Dynamic Determination of Variable Sizes of Chunks in a Deduplication System*

Michael Hirsch[a], Shmuel T. Klein[b], Dana Shapira[c], Yair Toaff[a]

[a] *Toga Networks, Hod Hasharon, Israel*
mikizvi@gmail.com, Yair.Toaff@gmail.com

[b] *Department of Computer Science, Bar Ilan University, Ramat Gan, Israel*
tomi@cs.biu.ac.il

[c] *Computer Science Department, Ariel University, Israel*
shapird@ariel.ac.il

Abstract

Deduplication is a special case of data compression in which repeated chunks of data are stored only once. The input data is cut into chunks and a cryptographically strong hash value of each (different) chunk is stored. To restrict the influence of small inserts and deletes to local perturbations, the chunk boundaries are usually defined in a data dependent way, which implies that the chunks are of variable length. Usually, the chunk sizes may spread over a large range, which might have a negative impact on the storage performance. This can be dealt with by imposing artificial lower and upper bounds. This paper proposes an alternative by which the chunk size distribution is controlled in a natural way. Some analytical and experimental results are given.

Keywords: Deduplication, compression, chunk size

1. Introduction

Deduplication is a lossless data compression technique that is somewhat similar to the classical first method of Lempel and Ziv [14] known as LZ1 or LZ77: the size of an input file is reduced by trying to replace a substring

*This is an extended version of a paper that has been presented at the Prague Stringology Conference (PSC'15) in 2015, and appeared in its Proceedings, 78–89.

starting at the current position, by a pointer to some earlier occurrence, if it exists. The pointers are of the form (offset, len), where offset is the number of characters in the original text one has to go backwards to find the beginning of the substring to be replaced, and len is its length. Compression is achieved because the pointers need less space for their representation than the strings they substitute. Here is an example, taken from Friedrich Schiller's poem *Das Lied von der Glocke*:

`von-der-Stirne-heiß-rinnen-muß-der-Schweiß-` $\cdots$

could be replaced by

`von-der-Stirne-heiß-rin`(11,2)(23,2)`mu`(11,2)(27,5)`chw`(23,4) $\cdots$,

Deduplication takes this basic idea a step further to a larger scale, in which the elements to be replaced are not just short substrings but entire blocks of data, referred to as *chunks*. Obviously, the handling of larger text blocks has also disadvantages since the probability of finding identical blocks decreases, and the application area of deduplication is therefore quite different from that of standard lossless compression. For instance, purely random data cannot be compressed at all. There are, however, applications in which even such incompressible files, if they appear more than once, may yield some savings. An example could be a large backup system, in which the entire available electronic storage of some corporation has to be copied and saved at regular time intervals for security reasons and to prevent the loss of data. The special feature of such backup data is that only a small fraction of it differs from the previously stored backup. Deduplication handles also such files, by storing duplicates only once. The challenge is, of course, to locate as much of the duplicated data as possible.

One of the approaches to build a deduplication system is by means of *Content Addressable Storage*. Partition the input database, which is often called the *repository*, into fixed or variable sized chunks and apply a cryptographically strong hash function h, e.g., SHA-1 or MD5, on each of these input chunks. That is, if C_i and C_j are different, the probability for $h(C_i) = h(C_j)$ is so low that one can safely ignore it. Equal hash values may thus be assumed to imply identical chunks. The different hash values, along with the addresses of the corresponding chunks, are stored in a fast-to-access data structure, like a hash table or a B-Tree [9, 12], as depicted in Figure 1 showing an example scenario of a deduplication system. The chunk C_4 starts at address 420 and applying the hash h on C_4 yields the value $a = 36844$; the address 420 is therefore stored in the hash table at entry a.

When a fresh copy of the data is given, e.g., for a weekly or even daily backup, the new data, often called a *version*, is also partitioned into similar chunks. The hash value of each of these new chunks is searched for in the table, and if it is found, one may conclude that the new chunk is an exact copy of a previous one, so all one needs to store is a pointer to the earlier occurrence. For our example, if for a new version chunk D we also get $h(D) = a$, the address 420 is retrieved from the hash table, and we conclude that D is identical to the chunk starting at 420, which is C_4. There are also approaches to deduplication which relax the request for identical chunks and replace one chunk by another even if they are only *similar*, adding of course also the (few) differences to enable the recovery of the original data [1, 2, 10, 11].

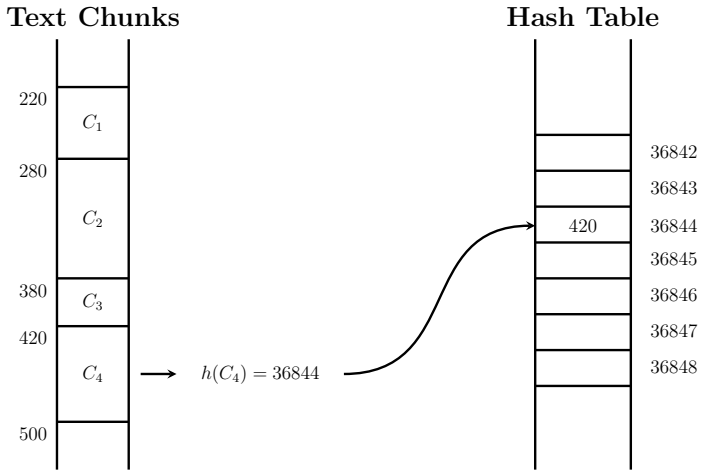

FIGURE 1: *Schematic view of a deduplication system.*

A simple approach would be to choose the chunk size as a constant. However, this results in a high sensitivity to small insertions and deletions. Indeed, even a single added or omitted byte could shift all subsequent chunk boundaries accordingly, invalidating the hash approach. The solution is to let the boundary of the chunk to be dependent on the content itself, which implies variable length chunks.

