Read Performance Enhancement In Data Deduplication For Secondary Storage

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Pradeep Ganesan

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Adviser
David Hung-Chang Du

May, 2013

© Pradeep Ganesan 2013
ALL RIGHTS RESERVED

Acknowledgements

I extend my sincere gratitude to my adviser Professor David H.C Du for his time and support in instilling research interests in me and helping me shaping it up. His invaluable guidance helped me throughout and his perennial enthusiasm always inspired me. I also thank Young Jin Nam who was a visiting faculty at University of Minnesota Twin Cites, for his help during my initial days of research. Along with my adviser, his continual assessment of this work was of great help to me. I thank my other reviewers Professor Jon Weissman and Professor David Lilja for their insightful comments.

I would also like to acknowledge Dongchul Park and other fellow students and faculty from the Center For Research In Intelligent Storage (CRIS) at University of Minnesota Twin Cities for their feedbacks during discussions. This research was partially supported by the National Science Foundation under NSF Grant 0960833 and NSF Grant 0934396.

I thank my father Ganesan, mother Eswari and brother Boopalan for their moral support for pursuing my interests.

Dedication

To my family and friends,

and in memory of my dear

late paternal grandparents, late maternal grandfather and late friend Santosh

Abstract

Data deduplication, an efficient technique to eliminate redundant bytes in the data to be stored, is largely used in data backup and disaster recovery. This elimination is achieved by chunking the data and identifying the duplicate chunks. Along with data reduction it also delivers commendable backup and restore speeds. While backup process pertains to write process, the restore process defines the read process of a dedupe system. With much emphasis and analysis being made to expedite the write process, the read performance of a dedupe system is still a slower process comparatively. This work proposes a method to improve the read performance by investigating the recently accessed chunks and their locality in the backup set (datastream). Based on this study of the distribution of chunks in the datastream, few chunks are identified that need to be accumulated and stored to serve the future read requests better. This identification and accumulation happen on cached chunks. By this a small degree of duplication of the deduplicated data is introduced, but by later caching them together during the restore of the same datastream, the read performance is improved. Finally the read performance results obtained through experiments with trace datasets are presented and analyzed to evaluate the design.

Contents

Acknowledgements i

Dedication ii

Abstract iii

List of Tables vi

List of Figures vii

1 Introduction 1

1.1 Deduplication Technology 3

1.1.1 Datastream 3

1.1.2 Chunking and Fingerprinting 4

1.1.3 Duplicates Identification 4

1.2 Types of Deduplication 4

1.2.1 Primary Data and Secondary Data Deduplication 5

1.2.2 Source-based and Target-based Deduplication 5

1.2.3 Inline and Offline Deduplication 6

1.3 Baseline Deduplication Scheme 6

2 Motivation 9

2.1 Cache Pollution Issue in Deduplication Read Process 9

3 **Proposed Scheme - Reminiscent Read Cache** **11**
3.1 The Framework 11
3.2 Components and Operations 12
3.2.1 Utilization Analysis (UA) 12
3.2.2 Locality Preservation 13
4 **Performance Evaluation** **19**
4.1 Datasets 19
4.2 Experimental Setup 19
4.3 Threshold Analysis 20
4.4 Experimental Results and Observations 22
4.5 Cache Size 30
5 **Related Work** **32**
6 **Conclusion and Future Work** **36**
6.1 Conclusion 36
6.2 Future Research Direction 36
Bibliography **38**

List of Tables

4.1 Dataset DS#6 : Restore of Backup-1 (a) and Backup-2 (b) : Cache size = 8 containers, Data size of each version = 4GB (DC = Dupe Containers, RC = Regular-sized Containers) 22
4.2 Deduplication Gain Ratio (DGR) of all backup versions of each dataset 22

List of Figures

1.1 Baseline Dedupe Scheme 8

3.1 Proposed Scheme Framework 12

3.2 Reminiscent Read Cache with Dupe Buffer 16

4.1 Container Utilization During Restore of First Backup Version Using Baseline Technique 21

4.2 Read Performance using Different Thresholds 21

4.3 Read Performance Analysis for Dataset DS#1 23

4.4 Read Performance Analysis for Dataset DS#2 24

4.5 Read Performance Analysis for Dataset DS#3 25

4.6 Read Performance Analysis for Dataset DS#4 26

4.7 Read Performance Analysis for Dataset DS#5 27

4.8 Read Performance Analysis for Dataset DS#6 28

4.9 Cache Size : The graph shows the read performance results for dataset DS#6 with different cache sizes of 8 to 64 Regular-sized Containers space 30

4.10 Utilization Ratio for Dataset#6 (includes both Regular-sized Containers and Dupe Containers) 31

Chapter 1

Introduction

With ever increasing size of data in the digital world it becomes extremely important to store them efficiently. The storage space required is further increased by the necessity to backup data to handle data losses. This demands an efficient way of storing data by consuming lesser storage space than the data size. This could be achieved by reducing data using different compression techniques but these techniques are not efficient for backup data since there will be multiple backup versions of the same dataset. With such multiple versions most of the data are redundant except for the updates in the subsequent versions. Compression technique has to be applied on this redundant data again and again. Hence, another method called **data deduplication** was devised to reduce the size of backup data. This technique is different from the conventional compression techniques which use encoding algorithms to compress data.

The underlying storage used for backup is called **secondary storage**. Traditionally tapes were used for secondary storage due to the data sequentiality property in backup data and low cost while the hard disks were more used as primary storage for primary data, directly accessed by the host filesystem, applications and users. However, in the last decade when the cost of hard disks plummeted they replaced the tapes in secondary storage as well. Disks are much faster than tapes for sequential reads and writes. During

backup that uses *data deduplication* when the data is streamed (write process) to the secondary storage, before it is stored its size will be reduced and then sent to the disks. This is achieved by eliminating the duplicate data in the bytestream that is streamed. Later, when this data is required to be restored back to the primary storage from the secondary storage, it will be streamed back (read process) in the same sequence as it was in the write process and the original data is constructed back. This is the data reconstruction step. Also, the dedupe process discussed in this work is applied at the byte level and not at the file level.

Although the write process is very critical with regards to storage system performance (write performance), the read process is equally significant. Restore speed directly impacts the RTO of the system. Those systems which deal with critical data like that of revenue sector, defense sector, etc., cannot afford longer downtimes. This signifies the importance of **read performance**. The system discussed in this work intends to enhance this read performance by creating duplicate copies of some of the deduplicated chunks during read process. This is done by analyzing the position of chunks in the datastream that is being currently read back and cached in the read cache. The appearance of a chunk in a backup version of a datastream with regards to its locality in the recently accessed chunk sequence is studied to decide upon whether it should be duplicated or not. Hence, the read cache used is called '**Reminiscent Read Cache**'. It is also assumed that the read sequence of data is the same as the write sequence.

Majority of the prior works are related to write performance improvement thereby efficiently detecting and removing as many duplicates as possible with the help of efficient variable-sized block data addressing [1], index optimization [12], and data container design [3]. Nam et al. originally proposed an indicator value for degraded read performance, called Chunk Fragmentation Level (CFL) [4]. Later, in [5] they proposed an

approach to improve the read performance using the CFL value. [6] and [7] also address read performance issue. These three techniques are applied during write process thus degrading the write performance. whereas in this work, the technique proposed is applied during the read process and based on various other factors.

1.1 Deduplication Technology

Data Deduplication is one of the widely used data reduction technique that saves storage space. It has varying benefits depending on the context and workload used. It saves enormous storage space if used for backup applications. This is because more than 90% of the backup data are duplicate data caused by repeated backup of the same data namespaces along with the updates from last backup.

1.1.1 Datastream

The incoming bytestream of the backup data to the secondary storage is called the *datastream*. Deduplication technique deals with this data at byte-level and eliminate the duplicates. This datastream represents a large group of files at the primary storage, usually the complete primary data of the native file system. When all these files are backed-up they are always streamed together which the secondary storage views as a single stream of bytes. Hence, the file boundary information is not available and in fact not required at the secondary storage. Also, backups are generally repetitive and timed, for example, daily backups, weekly backups etc. These different backups of the same data are called backup versions. Each version corresponds to a datastream when the backup runs. Also, the files are streamed in the same order during every backup. The only difference between the subsequent backup versions would be the updates made or the new data added after the previous backup.

