University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Michał Kaczmarczyk

Fragmentation in storage systems with duplicate elimination

PhD dissertation

Supervisors:

prof. dr hab. Krzysztof Diks
Institute of Informatics
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw

dr Cezary Dubnicki
9LivesData LLC

January 2015

Author's declaration:
aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

January 15, 2015 .....
date *Michał Kaczmarczyk*

Supervisor's declaration:
the dissertation is ready to be reviewed

January 15, 2015 .....
date *prof. dr hab. Krzysztof Diks*

.....
dr Cezary Dubnicki

Abstract

Deduplication inevitably results in data fragmentation, because logically continuous data is scattered across many disk locations. Even though this significantly increases restore time from backup, the problem is still not well examined. In this work I close this gap by designing algorithms that reduce negative impact of fragmentation on restore time for two major types of fragmentation: internal and inter-version.

Internal stream fragmentation is caused by the blocks appearing many times within a single backup. Such phenomenon happens surprisingly often and can result in even three times lower restore bandwidth. With an algorithm utilizing available forward knowledge to enable efficient caching I managed to improve this result on average by 62%-88% with only about 5% extra memory used. Although these results are achieved with limited forward knowledge, they are very close to the ones measured with no such limitation.

Inter-version fragmentation is caused by duplicates from previous backups of the same backup set. Since such duplicates are very common due to repeated full backups containing a lot of unchanged data, this type of fragmentation may double the restore time after even a few backups. The context-based rewriting algorithm minimizes this effect by selectively rewriting a small percentage of duplicates during backup, limiting the bandwidth drop from 21.3% to 2.48% on average with only small increase in writing time and temporary space overhead.

The two algorithms combined end up in a very effective symbiosis resulting in an average 142% restore bandwidth increase with standard 256MB of per-stream cache memory. In many cases such setup achieves results close to the theoretical maximum achievable with unlimited cache size. Moreover, all the above experiments where performed assuming only one spindle, even though in majority of today's systems many spindles are used. In a sample setup with ten spindles, the restore bandwidth results are on average 5 times higher than in standard LRU case.

Keywords: deduplication, fragmentation, backup, caching, forward knowledge, restore, streaming access

ACM Classification: E.5, H.3.1

Streszczenie

Fragmentacja jest nieuniknioną konsekwencją deduplikacji, ponieważ pojedynczy strumień danych rozrzucany jest pomiędzy wiele lokalizacji na dysku. Fakt ten powoduje znaczące wydłużenie czasu odzyskiwania danych z kopii zapasowych. Mimo to, problem wciąż nie jest dobrze zbadany. Niniejsza praca wypełnia tę lukę poprzez propozycje algorytmów, które redukują negatywny wpływ fragmentacji na czas odczytu dla dwóch najważniejszych jej rodzajów: wewnętrznej fragmentacji strumienia oraz fragmentacji pomiędzy różnymi wersjami danych.

Wewnętrzna fragmentacja strumienia jest spowodowana blokami powtarzającymi się wielokrotnie w pojedynczym strumieniu danych. To zjawisko zdarza się zaskakująco często i powoduje nawet trzykrotnie niższą wydajność odczytu. Proponowany w tej pracy algorytm efektywnego zarządzania pamięcią, wykorzystujący dostępną wiedzę o danych, jest w stanie podnieść wydajność odczytu o 62-88%, używając przy tym tylko 5% dodatkowej pamięci.

Fragmentacja pomiędzy różnymi wersjami danych jest spowodowana duplikatami pochodzącymi z wcześniejszych zapisów tego samego zbioru danych. Ponieważ pełne kopie zapasowe tworzone są regularnie i zawierają duże ilości powtarzających się danych, takie duplikaty występują bardzo często. W przypadku późniejszego odczytu, ich obecność może powodować nawet podwojenie czasu potrzebnego na odzyskanie danych, po utworzeniu zaledwie kilku kopii zapasowych. Algorytm przepisywania kontekstowego minimalizuje ten efekt przez selektywne przepisywanie małej ilości duplikatów podczas zapisu. Takie postępowanie jest w stanie ograniczyć średni spadek wydajności odczytu z 21,3% do 2,48%, kosztem minimalnego zwiększenia czasu zapisu danych i wymagania niewielkiej przestrzeni dyskowej na pamięć tymczasową.

Obydwa algorytmy użyte razem działają jeszcze wydajniej, poprawiając przepustowość odczytu przeciętnie o 142% przy standardowej ilości 256MB pamięci cache dla każdego strumienia. Dodatkowo, ponieważ powyższe wyniki zakładają odczyt z jednego dysku, przeprowadzone zostały testy symulujące korzystanie z przepustowości wielu dysków, gdyż takie konfiguracje są bardzo częste w dzisiejszych systemach. Dla przykładu, używając dziecięciu dysków i proponowanych algorytmów, można osiągnąć średnio pięciokrotnie wyższą wydajność niż w standardowym podejściu z algorytmem typu LRU.

Słowa kluczowe: deduplikacja, fragmentacja, kopia zapasowa, pamięć cache, wiedza przyszła, odzyskiwanie danych, dostęp strumieniowy

Acknowledgments

I would like to express my greatest appreciation to my supervisors: Professor Krzysztof Diks, for giving me this opportunity, believing in the practical computer science with insights from a more theoretical perspective; and Dr Cezary Dubnicki for his close cooperation, precious insights during many consultations, his time and patience with my progress and sharing his professional attitude towards science.

Many thanks to my colleagues: Wojciech Kilian, Marcin Barczyński and Bartłomiej Romański. Your support with development of testing tools helped me a lot with performing so many experiments in an endless number of scenarios. Thank you also for all the discussions and the time spent reviewing my ideas and papers along with others: Dr Marek Biskup, Dr Michał Strojnowski, Michał Wełnicki and Krzysztof Lichota.

The companies of NEC and 9LivesData are the ones which should not be omitted here. Without the knowledge I could acquire while working for both of them and the hardware which I was allowed to use for testing, none of the below would take place. Additionally, I would like to thank the University of Warsaw for allowing me to publish this dissertation and friendly attitude throughout the time of our cooperation.

I would also like to thank my parents, my two great sisters and my grandmother Katarzyna for everything they did for me. Their love and support gave me the opportunities which in some aspect materialize at this moment. Special thanks to Professor Andrzej Potocki OP. You did an excellent job in encouraging me to work in critical moments. Your constant interest in my work (regardless your extremely different field of interests), passion for true science and spiritual care were a huge support for me throughout the last two years.

Last, but not least, my wife Monika. I cannot find words to express my gratitude not only for the (indirect) impact you made on this thesis. Your presence in my life changed everything to the level I could never dream of. Simple thank you is not enough, but there are moments when there is nothing more one can say. Therefore thank you for your love, inspiring passion for other people and attitude toward me and our son Jakub.

Contents

Chapter 1

Introduction

This chapter presents the background of backup systems. Next, it states the motivation of my research along with brief description of the fragmentation problem in popular systems with deduplication. The final sections show main contributions of the thesis and provide the outline for the whole dissertation.

1.1 Motivation

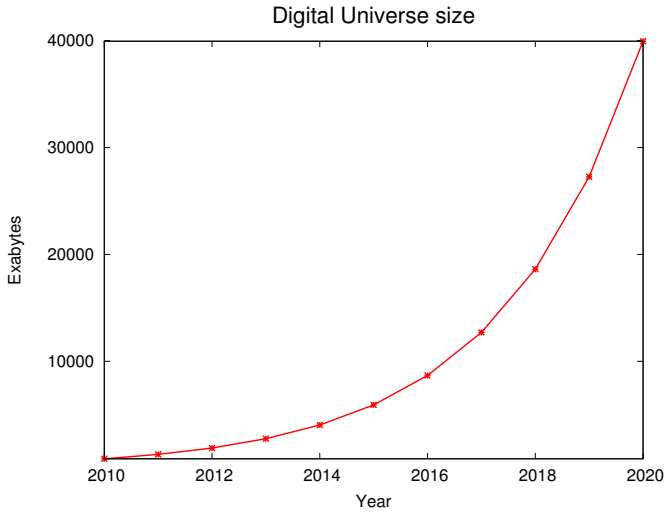


Figure 1.1: The Digital Universe size. Source: IDC's Digital Universe Study, December 2012 [35]

The digital world becomes bigger and bigger every day. Since 2007 [36] International Data Corporation has been sizing up what it calls the Digital Universe, or the amount of digital information created and replicated

in a year. The most recent study [35] shows that the digital universe will about double every two years to achieve an impressive number of 40 trillion gigabytes in 2020 (see Figure 1.1).

Since practically all of the new data created is stored digitally, the exponential growth in the amount of data created leads directly to a similar increase in the demand for storage. The average annual increase in the transactional data stored amounts to 30-50%. The growth of WORM data (write once, read many), e.g. medical data (such as X-rays), financial, insurance, multimedia data, is 100% per annum [19]. Additionally, in many areas, legislation [1, 60] requires keeping data for a long time, which further increases storage needs. It is easy to imagine the need to store company strategic data or information which cannot be easily recreated, but recent events have shown a demand for archiving even public Internet content in general. The reason for this is to preserve the Web for future generation as a space of "cultural importance". Such project is lead by British Library [11], and it has already collected thousands of websites in the British Internet together with their evolution over time.

The recent report [34] also shows that nearly 75% of our digital world is a copy, which means that only 25% of created data is unique. When we look at this number within secondary storage market it can indicate even less than 5% of unique data stored [37, 85]. This fact is one of the key reasons for systems with duplicate elimination to become very popular on the backup market since they appeared about 10 years ago. Having to store actually only a few percent of all the data significantly lowered the price of disk-based backup storage which enabled features such as an easy access to any backup from the past and efficient replication over a network for disaster recovery. Additionally, high write throughput delivered by systems available [57, 65] assures small backup window, which together with fractional storage cost makes more frequent backup service possible (both to schedule and keep).

As estimated [2], the market of such systems, called purpose-built backup appliance (PBBA), is to grow up to $5.8 billion (8.6 billion gigabytes shipped) in year 2016 from $2.4 billion (465 million gigabytes shipped) in 2011 (see figure 1.2).

Introducing secondary storage systems with duplicate elimination was enabled by key technologies such as distributed hash tables [24], stream chunking [68], erasure coding [82], fast duplicate elimination [85], to name a few. A lot of effort had been put into testing the effectiveness of the approach in reducing both: time needed to perform backups and storage space required to save them [23, 52, 67]. The effect is visible in the market popularity. Today, storage systems with data deduplication deliver new records of backup bandwidth [32, 57, 65] and the world is being flooded with various dedup solutions

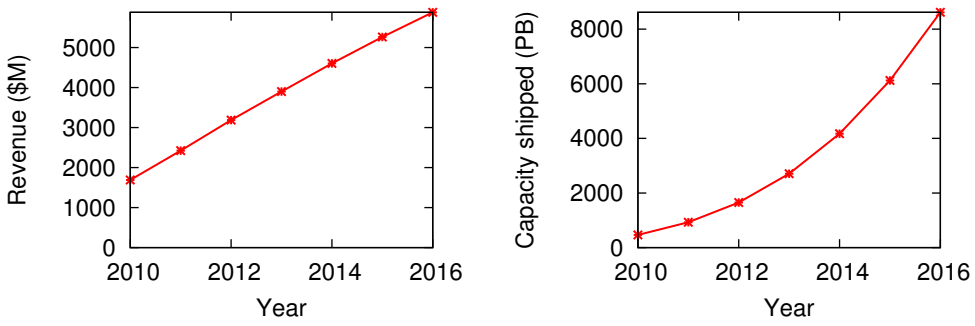


Figure 1.2: Worldwide Purpose-Built Backup Appliance Revenue (left) and Capacity Shipped (right) Forecast, 2010-2016. Source: IDC's Market Analysis, April 2012 [2]

proposed by many vendors [3, 26, 29, 31, 40, 56, 66, 75, 78]. In practice, deduplication has become one of indispensable features of backup systems [2, 4, 9] and a field of extensive research [23, 42, 45, 46, 54, 70, 76, 77, 80, 81, 85].

1.2 Problem statement

The data fragmentation on standard magnetic hard drives (HDDs) appears when two or more pieces of data used together are stored far form each other, therefore reducing the performance achieved with every access to them. Unfortunately, the problem of fragmentation in deduplication backup systems is strictly connected with their main feature – the deduplication itself. In most modern deduplication systems, before the data is written, it is chunked into relatively small blocks (e.g. 8KB). Only after the block uniqueness is verified, it is stored on the disk. Otherwise, the address of an already existing block is returned. As such block could potentially be stored far from the most recently written ones, the restore of the exactly same stream of data becomes inefficient. This is the place where the fragmentation story begins.

1.2.1 Impact of fragmentation on restore bandwidth

The restore bandwidth (time to recover data when needed) is one of the major factors describing performance of deduplication system, along with data deduplication ratio (storage space which can be saved) and maximal write performance (backup window length). Actual restore performance achieved by a regular customer in his working environment can often differ from the

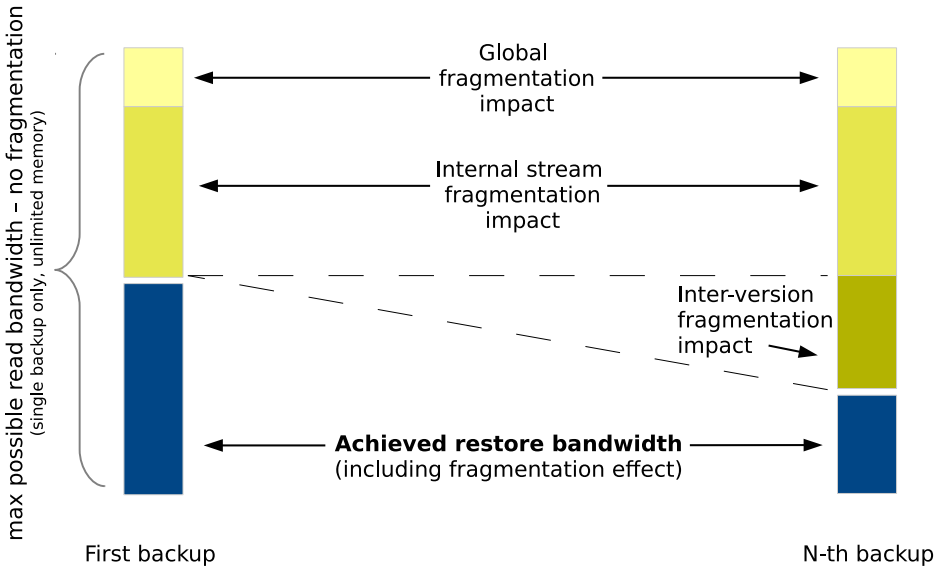


Figure 1.3: The schematic cost of having each kind of fragmentation as a decrease of the restore bandwidth achievable by a system with no deduplication.

ones showed by the system manufacturer for various reasons [48, 62, 63, 64, 83]. In particular, the restore bandwidth is usually moderately good for an initial backup saved to an empty system, but deteriorates for subsequent backups [42, 45, 54]. The primary reason for this are the different kinds of *data fragmentation* caused by deduplication. Those are:

- *inter-version fragmentation* - caused by periodical backups (daily, weekly, monthly) of the same data,
- *internal stream fragmentation* - caused by the same block appearing many times in a single backup,
- *global fragmentation* - caused by the same blocks appearing in backups with no logical connection to each other.

The schematic cost of having each of the above factors, appearing as a decrease in restore bandwidth, is presented in Figure 1.3. In this work I am going to look closer into the two main ones (as discovered during further analysis): inter-version fragmentation and internal stream fragmentation.

1.2.2 Inter-version fragmentation

Inter-version fragmentation can be observed only in systems with in-line deduplication, where the process of eliminating redundant blocks is done dur-

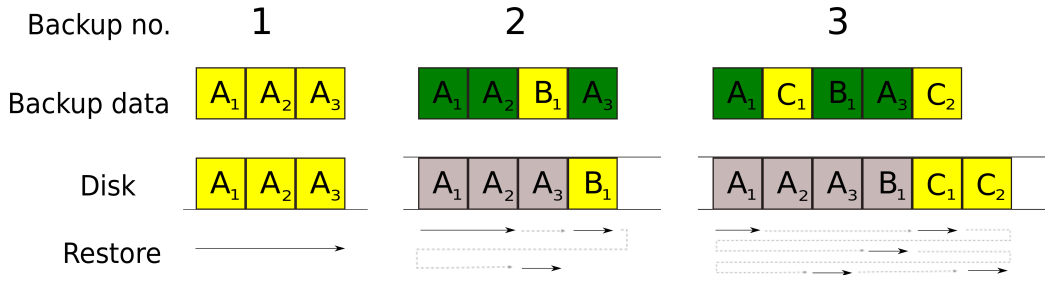


Figure 1.4: Backup inter-version fragmentation process caused by duplicate elimination.

ing the backup. Such systems are the most popular on today's market [2]. As in this solution duplicate blocks are never stored, such fragmentation results in data logically belonging to a recent backup scattered across multiple locations of older backups. This effect becomes bigger with each backup, as more and more of its data is actually located in an increasing number of previous backups implying increasing number of different disk locations. Depending on the data set, its characterization and backup pattern, my experiments show a decrease of read performance from a few percent up to more than 50%. As my data sets cover not more than 50 consecutive backups, I expect this percentage to be even higher when more backups are performed.

This most severe (as increasing) fragmentation of subsequent backups can be avoided with so-called post-process (off-line) forward-pointing deduplication. In such approach, a backup is written without any deduplication, and later the deduplication is performed in the background to preserve the latest copy of a block [47, 83]. As a result, the fragmentation does not increase and the latest backup does not become more fragmented with its age. Since the latest backup is the most likely to be restored, this solution looks promising. Unfortunately, it suffers from many problems, including (1) an increased storage consumption because of space needed for data before deduplication and (2) a significant reduction in write performance of highly duplicated data, because writing new copies of duplicates is usually much (a few times) slower than deduplicating such data in-line [32, 46]. The latter problem occurs because writing new data requires transferring it across the network and committing it to disk, whereas hash-based deduplication needs only comparison of a block hash against hashes of blocks stored in the system assuring much smaller resource usage (network, processor and disk).

To illustrate the inter-version fragmentation problem, let us assume a full backup of only one file system is saved every week to a system with backward-pointing deduplication. In such system the oldest copy of the block is pre-

served, as is the case with in-line deduplication, because the new copy is not even written.

Usually, a file system is not modified much between two backups and after the second backup many duplicates are detected and not stored again. In the end, the first backup is placed in continuous storage space and all the new blocks of the second backup are stored after the end of currently occupied area (see Figure 1.4). Such scenario is continued during following backups. After some number of backups, blocks from the latest backup are scattered all over the storage area. This results in large number of disk seeks needed for reading the data and in consequence, a very low read performance (see the restore process scheme of the last backup in Figure 1.4).

Such process can be very harmful to emergency restore, because the above scenario is typical to in-line deduplication and leads to the highest fragmentation of the backup written most recently – the one which will most likely be needed for restore when user data is lost.

1.2.3 Internal stream fragmentation

The factor which can also introduce a large restore bandwidth penalty is internal stream fragmentation. Even though it is caused by deduplication as the previous one, it is limited to a single backup only. This results in a different set of characteristics, such as rather constant impact on all the backup versions of the same data and the variety of deduplication backup systems affected (including off-line). My experiments have shown that internal stream deduplication, the exact cause of internal stream fragmentation, is usually quite significant as 17-33% of blocks from a single backup appeared more than once within the backup. By default, they are eliminated by deduplication mechanism therefore saving the precious space for user data. Unfortunately, this happens with a cost of up to 70% performance degradation which is visible when the restore is necessary. Further analyzes have also shown that the LRU caching algorithm, which is commonly used with restore in backup systems, does not work well in the described scenario, very often filling the memory with useless data.

To illustrate the internal stream fragmentation problem it is enough to backup a single stream of data, with some average number of internal duplicate blocks, to a system with deduplication. As the system will store only one copy of each block, the normally sequential backup will not be stored in such way on the disk any more (see Figure 1.5). This results in a large number of disk seeks needed for reading the data, and in consequence, a very low read performance (see the restore process scheme of the backup in Figure 1.5).

In the end, the internal data fragmentation causes both ineffective cache

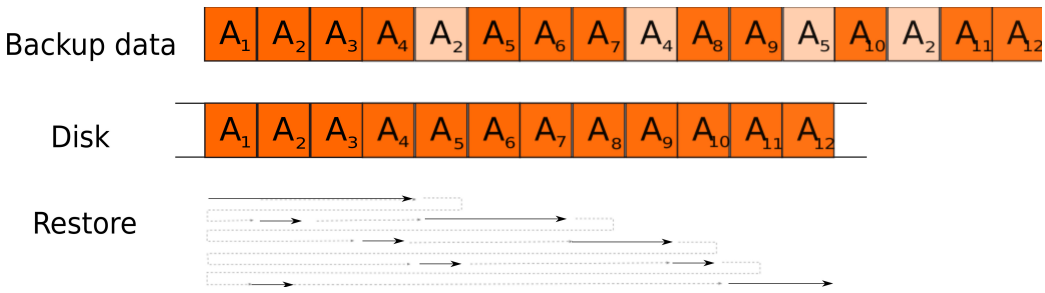


Figure 1.5: Backup internal stream fragmentation caused by duplicate elimination.

memory consumption and lower restore bandwidth. The problem characteristics, though, is much different to the one caused by inter-version fragmentation. First, the impact on the performance is more or less constant for all the backups from a data set, starting from the first one. Second, the problem affects all deduplication systems (including off-line) in equally significant way.

1.3 Thesis contributions

Considering the described scenarios, the goal of this work is to show how to avoid the reduction in restore performance caused by specific data location on disk in backup systems with deduplication. The result should be achieved with no impact on the original deduplication effectiveness nor backup performance. In other words, the ideal deduplication solution should provide high write bandwidth, as provided currently by the in-line approach, and high restore performance, without any read penalty caused by any kind of fragmentation.

The main contributions of this thesis are:

- detailed analysis and description of fragmentation problems specific to storage systems with deduplication (especially in-line) based on real traces gathered from users;
- identification of requirements and possible trade-offs for algorithms solving the problems found;
- proposal of *Intelligent Cache with Forward Knowledge* as a solution greatly improving read cache effectiveness and dealing with internal stream fragmentation by leveraging backup system characteristics;

- proposal of *Context Based Rewriting Algorithm (CBR)* to fight inter-version fragmentation with no deduplication loss and minimal backup write performance impact, together with a number of features addressing important trade-offs such as write bandwidth, latency, restore performance and temporary use of additional space;
- analysis of the requirements satisfaction and trade-offs resolution of the proposed algorithms, together with a set of experiments based on real user traces to prove the effectiveness of the chosen solutions;
- analysis of the scalability of fragmentation problem and the proposed solutions.

The experiments performed on real user traces proved the quality of presented algorithms. CBR limited the inter-version fragmentation impact from 21.3% to only 2.48% on average with an increase in backup time of not more than a few percent. The Intelligent Cache on the other side provided 62%-88% of average additional performance boost with only 5% of additional memory, achieving often the performance level of unlimited cache while having less than 256MB. The two algorithms combined end up in a very effective symbiosis resulting in an average 142% restore bandwidth increase, being often close to the maximal theoretical limitation. This effect is even brighter in larger configurations with many spindles. For example with ten disks, the combined algorithms are able to provide 8 times higher restore bandwidth while, with the same hardware extension, current approach showed only 60% gain.

1.4 Outline of dissertation

The thesis is organized as follows. The next chapter provides information about deduplication and storage systems in general. The motivation for this work, closer look at the nature of the problem of fragmentation, its different sources and a few examples are given in Chapter 3. Chapters 4 and 5 present solutions for two different issues which appear in storage systems with deduplication. Intelligent cache with forward knowledge tries to provide the effective usage of read cache in presence of internal stream fragmentation, while content based rewriting algorithm (CBR in short) deals with inter-stream fragmentation in order to assure the most effective block placement for future restoration of the most recent backup. Both solutions are followed by the discussion and trade-offs. Chapter 6 contains evaluation of both algorithms on real traces gathered from different users, including discussion

of performance results together with the assumptions and the methodology used in experiments. This chapter also includes both separate and joined experiments together with a section about scalability of both solutions. Related work is discussed in Chapter 7 together with other solutions to the fragmentation problem. Finally, Chapter 8 contains conclusions, insights on possible algorithm extensions and other directions for future work.

Chapter 2

Backup and Deduplication

This chapter provides detailed information on the historical background of processing backups. Second part describes duplicate elimination characteristics together with an analysis of benefits and drawbacks of the deduplication solution. Finally, the last section looks at the today's market in order to verify the importance of secondary storage with deduplication in a real world usage scenarios.

2.1 Secondary storage systems

2.1.1 Requirements

By its definition, backup is a copy of a file or other item made in case the original is lost or damaged. Such simple and easily looking task does not sound very challenging when it comes to a backup of a single desktop. The scenario changes dramatically though when we move into a medium to big size company with hundreds of users, terabytes of data produced every day and the requirement to perform backup every night or weekend (short backup window) for internal safety reasons. One cannot forget the backup policy which requires keeping many backups of the same data set (one for each day/week), which can differ by only a few bytes between each other, nor easy management of even a very large system (at petabytes level). Some value the possibility of setting individual resiliency for each set of data, while others see features such as deletion on demand (very complicated in distributed environment [77]) or uninterruptive update together with easy system extension as the most crucial ones. Easy and fast remote replication is also seen as an important addition together with the price - the lowest one possible. As one may expect, each of those two constraints usually introduce trade offs which

are not easily to be dealt with [23]. What is more, one need to remember about the main reason for backup systems to exist: the emergency restore. Without fast recovery to minimize the expensive downtime all other features seem much less attractive.

Characteristic	Primary storage	Secondary storage
Write bandwidth	in MBs/s	in TBs/h
Transactions latency	in milliseconds	in seconds
Restore start time	milliseconds	seconds to minutes
Data access	random	stream (mostly)
Data resiliency	optional	essential
Interface	any file system	VTL, OST, NFS, CIFS
Required capacity	in TBs	in PBs

Table 2.1: Primary vs secondary storage comparison

It is important to underline the differences between secondary (backup) and primary storage, which are required for the understanding of further sections (see Table 2.1). The latter systems are the ones used for everyday tasks in a similar way people use hard disks in their computers. While with backup, we would expect huge streaming throughput, data resiliency and maximal capacity, here the low latency will be crucial for all operations (read/write/delete), even the ones which would require random access [76]. On the other hand, in the same primary systems bandwidth and data resiliency, although important, will not be the one mostly required. Such small, but subtle difference becomes even bigger when we consider features such as compression, encryption and data deduplication taking place on the critical path of every backup operation.

2.1.2 History

Even though the first general-purpose computer was build in year 1946 and the backup evolution seems quite short, it is also a very intense one. The first available punched cards could store less than 100 bytes of data while the newest devices can keep more than 1TB. This huge leap in such short period of time show the amount of work put into developing the technology for every user to get the maximum out of the computer experience.

The punched cards, the first medium which could be considered as a backup, were already in use since the end of 19th century. With the appearance of

Figure 2.1: Storage of IBM record cards at the Federal Records Center in Alexandria, Virginia, November 1959. Source: www.archives.gov

computers they were easily adopted in order to become (in 1950s) the most widely used medium for data storage, entry, and processing in institutional computing. Punched cards were essential to computer programmers because they were used to store binary processing instructions for computers. In fact, NASA used punched cards and computers to read them in order to perform calculations as part of the first manned space flight to the moon. Luckily, punching an exact copy or two cards at once was an easy way to produce instant backups.

As the use of punched cards grew very fast, storing them became a hassle; eventually requiring large storage facilities to house cartons upon cartons of punched cards (see Figure 2.1). This problem was to be solved by magnetic tapes, which were becoming more and more popular. Even so, punched card programs were still in use until the mid-1980s [15, 41].

Since one roll of magnetic tape could store as much as ten thousands punch cards, it gradually became very popular as the primary medium for backup in 1960s. Its reliability, scalability and low cost were the main reasons for the success which made the technology to the top of most popular ways to perform backup in 1980s. During following years the technology had been improved in order to deliver higher bandwidth and better data density. In September 2000, a consortium initiated by Hewlett-Packard, IBM and Seagate (its tape division was spun-off as Certance and is now part of

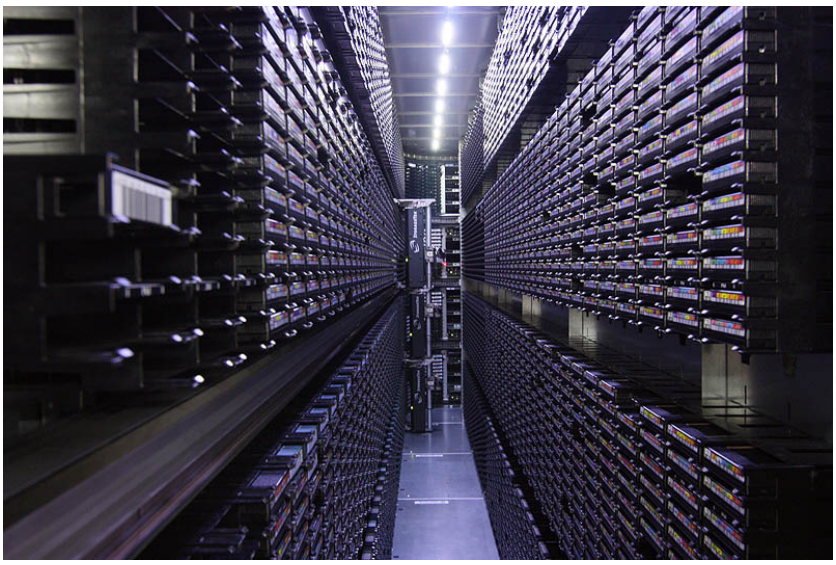


Figure 2.2: Tape Library Autoloader. Source: www.storagetutorials.com

Quantum Corp.) released the technology called Linear Tape-Open (LTO) Ultrium which introduced a common standard developed and used until now. The latest generation (LTO-6) was announced in June 2012 and delivered: 6.25TB capacity and data transfer rate at the level of 400MB/s together with features such as WORM (write once read many), encryption and partitioning [79]. In order to provide automation and transfer to/from many streams at once, the dedicated robots/libraries with many tape drives are available (see Figure 2.2).

Introduction of hard disk drives (HDD) did not change much in the backup market because of their high price, large size and low capacity. The new technology, which brought the possibility of random access to the data, first found its place in desktops, but at the end of 1980s it was used for the backup as well. Further development in this direction was possible thanks to the introduction of redundant array of independent disks (RAID), which are still common within the world of fairly small data, but the limitations of size and resiliency were too severe for medium and large companies. In year 2013 a single 3.5 inch hard drive could provide up to 4TB of capacity and over 200MB/s transfer rate. Even though those values are comparable with the ones available with modern tapes, the price to be paid is a few times higher.

Local Area Network supported by Network Attached Storage (NAS) and Storage Area Network (SAN) became the next big player in the backup market. Keeping the data remotely makes the backup more convenient (no additional media to attach), faster and easily replicable. Furthermore, the

Figure 2.3: Purpose-Built Backup Appliance example: A single node with 12 disks – the smallest building block of NEC HYDRAstor system (up to 165 of such nodes can work together in a single system). Source: www.nec.com

use of hard drives allows nearly instant access to any data and usage of algorithms such as deduplication, which can make backup more efficient and much cheaper. Since the new millennium, backup systems are not only attached through network, but they can form a separate living community of nodes being able to deliver features not possible before. Thanks to using many servers in a single system one can get intelligent data resiliency with automatic healing process in case of any disk, but even a machine or switch failure. What is more, the combined power of all computers can provide huge levels of throughput (over 900TB/hr [57]) and capacity (over 100PB [57]) in order to enable data collection from many different sources in a short backup window. Even though the systems available today are rather local or at the size of a data center, they can talk between each other to replicate data over a large distance and transfer only the data which is not duplicated. Software, on the other hand, provides a whole set of important features, enables easy management of a cluster and provides interface exporting the system as one unified space through network interfaces such as NFS or CIFS. Lower prices, potentially unlimited scaling possibilities and higher density of disk drives combined with deduplication technology and supported by remote replication, load balancing, fault tolerance and fast recovery made the systems, known as purpose-built backup appliances (see Figure 2.3), to be the first choice as the short-medium term backup solution today [2, 17].

Solid State Drives (SSD) would seem to be the logical successor to the current spindle-based disks in different kind of usage. They are fast, need less power and prevent problems such as access to large indexes and stream data fragmentation (no streaming access required any more). Unfortunately, they have a few considerable downsides which makes them not a good choice for business solution, especially the ones where large amounts of storage space are required. Even though we can find SSD drives with a price below $1 per GB, it is still far from $0.05, which is to be paid for a regular drive with spindles (own research: June 2013). With these prices and in general a few times smaller maximal capacity, it is difficult to predicate any revolution even taking into

account the fact that considerable price drop, we have experienced during recent years, continues. On the other hand, the small evolution is possible here and slowly takes place. As recent research suggests, SSD drives can be quite easily adopted for large indexes [44, 84] and for improving deduplication throughput [21, 50], which seem to be very useful in today's backup.

Over the last 30 years many other media have appeared which could be used as a backup solution but have not become popular especially in an enterprise environment. The most common devices were different kind of disks: floppy, compact (CD), versatile (DVD), HD-DVD, Blu-Ray. With each one the capacity, transfer rates and other indicators became better and better but they were still not enough to compete with hard disks or tapes. The main problems are as usual: the price, access time, too little storage space and complicated management.

The most recent idea of the backup is known as online backup and connected with the cloud concept. It is a strategy for backing up data that involves sending a copy of the data over a proprietary or public network to an off-site server. The server is usually hosted by a third-party service provider, who charges the backup customer a fee based on capacity, bandwidth or number of users. Online backup systems are typically built around a client software application that runs on a schedule determined by the level of service the customer has purchased. To reduce the amount of bandwidth consumed and the time it takes to transfer files, the service provider might only provide incremental backups after the initial full backup. Third-party cloud backup has gained popularity with small offices and home users because of its convenience, as major expenditures for additional hardware are not required and backups can be run dark, which means they can be run automatically without manual intervention. In the enterprise, cloud backup services are primarily being used for archiving non-critical data only. Traditional backup is a better solution for critical data that requires a short recovery time objective (RTO) because there are physical limits for how much data can be moved in a given amount of time over a network. When a large amount of data needs to be recovered, it may need to be shipped on tape or some other portable storage media [71]. The most important issues here are also the data security, availability, privacy and the risk of using the data by the service provider in some undefined way. Especially large companies will prefer keeping the sensitive data in their own system without taking a risk of giving the control away. It is important to state that the technology used here remains basically the same or very similar to described above network backup. What is different is the required agreement between sides, software being used and the concept of interaction between customer and service provider.