A general paradigm for cutting the data string consisting of a sequence of bytes $s_1 s_2 \cdots$ into pieces is to use a so-called *rolling hash*, which calculates a hash value for any consecutive sequence of k bytes. We shall refer to such a sequence as a *seed*. Starting with the byte indexed k, each byte can be considered as the last of a seed. The condition for deciding whether the last

byte of the seed, s_j, with $j \geq k$, will also be the last byte of the current chunk, is that

$$h(s_{j-k+1}s_{j-k+2}\cdots s_j) = C,$$

where h is the hash function and C is some constant chosen from the set of values $\{h(i)\}$. Since hash functions are supposed to return uniformly distributed values, the probability of this occurring is $1/M$, where M is the size of the set of possible hash values, and it is independent of the specific value C chosen. The expected size of the chunks is then M. However, in practice, the sizes of the chunks may greatly vary, which is why it is necessary to impose lower and upper limits. For example, if we aim at an average size of 4K, we might not even check at the beginning, thereby assuring that the chunk size will not be below, say, 1K. Similarly, if the condition has not been fulfilled by any seed and we reach already a chunk size of, say, 8K, we might just cut the chunk at this point, regardless of the hash value.

While this strategy will indeed force the chunk size to be between 1K and 8K in our example, these extreme values are "artificial" cutoff points. They impose breaks in the flow of data that are not robust and not reproducible in the case of relatively small inserts or deletes. In general, the distribution of segment sizes is geometric. Cutting off an arbitrary section at the start actually eliminates a very large number of potential segment boundaries. Chopping the tail at an arbitrary size cuts a tail of infinite length, affecting the mean segment size more than would be expected.

Furthermore, segmentation techniques based on these rules produce a very inconvenient distribution of segment sizes because of their geometric distribution. There are a very large number of very small segments and a significant number of very large segments. This stresses the storage subsystem of a program that must store and index these segments.

The problem of segmentation has been the subject of much literature, one of the first being [8]. A brief survey can be found in [4]. Some of the approaches, e.g. [3] are more rigorous. A good description of segmentation appears in the text of [7].

Here we propose a method that tries to rectify the shortcomings of minimal and maximal segment size, while also providing segment sizes that are bunched around the mean size. The basic idea is a new way of text segmentation, in which the probability of declaring a segment boundary changes with the number of bytes read since the previously declared segment boundary. This enables us to control the segment size distribution with much greater accuracy than what is possible with existing segmentation techniques.

Initially, it is highly unlikely (but still possible) that a boundary will be

declared. This means that there are very few small segments, and hence no need to impose an artificial minimum segment size. As more bytes pass since the end of the previous segment, the criterion for declaring a segment is relaxed. By relaxing the criterion eventually completely, we encourage the distribution to tail off as sharply or as loosely as we need. This means that no artificial maximal segment size is needed. This property is especially important, because data may contain very long sequences (e.g., stretches of blanks or zeros) that may not trigger declaring a segment boundary. These can safely be chopped at an artificial maximal size without affecting deduplication.

This relaxation of the segmentation criteria is strictly defined as a family of functions such that each later member "includes" all the previous ones. This provides robustness to inserts and deletes. By tuning this relaxation, we are able to produce approximately any segment size distribution we prefer. We may choose one tailored to the needs of the storage subsystem that must store the unique segments.

In the next section, we present the details of the proposed method and extend the ideas in Section 3 to the usage of fractional bits. Finally, Section 4 brings some experimental results.

2. New segmentation procedure

Instead of working with a single hash function h and a single constant C, we shall use a sequence of functions and constants h_i and C_i, $i = 1, 2, \ldots, n$, fulfilling the following conditions:

1. All functions are easy to calculate;
2. there exists an increasing sequence of probabilities $p_1, p_2, \ldots, p_n$ such that for any seed S of fixed length k, $\Pr(h_i(S) = C_i) = p_i$, where $\Pr()$ denotes the probability function;
3. the conditions are inclusive in the sense that

$$\forall S \quad \forall j > i \qquad h_i(S) = C_i \quad \longrightarrow \quad h_j(S) = C_j.$$