1.1.2 Chunking and Fingerprinting

To compare and identify the duplicate data within a datastream or across datastreams it is required to divide the incoming datastream into multiple smaller segments. These segments are called chunks. The bytes of these chunks are compared and the ones with the same byte sequence are termed as duplicate chunks. It is easier to split the stream into chunks of the same size and compare them for duplicates. There are also techniques like Rabin Fingerprinting [8] that does content-based variable sized chunking. It was observed that variable sized chunks leads to identification of more duplicates among the chunks than fixed sized chunks. Typically variable sized chunks are of size 8KB to 32KB.

1.1.3 Duplicates Identification

To compare the chunks against each other it is not possible to compare bitwise. Hence, a unique fingerprint is calculated for each chunk based on its content. Secure Hash Algorithm (SHA) or MD5 is used to calculate a hash value for each chunk. These are proved to be collision-free algorithms. These hashes serve as the chunk indexes. Index lookup has a significant impact on the system performance as well. Efficient ways of storing and retrieving these indexes have been discussed in works like [3] and [12]. With these fingerprints it now becomes easier to compare the chunks to find the duplicate and unique chunks.

1.2 Types of Deduplication

Deduplication can be classified based on the data on which it is used, the location where it is used and the timeframe when it is used. Below are the brief explanations of these types.

1.2.1 Primary Data and Secondary Data Deduplication

Depending on the type of data to which deduplication is applied, they can be classified as *primary data deduplication* and *secondary data deduplication*. If the native file system on a host or a cluster supports deduplication to be applied on the data that it manages, then it is called primary data deduplication. For primary data deduplication control over the file metadata is required and there is no data sequentiality as in backup storage. Also, since the file system works with blocks of fixed size it is advisable to use fixed sized chunks (equal to that of a block) to avoid any degradation in system performance.

On the other hand, as it has been discussed previously deduplication applied on backup data stored in secondary storage is called secondary data deduplication and variable sized chunking leads to better duplicate elimination.

1.2.2 Source-based and Target-based Deduplication

Source-based deduplication is client-side deduplication where the dedupe engine runs in the source where the original data (primary data) resides. This consumes processor cycles at the source as the chunking is done here. However, since the duplicates are identified at the source, only the unique data is sent through the network to be stored at the target (typically secondary) storage. This save network bandwidth.

Whereas, *target-based deduplication* is the one where the dedupe algorithm runs at the target storage system where the data is backed-up to. Primary data is as such sent to the target storage systems which in-turn chunks the data and eliminates the duplicates before physically storing them in disks. This is a faster method since the host's primary storage does not incur any overhead caused by chunking and is totally unaware of the dedupe process.

1.2.3 Inline and Offline Deduplication

Another way of classification is based on when deduplication algorithm is applied. Data generated could be reduced before storing them in the target devices. This is *inline deduplication*. This is a costly operation in terms of processor cycles used thus causing an overhead in system performance but storage space required during system initialization would be lesser than the actual data size. This saves cost.

When data is allowed to be stored without being reduced and dedupe process is later applied on the stored data as an offline process during the idle time of the system, then this approach is called *offline deduplication*. This is doesn't require any extra processor cycles and hence faster but the storage space required initially will be far greater than that is required for inline process.

Inline deduplication can be beneficial when used in backup systems provided inline applies to the context of target storage system. Target is not performance-sensitive and works better with target-based deduplication. Offline deduplication is better for primary workloads since it is doesn't cause any latency in primary data I/O.

However, inline primary data storage deduplication has recently been shown some interest by industries like in [17]. Inline primary data deduplication has the tendency of degrading the system performance largely if not designed suitable for the characteristics of primary workload that the system deals with. Also, storage savings is not as much as it is in secondary storage.

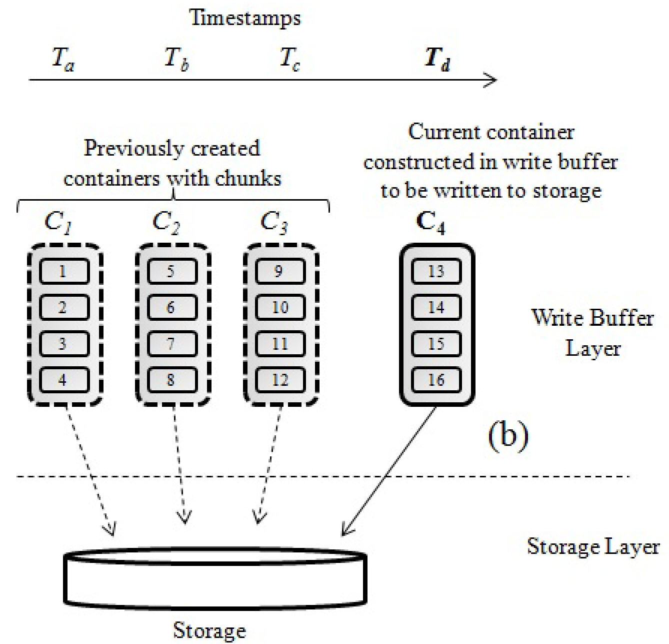
1.3 Baseline Deduplication Scheme

As explained in Section-1.1 , the bytestream (Fig-1.1(a)) of the backup dataset (datastream) that is to be written to disks in the underlying storage is first chunked into varied sized chunks using a content-aware chunking technique [8]. A fingerprint is created for

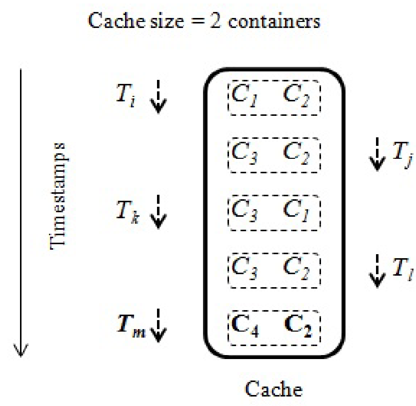
each chunk using SHA1 hashing technique and this acts as the chunk's reference in the chunk's metadata. The same is used in a Hash Table (HT) Index that is used to find the duplicate chunks. Both this baseline and the proposed schemes assume that there is enough DRAM space to accommodate the complete HT Index. If a referred chunk already has an entry in the HT Index it is termed as duplicate, otherwise it is a unique chunk. All chunks pass through the dedupe engine but only those that are unique are allowed to be stored in the disks while the duplicate chunks are discarded. During the restore process of the datastream in the same order of the chunk sequence as the backup process, the required chunks are read from the disks using the chunk metadata.

During the write process (Fig-1.1(b)) chunks are buffered in a write buffer in the main memory of the storage system before being written sequentially to disks as one unit. This unit of write is called container with a unique ID and it also serves as the unit of read I/O. This is referred to as the regular-sized container in the proposed scheme. Size of a container is fixed in baseline deduplication scheme. This utilizes the sequential read/write advantage in disks. Similarly, during the read process (Fig-1.1(c)) to serve a chunk request the whole container containing this chunk along with the other chunks, is read from disks and cached in the main memory. This sequential read of chunks in a container exploits the spatial locality of the chunks in disks. It also assumes that when a chunk is read its neighbors would also be potentially read subsequently, thus addressing the temporal locality of the chunks in the chunk sequence of the datastream. The above two concepts explain the chunk pre-fetching indirectly. The size of the cache referred above is fixed and is based on LRU as the eviction policy with container as the caching unit.

(a) Chunk Sequence in Datastream during Write and Read Processes



(b) Write Buffer during Write Process



(c) Read Cache during Read Process

Figure 1.1: Baseline Dedupe Scheme

Chapter 2

Motivation

During the read (restore) process of the backup data from secondary storage to primary storage, when the chunk at the current position in the chunk read sequence is not available in cache, it requires to fetch the container containing this chunk from disks. This container is read from disks and cached in main memory to serve a chunk read request. Sometimes it turns out that only few of the chunks in a cached container will serve cache hits for the forthcoming subsequent chunk requests while others are not used before it is evicted from cache [5]. This under-utilization of cached containers is due to the presence of duplicate chunks dispersed in the datastream which results in **cache pollution**. This dispersion is termed as chunk fragmentation in [6].

2.1 Cache Pollution Issue in Deduplication Read Process

Cache pollution is illustrated and explained in Fig-1.1 as a simple scaled down depiction of the cache explained thus far. Fig-1.1(a) represents the chunk sequence in a datastream where each block is a chunk. For simplicity and to emphasize only the cache pollution issue through this Fig-1.1, it is assumed that all the chunks in this figure are of the same size. However, in reality these will be of various sizes and accordingly might cause

different containers to contain different number of chunks since the container size is fixed. Fig-1.1(b) represents the state of the write buffer at different timestamps just before it is written to disks. For example, at timestamp T_a the container constructed in write buffer had chunk-1 to chunk-4, and so on. Fig-1.1(c) depicts the state of the read cache at different timestamps while serving the chunk read requests of the datastream (Fig-1.1(a)) during read process, with the size of the cache as 2 containers. For example, at timestamp T_i the two containers cached were C_1 and C_2, caused by the first read request for chunk-1 and chunk-5 respectively in the datastream beginning from the position 'Start'.