2.2 Duplicate elimination

Deduplication is usually defined as a technology that eliminates redundant data. When data is deduplicated, a single instance of duplicate information is retained while the duplicate instances are replaced with pointers to this single copy. The whole process is completely hidden from users and applications, which makes it easy to use and not require any dedicated software modifications.

In order to be easily compared and found, each piece of data requires a unique identifier which is much shorter (about 160 bits) than the data itself (usually about 8KB). In secondary storage, such identifier is calculated based on the content of data to be stored and makes it easy to locate any existing incoming piece of data using dedicated indexes. Systems which identify their data in such way are defined as Content Addressable Storage (CAS) and have been an area of research for more than 10 years already [67].

Deduplication is sometimes confused with compression, another technique for reducing storage requirements. While deduplication eliminates redundant data, compression uses algorithms to save data more concisely. Some compression is lossless, meaning that no data is lost in the process, but "lossy" compression, which is frequently used with audio and video files, actually deletes some of the less-important data included in a file in order to save space. By contrast, deduplication only eliminates extra copies of data; none of the original data is lost. Also, compression doesn't get rid of duplicated data – the storage system could still contain multiple copies of compressed files.

2.2.1 Characteristics

Granularity

Data deduplication can generally operate at the file or block level. The former one eliminates duplicate files, but is not a very efficient way of deduplication in general as any minimal modification requires to store the whole file again as a different one [61]. Block deduplication looks within a file and chunks it into small blocks. Each such block is then processed using a hash algorithm such as SHA-1 or SHA-256 in order to generate a unique hash number which is stored and indexed. If a file is updated, only the changed data is stored. That is, if only a few bytes of a document or presentation are changed, only the changed blocks are saved. This behavior makes block deduplication far more efficient. However, block deduplication takes more processing power and uses a much larger index to track the individual pieces.

Algorithm

Two main abstractions for duplicate elimination algorithm on block level are called: fixed and variable size chunking. After a number of tests it turned out that having blocks of fixed length does not work well with possible updates [67]. By simple modification of a few bytes at the beginning or in the middle of a file all the following content had to be rewritten as new data with different block boundaries in order to preserve its size. Variable chunking length [23, 51, 85], on the other hand, makes use of a dedicated algorithm (such as Rabin fingerprinting [68]) which enables synchronization of block boundaries shortly after any modification takes place. Thanks to that, the following part of the modified file can be cut into the identical blocks which can then be deduplicated to those already present after backup of the unmodified original file.

Usually, a block size produced in such way in modern systems is within some boundaries (e.g. 4-12KB) with an average value somewhere in the middle. The most common average values used are between 4KB and 64KB and have significant impact on overall deduplication ratio along with some other system features such as the scope of deduplication, data fragmentation. Some dedicated algorithms try to optimize this impact by allowing usage of many different block sizes during a single backup (i.e. 64KB with 8KB). As research shows [43, 70], the results are quite promising.

Point of Application

A secondary storage system is being used by a set of clients performing backup. Each backup stream requires to be chunked into blocks together with hash calculation for each one of them in order to verify its existence in the system. Those operations can take place either on the client or server side. The former one, called source deduplication, will require dedicated software to be installed on the client, but at the cost of some processing power (hash calculation) it can offer much lower network usage. The latter, on the other hand, called target deduplication, is completely transparent for the clients, simply providing the storage space through network interfaces and therefore extremely easy to use performing the hashing and all other required operation internally. Both options are available on the market and deployed based on customer requirements.

Time of Application

Within systems with target deduplication there are two groups which differ in time when the process is applied. Off-line (post-processing) deduplication [62,

63, 75] is the simplest way where, in the first phase, all data from the current backup are stored continuously in the system. After the operation is finished the actual deduplication is performed in the background in such a way that the blocks from the latest backup are a base for eliminating duplicates from older backups [47, 62]. On one hand, such approach makes sure that all the data from newest backup is located in one continuous space, which makes it easier to read, but on the other, it causes a number of different issues. The problem though is with even a few times lower backup performance, lack of possibility to conserve network or disk bandwidth (i.e. deduplication on client or backup server) and the space required to hold an entire backup window's worth of raw data (landing zone). Even though the landing zone can be minimized by starting the deduplication process earlier and performing it part by part (staging), the system resources needed for that operation will make the current backup slower, which would add one more negative effect [13]. What is more, the off-line process becomes quite expensive as after each backup about 95% of its size (assuming 20:1 dedup ratio) has to be found in entire storage space and deleted in the background.

The other kind, called in-line deduplication, makes sure the duplicate data is found during the write process and never stores a block which is already present in the system. It requires fast algorithms in order to verify the block existence on the fly and return either the duplicated or new pointer, depending on the result. Such path is complicated in general, but by making sure that no duplicate data is found in the system, it does not require any cleanup after the backup. Also, as checking the hash existence (often in index placed in memory [85]) can be three times faster [42] than storing a new block on disk, it delivers much better bandwidth. The problem with such approach is a progressing fragmentation, which will be described in details in the next chapters of this work.

Scope

The final characteristic of deduplication is connected with its scope. The most intuitive global version, where each duplicate block existing in the system is always identified, is not that common because of the implementation and technical issues which appear. The main problem is with the huge global index, which should always be up to date and allow fast identification of required block. One of the issues here is to identify whether a block is a duplicate or not. This is often done with a Bloom filter [10] and used by distributed systems such as Google's Big Table [16] and DataDomain [85]. It helps to avoid expensive look-up for blocks which will not be found. On the other hand, techniques such as using larger block size [23] and exploiting chunk

locality for index caching as well as for laying out chunks on disk [69, 85] reduce the amount to data required in RAM. As a result, only small percentage of requests needs an access to the full index which is placed on disk. When we move into distributed environment the problem is even more complicated, which results in only one commercially available system with global deduplication (HYDRAstor [23]), which uses a dedicated Distributed Hash Table [24] in order to deal with the task.

Other existing solutions are either centralized ones (such as EMC Data Domain) or use a different kind of techniques limiting the required memory at the cost of deduplication. Sparse Indexing [46], for example, is a technique to allow deduplication only to a few most similar segments based on a calculated hash, while Extreme Binning [7] exploits file similarity in order to achieve better results for workloads consisting of individual files with low locality.

2.2.2 Deduplication ratio

Deduplication ratio (or duplicate elimination ration, dedup ratio) is defined as a data size reduction ratio achieved thanks to deduplication technology. For example if about 100TB of data was written into a systems and thanks to the deduplication only 5TB where actually stored on disks the deduplication ratio would be shown as 20:1. The actual values can vary widely depending on data stream characterization, chunking algorithm, block size and retention policy. As research articles confirm, metadata size in relation to all the stored data must also be taken into consideration [70] together with performance required to calculate the hashing or update the metadata and store/locate the data. At last, one needs to remember about the issues with scalability of the system and the time to reconstruct the data. All of the above certainly impacts the deduplication ratio, which can range from 4:1 to 200:1 and more [49]. When aggregated, a compression of 10-20 times or more (less than 5% of the original storage capacity) can be achieved, which is with some deviation confirmed with other sources, both business [4, 5] and scientific [45, 70, 85].

Most modern backup systems use variable size chunking, because of its advantages described in Section 2.2.1. As it was shown in many articles [55, 70], the average value target of variable block size has a noticeable impact on the data deduplication ratio. When looking at the data only one can always expect smaller blocks to perform better in terms of space savings, but needs to remember about the problems which can appear. The usage of small blocks cause higher memory requirements (bigger index), backup performance degradation (more blocks to verify and deliver), and data fragmentation (smaller random reads possible) causing restore bandwidth prob-

Policy name	One bkp size	Modified data in each bkp	Number of bkps	Sum data stored	Sum data written	Dedup ratio
Daily	1 TB	1%	365	4.65 TB	365 TB	78.49
Weekly	1 TB	6.79%	52	4.53 TB	52 TB	11.48
Monthly	1 TB	26.03%	12	4.12 TB	12 TB	2.91

Table 2.2: Different backup policy vs deduplication ratio - simulation results (assuming the random data modification rate at 1% per day; no compression, initial backup and each modification consists of only unique blocks)

lems. What is more, each block of data requires a small, but noticeable piece of metadata stored which does not depend on the data size. Unfortunately, when taken into account, it may waste all the savings provided by applying smaller block size. When looking at the market the most common block size used is 8KB (i.e. EMC Data Domain - global leader [29]), but there exists competition with block size even 64KB (NEC HYDRAstor [57]) or 4KB (HP StoreOnce [39]) on the other side.

After all, every single backup will deduplicate best with some individually defined block size. Furthermore, in order to achieve best results each part of a stream could be divided into different size segments regarding its modification scheme. Even though in general the problem looks extremely complicated, some simplified solutions appeared letting to use two sizes of blocks during a single backup. The decision on whether the block should be small or large is based on the previously stored information. According to Romanski et al. [70] such approach can result in 15% to 25% dedup ratio improvement achieved with almost 3 times larger average block size.

Often underestimated factor, when calculating duplicate elimination ratio, is a retention policy. As the biggest power of deduplication comes from elimination of duplicates from previous backup of the same data, the information about number of such backups is crucial for the purpose of calculations. Lets assume the size of our example file system to be 1 TB and the modification rate at a level of 1% of blocks per day (to simplify the calculation we assume that our backup does not increase in size and random blocks are modified every day). Having such system, user can choose one of three simple backup policies: daily, weekly and monthly. Each of them defines a frequency of the full backup to be performed. After a year with each of the policies, we will end up with having similar amount of data occupied in our system (4.1TB-4.6TB), but with significantly different amount of data

written (12TB-365TB). Therefore, each of them calculates into a completely contrasting deduplication ratios: 78.49, 11.48 and 2.91 (see Table 2.2). Each policy is simply unique and at different costs (i.e. time spend on backup during month) protects data in a different way. The calculation shows only the fact that each specified case is unique and taking only deduplication ratio into account has its own drawbacks. In general the average number of duplicates in a backup (except the initial one) seems to be more precise as an indicator of deduplication power.

Similar effect can be achieved when choosing between incremental and full backup. The former one will most probably take less time to perform but more to finally restore the data as the latest full backup and all incrementals until given time need to be patched together. The latter one, even though it takes more time, thanks to deduplication it will not consume more storage space. It is also important to note that from a statistical point of view even though the data stored is similar, the final deduplication ratio in both cases will look much different.

The compression is one more task usually applied before the data is stored in the system. Keeping only essential data may need more processor power to compress and possibly decompress in the future, but can often increase the overall data reduction ratio (compression together with deduplication ratio) by a factor of 2 or more. Such space saving is usually worth the effort especially with larger block sizes, where compression becomes more effective.

Finally, the basic impact on the deduplication ratio has the individual backup stream characteristic. The stream content and it's internal redundancy is an important start. Taking for example mailboxes, the first backup may result in less than 50% of unique data sored in the system (improving deduplication ration by a factor of 2), while having the first backup of a movie database will not show any savings at all. Starting from the second backup the percentage of duplicates usually stabilizes but at different level for each data set. It depends mostly on the modification rate/pattern and the period between backups. Those two numbers combined with a number of full backups kept in the system will have a major impact on the final score achieved.

2.2.3 Benefits

Although the deduplication can be used in any environment, it is ideal for highly redundant operations such as backup, which requires repeatedly copying and storing the same data set multiple times for recovery purposes over 30- to 90-day periods. The described usage pattern makes the technology to be especially useful ending with over 20 times reduction of the data to be

stored (depending on many different features - see Section 2.2.2 for details). Such result can end up in high money savings or enable possibilities not achievable before.

Probably the most important result of introducing data deduplication in secondary storage is a huge technological leap in the area. Thanks to limiting required storage space, it enables the previously expensive disk based systems to compete with tapes bringing into secondary storage world features not available before. Those are: immediate and random access to the data (emergency restore), high transfer rates, one combined storage space, many streams of backup, cheap and definable data resiliency class, easy and fast replication, maintained data integrity.

What is more, having the possibility to verify the existence of data based on short (i.e. 160 bit) hash of the data opens a way to save network bandwidth. A dedicated software may be used to produce hashes at the client (source deduplication - see Section 2.2.1) and send only the data which are not present in the system. Assuming the hash size lower than 0.5% of the data size and 20:1 deduplication ratio, only 5.5% of all the data needs to be transferred over the network in order to perform a regular backup. Such approach not only makes the process much faster (to make the backup window smaller), but also it does not require the network from client to allow high bandwidth vales. This feature is even more important in case of replication when master and replica sides are placed in different states or countries.

Overall, the data deduplication technology is not only a single feature added to an existing software. It is a start of a whole new era in secondary storage – the era of servers and hard disk with all the features they provide such as instant random access, extremely high bandwidths, constant data monitoring. Supported by network saving replication and the competitive price, it creates a complete and well equipped solution in terms of secondary storage.

2.2.4 Drawbacks and concerns

Whenever the data is transformed in any way users may be concerned about their integrity. The deduplication process looks for the same copy of a block somewhere in the system and may end up with the data of one stream scattered over many locations on disks and servers. Such way of saving storage space makes it almost impossible to read the required data without the exact recipe stored somewhere in the metadata and in the exact opposite way it was written. All this put high requirements on the quality of the software from vendors and implies fair amount of trust in the process from the customers.

Each deduplication system has to be able to find and compare the blocks

in order to verify their identity. As described before, the hash function is an easy and effective way to find a candidate for verification but it turns out that reading such candidate in order to verify its content with newly written block byte by byte would make the storing process very time consuming. In order to remove this overhead the industry relies on hash comparison only in order to determine the identity of two blocks. Of course a single hash of length 160 or 256 bit in theory can be used to identify a lot of 8KB blocks but as it was verified, assuming the collision-resistant function (i.e. SHA-1 [25]) and the amount of blocks which can be stored in a system, the probability of such collision is extremely low, many orders of magnitude smaller than hardware error rates [67]. Though when the data corruption appears it will most probably be rather a problem with IO bus, memory or other hardware components.

One more concern is connected with computational power necessary to perform the algorithm and other required functions. This is the case in source deduplication, as this option requires at least some of the calculations to be performed on the client machine. Such additional cost should be calculated in the early phase of comparing the solutions before purchase. Alternatively, the system with target deduplication can be used, as the one without any power requirements on the client.

Finally, going into a system with many disks and tens or hundreds of servers keeping all the data accessible and not loosing any may be an issue. Such system requires efficient distributed algorithms, self-healing capabilities and incorporated intelligence in order to allow fairly easy management. With thousands of disks the probability of braking one becomes quite high, so that features allowing easy disk/node replacement without spoiling the overall availability become important. Fortunately, there exist systems having all the above features being able to work in configurations with over 100 nodes assuring even 7.9PB of raw capacity [57].

2.3 Today's market

According to Information Storage Industry Consortium (INSIC) the use of tape technology, the most common during last 30 years as secondary storage, has recently been undergoing a transition [17]. The type systems are moving out from backup system market towards third tier backup (quite recently created category for long time retention backup with infrequent or no access), archive (data moved for longer term storage) and regulatory compliance data (preserved for duration defined by regulation). All those use cases involve keeping a single copy of data for a long time often without reading it at

Disk	Tape
Advantages: • Rapid random access of single/small files • Multiple stream handling • Read/write throughput (since one stream is usually stored on many hard drives) [65] • Deduplication to archive high compression ratios, enable remote replication and other functions • Many formats of backup • Manageable reliability with large systems	Advantages: • Low energy usage and cost per GB of data stored • Data life span of up to 30 years • Significantly higher reliability than single SATA disk
Appropriate use: • Fast storage and data retrieval in any way • Active, newly created and frequently accessed data • Multi-stream access • Redundant data (i.e. backup) • Efficient replication	Appropriate use: • Storage and restore of large files or quantities of streaming data • Less heavily accessed data • Data that must be retained for some period of time and are rarely or never read • Unique data

Table 2.3: Disk and tape comparison [17]

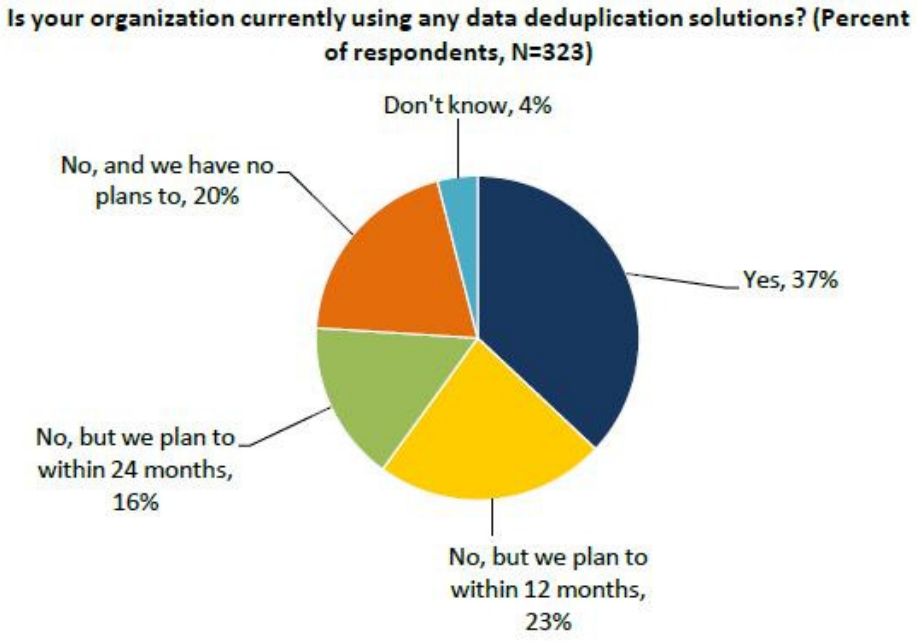


Figure 2.4: Survey on data deduplication solutions in organizations. Source: Enterprise Strategy Group, 2012 Data Protection Trends

all. For those purposes tape may still be a better choice due to the price, better duration, smaller energy cost and no deduplication requirement (see Table 2.3).

The above tendency is also visible when asking organizations about using data deduplication solutions. The survey performed in year 2012 by Enterprise Strategy Group (see Figure 2.4) on over 300 respondents had shown 76% of them having used or planning to use a deduplication solution (compared with 43% in year 2008 [30]). On the other hand there are numbers developed by the market itself. The whole tape market (with its media and robotics, including archiving and other purposes) in year 2011 closed in total of $3 billion [59] (after 10% drop) while for deduplication systems it was $2.4 billion [2] (after 43% grow). While the tape market was still bigger, it looks like the usual 20x deduplication ratio, high write bandwidth, scalability, the ease of remote replication and fast emergency restore are considered important when the decision is to be made at the company.

Even though the deduplication systems grow at the extensive rate, they are most probably not going to eliminate tape usage totally. As data collected from companies suggest [17], they are rather going to use both disk based and tape systems for the backup (62% in year 2010 comparing to 53% in year 2008). Taking all the above information into perspective, there seems to be

a tendency to use the disk-AND-tape model as the most successful methodology for data protection with disk-based systems as a main component for backup up to 6 months and tapes used for archive and data requiring longer retention period.

There is no doubt that thanks to deduplication the second big step in global secondary storage is in progress (the first one was the transition from punched cards to tapes in 1980s). On the other hand, the number of published papers for last few years places the topic under extensive research [23, 42, 45, 46, 54, 70, 76, 77, 80, 81, 85]. At this scale each innovative approach, algorithm or discovery may end up having large impact on everyone from vendors to systems administrators worldwide. Even though a lot of knowledge has already been presented, there are still strategic areas waiting to be discovered. One of them is stream fragmentation, as a side effect of deduplication, and critical restore in general.

Chapter 3

The problem of stream fragmentation

This chapter describes the importance of restore process in backup systems. Next, it presents the description of a problem caused by different aspects of stream fragmentation with a separate section addressing the scalability issues. The detailed impact of each fragmentation aspect is presented based on real world workloads to show the nature and size of the problem. Final discussion summarizes the basic options to reduce the negative impact of fragmentation.

3.1 The role of restore in backup systems

Even though restore does not happen as often as backup, it is used not only in case of lost data but also in order to stream the full backup to tape (third tier backup) and replicate changed data off-site. As a result, there exist even 20% of systems with actually more reads than writes, while on average reads are responsible for about 26% (mean; 9% median) of all the I/Os in an average backup system even when the replication activity is excluded [81].

Each attempt to restore data from a backup system can be caused by a number of reasons. Accidentally deleted file or access to a previous version of some document are actually one of the simplest requests to handle in a short time when considering disk based systems with easy random access to all the data. On the other hand, restoring full backups consisting of many GBs of data is a whole different problem of providing the maximal bandwidth for many hours. Even though such scenario does not necessarily mean some outage in the company (it can be a transfer of the data to some other place) this should be the case to be handled extremely well in the first place. The

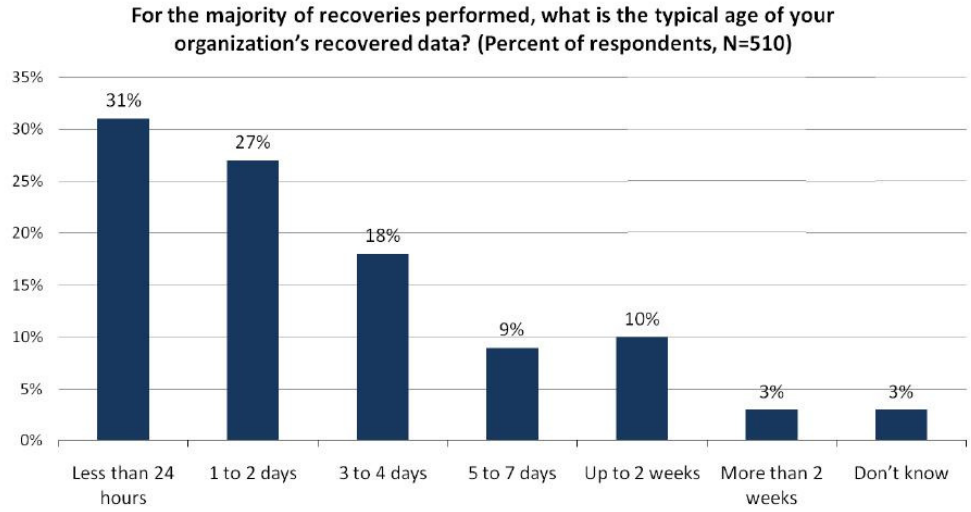


Figure 3.1: Typical age of recovered data. Source: ESG Research Report "2010 Data Protection Trends" [30]

recovery time objective (RTO), being one of the most important factors of any backup system specification, actually makes the investment of thousands of dollars in a backup system rational for a vast majority of companies. Every emergency issue in this area may be seen as a major test for the backup system and the final verification of the investment for the company.

When analyzing the usual restore process some of its characteristics can be noticed. Very important one is the fact that not every backup has the same significance, which makes the restore process valued differently. First, it is the data itself which may be simply less critical for the company. Second, it is the time when the backup was taken and its usefulness for restore in case of emergency. Figure 3.1 shows a result of a survey performed by Enterprise Strategy Group on 510 respondents. Not surprisingly, the data restored most often are the ones backed up very recently. Based on the results only 6% of restores are older then two weeks and the majority of them (58%) is recovered from last 48 hours.

To sum up, the big picture which appears above makes a clear goal for the verification of a backup system true value. It is the restore bandwidth of the latest backup. Even though this statement sounds very trivial, it has major consequences especially for the backup systems with deduplication, which are very close to become the most common in today's world and during the years to come.

3.1.1 Backup procedure

Each company has its own backup policy, which should be the best answer to the data safety and disaster recovery requirements. One of the most common strategies is to perform a backup of all company data during the weekend and smaller, incremental backups every day [81]. This is usually caused by a very limited backup window every working day (the time available for the backup to finish) and a larger one during the weekend. When using deduplication system, the full backup can be performed even every day, as with such solution only new and modified data is actually stored (its size is more or less equal to the incremental backup), while all the other duplicate data are confirmed very quickly to the backup application making the process many times shorter than regular full backup.

Next characteristic to the backup policy is the retention period which may also be different in many companies [18]. The original idea was to limit the space used for backups which were less likely to be helpful in case of emergency restore. Usually the choice was to keep some (usually 5-30) most recent daily backups, about 4-26 weekly backups, close to 12-24 monthly backups and a few yearly. Very often the backups older than 3-6 months were moved to the so-called archive storage, which implies extremely low probability of usefulness. After introduction of deduplication systems the scenario is slowly changing. Thanks to the new technology each additional backup add almost no new data to the storage space, therefore, a company can keep daily backups for a year paying only slightly more (metadata) than keeping only the actual data and modifications. Having such technology makes keeping high granularity of backups possible at an extremely low price, which may eventually help to recover the exact state of given documents from the required date regardless of the time passed.

When looking at the single backup procedure one can notice another simple, but very important fact, which is related to data order and placement. Each storage system usually receives data in a so called stream: a sequence of bytes in some defined, logical order with beginning and end. Usually a backup application is responsible for creating such stream from a file system or directory which is to be stored. In case of storage file system mounted directly through NFS or CIFS such stream is equivalent to each transferred file (usually a quite big tar archive). Having a logical order each stream guarantees that every access to its data will be done sequentially and in the same order as it was written. This assumption is important for all backup systems, enabling them to achieve good performance. The access to data in a non sequential way would make those systems not usable from the market perspective [67, 85].

Prefetch size (in KB)	Time of one single prefetched read operation (I/O) in ms	Average time to read 1GB of data (in s)	Read time difference in % of 2MB prefetch result
16384	104.10	6.26	-46.00%
8192	58.38	7.47	-39.43%
4096	35.52	9.09	-26.28%
2048	24.10	12.34	0.00%
1024	18.38	18.82	+52.57%
512	15.52	31.79	+157.71%
128	13.38	109.62	+788.54%

Table 3.1: Impact of prefetch size on the actual restore time based on common enterprise data center capacity HDD specification [72, 73] (sustained data transfer rate: 175MB/s, read access time: 12.67ms (average read seek time: 8.5ms, average rotational latency: 4.17ms [7200 rpm]))

3.1.2 Verified combination: Prefetch and cache

The sequential access to data significantly helps to reduce the problem of the biggest bottleneck in restore performance, which is reading the data from the actual hard drive. Having the fact of optimal data placement, when it comes to popular HDDs, enables engineers to use simple but effective techniques improving the restore performance many times, when compared to the random or undefined data access pattern.

Prefetch

The most common and effective technique in case of sequential data is to prefetch it in fixed or variable big chunks from the hard drive to system memory. In the result of such operation user read request of only one block (e.g. 8KB) triggers read from disks of a much larger chunk (e.g. 2MB) placing all the read blocks (2MB / 8KB = 256) in system memory for further use. Thanks to such approach, in the case of sequential access it enables many following read operations to retrieve the data from the memory without paying the price of disk access.

This algorithm is actually a consequence of the HDD construction, which

makes reading small portions of data very inefficient. The two main characteristics for each disk are: the data access time and transfer rate. The first one is the most problematic here. Before starting to transfer the data to the system memory, the disk has to move its head to the proper track (seek time) and wait for the required data to appear under the head (rotational latency). The whole process is very expensive and assuming constant transfer rate, the number of such data accesses determines the total read performance (see Table 3.1).

In addition, it is important to notice that the disk technology in terms of bandwidth and capacity is in constant development. Unfortunately at the same time both seek time and number of rotations stay basically at the same level for many years already. In fact, as this work was almost completed, Seagate announced new version of their Enterprise Capacity 3.5 HDD with 29% higher sustained data transfer rate (226MB/s), but with no changes in the read access time [74]. Such unequal development makes the problem of fragmentation even more severe as accessing the data alone is taking larger and larger part of the total restore time.

Cache

After the data is prefetched from disk it is stored into a dedicated system memory called buffer cache (or read cache), which is usually much larger than the actual prefetch. The reason for that is lack of the ideal sequential load in the reality. In case of a small cache each non-sequential disruption (read from a different place on disk) would require reloading the data after coming back to the previous read sequence. Thanks to a larger size the cache can be not only resilient to some extent in the described scenario but also support read in not exact order (in case of data reordering during write) and the access to many streams at the same time. In case of duplicate elimination backup systems one more function of the cache becomes quite interesting and important. It can simply hold blocks of data, which are requested many times during a relatively short period, allowing additional improvement in the achieved restore bandwidth.

As the memory for cache is always limited it requires dedicated cache eviction/replacement policy. Each of many existing algorithms has its own best suitable usage. For the backup systems the most commonly used policy is Least Recently Used (LRU) [45, 54, 81, 85]. The main goal in this case is to discard the least recently used blocks first. Although the algorithm requires keeping track of what was used and when to make sure the correct item is removed, some optimizations exist to make it less expensive. The experiments with a few other well known algorithms such as Most Recently

Used and Least-Frequently Used on the traces presented in this work also showed the much better results with the LRU.

It is important to state that for the page replacement policy (which is somewhat similar) the most efficient algorithm actually exists and is called: Bélády's optimal algorithm [6]. In order to achieve the optimal cache usage it first discards the page from memory that will not be needed for the longest time in the future. Unfortunately, since in general it is not possible to predict when the information will be needed, the algorithm is not implementable in practice for the majority of known scenarios. Also, the pages in memory differ from blocks so moving it into backup system environment is not straightforward but can bring interesting insights.

Efficiency issues

Even though the prefetch/cache algorithm effectively helps achieving reasonable restore bandwidth, it sometimes does not work very well. One case is when the access pattern is actually only partly sequential. Such pattern results in reading from disk possibly a lot of data which will never be used, and waste both the time during the actual read operation and the space in the memory, which effectively makes the cache even a few times smaller than actual memory reserved.

The other problem is connected with blocks loaded to cache many times. Such scenario may happen in case the block was either evicted from cache before it was used (too small cache or too random access) or even though it was already used it was required more than once (internal stream duplicate). When it comes to backup systems with duplicate elimination, especially the second scenario was surprisingly intensive in the traces I have explored even within one sequential stream of data.

3.2 Fragmentation problem in systems with duplicate elimination

In general fragmentation is defined as a state of being broken into fragments. For the purpose of this work we focus on a sequential stream of data which is backed up and the way it is stored on the disk drive in systems with duplicate elimination. As we are generally interested in the practical more than a theoretical point of view, as fragmentation we consider only such block reorder which requires additional I/O operation (disk accesses) when using described above prefetch/cache algorithm in comparison to the number of the I/Os needed in the case of perfect sequential data placement.

Issue	Observable impact on read bandwidth	Impact with increasing number of backups
Inter-version fragmentation	from -10% to -50%	increasing
Internal stream fragmentation	from +50% to -70%	stable
Global fragmentation	from 0% to -10%	stable*

Table 3.2: Impact of different kinds of fragmentation on the final restore after some number of backups based on my experiments (depending on the data).

* note that global fragmentation will most probably slowly increase, but rather than the increasing number of backups it will be with the total amount of unique data stored in the system

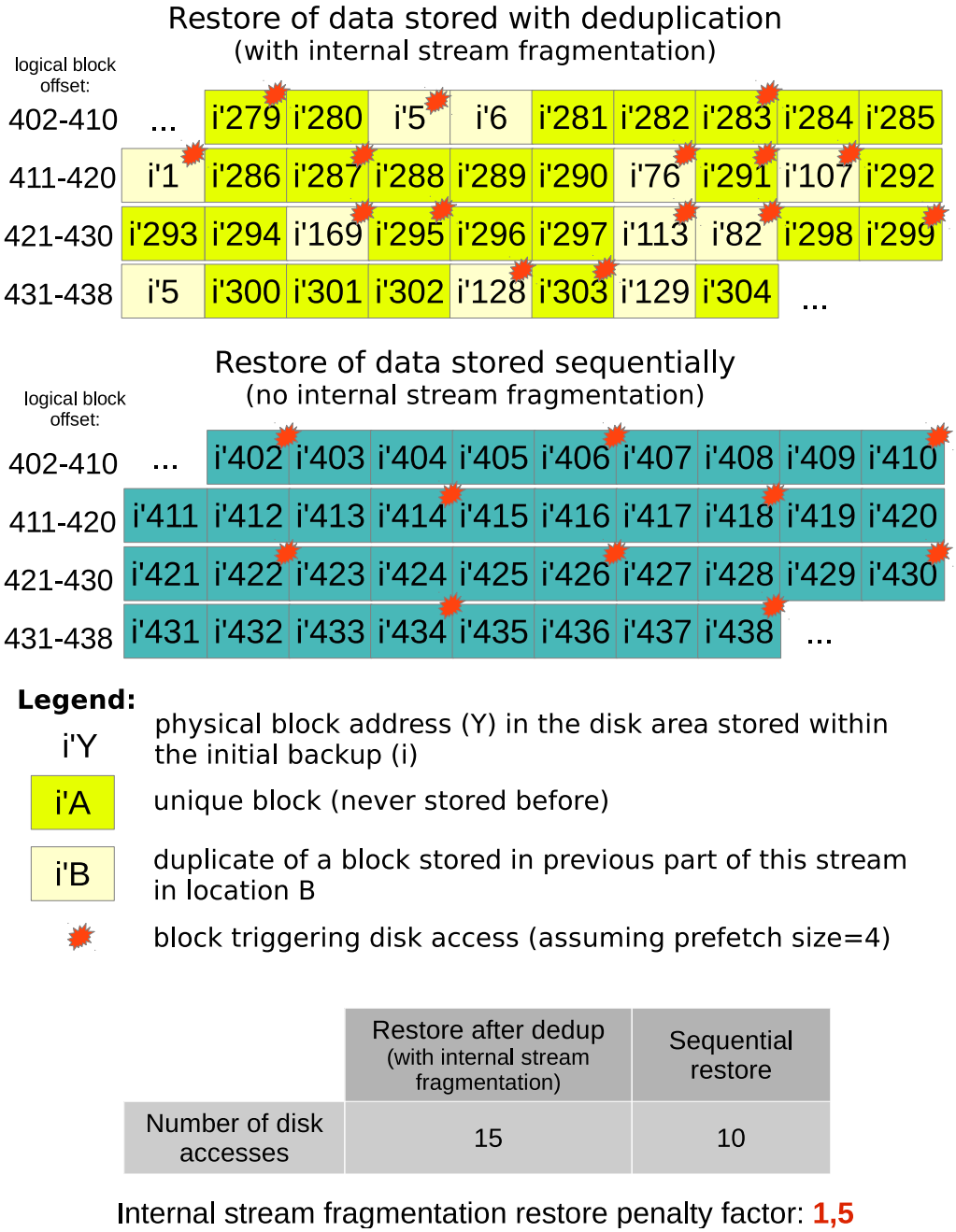
Backup systems with duplicate elimination differ much from those without such feature within the usage of storage space for the data. From the external point of view each backup process may still be seen as sequential but when it comes to the data which are deduplicated, only some will eventually get to the hard drive. Unfortunately, such write pattern highly increases the inefficiency problems in prefetch/cache algorithm described in Section 3.1.2 causing fragmentation. The concept of deduplication from its design will always eventually enforce storage of two blocks as neighbors on the disk which are in fact placed many MBs from each other in the actual logical stream, or do the opposite with the two logically sequential blocks. Such data placement required in order to save storage space opens a quite new area for researchers to identify and solve the new problems which appear.