The sequence of functions h_i is then used to partition the potential chunk that is being built into three regions, delimited by the four values A_L, P_L, P_U, A_U, corresponding to the absolute lower, preferred lower, preferred upper and absolute upper limits for the occurrence of the (right) chunk boundary, as depicted in Figure 2 below. The target value of the expected size, E, is indicated by the black bar. Preferably, we want this value to fall between P_L and P_U, however, we might tolerate exceeding these limits, but

not below A_L and not above A_U. This is achieved by choosing one of the indices j_0, $1 < j_0 < n$, and setting $p_{j_0} = 1/E$. Recall that our procedure for cutting the chunk being built at the current position is checking whether $h_j(S) = C_j$, where S is the seed extending up to the current position, and repeating this test for every byte, i.e., considering overlapping seeds. We shall use the same function h_{j_0} while the chunk size is in the preferred (grey) zone, between P_L and P_U. However, the range between A_L and P_L will be partitioned into sub-intervals in which the hash functions used are, in order, $h_1, h_2, \ldots, h_{j_0-1}$, and similarly, the range between P_U and A_U will be partitioned into sub-intervals in which the hash functions used are, in order, $h_{j_0+1}, \ldots, h_n$. By setting the last probability $p_n = 1$, the test for h_n has probability 1 to succeed, therefore A_U is indeed an upper limit.

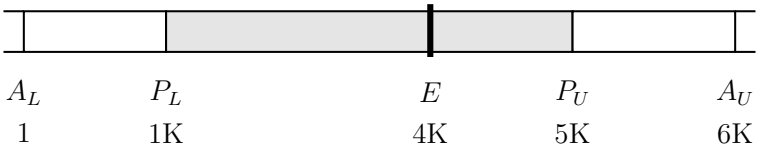

FIGURE 2: *Possible regions for chunk boundaries.*

The main advantage of the proposed method is then that the chunk size needs no artificial lower or upper limits, because these limits are obtained in a natural and consistent way, so that the chunking mechanism can be applied without all the drawbacks mentioned above.

The method works because of the chosen conditions on the sequence of hash functions. The first condition is a basic requirement of any hash function. The second condition lets us define the cut-off condition differently depending on the number of the already accumulated bytes in the current chunk: we shall start with a very low probability of setting the boundary of the chunk, so that very small chunks will almost surely not appear. The closer we get to the target size, say 4K, the larger the probability will get, and within a range to be chosen around the ideal chunk size, say, between 1K and 5K, the probability will be constant. Once we have passed this upper limit, the cut-off probability will start rising, so that it will get increasingly difficult to extend the chunk further. An absolute upper size of the chunk can be imposed by defining $p_n = 1$, that is, the first seed considered when getting to the last function will be declared as being the last seed of the current chunk.

The third condition deals with inserts and deletes. This is best explained

by considering Figure 3 below. The top line represents two consecutive chunks of the original data. Suppose now that a short sequence of new bytes is inserted, as in the middle line of the figure. There is of course the possibility that one of the newly added seeds will fulfill the cut-off condition, but if the inserted block is small, the probability for this to happen is so small that it can be safely ignored. If no new boundary has been declared, the seed S which ended at position A in the original layout has been pushed further to position B, which implies that the test applied on it is $h_j(S) = C_j$ for some $j > i$, therefore S will be declared as boundary and subsequent chunks will not be affected.

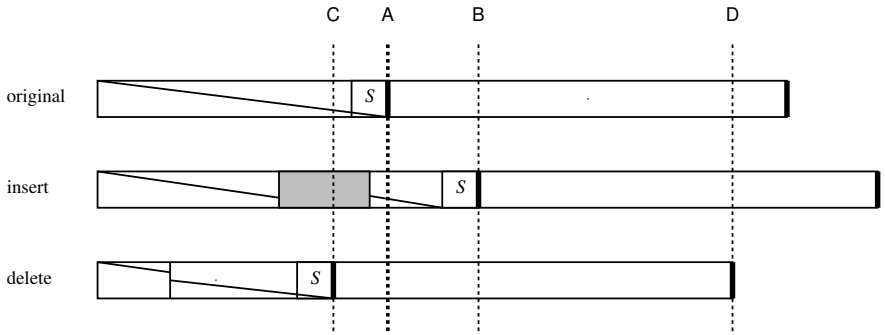

FIGURE 3: *Schematic representation of the effect of insert and delete.*

If some bytes have been deleted from the first chunk, as displayed in the lowest line of the figure, the seed S is moved to an earlier position C, so the condition checked on it, $h_t(S) = C_t$ for some $t \leq i$, might be stricter than before. It is thus possible that the boundary at level C will be missed. But depending on the number of deleted bytes, the condition might also be the same (if $t = i$), or, if $i - t$ is small, the probability of getting even this cut-off point might not be too low. In any case, even if this chunk limit is lost, it is possible that the next one, which has now been moved backwards to position D, is still to the right of A, so it will be caught.

