment $[\Sigma_{k=0}^{i-1} p_k,\ \Sigma_{k=0}^{i-1} p_k + p_i)$. When evicting a fingerprint from the cache, we generate a random number r and find the segment I to which r belongs. We then evict one cache entry from the cache corresponding to interval I.

As the fingerprint cache management of HPDedup relies on the accurate *LDSS* estimation of using unseen estimation algorithm, one may also think about directly estimating the *LDSS* of data streams by the number of duplicate fingerprint samples sampled by reservoir sampling. However, since whether a sampled fingerprint is duplicate is not independent, this method is not feasible. Figure 4 compares the effectiveness of using RS-only (Reservoir sampling only, dash lines) or RS + Unseen (Reservoir sampling with unseen algorithm, solid lines) during the *LDSS* estimation. It is clear that RS + Unseen based *LDSS* estimation is able to provide much higher inline deduplication ratio with a smaller estimation interval compared with RS-only method. This shows the effectiveness of temporal locality estimation using the unseen algorithm.

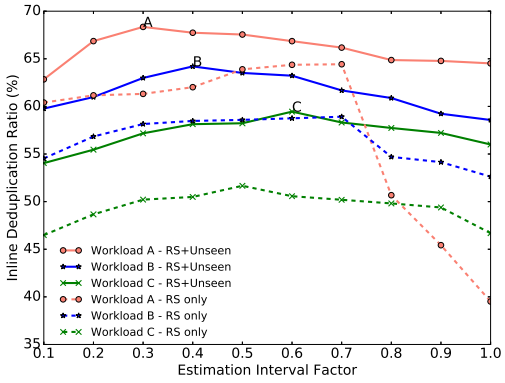


Figure 4: Inline Deduplication ratio vs. Estimation Interval Factor for three different workloads.

Moreover, a proper value of *estimation interval* is important for achieving the good cache efficiency. Too large interval may include some out-dated information which cannot reflect the current temporal locality of duplicates for the workloads. Too small interval, on the other hand, cannot accurately capture the temporal locality. Since *LDSS* is used to estimate the number of duplicates which can be detected in the fingerprint cache, the *estimation interval* can be set to a factor of the number of fingerprint cache entries. The solid lines in Figure 4 shows the inline deduplication ratio of HPDedup for three different workloads (details are shown in Section V) while choosing different *estimation interval factor*. The cache size is set to 160MB. From the workload A to C, the overall temporal locality for the workload decreases. The *estimation interval factor* needs to be set to larger values for the workloads with worse temporal locality. Correspondingly, we can see that the optimal *estimation interval factor* for workload A, B and C are around 0.3, 0.4 and 0.6, respectively. In practice, a good approximation is to set the *estimation interval factor* to 1 - d where d is the historical inline deduplication ratio for the mixed streams.

The temporal locality estimation is triggered by the following three events: 1. the finish of an *estimation interval*; 2. a significant drop of inline deduplication ratio; 3. the join or quit of virtual machines/applications.

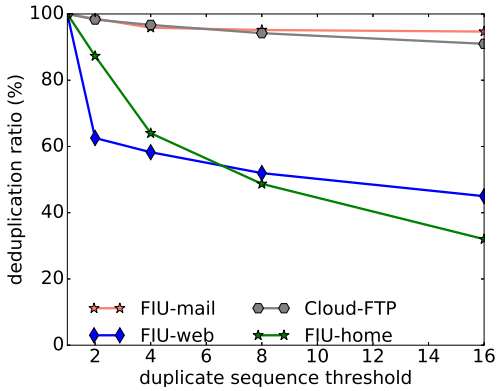


Figure 5: Deduplication ratio vs. Threshold. Deduplication ratio versus threshold for different threshold of duplicate sequence length.

C. Spatial Locality Aware Threshold for Deduplication

To alleviate disk fragmentation problem of deduplication for data reads, some primary storage deduplication techniques only eliminate duplicate block sequences with length greater than a given *threshold*. However, as pointed by [4], for the applications with many small random I/Os, doing so may not find any duplicates. In the deduplication of primary storage systems in the cloud, the spatial locality of data streams for different applications/services varies significantly. To explore the relationship between deduplication ratio and threshold, we analyze both the FIU traces and the trace we collect. As shown in Figure 5, different workloads show different trends. When the threshold increases from 1 to 16, the inline deduplication ratio for FIU-mail and Cloud-FTP reduces by only 4.3% and 9.1%, respectively. The inline deduplication ratio for FIU-web drops by around 38.1% when the threshold increases from 1 to 2. When the threshold is 16, the inline deduplication ratio is 43.1% of that under a threshold of 1. For FIU-home trace, the inline deduplication ratio keeps dropping. When the threshold is set to 16, the inline deduplication ratio is only 32.0% of that under a threshold of 1 .

Figure 5 indicates that the threshold value should be adaptive to data stream characteristics. It is noteworthy that there exists a tradeoff between the write latency and read latency while choosing a proper threshold. Write operations prefer shorter threshold while read operations prefer longer threshold. For writes, long sequence indicates more comparisons before writing data blocks to disks. For reads, long threshold can avoid many random I/Os thus reducing the read latency.

