A Framework for Analyzing and Improving Content-Based Chunking Algorithms

Kave Eshghi, Hsiu Khuern Tang
Intelligent Enterprise Technologies Laboratory
HP Laboratories Palo Alto
HPL-2005-30(R.1)
September 22, 2005*

E-mail: kave.eshghi@hp.com, hsiu-khuern.tang@hp.com

stateless chunking algorithm, hash function, storage overhead, archival file systems, low bandwidth network, file system

We present a framework for analyzing content-based chunking algorithms, as used for example in the Low Bandwidth Networked File System. We use this framework for the evaluation of the basic sliding window algorithm, and its two known variants. We develop a new chunking algorithm that performs significantly better than the known algorithms on real, non-random data.

* Internal Accession Date Only

Approved for External Publication

A Framework for Analyzing and Improving Content-Based Chunking Algorithms

Kave Eshghi (kave.eshghi@hp.com)
Hsiu Khuern Tang (hsiu-khuern.tang@hp.com)
Hewlett-Packard Laboratories
Palo Alto, CA

February 25, 2005

Abstract

We present a framework for analyzing content-based chunking algorithms, as used for example in the Low Bandwidth Networked File System. We use this framework for the evaluation of the basic sliding window algorithm, and its two known variants. We develop a new chunking algorithm that performs significantly better than the known algorithms on real, non-random data.

1 Introduction

In this paper we consider a class of algorithms, which we call stateless chunking algorithms, that solve the following problem: Break a long byte sequence S into a sequence of smaller sized blocks or chunks $c_1, c_2, \ldots, c_n$ such that the chunk sequence is stable under local modification of S. Intuitively, stability under local modification means that if we make a small modification to S, turning it into S', and apply the chunking algorithm to S', most of the chunks created for S' are identical to the chunks created for S. The 'stateless' in the name of the algorithm implies that to perform its task, the algorithm relies only on the byte sequence as input. In other words, the algorithm produces the same chunks no matter where and when it is run. This excludes, for example, algorithms that consider the history of the sequence, or the state of a server where other versions of the sequence might be stored. From now on we use the term 'chunking algorithm' to mean a stateless chunking algorithm that is stable under local modification.

The scheme traditionally used in file systems for breaking a byte sequence into blocks is to use fixed size blocks. The main problem with this idea is that inserting or deleting even a single byte in the middle of the sequence will shift all the block boundaries following the modification point, resulting in different blocks. Thus, on average, after every insertion or deletion to the file half the blocks would be different. This is far from the stability under local modification property that we desire.

The sliding window chunking algorithm [3], which offers a solution to this problem, is the starting point of our investigation. The contributions of this paper are as follows:

- We provide an analytic framework for evaluating and comparing chunking algorithms. We use this framework to analyze the basic sliding window algorithm and two of its existing variants.
- We propose a new algorithm, the Two Thresholds, Two Divisors Algorithm (TTTD) that performs much better than the existing algorithms.
- We provide experimental validation of the superiority of TTTD based on a large collection of real files.

2 Motivation

The Low Bandwidth Networked File System [3] introduced a scheme for minimizing the amount

of information passed between a file system client and server, by taking advantage of the fact that often an earlier version of a file that is to be transmitted between the client and server already exists on the receiving side. The key idea was to use a chunking algorithm to break up the files. Consider the case where the client wishes to save a file on the server. The server maintains a table of the hashes of the file chunks that it has available, with a pointer to where the chunk itself is stored. The client chunks up the file that it wants to save, and sends the server the list of hashes of the chunks. The server looks up the hashes in its table, and sends back to the client a request for the chunks whose hash it could not find. The client then sends those chunks to the server. The server reconstructs the file from the chunks.

It is often the case that the server has an earlier version of the file that the client wants to store. This can happen, for example, when the new file has come about by retrieving and editing an existing file. In such cases, due to the stability property of the chunking algorithm, the server would already have most of the chunks belonging to the client's file. As a result, only a fraction of the file is actually transmitted, but the server can reconstruct the file without any ambiguity or error.

Our goal in this paper is to provide a framework for analyzing the performance of chunking algorithms in an LBNFS like system. Using the intuition provided by our analysis, we introduce a more sophisticated chunking algorithm that outperforms the algorithm used by LBNFS.