An implementation can set the limiting values as shown in Figure 2. To define the sequence of functions h_i, we first choose a random large prime number P. In practice, since arithmetic will be performed modulo P and given that typical CPUs at present mostly have 64-bit capabilities, it will be convenient to restrict ourselves to 64-bit operations, implying $P < 2^{64}$. If we were to choose a new random prime in every calculation, as is done for the Karp-Rabin pattern matching [6], there would be no need to impose also a lower limit on P, since the probability of repeatedly choosing small

primes is negligible (for a number to be "small", several of its randomly chosen leading bits have to be zero) . But in our case, since the intention is to use a single prime for the entire system, we should prevent a bad choice by imposing also, say, that $P > 2^{60}$. This assures that P has at least 60 significant bits, without being too restrictive, since the number of primes in the given range is of the order of 2^{55}. Let $r_1, r_2, \ldots, r_n$ be a decreasing sequence of integers, subject to the constraints

$$32 = r_1 > r_2 > \cdots > r_{j_0-1} > r_{j_0} = \log_2 E > r_{j_0+1} > \cdots > r_{n-1} > r_n = 0,$$

where E is the target value for the expected size defined earlier, the functions h_i, for $i = 1, 2, \ldots, n$, will then be defined as

$$h_i(S) = (S \bmod P) \bmod 2^{r_i},$$

in other words, $h_i(S)$ are the r_i rightmost bits of the remainder of S modulo P.

The next step is to choose a random 32-bit constant C, and to define

$$C_i = C \bmod 2^{r_i},$$

that is, the C_i are the r_i rightmost bits of C. Theoretically, we could have chosen the C_i at random, if indeed the hash functions gave uniformly distributed values. Practically, it will be convenient to have all the C_i as suffixes of different lengths of the same binary string, which enables us to fulfill the third of the set of conditions defined at the beginning of this section.

In our particular implementation, we choose the following parameters:

$$\begin{gathered} n = 18, \quad j_0 = 11, \quad r_{11} = 12, \\ (r_1, \ldots, r_{10}) = (32, 30, 28, 26, 24, 22, 20, 18, 16, 14), \\ (r_{12}, \ldots, r_{18}) = (11, 9, 7, 5, 3, 1, 0). \end{gathered}$$

Figure 4 is a plot of the number of bits involved in the hashing (which is minus the log of the probability of declaring the current position as a boundary point) as function of the current size of the chunk being built. We see that we start with a very low probability, 2^{-32}, which gradually gets larger (i.e., the number of bits decreases). The sizes of the corresponding ranges start with 2 bytes for 32 bits and 2 bytes for 30 bits, and then double at each step (4 bytes for 28 bits, 8 bytes for 26 bits, ..., 512 bytes for 14 bits). This corresponds to the range from A_L to P_L and spans exactly 1K. Then from 1K to 5K we stay with 12 bits, that is, probability 2^{-12}, and then

continue increasing the probabilities, this time on ranges that start with 512 bytes for 11 bits, then halving to 256 bytes for 9 bits, up to 64 bytes for 5 bits, 32 bytes for 3 bits and 31 bytes for 1 bit. There is also a possibility for 0 bits, but a range of only 1 byte is assigned, since it guarantees success at the first try. This partition of the interval corresponds to the example values given in the bottom line of Figure 2.

Denote by w_i the number of times the procedure is applied with r_i if it still continues, that is, no boundary for the current chunk has yet been set. We then have for this example setting:

$$(w_1, \ldots, w_n) = (2, 2, 4, 8, \ldots, 512, \mathbf{4096}, 512, 256, 128, 64, 32, 31, 1),$$

where w_{11} corresponding to r_{j_0} has been boldfaced.

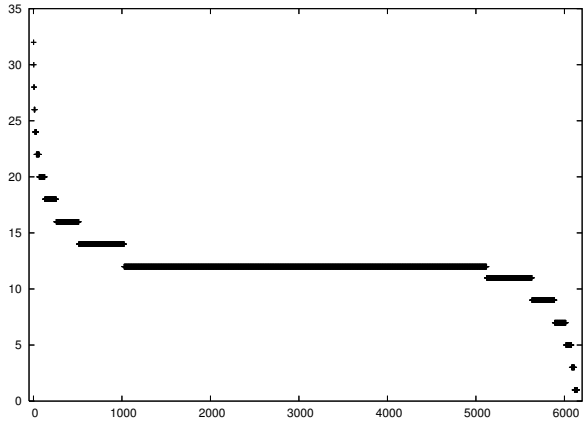

FIGURE 4: *Plotting probability of declaring a boundary as function of chunk size.*

Denote the length of a given chunk by L, which is a random variable whose expected value we are interested in. To evaluate the expected size of the chunk for the given settings, we shall use the formula $E(L) = \sum_{i=1}^{A_U} \Pr(L \geq i)$. The probability of getting a chunk size L which is $\geq M$ is the probability of getting failures on the first M trials, and can be evaluated as follows. Let k be the index of the range to which the current size M belongs, that is, given M, we find k which satisfies

$$r_{k-1} > M \geq r_k.$$

We can then calculate the probability as:

$$\Pr(L \geq M) = \Big[\prod_{t=1}^{k-1} \big(1 - 2^{-r_t}\big)^{w_t} \Big] \quad \big(1 - 2^{-r_k}\big)^{M - \sum_{t=1}^{k-1} r_t} .$$

Figure 5 displays these cumulative probabilities for our example distribution. For these values, we get as expected value for the chunk size: $E(L) = 3744$.