In general three kinds of fragmentation problem exist, each caused by a different aspect of data deduplication with very individual impact on the final restore bandwidth (see Table 3.2). The detailed description and analysis of each area can be found in the following sections.

3.2.1 Internal stream fragmentation

The experiments show that having only one single backup in the entire system with deduplication may already cause degradation in its restore performance compared with the system without this feature. Such phenomenon is called *internal stream fragmentation* and is caused by identical blocks appearing many times in a single stream.

Figure 3.2 shows a part of the initial backup (from logical block offset 402 to 438). In the presented sequence one can notice blocks which are stored



	Restore after dedup (with internal stream fragmentation)	Sequential restore
Number of disk accesses	15	10

Internal stream fragmentation restore penalty factor: **1,5**

Figure 3.2: Impact of internal stream fragmentation. An example of restore process of blocks 402-438 from some initial backup with two different data storage algorithms.

in a different location on the disk than others (i'5, i'1, i'76 etc.) as they are duplicates stored already in the previous part of the same stream. The problem with such blocks is that in order to read them the disk drive has to move its head to a different location than current front of reading (between i'279 and i'304), which costs an extra I/O. What is more, the algorithm will usually try to read a full prefetch of the data placing it in cache. This wastes the allocated memory as, in many cases, only a small fraction of such blocks will ever be used. The whole process can be very expensive when it comes to the final restore bandwidth.

Note that blocks i'6 and i'129 do not cause the additional disk access even though they are not in the main sequence (from i'279 to i'304). This is due to the fact that those blocks will be present in the cache memory while reading thanks to previously read blocks i'5 and i'128 with no additional I/O required. What is more, one can notice two blocks named i'5 while only the first is marked as causing disk access. It simply assumes that as the block i'5 was read only 27 blocks earlier, it will be still present in the cache during the restore of its second copy.

Having looked at Figure 3.2 and assuming example prefetch size of 4 blocks and the cache of 100 blocks (quite large as it fits 25% of a stream so far), we can visualize the difference in the number of I/O required in two interesting cases. When the shown part of the stream is stored in a system without deduplication we need 10 I/O (= 10 x prefetch of size 4) to read the entire part. The reason for this is the sequential read of 37 blocks (from 402 to 438) as in such system logical address are identical to physical ones. On the other hand, when using deduplication we need 7 I/Os to read the continuous data from i'279 to i'304 (26 blocks) and 8 additional I/Os to read the duplicate data (see Figure 3.2). When comparing both results the difference between described scenarios is at the level of 50% (10 vs 15 I/Os) which means half the time more for the system with deduplication to restore the same backup data. Note that we have assumed a moderately large cache size as otherwise we might need to reconsider adding an extra I/O to read the second i'5 block (logical offset 431), as it could have been evicted from the cache meanwhile (between reading offset 404 and 431).

Fortunately, the appearance of internal duplicate blocks can be cleverly twisted in order to decrease rather than increase the total time required for the stream restore. Let us assume the same initial backup is read from the very beginning (starting from logical offsets 1,2,3...) but with unlimited cache. In such case, after achieving block 402 (disk location: i'279) all the blocks marked as duplicates will be already present in the memory. As a result, when requesting the part presented in Figure 3.2 only 7 I/Os will be required instead of original 10 in the system without deduplication, ending up in a

restore time smaller by 30%.

In general, even though it was expected that duplicates can also appear within one stream, a rather surprising fact is the scale of such appearance and its negative impact on the restore bandwidth in the experiments. The better news is a more or less constant number of internal duplicate blocks and similar impact on every backup regardless of the time and number of backups performed before. The important fact, on the other hand, is the observation of unlimited cache size impact, which will be further analyzed and inspire the presentation of alternative cache eviction policy supported by limited forward knowledge (see Chapter 4).

3.2.2 Inter-version fragmentation

As backups are performed regularly (daily/weekly/monthly [18]) each piece of one logical stream can be found in various versions of the same data set. Every such version differs from the previous one by the amount of data modified within one backup cycle (usually very small percentage), which makes the consequent versions of the same data set very similar.

Each backup system with deduplication will discover duplicated blocks and eventually store only the ones which have changed. The most popular in-line solutions (see comparison with off-line version in Section 2.2.1) will place all the modified/new blocks together, regardless of their actual position in the logical backup stream. Unfortunately, after tens or hundreds of backups such data placement leads to the latest backup being scattered all over the storage space.

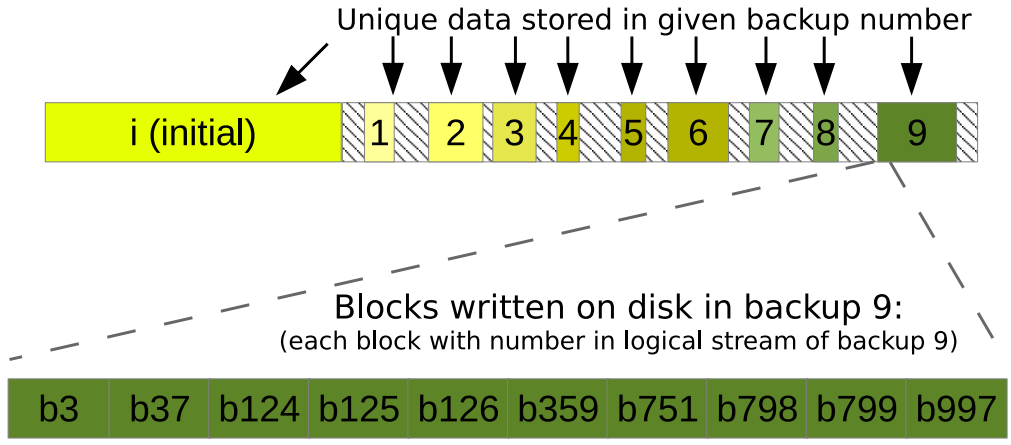


Figure 3.3: Location of data stored in each backup version with in-line deduplication (inter-version fragmentation)

Figure 3.3 shows ten versions of a sample backup set stored in a system with in-line deduplication. Each version is stored in one continuous segment on the disk, but as the initial one stores all it's data, the versions from 1 to 9 add only data which were not present in previous backups (all the duplicate blocks are eliminated and not stored on the disk). As a result, blocks belonging to the logical backup 9 can be found on disk in each of the sections initial and 1 to 9.

The restore process of the first 38 blocks of backup 9 is visualized in Figure 3.4. Assuming a prefetch size of 4 blocks and even unlimited cache memory, reading all the blocks in the shown example requires 21 I/Os (see marked blocks), while in the system where all the data are placed always sequentially only 10 I/Os (38 divided by prefetch size) are enough. Finally, an over doubled restore time is the actual cost of fragmentation in the described scenario.

The fragmentation achieved in such way is called *inter-version fragmentation*. The distinctive fact here is that, such kind of fragmentation is not present when one starts using the system, and increases during the following backups with a rate very unique for each data set and usage pattern. As the process is rather invisible during the common backup cycle, it will usually appear when the restore is necessary, which may uncover the problem of a few times lower restore bandwidth than expected. Such discovery may have very expensive consequences in case the restore was an urgent issue.

As regards inter-version fragmentation, two facts seem to clearly visualize the core of the problem. The first one is the character of changes which is slow and increasing with the number of backups, while the other is the knowledge about the typical age of recovered data (see Figure 3.1) described in Section 3.1. Given the most recent backup is the most likely to be restored, the issue seams to be very serious, but on the other hand, gathered information give an interesting insight when trying to solve the problem.

3.2.3 Global fragmentation

Global fragmentation is actually very similar to the internal one. The only but significant difference is that problematic duplicates do not come from the earlier part of the same stream but from a completely unrelated one. This is due to the fact that with internal fragmentation the problem was caused by the second and further block appearances in the stream, which allowed us to fight with its negative consequences by keeping the already restored blocks in the long enough cache. In case of the global fragmentation the issue appears with already the first block appearance (further ones should rather be qualified as internal fragmentation) and as the block is outside of the

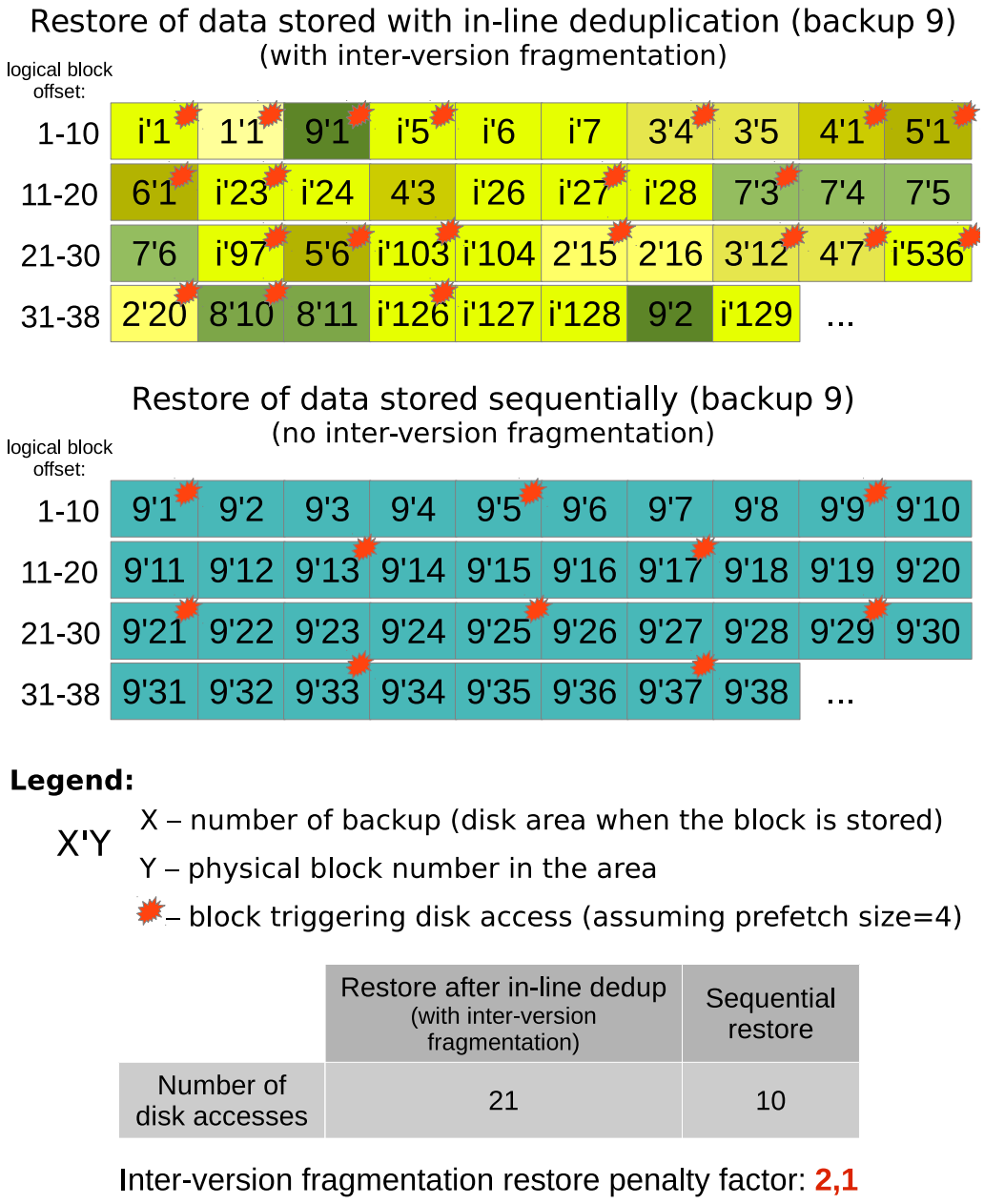


Figure 3.4: Impact of inter-version fragmentation. An example of restore process of blocks 1-38 from the 9th backup of some data set with two different data storage algorithms.

Data set	Global duplicates (blocks present in other data sets)	Impact on restore bandwidth
DevelopmentProject	1.40%	-6.99%
GeneralFileServer	0.01%	-0.06%
IssueRepository	0.55%	-8.91%
Mail-30daily	0.05%	-0.94%
Wiki	1.47%	-5.10%

Table 3.3: Impact of global fragmentation on each data set (*UserDirectories* and *Mail* data set was gathered using different hashing function therefore they could not be included in the experiment; instead of *Mail* other, similar data set was taken gathered using the same hashing function)

current backup set, it can be found in just about any location within the whole system.

I have performed a simple experiment on five independent data sets in order to verify the amount of global duplicate blocks and the impact of global fragmentation on restore performance. For each data set the first backup was chosen as a data set representative. The backup system was prepared by storing all representatives but the one tested which was loaded as the last one. By comparing the number of duplicate blocks and the bandwidth with the scenario when such backup is stored as the only one in the system, we can visualize the scale of the problem.

The results in Table 3.3 show actually a very small amount of global duplicate blocks present in other independent data sets (between 0.01% and 1.47%). Even though the outcome suggests a relatively small impact on the restore bandwidth (between 0.06% and 8.91%), the actual numbers can differ and will most probably slowly increase with the number of independent data sets and the total size of unique data stored in the system.

What can be surely done to eliminate global fragmentation is to backup together (in one stream) all the data which can potentially have common blocks such as mail/home backups of different employees or virtual machine system partition images. Unfortunately, such approach makes sense only until there exists probability of restoring those data together as otherwise it does not help. The goal of the described modification is to transfer the global fragmentation into the internal one which is much easier to deal with.

On the other hand, as the test result (Table 3.3) suggests, independent data sets share only a very small amount of data causing sometimes considerable amount of fragmentation (see *IssueRepository*). In order to prevent such scenario one could decide not to deduplicate against other data sets but only against previous version of the current one. Such approach will eliminate the global fragmentation at the cost of usually small additional blocks stored.

The global fragmentation is definitely the most problematic and complex one, both to analyze and to solve, when assuming that no duplicate blocks are allowed to be stored. Eliminating duplicate blocks to any of the current system data makes our backup dependent in some way on another completely unrelated one or possibly more. Even though some globally optimal position for each common block exists, its calculation is usually complicated and even if found, at least some of the involved backups will anyway suffer from fragmentation. What is more, the impact of such factor actually cannot be verified, as each of given traces will behave differently based on the other data present in the system.

The described complications, usually a small amount of global duplicate blocks in the backup stream and rather constant impact on the restore performance (with constant number of data sets), result in much higher priority of the other problems: inter-version and internal stream fragmentation. Taking that into account and the character of global fragmentation, rather hard or even impossible to verify, I have decided not to analyze this problem further in this work.

3.2.4 Scalability issues

The whole new perspective has to be taken into account when large deduplication backup systems are to be examined. Having tens or hundreds of servers together with even thousands of hard disks all the issues tend to reach another level. On one hand, there is more hardware to handle requests and mask the potential problems, but on the other, the scalability objectives require scaling the system capabilities together with its size.

Usually when restoring backup stream from a large system many disks are involved in the process. Because of the erasure coding [82] or RAID usually present, even each single block is cut into smaller fragments and then placed on many hard drives. More disks means better resiliency and higher potential single stream performance but unfortunately, together with multiplication of fragmentation issues and sometimes even more expensive access to a single block. Assuming, that one continuous stream is held by 10 disks, in order to read it and preserve the close to optimal bandwidth (i.e. close to 1750MB/s instead of 175MB/s with one disk [72]) one should prefetch about 2MB of

data from each disk, ending up with total prefetch of 20MB (see similar observations in [45]). As such big prefetch has much higher chance of being ineffective, in practice most systems use much smaller buffer agreeing on suboptimal choice and limiting maximal possible performance [45]. Bigger overall prefetch means also higher probability of wasting cache memory by prefetching not needed data and higher maximal fragmentation, as a result requiring a few times bigger cache. Last but not least, in case of one disk drive the minimal size of useful data was 8KB out of 2MB prefetch (0.4%), while with a scalable solution sometimes it was even 8KB out of 20MB (0.04%), significantly increasing the cost of each random read. Note that with RAID configured with larger stripe size than deduplication block size, one block may not be cut into many fragments. Still, assuming typical stripe sizes of 4-128KB and the fact that we never read less than the prefetch size of data (2-20MB) all the drives will be used anyway, which leaves the user in a similar scenario to the erasure coded one.

In general, it is much easier to assure good bandwidth having more spindles, but with a big system one should expect much more than a decent single stream performance of a single disk drive. In case of emergency one should expect the restore process of the number of streams usually backed up every day/week, which suggests keeping the scalability of reads at the same level as writes, which are usually performed in one or a very limited number of disk locations. Regardless of that, even in the simplest scenario of restoring single stream the maximal performance with using minimal amount of power and system memory is desirable.

3.3 Problem magnitude

In order to visualize the real scale of the fragmentation problem I have performed simulations on six different data sets gathered from customers of commercial system HYDRAstor. The detailed description of all data sets and the experimental methodology can be found in Section 6.1.

Additionally, I have implemented a caching algorithm based on the Bélády's optimal algorithm called *adopted Bélády's cache* (see Section 3.1.2 for details of the Bélády's algorithm). Even though it is not optimal when moving from pages to blocks (see Section 8.2.2 for discussion on its lack of optimality in case of prefetching blocks), it states the achievable performance level very well and can be used as a benchmark of near-optimal solution.

3.3.1 Impact of different kinds of fragmentation on the latest backup

In Figure 3.5 the topmost line corresponds to restore bandwidth achievable by the latest backup with given cache memory size and *adopted Bélády's cache*. The other lines are the results of simulations with real backup system and most common LRU cache eviction policy. While the middle one shows the numbers with only latest backup present in the whole system, therefore showing the impact of internal stream fragmentation, the bottom one represents the latest backup bandwidth after all the backups from the data set are stored, though including the inter-version fragmentation as well.

The results were gathered for different cache sizes and visualized as a percentage of restore bandwidth achieved for a system without deduplication (assuming sequential data location and the cache size to fit one prefetch only). Note that with using unlimited memory the internal fragmentation does not exist (only inter-version fragmentation is visible), as in case of a read request for any duplicate block the algorithm will always receive it directly from the memory. Furthermore, the restore bandwidth with such unlimited cache can be regarded as the maximal as long as there is no inter-version fragmentation nor data reordering in the backup stream.

One can easily notice high, even above 100%, maximum bandwidth level for each data set starting from some memory level. This phenomenon is in fact the positive impact of internal stream duplicate blocks described in Section 3.2.1 (reading duplicate data which are already in the memory). Even though for some data sets such values would be possible even for realistic cache sizes (512MB and below), in practice the results show up to 70% performance degradation (see: *Mail* and *IssueRepository* charts). What is more when adding the impact of inter-version fragmentation (up to 50% degradation) the final result can reach even 81% below the optimal level (*IssueRepository*) which is 75% below the level of a system without deduplication.

In general, it is very difficult to argue about the importance of either of inter-version or internal stream fragmentation. Even though they both add up to the restore performance degradation of the latest backup, their origin and characteristics are much different. Also, the impact of each of them highly depends on the data set used for measurement. More important, the inter-version fragmentation increases with each backup, which makes the moment of measurement very significant.

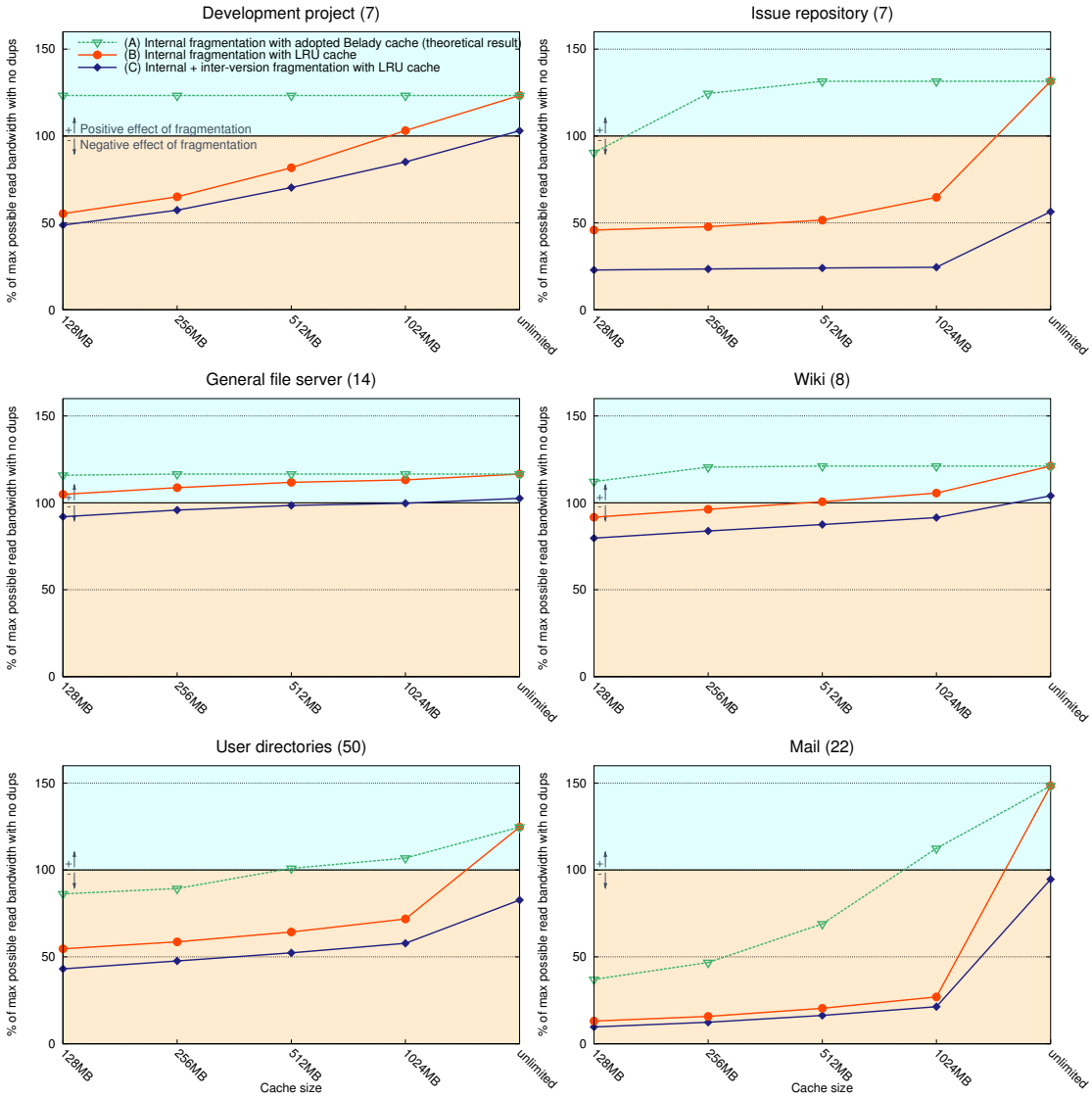


Figure 3.5: Impact of different kinds of fragmentation and cache size on the latest backup (total number of backups in each set can be found in parentheses next to the backup set name). Simulation methodology: (A),(B) - restoring the last backup when it is the only one stored [with given cache algorithm]; (C) - restoring the last backup when it is stored after all previous backups from a given data set [with standard LRU cache]

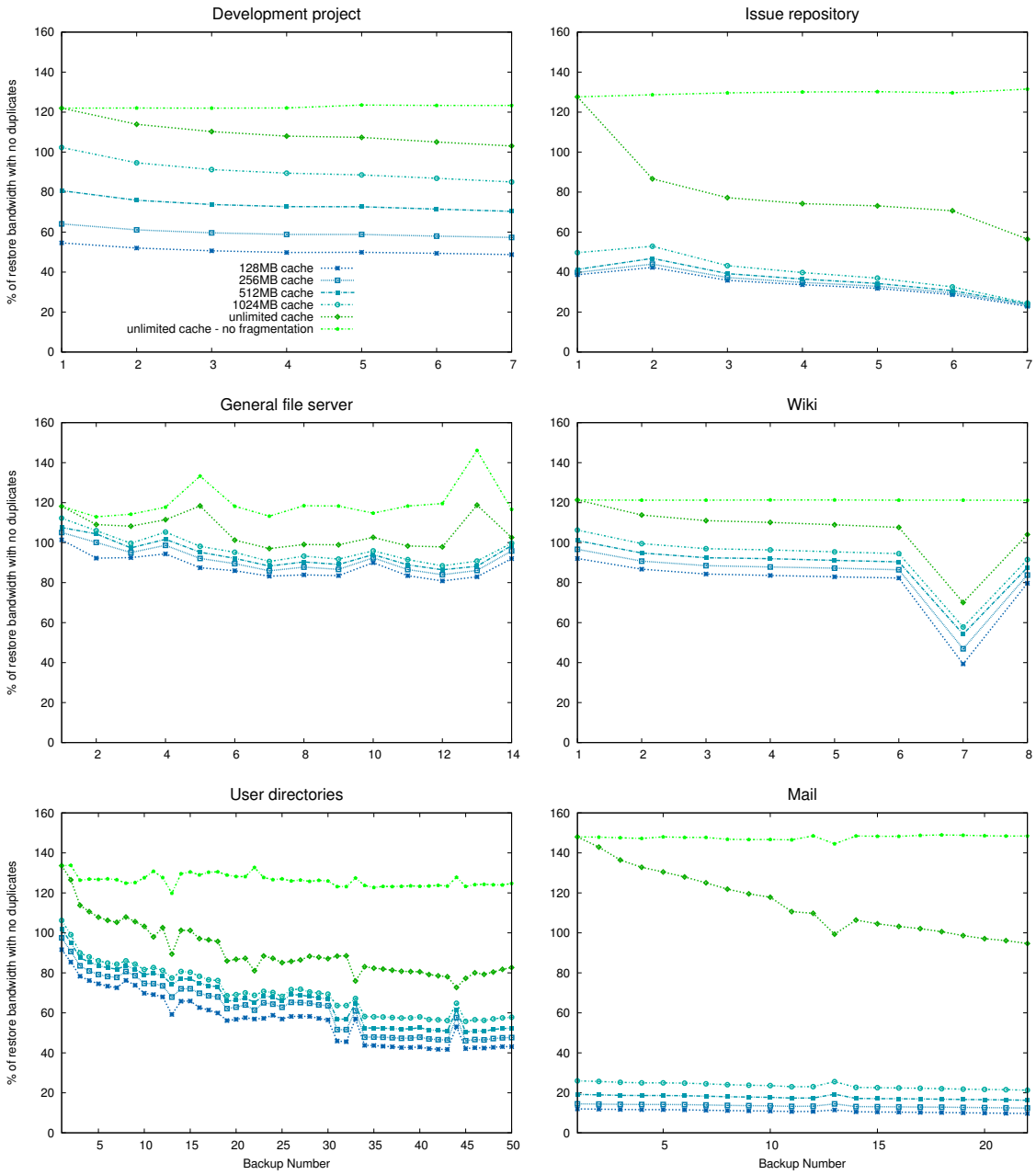


Figure 3.6: Problem of fragmentation after each backup on different data sets with defined LRU cache sizes

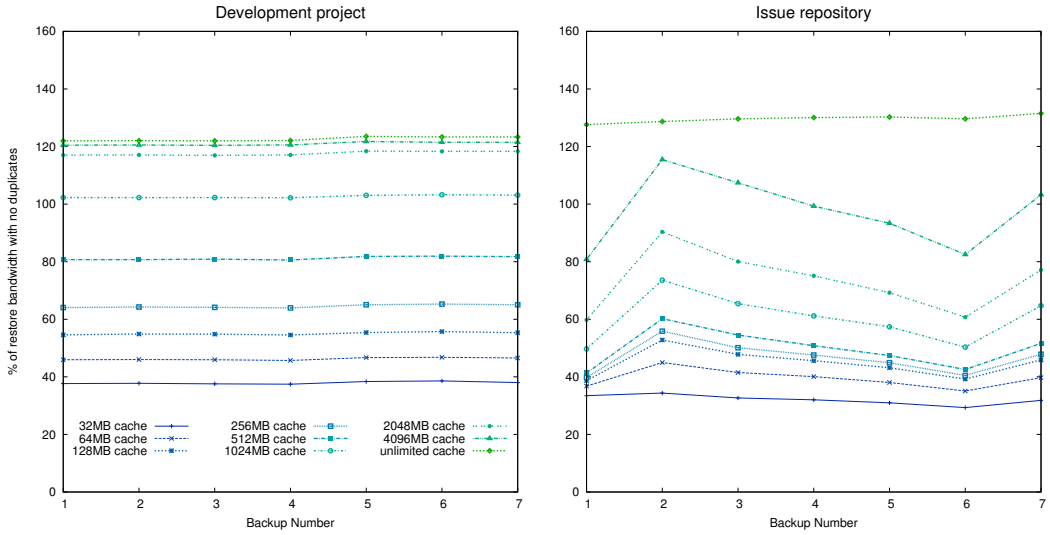


Figure 3.7: Maximal bandwidth for each backup limited by the internal stream fragmentation of each backup (no inter-version fragmentation). Results with different LRU cache sizes.

3.3.2 Fragmentation in time

The perspective of time or the actual number of backups performed, is very important when it comes to the backup systems with in-line duplicate elimination. Figure 3.6 shows the problem of fragmentation after each performed backup. The top line represents the bandwidth achievable with unlimited cache (eliminates internal stream fragmentation) and no inter-version fragmentation to show the maximal performance level achievable for each backup in each data set. All the other lines include both kinds of fragmentation.

Unfortunately, while having not more than 50 backups, it was difficult to show the impact of the problem which could be verified best after many years of regular backups. Some approximation, though, is given by Lillibridge et al. in [54] through a synthetic data set of 480 backups covering a period of 2 years and showing a drop of up to 44 times when no defragmentation was used. Even though it was generated by HP Storage Division based on the customers involving high fragmentation, it still visualizes the problem well.

As my experiments show (see Figure 3.7), the level of internal stream fragmentation is more or less stable for most backups within one set and usually stays at the level of first initial backup. Therefore, the decrease with every additional backup is in general caused by inter-version fragmentation.

As such performance drop, expressed as the percentage of the initial backup, is similar regardless of the cache size the actual scale of the problem can be easily noticed while looking at two topmost lines in Figure 3.6. Both of them with unlimited memory (which disables impact of internal stream fragmentation), but only the upper one without inter-version fragmentation included. The lines for each cache size and no inter-version fragmentation were omitted due to the clearness of charts, but the detailed impact of each factor on the latest backup is presented on Figure 3.5.

3.3.3 Cache size impact on restore time

As shown in Figures 3.7 and 3.5, the cache may be considered as the weapon used to fight internal stream fragmentation. Even though it does the work (especially when unlimited memory is available), the price is very high. For example, when starting with 32MB of cache memory with *DevelopmentProject* one need to use 16 times more memory (512MB) in order just to double the restore bandwidth and still be under the 100% line for system without duplicates. Similarly, with *IssueRepository* to achieve the same result, the memory required is even 64 times higher (2048MB). Additionally, when having in mind that modern backup systems handle many backup streams at once, the required memory would have to be multiplied again by many times, making the total system requirements enormous.

What is more, even though the increasing memory does improve the performance of usual backup, the help received is very ineffective. As the algorithm with *adopted Bélády's cache* show (the total topmost line in Figure 3.5), in most cases having only 128MB or 256MB cache memory backup should be able to allow the restore with near maximal possible bandwidth, which is from 20% (*GeneralFileServer* 256MB) up to 519% (*IssueRepository* 256MB) higher than the one achieved with conventional cache usage (LRU) and usually above the level of non duplicate system bandwidth. The only data set which differs much is *Mail*, where the internal duplicates pattern causes even the *adopted Bélády's cache* not to achieve non duplicate bandwidth levels with reasonable amounts of memory.

On the other hand, as regards the inter-version fragmentation, the additional memory does not seem to help much (Figure 3.5). The impact on increasing restore time caused by this aspect of fragmentation is similar regardless of the cache size and equal to 13-20% for the shortest sets (*Wiki*, *DevelopmentProject*, *GeneralFileServer*), 34-46% for the *Mail* and *UserDirectories* and up to even 91-171% for the most fragmented *IssueRepository* after only 7 backups.

Simulation results of using different cache sizes within one data set show

only moderate impact of the memory size on the actually achieved bandwidth but also indicate the reason of such observation. While the inter-version fragmentation problem does seem to be more or less memory-independent, the second issue connected with internal fragmentation is simply caused by the poor memory effectiveness reached by the common Least Recently Used cache eviction policy. As the experiments with *adopted Bélády's cache* show (see Figure 3.5), the potential solution of this problem may offer higher restore bandwidth with using even 8 times smaller amount of memory (in all data sets having 128MB with *adopted Bélády's cache* is better than 1024MB with LRU).

3.4 Options to reduce the negative impact of fragmentation during restore

Fragmentation is a natural by-product (or rather waste) of deduplication. It is possible to completely eliminate fragmentation by keeping each backup in a separate continuous disk space with no interference between backups, however, in such case there will be no deduplication.

Another approach to practically eliminate impact of fragmentation on restore performance is to use a big expected block size for deduplication. In such case, when fragmentation happens, it will not degrade restore speed much, because the seek time is dominated by the time needed to read block data. For example, with 16MB expected blocks size, read disk bandwidth of 175 MB/s and 12.67 ms read access time [72], a seek on each block read will increase the restore time by less than 14%. However, optimal block size for deduplication varies between 4KB and 16KB depending on particular data pattern and storage system characterization (we need to include block metadata in computing the effectiveness of dedup [70]). With much larger blocks, the dedup becomes quite ineffective, so using such big blocks is not a viable option [43, 55, 70].

An interesting solution would be to use reduced deduplication in order to fight fragmentation. In this approach whenever some currently written block is far away on the disk during backup, it can be simply stored on the disk regardless of the existing copy. Unfortunately, as one of the solutions show [45], this path leads to lower duplication ratio especially when moving towards reasonable restore results. An interesting trade-off would be to fight global fragmentation this way (as it is usually caused by a small number of duplicates) but use other techniques, which would save the full deduplication, to solve inter-version and internal stream fragmentation.