2.1 Archival, Versioned File Systems

While our main motivation has been low bandwidth file synchronization applications such as LBNFS, chunking is also useful in archival file systems. A number of authors [4],[1] have advocated a scheme for archival, write-once storage of data blocks where the hash of each block is used as block identifier, and the storage system's data structures (files, directories) are built on top of this basic scheme, employing familiar file systems data structures (inodes, etc.) using hash identifiers as pointers.

When data blocks are identified by their hashes, if two data blocks have the same hash, only one of them is stored. Since the hash functions are collision resistant, having the same hash value means (with a very high probability) the two blocks are identical. As a result,we avoid wasting resources on storage of duplicate blocks. If a data block is different from another data block their hashes would be different, and both blocks are stored.

In a file system, each file is broken into a number of blocks. In the context of a a hash based archival file systems as described above, there are many files which come about as local modifications of other files. Using a chunking algorithm to break the files into blocks allows sharing of storage resources for such files. Using fixed size blocks, however, reduces this resource sharing due to the boundary shifting effect.

3 The Basic Sliding Window Algorithm (BSW)

The Basic Sliding Window Algorithm [3] avoids boundary shifting by making chunk boundaries dependent on the local contents of the file, rather than distance from the beginning of the file. The sliding window algorithm works as follows: There is a pre-determined integer D. A fixed width sliding window is moved across the file, and at every position in the file, the contents of this window are fingerprinted. A special, highly efficient fingerprint algorithm, e.g. Rabins fingerprint algorithm [5], is used for this purpose. A position k is declared to be a chunk boundary if there is a D-match at k (see Figure 1).

Definition 1 (fingerprint match) *Given a sequence $S = s_1, s_2, \ldots, s_n$, a fingerprint function h and a window length l, there is a D-match at k if, for some pre-determined $r < D$,*

$$h(W) \bmod D = r$$

where $W = s_{k-l+1}, s_{k-l+2}, \ldots, s_k$ is the subsequence of length l preceding the position k in S.

Note: The value of the residue r plays no part in our analysis. The only point to keep in mind is that it has to be fixed for a particulary value

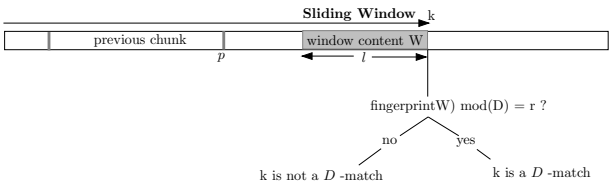

Figure 1: The Sliding Window Algorithm

of D. In practice, we use the value $D-1$ in our algorithms.

The advantage of this scheme over the use of fixed sized chunks is that, since chunk boundaries are determined by the contents of the file, when data is inserted into or deleted from the file, boundaries in the areas not affected by the modification remain in the same position relative to each other. Thus all the chunks other than the chunks in which the modification occurs stay the same. In other words, BSW is stable under local modifications.

4 Analytic Framework

We need a quantitative framework for analyzing and comparing different chunking algorithms. In this section we introduce the notions of *modification overhead* and *modification index* for this purpose. Intuitively, modification overhead measures the number of bytes that need to be transferred not because they are new data, but because chunk boundaries do not in general correspond to the beginning and end of the modified sequence. Modification index is the normalization of modification overhead by dividing it with mean chunk size. The more precise definition of these concepts is presented below.

