Throughput: A Key Performance Measure of Content-Defined Chunking Algorithms

Conference Paper · June 2016

DOI: 10.1109/ICDCSW.2016.32

3 authors:

Bertil Chapuis
University of Lausanne
16 PUBLICATIONS **10** CITATIONS
SEE PROFILE

Benot Garbinato
University of Lausanne
130 PUBLICATIONS **1,154** CITATIONS
SEE PROFILE

Periklis Andritsos
University of Toronto
131 PUBLICATIONS **1,113** CITATIONS
SEE PROFILE

Throughput: A key performance measure of Content-Defined Chunking Algorithms

Bertil Chapuis Benoît Garbinato Periklis Andritsos
Université de Lausanne
{bertil.chapuis,benoit.garbinato,periklis.andritsos}@unil.ch

***Abstract*—Data deduplication techniques are often used by cloud storage systems to reduce network bandwidth and storage requirements. As a consequence, the current research literature tends to focus most of its algorithmic efforts on improving the *Duplicate Elimination Ratio* (DER), which reflects the compression achieved using a given algorithm. Yet, the importance of this indicator tends to be overestimated, while another key indicator, namely *throughput*, tends to be underestimated. To substantiate this claim, we reimplement a selection of popular Content-Defined Chunking algorithms (CDC) and perform a detailed performance analysis. On this basis, we show that the gain brought by algorithms that are aggressively focusing on DER often come at a significant cost in terms of throughput. As a consequence, we advocate for future optimizations taking throughput into account and for making balanced tradeoffs between DER and throughput.**

***Index Terms*—content-defined chunking; duplicate elimination ratio; rolling hash function; performance; throughput**

I. Introduction

In recent years, we have witnessed a rapid shift from desktop computers to mobile and pervasive devices. That is, people nowadays tend to be using multiple devices to access their data, which they expect to be available and consistent throughout all their computing devices. In addition, users also expect to be able to have access to all past versions of a given piece of data, e.g., whenever editing some file, they want to be able to come back to whatever intermediate version of the file they have created.

Faced with such requirements, cloud storage services usually have to deal with redundant and superfluous copies of data. Some files can be identical or share a significant part of their content. In this context, data deduplication techniques aim at removing redundancies in order to reduce storage requirements and network bandwidth. Deduplication can occur at the level of whole files or at the level of their subparts also called chunks. When dealing with chunks, file descriptors containing chunk references become necessary to describe the original data. As a rule of thumb, the smaller the chunks, the greater the deduplication. However, with small chunks come large file descriptors and a loss of locality in the data since chunks can be stored at completely different places. Furthermore, as for compression algorithms, deduplication algorithms can incur space-time tradeoffs in order to improve the efficiency of the deduplication.

Interestingly, there exists a gap between the research literature on the deduplication algorithms and what is commonly used in practice. On one hand, most of the research literature aggressively focuses on the efficiency of data deduplication algorithms and trades computing power for storage space [3], [4], [6], although many devices remotely accessing data tend to have limited computing resources (and battery life). On the other hand, most popular cloud storage services, such as Dropbox, rely on rather simple chunking algorithms [2]. In addition, in [7], Meyer et al. recently demonstrated that, in the context of snapshots taken over 857 desktop computers during four weeks, whole file deduplication achieves approximately three quarters of the space savings of chunk level deduplication. All these facts are questioning the gain brought by the most aggressive chunking techniques in the context of cloud storage services.

In this workshop paper, we try to explore this gap by investigating the tradeoff between deduplication efficiency and another key performance measure, namely *throughput*, that measures the number of bytes processed per second. We show that the impact of optimizations focusing on deduplication efficiency is rarely put in perspective with throughput, suggesting that the importance of this metric tends to be largely underestimated in current research.

In Section II, we start by describing the algorithmic basis of most content-defined chunking algorithms found in the literature, which also serves as basis of all the chunking algorithms we implemented for this paper. These algorithms are then described and discussed in Section III. The main findings of this research are presented in Section IV, which discusses the performance of those algorithms and their respective impact on DER and on throughput. In particular, we focus on the performance of rolling hash functions, which are at the core of most content-defined chunking algorithms. In this section, we also sketch a simple optimization to reduce the negative effect of some of these algorithms on throughput. Finally, Section V puts our findings into perspective and discusses possible future work.