Given backup systems usually consist of many servers and disks, they can also be used in order to speed up the restore. If the performance from one drive is at the level of 25% of the one achieved by system with no deduplication one can use four (or more) disks in order to reach the desired level (together with prefetch and cache multiplication and all the consequences). The only modification necessary would be to divide the single stream between the chosen number of drives, which is often the case anyway (e.g. RAID). Although this proposal means rather masking than solving the problem, it will work whenever sufficient number of not fully utilized devices are available.

Finally, there is one more potentially good solution for the problem of inter-version fragmentation called off-line deduplication (see Section 2.2.1 for details). In this approach as the latest backup is always stored as single sequential stream, the restore performance is optimal (assuming no internal duplicates). Unfortunately, the number of problems with this deduplication concept results in a very small percentage of such solutions present on the market.

The options presented above, although possible and sometimes even very easy to introduce, require either fair amount of additional resource or propose trade-offs which are not easily acceptable (i.e. restore bandwidth at the cost of deduplication effectiveness). On the other hand, just by looking at the details of the backup and restore process one can find a number of interesting characteristics. Using them in a dedicated way may actually solve the problem with only minimal cost and surprisingly reach restore bandwidth levels not achievable before, sometimes even higher than those provided by backup systems with no deduplication.

Chapter 4

Cache with limited forward knowledge to reduce impact of internal fragmentation

As it was stated in the previous section, one of the main reasons for a usually low restore bandwidth in systems with duplicate elimination is internal stream fragmentation. When analyzing the test results for each cache size (see Figure 3.5), one can notice much higher restore bandwidth achieved with the *adopted Bélády's cache* when compared with the common solution with LRU cache, even when only single backup is present in the backup system (without inter-version fragmentation). Even though the results differ much depending on the data set, the average increase for all cache sizes is above 80%, while for example with 256MB cache size the values differ from 7% and 25% for *GeneralFileServer* and *Wiki*, up to 160% and 197% for *IssueRepository* and *Mail*.

The actual problem visualized in the above results is inefficient usage of cache memory. Because of the poor quality of prediction delivered by LRU very often the block is evicted from cache before it is actually used (or reused), while at the same time many blocks not needed at all occupy memory. This is true especially in backup systems with deduplication where many copies of the same block appear quite often in a single stream (see Table 6.1 for more details).

In this chapter I would like to present an algorithm of cache eviction policy with limited forward knowledge, whose purpose is to alleviate the consequences of internal stream fragmentation by keeping only the blocks which will be referenced in the near future. The side effect of the proposed solution is also a more effective cache usage in general, which provides benefits also when used for streams with inter-version fragmentation.

4.1 Desired properties of the final solution

In order to successfully replace LRU as a cache replacement policy the new solution should:

- provide the restore bandwidth close to the one achieved with *adopted Bélády's cache* (and of course significantly higher than LRU),
- not require additional data to be stored (maximal deduplication effectiveness should be kept),
- enforce only small if any modifications in restore algorithm,
- not require any changes outside of the restore algorithm,
- not require much additional resources such as disk bandwidth, spindles and processing power,
- offer a range of choices in addressing trade-offs if necessary.

4.2 The idea

Each data set before being stored in a backup system is usually compacted into one large (tens or hundreds of GBs [81]) logical stream by a backup application. Many read [45] and deduplication [85] algorithms already rely on such backup policy and tend to optimize the path of streaming access, which is in fact very common in backup systems. In my idea I would like to further optimize this well known property during the restore process.

As the problem of internal stream fragmentation seems to appear quite often, any forward knowledge can be very useful in order to keep in memory only those blocks which would reappear in the nearest future. The idea itself is present in the Bélády's algorithm [6], but the major issue making it useless in general is that such information is difficult or even impossible to get. Luckily, in a backup system this characteristic is different as backups are generally very big and accessed in the same order as they were written. When starting a restore one can usually find out the whole restore recipe, which means having access to actually infinite knowledge about blocks being requested in the future.

Even though the idea of using all forward addresses is tempting, in reality it is not necessary as they would occupy precious memory which could otherwise be used for the actual cache (to keep the data). My experiments showed that having limited forward knowledge (fixed amount of information

about future access to blocks) is enough in order to deliver good restore performance. Moreover, the achieved results are often very close to the ones of the *adopted Bélády's cache* which has infinite forward knowledge.

The high level scheme of the algorithm presented in this chapter is shown in Figure 4.1.

4.3 System support

To implement the limited forward knowledge cache I assume a backup system supporting the following abilities:

- *one address for all blocks with identical content:* Blocks with the same content should also have the same address. In case of backup systems this property is assured by content addressability [67];
- *whole backup restore with single stream identifier:* Single identifier delivered to the system should be sufficient in order to read the entire backup. Thanks to that the system gains access to the metadata for blocks restored in the future and is able to deliver forward knowledge;
- *ability to prefetch metadata in advance:* The system should be able to read defined amount of metadata first before retrieving the actual data. Such metadata will be required for the cache eviction policy to assure better memory usage.

Most systems with deduplication already support content addressability [23, 85] and provide mechanism for reading the whole stream, given for example the file path as identifier. Also every restore requires the metadata, which are gradually read from dedicated place in the system (usually separate from the data) in order to receive the full recipe and the addresses of the actual data blocks. Small reorganization in order to read more of such metadata before beginning of a stream restore can be easily introduced. As a result, the algorithm described in the next section can be seen as generic and adoptable to a wide range of systems with deduplication.

4.4 Algorithm details

4.4.1 The system restore algorithm

Looking from the system level, a restore of a stream starts by receiving the stream identifier (see Figure 4.2). Even though such operation unlocks the

```
1  class Cache function insertBlocks(blocksWithData, forwardKnowledgeOracle)
2  {
3    for (blockID, blockData) in blocksWithData
4    {
5      if (forwardKnowledgeOracle.contains(blockID))
6      {
7        this->insertBlockWithPriority(
8          forwardKnowledgeOracle.getPriority(blockID), blockID, blockData
9          )
10     }
11   }
12   this->removeBlocksWithLowestPriorityExceedingMemoryLimit()
13 }
14
15 function readStream(streamID)
16 {
17   currentOffset = 0
18   endOffset = FORWARD_KNOWLEDGE_SIZE_CONST
19
20   //initialize forward knowledge
21   blockIDs = readBlockIdsForStream(StreamID, currentOffset, endOffset)
22   forwardKnowledgeOracle.addForwardKnowledge(blockIDs)
23
24   while notFinishedReading(streamID)
25   {
26     blockID = getNextBlockID(streamID)
27     if (not cache.containsBlock(blockID))
28     {
29       blocksWithData = disk.readBlockWithPrefetch(blockID)
30       cache.insertBlocks(blocksWithData, forwardKnowledgeOracle)
31     }
32
33     blockData = cache.getBlockData(blockID)
34     blockSize = blockData.size()
35     returnToUserAsynch(blockData)
36
37     //update forward knowledge
38     forwardKnowledgeOracle.updatePriorityForReadBlock(blockID)
39     currentOffset = endOffset
40     endOffset = endOffset + blockSize
41     blockIDs = readBlockIDsForStream(StreamID, currentOffset, endOffset)
42     forwardKnowledgeOracle.addForwardKnowledge(blockIDs)
43   }
44 }
```

Figure 4.1: Reading a stream from a backup system with limited forward knowledge - scheme

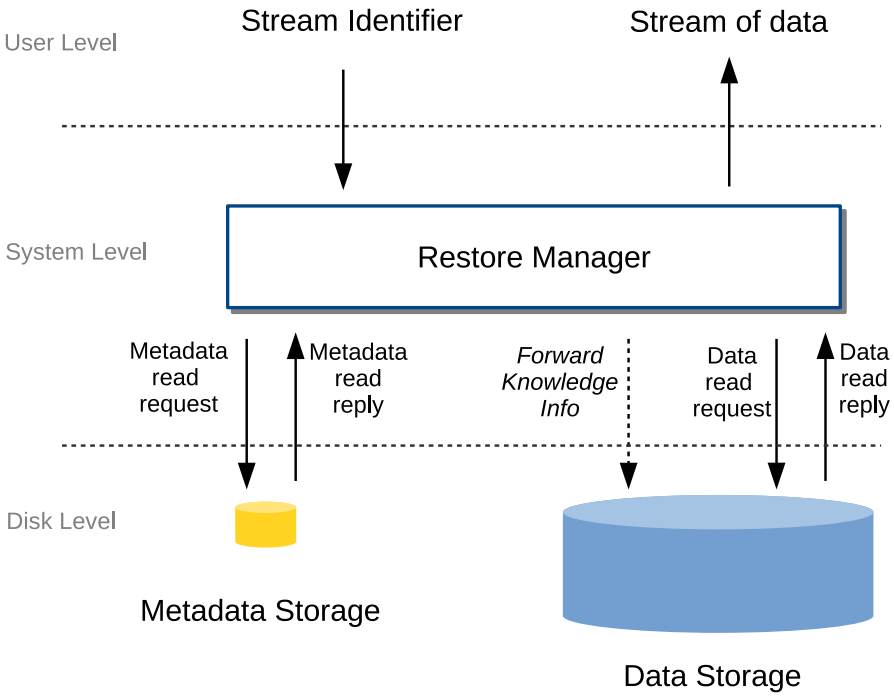


Figure 4.2: The data restore process - scheme.

access to all metadata required, usually only some small amount is read in order not to occupy too much memory. Based on this, the requests to read blocks with dedicated addresses are sent. When the restore proceeds, the additional metadata is read and more requests are issued. The whole process is very smooth in order to provide constant load fully utilizing the system and its resources.

The basic idea of the proposed solution is to use the information which is already in the system. As having the knowledge about future blocks to be read can be very useful for caching policy, the algorithm should be able to read some reasonable amount of such forward knowledge.

4.4.2 The disk restore process

At the disk level, when it comes to the data storage, the standard prefetch and cache algorithm is used (see Section 3.1.2), but with modified cache eviction policy (see Figure 4.3). Thanks to the forward information received, a dedicated oracle with limited forward knowledge can be created. Its information about next block occurrence helps with assuring close to optimal memory allocation in cache.

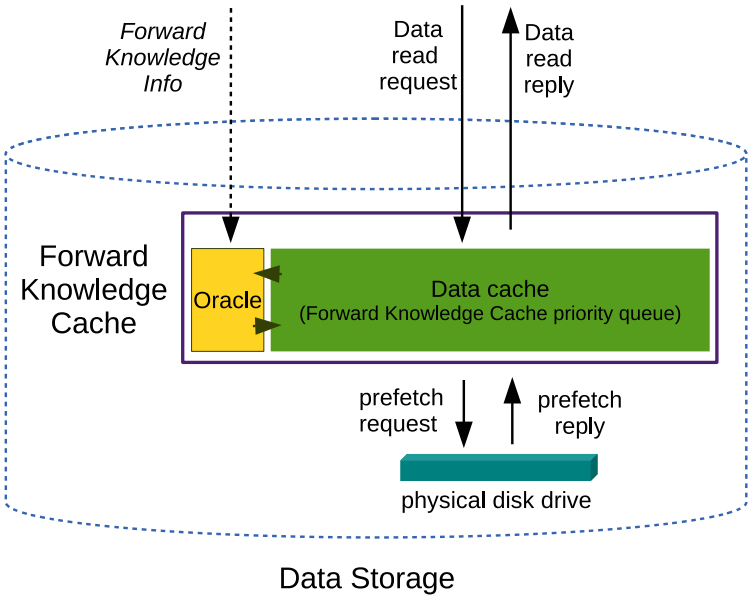


Figure 4.3: The forward knowledge cache - scheme.

Whenever the name **cache** is used in this thesis it always refers to the memory where the actual data is kept, common for all caching algorithms (*data cache* area on the above figure). As a result, it does not cover additional memory required by specific solutions. *LRU cache*, *Forward Knowledge cache* and other similar names are used to refer to the entire algorithm utilizing corresponding cache eviction policy.

The oracle with limited forward knowledge

The oracle is designed as a map keeping the identifiers of all known forward but unread blocks together with sorted list of block positions in which they appear in a stream (see Figure 4.4). The update with forward information will add an identifier of a block if not present and push the proper block position at the back of its list. When necessary, the structure may return for a given block the closest future position in which it will be required, or update the most recently read block by removing its closest (current) position from the list of next block occurrences. With the additional data, the oracle with limited forward knowledge requires dedicated memory different from the one where the cache data are kept. For each block address from the total amount of forward knowledge, both the block identifier and its position in the stream are required. Fortunately, the size of both can be limited to use only fraction

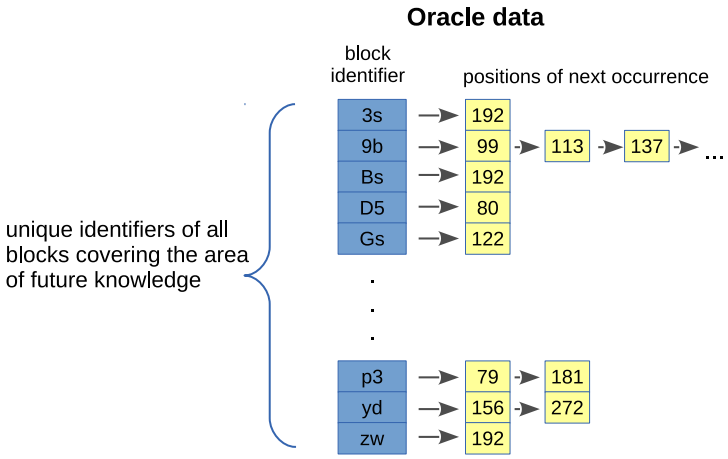


Figure 4.4: The design of oracle with limited forward knowledge.

of memory that is required for the actual cache.

Each block in a system can be identified by its address. In deduplication backup systems such address is usually based on a block content and is calculated using a hash function such as 160-bit SHA1. Such original address (hash) size is designed to assure an extremely low probability of collision in order not to assume two different blocks as identical ones (see Section 2.2.4 for details). Fortunately, in the case of the oracle structure such information does not need to be that precise. First, even when some hash collision appears, the only thing which happens is keeping in memory a single block, which will in fact not be needed in the future (and will be easily detected and removed when the expected occurrence does not happen). Second, with limited forward knowledge the algorithm limits also the subset of the whole storage system in order to find the collision (i.e. to a few GBs). In order to set an example there is a 1 to 10 million chance for a hash collision within about 2 million of blocks (=16GB of data, assuming 8KB block size) while having 64bit (8 bytes) hash function and assuming its uniform distribution ($\frac{N-1}{N} \cdot \frac{N-2}{N} \cdot ... \cdot \frac{N-(k-1)}{N}$; where $N = 2^{64}, k = 2 \cdot 10^6$). This leads to the conclusion, that 64bit identifier is good enough for the purpose of providing required functionality.

The exact information about block location in the stream is also not required in the algorithm. As its only purpose is to roughly compare the blocks position on the quite large part of a stream (i.e. a few GBs), it is enough to divide the stream into *sections* and keep only this reduced information in order to save the memory. Such a section may cover for exam-

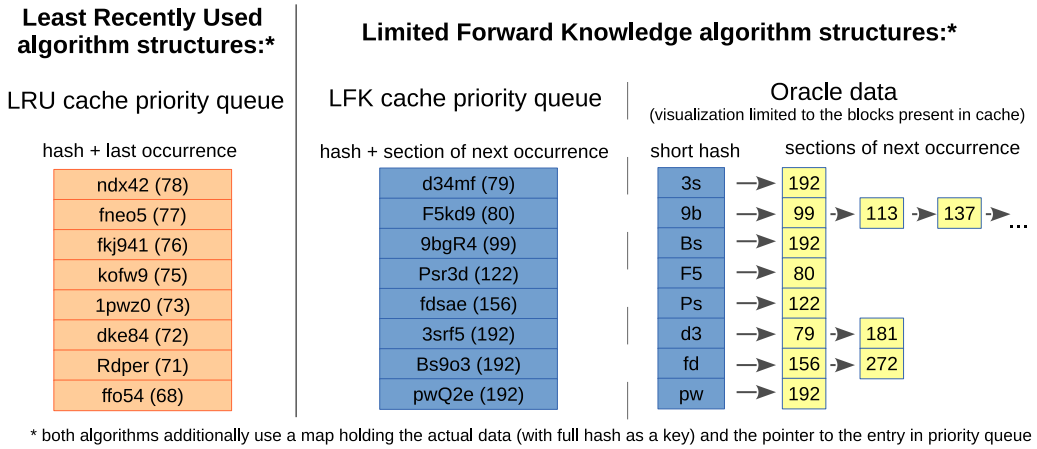


Figure 4.5: Comparison of structures used by LRU and Forward Knowledge algorithms

ple 8MB (arbitrary number) and be identified by its number sequentially from the beginning of a stream. As it would be desired to keep the section identifier limited (i.e. 2 bytes) in case all numbers are used, renumbering operation can be performed to subtract the number of current section from all the numbers stored in the oracle. In our example such operation, even though cheap as performed in memory, will be executed once every $8MB \cdot 64K(2bytes) - 8GB(of forwardknowledge) = 504GB$ of data restored in a single stream (which in reality can happen in only a few % of cases according to backup workload analysis of over 10000 of systems by Wallace et al. [81]).

Cached blocks locations

The forward knowledge cache is in general organized as a standard map with block addresses as keys and the data as values. The only difference from LRU is the kind of information kept (see Figure 4.5). Instead of the list storing least recently used blocks (LRU priority queue) the ordered list of blocks with the closest occurrence is kept - FK Cache priority queue (with ability to binary search in case a block with new location is added). All the operations, such as updating or adding blocks, are very similar to the operations on LRU structures, beside the fact that instead of the latest usage the next block occurrence information is stored.

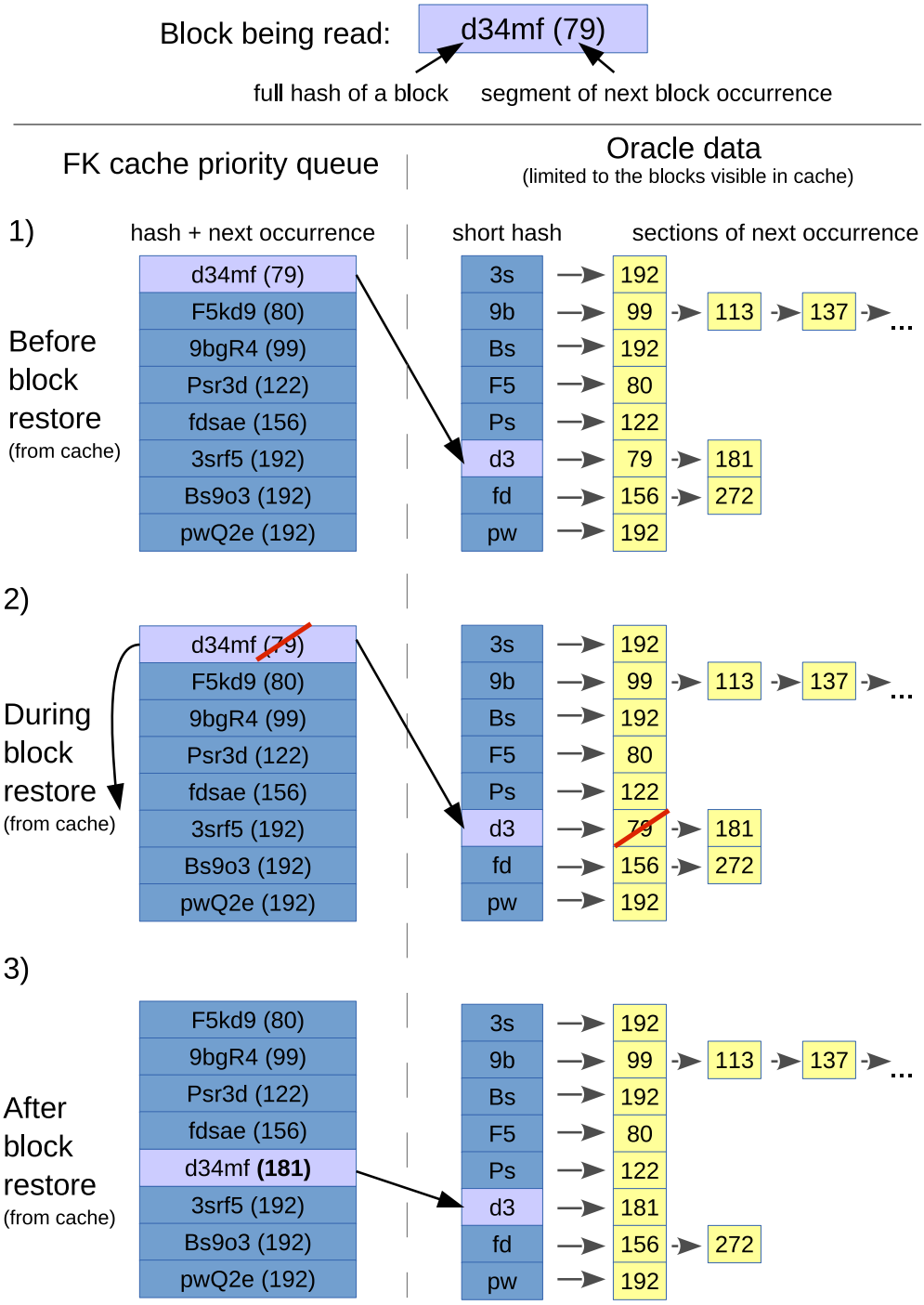


Figure 4.6: Reading a block already present in cache

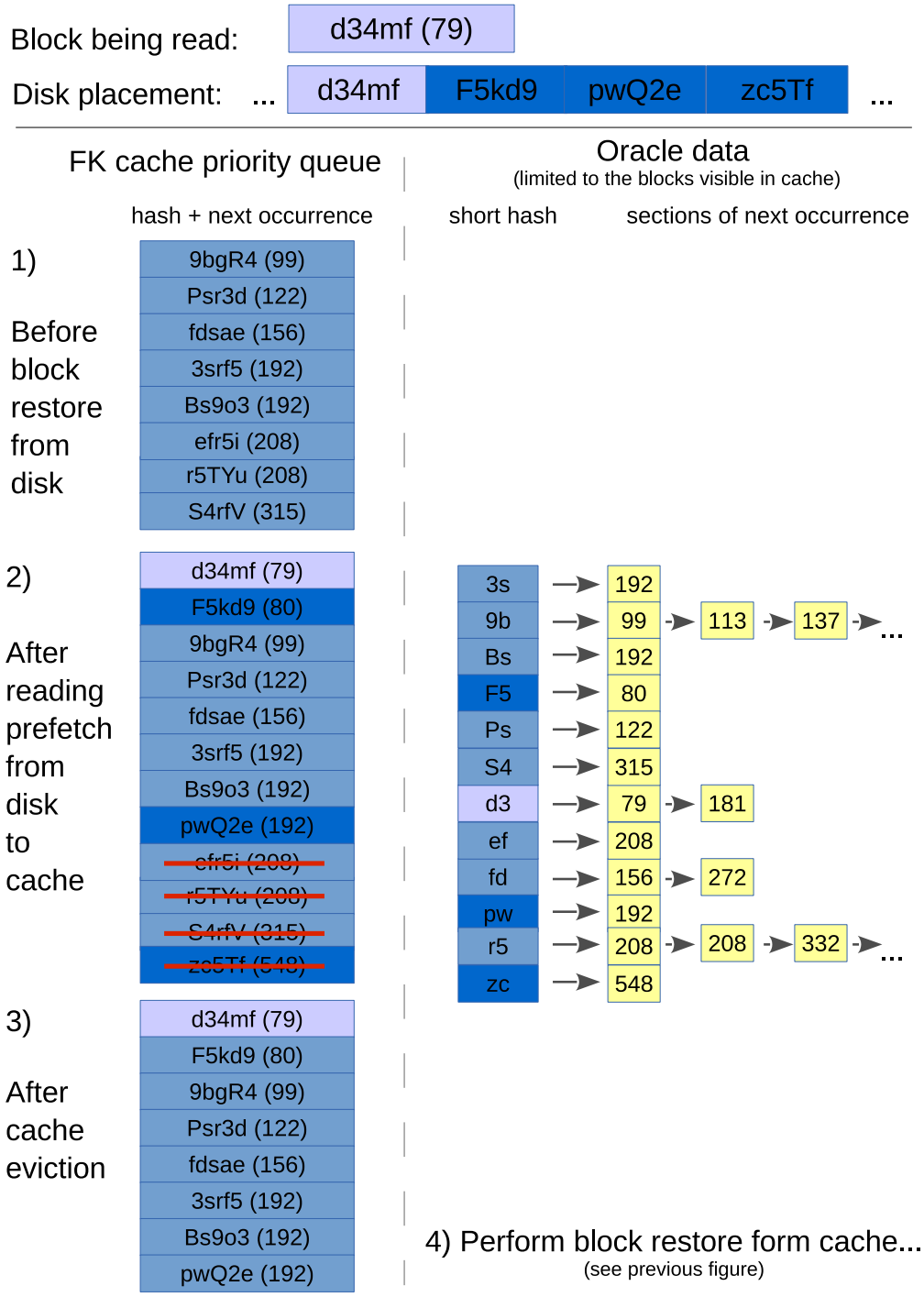


Figure 4.7: Reading a block not present in cache (with eviction policy)

The eviction policy

Figures 4.6 and 4.7 show the example of block restore and cache eviction policy in two cases. The first one, when the block was found in cache, and the second, when it had to be restored from disk. In the former case the only operation performed is the actual update of the restored block in both cache and oracle structures. The latter one is more complicated and includes also the cache eviction process. In general, it consists of the following steps:

- read the block from the disk to cache together with its prefetch (updating cache with the information about next section occurrence provided by the oracle)
- update the cache priority queue keeping blocks ordered by the section of the next occurrence with restored blocks
- remove the blocks exceeding the maximal cache size with the most time to the next occurrence (highest section numbers)
- continue updating the structures in the same way it is done when the restore is performed from cache (4.6)

In a case when there is no known section of next occurrence in the oracle for the most recently prefetched blocks and there is still some space left in the cache memory, a few choices can be made. We can keep some of such blocks in the cache (for example by assigning an artificial and large section number and use an LRU or other algorithm for evicting them) or free the memory to use it for some other purpose in case dynamic memory allocation to different structures is possible. As my experiments showed that the first option does not provide noticeable performance gain, the better choice would be to use the additional memory for other system operations if necessary (such as restores, writes and background calculations) or to dynamically increase the oracle size, which would result in providing more forward information until all the available memory is efficiently used.

The algorithm presented above is very similar to the *adopted Bélády's cache*. Actually it behaves identically until the cache memory is utilized in 100% by blocks indicated by forward knowledge. Any lower utilization indicate worse behaviour than the *adopted Bélády's cache*. The reason for such scenario is always the limitation in forward knowledge size and characteristics of the individual stream (duplicate blocks outside of forward knowledge area).

4.4.3 Memory requirements

As the cache itself in the algorithm with Forward Knowledge is build in a very similar way to the one with LRU algorithm, its memory requirements do not change. Separate and dedicate memory, though, will be required by the oracle with its forward knowledge. Another requirement may be an additional list of all the block addresses waiting to be restored after they are received as a forward knowledge, but before they are actually used to restore the data. As the forward knowledge size may cover many gigabytes, it can take many seconds before the addresses are used to restore the data (I assume that addresses are delivered in order to fill the forward knowledge while the restore is performed as fast as possible), which means they require dedicated memory. The alternative approach described in detail in Section 4.4.4 may be not to occupy the additional memory but restore the metadata twice: once for the sake of forward knowledge and the second time for the restore itself. Whichever solution is used, the proper memory size should be included in the total memory assigned for the restore cache.

The detailed amount of additional memory required can be calculated as follows. Each entry in the oracle equals at most one short hash (8 bytes) and one section number entry (2 bytes). To be detailed we need to include the structure overhead as well. As standard map requires keeping pointers which are expensive (8 bytes per pointer while we keep only 10 bytes per entry), the hash table with closed hashing is a much better choice here, possibly at the cost of in-memory access time. Still, for acceptable results in this case the memory allocated should be at least 30% higher [38] than requested, which ends up with about 13 bytes per entry. Together with the full address in the waiting list of 20 bytes size (160 bits, assuming SHA-1) the total of 33 bytes is the cost of having one block (8KB) forward knowledge, which further means 4.12MB per every 1GB of data. For best results, a few GBs of forward knowledge is desirable (in detail it depends on each exact data set characteristics).

4.4.4 Discussion

Alternative solution

The important observation is that keeping only the list of addresses for the data to be read in the future consumes already two thirds of the additional memory required (20 out of 33 bytes kept per block). An idea worth consideration in order to minimize this impact is presented below.

The easiest way in order not to keep the additional memory allocated is to read the addresses again from the system. In such case, there would be two

stream accesses to metadata: one to provide proper information for oracle and the other asking for the concrete block addresses to be restored. Given the size of a block address is 20 bytes per 8KB block, the whole operation would require reading 0.256% more data than with the original solution, leaving only a small requirement of about 1.62MB of memory per each 1GB of forward knowledge (instead of 4.12MB).

This solution sounds very tempting, especially in cases when only small amount of memory is available. The exact choice would definitely require the detailed analysis of a given system and other possible consequences of both approaches.

The impact of different metadata read order

As the pattern of metadata restore is to be significantly modified with the proposed algorithm, the question of its impact on restore performance appears. The discussion on this subject is difficult in general, as it requires the detailed knowledge of a given system and its metadata restore process. Fortunately, as the metadata is usually only small portion of all data to be restored (0.256% with 20 bytes for each 8KB), even reading it all again should not generate much additional work. Also, when the systems with high metadata overhead of over 100 bytes per block [70] are taken into account, the total restore bandwidth degradation in the same scenario would still be lower than 1.5%, which should be hardly noticeable.

4.5 Trade-offs

The major trade-off in the area of intelligent cache with forward knowledge is with the size of memory dedicated for the standard cache and the forward knowledge. Depending on the data set characteristics and the total amount of memory available, only very small amount of forward knowledge can already assure the effective cache usage in some cases, whereas in others very big forward information even at the cost of a much smaller cache size is a much better choice.

The best solution to this problem would be not to state any hard division between the cache and the oracle. Thanks to such approach, the system would be able to extend the future knowledge in case the cache memory is not fully utilized or decrease it otherwise. Even though the described scenario is tempting, it is much more complicated and requires detailed testing, especially in the real storage system case where distributed communication may be an issue. Those concerns made me offer the version with hard division,

keeping the details of this solution for the future work.

One more trade-off is connected with the section size. Since in order to save the memory the exact location of next block occurrence is not kept, some evictions can be made not in the order desired. Such scenario can happen when many blocks are located in a single section and one is to be evicted. Fortunately, such event does not impact performance much as the reordering can happen only within the blocks with the longest time to next occurrence (the least important ones). Also, the achieved performance can never be lower than the one in the same scenario but with the cache memory reduced by the size of a single segment. In the typical scenario of 256MB cache and 8MB section size, the performance would never be worse than with 248MB cache and the exact knowledge about each block position.

Chapter 5

Content Based Rewriting algorithm to reduce impact of inter-version fragmentation

The experiments presented in Section 3.3.2 show the negative impact of inter-version fragmentation caused by in-line duplicate elimination. Even though the values in some cases do not seem very significant (restore bandwidth decrease of about 12%-17% after 7-14 backups in case of *GeneralFileServer*, *Wiki* and *DevelopmentProject*) the fact of relatively small number of backups in those data sets and visible tendency of the problem to increase in time supported by the experiments with longer data sets (about 19%-36% decrease after 22-50 backups in case of *Mail* and *UserDirectories*) suggest potentially high impact in real world usage, where the number of backups created for one data set varies from 50 to over 300 every year. Moreover, the *IssueRepository* data set states that there exist cases where performing only 7 backups may already cost over 50% of potential restore bandwidth. My observations were confirmed on other, independent data sets by Nam et al. [53, 54] and longer ones (over 90 backups) by Lillibridge et al. [45].

In this chapter I would like to propose the Context Based Rewriting algorithm (CBR) dealing with the issue of inter-version fragmentation by changing the location of blocks to reflect the current streaming access pattern, and as a result, provide more effective prefetch and cache usage.

5.1 Desired properties of the final solution

The problem requires a solution without negative effects on other important metrics of deduplication system. Such solution should:

- eliminate the reduction in restore bandwidth caused by inter-version fragmentation for the latest backups;
- introduce no more than very little (preferably below 5%) write performance drop for ongoing backups;
- not degrade deduplication effectiveness (if necessary use only little and temporary additional space);
- not require much additional resources such as disk bandwidth, spindles and processing power;
- offer a range of choices in addressing trade-offs.

5.2 The idea

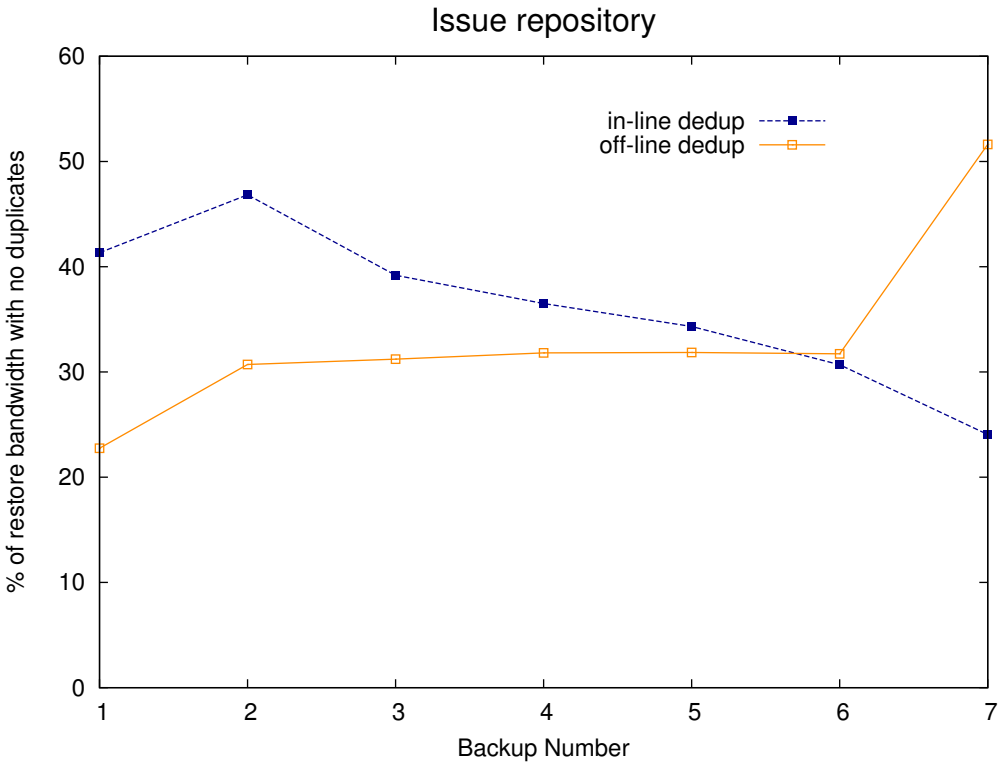


Figure 5.1: Comparison of backup fragmentation: in-line vs. off-line dedup (restore speed of each backup after all 7 *IssueRepository* backups with 512MB cache and LRU cache eviction policy)

In most cases, the latest backup is restored after failure, because users are usually interested in having the most up to date information restored (see Figure 3.1). Based on this observation, I would like to eliminate fragmentation for the latest backups at the expense of the older ones (in order to preserve deduplication ratio). An example of such approach is given in Figure 5.1. It presents the drop in restore performance caused by fragmentation across backup versions as a function of version number in two cases: (1) in-line dedup with fragmentation decreasing with the backup age; and (2) off-line dedup, which results in the latest backup written continuously to disk and fragmentation increasing with the backup age. By introducing the Context Based Rewriting algorithm I would like to add a defragmentation capability to the in-line deduplication feature in order to achieve the defragmentation effect similar to the off-line dedup, but without the associated costs.