4.1 Notation

- For two byte sequences P and Q, we use PQ to denote their concatenation. If $PQ = S$, P is a *prefix* of S and Q is a *postfix* of S. When P is a prefix of S, we use $rem(S, P)$ to denote the sequence that, when it is concatenated to P, results in S. We use $|S|$ to denote the length of S.
- For two sequences S and S', we use $premax(S, S')$ to denote the longest shared prefix of S and S'. Let $r = premax(S, S')$. We use $postmax(S, S')$ to denote the longest shared postfix of $rem(S, r)$ and $rem(S', r)$.
- We define $shared(S, S')$ as $|premax(S, S')| + |postmax(S, S')|$. We define $new(S, S')$ as $|S'| - shared(S, S')$
- We use the terminology 'the expected value of the function $f(S)$ over random large S' to mean the expected value of $f(S)$ when S is a random byte sequence of length n where n is large enough to ignore boundary effects. When we use this statement, we make the assumption that the expected value of $f(S)$ is convergent when $n \to \infty$
- For a chunking algorithm A and a sequence of bytes S, $H^A(S)$ denotes the sequence $c_1, c_2, \ldots, c_n$ of chunks generated by running A on S. $\mu^A(S)$ denotes average size of the chunks in $H^A(S)$. μ^A, called average chunk size for A, denotes the expected value of $\mu^A(S)$ over random large S. $\sigma^A(S)$ denotes the standard deviation of of the chunks in $H^A(S)$, and σ^A denotes the expected value of $\sigma^A(S)$ for large random S.
- $\Delta^A(S, S')$ is defined as the total size of the chunks that occur in $H^A(S')$ and not in $H^A(S)$.
- Given a threshold value t, The sequence S' is a *local modification* of S if $shared(S, S')/|S| > t$. S' is a *random local modification* of S if it is randomly chosen from among all the local modifications of S. The threshold value is not significant in our analysis, as long as it is not so small that the two sequences do not share anything. A reasonable value for t could be 0.1

Definition 2 (modification overhead) *Let S be a sequence of bytes. Then $V^A(S)$ is defined as the expected value of :*

$$\Delta^A(S, S') - new(S, S') \tag{1}$$

where S' is a random local modification of S.

The intuition behind the definition of $V^A(S, S')$ is as follows: $\Delta^A(S, S')$ measure the number of bytes in the chunks that occur in $H^A(S')$ and not in $H^A(S)$. But some of these chunks will contain (at least partially) 'new' data, i.e. data that was not in S and was inserted in the transition from S to S'. $new(S, S')$ measures the size of this new data, so by subtracting $new(S, S')$ from $\Delta^A(S, S')$ we can measure the pure overhead of the chunking operation.

For some algorithms, $V^A(S)$ depends on the size of S. For example, for the fixed chunk size algorithm, $V^A(S) \approx |S|/2$. In the case of the algorithms based on the sliding window algorithm, however, $V^A(S)$ is independent of S for large random S. For these algorithms, we define V^A as the expected value of $V^A(S)$ for large random S and call it the modification overhead of the algorithm.

Definition 3 (overhead index α) *For the algorithms for which V^A is defined, we define the overhead index of A, denoted as α^A, as follows:*

$$\alpha^A = \frac{V^A}{\mu^A}$$

V^A which measures, on average, the number of overhead bytes that need to be stored as a result of the chunking algorithm, might seem to be all we need to use for comparing one algorithm against another. But this ignores another cost of the chunking operation: the cost of the meta-data that is necessary to represent a file as a sequence of chunks. In fact, all the algorithms considered here have parameters (such as D for the basic sliding window algorithm) that can be changed to make V^A arbitrarily small, but this is done at the cost of increasing the number of chunks generated, thus increasing the cost of the meta-data.

Whatever the meta-data scheme used, the cost of the meta-data depends on the number of chunks generated for a file. Since, on average, the number of chunks generated is inversely proportional to the average chunk size, and since the average chunk size (μ^A) is independent of the file and its size, by dividing V^A by μ^A we derive a scale free measure of the overhead costs of an algorithm that factors out the meta-data costs. This is the justification for using α^A as the overhead index of the algorithm.

4.2 Example

Let us say there are two algorithms, A1 and A2, with overhead indices 2.44 and 1.52. We choose the parameter values for the two algorithms so that the average chunk size is 1000 for both. There are two text files, F and F', one on the server, one on the client, where F' has come about by one local edit of F, i.e. removing a section of F and replacing it with new text. Our goal is to transfer F' to the server using the LBNFS scheme, i.e chunking both of them, sending the hashes, and transferring the chunks of F' that don't exist in F. Now consider two scenarios: in one we use A1 as the chunking algorithm, in the other we use A2. In both scenarios we need to transmit the meta-data, i.e. the sequence of hashes of F' chunks. The size of the hash sequence is roughly equal in the two scenarios, since the average chunk sizes, and hence the number of chunks, are roughly the same. In both cases there is an overhead, i.e. a number of bytes that need to be transferred not because they are new data, but because chunk boundaries do not in general correspond to the beginning and end of the edited sequence. The size of this overhead is $\alpha \times \mu$, where α is the overhead index and μ is the average chunk size. Thus in the case of A1 the size of the overhead is $1000 \times 2.44 = 2400$, whereas in the case of A2 the size of the overhead is $1000 \times 1.52 = 1520$. In other words, if F' differs from F by one byte, on average using A1 we will transmit 2400 bytes, and using A2 we will transmit 1520 bytes. This example shows the impact of the chunking algorithm's overhead index for LBNFS like schemes.