II. Content-Defined Chunking

Before delving into the details of our experimental study, we will describe the mainstream chunking algorithm. We should mention that, when measuring the efficiency of chunking algorithms, the Duplicate Elimination Ratio (DER) is a key indicator [4], [6]. It is calculated by dividing the size of the input dataset by the size of the chunks after deduplication. In other words, it reflects the compression achieved by using

/16 $31.00 © 2016 IEEE
DOI 10.1109/ICDCSW.2016.32

a given deduplication technique. In the literature, algorithmic improvements are usually justified by highlighting their effects on this ratio.

If hashing is applied to whole files, two files that differ in just one byte will still have different hash signatures. In order to mitigate this issue, a trivial approach consists in splitting files into fixed-size chunks (FSC). The main advantage of this technique lies in the fact that it offers a very high throughput and is easy to implement. While being successfully used by Dropbox, this approach comes with a major drawback: when bytes are added at the beginning of a file, all the following chunk boundaries are shifted [2], i.e., all chunks after the addition will have different hash sums and will become obsolete for that file.

Because of the aforementioned FSC limitation, a much more efficient family of algorithms for data deduplication, known as Content-Defined Chunking (CDC), has been proposed. In absence of further refinements, this algorithm is often referred to as Basic Sliding Window (BSW), and is shown in Algorithm 1. BSW takes as input a sequence of bytes, a window size, a divisior and a boundary parameter. These algorithms heavily rely on rolling hash functions such as the Rabin-Karp algorithm [10]. Given a window of bytes, rolling hash functions take into account incoming and outgoing bytes. As illustrated in Figure 1, the idea is to create a fixed-size sliding window (*windowSize* in Algorithm 1), depicted by a

Algorithm 1 Basic Sliding Window Algorithm

```
1:  procedure BSW($bytes, windowSize, divisor, boundary$)
2:      $boundaries \leftarrow \varnothing$
3:      $position \leftarrow 0$
4:      $hash \leftarrow 0$
5:      while $position < length(bytes)$ do
6:          $byte \leftarrow bytes[position]$
7:          if $position < windowSize$ then
8:              $hash \leftarrow updateHash(hash, byte)$
9:          else
10:             $hash \leftarrow rollHash(hash, byte)$
11:         if $position \geq windowSize - 1ba$ and
             $hash \bmod divisor = boudary$ then
12:             $boundaries \leftarrow boundaries \cup \{position\}$
13:         $position \leftarrow position + 1$
14:     return $boundaries$
```

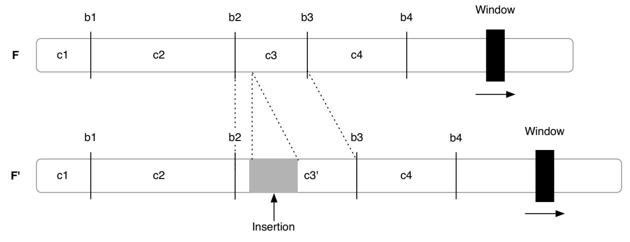

Fig. 1. Illustration of Basic Sliding Window (BSW)

black box. The sliding window slides for the whole length of the byte sequence (line 5 in Algorithm 1). If it has not filled a sequence of bytes equal to the *windowSize* parameter (line 7 in Algorithm 1), the value of the hash function is updated (line 8 in Algorithm 1). If the sliding window exceeds the *windowSize* (line 9 in Algorithm 1), then a rolling hash function is employed to account for outgoing bytes (line 10 in Algorithm 1). When the sliding window fills an area equal or greater to the fixed *windowSize*, we check for new content-based boundaries (lines 11 and 12 in Algorithm 1). Now, if we repeat the same process on some other byte sequence F', a modified version of F, we see that unlike FSC, CDC algorithms can identify chunk boundaries that resist to additions and modifications anywhere in the byte sequence. In this paper, we focus on a subset of CDC algorithms, which are basically variants of the BSW algorithm presented above, while using FSC as a baseline for our evaluation.