HPDedup maintains two vectors V_w and V_r for each stream. V_w is used to store the occurrence number for the

largest length values of sequential duplicates. V_r is used to store the occurrence number for the length values for sequential read. Both V_w and V_r have 64 items. For instance, if $V_w[3] = 100$, there are 100 sequential duplicates with length 3 since V_w is reset. If $V_r[3] = 100$, there are 100 sequential reads with length 3 since V_r is reset.

Initially, the threshold is 16. The two vectors collect data when requests come. When the threshold update is triggered, given the histogram vector V_w and V_r, the threshold T is computed by

$$T = (1 - r) \cdot \overline{Len_d} + r \cdot \overline{Len_r}$$

where $\overline{Len_d}$ and $\overline{Len_r}$ are the average length of duplicate block sequence and average read length, respectively. T, therefore, is the balance point of the read and write latency. r is the read ratio among all requests. $\overline{Len_d}$ and $\overline{Len_r}$ are computed according to the data collected in V_w and V_r, respectively. To cope with the changes of duplicate pattern for each stream, the two vectors is reset to all 0s when the total deduplication ratio decreases by over 50% since the last threshold update.

V. Evaluation

The prototype of HPDedup is implemented in C. To evaluate the performance of HPDedup, we use real-world traces to feed into HPDedup. We compare the performance of HPDedup with the following deduplication methods: locality based inline deduplication (iDedup [4]), post-processing deduplication schema (e.g., [2] [14]) and hybrid inline-offline deduplication schema DIODE [20].

A. Configuration

The experiments are carried out in a workstation with Intel Core i7-4790 CPU , 32GB RAM and 128GB SSD + 1TB HDD. We use the *FIU-home, FIU-web, FIU-mail* [16] traces in our evaluation. These traces are from three different applications, namely remote desktop, web server and mail server respectively in FIU. Moreover, we also collect a trace from a cloud FTP server (Cloud-FTP) used by our research group. The trace is obtained by mounting a network block device (NBD) as the working device for the FTP server, from which we capture read/write requests through a customized NBD-server.

To the best of our knowledge, there is no available larger scale I/O traces containing both the fingerprint of data blocks and timestamps. We therefore use the four traces as templates to synthetically generate VM traces representing multiple VMs. Table III shows the statistics of the four workloads. The arrival order of requests in these workloads are sorted and merged based on timestamps. The generated trace has the same I/O pattern with the original traces. For the traces generated from the same template, the content overlap is randomly set to 0% - 40% which is the typical amount of data redundancies shared among users [21].

Table III: Workload Statistics

Trace	Num of requests	Write request ratio	Duplicate ratio
Cloud-FTP	21974156	83.94%	20.77%
FIU-mail	22711277	91.42%	90.98%
FIU-web	676138	73.27%	54.98%
FIU-home	2020127	90.44%	30.48%

In our experiments, we simulate a cloud host running 32 virtual machines. As shown in Figure 1, FIU traces show better temporal locality compared with Cloud-FTP trace. We mix traces to form three workloads with different overall temporal locality. Workload A contains 15 mail server traces, 5 FTP server traces, 8 remote desktop traces and 4 web server traces. Workload B contains 10 mail server traces, 10 FTP server traces, 6 remote desktop server traces and 6 web server traces. Workload C contains 5 mail server traces, 15 FTP server traces, 6 web server traces and 6 remote desktop server traces. The ratios of data size between the good-locality (L) traces and bad-locality (NL) traces are around 3:1, 1:1 and 1:3 for these three mixed workloads.

The *estimation interval factor* is set to 0.5 at the beginning and is adjusted by the historical inline deduplication ratio dynamically for each workload.

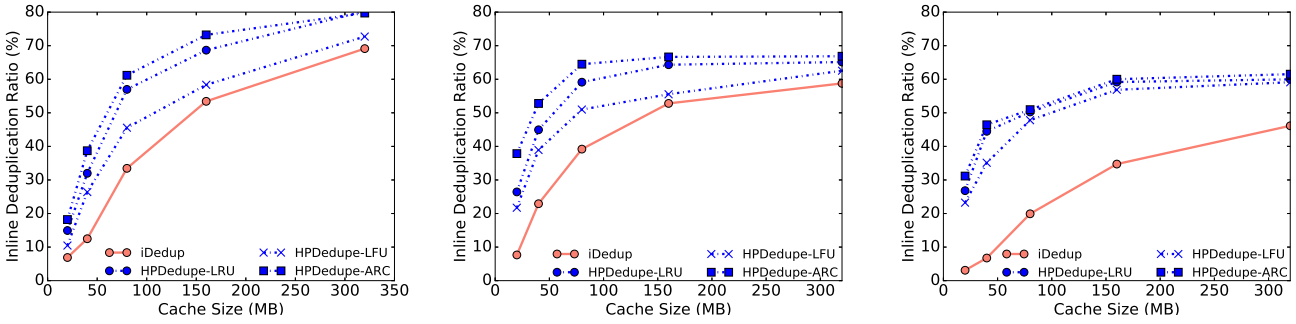
B. Cache Efficiency in Inline Deduplication

We compare HPDedup with iDedup, a well-known inline deduplication system that makes use of temporal locality of primary workload. We replay the mixed workloads to simulate the scenario where multiple applications/services run on the same physical machine. Each write I/O request for the three workloads is a 4KB block and MD5 is used as the hash function to calculate the fingerprints.