5 The Significance of Chunk Size Distribution

The sliding window chunking algorithm creates chunks of unequal size. This turns out to have a significant effect on the performance of the algorithm, and is our main motivation for seeking improvements to the algorithm. For the purpose of the analysis in this section, we assume that the size of the modification is small compared to average chunk size. This is a baseline, worst case analysis that establishes the basic framework and

limitations of the algorithms, and provides the intuition for improving on the basic sliding window algorithm. This assumption is relaxed in the rest of the paper, especially in the theoretical analysis of the various algorithms (in the companion paper [6]) and in the experimental results.

5.1 First Affected Chunk

Let S be the contents of a file, and $H^A(S) = c_1, c_2, \ldots, c_n$ the chunks generated by applying algorithm A to S. Consider a small modification to a file: say the insertion of a new line of text into a text file, resulting in the sequence S'. The starting point of the modification will fall within some chunk, let's say c_k, and $H^A(S')$ will not contain this chunk, but a modified version. If the modification is small, this chunk will often be the only chunk that is affected, so its size dominates $\Delta^A(S, S')$, which counts the number of bytes in the chunks that occur in $H^A(S')$ and not in $H^A(S)$.

We call the chunk where the start of the modification falls the *first affected chunk*. Let $\Phi^A(S)$ denote the expected size of the first affected chunk in $H^A(S)$, assuming that the modification point is equally likely at any point in the file. It can be shown that

$$\Phi^A(S) = \mu^A(S) + \frac{(\sigma^A(S))^2}{\mu^A(S)}$$

Intuitively chunk size variance affects the expected size of the first affected chunk because the modification is more likely to start in the middle of a larger chunk than a smaller chunk.

As discussed earlier, for large random S and the algorithms considered here, $\mu^A(S)$ and $\alpha^A(S)$ are independent of S. So, under these assumptions, $\Phi^A(S)$ is also independent of S. We denote $\Phi^A(S)$ for large random S as Φ^A.

5.2 The First Affected Chunk's Contribution to α

Recall that $\alpha^A = V^A/\mu^A$. Let K^A be the expected number of chunks affected by a small modification. While in general the expected size of the first affected chunk is larger than μ^A, the expected size of the chunks following this chunk is μ. This is because the modification point is more likely to fall on a larger chunk than a smaller chunk, but there is no reason to believe that the chunk following the modified chunk is larger or smaller than the average. Thus, when the modification is small,

$$V^A \approx \Phi^A + (K^A - 1)\mu^A$$

By definition of α^A,

$$\alpha^A \approx (\frac{\sigma^A}{\mu^A})^2 + K^A$$

$\frac{\sigma^A}{\mu^A}$ is the *coefficient of variation* of the chunk sizes generated by A. In the case of the sliding window algorithm and large random files, the coefficient of variation is ≈ 1 and $K^A \approx 1$, so $\alpha \approx 2$. Our experiments show that, in general, for non-random files the coefficient of variation of chunks generated by the basic sliding window algorithm is more than one, resulting in even larger α. Clearly, K^A cannot be less than one (since at least one chunk will be affected by a modification), so to improve on the basic algorithm our only chance is to reduce the coefficient of variation.

Modifying the algorithm to reduce the coefficient of variation, however, in general increases the number of chunks affected by a modification (i.e. K^A). At the extreme, breaking the file into a sequence of fixed size chunks reduces the coefficient of variation to near zero, at the cost of a modification on average affecting half the chunks in the file. This analysis shows that it is impossible to have α equal to one, since this implies fixed size blocks.