III. Related Work

Having as a basis the BSW algorithm that we just described in the previous section, several optimizations have been proposed. Muthitacharoen et al, [8], describe a variant of the BSW algorithm that uses Two Thresholds (TT). Instead of waiting indefinitely for a boundary match, this algorithm defines a threshold for the maximal size of chunks. Furthermore, a minimal threshold is also set and the chunks that are smaller than this threshold are merged. This optimization positively impact DER and comes with the significant advantage that the size of the chunks will be predictable.

In [3], Eshghi et al. describe a variation over the BSW algorithm that use Two Thresholds and Two Divisors (TTTD) when looking for content-defined boundaries. Since BSW uses a single divisor, the second divisor introduced by TTTD acts as a backup. In practice, it is used for avoiding arbitrary cuts when the maximal threshold is reached. In other words, even when the threshold is reached, a backup boundary, also defined on the basis of the content, will be used.

More recently, a new family of chunking algorithms based on CDC has been proposed, which scan the input data several times in order to improve efficiency based on DER. This is for instance the case of the Bimodal Content-Defined Chunking algorithm by E. Kruus et al. [4] and of the Frequency-Based Chunking algorithm by G. Lu et al. [6]. In these algorithms, while the efficiency in terms of DER is indeed improved thanks to multiple passes across files, these improvements come to the detriment of throughput.

In [1], Bobbarjung et al. propose an optimization of the Basic Sliding Window algorithm that uses a bit mask instead of a divisor. The underlying idea behind this optimization lies in the replacement of the modulo operations with a bit mask. A strong constraint comes with this optimization however, since modulo operations are only truly equivalent to bit masks when divisors are powers of two and hash sums are positive integers. As a consequence, the matches found using bit masks will not be exactly the same as the one found using modulo operations, but the probability of the match will remain the same. This optimization has no impact on the DER ratio but improves the throughput of the chunking algorithm. However, apart from [1], this last metric is often overlooked in the literature.

IV. Performance Evaluation

The aim of this study is to analyse two facets of deduplication algorithms: Duplicate Elimination Ratio (DER) and throughput. As mentioned in the previous section, the first metric reflects the compression achieved by a chunking algorithm. As the name suggests, DER is relative to the redundancies in the input dataset. The results presented in [7], suggest that in practice datasets are less redundant than most experimental datasets. In addition, a significant part of the redundancies can be captured with trivial deduplication algorithms. We define the second metric, throughput, as the amount of bytes per second that can pass through the algorithm. To evaluate performance in terms of DER and throughput, we used the following three different datasets.

- The first dataset is used to measure DER and consists in the source code of 20 stable versions of the linux kernel, from v3.0 to v3.19, that represents 9.5GB of data.[1] Since releases share most of their source code, we expect to find a lot of redundancies in this dataset and to reach a high compression ratio.
- The second dataset is used to measure DER and consists in the English dump of Wikipedia that accounts for 49GB of data once decompressed.[2] Since wikipedia is well curated, we expect to find very few redundancies in this dataset and to reach a low compression ratio.
- The third dataset is used to measure throughput and consists in a sequences of 100MB of random data. Regarding throughput, the nature of the input dataset has no impact since this measure only reflects the speed at which data is consumed. The same operations are always performed by the algorithms on the incoming and outgoing byte in order to find boundary breakpoints. For this reason, random data do not affect the results of the experiment and gives the same results as real world data. Furthermore, since we don't want to measure the throughput of the hard-drive, we load the dataset in-memory. This allows us to only measure the throughput of the chunking algorithm.

In order to perform our experiments, we implemented various CDC algorithms using Scala and the non-blocking I/O API provided by Java.[3] When measuring DER, we simply wrote chunks and metadata directly on the hard drive and measured the size of the output data. Regarding throughput, our performance benchmarks rely on JMH.[4] Finally, we performed our experiments using a Dell Power Edge T110 II with an Intel Xeon CPU clocked at 3.50GHz and 16GB of RAM.