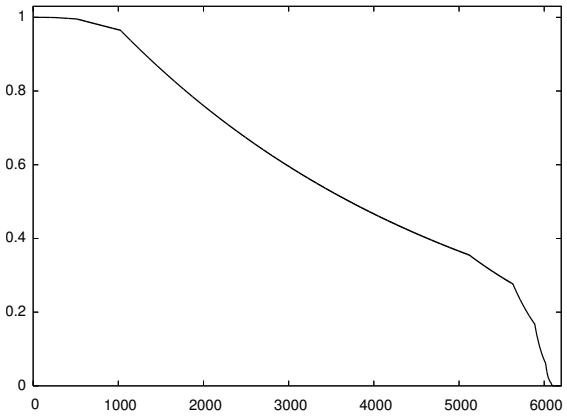

FIGURE 5: *Cumulative probabilities* $\Pr(L \geq M)$.

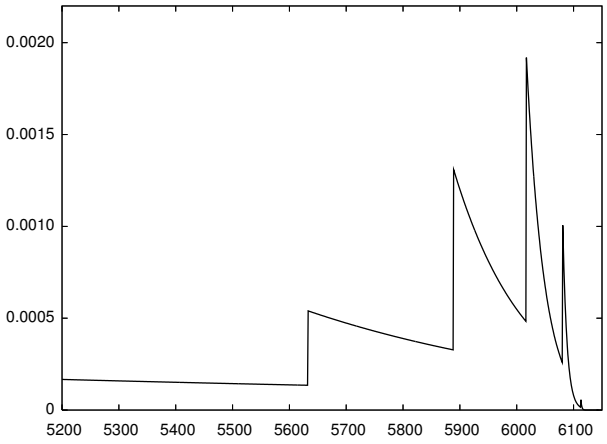

FIGURE 6: *Individual probabilities* $\Pr(L = M)$.

The individual probabilities can be derived from the cumulative ones by

$$\Pr(L = M) = \Pr(L \geq M) - \Pr(L \geq M - 1).$$

The hedgehog shaped graph in Figure 6 gives the tail of these probabilities for our example distribution. The spikes in this plot are due to the discrete nature of the distribution: using an integral number of bits for every test, the resulting probability function will not be continuous at integer points. If we prefer getting a continuous bell shaped Gaussian curve, we need to perform the tests with r_i bits without restricting the r_i to be integers. This calls for trying to deal with fractional bits or at least to simulate the behavior of the probability function as if fractional bits could be compared. This is done in the following section.

3. Cutting chunks using fractional bits

The hash functions used were of the form $S \bmod P$, where S is a sequence of k consecutive bytes considered as the binary representation of one large integer of length $8k$ bits, and P is some large prime number that has been chosen arbitrarily, but is fixed throughout the process. To simulate the fractional bits, let us first decide how fine grained the resolution ought to be. This is done by deciding on a step size ε, where the discrete steps correspond to $\varepsilon = 1$, and we could impose, e.g., $\varepsilon = 10^{-3}$. We thus need $\lceil -\log_2 \varepsilon \rceil$ additional bits in our hash values. Suppose we want to simulate the hashing as if it were working on ℓ bits, where ℓ is not an integer. Define the fractional part of ℓ as $f = \ell - \lfloor \ell \rfloor$, then $0 < f < 1$. We shall use either $\lfloor \ell \rfloor$ or $\lceil \ell \rceil$ bits, by first comparing just the $\lfloor \ell \rfloor$ first bits, and checking also the $\lfloor \ell \rfloor + 1$st bit with probability f'. This probability f' will be chosen as follows. Since we are simulating a sequence of Bernoulli trials, we want the probability of failure to be

$$2^{-\ell} = 2^{-(\lfloor \ell \rfloor + f)} = 2^{-\lfloor \ell \rfloor} \cdot 2^{-f}.$$

On the other hand, comparing only $\lfloor \ell \rfloor$ bits, and the additional bit with probability f', we get as probability for failure

$$(1 - f')2^{-\lfloor \ell \rfloor} + f'2^{-\lfloor \ell \rfloor - 1}.$$

Equating the two, we can derive f' as function of f:

$$f' = 2 - 2^{-f+1}.$$

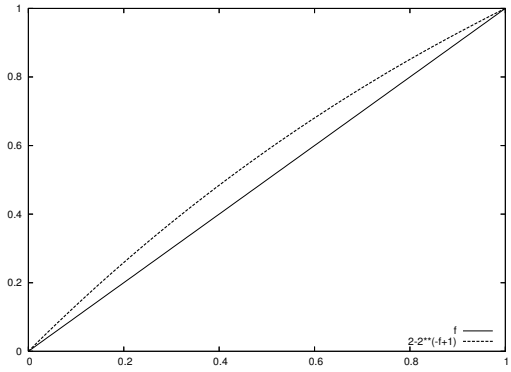

FIGURE 7: *Plotting* $f' = 2 - 2^{-f+1}$.

Figure 7 plots the value of f' as function of f and shows that f' is only slightly larger. For example, to simulate a comparison on $\lfloor \ell \rfloor + \frac{1}{2}$ bits, we should compare the additional bit with probability $f' = 2 - \sqrt{2} = 0.586$.