As it was already presented in Chapter 3.2, in a system with in-line deduplication the already existing blocks are not written again, making the backup process very fast. Unfortunately, such approach may lead to high fragmentation as the two neighbour blocks in the stream can end up being far from each other in the system. In order to prevent such scenario the CBR algorithm analyzes the blocks from incoming backup stream and their physical localizations in the system. In order to minimize the performance drop caused by inter-version fragmentation, the algorithm will move some of duplicate blocks to a new physical location to preserve good streaming access and make the prefetching effective. As the algorithm is performed during backup of the stream, the actual blocks to be moved are not read (which might cost a lot) but a copy delivered in the stream is written. The old copies are removed by the deletion process run periodically. In opposite to off-line deduplication only a small percentage of blocks is moved (rewritten) – the ones assuring the highest restore bandwidth gain. The scheme of the CBR algorithm during writing of a backup stream is presented in Figure 5.2.

Even though both cache with limited knowledge and CBR algorithm fight fragmentation, they present a completely different approach and aim at different kind of the issue. The first one does not modify the data in the system and allows effective cache memory usage during the restore by using the future knowledge available. Such approach allows caching duplicate blocks present internally in the stream, causing internal stream fragmentation. The latter algorithm presented in this chapter is completely different and does not deal with blocks reappearing in the stream. It's main goal is to make all the blocks structured in a more sequential way during backup and to fight the so-called inter-version fragmentation. Interesting fact, though, is that such approach results in a more effective prefetch leading to more accurate data loaded into cache, which links both solutions. The actual impact of each of

them separately and combined are further analyzed in my experiments.

5.3 System support

To implement our defragmentation solution described in the next section, the backup storage should support the following features:

- *content addressability [67]:* This is a base feature useful for the subsequent features described below;

- *deduplication query based on checking for block hash existence:* It is crucial that this query is hash-based and requires reading metadata only. For presented defragmentation solution it is not important if deduplication is attempted against all blocks in the system, or the subset of them (such as with sparse indexing [46]). However, it is required to avoid reading entire block data to perform dedup, because such operation would result in fact in reading the fragmented stream and a very low total write performance. Also, it has to be acknowledged, that with high fragmentation one may not have enough spindles even to read block metadata fast enough. However, there exist solutions to this problem based on flash memory [21, 50], whereas SSDs are too small and too expensive to hold entire backup data;

- *fast determination of disk-adjacent blocks:* Given two blocks, system should be able to determine quickly if they are close to each other on disk. This can be achieved when each query which confirms block existence in the system returns location of this block on disk;

- *ability to write a block already existing in the system and remove the old one.* This is needed when a decision is made to store a block again in order to increase future read performance. Such rewrite effectively invalidates the previous copy, as the new one will be used on restore.

Many systems with in-line deduplication such as DataDomain [85] and HYDRAstor [23] support the above features; for other systems such features or their equivalents can be added. As a result, the algorithm described in the next section can be seen as generic and adoptable to a wide range of systems with in-line deduplication.

```
1  class StreamContext function addBlock(block)
2  {
3    if (isDuplicate(block.getID()))
4    {
5      blockDiskLocation = getBlockLocationOnDisk(block.getID())
6    }
7    else
8    {
9      blockDiskLocation = NONE
10   }
11   this->addBlockWithLocation(block, blockDiskLocation)
12 }
13
14 function writeBackupStream(stream)
15 {
16   //streamContext is a small, continuous part of a stream of defined size
17   while notFull(streamContext) and not stream.isEmpty()
18   {
19     streamContext.addBlock(stream.extractNextBlock())
20   }
21
22   while not stream.isEmpty()
23   {
24     decisionBlock = streamContext.extractFirstBlock()
25
26     if not isDuplicate(decisionBlock.getID())
27     {
28       blockAddress = storeBlockOnDisk(block)
29     }
30     else
31     {
32       blockDiskLocation = getBlockLocationOnDisk(block.getID())
33       if  streamContext.numBlocksLocatedNear(blockDiskLocation) < THRESHOLD
34       {
35         blockAddress = storeBlockOnDisk(block)
36         scheduleToRemoveOldCopyInBackground(blockDiskLocation)
37       }
38       else
39       {
40         blockAddress = diskToBlockAddress(blockDiskLocation)
41       }
42     }
43     streamContext.addBlock(stream.extractNextBlock())
44
45     returnToUserAsynch(blockAddress)
46   }
47
48   flushStreamContextLeftoversToDisk(streamContext)
49 }
```

Figure 5.2: Writing a backup stream with CBR algorithm - scheme

5.4 Algorithm details

5.4.1 Block contexts

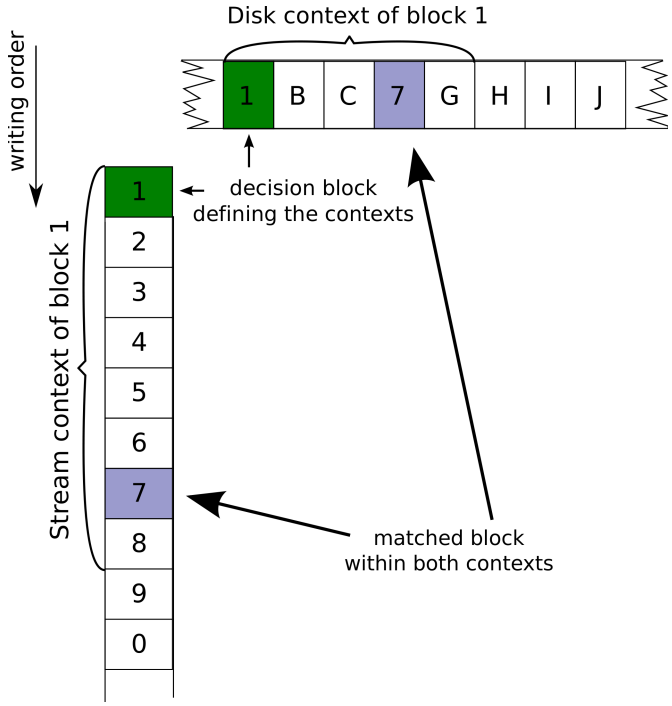


Figure 5.3: Disk and stream contexts of a block

The algorithm utilizes two fixed-size contexts of a duplicate – its *disk context* and *stream context*. The stream context of a block in a stream is defined as a set of blocks written in this stream immediately after this block, whereas its disk context contains blocks immediately following this block on disk (see Figure 5.3). When the intersection of these two contexts is substantial, reading of blocks in this intersection is very fast due to prefetching. In fact, this is quite often the case especially for an initial backup.

The problem of fragmentation appears when the disk context contains little data from the stream context. This occurs because of deduplication when the same block is logically present in multiple stream locations with different neighbours in each one of them. Even though such effect is also caused by internal duplicate blocks (internal stream fragmentation), it is practically eliminated by the cache with limited forward knowledge proposed in the previous chapter. The algorithm presented below lets us deal only with the blocks, which appear for the first time in the current backup stream.

Note that the disk context size is strictly connected with the restore algorithm and equals the prefetch size. The stream context size, on the other hand, cannot be lower than this value in order not to limit the maximal intersection. Based on the experiments, the usual sizes of disk and stream context where by default set to 2MB and 5MB respectively. The impact of other values will be described in section 5.5.

5.4.2 Keeping the contexts similar

The basic idea is to rewrite highly-fragmented duplicates, i.e. blocks whose stream context in the current backup is significantly different from their disk context. The attempt with such rewriting is to make both contexts similar in terms of common blocks percentage. After rewriting, the new copy of the block will be used for reading, which means also prefetching other blocks stored in the same backup (therefore reducing fragmentation), and the old copy is eventually reclaimed in the background.

The goal is to rewrite only a small fraction of blocks, because each rewrite slows down backup and consumes additional space until the old copy of the block is reclaimed. By default this parameter, called *rewrite limit*, is set to 5% of blocks seen so far in the current backup.

The algorithm iterates in a loop over the backup stream being written deciding for each encountered duplicate if it should be rewritten. The current duplicated block to be processed by the algorithm is called *the decision block*.

Since the data to be written is not not known in advance by the storage system, the decisions whether to rewrite duplicates are made on-line (without future knowledge, except for the stream context). Taking the above into account, the algorithm can always make a sub-optimal choice for a given duplicate: for example by deciding to rewrite it, although such rewrite "credit" may be better saved for another duplicate later in the stream; or by deciding not to rewrite a duplicate with a hope that a better candidate may appear later in the stream; but such candidate may in fact never materialize. Therefore, the challenge in the algorithm is to make good rewrite decisions.

5.4.3 Reaching rewrite decisions

In order to guide the rewriting process, we need to introduce a notion of rewrite utility of a duplicate. Also, two thresholds will be maintained and adjusted on each loop iteration: the minimal rewrite utility (constant), and the current utility threshold (variable).

Rewrite utility

If the common part of disk and stream contexts of a decision block is small, rewriting of such block is desired, as it can help to avoid one additional disk seek to read little useful data. In the other extreme, if this common part is large, such rewriting does not save much, as the additional seek time is amortized by the time needed to read a lot of useful data.

Therefore, for each duplicate in a backup stream, the *rewrite utility* is defined as the size of the blocks in the disk context of this duplicate which do not belong to its stream context, relative to the total size of this disk context. For example, the rewrite utility of 70% means, that exactly 30% of data in blocks in the disk context appear also as the same blocks in the stream context.

Minimal rewrite utility

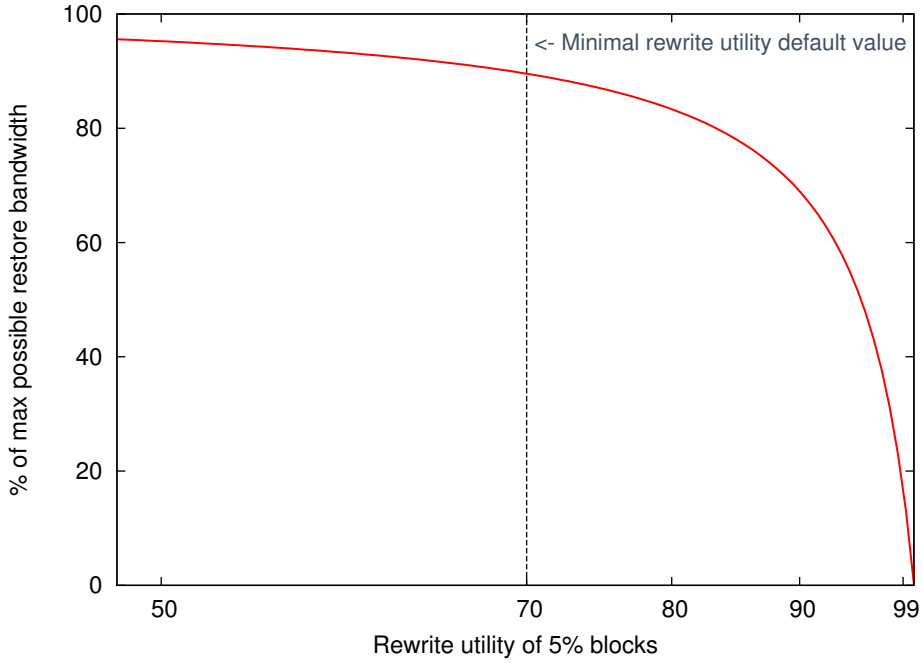


Figure 5.4: Read bandwidth decrease with having only 5% of blocks with a given rewrite utility (assuming other blocks are not fragmented).

The minimal utility is a constant parameter of the CBR algorithm in order to avoid rewriting which would improve restore performance only marginally. I have set the minimal rewrite utility to 70%. This value may look high, but lower minimal utility is not much useful as presented in the below analysis.

Let us assume a simple case of a backup with 5% of fragmented duplicate blocks, all with rewrite utility equal to the minimal rewrite utility. The remaining 95% of blocks are not fragmented (rewrite utility equal to 0%). Moreover, assume that a prefetch of each fragmented block does not fetch any useful data beyond blocks needed to satisfy the rewrite utility of this fragmented block. Such scenario assures the minimal possible gain with the maximal possible effort. In such case, rewriting all of the fragmented duplicates potentially improves restore performance by about 12% (see Figure 5.4), which is in my opinion sufficient to justify the rewriting. If the minimal utility was set to 50%, rewriting all fragmented duplicates in a similar backup would offer only 5% improvement, which simply seems not enough.

Note that there may be backups which suffer significantly from fragmentation, but for which all duplicates have rewrite utility just below the minimal utility. However, to reduce restore bandwidth drop caused by fragmentation for such backups, the algorithm would need to rewrite many more blocks than just 5%. For example, when having all the blocks with rewrite utility 70% rewriting 5% of blocks assures not more than 2.15% better performance. Fortunately, I have not encountered any such case in my experiments.

Current utility threshold

The current utility threshold is a variable parameter of the CBR algorithm defining the rewrite utility for current *decision block*. In order to calculate its value a *best-5%* set of blocks is defined as 5% (the default value) of all duplicates seen so far in the backup stream with the highest rewrite utility. Note that each rewritable block must be a duplicate, so in some cases fewer than 5% of all blocks may be kept, because there may be not enough duplicates in the stream.

To establish *best-5%*, the utility of rewriting each duplicate seen so far is calculated without taking into account actual actions taken by the algorithm. In each loop of the algorithm, the current rewrite utility threshold is set to the utility of rewriting the worst of the *best-5%* blocks. Such selection roughly means that if this value had been used as the current utility threshold for every decision block from the beginning of the backup up to the current point, and without a limit on the number of rewritten blocks, the algorithm would have rewritten all the *best-5%* blocks.

Initially, the current rewrite utility threshold is set to the minimal utility and is kept at this level for 500 blocks in order to allow defragmentation of the first blocks in the stream. As this part consists of only 4MB of data (usually out of many GBs), the 5% rewrite limit is not observed here.

Rewrite decision

The decision block is rewritten when its rewrite utility is not below the maximum of the current rewrite utility threshold and the minimal utility. Otherwise, all blocks in the context intersection are not rewritten, i.e. they are treated as duplicates in the current stream and marked to be skipped by future loops of the algorithm. Note that always each rewrite decision is subject to the 5% rewrite limit, which is computed on-line based on all blocks in the stream seen so far.

The decision is asymmetric: rewrite only the decision block or mark all blocks in the intersection as duplicates. That is, even if the decision block is to be rewritten, there is no decision to rewrite (or not) other blocks in the intersection, as they may have their context intersections big enough to avoid rewriting. However, once the verdict to keep the decision block as a duplicate is taken, all the remaining blocks in the intersection should also be kept as duplicates, to ensure that the read of the decision block will fetch also these additional blocks (i.e. the rewrite utility of the decision block remains low).

Block rewriting does not always guarantee that the size of the intersection of stream and disk contexts will be increased. For example, the stream context may contain duplicates only and the algorithm may decide to rewrite just one of them, because remaining are sequential. In such case, the size of the intersection is not increased. However, the rewritten block will still end up on disk close to other new or rewritten blocks. When such blocks are prefetched, they will most likely survive in read cache, reducing number I/Os needed for restore, so rewriting can be still beneficial.

5.4.4 Implementation details

Computing the context intersection

The stream context of the decision block is filled by delaying the completion of this block write until enough write requests are submitted for this stream. For each request, the duplicate status is resolved by issuing a modified dedup query (with extra result of block location on disk) based on secure hash of the data (i.e. SHA-1) [24, 25]. If there already is a query result filled in by one of the previous loops of the algorithm, such query is not issued. In case a duplicate is detected, the modified query returns the location of the block on disk and the block address (the hash) is returned without further delay. While filling in the stream context, each given block is examined by comparing distance on the disk to the decision block and qualified as duplicate appearing already in the disk context (or not). In such way, the intersection of the disk context and the stream context is determined.

Adjusting rewrite utility threshold

Since tracking utilities of *best-5%* is impractical, the algorithm keeps a fixed number of utility buckets (for example 10000). Each bucket is assigned disjoint equal sub-range of rewrite utilities, all buckets cover the entire utility range, and each bucket keeps the number of blocks seen so far with its utility in this bucket range. Such structure allows, with minimal cost, approximation of the rewrite utility of the worst of the *best-5%* blocks with reasonable accuracy – within the range of utility assigned to each bucket. Actually, only the buckets representing the values above the minimal rewrite utility are useful, but in both cases the memory required for such structure is negligible.

Filtering internal stream duplicates

My experiments show that actually every backup stream contains blocks which are duplicates of some others from the stream (internal stream duplicates). Since without decreasing the deduplication ratio, there is no on-line way to determine the optimal location of such internal duplicate, any disk location in a neighborhood of the corresponding duplicate block from a stream can be considered as a potentially good one. The important thing, though, is that during the backup of each version of the stream, the same logical location is chosen by the CBR algorithm for the purpose of rewriting and no other location triggers such operation. This is required in order not to rewrite the internal duplicate blocks from one place in the logical stream to another during each backup (thrashing). On the other hand, the cache with forward knowledge described in the previous section suggests that the first location in the logical stream should be considered as the one having the highest potential. Once read into cache, it can potentially stay there for long time serving also other requests to the same data. Therefore, the block should be considered for rewriting only when it occurs in the stream for the first time.

As the knowledge about being an internal duplicate does not need to be exact and the size of each backup can be known with some approximation before it is written, we can use a bloom filter [10] in order to use relatively little memory. Before being qualified for the stream context, each block should be verified in the filter for existence. If found, it should be written to the system by the standard mechanism (it can be a false positive). Otherwise, proper bits in the filter should be set indicating the block existence and the block should be qualified for the stream context and for the CBR algorithm. Note that the bits are never set to zero and the whole bloom filter is deleted when the backup is completed.

In such case, for each 1GB of expected data, we require about 240KB of memory in order not to exceed 0.1% false positive ratio (15 bits per key, 128·1024 keys, 7 hash functions) for the last bytes of the stream. Such number is acceptable, as with having at maximum 5% of blocks to be rewritten, usually below 4 (roughly estimating) of them will become falsely assumed as internal duplicates. As the 1GB of data require at least 500 I/O, the negative impact on the restore bandwidth will usually be much smaller than 1%.

Usually, the process of hashing does require additional processing power, but this case is different. Since in the considered systems, we already have the hash of the whole block calculated (160 or 256 bits), we can simply use some chosen bits of this hash as a good hashing function for the bloom filter. Such optimization make the final requirement on the additional processor cycles is negligible.

Read simulation during write

The presented CBR algorithm performs well in assuring more sequential disk access by rewriting a small number of blocks. In the end, though, what counts is the restore performance achieved, when reading a stream. Keeping this result at the same level, along with further decreasing number of rewritten blocks, would help to lower the cost paid during each backup.

In order to achieve that, a restore simulation during backup is performed with standard LRU cache eviction policy. Instead of the block hashes, block location identifiers are kept in the memory. Thanks to that we can simulate reading of blocks which are not yet stored to the system. The structure requires the LRU queue and the map to check whether the incoming block location is already in the cache, which should take no more than 384KB of memory with simulation of 128MB cache (3 x 8bytes x 128MB / 8KB), which delivered very similar results for all cache memory sizes in most data sets. After introducing this enhancement, the number of rewritten blocks became lower by about 20%-30% while keeping similar restore bandwidth.

Simulating the algorithm of cache with forward knowledge instead of LRU during backup, would most probably bring even better results in decreasing the number of rewritten blocks, but is more complicated (requires additional memory and delay) and will be considered for the future work.

Background operations

The CBR algorithm requires a background process removing the old copies of the rewritten blocks. This can be done together with other maintenance tasks already run from time to time, such as deletion, data scrubbing and

data sorting [23]. Until this removal is performed, the additional space used by rewrites temporarily reduces the deduplication ratio. As the percentage of such data is limited and the maintenance tasks are usually performed quite often, such solution should be easily acceptable.

Modifications to read operation

If data blocks are content-addressable, both old and new copies have the same address, so pointers to data blocks do not have to be changed when the old copy is replaced with the new one. To ensure good performance of the latest backup restore, only the procedure to access the latest copy of a block may need slight modifications if the system previously did not allow many copies of the same block. This can be done by keeping only the entry to the newest block in the block index.

5.4.5 Memory requirements

The part of the algorithm, which potentially requires significant amount of the memory, is the bloom filter used for the elimination of internal duplicate blocks, as described in Section 5.4.4. The memory required is about 240KB for each GB of the backup stream, which does not seem much, but bigger streams put larger pressure on this requirement.

Since the usual amount of memory used during stream backup is at the level of tens of GBs, the proposed solution is acceptable for stream sizes up to 100GB (24MB of memory) or even 1TB (240MB of memory) - depending on the system and the exact memory available. Note that according to data gathered from over 10000 backup systems by Wallace et al. [81], streams larger than 500GB use on average less than 5% of total capacity in the backup systems, making them very rare in general.

If necessary, it is always possible to divide one large stream into the smaller ones based on its logical content, assuring more common data placed together (see Section 3.2.3). The alternative solution is also to use less precise (higher number of false positives) or compressed bloom filter, at the cost of lower number of defragmented blocks or more complex access to its data.

Finally, the described above bloom filter and the stream context of the default size 5MB are structures required per each stream being stored into the system. This means, that the final amount of memory should be multiplied by the number of streams expected.

5.4.6 Discussion

Optimizing the on-line algorithm

The CBR algorithm is clearly on-line, as it looks only at the blocks seen so far (plus small stream context which could be considered as forward knowledge). Unfortunately, for the same reason it is optimal only in the case when current utility is stable throughout the whole stream. In the other cases, with large variations especially between the rewrite utility of blocks at the beginning and the end of the stream together with full utilization of 5% rewrite limit, the final result may not be that good (even though still better than before defragmentation).

All the malicious scenarios can be addressed optimally by setting the fixed rewrite utility for the whole stream. The best value of such utility would be the one computed by the current utility algorithm and achieved at the end of the stream. Unfortunately, such information would require future analysis before storing the stream. A simple approximation could be done to use the statistics gathered during backup of previous version of the same data set.

Fortunately, in all data sets tested the above problems were at the minimal level also because the number of blocks rewritten was always below the 5% level. Therefore, even with the on-line algorithm the final results were quite close to the ones achieved with no inter-version fragmentation.

The off-line CBR

A simple modification of the CBR algorithm can be introduced, which seems to eliminate its cost and preserve the advantages: first, identify the blocks to be rewritten, and rewrite them later in the background, after backup is finished. This does not work well, however, because rewriting would require reading the fragmented blocks, which could be extremely slow (exactly because they are the most fragmented). In the in-line version of the CBR those blocks are actually received almost for free, when a user is writing the data.

5.5 Trade-offs

Stream context size

The algorithm uses by default 5MB as stream context size, because it is big enough to allow the CBR to be effective and small enough so increase in write latency, due to filling this context is acceptable. Assuming a backup system achieving 100 MB/s for a single stream [85], it will take not more than 50ms to fill in the context. Other values between 2MB and 20MB were also

verified and are acceptable to lower or increase the desired latency with only slightly different final bandwidth results, but larger variation in number of duplicates rewritten (larger stream context means less blocks to be rewritten but a bit worse restore bandwidth). On the other hand, when the delay is crucial in some system, it is possible to define the size of stream context by the maximal acceptable delay we are able to wait for the write confirmation. In such case the stream context of each block will be different but it should still provide reasonable defragmentation results.

Note that the delay from the above examples will be introduced only for non-duplicate blocks, which already have a significant latency.

Number of rewrites

Even though the default limit for the number of rewrites is set to 5% of all blocks appearing so far in the stream, this value can be easily modified in case of some individual requirements. Having a higher limit will make all the blocks with the rewrite utility above the minimal one to be rewritten and may be very useful for a stream which was backed up for a long time without CBR defragmentation. Of course, the time of such backup will proportionally increase but from the next one the limit may be brought back to 5%.

Also, decreasing the limit may be useful in cases where only minimal bandwidth drop is acceptable (e.g. below 1%). In such case the algorithm will do well in choosing the most fragmented blocks to be rewritten, providing the highest gain with the smallest associated cost.

Chapter 6

Evaluation with trace driven simulations

This chapter contains description of experimental methodology, data sets and testing scenarios. Next, it provides separate evaluation for each of the algorithms presented in Chapter 4 and 5. The combined effect of both solutions is also presented. Finally, the scalability evaluation shows the possible impact of fragmentation and the above solutions on many large systems present on the today's market.

6.1 Experimental methodology

The goal of my experiments is to show the problem in the environment common for all or at least most of the systems without any bottlenecks but the disk itself and not look into details of each particular implementation. This way, I give priority to evaluate the severity of the actual fragmentation problem and efficiency of its solution without obscuring the experiments with architectural assumptions, which usually are simply the limitations of some given implementation. In other words, the results presented in this section can be viewed as the upper bound on the performance, especially significant for the most popular systems with in-line deduplication. Note that even though the systems with off-line deduplication do not suffer from inter-version fragmentation, they still have to deal with the one present internally in the stream. For that, the cache with forward knowledge presented in Chapter 4 and evaluated here works very well.

With the additional help of my colleagues I have prepared a 12,000 line C++ simulator capable of performing parallel testing (on many cores and machines) in thousands of possible configurations. Having the actual traces

gathered from real users, the simulator produced results and statistics which lead to the conclusions and finally the numbers presented in this work. Even though this is only a fraction of the results achieved, the numbers presented are the most important ones having the highest impact on analyzing and overcoming the fragmentation problem present in backup systems with deduplication.

6.1.1 Backup system model

I propose a backup system model general enough to represent the common part of the vast majority of backup systems with in-line deduplication, simple enough to be implemented with respect especially to the main problem characteristics, and efficient enough in order to perform a large number of experiments in a limited time.

Write simulation

In the model I have assumed a simple storage subsystem that consists of one continuous space (something as a single large disk) which is empty at the beginning of each measurement. The write process in the simulator was designed to keep all the characteristics present in systems with in-line duplicate elimination described in Section 3.2 with the main ones such as locality preserving blocks placement [85] and placing new blocks after currently occupied area. Such write algorithm assures maximal write bandwidth and minimal resource utilization, which were always the priorities while performing a backup.

The data used for the simulations was gathered from real users. In order to optimize the process, each version of a given data set was chunked using Rabin fingerprinting [12, 68] into blocks of the average size 8KB (as the most popular in today's backup systems). After such process the traces with short hash only (64 bit) and size of each block were stored and used for all the simulation. Thanks to that it was not necessary to perform chunking nor hashing each time the experiment was performed, and it was possible to keep the whole disk image in the system memory, which made the testing process very efficient.

Restore simulation

As described in the Section 3.1.2, reading with using prefetching and caching is the most commonly used within the storage environment.

In all experiments fixed-size prefetch is used, so we can assume that the read bandwidth is inversely proportional to the number of data I/O oper-

ations during restore. Although certainly there are systems for which performance is influenced by other factors, I believe that correlating achieved bandwidth with the number of fixed-size I/Os allows us to focus on the core of the fragmentation problem and disregard secondary factors such as network and CPU speeds.

I assumed constant prefetch size of 2MB as the most efficient with today's disk drives even with most fragmented data (see next section for justification). The cache size varies between 128MB up to 1GB per single stream being restored for better problem visualization, while the experiments with unlimited size of cache provide important information about maximal theoretical limitations. The common LRU data replacement policy, as the most popular one [42, 45, 54], is used in order to show current performance level.

Note that in the experiments with forward knowledge only the blocks with known future appearance are kept in cache. If the forward knowledge is short or there is only a small number of blocks which are to be used in the future, the cache memory may not be fully utilized. Such approach is not optimal but I have decided to use it in order to clearly visualize the limitations. Also, my experiments showed that keeping memory fully utilized in a way similar to LRU helps only a little or does not help at all, especially when having larger forward knowledge. Based on the results, it is clear that any additional memory should be used in the first place to extend the forward knowledge, which suggests dynamic memory division between the oracle and the cache when it comes to specific implementation.

The choice of disk context/prefetch size

Prefetching is very effective for reading data placed sequentially on disk. In order to show this in the environment with deduplication, I have performed a simulation with fixed prefetch size modified from 512KB up to 8MB for all six data sets (see Figure 6.1). Since the comparison here is done with using different prefetch sizes, extrapolation of performance based on the number of I/Os only cannot be done any more (comparison result in such case depends on how much data a disk can read in a seek time). Therefore, I have used common enterprise data center capacity HDD specification [72] to be able to reason about achieved performance.

As we can see on the charts, in 4 out of 6 cases for both fragmented and not fragmented data the shortest restore time is achieved with prefetch size equal 2MB. The only exceptions are *Wiki* and *GeneralFileServer*, for which 8MB prefetch is slightly better. Based on those results, I have decided to use 2MB prefetch for majority of the tests, as the most representative one for both fragmented and not fragmented data with common LRU cache. The

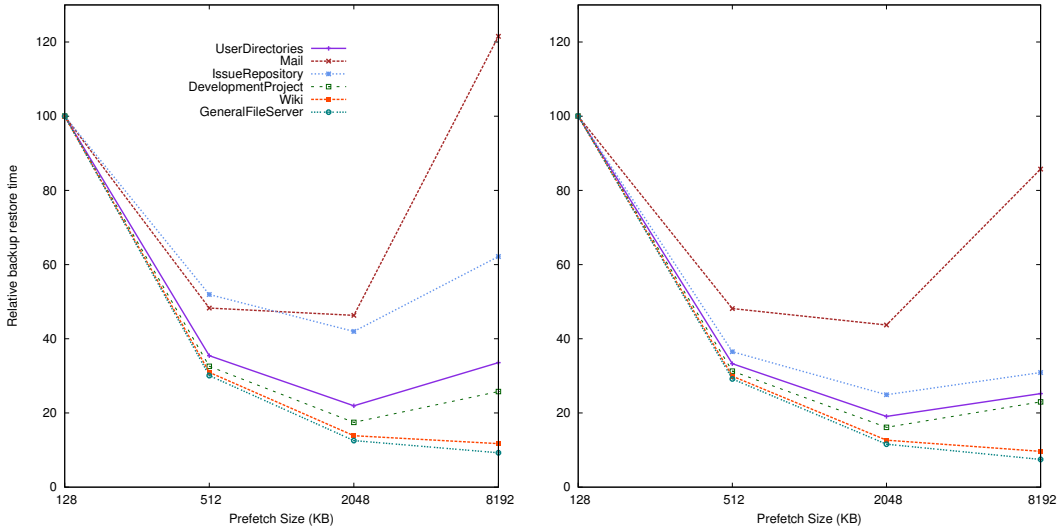


Figure 6.1: Relative restore time of the latest backup for each backup set, with inter-version fragmentation (left chart) and without it (right chart), as a function of prefetch size. Experiment performed with LRU cache eviction policy, 512MB cache size and common enterprise data center capacity HDD specification [72] (sustained data transfer rate: 175MB/s, read access time: 12.67ms)

two exceptions are clearly marked in separate sections and show possibility of further restore bandwidth increase, when using larger prefetch sizes with forward knowledge cache and after taking the scalability perspective into account.

Although the variable prefetch size can also be an option, it can only mask the fragmentation to some extent, especially when the streaming access is concerned. By reading smaller amounts of data when random read is detected, it may improve the current performance, but it may also decrease it if the streaming access is detected not soon enough. Also, each time the prefetch is modified from the maximal value, also the maximal possible performance suffers. Moreover, such solution requires many assumptions about the system and its architecture. Therefore, I decided to use the fixed prefetch size in my experiments and in order to extrapolate bandwidth based on number of I/Os performed in the test.

This measure ignores some speed variances due to file system physical fragmentation, faster seeks when particular I/Os are close to each other and

slower when they are far away in favor of the dominant cost: the single I/O read time.

6.1.2 Omitted factors

In my experiments I have omitted incremental backups (in systems with duplicate elimination, they are actually very similar to full backups, as only the new data is stored), which are often performed every day by many users. Unfortunately, the users who kindly agreed to usage of their data in my experiments did not have them. Even though the experiments with such data would be valuable, they would only extend the picture already presented by my experiments. What is sure, such backups cannot negate nor lower the problem of fragmentation, as after the whole week they end up having written similar new data in the same storage area. In fact, as the day to day modifications are smaller and more frequent, they may even make the problem more severe as the new data from one week is now divided into 5-7 areas instead of one.

In modern backup systems, being able to handle many backups at once is one of the key features. Even though in my experiments only a single stream load is verified, such approach lets me provide a repeatable way to perform experiments and show the results with optimal block placement on the disk (no data from other sets nor containers limiting the power of prefetch). Writing many streams at once leads to many issues connected with the implementation, which would require looking into the problem separately from each system perspective. As this was not my goal, I decided to provide the simplest implementation, which should actually be close to the optimal case for each system from both write and restore bandwidth points of view. Each additional stream being written at the same time requires solving at least the problem of storing all the streams in separate containers, which potentially introduces additional fragmentation.

The aspect of data retention, and therefore their deletion, is always present with backups systems and especially difficult when deduplication is taken into account. As a single backup system is stored for a quite long time, at some point a decision needs to be taken which backups to remove. This influences also data fragmentation. Actually, experiments show that the exact schedule for deleting backups does not particularly affect the results in another way than changing the overall deduplication factor [45]. Also, in case of my experiments, the number of backups in each data set is relatively small, therefore, applying a data retention policy to it and verifying the fragmentation changes would not allow me to draw sufficiently general conclusions.