A deeper understanding of the underlying concept of CDC algorithms is necessary to understand what impacts throughput. Rolling Hash Functions (RHF) are at the core of most CDC algorithms. So in the next section, we first analyze the properties of RHFs. Then, on this basis, we analyze deduplication algorithms in more details and measure their DER as well as their throughput, in order to analyze how the two metrics impact one another.

A. Performance of Rolling Hash Functions

Several Rolling Hash Functions (RHF) are mentioned in the literature and may constitute good candidates for CDC algorithms [11], [5]. We ported the following RHF in Scala: a Randomized Rabin-Karp hash function, a Cyclic Polynomial hash function and the custom Adler32 hash function used by RSync simply referred to as RSync hereafter.

Configuration. In [9], Policroniades and Pratt suggest that the size of the rolling hash windows does not have a significant impact on DER, so we followed their recommendation of a 48 byte window. Our own performance analysis shows that this parameter does not significantly impact the throughput of the RHF when the window remains small. Since the Randomized Rabin-Karp and the Cyclic Polynomials functions require lists of random numbers, which are equal in size to the targeted vocabulary (bytes or characters), we generated pseudo random series of 256 integers that target the byte vocabulary. The RSync hash function has an offset parameter that we initiated to 0 for our experiments.

Hash Distribution. In order to compute the distribution of the RHFs, we generated hash signatures using random

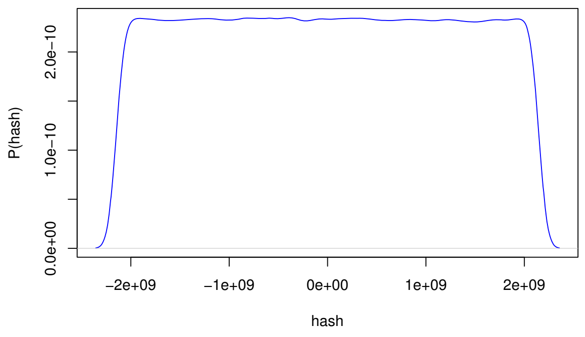

Fig. 2. Rabin-Karp Hash Distribution

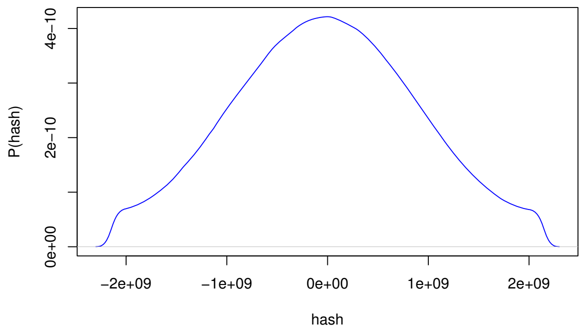

Fig. 3. RSync Hash Distribution

[1]https://github.com/torvalds/linux
[2]http://burnbit.com/download/339061/enwiki_20150205_pages_articles_xml_bz2
[3]We intent to release under the Apache Software Foundation License.
[4]http://openjdk.java.net/projects/code-tools/jmh/

data. Figure 2 shows the uniform distribution of the Rabin-Karp hash function. The Cyclic Polynomial function has a very similar uniform distribution. The RSync hash function, is sometimes mentioned for having very good performance. Unfortunately, as shown in Figure 3, this function comes with a near-Gaussian distribution. Having hash signatures that occur with different probabilities has a direct impact on the frequency at which CDC algorithms detect boundaries. As a result, the size of the chunks will be less predictable. By consequence, if we consider current CDC algorithms, this RHF should be used with caution or even disqualified. However, we think that a smart usage of this property could leave room for optimization by replacing some costly operations in CDC algorithms with inexpensive ones. For example, the costly modulo operation used in Algorithm 1 to find boundaries could be replaced with a hash that comes out with a given probability. In other words, instead of acting on the divisor to alter the probability of finding a boundary, choosing a specific boundary on the Gaussian distribution of the RSync function could lead to the same result without the division. We intend to further explore this idea in our future work.