A first thought about how to implement the comparison of the $\lfloor \ell \rfloor + 1$st bit with probability f' could be to generate a random number r between 0 and 1, and then perform the additional comparison if and only if $r \leq f'$. Such a strategy would, however, hurt the consistency of the chunking procedure: if the same chunk reoccurs, this would not guarantee the same decision at the comparisons of the last seeds of length $\lfloor \ell \rfloor + 1$ for both occurrences, so the system could fail in detecting a chunk that might be deduplicated. To rectify this, instead of r, one should rather use a number r' depending solely on the currently processed chunk, similarly to a pseudo-random number generator. For example, consider an arbitrary, yet constant, subsequence of the bits currently forming the processed chunk S, denote the number represented by this subsequence as S', choose a random large prime Q, which is different from the prime P chosen earlier, and then set the threshold probability to be

$$r' = \frac{(S' + |S|) \bmod Q}{Q},$$

where the current length of the chunk has been added to avoid a bias in the case of long stretches of zeros.

As alternative, the probability f' can be simulated by exploiting the unused bits generated by the hashing function. Suppose the (first) hash function h we apply on each seed S returns a 64-bit value. Since only at most 32 bits of them are actually used by the functions h_i, r could be defined

by a some fixed subset of the remaining bits.

Once the question of how to process fractional bits has been handled, the next step was to define the number of bits used in the sequence of hash functions as a continuous decreasing function. The first option would be to decrease the number of bits linearly from 32 to 0, in 4K steps. This, however, gives a quite narrow distribution of the chunk sizes, which all fall between roughly 2K and 3600, with average 3026. Starting with less than 32 bits, but leaving the 4K steps, reduces the average and broadens the bell shaped distribution. If we aim at getting an average chunk size of 2K, we should start at 18.3 bits. Decreasing this number in 4K regular steps to 0 yields then the solid line plots in the graphs of Figures 8(a)–(c). Figure 8(a) shows the decrease in the number of bits used in the hashing function, as a continuous function of the number of bytes in the current chunk. Figures 8(b) and 8(c) are the corresponding cumulative and individual probabilities for the possible chunk sizes, i.e., $\Pr(L \geq M)$ and $\Pr(L = M)$ for a size M of a chunk, $1 \leq M \leq 5000$.

The decrease of the number of bits could also be chosen proportional to the harmonic sum rather than linearly, as would be suggested by Zipf's law [13], which is supposed to describe the distribution of many real-life phenomena. If B_i denotes the (not necessarily integral) number of bits used to decide if the cutoff point should be after the i-th byte, then we have, for example, $B_k = 32$ (recall that k is the size of the seed), and for $i \geq k$,

$$B_{i+1} = B_i - \frac{32}{i \cdot H_n},$$

where H_n is the n-th harmonic number, equal to $\ln n - 0.577$. For $n = 4K = 4096$, we have $H_n = 8.895$. This would exhibit a steeper decrease at the beginning but the difference between consecutive steps would be decreasing by itself.

The plots corresponding to the harmonic decrease appear as dashed lines in the graphs of Figures 8(a)–(c). Using again 4K steps to decrease the number of bits harmonically from 32 to 0 gave a nicely symmetrical bell shaped curve for the distribution of the chunk lengths, but the average was low at 487, and practically all the values were smaller than 1K. To move the average further up and broaden the curve, the decreasing steps could be multiplied by some constant $\alpha > 1$, so that one gets

$$B_{i+1} = B_i - \frac{32}{\alpha \cdot i \cdot H_n}.$$

The dashed line in the plots correspond to $\alpha = 1$ and the dotted lines to $\alpha = 1.34$, which yielded an average chunk size of 2K. The first few elements