In the read cache, at timestamp T_k container C_1 replaces C_2 by LRU to serve the read request for chunk-1 (its third occurrence in the datastream which is a duplicate). This is followed by chunk-10 which results in a cache hit on C_3 and it continues. When read position proceeds to chunk-5, at timestamp T_l container C_1 will be evicted and replaced by C_2. By this time only one chunk from C_1 would have been used. Hence, it was under-utilized and was residing in cache without serving the expected cache hits of it. It has caused cache pollution. **Under-utilization** is defined as the case when a cached container is not utilized up to a threshold level. Also, it can be observed that instead of just 4 container reads, there was a total of 6 container reads from disks which degrade the read performance. Reminiscent Read Cache model addresses the above explained issue.

Chapter 3

Proposed Scheme - Reminiscent Read Cache

3.1 The Framework

Reminiscent Read Cache model deals with only single datastream for read/write. Most importantly, it involves modifying the LRU cache into one that also remembers the chunks in the cached containers that were accessed in the past. Based on this, a minor degree of duplication is introduced by duplicating and accumulating used chunks from under-utilized cached containers, in a buffer in the memory. These chunks are actually duplicates of already read chunks in the datastream chunk sequence. The buffer is called **Dupe Buffer** and is of fixed size, equal to that of a regular-sized container. The duplicated chunks called **Dupe Chunks**, are accumulated in a container in this buffer called the **Dupe Container** which is periodically flushed to disks based on the presence of its ID in a **Past Access Window**, restricting it to different sizes. By this **variable-sized Dupe Containers** are generated in the system. Four different known sizes are used, namely, quarter, half, three-quarters and full sized containers where full size is that of a regular-sized container. Interestingly, the Dupe Buffer can also be considered to be

part of the cache and can serve cache hits. Fig-3.1 portrays the different components in this scheme which are explained below along with their functionalities.

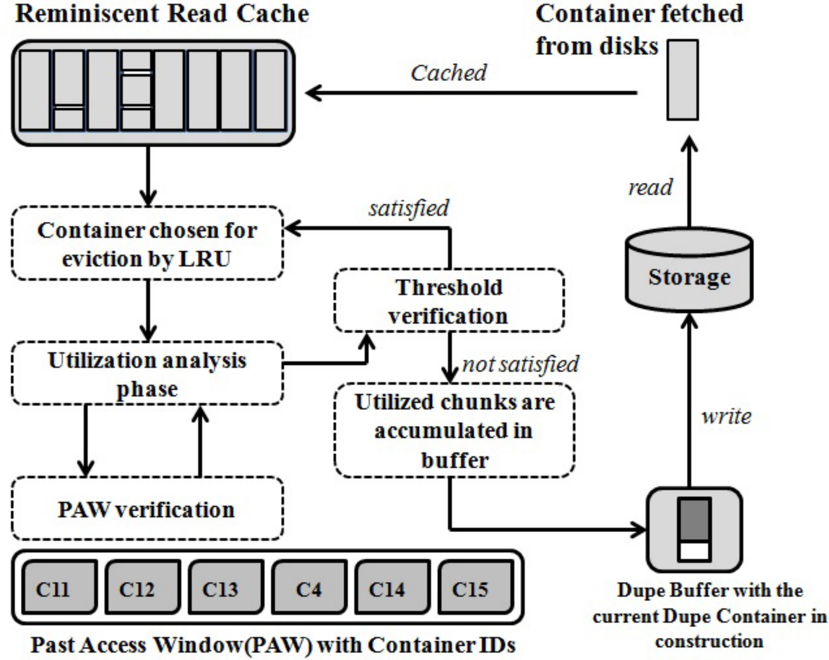


Figure 3.1: Proposed Scheme Framework

3.2 Components and Operations

3.2.1 Utilization Analysis (UA)

This is the process where a container victimized for eviction from cache is checked for how much of its chunks were accessed in the past when it was residing in the cache. This traction of the access history for the currently cached containers is used to identify the under-utilized containers. For such containers only the used chunks would be buffered into the Dupe Container in the Dupe Buffer. A container is termed as under-utilized if the **Utilization Ratio** 'UR' is not above the **threshold** 'T' set by the system. UR is the portion of a cached container that was used while being cached. Let $chunkSize_i$

denote the size of chunk_i and *containerSize* be the size of a regular-sized container, then

$$UR = \frac{\sum_{i=1}^{l} chunkSize_i}{containerSize} \tag{3.1}$$

where 'l' is the number of utilized chunks and $0 \leq \text{T} \leq 1$. The following inequality,

$$UR \leq T \tag{3.2}$$

is used by the Reminiscent Read Cache to identify those containers which are under-utilized.

Fig-3.1 depicts the above process. The chosen LRU victim container before being discarded from cache is put through the Utilization Analysis. In UA, utilization-ratio is calculated and verified against the threshold. UA also uses Past Access Window to decide on the size of the Dupe Container. The used chunks from the under-utilized cached containers are selected, duplicated and buffered in Dupe Buffer.

3.2.2 Locality Preservation

Another key contribution of this scheme is to preserve the spatial and temporal localities of the duplicate chunks that are dispersed in the datastream. Through the creation of regular containers in the baseline scheme and this proposed scheme during the write process, these locality information are preserved, provided the chunks in the container are in the same order as in the datastream as well. However, as highlighted in Section-1.3 there are duplicate chunks that appear in the datastream. Only these duplicate chunks are considered separately and it has been intended to preserve the spatial and temporal localities that prevail among these by accumulating them into the variable sized Dupe Containers. To achieve this the Past Access Window described below is made use of.

3.2.2.1 Past Access Window (PAW)

This is a list of container IDs that keeps track of the recently cached 'n' containers. For the experiments this number has been fixed to be thrice the number of regular-sized containers the cache can accommodate. The used chunks from a under-utilized container should either be buffered in the current Dupe Container in the Dupe Buffer or in the next one by flushing the current one to disks. This decision is taken based on the presence of the container ID of the under-utilized container in PAW. If it is not, the current Dupe Container size in buffer is restricted to the least known size (of the four different known sizes) greater than the current size and a new one is created. Fig-3.1 shows when is that PAW is used by Utilization Analysis. The list in the figure has 6 container IDs of those that were most recently cached.

This takes into account the chunk dispersion level of the duplicates in the datastream on an approximate scale. Chunk dispersion is the degree to which the duplicate chunks are spread in the datastream (termed as chunk fragmentation level in [5]). This is not quantified as a measure in this work but approximated by using PAW. By this, certain degree of the spatial and temporal localities of duplicate chunks in the stream (by creating the Dupe Containers) are maintained.

3.2.2.2 Spatial and Temporal Localities in Variable Sized Containers

Algorithm-1 shows the parameters based on which the size of a Dupe Container is decided in order to preserve the localities of the duplicate chunks. The function *Check-Chunk-Dispersion()* is used to validate the chunk dispersion level. If the current size of the Dupe Container currently being filled up in Dupe Buffer is closer to any of the four known sizes and on adding the used chunks of the under-utilized cached container that is to be evicted the current size surpasses this known size, then the size of the Dupe Container is restricted to this known size. It is flushed to disks and a new container is

opened in the buffer. Otherwise, if the current size of the Dupe Container is not closer to any of the four known sizes but on adding the used chunks of the under-utilized cached container that is to be evicted surpasses any of the known sizes and the container ID of this to-be-evicted container is not present in Past Access Window, then again the size of the Dupe Container is restricted to this known size. (The container portions given in percentage are chosen purely by intuition currently but as part of future work this parameter would be made adaptive to workloads).

What is achieved by restricting the sizes of Dupe Containers to smaller sizes is the preservation of temporal locality. Since the duplicate chunks are spread (dispersed) across the datastream, grouping them into Dupe Containers has to be minimal. If more duplicates are grouped together to increase the spatial locality, temporal locality is lost because caching such containers might require a longer stay in cache to serve the read requests of these duplicates. But before being utilized better they will be evicted by LRU. Hence, these will cause unexpected cache pollution. Whereas, if the LRU victim container is a fairly recently cached container (a time window defined by PAW size) then the buffering of these duplicate chunks into the same Dupe Container is reasonable, otherwise, they will have to go into next Dupe Container (causing the flush of the current one to disks).