Throughput. In terms of throughput, both the Randomized Rabin-Karp and the RSync functions perform well. On our infrastructure, we measured fairly similar throughputs of 307MB/s for the Rabin-Karp function and 293MB/s for the RSync function. In our context the Cyclic Polynomial Hash function was a little slower with a throughput of 249MB/s. These numbers tend to confirm that the Rabin-Karp hash function is a natural choice for CDC algorithms.

B. *Performance of Chunking Algorithms*

Regarding deduplication algorithms, we reimplemented and compared the following algorithms: Fixed-Sized Chunking (FSC), Basic Sliding Window (BSW) [3], Two Thresholds (TT) [8], [9], Two Divisors (TD) [3], Two Thresholds Two Divisors (TTTD) [3]. We also implemented optimized versions of these algorithms (prefixed with O) which use bit masks instead of divisors as suggested in [1]. To our knowledge, the effect of this optimization on the DER and more interestingly on the throughput of CDC algorithms has not been clearly measured in the literature. Additionally, we did not find studies comparing the TT with the TTTD algorithms.

Configuration. In [3], Eshghi et al. use hill climbing to discover the best configuration parameters for CDC algorithms. We base our configurations on these settings and aim at having an average chunk size of 1KB. Figure 4 summarizes the parameters we provided to the different algorithms. In this configuration, D stands for the primary divisor, D' for the secondary divisor, $Tmin$ for the minimal threshold and $Tmax$ for the maximal threshold. Some algorithms do not use all these parameters.

Chunk Size Distribution. In order to measure the chunk size distribution, we ran our algorithm implementations on 100Mb of random data and collected the sizes of the resulting chunks. Figure 5 highlights the chunk size distribution of the BSW algorithm. Without thresholds, such an algorithm may have to deal with corner cases, where no boundaries are found and chunks become very large. Such cases are not ideal since more memory may be required during the chunking process and the storage infrastructure may have to deal with an unpredictable chunk size. Figure 6 highlights the chunk size distribution of the TT and the TTTD chunking algorithms. It is interesting to see the effect of the second divisor introduced by the TTTD chunking algorithm. All the chunks that are reaching the upper threshold with the TT algorithm are redistributed into smaller chunks by the backup divisor of TTTD. As we will show in the following paragraph the second divisor significantly impacts

Algorithm	D	D’	Tmin	Tmax
FS	-	-	-	1024
BSW	1000	-	-	inf.
OBSW	1024	-	-	inf.
TT	1000	-	460	2800
OTT	1024	-	460	2800
TTTD	540	270	460	2800
OTTTD	512	256	460	2800

Fig. 4. CDC Configuration Parameters

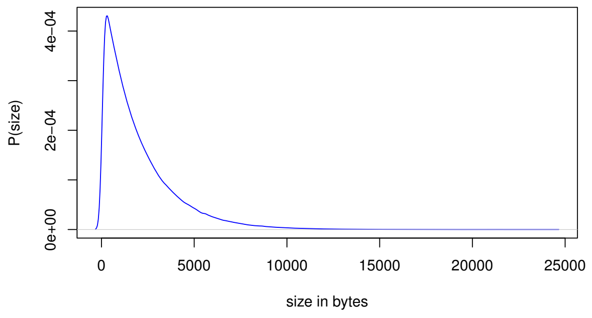

Fig. 5. BSW Chunk Size Distribution

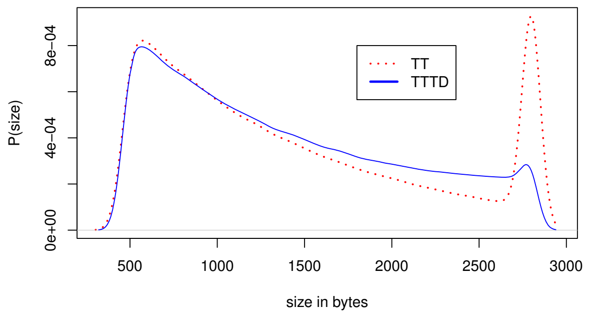

Fig. 6. TT and TTTD Chunk Size Distribution

the throughput while the DER of both algorithms remains very close. In consequence, adding uniformity to the size of the chunks is the main advantage brought by TTTD over TT.