Each entry of the deduplication metadata is about 64 bytes and contains the fingerprint and the block address. According to the size and footprint of the traces, the total memory size for fingerprint cache is set from 20MB to 320MB in the experiments. The deduplication threshold is set to 4 for both iDedup and HPDedup.

Figure 6 shows the inline deduplication ratio versus cache size for iDedup and HPDedup. Here, inline deduplication ratio is defined as the percentage of duplicate data blocks that can be identified by inline caching. For the cache replacement policy of each stream, LRU, LFU and ARC cache replacement policies are supported by HPDedup. LRU was claimed to be the best cache replacement policy by iDedup [4].

When the portion of NL workload increases, the gap between iDedup and HPDedup becomes larger. HPDedup-LRU, HPDedup-LFU and HPDedup-ARC improve the inline deduplication ratio significantly compared with iDedup. For workload A, HPDedup-ARC improves the inline deduplication ratio by 10.58% - 27.72%. HPDedup-LRU improves the inline deduplication ratio of iDedup which also uses LRU



(a) ***Workload A (L:NL = 3:1)*** (b) ***Workload B (L:NL = 2:2)*** (c) ***Workload C (L:NL = 1:3)***

Figure 6: Inline deduplication ratio vs. Cache size for iDedup and HPDedup with different cache replacement policies.

cache by 8.04% - 23.52%. HPDedup-LFU shows less improvement (3.56%-13.90%) compared with HPDedup-ARC and HPDedup-LRU. Similarly, for workload B, HPDedup-ARC, HPDedup-LRU and HPDedup-LFU achieve 8.09%-30.19%, 6.36%-22.02% and 3.77%- 16.02% improvement of inline deduplication ratio, respectively. For workload C, HPDedup-ARC improves the inline deduplication ratio by 15.38% - 39.70%. HPDedup-LRU and HPDedup-LFU achieve 13.86% - 37.75% and 12.97% - 28.37% improvement, respectively. The improvement is larger when the cache size is small due to cache resource contention. Moreover, HPDedup-ARC outperforms HPDedup-LRU and HPDedup-LFU because ARC cache replacement policy makes the size of T1 (LRU) cache and T2 (LFU) cache adaptive to the recency and frequency of the workloads. It is also noteworthy that with the increasing of non-locality workload share (from Workload A to C), the inline deduplication ratio improvement achieved by HPDedup becomes larger. The reason is that non-locality workloads provide more space for the optimization of HPDedup.

Note that for HPDedup-ARC, extra memory overhead is introduced by ARC caching replacement policy itself to track the evicted fingerprints and their metadata. The overhead is non-trivial for fingerprint cache. The analysis of overhead for obtaining statistics information about evicted fingerprints in existing cache replacement policies is out of the scope of this paper. We leave the discussion to the future work. In the following experiments, HPDedup-LRU is used by default.

Overall, the weak locality in workloads results in low inline deduplication ratio. **With the locality estimation method in HPDedup, the allocation of inline fingerprint cache dynamically gives the streams with better locality higher priority. Hence, HPDedup improves the overall cache efficiency for multiple VMs/applications running on the same physical machine in the cloud.**

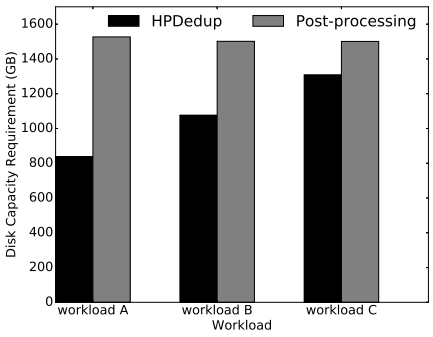


Figure 7: The disk capacity requirements for HPDedup and post-processing deduplication schemas.

C. Disk Capacity Requirement

In this subsection, we compare the size of the data before performing post-processing deduplication for HPDedup and pure post-processing deduplication (e.g., [2]). The size is also the maximum required disk size for the deduplication mechanisms. Figure 7 shows the result of comparisons.