Grouping of dispersed duplicate chunks into Dupe Containers together preserves their spatial locality. The objective is to have a Dupe Container utilized as far as possible when cached in future during read process. Again this is controlled by using smaller sizes for Dupe Containers.

When a container cached is a Dupe Container, Utilization Analysis is not applied on them. Otherwise, this will lead to a high degree of duplication that is induced. Multiple copies of the same chunk are allowed to be present in different Dupe containers to maintain the temporal locality of these chunks in those neighborhoods in the datastream,

wherever they are present. By this, when this container is cached when any of the subsequent backup versions of the datastream is restored (read), it serves the read sequence better in their respective neighborhoods (localities).

Fig-3.2 is a depiction of the Reminiscent Read Cache. In the fourth slot in the cache, there are three Dupe Containers (DC) that are accommodated. Two are quarter-sized and the other is half-sized DC. Similarly the second slot is occupied by two DCs. The remaining slots have regular-sized containers (RC). It also shows the Dupe Buffer with its in-memory DC that is being constructed and partially filled.

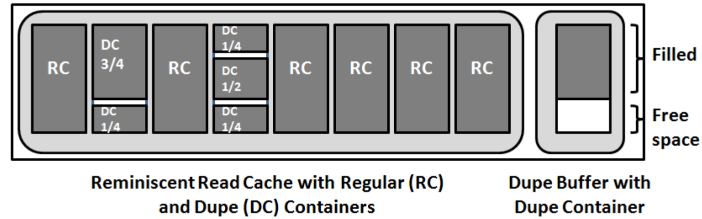


Figure 3.2: Reminiscent Read Cache with Dupe Buffer

Algorithm 1 Locality Preservation in Dupe Container

Function Decide-Buffer-Container-Size(victimID)

// **Presetting container sizes**

QUARTER CONTAINER = MAX CONTAINER SIZE / 4

HALF CONTAINER = MAX CONTAINER SIZE / 2

*THREE QUARTERS CONTAINER = 3 * MAX CONTAINER SIZE / 4*

REGULAR CONTAINER = MAX CONTAINER SIZE

Size Check = Current Dupe Container Size + Total size of accessed chunks from the victimized container (victimID)

// **Find and set optimal size for the current Dupe Container in Dupe Buffer**

if *Check-Chunk-Dispersion(Current Dupe Container Size, QUARTER CONTAINER, 70%, Size Check)* **then**

return

else

if *Check-Chunk-Dispersion(Current Dupe Container Size, HALF CONTAINER, 80%, Size Check)* **then**

return

else

if *Check-Chunk-Dispersion(Current Dupe Container Size, THREE QUARTERS CONTAINER, 90%, Size Check)* **then**

return

else

Set the Dupe Container size to be MAX CONTAINER SIZE

end if

end if

end if

Function Check-Chunk-Dispersion(Dupe Container Size, Variable Container, Container Portion%, Size Check)

// **Conditions to set Dupe Container size to Variable Container Size**

if (*Current Dupe Container Size* $\geq$ *Container Portion% of Variable Container* **and** *Current Dupe Container Size* $\leq$ *Variable Container Size* **and** *Size Check* $>$ *Variable Container Size*)

OR (*Current Dupe Container Size* $\leq$ *Variable Container Size* **and** *Size Check* $>$ *Variable Container Size* **and** *victimID not in Past Access Window*)

then

Set the Dupe Container size to be Variable Container Size

Flush the Dupe Container to disks and create a new one in Dupe Buffer

return true

else

return false

end if

Chapter 4

Performance Evaluation

4.1 Datasets

Six different backup datasets are used for the experiments. Each dataset (DS) has 2 or 5 backup versions. Datasets DS#1, DS#2, DS#3, DS#4 and DS#6 are traces representing data of size 20GB while DS#5 represents 8GB data. They are trimmed versions of traces representing data of much larger sizes. DS#1, DS#2 and DS#3 are exchange server data, DS#4 contains system data of a revision control system, DS#6 contains /var directory of that machine and DS#5 is home directory data of several users. Average chunk size in these datasets is 8KB.

4.2 Experimental Setup

For experimental purpose these small trace datasets were used with smaller and fixed cache size of 8MB/16MB, to be on scale with these datasets sizes. The container size is fixed as 2MB. This implies that 8MB/16MB cache accommodates 4/8 containers. In future, this model is intended to be scaled up for large datasets. Also, currently it is assumed that there is only one datastream that is being written or read at a given time in the system, while in reality there will be multiple streams. For example if there

are 16 streams being restored, each allocated a cache space of 16MB, it sums up to 256MB. This cache space for 20GB of data scales up to ∼0.2TB cache space for 4TB data, which is in par with enterprise backup scales. With such a large scale of data this model would require improved chunk pre-fetching and caching techniques since there might be duplicates across datastreams which might potentially lead to more cache pollution as described in Section-2.

The setup uses Seagate Cheetah 15K.5 Hard Drives simulated by DiskSim v.4.0 [9]. RAID-0 is configured across 4 disks each of size 146GB with stripe unit size as 64KB. Reminiscent Read Cache system is compared against the baseline dedupe scheme.

The Dupe Containers generated in the proposed scheme are of sizes 0.5Mb, 1Mb, 1.5MB and 2MB. Since this scheme also includes a Dupe Buffer of size equal to that of a regular-sized container (2MB) in memory, which can also serve cache hits, it is required to provide a fair evaluation. Hence, for experiments this scheme used (8MB/16MB cache + 1-Dupe Buffer)=10MB/18MB memory space which is 5/9 regular-sized container space. If Dupe Containers are cached, more than 5/9 containers could be accommodated but cache size is fixed at 8MB/16MB as shown in Fig-3.2. Thus, the cache size is represented with the number of regular-sized containers it can hold. For baseline dedupe scheme the cache space itself is 10MB/18MB (5/9 containers) for fair comparison.

4.3 Threshold Analysis

In the Utilization Analysis phase the under-utilized cached containers are identified using a threshold value. This is the minimum required *utilization ratio* of the cached container chosen by LRU for eviction, so that the accessed (used) chunks will not be considered for buffering in Dupe Buffer. Fig-4.1 shows the utilization ratios (in percentage) of the cached containers during the read process for different datasets in % using baseline deduplication scheme. Vertical axis represents the % of the total number of

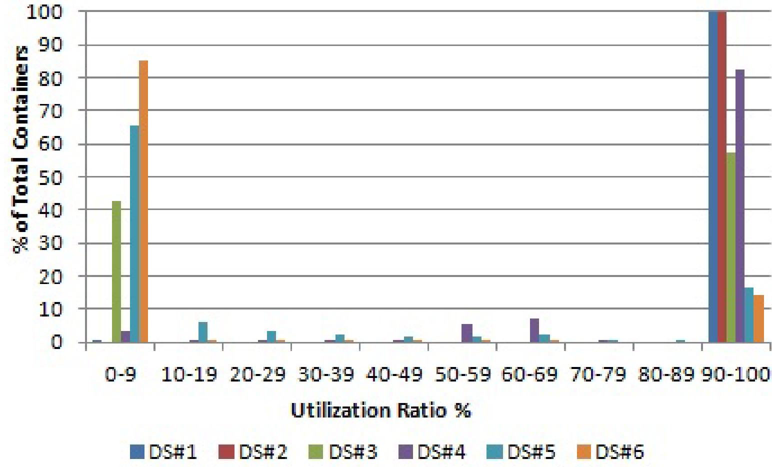


Figure 4.1: Container Utilization During Restore of First Backup Version Using Baseline Technique

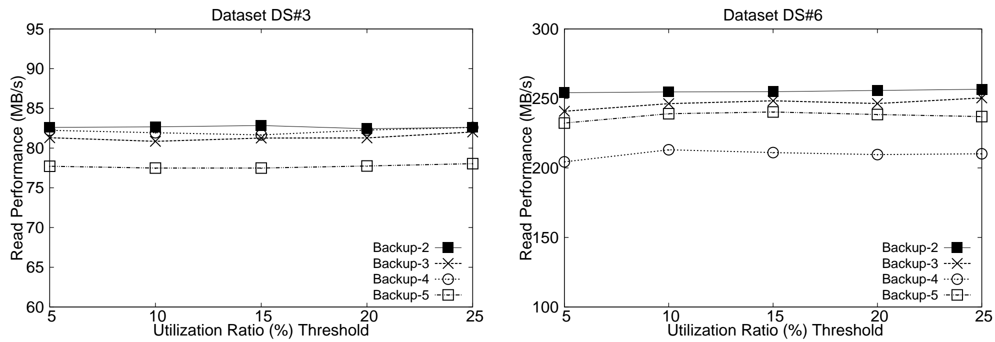


Figure 4.2: Read Performance using Different Thresholds

containers that were cached during read process that fall in different ranges of utilization ratios. For example, during DS#6's read process, 85% of the cached containers were utilized 0%-9%, 14% of them were highly utilized (90%-100%) while the rest 1% is scattered in rest of the ranges. In common, for all datasets most of the containers fall either in 0%-9% or 90%-100% utilization ranges. This suggests a smaller threshold value because a higher value will not result in much difference in performance unless it falls in 90%-100% range. If chosen in 90%-100% range, it will only lead to the creation

of more Dupe Containers during read process which will degrade the read performance.