Duplicate Elimination Ratio. As already discussed, the Duplicate Elimination Ratio (DER) is the widely used metric to evaluate chunking algorithms [6], [4] and is calculated by dividing the size of the input by the size of the chunks after deduplication. For the sake of clarity, our figures display the DER as labels below the histogram bars. The Y-axis uses percentages that relate to the size of the input dataset. This allows to easily stack and visualize the overhead introduced by the metadata and puts DER in perspective with the size of the input data. Regarding the Linux dataset, since it is composed of several major versions of the Linux kernel, we expected a high deduplication ratio. In practice, as shown in Figure 7, such dataset really justifies the usage of CDC algorithms in backup systems, since the size of the output is smaller by nearly one order of magnitude. When the dataset is versioned and contains a lot of redundancies, it really make sense to prefer CDC over FSC or whole file deduplication. When comparing CDC algorithms in terms of their DER, we notice little improvement when comparing the successive optimizations of the BSW algorithm, namely TT and TTTD. However, as depicted here, when the size of the input dataset is taken into account, it is surprising to note how little this gain is. This allows us to question the meaning of the DER metric and to recommend more meaning full metrics which put the gain in perspective with the size of the input dataset. Regarding the Wikipedia dataset, since it is a snapshot that comes from a normalized database, we expected very few redundancies. This was confirmed in practice, since we only identified approximately 1GB of redundant data. In fact, Figure 8 shows that, in such dataset, deduplication even comes with a small penalty since file descriptors also need to be stored. This confirms that the gain brought by optimisations focusing only on DER tend to come with little benefits.

Throughput. We now focus on the throughput metrics to highlight the cost of some optimizations that focus on DER. The first thing to notice is that fixed-size chunking outperforms all CDC algorithms with a throughput of more than 2GB/s. As shown in Figure 9, with a highest throughput of approximately 130MB/s, CDC algorithms are at best 15 times worse than fixed-size chunking when using bit mask optimizations. Interestingly, the BSW algorithm does not perform better than its DER optimized counterparts in terms of throughput. This allows us to identify the positive effects of introducing minimal and maximal thresholds in CDC algorithms. The lower threshold eliminates a range of bytes from the computation, while the upper threshold reinforces this effect by increasing the number of chunks. However, the introduction of a second divisor by TTTD over TT comes at a significant cost which can be mitigated by using the bit mask optimization. As demonstrated with OTTTD and OTT, using bit masks allows to avoid costly division operations. In consequence, it is useful to search for configuration parameters that are slightly distanced

