SyncCoding: A Compression Technique Exploiting References for Data Synchronization Services

Wooseung Nam
ECE, UNIST
Ulsan, South Korea
wsnam@unist.ac.kr

Joohyun Lee
ECE, OSU
Columbus, OH, USA
lee.7119@osu.edu

Kyunghan Lee
ECE, UNIST
Ulsan, South Korea
khlee@unist.ac.kr

***Abstract*—In this work, we raise a question on why the abundant information previously shared between a server and its client is not effectively utilized in the exchange of a new data which may be highly correlated with the shared data. We formulate this question as an encoding problem that is applicable to general data synchronization services including a wide range of Internet services such as cloud data synchronization, web browsing, messaging, and even data streaming. To this problem, we propose a new encoding technique, SyncCoding that maximally replaces subsets of the data to be transmitted with the coordinates pointing to the matching subsets included in the set of relevant shared data, called *references*. SyncCoding can be easily integrated into a transport layer protocol such as HTTP and enables significant reduction of network traffic. Our experimental evaluations of SyncCoding implemented in Linux shows that it outperforms existing popular encoding techniques, Brotli, LZMA, Deflate, and Deduplication in two practical use networking applications: cloud data sharing and web browsing. The gains of SyncCoding over Brotli, LZMA, Deflate, and Deduplication in the encoded size to be transmitted are shown to be about 12.4%, 20.1%, 29.9%, and 61.2% in the cloud data sharing and about 78.3%, 79.6%, 86.1%, and 92.9% in the web browsing, respectively. The gains of SyncCoding over Brotli, LZMA, and Deflate when Deduplication is applied in advance are about 7.4%, 10.6%, and 17.4% in the cloud data sharing and about 79.4%, 82.0%, and 83.2% in the web browsing, respectively.**

I. Introduction

During the last decade, cloud-based data synchronization services for end-users such as Dropbox, OneDrive, and Google Drive have attracted a huge number of subscribers and have enabled a new life style of carrying no physical storage device. These new services are now regarded as one of the indispensable applications of the Internet. However, looking back the history, data synchronization services had been utilized in various fields including state-variable synchronization in distributed systems, scientific data synchronization among research centers, and sensor data synchronization between sensors and a data sink. Also, if we define a data synchronization service in a more generalized manner, as a data transfer service that manages two systems to have a set of identical data, the scope of data synchronization services surprisingly expands. Such a broadening of the definition allows web browsing, file downloading, and even multimedia streaming classified into data synchronization services as these progressively synchronize a set of data from a server to a client.

Given this prevalence of data synchronization services consuming a huge portion of the Internet bandwidth, we raise one question: *"how can the previously synchronized data be exploited for the synchronization of a new data?"* To be more specific, the question can be asked again as follows: "how can Dropbox make use of previously synchronized files of tens of gigabytes for sharing a newly added file?", "how can Chrome browser utilize previously visited webpages and the contents therein to open up a new webpage?", or "how can a satellite utilize the previously shared observation data to send down a new observation data to a ground station?"

To our best knowledge, interestingly, this question has been under-explored in the literature and has not been clearly answered by a practical system. *Index code* [1], that suggested the concept of encoding a block of data with its relations to other blocks of data, was of its first kind, but the selection process of a subset of blocks to improve efficiency was out of its scope. *Deduplication* [2], which finds duplicated data chunks in a storage system for the elimination of redundancy, is highly correlated but because Deduplication mostly works in the level of files or bit chunks of a fixed size, its ability to exploit the previously shared data is highly limited.

In this paper, we systematically answer our question by proposing a new data compression technique called *SyncCoding* that can be used widely for data synchronization services and can be easily integrated into transport layer protocols such as HTTP and TCP for the reduction of network data traffic. The intuition behind SyncCoding is to quickly choose a set of data from the previously synchronized data, which holds high similarity with the new data to be synchronized, and to encode the new data by intelligently fragmentizing it to bit sequences that can be referenced from the chosen set. By its nature, SyncCoding works very effectively with the types of data that are created on similar topics, directed by similar formats, or authored by the persons of similar writing styles. We find it interesting that a large portion of data being handled by aforementioned data synchronization services such as web pages of a website authored by a programmer, documents on the same topic collected in a folder of a cloud storage, and technical reports under a given format fall into the category where SyncCoding can be effective.