To choose a threshold for the experiments the proposed system was tested using different threshold values of 5%, 10%, 15%, 20% and 25% for datasets DS#3 and DS#6. The respective read performance for their subsequent backup version after the first backup is shown in Fig-4.2. It shows that for DS#6 there is lesser variation in performance for thresholds >15% while it equally varies for thresholds <15% and >15% with DS#3. Also, to avoid more Dupe Containers it was decided that the smaller value of 15% be used as threshold for the rest of the experiments.

4.4 Experimental Results and Observations

Table 4.1: Dataset DS#6 : Restore of Backup-1 (a) and Backup-2 (b) : Cache size = 8 containers, Data size of each version = 4GB (DC = Dupe Containers, RC = Regular-sized Containers)

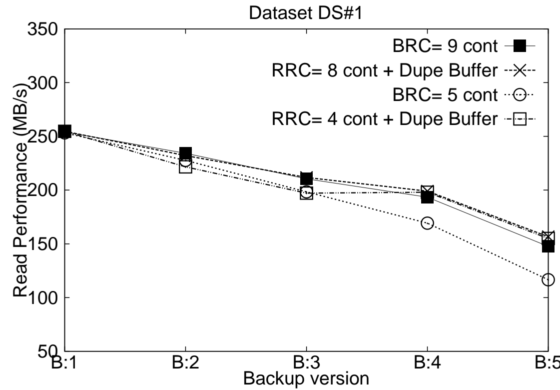
	Baseline Scheme		Reminiscent Read Cache Scheme				
	Total read size	#Cont. evictions	Total read size	RC read size	DC read size	DC write size	#Cont. evictions
(A)	24.30GB	12431	5.87GB	5.69GB	0.18GB	0.04GB	3405
(B)	24.52GB	12547	3.79GB	3.66GB	0.13GB	0.01GB	2230

Table 4.2: Deduplication Gain Ratio (DGR) of all backup versions of each dataset

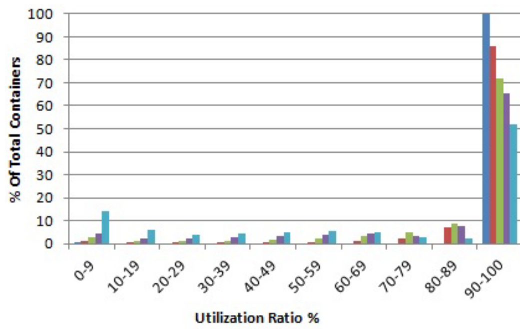
	Backup-1	Backup-2	Backup-3	Backup-4	Backup-5	avg. DGR
DS#1	0.999	0.035	0.069	0.056	0.312	0.29
DS#2	1.000	0.280	0.247	0.149	0.206	0.37
DS#3	0.996	0.952	0.977	0.973	0.966	0.97
DS#4	0.905	0.554	0.636	0.208	0.206	0.50
DS#5	0.544	0.224	–	–	–	0.38
DS#6	0.841	0.033	0.025	0.119	0.026	0.20

Figures 4.3 - 4.8 show the read performance results along with the utilization ratios

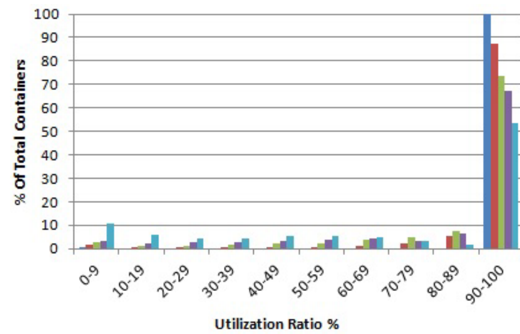
for both baseline and reminiscent read cache schemes. In DS#1 and DS#2 with 4 cached containers (RRC= 4 cont + Dupe Buffer) it is better than baseline scheme (BRC= 5 cont) for the subsequent backup versions as intended.



(a) Read Performance Results (BRC - Baseline Read Cache, RRC - Reminiscent Read Cache)



(b) Baseline Dedupe Scheme



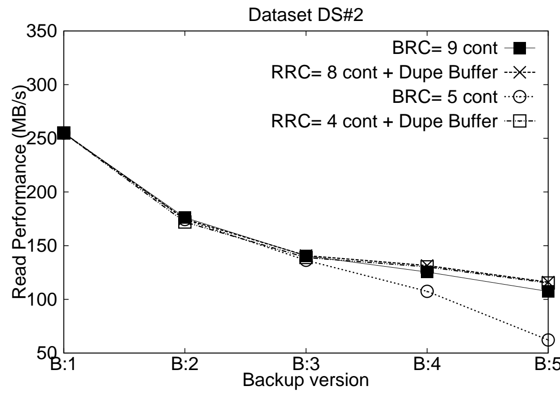
(c) Reminiscent Read Cache Scheme (Includes both Regular-sized and Dupe Containers)

Backup-1 Backup-2 Backup-3 Backup-4 Backup-5

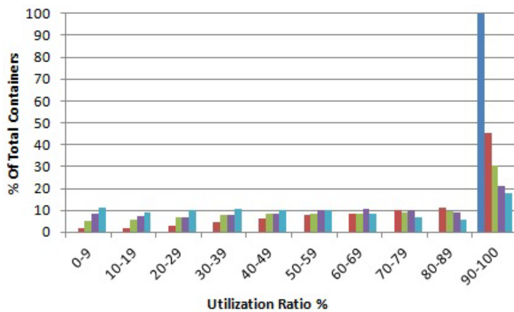
Figure 4.3: Read Performance Analysis for Dataset DS#1

This proves that the Dupe Containers generated during read process of one version helps in improving the read performance of the subsequent versions. Though restore from different versions of a same dataset might be a rare phenomenon in practical cases, these experiments are performed to study the behavior of the proposed scheme. When

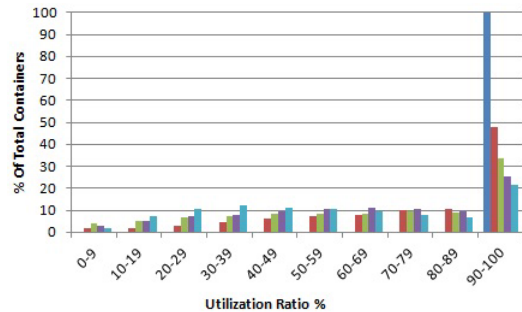
the cache size is increased to 8 cached containers it yields better performance than 4 cached containers. In some cases, irrespective of the scheme used, the subsequent versions yield lesser read performance than its predecessor. This is because of varying deduplication in these versions.