For HPDedup, the inline fingerprint size is set to 160MB and LRU cache replacement policy is used for simplicity. As shown in Figure 7, HPDedup significantly reduces the disk capacity requirements for storage space. The data size has been reduced by 45.08%, 28.29% and 12.78% for workload A, B and C, respectively. The better the locality is in the workload, the more duplicates can be detected in the inline phase and the more duplicate data writes can be eliminated. **This clearly shows the benefit of a hybrid deduplication architecture of HPDedup as hundred GBs of data writes can be reduced by only maintaining a 160MB inline fingerprint cache.**

D. Average Hits of Cached Fingerprints

Inline deduplication of HPDedup does not require disk access to identify duplicates therefore is faster than the post-processing based deduplication. We use *average hits of*

cached fingerprints as an indicator of inline deduplication performance. The indicator is obtained by monitoring the number of fingerprint entering the fingerprint cache. With a high average hits value, the inline deduplication is able to detect a large portion of duplicates and reduces the load of the more expensive post-processing deduplication.

In the following, we compare HPDedup with DIODE [20] on this metric. DIODE uses file extensions to decide whether to perform inline deduplication on these files. Files are classified roughly into three different types. The inline deduplication process skips the type of files containing audio, video, encrypted and other compressed data (called P-Type in DIODE). We use a full inline deduplication method [4] as the baseline.

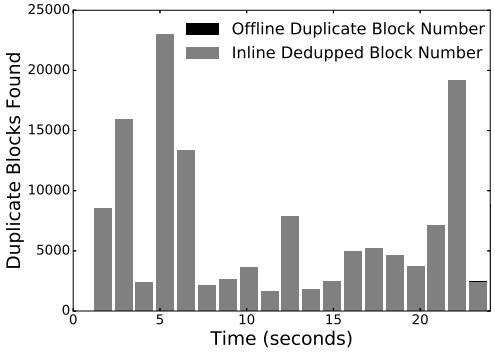
DIODE works at the file system level so that the information like file extensions are passed to the hypervisor layer in the form of hints, like the method used by [22]. The file type information of the Cloud-FTP trace is known so that the trace can be used to test DIODE. The files that are classified as P-Type are around 14.2% of the whole trace in size. The FIU traces are classified into U-Type (unpredictable type), which will be processed by the inline deduplication, just like that in the evaluation setting of DIODE in [20].

Table IV: The *Average Hits of Cached Fingerprints* for baseline, DIODE and HPDedup.

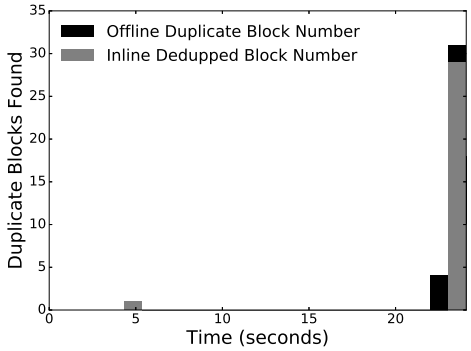
Schema	Cache Size	Workload A	Workload B	Workload C
Baseline	320MB	1.301	0.665	0.204
	160MB	0.781	0.551	0.148
DIODE	320MB	1.437	0.812	0.247
	160MB	0.805	0.598	0.181
HPDedup	320MB	1.812	4.024	5.918
	160MB	1.409	3.101	5.002

As shown in Table IV, HPDedup clearly outperforms the baseline and DIODE on the average hits of cached fingerprints. HPDedup outperforms DIODE by identifying data streams that have weak locality but do not belong to P-Type. It then avoids allocating cache space to these streams in order to make room for data streams with good locality.

This approach is effective. To show this, we compare the locality of the following two files in the Cloud-FTP workload: a Linux kernel 4.6 source code tar file and a VM image of CentOS 5.8 downloaded from OSBoxes. The two files are similar in size (2.7GB and 2.6GB). They are written to the primary storage after an inline deduplication process. The fingerprint cache size is set to 1% of the data size (27MB and 26MB). Figure 8 shows the number of duplicate blocks found in the two files. Note that for the VM image, nearly all duplicate blocks can be found through the inline deduplication process. DIODE treats both files as highly-deduplicatable (H-Type in DIODE). However, the number of duplicate blocks in the VM image file is significantly



(a) ***CentOS VM image***



(b) ***Linux Source Code***

Figure 8: The duplicate blocks found in the VM image and Linux Kernel Source Code tar file. The x-axis is the time during writing the files while the y-axis is the number of duplicate blocks.

higher than that in the source code tar file. Moreover, DIODE ignores all P-Type files during inline deduplication by letting them to be processed during inline deduplication. However, multiple writes of the same P-Type files result in duplication and cannot be eliminated through differentiating file types. Different from DIODE, HPDedup allocates much less cache to the Cloud-FTP stream while writing the Linux Source Code tar file but allocates more when writing the VM image file.

The result shows that simply using file type to guide cache allocation is insufficient. **HPDedup classifies data at finer-grained (stream temporal locality level) so that the efficiency of inline deduplication can be further improved.**

E. Locality Estimation Accuracy

HPDedup improves the efficiency of inline deduplication phase by allocating the fingerprint cache based on the temporal locality of each stream. Since *LDSS* is an indicator which describes the temporal locality of data streams, it is critical to achieve accurate *LDSS* estimation.