We need to find algorithms that optimize the benefits of reducing the coefficient of variation versus the undesirability of increasing the number of chunks affected by a local modification. This provides the intuition behind our attempts to improve on BSW. Notice that even though we arrived at this intuition by assuming that the modification is small, the resulting algorithms are evaluated without this assumption.

6 Reducing Chunk Size Variability

There are two possible strategies for reducing chunk size variability: trying to avoid very small chunks, and trying to avoid very large chunks. In this section, we explore both options, and then present an algorithm that combines the best features of the various approaches.

6.1 Eliminating Small Chunks: the Small Chunk Merge (SCM) Algorithm

Intuitively, this algorithm works as follows: first apply the basic sliding window algorithm to derive the chunk boundaries. Next, repeatedly merge chunks whose size is less than or equal to a threshold size, L, with the succeeding chunk until all chunk sizes are larger than L. By eliminating small chunks in this way, we hope, we can reduce chunk size variance relative to average chunk size, thus reducing the overhead index. In practice, this algorithm can be implemented as a variant of the basic sliding window algorithm, where we simply ignore the fingerprint matches that occur before the threshold L is reached. This is a known variant of the Basic Sliding Window algorithm, first mentioned in [3], although the size threshold used in LBNFS is far from optimal.

The performance of this algorithm depends on the ratio L/D, where D is the divisor used for finding fingerprint matches. Figure 2 shows the relationship between α and L/D for large random files. The theoretical result shown here refers to the analytic form of this relationship, detailed in [6]. From the graph it can be verified that the optimum value of α occurs when $L/D \approx 0.85$, where $\alpha \approx 1.5$. This is a significant improvement over the basic sliding window algorithm, where $\alpha \approx 2$.

6.2 Avoiding Large Chunks: Breaking Large Chunks into Fixed Size Sub-Chunks (BFS)

The obvious way to avoid large chunks is to modify the basic sliding window algorithm by imposing a threshold on chunk sizes, where we create a break-point if we reach the threshold without finding a fingerprint match. Conceptually, this is equivalent to the following: first use the basic sliding window algorithm to break the file into chunks. Then, for each chunk c_k whose size is bigger than a threshold T, break it into a sequence of sub-chunks $c_{k_1}, c_{k_2}, \ldots, c_{k_n}$ where the size of all sub-chunks $c_{k_1}, c_{k_2}, \ldots, c_{k_{n-1}}$ is equal to T, and the size of c_{k_n} is less than or equal to T. Again, this is a known variant of the Basic Sliding Window Algorithm, mentioned in [3].

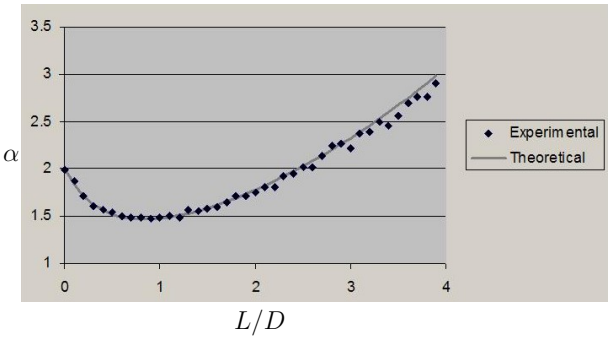

Figure 2: Theoretical VS Experimental Analysis of Chunking Overhead for the SMCR algorithm

While BFS reduces the variability of chunk sizes, in practice this improvement is negated by the increase in the number of chunks affected by a small modification. Figure 3 shows the relationship between the ratio T/D and the overhead index α. (T is the threshold, D is the divisor used for finding fingerprint mathces.)

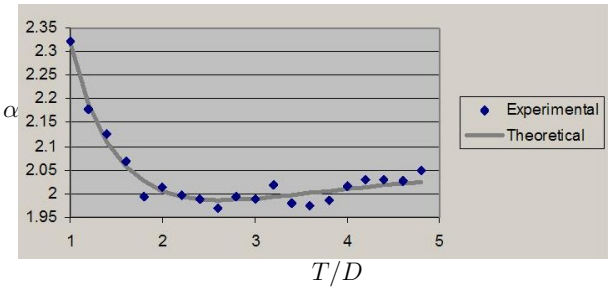

Figure 3: Relationship between α and T for the BFS algorithm

As can be seen from this figure, the best value for α is just under 2, which is only a slight im-

provement on the basic algorithm. The reason that the BFS algorithm fails to substantially improve on the basic sliding window algorithm is as follows: When we break the large chunks produced by the basic sliding window algorithm into fixed size sub-chunks, we re-introduce the boundary shifting problem, at least as far as these large chunks are concerned. In other words, if the modification adds or deletes even one byte in the first sub-chunk, all the subsequent sub-chunks are going to be affected. In practice, this means that for random files we do not make a significant improvement on the basic algorithm.