(a) Read Performance Results (BRC - Baseline Read Cache, RRC - Reminiscent Read Cache)



(b) Baseline Dedupe Scheme



(c) Reminiscent Read Cache Scheme (Includes both Regular-sized and Dupe Containers)

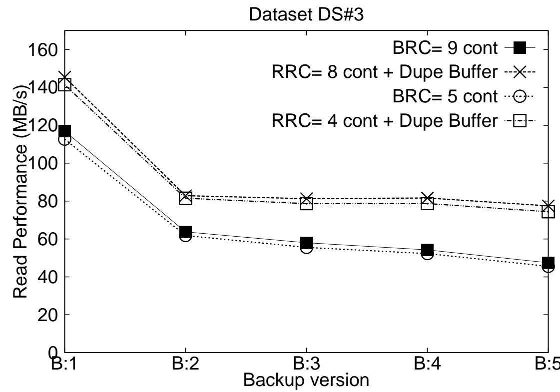
Backup-1 Backup-2 Backup-3 Backup-4 Backup-5

Figure 4.4: Read Performance Analysis for Dataset DS#2

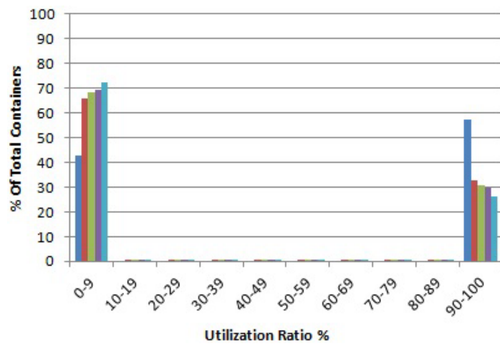
In [5] a term **Deduplication Gain Ratio (DGR)** is defined as the ratio of the stored datastream size to the original size. Lower the DGR, higher is the number of duplicate chunks in the datastream. Table-4.2 shows the DGR for different versions of

the datasets.

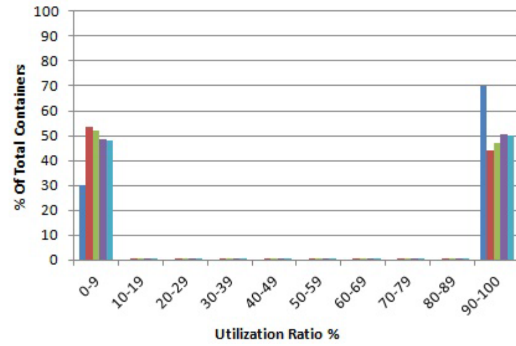
Interestingly, for DS#6 proposed scheme produces 8x performance improvement as against the other datasets which show much lesser improvement, except DS#5 (which like DS#6 shows large improvement). This dataset DS#6 is analyzed in detail below.



(a) Read Performance Results (BRC - Baseline Read Cache, RRC - Reminiscent Read Cache)



(b) Baseline Dedupe Scheme



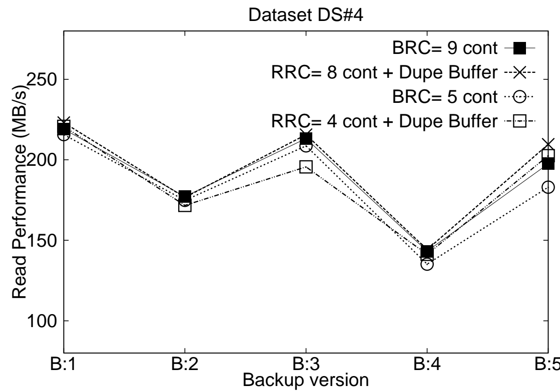
(c) Reminiscent Read Cache Scheme (Includes both Regular-sized and Dupe Containers)

Backup-1 Backup-2 Backup-3 Backup-4 Backup-5

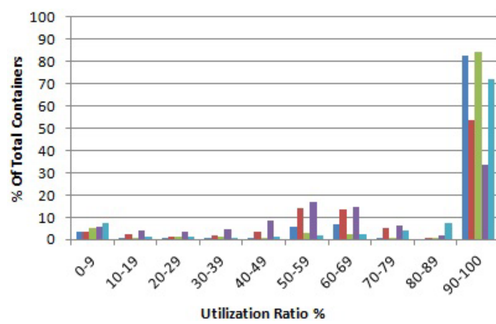
Figure 4.5: Read Performance Analysis for Dataset DS#3

As explained in Section-4.3, Fig-4.8(b) and Fig-4.8(c) represent the utilization ratio of the cached containers for DS#6. It shows that using baseline scheme for restore of

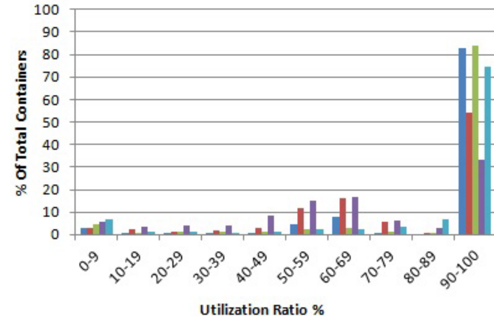
DS#6's backup versions, 85% of the cached containers in each version were utilized below the threshold (15%). They were under-utilized because of the presence of more duplicate chunks which are dispersed in the datastream which in turn cause more container reads to serve the read requests of these duplicates. Table-4.2 proves this.



(a) Read Performance Results (BRC - Baseline Read Cache, RRC - Reminiscent Read Cache)



(b) Baseline Dedupe Scheme



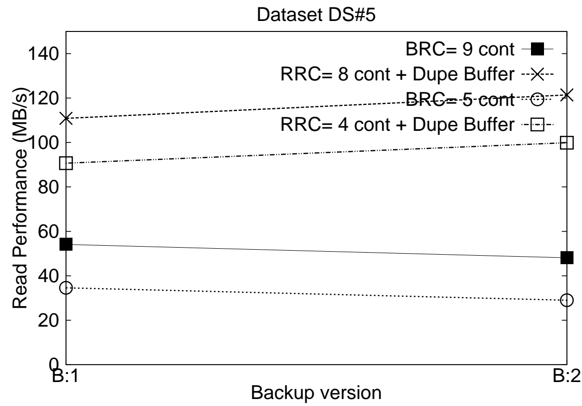
(c) Reminiscent Read Cache Scheme (Includes both Regular-sized and Dupe Containers)

Backup-1 Backup-2 Backup-3 Backup-4 Backup-5

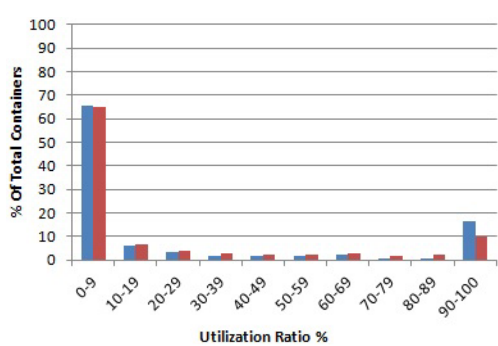
Figure 4.6: Read Performance Analysis for Dataset DS#4

First backup version of DS#6 has DGR of 0.841 which means that approximately 16% of the total data in the datastream are duplicate data. During the read process of this 16% data, the disk reads triggered to fetch containers containing the respective

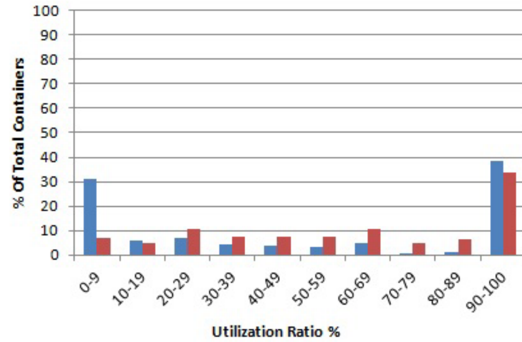
chunks of this data portion, have caused cache pollution. These containers that were cached were not better utilized. They were evicted soon after serving the required chunk reads. Table-4.1 shows that data of size 6 times the original size is read from disks. The next four backup versions of DS#6 have much smaller DGR value implying that most of the data are duplicate data.



(a) Read Performance Results (BRC - Baseline Read Cache, RRC - Reminiscent Read Cache)



(b) Baseline Dedupe Scheme



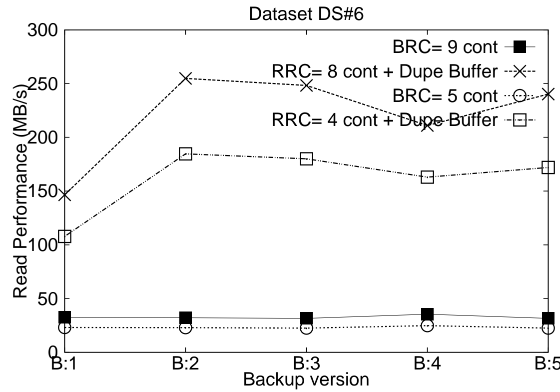
(c) Reminiscent Read Cache Scheme (Includes both Regular-sized and Dupe Containers)

Backup-1 Backup-2

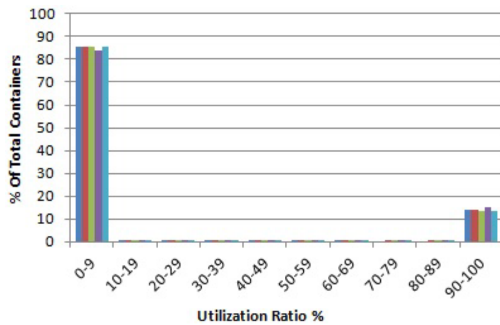
Figure 4.7: Read Performance Analysis for Dataset DS#5

As mentioned, Reminiscent Read Cache scheme also aims at producing better read

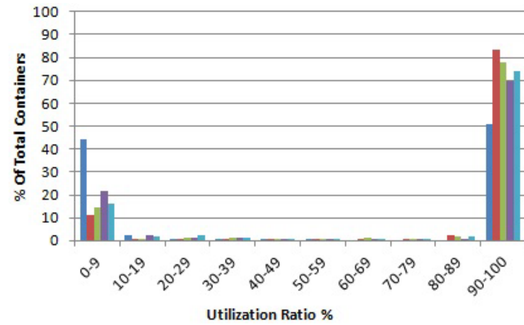
performance for the subsequent backup versions by collecting and duplicating the used chunks from under-utilized cached containers in the current version's read process into Dupe Containers. These are written to disks.