One of the factors omitted in my experiments is also global deduplication

data set name	number of backups	avg. one backup size	avg. duplicate blocks*	avg. internal duplicate blocks
DevelopmentProject	7	13.74 GB	96.42%	17.81%
IssueRepository	7	18.36 GB	85.42%	23.20%
Wiki	8	8.75 GB	97.76%	17.79%
GeneralFileServer	14	77.59 GB	83.43%	17.38%
UserDirectories	50	76.15 GB	92.58%	19.34%
Mail	22	25.91 GB	97.76%	32.55%

Table 6.1: Data sets characteristics (* – data excluding first backup)

(within the whole system), which can be found in some of the systems on the market [23]. The main reason for that is the difficulty of performing tests and giving reliable results along with limited impact factor. The details of my decisions were presented in Section 3.2.3.

6.1.3 Data sets description

In order to diagnose the problem of fragmentation and verify proposed algorithms, I have gathered traces representing real user data of over 5.7TB in size and consisting of 6 sets of weekly full backups. The characteristics of each set are described in Table 6.1, while the types of their content are presented below.

- *DevelopmentProject* - large C++ project cvs data, LDAP data, server configuration files
- *IssueRepository* - issue repository data (contains XMLs and attachments), server configuration files
- *Wiki* - wiki data, server configuration files
- *GeneralFileServer* - home directories of computer science research laboratory (netware)
- *UserDirectories* - linux home directories of 18 users in a software company (tar)
- *Mail* - mailboxes of 35 users in a software company (tar)

6.1.4 Testing scenarios

Each test always starts with an empty system and beside the parameters (such as cache and prefetch size, caching algorithm, forward knowledge size) can be performed in three different scenarios:

- *base* - all backups from a data set loaded one after another (includes internal and inter-version fragmentation)

- *defrag* - all backups from a data set loaded one after another with CBR defragmentation enabled (both internal and inter-version fragmentation with the last one limited by CBR algorithm). Note that, this result will be shown only in experiments actually using CBR algorithm.

- *max* - only the last backup from a set loaded into the system (only internal stream fragmentation). This result can be referred to as the potentially maximal bandwidth level for the stream [it actually is maximal when unlimited cache size is used]. It can be considered realistic only with off-line deduplication, but only together with associated costs (see Section 2.2.1).

The goal of each scenario is to visualize the system in a state of being fragmented (*base*), defragmented (*defrag*) and not fragmented (*max*) in order to measure the effectiveness of presented algorithms and compare different options with no deduplication version (the x axis in all experiments) and between each other. Note that regardless of the scenario, the internal stream fragmentation is always present in a stream as it cannot be eliminated without decreasing deduplication level and changing the logical structure. Also, as already stated in Section 3.2.1, it highly impacts the final results, making the numbers in all scenarios sometimes far from the level achieved with no deduplication (in both: negative and positive way).

Another important observation is that the *max* scenario together with unlimited cache size can be regarded as the maximum bandwidth achievable in theory (as whole backup is placed in the one continuous area in the order of reading and all the blocks once read will never be evicted from cache).

LRU cache	Cache with forward knowledge (FK)	*base* scenario (with inter-version fragmentation)	*max* scenario (no inter-version fragmentation)
1GB	128MB + 2GB FK	0.94 - 50.92% (avg. 18.67%)	0.85 - 37.4% (avg. 14.84%)
128MB	128MB + 2GB FK	9.45% - 200.62% (avg. 72.2%)	8.84% - 183.41% (avg. 73.63%)
256MB	256MB + 8GB FK	6.67% - 185.47% (avg. 70.36%)	6.79% - 196.95% (avg. 87.76%)
512MB	512MB + 8GB FK	3.78% - 197.73% (avg. 66.42%)	3.84% - 238.54% (avg. 82.03%)
1GB	1GB + 8GB FK	2.50% - 211.34% (avg. 61.97%)	2.57% - 253.56% (avg. 67.64%)

Table 6.2: Backup restore bandwidth increase of cache with forward knowledge over standard LRU cache eviction policy in various configurations (measurements for the latest backup in each data set).

6.2 Evaluation of forward knowledge cache

6.2.1 Meeting the requirements

Performance results

The cache with limited forward knowledge presented in Chapter 4 does very well in optimizing the memory usage during restore of every backup (including the latest one) for both fragmented and not fragmented data (including off-line dedup), assuring an average restore bandwidth increase between 62% and 88% (see Table 6.2).

Moreover, for 4 out of 6 not fragmented data sets having only 256MB of cache memory together with 8GB forward knowledge already provide results almost identical to the ones achieved with unlimited cache size. For two others (*UserDirectories* and *Mail*) possible options are either to stay with 256MB size of cache and gain 22%-73% of additional bandwidth even when comparing to LRU with 1GB cache, or to use the same size of 1GB cache with 22%-253% bandwidth boost and additional 20% possible with larger forward knowledge. The exact results are shown in Figures 6.2 and 6.3, while their detailed analysis can be found in the following sections.

In addition to the above characteristics, the cache with forward knowledge

enables a range of choices based on the resources available and the restore bandwidth requirements. It is possible to choose between the cheaper option with 8 times lower memory usage and still slightly better performance (1GB LRU vs 128MB with forward knowledge), and the one with the same amount of memory, but higher performance (see Table 6.2). Depending on the actual system usage pattern, both options sound very interesting with a significant leap from currently most popular LRU algorithm as the cache replacement policy.

Additional resource usage and possible trade-offs

As described in details in Section 4.4.4, the usage of limited forward knowledge requires additional resources, which should be included in the total costs. In the most effective case those are: memory (about 13MB for 8GB of forward knowledge) and bandwidth (about 0.256% decrease). Although the second one is small enough to become negligible, the first one can make some difference, especially when the total amount of cache memory is small. Even though assuming 256MB of cache as the most effective in general, having 8GB of forward knowledge causes only about 5% higher memory usage. This cost does not seem to be high, assuming the bandwidth increase and how well it approximates infinite forward knowledge.

Note that in my experiments this additional memory is not included by default in the total cache size. This enables clear and easy comparison between different forward knowledge sizes and their impact on the performance while keeping exactly the same cache size. Also each of the possible implementations require different amount of memory, which would be complicated to visualize and would require much more testing.

Tunability

The cache with forward knowledge is also tunable by setting the size of requested forward knowledge at the cost of additional memory. In general, the higher the forward knowledge the better the solution, but in detail, this property is limited and relies on the internal duplicates pattern, the size of cache memory and the state of the stream (fragmented or not). As already mentioned in Section 4.5, the desired solution would be to automate the memory division between the actual cache and the forward knowledge within some total amount of memory available in order to secure the best performance results.

Code modifications and deduplication

Although code modification is required to use the algorithm in some given implementation, it is very limited and does not impact deduplication effectiveness. The two modifications which are necessary consider only the algorithm responsible for the data restore in general and the cache memory management using the interfaces already available. The former one is requested only in order to fill the oracle with the actual forward knowledge, and it can be easily done by attaching proper information to each standard read request, making the modification almost invisible from other system components perspective. The latter one, on the other hand, is limited to the restore algorithm, only making it easy to simply swap the implementation. Such limited modifications make the algorithm suitable for most (or possibly even all) systems present on the market.

6.2.2 Setting the forward knowledge size

Figures 6.2 and 6.3 show the impact of cache with forward knowledge, both limited (to 512MB, 2GB, 8GB) and unlimited (the same as *adopted Bélády's cache* used before in this work), together with the comparison to standard LRU algorithm.

In both figures we can notice very good results when using actually any amount of forward knowledge, although the highest gain (in %, when compared with LRU) is almost always possible with the smallest cache size. This is because small amount of cache makes LRU algorithm highly ineffective, as before the block is requested again it already becomes evicted from cache (best visualized with *DevelopmentProject* and *GeneralFileServer* data sets). With forward knowledge each block in cache has its own purpose and is not evicted until used at least once (with some rare exceptions when prefetched blocks are to be read earlier than some others already present in the cache). Also, the small amount of memory makes the cache utilized in 100% in almost all the cases and throughout the whole experiment, which is not always true with higher values (see Section 6.1.1 for details). For example, not fragmented *DevelopmentProject* achieves already maximal bandwidth with only 128MB of cache memory, even when having infinite forward knowledge.

Increasing forward knowledge always helps to improve the achieved results. The gain, though, is highly correlated with the amount of cache used and the pattern of internal duplicate blocks present in a stream. The problem of duplicates defines the minimal size of memory necessary not to reread blocks from disk, which is in fact the desired size of the cache. Being able to find all the blocks to keep in memory in the limited forward knowledge and

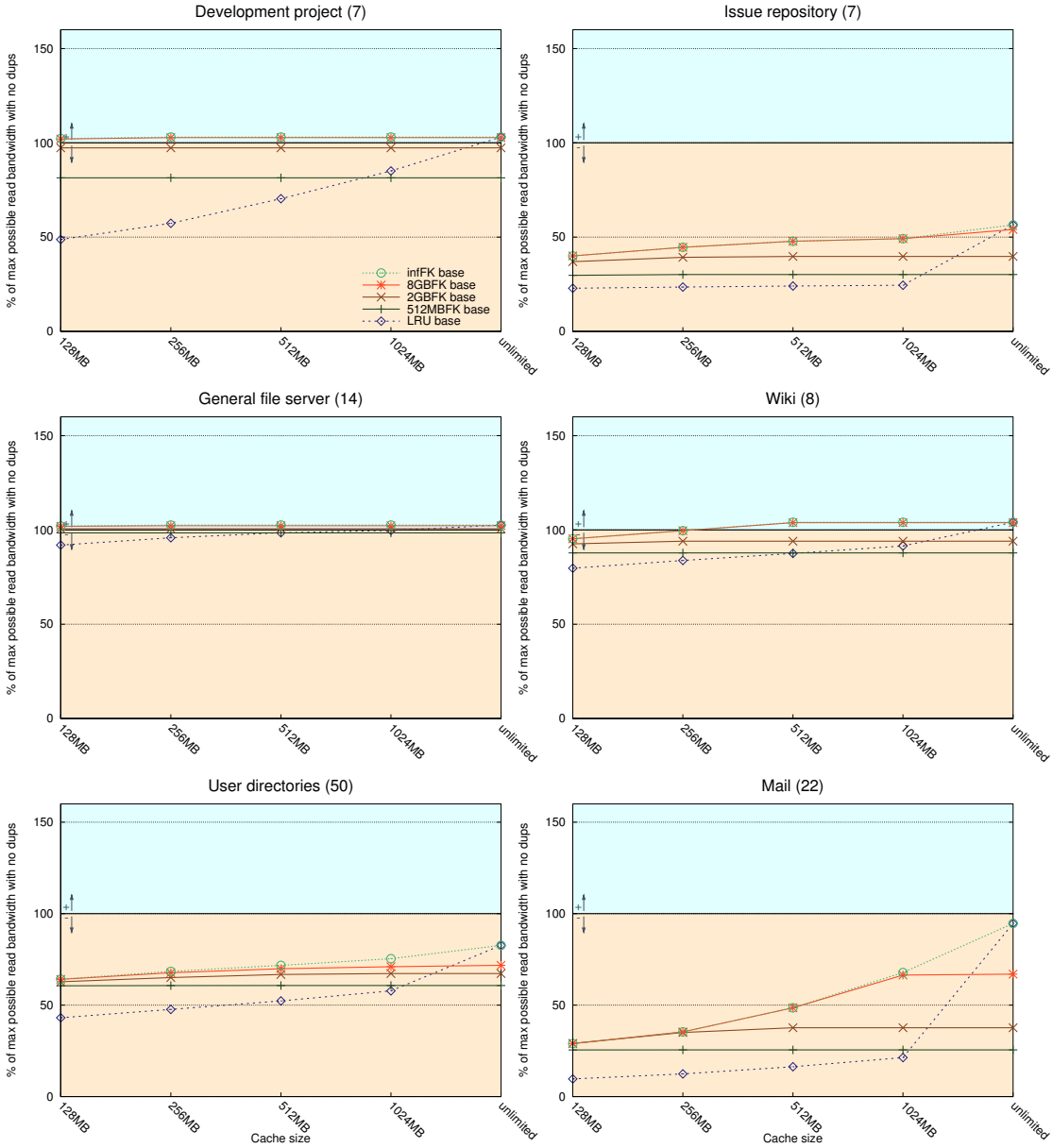


Figure 6.2: Impact of forward knowledge size on restore performance of the latest backup with storing previous backups of the set before (*base* scenario)

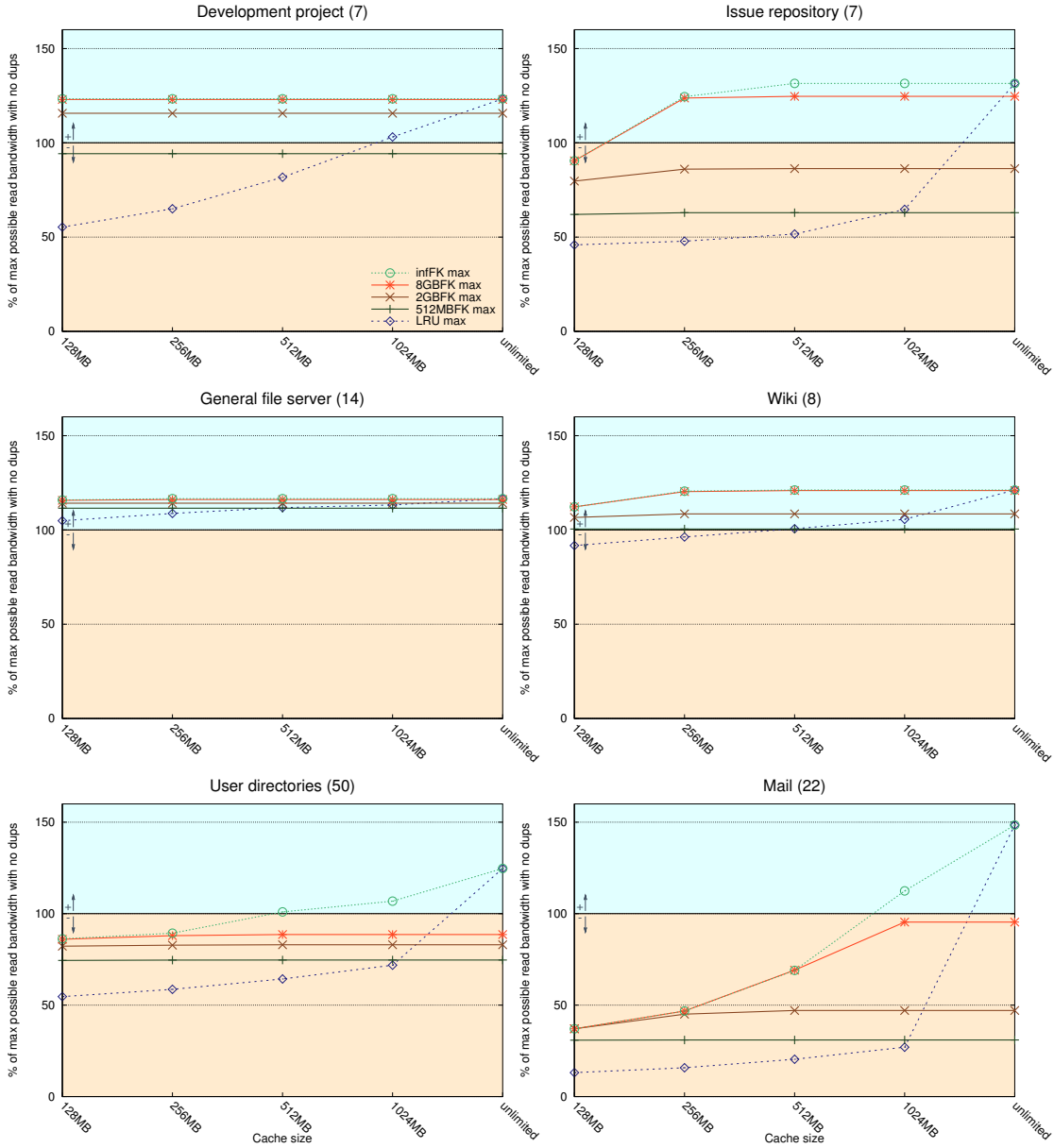


Figure 6.3: Impact of forward knowledge size on restore performance of the latest backup when it is stored as the only one in the system (*max* scenario)

having the required size of the memory makes the process the most effective. This characteristic can be noticed especially in case of *Mail* data set, which contains the highest amount of internal duplicates. On both figures (fragmented and not) having 1GB of cache and 8GB of forward knowledge gives significantly better results than with lower memory and forward knowledge sizes.

On the other hand, there are many cases where limited forward knowledge actually limits the cache memory usage. In my implementation, whenever the cache with forward knowledge is simulated, it keeps in the memory only the blocks which are to be used in the future (found in forward knowledge). Therefore, the cache amount in this case should be seen as a top limitation rather than the specific amount of memory in use. The actual value can vary throughout the simulation, but at some point it reaches its peak, which means that adding extra memory will not improve the results (unless more forward knowledge is used). Such scenario is best seen with forward knowledge limited to 512MB. In this case more cache than 128MB will not bring any visible benefits for any of the data sets presented as not more than 128MB will be actually used. With other limits for the future knowledge such border is different for each data set and can be easily read from Figures 6.2 and 6.3.

In order to have the whole picture, it is interesting to look at the forward knowledge with relation to the size of the whole backup. As we can notice when comparing Figures 6.2 and 6.3, one globally true claim seems to be that fragmented data needs less forward knowledge than not fragmented data (see next section for details), which leads to the conclusion that the memory for the forward knowledge should change with the life of a data set. Other insights are dependent on the stream detailed characteristics rather than on the stream size. When we look at the charts, having 2GB of forward knowledge is perfectly enough for all data sets with 128MB cache while for 256MB it is a bit short, especially for the *IssueRepository*, which is actually quite small. One thing which may change when having very large streams is the distance to optimal algorithm using unlimited memory, which is understandable. This is the case especially with *UserDirectories*.

6.2.3 Impact of fragmentation on required cache size

An interesting fact can be observed when comparing once more Figures 6.2 and 6.3 for the efficiency of cache memory usage with different forward knowledge sizes. While for the first one (with inter-version fragmentation) 8GB of forward knowledge is enough even for 1GB cache to stay within at maximum 8% of the algorithm with infinite forward knowledge (avg. 2.48%), the not fragmented option has higher requirements, because of more data worth

data set name	*base* (with inter-version fragmentation)	*max* (no inter-version fragmentation)
DevelopmentProject	212 MB	94 MB
IssueRepository	2654 MB	309 MB
Wiki	353 MB	269 MB
GeneralFileServer	184 MB	181 MB
UserDirectories	3916 MB	1887 MB
Mail	2230 MB	1366 MB

Table 6.3: Cache memory actually used when performing experiments with infinite forward knowledge (peak usage for the latest backup in each data set) [memory required for forward knowledge not included]

keeping is restored with every I/O. In this case 8GB of forward knowledge works extremely well for up to 256MB cache (at maximum 2.3% deviation from no limit option; avg 0.83%) with already showing shortage while having 512MB (max. 19.25%, avg. 5.48%). With this and bigger cache options, longer forward knowledge is required. Note that in my experiments only the blocks found in forward knowledge can be kept in cache (see Section 6.1.1 for details). If the forward knowledge is short or there is only small number of blocks which are to be used in the future, the cache memory may not be fully utilized, which can be often noticed on the figures when two results with different memory sizes are the same.

In order to measure the maximal memory requirements for each data set, I have performed a test with the unlimited amount of memory and infinite forward knowledge. The results in Table 6.3 show that data fragmentation has significant impact on required memory even in the case of having forward knowledge. With 3 out of 6 cases the memory requirements have doubled after allowing the inter-version fragmentation, while for *IssueRepository* they were multiplied by 9 times. The requirements for the remaining two data sets stayed at a quite similar level.

6.2.4 Experimenting with larger prefetch

Because of the observations from Section 6.1.1 most of my experiments were performed with fixed default prefetch of size 2MB, as it was the most effective for the most common LRU algorithm point of view and provided easy

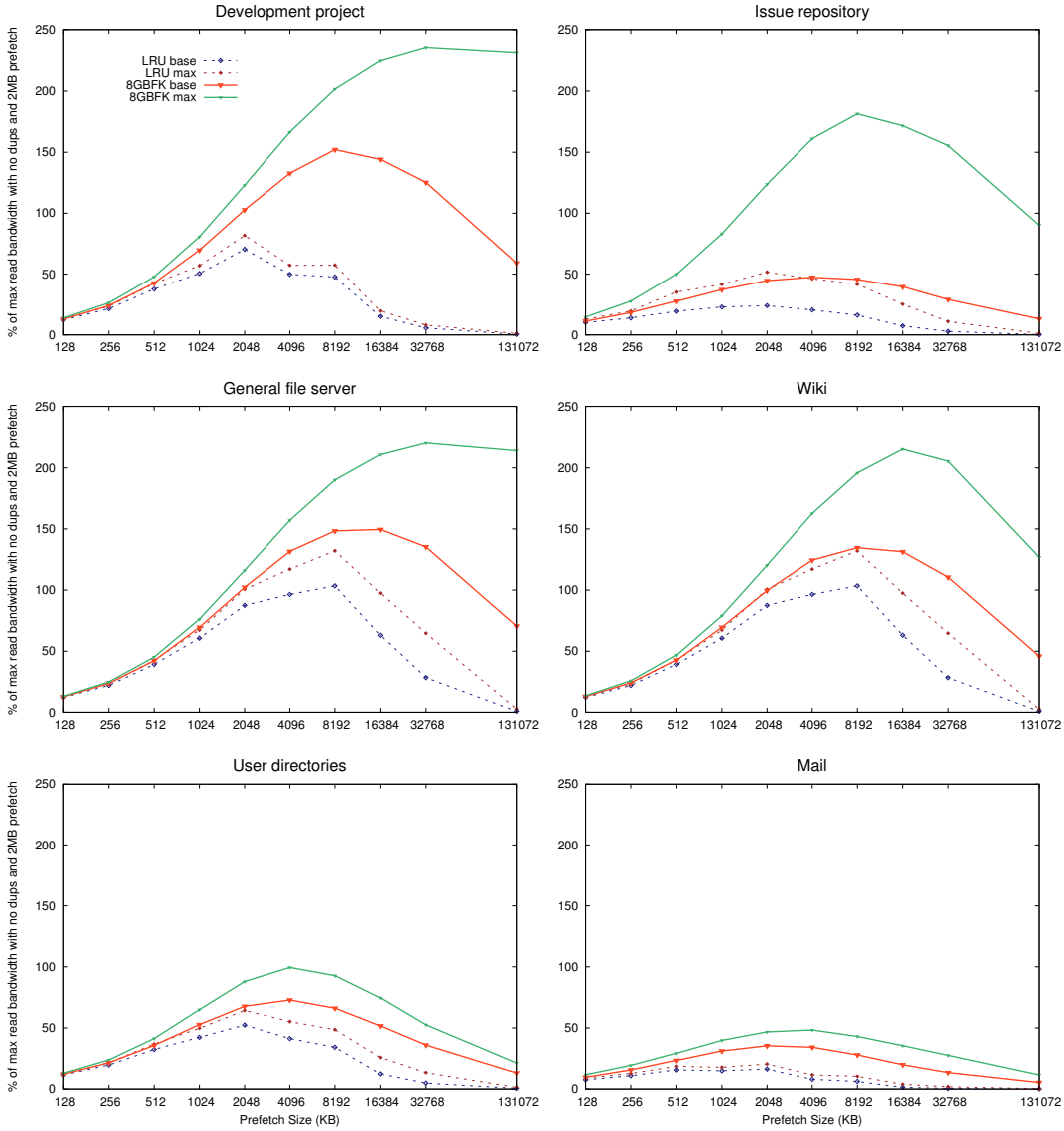


Figure 6.4: Impact of prefetch size on restore bandwidth of the latest backup with and without forward knowledge for both fragmented (*base*) and not fragmented (*max*) data [100 - restore bandwidth of a system with no deduplication and prefetch size 2MB]. Cache size in all cases equals 256MB. Common enterprise disk characteristics used [72].

comparison between different algorithms. Such level of prefetch size (2-4MB) is also similar to the one used in many papers [45, 54], suggesting that it can be regarded as the most common one. Nevertheless, it turned out that having caching algorithm with forward knowledge modifies those assumptions significantly. In order to visualize the difference in restore bandwidth with relation to prefetch size, I have performed a simulation with common enterprise disk characteristics [72] (sustained data transfer rate: 175MB/s, read access time: 12.67ms). The results shown in Figure 6.4 suggest that every backup in each condition (fragmented and not fragmented), and using different restore algorithm, has its own optimal prefetch size, which can differ a lot between each other. The one clear observation is that such optimal prefetch is always larger for not fragmented data when comparing to fragmented one, and for the forward knowledge algorithm when comparing to LRU. As a result, switching to the larger prefetch improves the restore bandwidth through a smaller number of I/Os which limits unproductive seeks. Thanks to the forward knowledge algorithm the prefetch size can be larger by 2 to 16 times than with LRU, therefore providing maximal restore bandwidth increase at the level of 11%-117% (avg 68.47%) for fragmented data and 27%-252% (avg. 120.24%) for not fragmented data. When comparing to the results with forward knowledge and 2MB prefetch, extending prefetch size can give an additional gain of 0%-48% (avg. 23.89%) for fragmented and 3%-92% (avg. 53.90%) for not fragmented data.

6.3 Evaluation of CBR effectiveness

6.3.1 Meeting the requirements

Performance results

The CBR algorithm presented in Chapter 5 is very effective when eliminating the inter-version fragmentation impact for all the traces. In the common scenario with 256MB of LRU cache the resulting restore bandwidth of the latest backup in each data is on average within 2.48% (from within 21.29%) of the maximal one, which is achieved with no inter-version deduplication (for example by storing single backup). Even though this indicates on average only 29.1% (8%-94%) restore bandwidth increase, the important fact is the perspective of further degradation which should be taken into account. Unfortunately, the true potential of the algorithm could not be shown here due to the lack of traces covering many years of backups (see Section 3.3.2 for details).

When looking more deeply into results shown in Figure 6.5, one can make

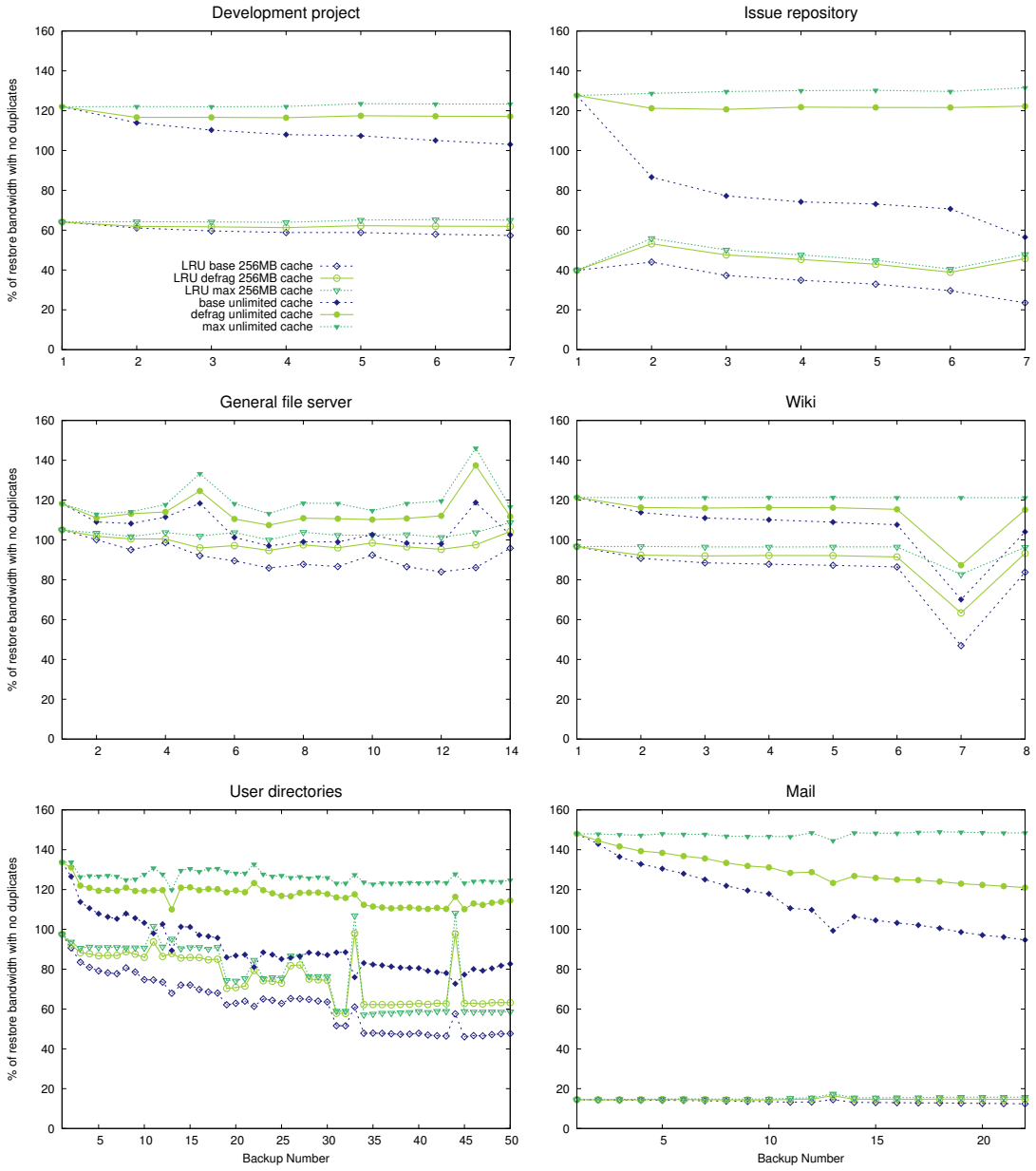


Figure 6.5: The restore bandwidth achieved by CBR algorithm in two scenarios: LRU basic algorithm with 256MB cache and unlimited cache size (measurement in all scenarios performed after each backup).

some interesting observations specific to each data set. For example, the biggest increase in fragmentation occurs for backups 2 and 7 of *IssueRepository*. This is caused most likely by data deletion, because these backups are the only ones significantly smaller than their predecessors. On the other hand, the peaks visible on *UserDirectories* and *Mail* charts are caused by not fully completed backups, while other peaks usually differ much in backup stream characteristics (number of duplicates, unique blocks, backup size) from usual ones in a set. Unfortunately, I was not able to verify the core reason of those deviations.

Additional space and resources used

My algorithm does not use additional space except for rewritten duplicated blocks, therefore, the additional space consumption is below 5% of all blocks. Actual number is much lower – between 0.35% and 3.27% (avg. 1.58%). Old copies of the blocks are removed in the background, for example as part of the deletion process running periodically, so the space consumption is only temporary. Additional disk bandwidth consumption is also limited to writing rewritten blocks.

Tunability

The presented algorithm is also easily tunable by setting the percent of blocks to be rewritten. The higher the percentage, the better restore performance at the expense of a bigger drop in write performance and more disk space required for storing temporarily old copies of the rewritten blocks.

6.3.2 Cost of rewriting

When evaluating the cost of the presented algorithm, I have estimated the slowdown of the backup process caused by rewriting. Since the CBR rewrites duplicates as non-duplicates, in order to establish such operation cost, I have modified the write path of a commercial backup system HYDRAstor [23, 56] to avoid checking for duplicates, and compared the resulting bandwidth to the bandwidth of unmodified system when writing 100% of duplicates.

As a result, the bandwidth of duplicates was 3 times higher than non-duplicates. Based on this number, I have used a factor of 4 slowdown for rewriting a block (1 for standard duplicate write/verification + 3 for extra write) vs. deduplicating it. For example, 5% blocks rewritten cause from 5.17% up to 15% slowdown. Since all rewritten blocks are duplicates, the actual slowdown depends on the percentage of duplicates in the original stream

data set name	B/W before defragmentation (*base*) as % of *max*	B/W after defragmentation (*defrag*) as % of *max*	% of blocks rewritten	write time increase
DevelopmentProject	88.15%	95.22%	0.86%	2.40%
IssueRepository	49.15%	95.63%	2.18%	4.59%
Wiki	86.89%	97.54%	2.29%	6.57%
GeneralFileServer	88.13%	95.83%	0.35%	0.94%
UserDirectories	81.24%	107.74%	0.52%	1.39%
Mail	78.67%	93.14%	3.27%	9.38%

Table 6.4: Impact of CBR defragmentation on the latest backup restore bandwidth of each data set with maximal bandwidth (*max* - without inter-version fragmentation) normalized to 100%. All the tests preformed with 256MB of cache memory and LRU eviction policy.

– the higher the percentage, the higher the slowdown, and 15% slowdown is achieved when all blocks in the stream are duplicates.

The maximum presented slowdown seems significant, but as the experiments show, the algorithm hardly ever reaches the maximal allowed rewrite (see Table 6.4). This is because I am very conservative since the minimal rewrite utility is set high at 70% and I always observe the 5% limit while processing backup streams. As a result, the CBR increases the backup time by 1%-9% (avg. 4.21%; see Table 6.4), which seems reasonable. However, there still exists a possibility to set smaller limit of rewritten blocks in order to decrease the potential costs and perform only the rewrites with maximal gain.

The alternative option to reduce the cost introduced by rewrites is to perform the algorithm only every n-th backup. Such solution should also work very well, and at some cost of restore bandwidth, introduce smaller overhead during the backup process.

Note that this trade off addresses also the amount of resources required performing off-line deduplication in the background and the temporary space needed after the backup as they are proportional to the number of rewritten blocks.