Figure 9 shows the observed *LDSS* for workload B. Here, the cache size is set to 160MB. To make the figures

concise, the traces generated from the same template are aggregated. Figure 9a shows the observed *LDSS* over time. The values of *LDSS* are normalized. FIU-mail streams show the largest *LDSS* which indicates that it has the best temporal locality. Nevertheless, as shown in Figure 9b, very little cache resource is occupied by FIU-mail streams when *LDSS* estimation is not used. Cloud-FTP streams whose *LDSS* is not high occupy the majority of cache resources. As shown in Figure 9c, with the guidance of *LDSS* estimation, cache resource is cleverly allocated to streams based on their temporal locality. **LDSS estimation allocates fingerprint cache resources according to the temporal locality of streams and improves the inline deduplication ratio by 12.53% (see Figure 6b). The improvement clearly shows the effectiveness of *LDSS* estimation in HPDedup.**

F. Fragmentation

In this subsection, we evaluate the fragmentation of files in storage system caused by deduplication. The length threshold of duplicate block sequence controls the fragmentation in both HPDedup and DIODE.

Both DIODE and HPDedup are able to adjust the threshold dynamically. Figure 10 shows the threshold change along time for workload A in DIODE and HPDedup.

The inline deduplication ratio for DIODE and HPDedup are 57.62% and 68.96%, respectively. As one may see, HPDedup is able to adjust the threshold for each stream while DIODE uses a global threshold. The FIU-mail and Cloud-FTP have a higher threshold than FIU-home and FIU-web. Since larger threshold leads to less disk fragmentation, this result shows that **HPDedup introduces less fragmentation while achieving higher inline deduplication ratio than DIODE**.

G. Overhead Analysis

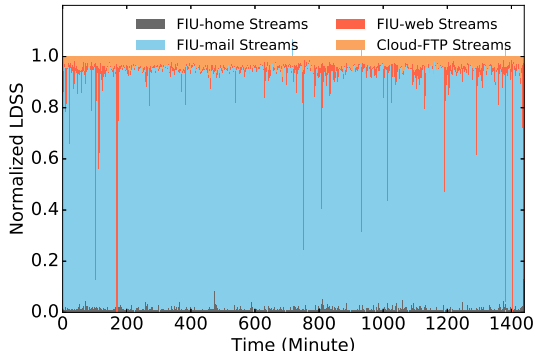
While HPDedup improves the efficiency of primary storage deduplication significantly, it inevitably incurs overhead. We analyze the overhead in this subsection. The overhead can be classified into computational overhead and memory overhead.

1) Computational Overhead: The computational overhead of HPDedup contains the following two parts: the histogram calculation time and the estimation algorithm execution time.

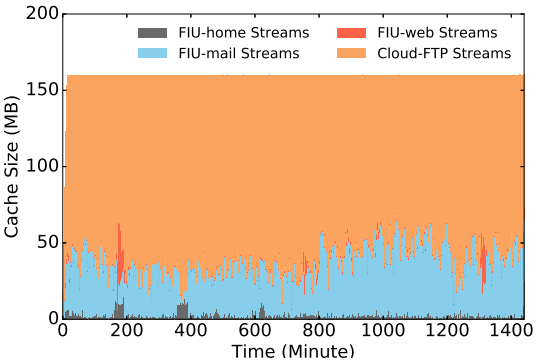
To calculate the histogram, we only need to scan the sample buffer and add the count of the fingerprints to corresponding bins of the histogram. Therefore, the time complexity is $O(n)$ where n is the number of samples. Figure 11a shows the time used for generating the histogram in our current implementation. Here, the sampling rate is 15%. We can see that the histogram calculation of an estimation interval with 1 million blocks takes less than 7ms.

Estimating the temporal locality of streams is achieved by using the method described in Section IV-A. The core of the estimation is to solve a linear programming problem. The linear programming problem can be solved in $O(n)$ [23] and even constant time [24] when the number of variables d is fixed and the number of constraints is fixed. In our context, the condition is satisfied because too frequent duplicates in the sampling buffer will be used in a straightforward way during the estimation and will not be put into the linear programming.

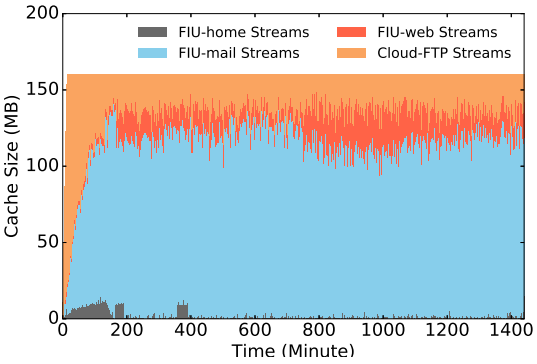
Note that the linear programming needs to be done for each stream. For every estimation interval, the temporal lo-