(a) Read Performance Results (BRC - Baseline Read Cache, RRC - Reminiscent Read Cache)



(b) Baseline Dedupe Scheme



(c) Reminiscent Read Cache Scheme (Includes both Regular-sized and Dupe Containers)

Backup-1 Backup-2 Backup-3 Backup-4 Backup-5

Figure 4.8: Read Performance Analysis for Dataset DS#6

As given in Table-4.1 1% of the actual data was duplicated and written to disks as Dupe Containers during the restore of the first backup version of DS#6. These containers when cached during restore of later versions, yield lesser read time and better utilization. Due to higher utilization, the number of container evictions also reduces

which implies lesser cache pollution. These Dupe Containers are also fetched during the read process of the same version when they are created. This also causes improved performance of the same version when compared to baseline scheme's performance. In case of DS#6, the first backup version itself has good amount of duplicates as against the other datasets which do not, except DS#5. This can be observed through the respective DGR values. Hence, the Dupe Containers created during restore of first backup version improve the read performance of this same version. Also, they improve the read performance of the next version and later versions by being fetched at read positions in datastream chunk sequence where duplicate chunks occur.

Table-4.1 shows that during read process of first backup version with baseline scheme, 6x of the original data size is read from disks. The same version with the proposed scheme results in reading 4.1x lesser data than baseline (1.5x of original data). The number of cached container evictions also goes down by 3.6x. In Fig-4.8(c) for first backup version the number of under-utilized containers is reduced to half while it is tripled in >90% range. Hence, as shown in Fig-4.8(a) with 8 cached containers this scheme gives 4.5x improvement. For the restore of second backup version, the DGR is 0.033 and hence 96.7% of the data are duplicates. There is no significant new data. With baseline scheme there is not much change in performance while with proposed scheme it gives 8x improvement compared to baseline. This is because of the caching of Dupe Containers created previously. Table-4.1 shows that disk read size remains same for this version with baseline but with this scheme it is 1.5x lesser than previous version. Container evictions is also reduced by 1.5x but it remains fairly same with baseline. Fig-4.8(c) shows that for this version almost 83% of the containers are utilized >90%. Hence, as in Fig-4.8(a), read performance is increased by 1.7x compared to previous version (8x compared to baseline).

4.5 Cache Size

Cache size is a significant factor that impacts the read performance. With increasing cache sizes it is generally expected that the read performance of a system increases. So far the experiments were focused on the behavior of the system for different traces based on its chunk dispersion.

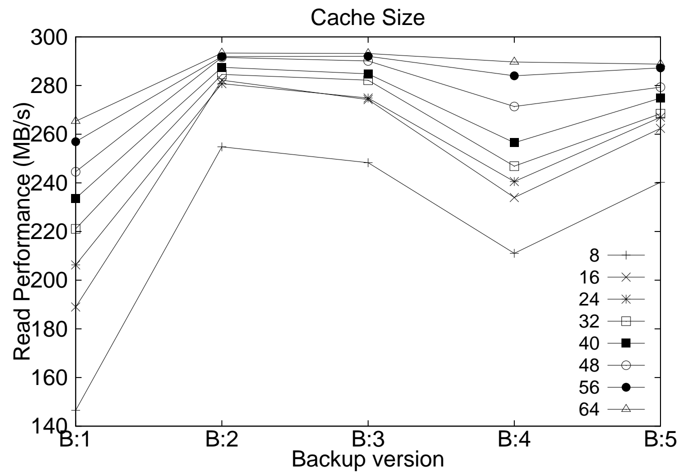


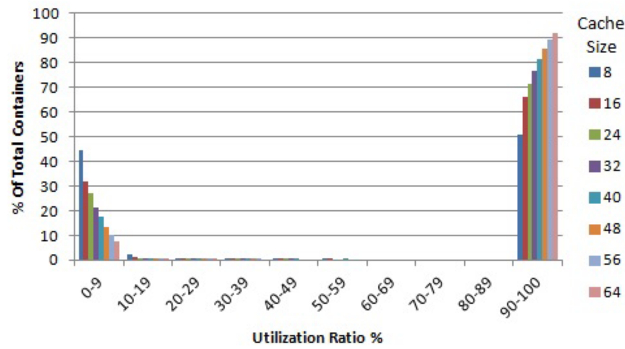
Figure 4.9: Cache Size : The graph shows the read performance results for dataset DS#6 with different cache sizes of 8 to 64 Regular-sized Containers space

This section discusses about how different cache sizes affect the read performance. For this discussion again dataset DS#6 which has high degree of chunk dispersion is taken for illustration. Experiments with cache size from 8 Regular-sized Container space to 64 Regular-sized Container space (in increments of 8) were conducted. It was indeed observed that with increasing cache sizes the read performance also steadily increases.

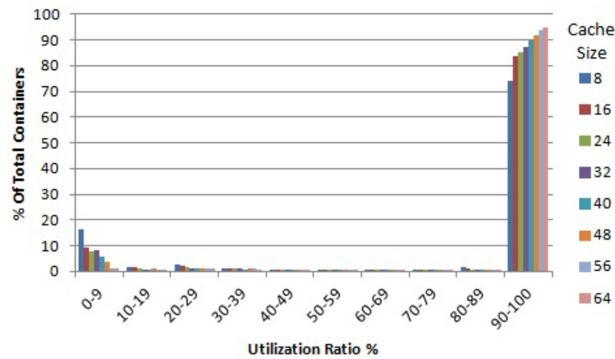
In Figure-4.9 it can be observed that in each version of backup there is a constant increase in read performance as the cache size increases. This implies that when projected to large enterprise server cache sizes of orders of gigabytes, this system will yield better performance during restore of large data. Higher cache sizes can accommodate more Dupe Containers thus taking better advantage of the induced spatial locality for

the duplicate chunks put together in these containers.

Utilization ratio of these containers increase as shown in Figure-4.10. Figure-4.10(a) shows the same for first backup version of DS#6. As it can be seen those containers that were under-utilized falling in the range 0%-9% steadily decrease while those in the range 90%-100% increases constantly. It is the same in the case of last backup version of DS#6 as well (Figure-4.10(b)). However, this increase in lesser yet steady. This explains the large difference in increase for backup-1 but comparatively lesser for backup-5 as seen in Figure-4.10.



(a) Utilization Ratio for Backup-1



(b) Utilization Ratio for Backup-5

Figure 4.10: Utilization Ratio for Dataset#6 (includes both Regular-sized Containers and Dupe Containers)

Chapter 5

Related Work

Deduplication as a technology was realized by using concepts from many other prior works. One of the significant part of deduplication is content-aware variable sized chunking of data at byte-level. This dates back to 1981 when Rabin proposed his technique called Rabin Fingerprinting Scheme [8]. This was later used in many applications and systems and deduplication was one such.

Low-bandwidth Network File System (LBFS) by Muthitacharoen et al. [10] uses Rabin Fingerprinting Scheme to identify duplicates while copying data or updating file changes between LBFS client cache and LBFS server. This method was later deployed by a network storage system called Venti [11]. This formed the root for using Rabin Fingerprinting Scheme in deduplication storage systems.