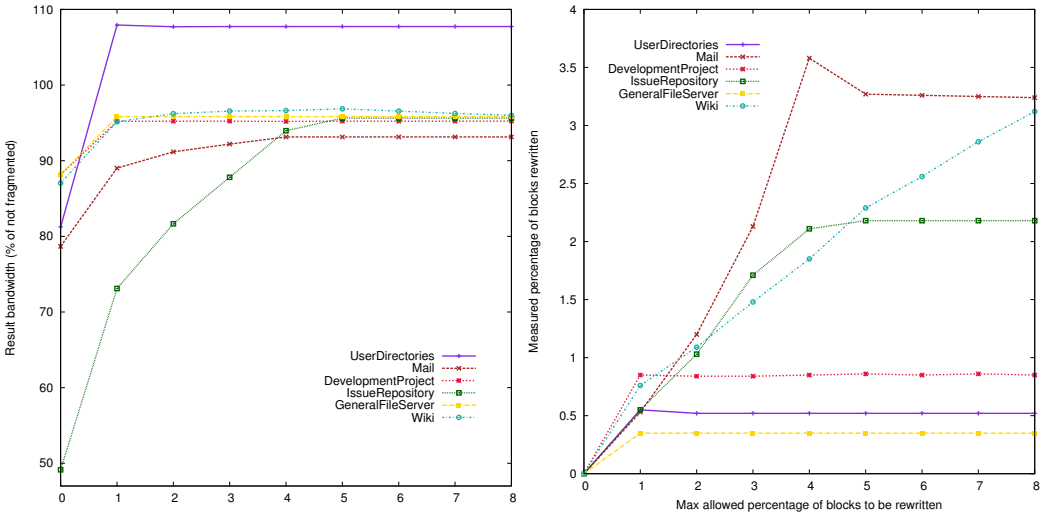


Figure 6.6: The impact of rewrite limit percentage on the latest backup in each data set

6.3.3 Setting the rewrite limit

To select the best value for the rewrite limit, I performed experiments varying this limit from 0% to 8% while keeping the minimal rewrite utility unchanged at 70%. The results for the latest backup in each backup set are given in Figure 6.6. Setting this limit to low values such as 2% or even 1% works well for all sets except *IssueRepository*, for which the rewrite limit of 5% offers the lowest reduction in restore bandwidth. Increasing this limit beyond 5% does not give additional boost and may increase the backup time significantly, so I decided to set this limit to 5% for all experiments. Even though with this setting the maximal theoretical write bandwidth drop is at the level of 15%, in reality it is on average only slightly above 4%. Also the maximum drop is achievable only with 100% duplicate stream, for which the bandwidth is already very high.

Note that, for most data sets the number of rewritten blocks is proportional to the restore bandwidth gained. This correlation is pretty weak in case of *Mail*, where internal fragmentation is very severe, and is not true for the case of *Wiki* data set. The latter one is caused by very unusual backup just before the last one, with about 15 times more blocks added and many more deleted than in standard backups of this set. The algorithm is trying to defragment the backup making a lot of rewrites, while the next one (the

last in the set) is rather similar to the others, which makes the algorithm to basically rewrite most of the blocks from the previous backup again.

The interesting fact is also that the *UserDirectories* restore bandwidth after defragmentation is actually better than the version with no fragmentation (stored as a single and only stream in the system). This is due to the block reordering, which luckily made the caching more effective. This observation also shows that there exists potential in writing backup in a slightly different order than the one requested by the user, but as some of my other tests suggest, such effect is possible only with LRU algorithm as it is not very effective in general (it would require forward knowledge about the whole stream being written and rearranging the block on the fly or expensive background operations). When the cache is equipped with forward knowledge such phenomenon does not happen.

Rewritten rewrites

My experiments show that even 39% to 97% of all rewritten blocks in the latest backup are the ones which were already rewritten in one of the previous backups. The highest number is reached in backups with very low number of new blocks, resulting in many iterations required to finally achieve the context of enough size. Even though they are rewritten again, it does not mean that they are unnecessary (the experiments disabling the possibility of rewrites already rewritten in previous or any backup showed 10-20% drop in final restore performance). In some cases the rewriting helps to decrease the rewrite utility only a little, not reaching below the required level, or simply moves the blocks to the neighbourhood, which increases the possibility of being read before needed, but without the visible impact on its rewrite utility value (because of restore cache). Both aspects are well visualized with the results of modified algorithm in such a way, that in order to rewrite a block, at least one non duplicate block should be found in its stream context (in order to assure the decrease of its rewrite utility for the next time). Such experiment significantly (even by half) reduced the number of rewritten blocks, but it also reduced the achieved restore bandwidth by a few percent. Similar results can be achieved with increasing the stream context up to even 100MB.

Since the overall number of rewritten blocks is still very small, I have decided to keep the version of the algorithm assuring better restore bandwidth.

6.3.4 Effect of compression

So far we have assumed that the backup data is not compressible. If we keep the prefetch size constant and equal to 2MB, the compressible data

data set name	*base* (with inter-version fragmentation)	***defrag* (CBR defragmentation)**	*max* (no inter-version fragmentation)
DevelopmentProject	212 MB	**99 MB**	94 MB
IssueRepository	2654 MB	**312 MB**	309 MB
Wiki	353 MB	**272 MB**	269 MB
GeneralFileServer	184 MB	**182 MB**	181 MB
UserDirectories	3916 MB	**2196 MB**	1887 MB
Mail	2230 MB	**1375 MB**	1366 MB

Table 6.5: Required cache memory to assure maximal performance with infinite forward knowledge (peak usage) for the latest backup in each data set – the effect of CBR defragmentation (extension of Table 6.3) [memory required for forward knowledge not included].

results in fragmentation increase and the CBR algorithm delivering even better relative improvements in restore bandwidth. For example, for 50% compressible data, the drop in restore performance increases on the tested data sets from 12%-51% (avg. 21.27%) to 16%-56% (avg. 26.12%), whereas the CBR defragmentation improves the restore of the latest backup by 11-121% (avg. 35.79%) instead of 8%-95% (avg. 28.94%), resulting in total drop reduction up to 10% (instead of up to 7% without compression). All results were achieved with a very similar number of blocks rewritten.

Obviously, selecting different prefetch size, based for example on compressibility of data, could change the above results.

6.3.5 Impact of CBR defragmentation process on required cache size

In order to verify the process of CBR defragmentation in terms of cache memory required, I have performed a test of reading the last backup of each data set after the defragmentation with infinite forward knowledge and potentially unlimited cache size. The actual peak memory usage in each case can be found in Table 6.5. The gathered numbers suggest that the CBR defragmentation process works very well in terms of limiting the memory usage and therefore making the latest backup similar to the one never fragmented in the memory usage area.

data set name	LRU *base*	LRU *defrag*	FK8GB *base*	**FK8GB *defrag***
UserDirectories	47.65	+32.61%	+42.12%	**+ 72.30%**
DevelopmentProject	57.33	+ 8.02%	+79.26%	**+103.52%**
IssueRepository	23.51	+94.56%	+89.75%	**+388.90%**
Mail	12.39	+18.40%	+185.47%	**+238.42%**
GeneralFileServer	95.84	+ 8.74%	+ 6.67%	**+ 16.28%**
Wiki	83.80	+11.29%	+18.87%	**+ 36.47%**
Average		+28.94%	+70.36%	**+142.65%**

Table 6.6: Comparison of different restore options with 256MB cache as % of increase over LRU *base* scenario bandwidth (latest backup only). Note that, LRU *base* results are shown as "% of max possible bandwidth with no duplicates" – the standard metric used throughout this work.

6.4 Combined impact of both algorithms

Figure 6.7 shows detailed results for both CBR defragmentation with limited forward knowledge cache algorithms in a single and combined options for the latest backup with different cache sizes. Two algorithms used to fight different aspects of fragmentation end up in a very effective symbiosis resulting in 16-388% bandwidth increase (avg. 142.65% - see Table 6.6) for different data sets with 256MB as an example.

Furthermore, the algorithm produces very good results when compared with the maximal possible restore bandwidth achieved with unlimited cache size having only 256MB of cache memory (see Table 6.7). In four out of six cases the results were at most 13% from the theoretical maximum leaving not much space for improvement, while the remaining two cases still fall behind. *UserDirectories* (-34.15%) is a quite big data set and require both bigger cache and higher forward knowledge in order to deliver better results, while *Mail* (-71.15%) includes large portion of internal duplicate blocks which require more memory for efficient caching. In this case more forward knowledge may be beneficial after reaching 1GB of cache.

Figure 6.8 shows the fragmentation process in time and the impact of each proposed algorithm with using 256MB of cache memory in comparison with the *base* LRU scenario and the *max* scenario with unlimited cache size. When looking at the charts joined impact of both CBR and limited forward

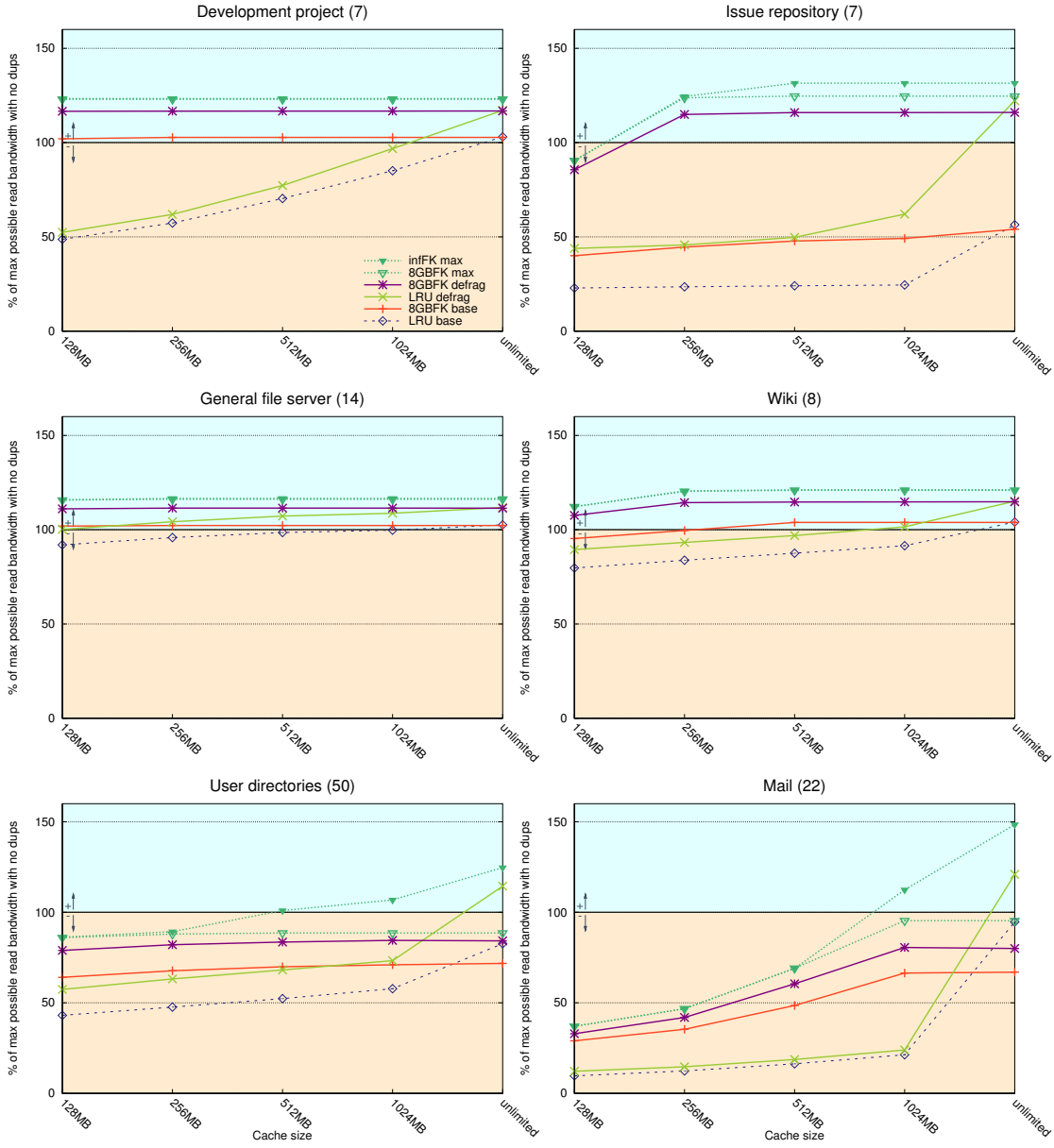


Figure 6.7: Impact of CBR with forward knowledge cache on the latest backup (total number of backups in each set can be found in parentheses next to the backup set name).

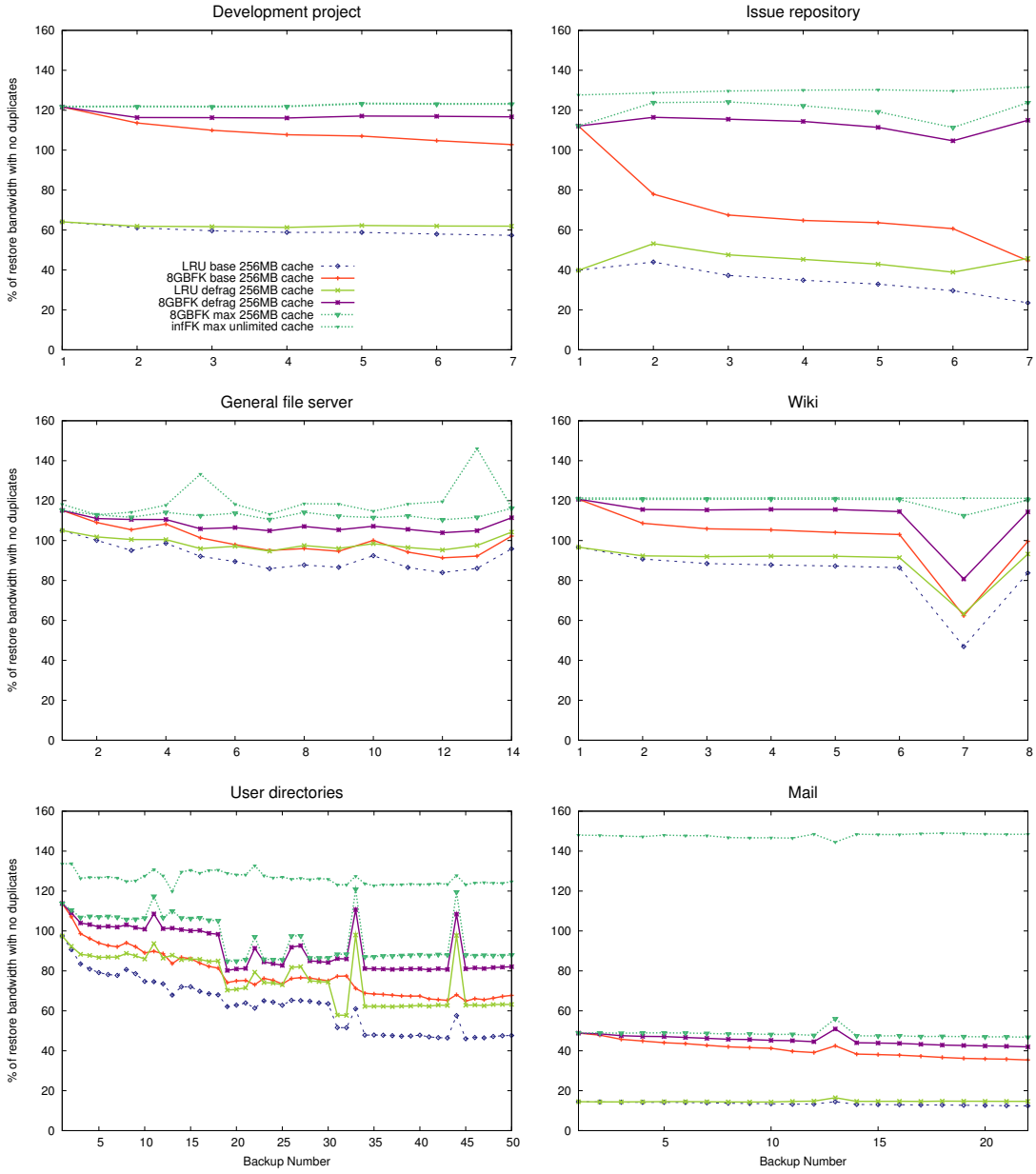


Figure 6.8: Impact of CBR with forward knowledge cache backup after backup on different data sets (measured after each backup).

data set name	LRU *base*	**FK8GB *defrag***	FK8GB *max*	infFK *max*
UserDirectories	-61.79%	**-34.16%**	-29.46%	-28.38%
DevelopmentProject	-53.52%	**- 5.40%**	- 0.29%	- 0.00%
IssueRepository	-82.12%	**-12.59%**	- 5.88%	- 5.34%
Mail	-91.65%	**-71.75%**	-68.49%	-68.49%
GeneralFileServer	-17.80%	**- 4.42%**	- 0.39%	- 0.00%
Wiki	-30.84%	**- 5.63%**	- 0.69%	- 0.50%
Average	-56.29%	**-22.32%**	-17.54%	-17.12%

Table 6.7: Comparison of different restore options (256MB cache) with relation to the maximal bandwidth (*max* scenario + unlimited cache)

knowledge algorithms works very well keeping the results extremely close when comparing to the scenario when the data was never fragmented at all (8GBFK *defrag* vs 8GBFK *max*). For all the backups there is only one case when the deviation is higher than a few percent.

On the other hand, based on the traces I was able to gather, it is quite difficult to predict whether this mentioned deviation can stay at the same small level for hundreds or thousands of future backup. Even if this is not possible, the impact of fragmentation will be limited to the fraction of the one showing without this solution and in fact may never be noticed by the potential end-user.

When looking at Figure 6.7 and the same results gathered in Table 6.8, we can notice one important fact. Thanks to using both algorithms it is possible to decrease memory demands 8 times (from 1024MB to 128MB) and end up with higher performance (11.35% - 249.61%; avg. 67.74%). What is more, for 4 out of 6 data sets the restore bandwidth results with 128MB cache were higher than with unlimited memory in the LRU case with fifth data set results very close (*UserDirectories* - 4.52% lower) and the last (*Mail* - 65.22% lower) left behind because of its high memory requirements and the specific pattern of internal stream duplicates.

The results suggest that many data sets require only fraction of memory, which is usually allocated today, and only some may benefit from the larger amount, but only when efficiently used. In general, the proper amount of memory should be allocated rather dynamically during the restore process based on the memory available, user requirements and the exact needs

data set name	LRU *base* with 1GB cache (1)	LRU *base* with unlim-ited cache (2)	FK8GB *defrag* with 128MB cache (3)	(3) vs (1)	(3) vs (2)
DevelopmentProject	85.08	103.04	116.62	37.07%	13.18%
IssueRepository	24.49	56.46	85.62	249.61%	51.65%
Wiki	91.51	104.06	107.58	17.56%	3.38%
GeneralFileServer	99.74	102.58	111.06	11.35%	8.27%
UserDirectories	57.8	82.68	78.94	36.57%	-4.52%
Mail	21.34	94.66	32.92	54.26%	-65.22%
Average				**67.74%**	**1.12%**

Table 6.8: Comparison of both new algorithms with 128MB cache memory and the old LRU in two options: with 1GB and unlimited cache size. Note that, the results in the first three columns are shown as "% of max possible bandwidth with no duplicates" – the standard metric used throughout this work.

requested by each data stream.

6.5 Scalability

Last but not least, with current systems using very often 10 or more disks in order to restore a single block [23, 45, 85] through RAID or erasure coding [82] and serving many streams at the same time, all the above results can be brought into another level. In my experiments I assumed 2MB prefetch for the whole stream, which in the above setup means only 200KB prefetch per disk. When using recent disk drives [72] such small prefetch means almost 6 times higher restore time from a single disk when compared with the 2MB (see Table 3.1).

As it has already been mentioned before in case of the systems with deduplication, the higher prefetch does not always mean higher performance. When looking at the results with common LRU algorithm (see Figure 6.9),

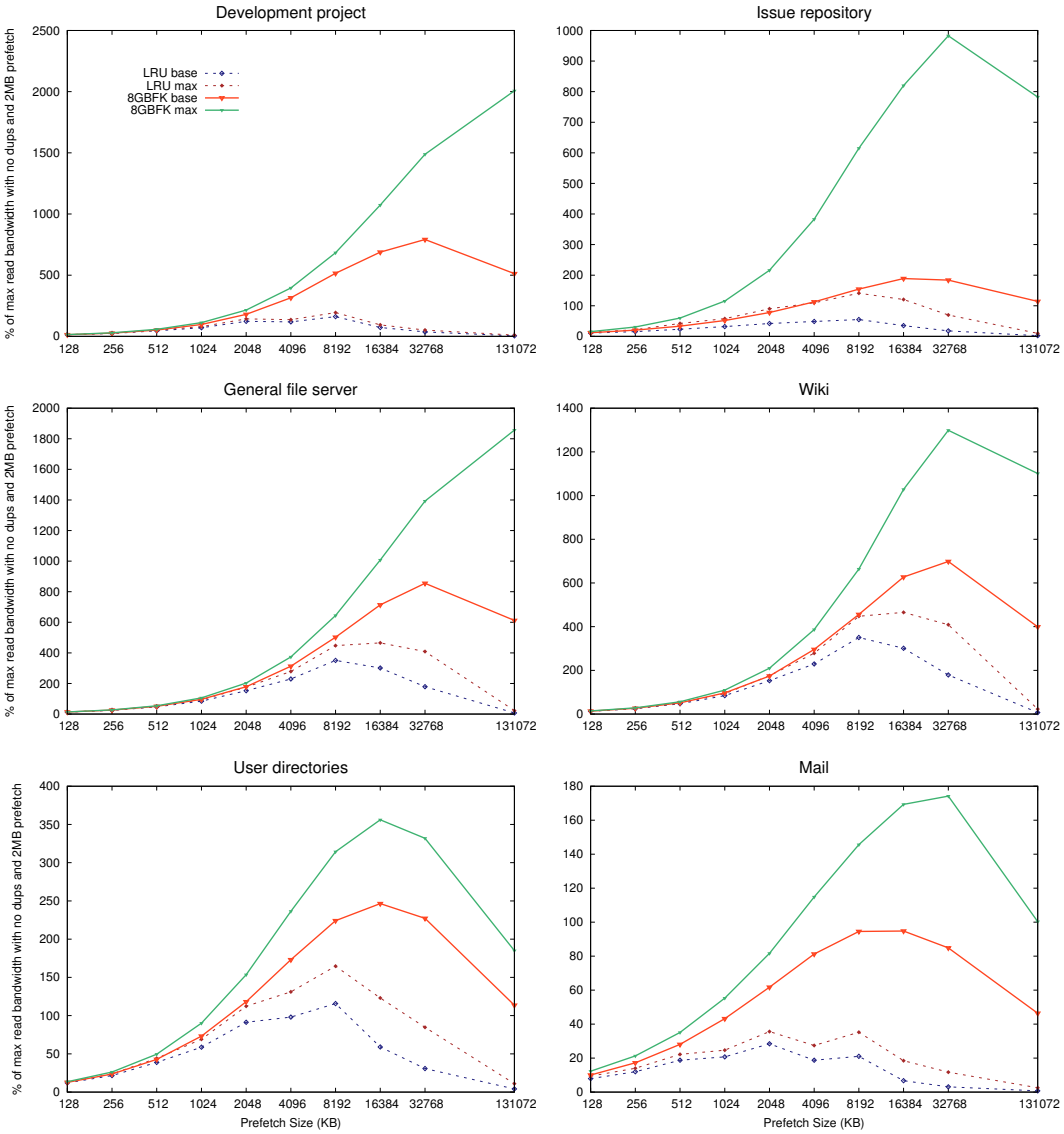


Figure 6.9: Impact of prefetch size on restore bandwidth of the latest backup when considering 10 disk bandwidth with and without forward knowledge for both fragmented (*base*) and not fragmented (*max*) data. Cache size in all cases equals 256MB. Common enterprise disk characteristics used [72]. Note the different scale on each graph.

even having the speed of ten drives (10 x 175MB/s) the further growth of the prefetch above 8MB (800KB/drive) gives slightly positive impact only in one case and only for not fragmented data.

The results look completely different when the cache algorithm with forward knowledge is taken into account. In all cases 32MB prefetch gives a few times better results and in two cases (*DevelopmentProject* and *GeneralFileServer*) even higher results with larger prefetch are available. In a single case only (*UserDirectories*) 16MB prefetch is slightly better than 32MB. In details, when moving from the best LRU prefetch (chosen separately for each data set) to best prefetch with forward knowledge algorithm, we can gain additional 75%-390% (avg. 184.23%) for fragmented and 116%-933% (avg. 396.77%) for not fragmented data. Comparing with the forward knowledge algorithm and 2MB prefetch simply increasing, the prefetch size can increase the results by up to 109%-379% (avg. 254.72%) and 132%-835% (avg. 531.25%) respectively.

Having the above numbers, increasing the prefetch size seems a very interesting option. One needs to remember, though, that such operation introduces higher latency variations, which may be important for some type of usage. Fortunately, with secondary storage system it will not be an issue, as bandwidth is the most important in this case and the higher latency can be easily accepted.

Examining the prefetch size brings one more observation. The larger the size the more visible the difference between fragmented and not fragmented data. As with LRU standard algorithm and its best prefetch size for each data set the defragmentation could give about 20%-156% (avg. 59.72%) bandwidth increase, the same gain for forward knowledge algorithm with its best prefetch size can achieve 44%-420% (avg. 164.18%). Such results suggest even higher importance of proper defragmentation algorithm.

In order to verify the ability to defragment data with proposed CBR defragmentation algorithm, I performed a simulation with only two parameters modified. The prefetch size was set to 32MB (3.2MB/drive), which seems to be close to optimal with 10 drives, and the stream context to 80MB, in order to preserve the proportion to prefetch size. The cache size was still 256MB with forward knowledge set to 8GB. The achieved results without any additional tuning were actually pretty good. The algorithm was able to gain 36%-333% (avg. 97.66%) of restore bandwidth, ending up with a result only 4% to 19% (avg. 8.88%) lower than totally not fragmented stream. The only data set which was hard to defragment in the above setup was *Mail*. In this case the final result was 34% lower from the not fragmented stream after a still significant 36% increase over the fragmented version.

To sum up, I performed one more experiment showing the importance of

using many disks for restore and the algorithms introduced in this thesis. Assuming 10 disks used I compared two algorithms with 256MB of cache: 8MB prefetch LRU (representing the level used often in today's systems [45, 54]) versus 32MB prefetch with forward knowledge (8GB) and CBR defragmentation. The resulting restore bandwidth of the latest backup depending on the data set was from 3.5 up to 16 times higher with an average of slightly above 8 times. Going simply to 8MB prefetch with LRU, which is best when considering all the data sets and 10 disk drives, gives only 60% increase, making the proposed solution still over 5 times better. This shows that the leap possible in case of the critical restore, thanks to the presented algorithms and using many disk drives, can be very significant.

Chapter 7

Related Work

This chapter compares the proposed solution to off-line deduplication and describes the related work in the topic of data fragmentation. The analyzed areas include: fragmentation measurement options, defragmentation algorithms and caching policies. Each of them covers the references and comparison to the algorithms presented in this thesis.

7.1 Comparison with off-line deduplication

One simple solution which satisfies some of the requirements for fighting inter-version fragmentation is already present on the market and is called off-line deduplication [62, 63, 75], described in Section 2.2.1. In its simplest form, all data from the current backup are stored continuously on disk, and the deduplication is done in the background in such a way that the blocks from the latest backup are a base for eliminating duplicates from older backups [47, 62].

As a result, the currently written stream has no fragmentation and older backups are fragmented proportionally to their age. Even though the algorithm was most probably not designed to deal with fragmentation, it is very effective for eliminating it in recent backups. However, since deduplicating a block is usually much faster than sending it over a wire and storing it on disk, off-line deduplicating systems may be slower than in-line deduplicating systems (or require more spindles and network bandwidth to avoid such problem).

The percentage of duplicates in a backup depends on the data patterns, but based on the characteristics of over 10000 systems gathered by Wallace et al. [81], we can assume the average value of deduplication ratio at a level of 10 times reduction, which results in about 90% of duplicates in an average

Aspect	Off-line dedup	In-line CBR	Off-line CBR
duplicated stream write bandwidth	low	high	high
read bandwidth	maximal	close to maximal	low after backup, later close to maximal
additional disk space	low (with staging)	low (not more than 5%)	low (with staging)
additional disk bandwidth and spindles	high (write all duplicates and remove)	low (no duplicates written except for 5% rewrites)	medium (additional reads)

Table 7.1: Comparison of defragmentation solutions.

backup. As explained in section 6.3.2, deduplication without writing the data can be 3 times faster than writing the data first and then deduplicating it in the background. Therefore, writing a backup stream with 90% of duplicates costs 300 time units with off-line deduplication and only 110 time units using a system with in-line dedup, even if such system does a dedup query for each block. As a result, using off-line dedup results in a backup window more than 170% larger. This is clearly not acceptable, as backup window usually cannot be extended much.

The idea of the context rewriting algorithm was to keep most of the defragmentation assured by off-line deduplication solution and provide the flexibility being able to fight its biggest issues described in section 2.2.1. In fact, when modifying the configuration parameters of the algorithm, all the blocks would be rewritten and all the duplicates would be eliminated in the background making both solutions very similar. On the other hand, with the border of rewritten blocks set to 5% preserving the performance and other aspects of in-line duplicate elimination, the fragmentation may be improved by a major factor.

Beyond off-line deduplication and the in-line CBR, there is at least one more option – to perform the context-based rewriting in the background, i.e. off-line, mentioned already in section 5.4.6. Such solution does not affect backup writing at all, but it needs a long time to complete reading the

fragmented data and rewriting them in the background. Additionally, a restore attempted before block rewriting is completed will still suffer from low bandwidth.

The comparison of all mentioned alternatives is presented in Table 7.1. I would like to note here that storage consumption of both off-line options can be improved by staging, i.e. by running the process of removing the duplicates (or rewriting some of them) in parallel, but little behind the process of backup writing. Staging, however, requires more resources such as CPU, available disk bandwidth and spindles.

7.2 Fragmentation measurement

Chunk Fragmentation Level (CFL) has been been introduced by Nam et al. [53] in order to visualize the fragmentation problem of a stream. Assuming that the data were stored in fixed size containers (2MB or 4MB), the idea was to divide the optimal chunk fragmentation (size of the stream divided by the container size) by the current chunk fragmentation (the actual number of containers read during restore), limiting the maximal value of achieved result to 1. The resulting number was to be proportional to the achieved performance. Unfortunately, the number did not consider the existence of read cache, which is very important when measuring restore performance and made the experiments not realistic.

The second version of this algorithm [54] did include the existence of read cache in the current chunk fragmentation calculation, but some other flaws remained. The maximal value of 1 seems to be an artificial limitation and does not reflect the real restore performance in case there is a high and well cached internal stream deduplication, which, as my experiments show, can often happen. The other limitation is actually the strong dependence on the writing algorithm (container size) together with its usage in cache eviction algorithm. Keeping whole or no container in the cache does not seem like an optimal option for the cache either, especially that usually only some blocks from the container will be necessary. On the other hand, as the LRU cache replacement policy is in general not very effective, the impact of such algorithm is rather small - the problem would be much larger if more effective cache eviction policy was used, such as the cache with forward knowledge.

Lillibridge et al. [45] propose actually a very similar indicator called "speed factor". It is also based on the number of containers, but it is defined in a bit different way as 1 divided by the number of containers read per MB. Assuming the container size the same as with CFL (4MB), the "speed factor" 1 equals CFL 0.25. When comparing both indicators, the CFL looks

Fragmentation metric	Value indicating bandwidth level with no deduplication	Correlation with bandwidth	Dependence on the backup algorithm	Level of caching used with simulation
Chunk Fragmentation Level (CFL)	1	yes (but not above 1)	yes (container)	container
Speed factor	4	yes	yes (container)	block
% of restore bandwidth with no duplicates	100%	yes	no	block

Table 7.2: Comparison of different fragmentation metrics

a bit better only because the value of 1 clearly shows the speed of the system with no deduplication and no fragmentation. On the other hand, "speed factor" is not limited in any way, showing the exact values even when the impact of internal stream deduplication is positive. Unfortunately, such feature is just theoretical as the algorithms used in the experiments did not allow the "speed factor" value of 4 (equal to CFL 1.0) and above, even with unlimited cache memory used. Some limitation in both algorithms is still strong dependence on the container size created during the backup process.

The indicator proposed by me: "% of restore bandwidth with no duplicates" is actually very similar to the ones above with some modifications (see comparison in Table 7.2). First, its name clearly presents its meaning, which makes it very intuitive to use. Second, it does not limit the results in any way, predicting the output performance very well even in cases when it is better than in systems with no deduplication. Third, it is highly independent from the writing algorithm and does not depend on the used container size, which can help in making it usable in the wide area of systems and in order to experiment with different prefetch values. Of course, it can be also easily limited to reflect the exactly same behavior like the ones with fixed container sizes. The last but not least factor is the cache eviction policy used by the simulation. My experiments showed that with no doubt it is an extremely important factor when measuring fragmentation and may have a very high impact on the achieved results.

7.3 Defragmentation algorithms

Most recently, the topic of improving read performance in storage systems with deduplication became quite popular in the published papers. The solution proposed by Lillibridge et al. [45] involves a technique called "container capping", which can be regarded as a kind of defragmentation. The solution does well in improving read bandwidth by assuring restore only from limited number of containers, but the results shown are compared only with the original algorithm designed by the author [46], which is rather poor and cause high fragmentation by design (by analyzing the results 12-44 worse restore bandwidth, when compared to a simple system with no deduplication). Unfortunately, there is no comparison with the restore bandwidth achieved with no inter-version fragmentation nor the algorithm with unlimited cache, which would be very interesting and would have made the results at least partly comparable with the ones presented in my work. Without that, the level of internal deduplication cannot be determined together with its impact on the final results, which can potentially be significant as shown in the experiments. One information we can get from the charts is that with capping set to 10 (achieving the highest restore bandwidth of all options analyzed in the article) the algorithm achieves 50-90% (assuming speed factor 4 equals 100%) of bandwidth possible in case of a system with no deduplication. This result would sound moderately well, but only if we do not consider the negative impact on the cumulative deduplication factor, which in such setup is equal to 17-50% (depending on the data set). This cost is very high and causes lower write bandwidth, which is not mentioned in the text. Compared to Lillibridge's research, none of the algorithms presented in my work modify the deduplication ratio and only one slightly decreases write bandwidth. Beside the algorithms, the study also showed the significance of the fragmentation problem on interesting long term traces (covering even 2 year period), which is something difficult to find. Unfortunately, the traces turned out not to be available for other researches, which did not allow me to compare the results directly.