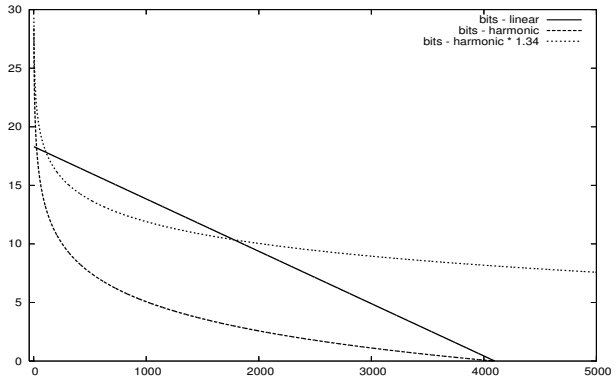

FIGURE 8(a): *Continuous number of bits in hash function.*

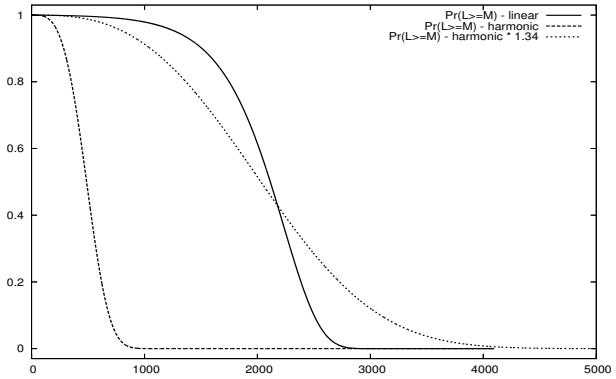

FIGURE 8(b): *Cumulative probabilities for continuous decrease.*

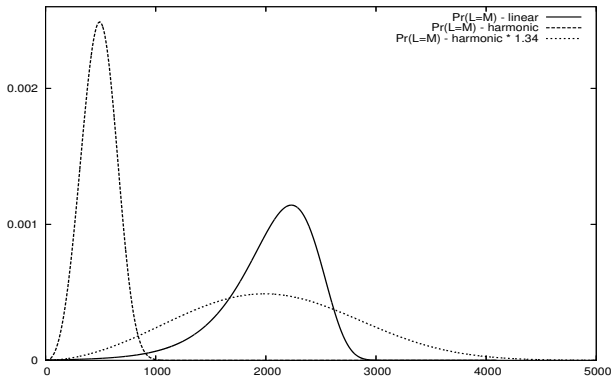

FIGURE 8(c): *Individual probabilities for continuous decrease.*

of the B_i sequence are then 32, 29.32, 27.97, 27.08, 26.41, etc, but even after 5900 steps, the number of bits used is still about 7.14.

We also experimented with other decreasing functions than the harmonic sum, e.g., having the difference $B_n - B_{n+1}$ between consecutive bit-sizes proportional to $1/\sqrt{n}$, $\log n/n$, and others, but the harmonic decrease with parameter α gave the best results.

4. Experimental results

The setup for deduplication experiments in order to get some idea on the performance of new proposed ideas presents challenges. While there are well established test cases which have been agreed upon in the compression community, like the Calgary or the Canterbury [5] corpora, there is no equivalent for deduplication tests. The reason is mainly that the performance does not depend on the nature of the files, but rather on the their repetitiveness. Thus even an individual file containing random data, which cannot be compressed, may still profit from deduplication if it or any of its sub-parts appear more than once in the repository.

chunking strategy	All chunks			Unique chunks		
	number in million	avg bytes	std	number in million	avg bytes	std
constant	2.0	2708	3014	1.2	2952	3087
variable probability	2.1	2620	1532	1.3	2664	1492
with fractional bits	1.8	3078	1592	1.2	3118	1553

TABLE 1: *Details on the different chunking procedures.*

The other problem is that for deduplication to be interesting, there is a need to handle huge corpora. As there is no possibility to find data that could be deemed to be representative, the experimental results are presented as examples only, without claiming that one could extrapolate from them information on the performance in general. Nevertheless, the results on our real-life tests may be considered as support, if not as evidence, for the feasibility of our approach.

Our test files were a collection of different Ubuntu Linux OS variants and versions[1] of total size 5.05 GB. This repository was first processed by a

[1] Ubuntu server .vdi files, versions 10.04.2, 11.10, 12.04, 12.10

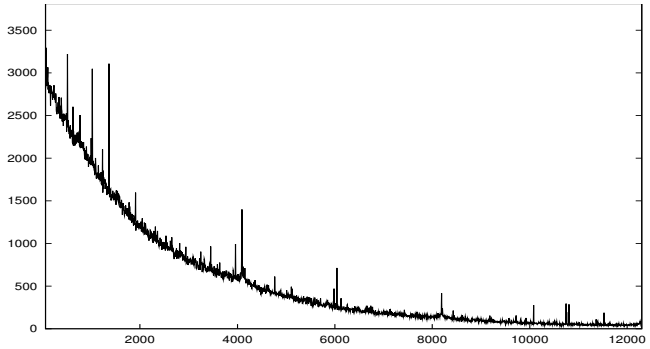

FIGURE 9(a): *Chunk distribution with constant cutoff probability.*

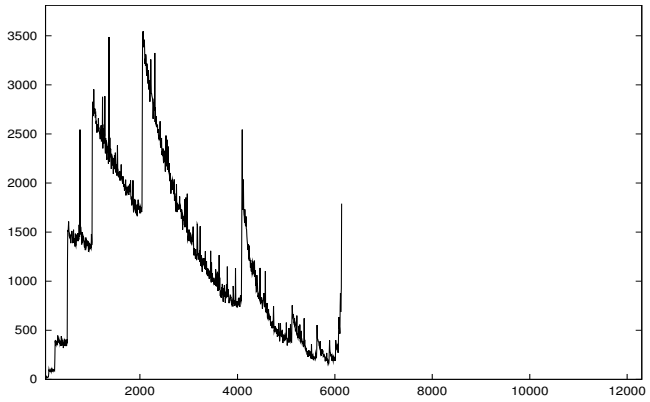

FIGURE 9(b): *Chunk distribution with varying cutoff probability, using integral bits.*

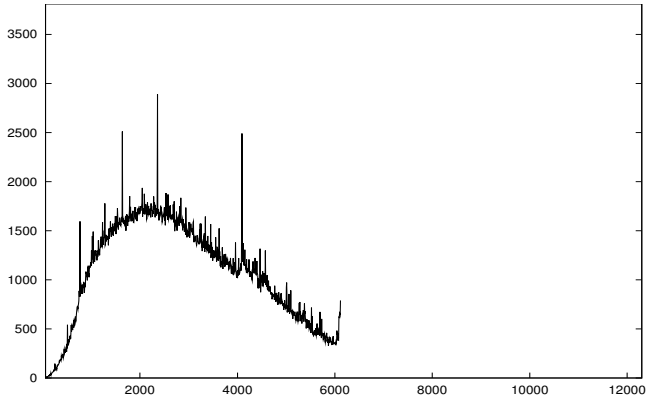

FIGURE 9(c): *Chunk distribution with varying cutoff probability, using fractional bits.*

chunking procedure using a constant probability for setting the boundaries, aiming at an average chunk size of about 2K. Then the experiment was repeated with the varying cutoff conditions proposed herein. For all settings, a seed size of 48 bytes = 384 bits was chosen. The maximal length was set to $A_U = 6$K. Table 1 gives some statistical details.