SyncCoding not only paves a road to the ideal utilization of the previously synchronized data but also gives a quantitative answer on how much helpful the synchronized data can be for compressing a data. In order to do so, we take the following steps: 1) we revisit the algorithm of LZMA (Lempel-Ziv-Markov chain algorithm) [3], the core of *7-zip* compression format [4] that is known as one of the most advanced data

978-1-5090-6501-1/17/$31.00

compression technique and reveal how it works in detail, 2) we design the work flow of SyncCoding by the analogy with LZMA under the existence of previously synchronized data, called potential references, 3) we analyze in what conditions SyncCoding outperforms LZMA in the size of compressed data and suggest practical heuristic algorithms to select actual references from the potential references in order to meet the conditions, and 4) we implement and validate SyncCoding in a Linux system and evaluate its compression characteristics in realistic use cases of cloud data sharing and web browsing.

Our extensive evaluation of SyncCoding in the cloud data sharing scenario with a dataset of RFC (Request For Comments) technical documents reveals that SyncCoding compresses on average a document about 10.6%, 20.1%, and 61.2% more compared to LZMA after Deduplication, without Deduplication, and only Deduplication, respectively. Our further evaluation of SyncCoding in the web browsing scenario shows that SyncCoding outperforms commercial web speed-up algorithms, Brotli [5] and Deflate [6] after Deduplication and without Deduplication by 79.4% and 83.2%, and 83.9% and 87.0%, respectively, in the size of compressed webpages of CNN. We find that this substantial gain captured by SyncCoding comes from the similar programming style maintained over the webpages of a website and confirm that the gain is persistent over various websites such as CNN and NY Times.

II. RELATED WORK

Reforming a given bit sequence with a new bit sequence to reduce the total number of bits is called data compression and it is also known as source coding. When the encoded bit sequence can be perfectly recovered to the original bit sequence, it is called *lossless* compression which is of our interest. A bit sequence is equivalent to, hence interchangeable with, a symbol sequence where a symbol is defined by a block of bits which repeatedly appears in the original bit sequence (e.g., ASCII code). Shannon's source coding theorem [7] tells that a symbol-by-symbol encoding becomes optimal when symbol i that appears with the probability p_i in the symbol sequence is encoded by $-\log_2 p_i$ bits. It is well known that Huffman coding [8] is an optimal encoding for each symbol but is not for a symbol sequence. Arithmetic coding [9] produces a near-optimal output for a given symbol sequence.

However, when the unit for encoding goes beyond a symbol, the situation becomes much more complicated. An encoding with blocks of symbols that together frequently appear may reduce the total number of bits, but it is uncertain what will be the optimal block sizes that give the smallest encoded bits. Therefore, finding the real optimal encoding for an arbitrary bit sequence becomes NP-hard [10] by the exponential complexity involved in testing the combinations of the block sizes.

LZ77 [11], the first sliding window compression algorithm, tackles this challenge by managing dynamically-sized blocks of symbols within a given window (i.e., the maximum number of bits that can be considered as a block) by a tree structure. In a nutshell, LZ77 progressively puts the symbols to the tree as it reads symbols and when there is a repeated block of symbols found in the tree, it replaces (i.e., self-cites) the block with the distance to the block and the block length. This process lets LZ77 compress redundant blocks of symbols.

Deflate [6] combines LZ77 and Huffman coding. It replaces matching blocks of symbols with length-distance pairs similarly to LZ77 and then further compresses those pairs using Huffman coding. LZ78 and LZMA are variants of LZ77, of which their encoding methods for length-distance pairs are improved. LZMA is the algorithm used in 7z format of the 7-zip archiver. We will later discuss about the operations of LZMA in detail in Section III.

Unlike aforementioned compression algorithms, there exist several techniques that include external information in addition to the source data for encoding. Index code [1] generalizes the broadcast problem with the existence of side information at the receivers and analyzes the properties of the optimal encoding of a new data block under the given side information denoted as a graph that captures the relations among the data blocks. Index code is still far from SyncCoding because it gives no attention on how to acquire the side-information, which is challenging in practice.

There are simpler ways of exploiting external information such as Star encoding (*-encoding) [12] that uses an external static dictionary shared between a server and its client. A similar yet a more efficient approach has been made at Length Index Preserving Transform (LIPT) [13] with an English dictionary having about 60,000 words. *Brotli* [5], one of the latest encoding technique, has a pre-defined shared dictionary of about 13,000 English words, phrases, and other sub-strings extracted from a large corpus of text and HTML documents. Brotli is known to achieve about 20% of compression gain over Deflate in the encoding of webpages in a web browser [14]. Exploiting a static shared dictionary is useful in general, but its efficacy is highly limited as each replacement is bounded by the length of words.