(a) ***Normalized Observed LDSS***

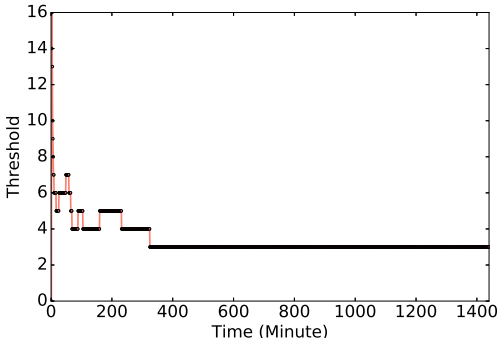


(b) ***Cache Size Distribution (without LDSS estimation)***

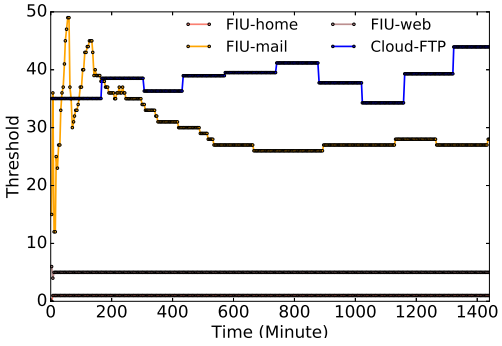


(c) ***Cache Size Distribution (with LDSS estimation)***

Figure 9: LDSS estimation accuracy.

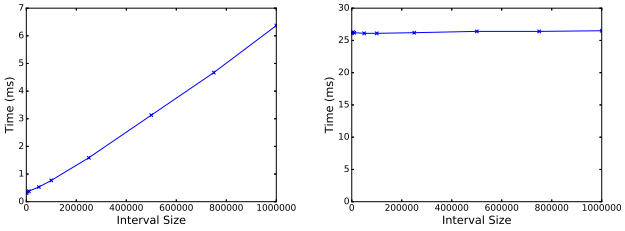


(a) ***DIODE***



(b) ***HPDedup***

Figure 10: Threshold vs. Time for HPDedup and DIODE (cache size: 160MB).



(a) ***generating histogram for samples*** (b) ***temporal locality estimation***

Figure 11: The time cost for HPDedup.

cality estimation takes about 26ms for each stream regardless the estimation interval size (see Figure 11b). The computing overhead is acceptable as the process is performed in background and does not affect the data write performance.

2) Memory Overhead: The primary memory overhead of HPDedup comes from the sampling buffer. With an estimation interval of size *EI* and a sampling rate p, the memory cost for tracking the histogram of samples is as below:

$$EI \cdot p \cdot (fpSize + counterSize)$$

where $fpSize$ and $counterSize$ are the memory cost for storing fingerprints and the occurrence count, respectively. For instance, when the cache size is 160MB, there would be approximately 2.62M cache entries. For the sampling rate of 15%, the memory overhead is only 4.49MB (2.81% of cache size) even if we choose a large estimation interval factor (e.g., Workload C, 0.6). In practice, for the mixed streams with better temporal locality, the memory overhead is much less (e.g., 2.25MB for Workload A and 2.99MB for Workload B) as the *estimation interval factor* can be set to smaller values. **Compared with the improvement (19.80% - 25.81%) of inline deduplication ratio for the three workloads with 160MB cache size), the memory overhead of HPDedup is acceptable.**

VI. Related Work

A. Primary Storage Deduplication Mechanisms

Data deduplication achieves a great success in backup storage systems. Recent research exploits various ways to apply deduplication in primary storage systems for both reducing data size in storage devices and improving I/O performance. Existing work can be classified into three categories: *Inline primary storage deduplication*, *Post-processing/Offline primary storage deduplication* and *Hybrid inline and post-processing deduplication*.

Inline primary storage deduplication. Most inline primary deduplication exploits the locality in primary workloads to perform non-exact deduplication. iDedup [4] exploits the temporal locality by only maintaining an in-memory cache to store the fingerprints of data blocks. To exploit the spatial locality, iDedup only eliminates duplicates in long sequences of data blocks. POD [5] aims at improving the I/O performance in primary storage systems and mainly performs deduplication on small I/O requests. HANDS [6] uses working set prediction to improve the locality of fingerprints. Koller et al. [16] use a content-aware cache to improve the efficiency of I/O by avoiding the influence of duplicated data. PDFS [15] argues that the locality may not commonly exist in primary workloads. To avoid the disk bottleneck of storing fingerprint table, a similarity based partial lookup solution is proposed. Leach [25] exploits the temporal locality of workloads by a splay tree. These work do not consider scenarios involving VMs and containers in the cloud where workloads for the primary storage contain a mix of data streams with different access patterns.

Post-processing/Offline primary storage deduplication. Post-processing deduplication performs deduplication during the idle time of primary storage systems. Ahmed El-Shimi et al. [2] propose a post-processing deduplication method built in Windows Server operating systems. Similar with HPDedup, DEDIS [14] is built in the Xen hypervisor to provide data deduplication functionality to multiple virtual machines. The main purpose of post-processing primary storage deduplication is to avoid the high I/O latency introduced by inline on-disk dedupe metadata lookup. However, even though the locality does not always exist in primary workloads, it is much more efficient to use inline caching rather than post-processing to eliminate duplicates in the

portion of workloads with decent locality. The contribution of HPDedup is to differentiate the deduplication procedure for primary workloads according to the temporal locality of workloads.