As can be seen, average and standard deviation are very close for the constant variant, as is expected for an exponential distribution. For the variable probabilities, corresponding to integral or fractional bits, the standard deviation is much smaller, indicating that most values are closer to the mean, which is about 2600 to 3K. The plots in Figures 9(a), 9(b) and 9(b) are histograms showing the distribution of the chunk sizes obtained by these procedures for the unique chunks, Figure 9(a) using the constant cutoff condition, and Figures 9(b) and 9(c) corresponding to the procedure proposed in this work based on varying probabilities to declare a chunk boundary, the former using only integral bits, the latter allowing also fractional bits, as explained above. The y axis gives the number of chunks as a function of a given size x on the x-axis. Although the average chunk size for Figure 9(a) was close to 3K, there was a very long tail in the distribution with the constant condition, and we display here only the values up to a size of 12K, where there were still around 70 occurrences for any chunk size. In spite of the fluctuations due to various anomalies of the real-life input data, the exponentially decreasing trend of the function in Figure 9(a) is clearly noticeable.

By contrast, the distribution in Figure 9(b) corresponding to varying cutoff conditions is hedgehog shaped with an underlying Gaussian bell curve, and in the plot of Figure 9(c), the continuous bell shape is evident.

5. Conclusion

This paper is concerned with the determination of chunk boundaries in a deduplication system. Fixed sized chunks can be ruled out because of the problems they cause in the cases of even small inserts and deletes. Previous work proposes variable length chunks, where the chunk boundaries are defined in a data dependent way. This implies that the chunk sizes are often spread over a large range, which might have a negative impact on the storage performance. The contribution of this work is a dynamic method to set chunk boundaries by which the chunk size distribution is controlled in a natural way.

The idea is to define a sequence of hash functions that are related to the bits of some randomly chosen numbers. To achieve smooth distributions, the

work has been extended to simulate the behavior of fractional bits, which, to the best of our knowledge, have not been treated so far in this context. All the methods have been tested on a real life database of several Ubuntu Linux OS variants.

References

[1] Aronovich L., Asher R., Bachmat E., Bitner H., Hirsch M., Klein S.T., The design of a similarity based deduplication System, *Proc. SYSTOR'09*, Haifa, (2009) 1–14.

[2] Aronovich L., Asher R., Harnik D., Hirsch M., Klein S.T., Toaff Y., Similarity based deduplication with small data chunks, *Discrete Applied Mathematics* **212** (2016) 10–22.

[3] Bjørner N., Blass A., Gurevich Y., Content-dependent chunking for differential compression, the local maximum approach, *Journal of Computer and System Sciences* **76**(3–4) (2010) 154–203.

[4] Cai B., Zhang F.L., Wang C., Research on chunking algorithms of data de-duplication, *Proceedings of the 2012 International Conference on Communication, Electronics and Automation Engineering*, Xi'an, China, *Advances in Intelligent Systems and Computing* **181** (2013) 1019–1025.

[5] http://corpus.canterbury.ac.nz/

[6] Karp R.M., Rabin M.O., Efficient randomized pattern-matching algorithms, *IBM Journal of Research and Development*, **31**(2) (1987) 249–260.

[7] Moulton G.H., Whitehill S.B., Hash file system and method for use in a commonality factoring system, U.S. Pat. No. 6,704,730, issued March 9, 2004.

[8] Muthitacharoen A., Chen B., Mazières D., A low-bandwidth network file system, *Proc. of the 18th ACM Symposium on Operating System Principles*, Banff, Alberta (2001) 174–187.

[9] Quinlan S., Dorward S., Venti: A new approach to archival storage, *Proceedings of FAST'02, the 1st USENIX Conference on File and Storage Technologies* , Monterey, CA (2002) 89–101.

[10] Song B., Xiao L., Qin G., Ruan L., Qiu S., A deduplication algorithm based on data similarity and delta encoding, in *Geo-Spatial Knowledge and Intelligence*, Springer, Singapore, (2017) 245–253.

[11] Xia W., Jiang H., Feng D., Hua Y., Similarity and locality based indexing for high performance data deduplication, *IEEE Trans. Computers* **64**(4), (2015) 1162–1176.

[12] Zhu B., Li K., Patterson H., Avoiding the disk bottleneck in the Data Domain deduplication file system, *Proceedings of FAST'08, the 6th USENIX Conference on File and Storage Technologies*, San Jose, CA (2008) 279–292.

[13] Zipf G.K., *The psycho-biology of language*, Boston, Houghton (1935).

[14] Ziv J., Lempel A., A universal algorithm for sequential data compression, *IEEE Trans. on Information Theory* **23** (1977) 337–343.