Deduplication [2] is an existing redundancy elimination technique for file systems, which replaces repeated data chunks of a file with the matching chunks of other files in the system and is also used to reduce network traffic using concurrent data [15]. It is widely used and shown to be effective in large-scale storage systems and cloud storage services as their users are observed to store a wide variety of redundant files such as program packages and video files of high popularity. For instance, a cloud storage service run by Google [16] treats a file to upload to the cloud as uploaded instantly when the same file of the same hash value exists somewhere in the cloud storage. Dropbox is also known to use Deduplication in conjunction with delta encoding [17] that is another popular method for the version management of files as in *Diff* [18]. Because Deduplication mostly focuses on high volume data, it gives little attention to text-oriented data such as documents and webpages, which are relatively small. A popular open-source implementation of Deduplication, OpenDedup [19] also gives its minimum chunk size option for redundancy elimination, starting from 1 kB. 1kB is sufficiently small and detailed for large files but it is way too large for documents.

Unless the documents of interest are exactly the same or are just different versions of the same file, redundancy elimination with 1kB chunk is not realistic. While OpenDedup only deduplicates exactly matching chunks, more recent Deduplication techniques such as [20] can find chunks with differences of a few bytes and are known to be more effective.

III. LZMA PRIMER

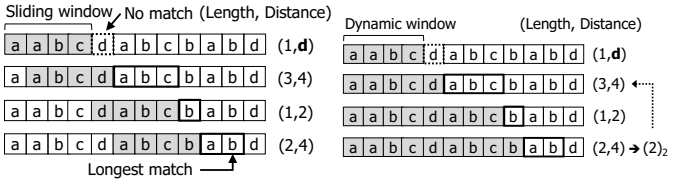
SynCoding is implemented based on LZMA. Therefore, in order to explain how SyncCoding is implemented, we give a short primer of LZ77 and LZMA algorithms.

LZ77 encodes a sequence of symbols by maintaining a sliding window of size w within which the blocks of symbols appeared in the window are systematically constructed as a tree. Since the window is sliding, the blocks of symbols captured in the tree will change as the encoding proceeds. The compression of bits in LZ77 occurs when a repeated block of symbols is replaced with a length-distance pair, where the length and the distance denote the length of the block of symbols and the bit-wise distance from the current position to the position where the same block of symbols appeared earlier within the window. Every time a block of symbol is replaced by a length-distance pair, LZ77 tries to find the longest matching block in the window in order to reduce the number of encoded length-distance pairs as the reduction directly affects the compression efficiency. A sample encoding with LZ77 when the window size is 4 is illustrated in Fig. 1 (a). The window size in LZ77 which is static may bring a performance issue. When the window size is small, the amount of blocks of symbols that can be kept in the window is limited, hence reducing the chances of compression.

LZMA works very similarly to LZ77 but with two major improvements. The first is that LZMA adopts a dynamic window that has its initial size as one and grows as the encoding proceeds. Because the window grows, LZMA does not suffer from being constrained by a small static window size. The second is that LZMA further reduces the number of bits representing a length-distance pair by specifying a few special encoded bits that are used when the current distance is the same with the distances that are most recently encoded. Reusing the distance information with fewer bits helps a lot when the data to compress has a repetitive nature (e.g., repetitive sentences or paragraphs in a file). The look up of the distances is typically done for the last four pairs. A sample encoding with LZMA is depicted in Fig. 1 (b). These small changes cause LZMA can compress data more than LZ77 [21].

The optimality of LZ77 was proved earlier by Ziv and Lempel [22] in the sense that the total number of bits required to encode a data with LZ77 converges to the entropy rate of the data, where the entropy rate is defined with the symbol-by-symbol manner. Since LZMA is more efficient than LZ77, it is not difficult to prove that LZMA also converges to the entropy rate by extending the proof in [22].

Now, our interest lies in how the number of bits required for SyncCoding can be compared with that of LZMA and whether



(a) Encoding of LZ77 (b) Encoding of LZMA

Fig. 1. Sample encoding of (a) LZ77 and (b) LZMA over a sequence of symbols. Whenever a match exists, the longest match is encoded with a length-distance pair. No match lets the symbol be encoded. When there is a distance value repeated recently, LZMA points to it instead of directly encoding it.

it is less or more. To this end, we explain how the number of bits required for LZMA can be mathematically evaluated.

Let $T_{\text{LZMA}}(\{S\}_1^N)$ be the total required bits of the output encoded by LZMA for a given sequence of N symbols $\{S\}_1^N$. Suppose that p_{LZMA} is the number of phrases to be encoded in LZMA, where a phrase is defined by a block of symbols. Note that as encoding progresses, the length of a new phrase (i.e., the number of symbols in the phrase) is determined by the longest matching sub-sequence of symbols that can be found in the sliding window. Then, $T_{\text{LZMA}}(\{S\}_1^N)$ becomes the bits required to encode all the length-distance pairs for the phrases, $\sum_{i=1}^{p_{\text{LZMA}}}\{f(l_i)+g(d_i)\}$, where l_i is the length of phrase i, d_i is the matching distance of phrase i, and $f(l_i)$ and $g(d_i)$ denote the bits to encode l_i and d_i, respectively. The matching distance d_i is the bit-wise distance from the current position to the previous position of the same phrase.