Another way for assuring demanded read performance was presented by Nam et al. [54]. The basic idea here is to use Chunk Fragmentation Level [53, 54] indicator to monitor simulated read performance during write and enable selective deduplication when this level is below some defined threshold. As it was shown that CFL is a good indicator to do that, such algorithm guarantees some predefined read performance while storing data. In practice this result is achieved with moderately high cost. As selective deduplication works only part time, some places in the stream where fragmentation could be significantly improved at low cost are omitted, whereas

requiring blocks to be stored in perfect sequence makes that a lot of unnecessary duplicate blocks are stored again. Based on the above observations, and in some cases a very low backup bandwidth (even 70-90% drop while assuring CFL=0.6 for restore), I can only assume that the level of such blocks is high, as the impact of algorithm on deduplication ratio was not mentioned in the article. The algorithms presented in this work, on the other hand, does not introduce additional storage consumption and try to fix the fragmentation problem at the cost not higher than the one defined by the user. Such approach is much more efficient as I try to improve the fragmentation at the smallest possible cost. Having an option with assured performance is also possible (in an easy way: by setting current rewrite utility to some fixed value; or a more complicated way: to set it by simulating restore performance during write), but at the cost of variable write bandwidth, which may not be acceptable. Such solution would still be better than the one proposed by the author as the blocks rewritten at first would be the ones introducing the highest fragmentation.

Fu et al. proposed History-Aware Rewriting (HAR) [33], an interesting attempt to exploit historical information in order to minimize the required rewriting and maximize potential gain. HAR keeps data in 4MB containers and rewrites duplicates to ensure that at least 50% of each container read on a backup restore is useful backup data. The biggest problem with this solution is that HAR significantly degrades deduplication by block rewriting, whereas CBR does not. Moreover, problem mentioned by the authors is the out-of-order containers with blocks from different parts of the stream, natural in case of in-line deduplication systems. As the algorithm uses the global knowledge for the whole previous backup such container is considered well utilized and not rewritten whereas during restore it may be loaded into memory many times. This is because in a case of streaming access the local information is required (considering the restore algorithm and available cache size) and not the global one. In this aspect the rather small stream context size in CBR is actually an advantage leveraged by the authors in the hybrid HAR+CBR implementation. In my opinion making the CBR aware of the actual restore algorithm would actually be the best way to go here, but making the HAR to use more local than the global knowledge could also be a good solution. HAR paper actually contains a comparison with CBR, but it was simulated without internal duplicates filtering (implemented, but not described in the original CBR paper), which made CBR results rather poor in the HAR paper.

RevDedup [58] is a system which fights fragmentation by performing on-the-fly reverse deduplication. After storing such a block, the older copy is immediately located and removed. Interesting approach is also used to handle the null (zero-filled) blocks, which can be often found in virtual machine

images. In such case the server skips the disk writes and when necessary, generates the null data on-the-fly. Unfortunately, the whole system is tailored for virtual machine images with many solutions such as fixed block sizes and large segments (4MB), which are not applicable to storage systems in general. The solution succeeds in highly improving the restore bandwidth, but on the other hand, even with clever null blocks handling the system suffers from a much lower (30-65%) backup throughput when compared with the conventional deduplication, and achieves lower deduplication ratio.

Srinivasan et al. [76] describe very similar issues with fragmentation discovered in primary storage. The solution proposed by iDedup is to amortize seeks by keeping a minimum sequence length on disk by stored blocks. In this case the task is more complicated as for primary systems latency is one of the most important factors. The various results show increase in average request size and better client response time, but the difference is not significant. Also, no restore bandwidth was measured (probably due to different purpose of this system). On the other hand, the drop in deduplication ratio at a level of 30-50% seems significant even for a primary storage system.

7.4 Caching

Forward assembly area [45], the second technique proposed by Lillibridge et al., beside container capping, is aimed to help with better cache memory usage by using the backup's recipe (similar to forward knowledge) known at the restore start. In the simplest case the authors restore the full backup in M-byte slices with necessary memory allocated for the one being read called forward assembly area. To restore a single M-byte slice, they first read in the corresponding part of the recipe into the memory buffer and determine which chunks are needed to fill the required byte ranges. Each time the earliest unfilled chunk spot in the assembly area is localized, the corresponding container is restored while filling all the parts of the assembly that need chunks form that container. After the process is completed the data can be returned to the user and the assembly area can be clear in order to read the next M-byte slice.

The interesting fact is that the solution works well only on highly fragmented data (no capping or high capping levels), which in the way it is defined, can be also observed in my experiments. Unfortunately, with more reasonable capping values (10,15,20 - as defined in the paper) this makes the algorithm not really useful. The main problem here is that the whole forward area needs to have the memory reserved even though it may not be used for most of the time (1GB of forward assembly area at the cost of

1GB of RAM). This approach significantly limits the maximal possible size of forward assembly area, which as a result makes the algorithm less effective for not fragmented streams. Compared with the forward assembly area, the cache with forward knowledge presented in this work requires even as low as 1.62MB of memory for 1GB of forward knowledge and it uses all the cache very effectively only for keeping the blocks, which will actually be needed in the future. Actual difference can be seen in Figure 6.3, where the option with 512MB cache and 512MB of forward knowledge looks very similar to the 512MB forward assembly area (besides the fact that with my algorithm the reappearing block can be held in memory throughout the whole stream, while with forward assembly area the guarantees not to read it again are only for the size of the area). As a result, the user can get higher restore bandwidth with a few times smaller memory cost with the forward knowledge cache.

Fu et al. solution [33], mentioned already in previous section, introduces The Optimal Restore Cache beside History-Aware Rewriting. The algorithm is also based on the Bélády's optimal algorithm, but with full forward knowledge and 4MB containers. Such large containers reduce overhead due to keeping full knowledge, but also result in unneeded data blocks kept in the cache. In contrast, the cache proposed in this thesis uses only limited forward knowledge, but with much finer unit - 8KB blocks. This allows to keep in the cache only the data to be used soon. Also, as my experiments showed, having only limited forward knowledge is perfectly enough to implement close-to-optimal cache eviction policy. Moreover, moving to the level of blocks instead of containers reduces the memory requirements possibly even a few times. This is because of both sparse containers, with partly unused data and, even more, out-of-order containers, of which only small amount of blocks is required locally. In general, for 3 traces tested, HAR with Optimal Cache reduced the performance drop, comparing to a system with no deduplication, to between 25% and 50% with one spindle and 1GB cache; whereas the cache with limited forward knowledge and CBR are able to eliminate this drop almost completely on average with only 256MB cache (albeit for different 6 traces).

All studies of fragmentation in backup systems other than the above [42, 53, 54] simply use LRU cache to measure achieved results and verify the efficiency of proposed solution. In addition, Wallace et al. [81] performed a wide study of backup workloads in production systems reporting the hit ratio for LRU read cache when restoring final backups. On the charts showed, we can observe the impact of additional cache memory. Unfortunately, when looking at the only reasonable choice (container level caching) starting from 128MB of memory up to 32TB, most of the results look very similar and cannot be read with required precision, which makes the usefulness of such data representation very low for our purpose. Note that in case of no duplicate

stream access with 4MB container the expected stream hit ratio is 99.8% (1 read every 500 blocks of size 8KB), while 99.6% shows already two times more I/O operations therefore reducing the restore bandwidth by half. Also, in case of well cached internal stream duplicates the cache hit ratio can be above the 99.8% level.

In [6] Bélády shows the optimal cache replacement policy when having a complete sequence of block references to be used supplied by a pre-run of the program. Originally the algorithm was designed for paging, but it can be used anywhere until the single read size (from some slow device) and the smallest cache replacement size are equal. With similar assumption Cao et al. [14] performed a study of integrated prefetching and caching strategies giving four rules to be followed by every optimal solution. Unfortunately, they do not apply directly for the same reason for which the Bélády's algorithm is not optimal in case of streaming access in backup systems. Assuming the prefetch containing many blocks which are read at once and the cache eviction policy which can operate on each single block, the potential cost of reading again for each candidate for removal should be calculated. As the blocks are read in batches, this cost should be always calculated with consideration of all the blocks read in one batch and should be divided by the number of blocks actually needed. Such approach, on one hand, may allow an optimal usage of cache dedicated for data, but on the other, may require additional storage for meta information with unknown final result. As my experiments show, the cache with limited forward knowledge which uses simplified additional information works very well and in many cases actually results in a performance very close to the maximal one (achieved with unlimited cache size).

7.5 Other related work

A few papers investigated improving metadata read for faster duplicate elimination. Zhu et al. [85] describes a solution with Bloom Filter and stream-oriented metadata prefetch, whereas Lillibridge et al. [46] argues that sparse indexing (eliminating duplicates only within previously selected few large segments) is better due to smaller memory consumption. These solutions assume streaming write pattern, whereas SSD can be used for elimination of random duplicates [21, 50]. Such approach makes the fragmentation problem even harder, as more fine-grained duplicates can be detected. Additionally, none of the above techniques solves the problem of reading the fragmented data and in all cases fragmentation increases with subsequent backups. The interesting fact is that the CBR defragmentation algorithm improves the ef-

fectiveness of some former solutions as a side effect, by making the access to both data and metadata of the defragmented stream more sequential.

If we relax the requirement on defragmentation solution of not degrading deduplication effectiveness, we can try to deduplicate only within a subset of data, therefore potentially reducing fragmentation. Besides sparse indexing, such approach is possible with extreme binning [8], with large segment similarity such as in ProtectTier [3], subchunk deduplication [70], and with multi-node systems restricting dedup to one or a subset of nodes such as Pastiche [20] and DataDomain global extension [22, 28]. Unfortunately, even if we consider very few (2-3) segments of previous backups to deduplicate the current segment against, those segments may already be not sequential on disk, because they may contain also duplicates from other, older segments.

Some vendors, such as EMC, try to fight the fragmentation with time and resource consuming housekeeping processes [48, 85]. The description of this process has not been published, but one possible approach is to selectively rewrite subset of duplicates in the background, i.e. in a way similar to my CBR approach, but done off-line. More on such algorithm is given in Section 7.1. Other systems, such as HYDRAstor [23, 56], use bigger block size (64KB), which reduces the severity of the fragmentation problem, but may also lower the deduplication. However, big block size facilitates also global deduplication which in sum increases deduplication ratio. Finally, we can eliminate fragmentation by deduplication with logical objects. In early versions of EMC Centera [27], the unit of deduplication was entire file, which worked well for Centera's target market, i.e. archiving, but is not the right solution for backups, because file modifications make such dedup ineffective.

What is important, none of the above solutions mentions usage of forward knowledge which is easily accessible when it comes to backup solutions. As my experiments show, this additional information makes a significant difference when it comes to restore performance and the efficiency of cache memory used.

Chapter 8

Conclusions

This chapter contains summary of the dissertation and proposals of directions for future work.

8.1 Summary

In this work I described data fragmentation problem in backup systems with deduplication and proposed solutions for two different aspects of this issue. Additionally, I quantified the impact of different kinds of fragmentation caused by the deduplication on backup restore bandwidth with and without introduced algorithms. To support my results I performed a large number of experiments on real backup traces gathered from users.

The fragmentation problem is quite severe, and depending on each data set characteristics, it may result in restore bandwidth drop of more than 4 times (including inter-version and internal stream fragmentation) while assuming the usage of a single drive and comparing to systems with no deduplication. Even bigger drop should be expected when more spindles are being used. As my experiments were driven by six sets of real backup traces with only 7-50 backups in each set, the problem has still high potential for further growth with backups spanning many months or years. Finally, in the most popular systems with in-line deduplication, fragmentation affects the latest backup the most – the one which is also the most likely to be restored. To deal with the problem, I have proposed two algorithms addressing two different aspects of fragmentation.

The first algorithm is a dedicated cache with limited forward knowledge, and is aimed at dealing with the internal stream fragmentation caused by many duplicates present in a single backup. Thanks to the design, tailored to backup systems, the solution uses the forward knowledge already present

with each backup in order to provide effective usage of cache memory – the one dedicated for the actual data to be reused (cache in short). Moreover, depending on memory limitations and stream characteristics, the algorithm transfers most of the negative impact caused by internal stream fragmentation into a positive one. This is possible by keeping the blocks used many times in the memory, resulting often in even better performance than in systems with no deduplication, where the data is placed sequentially.

As a result, when using forward knowledge the average restore bandwidth increases in 128MB-1GB cache configurations by 62% - 88% when compared with the standard LRU approach. The effectiveness of used memory is also very well shown when comparing 128MB option with only 2GB of forward knowledge (131.25MB of memory used in total) to 1GB LRU cache. In this case, even though with almost 8 times less memory, the proposed algorithm still achieves on average 16% better restore bandwidth. Another interesting fact is that with 256MB of memory, 8GB of forward knowledge is often able to provide restore results nearly as high as with infinite forward knowledge. Moreover, in 4 out of 6 cases the results are almost identical to the ones achieved with unlimited cache size.

The second algorithm called context-based rewriting is aimed directly at the inter-version fragmentation problem caused by many backups of the same file system changing slowly in time. By rewriting not more than 5% of all blocks during backup, the algorithm improves restore bandwidth of the latest backups, while resulting in increased fragmentation for older ones, which are rarely read. Old copies of the rewritten blocks are removed in the background, for example during periodic deletion and space reclamation process, which is already required in storage systems with deduplication.

My trace-driven simulations have shown that rewriting a few selected blocks (1.58% on average) reduces maximal write performance a little (1-9%), but practically eliminates the restore speed reduction for the latest backup from 12-51% to at most 7% (avg. 2.6%) of the maximal bandwidth with LRU cache.

As both of the proposed algorithms deal with different aspects of data fragmentation, I have combined them together in order to achieve even better results. The actual numbers show 16% up to 388% higher restore bandwidth over standard LRU with an average of over 140% (both with 256MB cache, but the combined version having additional 13MB for the forward knowledge structures). The results show the algorithms to be complementary to each other, as the effect is even higher than the one which could be expected after simply adding the gain achieved by each of them (which would give an average improvement at the level of 99%). Moreover, combined algorithms with only 128MB of cache, due to effective block rewriting and efficient memory

usage, provide better results than the standard LRU, with even unlimited cache available and leaving the space for further limitations of the memory used while keeping reasonable performance. This is important as the memory showed is required per each stream being read, while in case of a critical restore there can be many of streams restored at once.

The presented algorithms perform very well when assuming only a single disk drive in the system, but even more interesting is their behavior in more real-life scenarios, where the restore of one stream is performed from many disk drives at once. The experiments show that in such environment the problem reaches another level, making the restore even more ineffective. Fortunately, the combined algorithms show their strength in such scenario as well by effectively utilizing the setup and reaching on average 5 times higher bandwidth.

Even though the problem of data fragmentation has already been known for some time [47, 48, 62, 63, 64, 83, 85], for a few years there has been no published work in the subject. The first papers trying to dig into this topic appeared in 2012 [42, 54, 76], with a few additional ones published in 2013 [45, 58]. This suggests that the subject has become more interesting for the community and potentially still requires research to definitely understand the problem and provide a solution flexible enough to be useful with different approaches. I believe that my work is a major step forward in this direction through: (1) detailed analysis of the problem with naming the three reasons of observed slowdown, (2) the proposal of two independent algorithms for solving the most severe aspects of the issue and (3) providing the results of various experiments, leaving the community with better understanding of the data fragmentation problem in backup systems with deduplication.

8.2 Future work

8.2.1 Perfect memory division during restore

As already discussed in Chapter 4.5, the fixed memory division between the actual cache and the forward knowledge is not the optimal choice when it comes to different data sets and even within the restore process of a single stream. The solution here would be the dynamic memory division. The idea is to extend the forward knowledge when the rest of the memory is not yet fully occupied by the actual data, and decrease it when there is not enough space to keep the blocks read and required in the future. The key is to constantly preserve the state where all the read blocks, which are to be found in forward knowledge, can be stored in the memory while keeping it utilized in nearly

100%.

The idea is in general quite simple, but the difficulty here is with the latency of each such operation always present in distributed systems. It will require some dedicated algorithm making the division rather smooth and dedicated communication interface between the layer providing the metadata and the cache itself. Nevertheless, such algorithm should enable even more effective cache usage than fixed memory allocation presented in this work.

8.2.2 Optimal cache memory usage

Having fixed amount of cache, the presented algorithm of evicting blocks which will not be used for the longest time in the future is not optimal as the Bélády's optimal algorithm [6] is when it comes to page replacement policy. In the latter case the page is actually treated as an independent unit which can be deleted or read separately in case of a page fault, making the case with data blocks in a backup stream different.

As within a backup neighboring blocks are very often logically connected between each other in terms of the time of being read, it would be good to utilize this observation when it comes to memory management. The idea is to look not only at the distance to the block when eviction is necessary, but actually at the cost of each operation. When doing so, it may appear that instead of keeping blocks located in the stream being restored in the N,N+1 position in the future, it is actually better to keep the ones located in N+2 and N+3 positions. Such scenario can happen when the first two are readable from the disk with only one I/O, while for the latter ones two I/Os are required.

The potential of such solution in increasing the bandwidth and/or even more effective usage of small amounts of memory is difficult to predict. On one hand, with 256MB of memory 4 out of 6 data sets in my experiments already achieve maximal possible bandwidth (similar to the one with unlimited cache size), but on the other, there is still potential when it comes to *UserDirectories* and *Mail*. Of course, in all cases it is possible that thanks to the actually optimal implementation even smaller amount of memory will provide the bandwidth very close to the maximal one, available when no memory limitations are present.

8.2.3 Variable size prefetch

One more general proposal of improving the overall restore performance is to use variable prefetch size. The idea is to modify the prefetch size based on some stream characteristics known in advance or gathered during the stream

restore. Thanks to that, for example, one can use a very small prefetch when the data requests are more or less randomly scattered over the disk or use a very large one when they are requested in the exactly sequential order. Even though the algorithm may be very useful when the order of requested data can differ a lot or can be known in advance with relation to each block placement on the disk, in case of backups systems it seems to have a limited usability. The main problem here is that it does not actually fix the potential fragmentation problem, but only tries to mask it with using smaller prefetch, which still leads to restore degradation.

8.2.4 Retention policy and deletion experiments

The interesting area which is still to be examined in terms of factors impacting the restore performance of the latest backup is the retention policy. The first idea here is to verify the frequency of backups (daily vs weekly vs monthly) and its impact on fragmentation level together with the results achieved with CBR defragmentation. The second one is to verify the impact of the number of previous versions of one backup set kept in the system (assuming fixed backup frequency within one experiment). On one hand, having more data makes it also more difficult to read the blocks which are actually needed, but on the other, simply deleting the old versions does not make the current backup change the location on the disk. Having some intelligent concatenation mechanism between the data belonging to the same streams, may be a solution here.

8.2.5 Possible extensions to CBR algorithm

When it comes to the context based rewriting algorithm, the future work may explore the following issues:

- allowing for slightly reduced deduplication when fragmentation suffers a lot
- simulating the cache algorithm with forward knowledge during write (exactly the same or similar to the one used with restore)
- applying the statistics gathered during previous backups while choosing the optimal current utility threshold
- performing CBR defragmentation once every n-th backup in order to save write bandwidth

- when considering some block as a rewrite, taking into account other streams in which it is present

Even though the current CBR algorithm performs the defragmentation very well, leaving not more than 7% to the optimal result, the above extensions may reduce this gap even further together with the cost of write bandwidth reduction with every backup.

8.2.6 Global fragmentation

The last aspect of fragmentation left for further analysis is the global fragmentation present between the different data sets placed in one backup system. In Section 3.2.3 I have already described the problem and some possible approaches to the actual solution. While committing to the maximal efficiency of data deduplication, this aspect of fragmentation seems the most complex one in terms of providing the solution for any existing system and any combination of different data sets placed together. Throughout many different usage patterns the number of global duplicates can vary a lot together with its impact on both deduplication ratio and the amount of additional fragmentation. Limiting the deduplication scope to the previous version of the same stream may be a reasonable choice in case of a system with extremely high priority of the restore performance. Some of the extensions to the CBR algorithm proposed in previous section may also help in the aspect of global fragmentation.

Glossary

Backup A copy of a file or other item made in case the original is lost or damaged (see Section 2.1.1).

Backup data set A set of data being backed up together as a single logical backup stream.

Backup policy A policy used for performing backup by a given user. Usually consists of different order and frequency of full and incremental backups (see Section 2.2.2).

Backup retention period The period within which each single backup is kept in the backup system before it is removed (see Section 2.2.2).

Backup (storage) system A system used to keep backup data usually provided by some backup application (see Section 2.1).

Backup version The logical backup stream representing some data set at a given time. If a data set is backed up every week, each week's backup will become a different version of one data set (different and separate logical backup stream, but with not many differences when compared byte by byte).

Block hash A result of hash function computed over the data within a given deduplication block. With good hash function and enough hash length (i.e. 160 bits or more) the result can be regarded as a good and unique (with extremely high probability) identifier for the block enabling easy deduplication (see Section 2.2.1).

Data fragmentation A phenomenon in which logically sequential data are not placed sequentially on the disk making the restore process expensive. In this thesis I look especially at the fragmentation caused by duplicate elimination (see Chapter 3).

Deduplication A technique to save storage space by finding identical blocks and keeping only one copy of each of them (see Section 2.2).

Deduplication block The smallest sequential part of logical backup stream which can be compared with others (see Section 2.2.1).

Deduplication ratio A proportion of bytes written to bytes actually stored. Within backup systems often in a range between 10 and 20 (see Section 2.2.2). Sometimes shown as 1:10, 1:20.

Duplicate elimination See *Deduplication*.

Full backup A backup consisting of all the data. Can be used directly for recovery.

Global fragmentation A kind of data fragmentation caused by identical blocks appearing in two or more different data set (see Section 3.2.3).

In-line deduplication The technique to deduplicate blocks during the backup process. Thanks to that only the unique blocks are stored on disk (see Section 2.2.1).

Incremental backup A backup consisting of only the data which were changed from the last backup (full or incremental). Can be used for recovery only with the latest full backup and all subsequent incremental backups.

Inter-version fragmentation A kind of data fragmentation caused by identical blocks appearing many times between two backup versions of the same data set (see Section 3.2.2).

Internal stream fragmentation A kind of data fragmentation caused by identical blocks appearing many times within a single logical backup stream (see Section 3.2.1).

Logical backup stream An ordered sequence of bytes forming a single user backup stream (usually tens or hundreds of GBs [81]).

Logical block location The address of a block in a logical backup stream (two sequential blocks in a logical backup stream will always have two sequential addresses). See difference between logical and physical block location in Sections 3.2.1 and 3.2.2.

Off-line deduplication The technique to deduplicate blocks after the backup process. All blocks are always stored on disks and later in the background the redundant blocks are removed (see Section 2.2.1).

Physical block location The address of a block on a physical device (two sequential blocks in a logical backup stream will not necessarily have sequential addresses). See difference between logical and physical block location in Sections 3.2.1 and 3.2.2.

Purpose-Built Backup Appliances A kind of a backup system that utilize software, disk arrays, and server engine(s)/nodes. These products are standalone disk systems purpose built to serve as a target for backup and include features such as deduplication, compression, encryption, remote replication, and support interface (see Section 1.1).

Recovery Time Objective (RTO) Targeted duration of time and a service level within which a business process must be restored after a disaster (or disruption) in order to avoid unacceptable consequences associated with a break in business continuity. One of the key objectives in backup systems.

Secondary storage system See *Backup system.*

Bibliography

[1] 107th Congress, United States of America. Public Law 107-204: "Sarbanes-Oxley Act of 2002". July 2002.

[2] R. Amatruda. Worldwide Purpose-Built Backup Appliance 2012-2016 Forecast and 2011 Vendor Shares. *International Data Corporation*, April 2012.

[3] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 6:1–6:14, New York, NY, USA, 2009. ACM.

[4] T. Asaro and H. Biggar. Data De-duplication and Disk-to-Disk Backup Systems: Technical and Business Considerations, July 2007. The Enterprise Strategy Group.

[5] B. Babineau and D. A. Chapa. Deduplication's Business Imperatives. December 2010.

[6] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966.

[7] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *MASCOTS*, pages 1–9. IEEE, 2009.

[8] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2009)*, Sep 2009.

[9] H. Biggar. Experiencing Data De-Duplication: Improving Efficiency and Reducing Capacity Requirements, February 2007. The Enterprise Strategy Group.

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

[11] British Library. Web Archive: preserving uk websides. http://www.webarchive.org.uk/ukwa/.

[12] A. Z. Broder. Some aplications of rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[13] M. Campbell. Backup and Inline Versus Post-Processing Deduplication. *Unitrends.com*, April 2010. http://www.unitrends.com/blog/backup-and-inline-versus-post-processing-deduplication/.

[14] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '95/PERFORMANCE '95, pages 188–197, New York, NY, USA, 1995. ACM.

[15] S. Casey Morgan. The History of Data Storage and Backup Part 4: Punched Cards. 2013. http://www.storagecraft.com/blog/history-of-data-storage-and-backup-part-4-punched-cards/.

[16] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI'06: 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[17] I. S. I. Consortium. INSIC's 2012-2022 International Magnetic Tape Storage Roadmap. May 2012.

[18] R. Cook. How to optimize your backup tape rotation strategy. *SearchDataBackup.com*, February 2009.

[19] E. Corporation. EMC Centera. Content Addressable Storage. Product Description Guide.

[20] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 285–298, New York, NY, USA, 2002. ACM.

[21] B. Debnath, S. Sengupta, and J. Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *2010 USENIX Annual Technical Conference*, June 2010.

[22] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the 9th USENIX conference on File and Storage Technologies*, FAST'11, pages 15–29, Berkeley, CA, USA, 2011. USENIX Association.

[23] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: a Scalable Secondary Storage. In *FAST'09: Proceedings of the 7th USENIX Conference on File and Storage Technologies*, pages 197–210, Berkeley, CA, USA, 2009. USENIX Association.

[24] C. Dubnicki, C. Ungureanu, and W. Kilian. FPN: A distributed hash table for commercial applications. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 120–128, Washington, DC, USA, 2004. IEEE Computer Society.

[25] D. E. Eastlake and P. E. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), September 2001.

[26] EMC Avamar: Backup and recovery with global deduplication, 2008. http://www.emc.com/avamar.

[27] EMC Centera: Content addressed storage system, January 2008. http://www.emc.com/centera.

[28] EMC Corporation: Data Domain Global Deduplication Array, 2011. http://www.datadomain.com/products/global-deduplication-array.html.

[29] EMC Corporation: DataDomain - Deduplication Storage for Backup, Archiving and Disaster Recovery, 2014. http://www.datadomain.com.

[30] Enterprise Strategy Group. 2010 Data Protection Trends. April 2010.

[31] Exagrid. http://www.exagrid.com.

[32] D. Floyer. Wikibon Data De-duplication Performance Tables. *Wikibon.org*, May 2011. http://wikibon.org/wiki/v/Wikibon_Data_Deduplication_Performance_Tables.

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

[34] J. Gantz and D. Reinsel. The Digital Universe Decade - Are You Ready? *IEEE Trans. Parallel Distrib. Syst.*, May 2010. Sponsored by EMC Corporation.

[35] J. Gantz and D. Reinsel. The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. *IEEE Trans. Parallel Distrib. Syst.*, December 2012. Sponsored by EMC Corporation.

[36] J. Gantz, D. Reinsel, C. Chute, W. Schlichting, J. McArthur, S. Minton, I. Xheneti, A. Toncheva, and A. Manfrediz. The expanding universe: A forecast of worldwide information growth through 2010. March 2007.

[37] A. S. P. Group. Identifying the Hidden Risk of Data Deduplication: How the HYDRAstor Solution Proactively Solves the Problem. 2009. White Paper WP103-3_0709, NEC Corporation of America.

[38] G. L. Heileman and W. Luo. How caching affects hashing. In C. Demetrescu, R. Sedgewick, and R. Tamassia, editors, *ALENEX/ANALCO*, pages 141–154. SIAM, 2005.

[39] HP StoreOnce Backup, 2013. http://www8.hp.com/us/en/products/data-storage/data-storage-products.html?compURI=1225909.

[40] IBM ProtecTIER Deduplication Solution. https://www-304.ibm.com/-partnerworld/wps/pub/overview/HW21Z.

[41] D. W. Jones. Punched Cards. A brief illustrated technical history. 2012. http://homepage.cs.uiowa.edu/ jones/cards/history.html.

[42] M. Kaczmarczyk, M. Barczynski, W. Kilian, and C. 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.

[43] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX conference on File and Storage Technologies*, FAST'10, pages 239–252, Berkeley, CA, USA, 2010. USENIX Association.

[44] S.-W. Lee and B. Moon. Design of flash-based dbms: an in-page logging approach. In *SIGMOD Conference*, pages 55–66, 2007.

[45] M. Lillibridge, K. Eshghi, and D. 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'13, pages 183–198, Berkeley, CA, USA, 2013. USENIX Association.

[46] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *FAST'09: Proceedings of the 7th USENIX Conference on File and Storage Technologies*, pages 111–123, 2009.

[47] J. Livens. Deduplication and restore performance. *Wikibon.org*, January 2009. http://wikibon.org/wiki/v/Deduplication_and_restore_performance.

[48] J. Livens. Defragmentation, rehydration and deduplication. *AboutRestore.com*, June 2009. http://www.aboutrestore.com/2009/06/24/defragmentation-rehydration-and-deduplication/.

[49] H. Macarthur. Data deduplication: The real benefits. *computerweekly.com*, January 2008. http://computerweekly.com/news/1289693/Data-deduplication-The-real-benefits.

[50] D. Meister and A. Brinkmann. dedupv1: Improving Deduplication Throughput using Solid State Drives (SSD). In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, May 2010.

[51] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 174–187, New York, NY, USA, 2001. ACM.

[52] A. Muthitacharoen, B. Chen, and D. Mazires. A low-bandwidth network file system. In *In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01*, pages 174–187, New York, NY, USA, 2001. ACM.

[53] Y. Nam, G. Lu, N. Park, W. Xiao, and D. H. C. Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In P. Thulasiraman, L. T. Yang, Q. Pan, X. Liu,

Y.-C. Chen, Y.-P. Huang, L.-H. Chang, C.-L. Hung, C.-R. Lee, J. Y. Shi, and Y. Zhang, editors, *HPCC*, pages 581–586. IEEE, 2011.

[54] Y. J. Nam, D. Park, and D. 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.

[55] P. Nath, B. Urgaonkar, and A. Sivasubramaniam. Evaluating the usefulness of content addressable storage for high-performance data intensive applications. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 35–44, New York, NY, USA, 2008. ACM.

[56] NEC Corporation. HYDRAstor Grid Storage System, 2008. http://www.hydrastor.com.

[57] NEC HYDRAstor HS8-4000 Specification, 2013. http://www.necam.com/HYDRAstor/doc.cfm?t=HS8-4000.

[58] C.-H. Ng and P. P. C. Lee. RevDedup: A Reverse Deduplication Storage System Optimized for Reads to Latest Backups. *CoRR*, abs/1302.0621, 2013.

[59] B. Panzer-Steindel. Technology, Market and Cost Trends 2012. May 2012. CTO CERN/IT.

[60] E. Parliament. Directive 2006/24/EC "On the retention of data generated or processed in connection with the provision of publicly available electronic communications services or of public communication networks". March 2006.

[61] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

[62] W. C. Preston. The Rehydration Myth. *BackupCentral.com*, June 2009. http://www.backupcentral.com/mr-backup-blog-mainmenu-47/-13-mr-backup-blog/247-rehydration-myth.html/.

[63] W. C. Preston. Restoring deduped data in deduplication systems. *SearchDataBackup.com*, April 2010. http://searchdatabackup.techtarget.com/feature/Restoring-deduped-data-in-deduplication-systems.

[64] W. C. Preston. Solving common data deduplication system problems. *SearchDataBackup.com*, November 2010. http://searchdatabackup.techtarget.com/feature/Solving-common-data-deduplication-system-problems.

[65] W. C. Preston. Target deduplication appliance performance comparison. *BackupCentral.com*, October 2010. http://www.backupcentral.com/-mr-backup-blog-mainmenu-47/13-mr-backup-blog/348-target-deduplication-appliance-performance-comparison.html.

[66] Quantum Corporation: DXi Deduplication Solution, 2011. http://www.quantum.com.

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

[68] M. Rabin. Fingerprinting by random polynomials. Technical report, Center for Research in Computing Technology, Harvard University, New York, NY, USA, 1981.

[69] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 143–156, Berkeley, CA, USA, 2008. USENIX Association.

[70] B. Romanski, L. Heldt, W. Kilian, K. Lichota, and C. Dubnicki. Anchor-driven subchunk deduplication. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 16:1–16:13, New York, NY, USA, 2011. ACM.

[71] M. Rouse. Cloud backup. *searchdatabackup.techtarget.com*, February 2013. http://searchdatabackup.techtarget.com/definition/cloud-backup.

[72] Seagate Corporation. Common enterprise disk specification (based on Seagate Constellation ES.3 4TB, model 2012).

http://www.seagate.com/www-content/product-content/constellation-fam/constellation-es/constellation-es-3/en-us/docs/constellation-es-3-data-sheet-ds1769-1-1210us.pdf.

[73] Seagate Corporation. Seagate Desktop HDD 4TB, model 2013 - disk characteristics. http://www.seagate.com/www-content/product-content/barracuda-fam/desktop-hdd/barracuda-7200-14/en-gb/docs/-desktop-hdd-data-sheet-ds1770-1-1212gb.pdf.

[74] Seagate Corporation. World's fastest 6TB hard disk drive announcement — the Seagate Enterprise Capacity 3.5 HDD v4, April 2014. http://www.seagate.com/about/newsroom/press-releases/Seagate-ships-worlds-fastest-6TB-drive-enterprise-capacity-pr-master.

[75] SEPATON Scalable Data Deduplication Solutions. http://sepaton.com/solutions/data-deduplication.

[76] K. Srinivasan, T. Bisson, G. Goodson, and K. 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.

[77] P. Strzelczak, E. Adamczyk, U. Herman-Izycka, J. Sakowicz, L. Slusarczyk, J. Wrona, and C. Dubnicki. Concurrent deletion in a distributed content-addressable storage system with global deduplication. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 161–174, Berkeley, CA, USA, 2013. USENIX Association.

[78] Symantec NetBackup Appliances. http://www.symantec.com/backup-appliance.

[79] Ultrium LTO official website, 2013. www.lto.org.

[80] C. Ungureanu, A. Aranya, S. Gokhale, S. Rago, B. Atkin, A. Bohra, C. Dubnicki, and G. Calkowski. Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system. In *FAST '10*, pages 225–239, 2010.

[81] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. 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.

[82] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 328–338, London, UK, 2002.

[83] L. Whitehouse. Restoring deduped data. *searchdatabackup.techtarget.com*, August 2008. http://searchdatabackup.techtarget.com/tip/Restoring-deduped-data.

[84] S. Yin, P. Pucheral, and X. Meng. Pbfilter: indexing flash-resident data through partitioned summaries. In *CIKM*, pages 1333–1334, 2008.

[85] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

List of Figures

List of Tables