For non-random files, where there can be significant repetitive patterns in the data, the story is not as bad. Here, the hard threshold imposed on chunk sizes breaks up very large chunks that can come about in such data streams. Where the basic algorithm does particularly badly with these files, the BFS does better (see Table 1 for details). Moreover, the restriction of chunk sizes to an absolute maximum can be useful in simplifying the design of the system components (buffers, packet sizes, etc.)

6.3 Two Divisors (TD) Algorithm

The BFS algorithm fails to improve significantly on the basic algorithm due to the boundary shifting phenomenon within large chunks. The next algorithm that we consider, the Two Divisor (TD) algorithm, attempts to avoid this phenomenon by breaking large chunks based on a secondary, smaller divisor that has a higher chance of finding a fingerprint match. This algorithm is new; it has not been previously discussed in the literature.

Recall the basic sliding window algorithm: A position k is a chunk boundary if, for a predetermined D, there is a D-match at k (see figure 1). With TD, we use a secondary, 'backup' divisor D', where D' is smaller than D, to find breakpoints if the search for a D-match fails within the threshold T_{max}.

In practice, the algorithm is implemented as follows: we scan the sequence starting from the last chunk boundary, at each position updating the fingerprint. At each position we look for both D and D' fingerprint matches, remembering the position of the last D' fingerprint match if we find one.

If we find aD-match before reaching the threshold T_{max}, then we declare that position to be a breakpoint. If we reach the T_{max} threshold without finding a D-match, and there is an earlier D'-match, we use the position of that match as the breakpoint. Otherwise we use the current position (i.e. at the threshold) as the breakpoint.

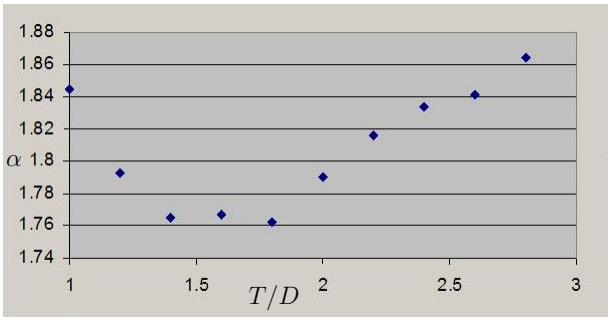

Figure 4: Relationship between α and T for the TD algorithm

Figure 4 shows the performance of this algorithm on large random files, where the value of D' is half D, i.e. finding D'-matches is twice as probable as finding D-matches. As can be seen from this figure, for an appropriate value of T (around $1.8D$) the overhead index is approximately 1.76, which, while not as good as the best performance of the SCM algorithm, is a considerable improvement on the basic sliding window algorithm and the BFS algorithm. Notice that this algorithm, just like BFS, imposes an absolute maximum value on the size of the chunks, so it enjoys the advantages of BFS in this regard, while improving on it in terms of efficiency.

6.4 Two Thresholds, Two Divisors Algorithm (TTTD)

The last algorithm we consider is the Two Threshold, Two Divisor (TTTD) algorithm. This algorithm is a combination of the SCM and TD algorithms. The idea is to modify the TD algorithm by applying a minimum threshold to the chunk sizes: we only look for fingerprint matches, both main and backup, once we are past the threshold. The detailed description of this algorithm (in pseudo

C) is shown in Appendix A.

TTTD has four parameters: D, the main divisor, D', the backup divisor, T_{min}, the minimum chunk size threshold, and T_{max}, the maximum chunk size threshold. It can be considered as a generalization of the other algorithms in that by setting these parameters to appropriate values we get the same behavior as the other algorithms. Table 3 in Appendix B illustrates how the other algorithms can be considered a special case of this algorithm.