Hybrid inline and post-processing deduplication. Combining inline and post-processing deduplication together has been exploited by RevDedup [26] in backup storage deduplication to improve the space efficiency. For primary storage deduplication, DIODE [20] also proposes a dynamic architecture of inline-offline deduplication. Like ALG-Dedupe [27], DIODE is an application-aware deduplication mechanism. File extensions are classified into three types according to their potential deduplication ratio. Moreover, whether performing inline deduplication on a file is determined by the extension of the files. However, our experiments show that file types are not sufficient for achieving good inline deduplication performance and there is a lot of room to improve. The key difference between HPDedup and DIODE is that HPDedup gives a dynamic locality estimation method to improve inline deduplication performance. Therefore, it reduces the load of the more expensive post-processing deduplication process.

B. Unseen Distribution Estimation

Estimating the number of duplicates in a time frame for a data stream is similar to estimating the distinct elements in a large set, for which various statistics based methods (e.g., [28], [29]) have been investigated. Fisher et al. [30] describe a method to estimate the number of unknown species given a histogram of randomly sampled species. Theoretical computer science community has been trying to address how to perform the estimation with less samples [31]–[34]. Recently, this line of work has been extended by Valiant and Valiant [35] to characterize unobserved distributions. They prove that only $O(\frac{n}{log(n)})$ samples are sufficient to provide an accurate estimation of the whole dataset, in which n is the size of the whole dataset. Harnik et al. [36] utilizes the theory to estimate the deduplication ratio in storage systems.

C. Dynamic Flash Cache Management

Using prediction or historical information of workloads to improve the cache efficiency has been explored in flash cache management [37]–[41]. These work studied cache admission policies and dynamic cache allocation to reduce the flash wear-out. To the best our knowledge, HPDedup is the first work to use locality estimation to deal with the cache contention problem in inline deduplication for primary storage.

VII. Conclusion

In scenarios where multiple virtual machines or containers run in the cloud, many applications are placed in the same physical machine. Removing duplicate I/Os from the primary storage in these scenarios is useful to both improve the capacity efficiency and I/O performance. We proposed HPDedup, a hybrid prioritized deduplication method for primary storage in the cloud. HPDedup used a dynamic temporal locality estimation algorithm to achieve high inline cache efficiency and left the relatively small number of duplicates that were not in the cache to the post-processing deduplication phase to handle. By doing so, HPDedup was able to achieve exact deduplication in primary storage systems. Comparing to the state-of-art inline deduplication methods, HPDedup significantly improved the inline cache efficiency therefore achieved high inline deduplication ratio. HPDedup improves the inline deduplication ratio by up to 39.70% compared with iDedup in our experiments. Meanwhile, the improved cache efficiency made the post-processing deduplication process less a burden for the performance of an inline primary deduplication system. For example, HPDedup reduces up to 45.08% disk capacity requirement compared with the state-of-the-art post-processing deduplication mechanism in our evaluation.

Acknowledgment

We would like to thank Gregory Valiant from Stanford University for helping us to further understand the unseen entropy estimation algorithm. This work is partially supported by the The National Key Research and Development Program of China (2016YFB0200401), by program for New Century Excellent Talents in University by National Science Foundation (NSF) China 61402492, 61402486, 61402518, 61379146, by the laboratory pre-research fund (9140C810106150C81001).

References

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

[2] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data deduplicationlarge scale study and system design," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 285–296.

[3] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Transactions on Storage (TOS)*, vol. 7, no. 4, p. 14, 2012.

[4] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti, "iDedup: latency-aware, inline data deduplication for primary storage." in *FAST*, 2012.

[5] B. Mao, H. Jiang, S. Wu, and L. Tian, "Pod: Performance oriented i/o deduplication for primary storage systems in the cloud," in *IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[6] A. Wildani, E. L. Miller, and O. Rodeh, "Hands: A heuristically arranged non-backup in-line deduplication system," in *IEEE 29th International Conference on Data Engineering (ICDE)*, 2013.

[7] J. An and D. Shin, "Offline deduplication-aware block separation for solid state disk," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[8] C. Constantinescu, J. Glider, and D. Chambliss, "Mixing deduplication and compression on active data sets," in *Data Compression Conference (DCC), 2011*. IEEE, 2011, pp. 393–402.

[9] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok, "Dmdedup: Device mapper target for data deduplication," in *2014 Ottawa Linux Symposium*, 2014.

[10] H. Yu, X. Zhang, W. Huang, and W. Zheng, "Pdfs: Partially dedupped file system for primary workloads," *IEEE Transactions on Parallel and Distributed Systems*.

[11] J. Kaiser, T. Süß, L. Nagel, and A. Brinkmann, "Sorted deduplication: How to process thousands of backup streams," in *Mass Storage Systems and Technologies (MSST), 2016 32th Symposium on*. IEEE, 2016.

[12] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[13] J. Rhoton, de Jan Clercq, and F. Novak, *OpenStack Cloud Computing*. Recursive Press, 2014.