When traditionally magnetic tapes were the preferred storage technology for backup and archival data, Venti [11] by Quinlan et al. changed this idea. It was the time when disk technology was approaching tape in cost per bit as mentioned in this work. Venti is an network storage system intended for archival data which was developed by Bell Labs. It uses write-once policy. The authors list the advantage of using hard disks over magnetic tapes for backup and archival. It uses collision-free hash values to represent data blocks of different sizes using complex data structures. This allowed incremental

backups which causes random seeks due to scattered data. Such random seeks might cause poor performance with tapes compared to disks. With the current advancement in storage technologies and large adoption of flash based storage, disks are considered fairly low speed storage devices. Tapes are being phased out. Hence, these random seeks in disks also impacts the restore speed causing significant concerns in system RTO.

Zhu et al. from Data Domain emphasized need for preserving spatial locality of data chunks and their metadata in [3]. They introduced the concept of container packing which preserves the spatial locality of the chunks. Additionally, they have discussed the need to write the metadata for all the chunks in a container into the same container in a smaller section called metadata section, in the same order as the chunks in the data section. This reduces the disk I/Os during restore as a cache miss for a chunk would fetch the complete container, thus serving the subsequent chunk reads as well. They claim that this stream-aware concept in their deduplication storage system is a major difference from Venti. However, it does not address the impact of chunk fragmentation [4] caused by duplicate chunks in the datastream.

Sparse Indexing [12] by Lillibridge et al. demonstrated a sampling based technique using a sparse index that fits into memory. The authors considered a group of chunks as a segment and identify similar segments by choosing and comparing the fingerprints of a few sample chunks. They deduplicate the current incoming segment against only the most similar segments that are already stored. By this they occasionally allow duplicate chunks to be stored.

Chunk Fragmentation Level (CFL) [4] by Nam et al. is a read performance indicator that could be used during the restore process to monitor the degradation in read performance continuously based on the chunk dispersion explained in Section-2. This is built over a theoretical model and the authors have investigated how this CFL value correlates with the read performance for a given datastream. They also take into

account the effect of chunk dispersion caused by concurrent datastreams during write process. Later, based on the this CFL value Nam et al. built a system [5] that provides the user with the read performance that is demanded. They achieve the demanded read performance through selective duplication of chunks during write process based on the constantly updated CFL value and a threshold.

Similarly, Kaczmarczyk et al. [6] built a system that rewrites highly fragmented chunks in the stream. They define two contexts, namely, stream context and disk context. Due to multiple datastreams that are streamed for backup, the locality (sequence) of the chunks in the disk may vary from how it appears in the streams. These two different contexts might cause high fragmentation of the duplicate chunks. Their goal was to make both these contexts similar by allowing duplicates to be written. The old copy of the chunk is reclaimed later in the background. They restrict the amount of duplicates allowed in the system to only 5% of the total duplicates. By doing this they also keep a check on the lost space savings. However, they do not discuss about the holes in disks caused by the reclamation of old chunks. This leads to disk fragmentation and becomes tedious to use these freed spaces unless the disks are defragmented.

The most recent work is from HP by Lillibridge et al. [7]. They investigated the behavior of chunk fragmentation and restore speed with different cache sizes. They have proposed two techniques, namely, container capping and forward assembly area methods. The former compromise deduplication to reduce chunk fragmentation by allowing duplicates. This is achieved by deduplicating sections of backup against only a few places. They place a cap on the number of container reads required per MB of backup data during restore. The latter exploits the advance knowledge of the chunk sequence in the stream during restore. They use efficient caching and prefetching method to enhance the restore speed by constructing a portion of the stream in advance using the paged-in containers in a buffer called forward assembly area. However, they did not

discuss about the computational cost of finding the positions of the chunks in this buffer. An efficient data structure is required to quickly scan through the chunk sequence to identify the positions, otherwise it will have an impact on the system performance.

Though [5, 6, 7] trade off the storage space savings (by allowing duplicates) similar to Reminiscent Read Cache system (by inducing duplicates), there is a major difference between these and the Reminiscent Read Cache system proposed in this work. While all of the above works (in case of [7] it is the container capping method) concentrate on write process to have a better read (restore) performance, this system acts during the read process itself. A common drawback in the other works is that they impact the write performance and trade off the backup speed for better read performance during restore. Whereas, this Reminiscent Read Cache system acts only during the read process and the write process is not disturbed. Hence, there is no degradation of the write performance. Also, through the preservation of spatial and temporal localities of duplicate chunks the read performance is improved.

Application of deduplication technique has seen a wider scope recently. Apart from its application in backup systems with secondary storage, it has now started to prove its significance at the primary data storage as well. Some of the works that concentrate on primary data deduplication are [13, 14, 15, 16, 17, 18]. Also, extensive studies have been made on the deduplication workloads [19, 20]. There are also works on deduplication across clusters like [2]. Data reliability is another interesting factor in deduplication. Since a single chunk is shared by many datastreams (in turn by many users or applications), it becomes important to protect it from getting lost. Nam et al. proposed a reliability-aware deduplication storage [21] that defines and uses two parameters, namely, chunk reliability and chunk loss severity. Also, flash-based indexing solutions have been built for deduplication storage like [22, 23] where chunk indexes are maintained in Solid State Drives.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this work, an elegant system called Reminiscent Read Cache based deduplication system is proposed and built to expedite the read process in Data Deduplication. An efficient cache is introduced that is capable of tracking the chunk access history in cached containers with the help of a Dupe Buffer. Unlike the earlier techniques the concept of variable sized containers has been introduced to reduce the data read from storage devices. This scheme also identifies portion of data to be duplicated which aids in improving the read performance. The proposed scheme was finally validated with experiments which proved to produce better read performance in data deduplication than the baseline scheme.

6.2 Future Research Direction

To further enhance this system, future research has been planned where advanced and customized pre-fetching and caching methodologies will be used when scaled to larger datasets with larger cache. This will also aid in handling multiple datastreams concurrently. With multiple datastreams each stream requires to have its own dedicated cache

space. Also, duplicates across datastreams will require the cache spaces to be aware of one another in a global namespace. Chunk fragmentation will have more adverse effects with multiple concurrent datastreams than a single stream.

The threshold value selection for minimum container utilization is currently static. This will be made adaptive in future. Also, depending on the workload this value will vary so that system delivers the optimal and expected performance. Workload characterization becomes critical in these scenarios. Without causing overhead the workload will be monitored and studied to make decisions on the threshold. The system will have a feedback module which can aid in the adaptive nature of the threshold.

Bibliography

[1] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. Hydrastor: a scalable secondary storage. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pages 197–210, Berkeley, CA, USA, 2009. USENIX Association.

[2] Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[3] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.

[4] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David H. C. Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications*, HPCC '11, pages 581–586, Washington, DC, USA, 2011. IEEE Computer Society.

[5] Young Jin Nam, Dongchul Park, and David H. C. Du. Assuring demanded read performance of data deduplication storage with backup datasets. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '12, pages 201–208, Washington, DC, USA, 2012. IEEE Computer Society.

[6] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 15:1–15:12, New York, NY, USA, 2012. ACM.

[7] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[8] Michael Rabin. Fingerprinting by random polynomials. *Harvard University Technical Report TR-15-81*, 1981.

[9] DiskSim v.4.0. *http://www.pdl.cmu.edu/DiskSim/*.

[10] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, 35(5):174–187, October 2001.

[11] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies*, FAST '02, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[12] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pages 111–123, Berkeley, CA, USA, 2009. USENIX Association.

[13] ZFS Deduplication. *blogs.oracle.com/bonwick/entry/zfs_dedup.*

[14] Linux SDFS. *http://www.opendedup.org.*

[15] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in san cluster file systems. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 8–8, Berkeley, CA, USA, 2009. USENIX Association.

[16] Yoshihiro Tsuchiya and Takashi Watanabe. Dblk: Deduplication for primary block storage. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, MSST '11, pages 1–5, Washington, DC, USA, 2011. IEEE Computer Society.

[17] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. idedup: latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[18] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Oltean, Jin Li, and Sudipta Sengupta. Primary data deduplication-large scale study and system design. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 26–26, Berkeley, CA, USA, 2012. USENIX Association.

[19] Nohhyun Park and David J. Lilja. Characterizing datasets for data deduplication in backup applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[20] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone,

Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[21] Youngjin Nam, Guanlin Lu, and David H. C. Du. Reliability-aware deduplication storage: Assuring chunk reliability and chunk loss severity. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–6, Washington, DC, USA, 2011. IEEE Computer Society.

[22] Guanlin Lu, Youngjin Nam, and David H. C. Du. Bloomstore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *MSST*, pages 1–11. IEEE, 2012.

[23] Biplob Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 16–16, Berkeley, CA, USA, 2010. USENIX Association.