We found the best values for the parameters of this algorithm by systematic experimentation, using a hill-climbing strategy for minimizing α^{TTTD}. The values presented in Table 3 in Appendix B yield an average chunk size of 1015 on real data. For another average chunk size m, these parameter values should be multiplied by $m/1015$.

For this choice of parameters, this algorithm performs just as well as SCM on random data, but significantly outperforms SCM (and all the other algorithms considered by us) on real, non-random data (details in next section).

Like BSF and TD, TTTD imposes an absolute maximum limit on chunk sizes. Moreover, in the optimal setting of parameter values the maximum chunk size is 2.8 times average chunk size. This gives it an extra appeal in simplifying system design.

6.5 Experimental Evaluation

Table 1 shows the results of a series of experiments that we conducted to evaluate the performance of the algorithms.

These experiments were performed on two data sets: one was a collection of large random files, generated for this purpose, and the other was a collection of real files of various types downloaded from the internet, mostly from large open source projects. These files were dominated by source code and documentation in various formats and languages. In Table 1, α_{rand} represents the overhead index on random data, and α_{real} the overhead index for non-random, real data.

Tables 2 and 3 in Appendix B show the experimental settings and algorithm parameters.

The most striking result of these experiments is

Algorithm	α_{rand}	α_{real}
BSW	2.04	2.44
BFS	1.98	2.07
TD	1.77	1.79
SCM	1.51	1.73
TTTD	1.51	1.52

Table 1: The overhead indexes for random and non-random data

that the performance of all the algorithms is worse on real data compared to random data. This difference is most pronounced with the algorithms without an upper size threshold, namely BSW and SCM. On investigating the cause of this worsening performance, we found that it is caused by files that have strongly repetitive patterns in them. Such repetitive patterns significantly increase the variance of chunk sizes within the file, leading to large chunking overhead.

The best performing algorithm above, TTTD, does almost as well on real data as it does on random data. This validates our intuition that using backup breakpoints in combination with minimum and maximum thresholds essentially removes the weak points of the other algorithms.

6.6 Example

Let us compare the two algorithms BSW and TTTD, when used for LBNFS. For the sake of comparison, let us use the parameter values specified in Table 3. These parameter values give average chunk sizes(on non-random data) of 1007 and 1015 for the two algorithms, i.e. average chunk sizes are very close. Now consider two text files, F and F', one on the server, and one on the client, where F' has come about by editing F, i.e. removing a section of F and replacing it with new text. Our goal is to transfer F' to the server using the LBNFS scheme, i.e chunking both of them, sending the hashes, and transferring the chunks of F' that don't exist in F. Now consider two scenarios: in one we use BSW as the chunking algorithm, in the other we use TTTD. In both scenarios we need to transmit the meta-data, i.e. the sequence of hashes of F' chunks. The size of the hash sequence

is roughly equal in the two scenarios. However, in both cases there is an overhead, i.e. a number of bytes that need to be transferred not because they are new data, but because chunk boundaries do not in general correspond to the beginning and end of the edited sequence. The size of this overhead is $\alpha \times \mu$, where α is the overhead index and μ is the average chunk size. Thus in the case of BSW the size of the overhead is $1007 \times 2.44 = 2457$, whereas in the case of TTTD the size of the overhead is $1015 \times 1.52 = 1543$. So BSW's overhead is 1.6 times TTTD's overhead.

Keep in mind that this overhead has to be paid for every contiguous edit in the file; thus if the file is edited in two places the amount of overhead is doubled.

LBNFS uses a variant of the BSW algorithm with very small (relative to average chunk size) minimum chunk size and very large maximum chunk size. Thus from a practical point of view its performance is the same as BSW. By switching to TTTD LBNFS could significantly reduce the amount of overhead bytes it needs to transmit between the client and the server.

7 Conclusion

We developed an analytic framework for evaluating chunking algorithms, and found that the existing algorithms in the literature, namely BSW, BFS and SCM, perform poorly on real data. We introduced a new algorithm, TTTD, which performs much better than all the existing algorithms, and also puts an absolute size limit on chunk sizes. Using this algorithm can lead to a real improvement in the performance of applications that use content based chunking.