[14] J. Paulo and J. Pereira, "Efficient deduplication in a distributed primary storage infrastructure," *ACM Transactions on Storage (TOS)*, vol. 12, no. 4, p. 20, 2016.

[15] H. Yu, X. Zhang, W. Huang, and W. Zheng, "Pdfs: Partially dedupped file system for primary workloads," *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[16] R. Koller and R. Rangaswami, "I/O deduplication: Utilizing content similarity to improve i/o performance," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, 2010.

[17] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao, "Cachededup: in-line deduplication for flash caching," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 301–314.

[18] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[19] P. Valiant and G. Valiant, "Estimating the unseen: improved estimators for entropy and other properties," in *Advances in Neural Information Processing Systems (NIPS)*, 2013.

[20] Y. Tang, J. Yin, S. Deng, and Y. Li, "Diode: Dynamic inline-offline de duplication providing efficient space-saving and read/write performance for primary storage systems," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*. IEEE, 2016, pp. 481–486.

[21] Z. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok, "A long-term user-centric analysis of deduplication patterns," in *International Conference on Massive Storage Systems and Technology (MSST)*, 2016.

[22] S. Mandal, G. Kuenning, D. Ok, V. Shastry, P. Shilane, S. Zhen, V. Tarasov, and E. Zadok, "Using hints to improve inline block-layer deduplication," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 315–322.

[23] N. Megiddo, "Linear programming in linear time when the dimension is fixed," *Journal of the ACM*, vol. 31, no. 1, pp. 114–127, 1984.

[24] N. Alon and N. Megiddo, "Parallel linear programming in fixed dimension almost surely in constant time," in *The 31st Annual Symposium on Foundations of Computer Science*, 1990.

[25] B. Lin, S. Li, X. Liao, J. Zhang, and X. Liu, "Leach: an automatic learning cache for inline primary deduplication system," *Frontiers of Computer Science*, vol. 8, no. 2, pp. 175–183, 2014.

[26] Y.-K. Li, M. Xu, C.-H. Ng, and P. P. Lee, "Efficient hybrid inline and out-of-line deduplication for backup storage," *ACM Transactions on Storage (TOS)*, vol. 11, no. 1, p. 2, 2015.

[27] Y. Fu, H. Jiang, N. Xiao, L. Tian, F. Liu, and L. Xu, "Application-aware local-global source deduplication for cloud backup services of personal storage," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 25, pp. 1155–1165, 2014.

[28] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes, "Sampling-based estimation of the number of distinct values of an attribute," in *VLDB*, 1995.

[29] C. X. Mao and B. G. Lindsay, "Estimating the number of classes," *the Annals of Statistics*, pp. 917–930, 2007.

[30] R. A. Fisher, A. S. Corbet, and C. B. Williams, "The relation between the number of species and the number of individuals in a random sample of an animal population," *The Journal of Animal Ecology*, pp. 42–58, 1943.

[31] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Sampling algorithms: lower bounds and applications," in *The Thirty-Third Annual ACM Symposium on Theory of computing*, 2001.

[32] T. Batu, S. Dasgupta, R. Kumar, and R. Rubinfeld, "The complexity of approximating the entropy," *SIAM Journal on Computing*, vol. 35, no. 1, pp. 132–150, 2005.

[33] P. Valiant, "Testing symmetric properties of distributions," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1927–1968, 2011.

[34] S. Guha, A. McGregor, and S. Venkatasubramanian, "Streaming and sublinear approximation of entropy and information distances," in *The Seventeenth Annual ACM-SIAM Symposium on Discrete algorithm*, 2006.

[35] G. Valiant and P. Valiant, "Estimating the unseen: an n/log (n)-sample estimator for entropy and support size, shown optimal via new clts," in *The Forty-Third Annual ACM Symposium on Theory of computing*, 2011.

[36] D. Harnik, E. Khaitzin, and D. Sotnikov, "Estimating unseen deduplication-from theory to practice." in *FAST*, 2016, pp. 277–290.

[37] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao, "Cloudcache: On-demand flash cache management for cloud computing." in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[38] J. Yang, N. Plasson, G. Gillis, and N. Talagala, "Hec: improving endurance of high performance flash-based cache devices," in *Proceedings of the 6th International Systems and Storage Conference*. ACM, 2013, p. 10.

[39] S. Huang, Q. Wei, D. Feng, J. Chen, and C. Chen, "Improving flash-based disk cache with lazy adaptive replacement," *ACM Transactions on Storage (TOS)*, vol. 12, no. 2, p. 8, 2016.

[40] J. Liu, Y. Chai, X. Qin, and Y. Xiao, "Plc-cache: Endurable ssd cache for deduplication-based primary storage," in *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*. IEEE, 2014, pp. 1–12.

[41] R. Koller, A. J. Mashtizadeh, and R. Rangaswami, "Centaur: Host-side ssd caching for storage performance control," in *Autonomic Computing (ICAC), 2015 IEEE International Conference on*. IEEE, 2015, pp. 51–60.