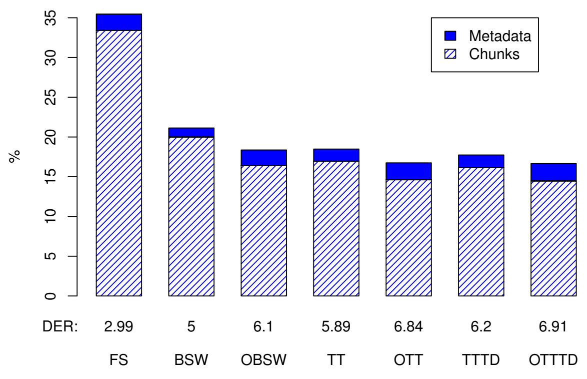

Fig. 7. Linux: Chunking Algorithms DER

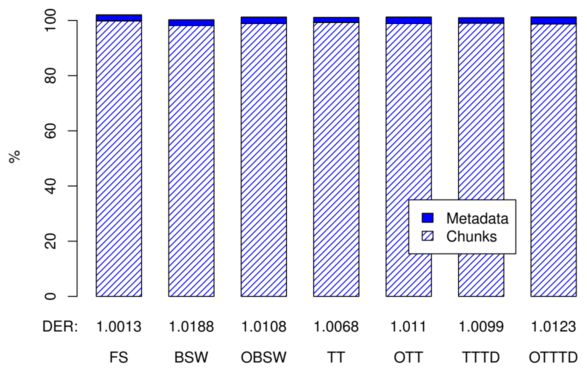

Fig. 8. Wikipedia: Chunking Algorithms DER

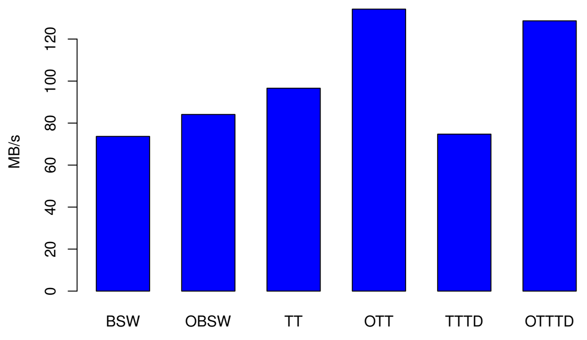

Fig. 9. Chunking Algorithms Throughput

from the ideal configuration settings described by Eshghi et al in [3] but that satisfy the constraints brought by bit masks.

These results concur with the conclusion of [7]. They clearly demonstrate that DER measures are relative to the input dataset and should be used with a lot of caution. In practice, datasets contain different kinds of data, which highly mitigate the benefits brought by CDC algorithms. The throughput measures presented here probably highlight the reason why some cloud storage services are choosing trivial deduplication schemes. The cost of looking for content-defined chunk boundaries is very high since each time a byte is consumed, it produces a hash sum which is used to identify chunk boundaries. Consequently, CDC algorithms introduce a lot of computation and if the deduplication process occurs on the end-user device, this cost will have a significant impact on its autonomy and battery life. In conclusion, it is legitimate to question costly DER optimizations and more importance should be given to the throughput of CDC algorithms.

V. Conclusion

The results presented in this paper tend to confirm the findings of Meyer et al. described in [7]. A significant part of the benefits brought by content-defined chunking algorithms can be obtained with trivial deduplication techniques such as fixed-size chunking. The results also demonstrate a new fact: such optimizations introduce a significant computational overhead which comes to the detriment of the throughput. As a consequence, the benefits brought by algorithms that focus on duplicate elimination ratio to the detriment of the throughput can be put to the question. We show that bit mask optimizations have a positive impact on the throughput but this incidence remains quite limited if we consider the throughput of a trivial fixed-size chunking algorithm. As a consequence, we believe that more importance should be given to the computational efficiency of content-defined chunking algorithms.

Today, these facts limit the attractiveness of content-defined chunking algorithms for cloud storage solutions, especially when the deduplication process occurs on end-user devices, which usually have limited capacities. However, if we consider that cloud storage services will soon give access to all past versions of a given piece of data, efficient deduplication algorithms in term of duplicate elimination ratio and throughput are necessary. Such algorithms will allow to store all changes on the data as they occur and at a reasonable cost.

The algorithms we reimplemented constitute a good basis for analyzing and improving the throughput of content-defined chunking algorithms. Our goal will now consist in finding efficient ways for improving the performance of this category of algorithms and bring it closer to the throughput of fixed-size chunking without impacting the duplicate elimination ratio. Making these algorithms more efficient will probably make them more attractive for cloud storage services and more suitable for devices with limited capacities.

References

[1] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki. Improving duplicate elimination in storage systems. *ACM Transactions on Storage (TOS)*, 2(4):424–448, 2006.

[2] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 481–494. ACM, 2012.

[3] K. Eshghi and H. K. Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30:2005, 2005.

[4] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content defined chunking for backup streams. In *FAST*, pages 239–252, 2010.

[5] D. Lemire and O. Kaser. Recursive n-gram hashing is pairwise independent, at best. *Computer Speech & Language*, 24(4):698–710, 2010.

[6] G. Lu, Y. Jin, and D. H. Du. Frequency based chunking for data de-duplication. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 287–296. IEEE, 2010.

[7] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7(4):14, 2012.

[8] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 174–187. ACM, 2001.

[9] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *USENIX Annual Technical Conference, General Track*, pages 73–86, 2004.

[10] M. O. Rabin. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[11] A. Tridgell. *Efficient algorithms for sorting and synchronization*. Australian National University Canberra, 1999.