References

[1] F. Dabek et. al. Wide-area cooperative storage with CFS, Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01), Chateau Lake Louise, Banff, Canada, October 2001

[2] U. Manber. Finding similar files in a large file system. In Proceedings of the Winter 1994 USENIX Technical Conference, San Francisco, CA, January 1994.

[3] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01), pages 174–187, Chateau Lake Louise, Banff, Canada, October 2001

[4] S. Quinlan and S. Dorward. Venti: a New Approach to Archival Storage. In First USENIX Conference on File and Storage Technologies, pages 89–102, Monterey, CA, 2002.

[5] M.O. Rabin. Fingerprinting by Random Polynomials. Tech. Rep. TR-15-81, Center for Research in Computing Technology, Harvard Univ., Cambridge, Mass., 1981. Tang,

[6] Tang, Hsiu-Khuern and Eshghi, Kave . The storage overhead in hash-based file-chunking algorithms. HP Labs Technical Report 2004.

Appendix A TTTD Algorithm

The C version of TTTD is shown below. Here, `Tmin` and `Tmax` are the minimum and maximum size thresholds, `D` is the main divisor and `Ddash` is the backup divisor. `input` is the input stream, `endOfFile(input)` is a boolean function that returns true when we reach the end of the input stream, `getNextByte(input)` gets the next byte in the input stream and `updateHash(c)` moves the sliding window, updating the hash by adding `c` and removing the character that is now not in the window. `addBreakpoint(p)` adds `p` to the sequence of breakpoints. `p` is the current position, and `l` is the position of the last break point.

```
int p=0, l=0,backupBreak=0;
for (;!endOfFile(input);p++){
  unsigned char c=getNextByte(input);
  unsigned int hash=updateHash(c);
  if (p - l<Tmin){
    //not at minimum size yet
    continue;
  }
  if ((hash % Ddash)==Ddash-1){
```

```
    //possible backup break
    backupBreak=p;
  }
  if ((hash % D) == D-1){
    //we found a breakpoint
    //before the maximum threshold.
    addBreakpoint(p);
    backupBreak=0;
    l=p;
    continue;
  }
  if (p-l<Tmax){
   //we have failed to find a breakpoint,
   //but we are not at the maximum yet
   continue;
  }
  //when we reach here, we have
  //not found a breakpoint with
  //the main divisor, and we are
  //at the threshold. If there
  //is a backup breakpoint, use it.
  //Otherwise impose a hard threshold.
  if (backupBreak!=0){
    addBreakpoint(backupBreak);
    l=backupBreak;
    backupBreak=0;
  }
  else{
    addBreakpoint(p);
    l=p;
    backupBreak=0;
  }
}
```

Appendix B Experimental Setup

The experimental set up was as follows:

For each file in the collection of files, run the following sequence of steps n times, where n is an input to the experiment:

1. Choose a random number x uniformly from the range $[0, fileSize]$. x is the location of the modification in the file.

2. Choose a random number *minus* from the range *deleteRange*. *minus* is the number of bytes to be deleted from the file at location x. *deleteRange* is an input for the experiment

3. Choose a random number *plus* from the range *insertRange*. *plus* is the number of bytes to be inserted into location x after the deletion. *insertRange* is an input parameter for the experiment.

4. For each algorithm A, compute $V^A(F, F')$, where F is the sequence of bytes in the file, and F' is derived from F by deleting *minus* bytes from position x and inserting *plus* random bytes in the same position. Record all the other relevant statistics for this file as well.

Table 2 shows the experimental settings.

window	*deleteRange*	*insertRange*	iter
50	[1000,3000]	[1000,3000]	100

Table 2: the settings for the experiments

Table 3 show the parameter values for the various algorithms, and the resulting average chunk sizes for random and non-random data. In this table μ_{real} and μ_{rand} represent the average chunk sizes for real and random data. The resulting overhead indexes were previously presented in Table 1

Alg.	D	D'	L	T	μ_{rand}	μ_{real}
BSW	1000	∞	0	∞	1004	1007
BFS	1000	∞	0	2800	942	936
TD	1200	600	0	2150	967	961
SCM	540	∞	460	∞	993	1040
TTTD	540	270	460	2800	983	1015

Table 3: Algorithm parameters and the resulting average chunk sizes