LZMA uses comma-free binary encoding [22] for $f(l_i)$, which is also used in LZ77. The comma-free binary encoding consists of two parts: 1) the prefix and 2) the binary encoding of l_i, denoted by $b(l_i)$. According to [22], the prefix and the binary encoding occupies $2\lceil\log_2\lceil\log_2(l_i+1)\rceil\rceil$ and $\lceil\log_2(l_i+1)\rceil$ bits, respectively. The summation of those quantifies $f(l_i)$ of LZMA.

$g(d_i)$ in LZMA falls into either of the following three cases. When the distance to encode is not the same with any of the four recently used distances, the distance is encoded by the binary encoding of a fixed number of digits which is determined by the size of the sliding window w. Therefore $g(d_i)$ always goes to $\log_2(w)$. There is one exception when $l_i = 1$ (i.e., the phrase consists of a single symbol), the symbol itself is encoded instead of the distance being encoded. Therefore, $g(d_i) = \log_2 C$, where C denotes the size of the symbol space (i.e., character space for a text encoding). When the distance is repeated from the four recently used distances, there exist two bit mappings of 4 bits or 5 bits by the following cases: 1) $g(d_i) = 4$ when the distance matches with the first or the second lastly used distance, 2) $g(d_i) = 5$ when the distance matches with the third or the fourth lastly used distance.

By the equations above, we can estimate the best case of LZMA, that happens when all the distances to encode for the phrases whose length is larger than two are found from the first or the second lastly used distance, i.e., $g(d_i) = 4$. Thus, we have the following lower bound for $T_{\text{LZMA}}(\{S\}_1^N)$.

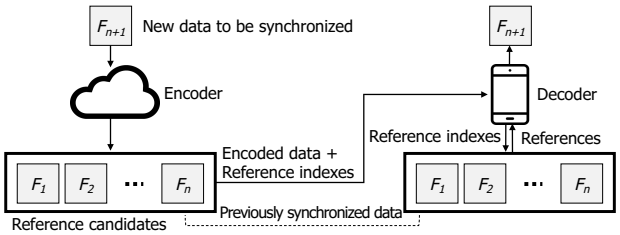


Fig. 2. The concept and basic operations of SyncCoding

Lemma 1 *$T_{LZMA}(\{S\}_1^N)$ is lower bounded by the following minimal possible total number of bits of LZMA:*

$$T_{LZMA}(\{S\}_1^N) \geq p^1_{LZMA} \cdot \lceil \log_2 C \rceil + 4(p_{LZMA} - p^1_{LZMA}) + \sum_{i=1,|l_i|\neq 1}^{p_{LZMA}} \Big(2\lceil \log_2 \lceil \log_2 (l_i+1) \rceil \rceil + \lceil \log_2 (l_i+1) \rceil \Big),$$

where p^1_{LZMA} is the number of phrases whose length is one (i.e., $l_i = 1$).

IV. SYSTEM DESIGN AND ANALYSIS

In this section, we formally state the problem that SyncCoding tackles and proposes the design of SyncCoding. Then, we provide a mathematical analysis for the design and explain how it can be compared with that of LZMA.

A. System Design

Suppose that there exist n files that are previously synchronized between a server and a client, denoted by F_i where $i = 1, \cdots, n$. Upon transmitting $(n+1)$-th file, F_{n+1}, from the server to the client, our problem is to answer how F_{n+1} should be encoded using the shared files, $F_1, \cdots, F_n$. Fig. 2 depicts this scenario in which the encoder (i.e., server) locates in a cloud system and the decoder is of a mobile device.

Given that the number of previously synchronized, we assume that we can somehow choose the most useful k files out of n files and use them only to encode F_{n+1}. We call those chosen files *references* and denote the set of references for F_{n+1} whose cardinality is k as R^k_{n+1}. Let us discuss the methods for choosing such k files in the next section.

For the compression, we let SyncCoding concatenate all the files in R^k_{n+1} to be a single large file and append it at the front part of F_{n+1} to create a virtual file to encode. We denote this virtual file, a compound of the file to encode and its references as V^k_{n+1}. Given V^k_{n+1}, we let SyncCoding simply encode it by LZMA in the hope that all the blocks of symbols that are commonly found in the references and the file to encode get converted to length-distance pairs, hence reducing the bits to encode. Note that when V^k_{n+1} is constructed, we let SyncCoding place the references in the order that a reference with higher usefulness is placed closer to F_{n+1}. Once encoding is done, we cut out the front part and extract only the encoded portion of F_{n+1}, denoted by E^k_{n+1}. SyncCoding transmits E^k_{n+1} to the decoder with the list of file indexes chosen as references, denoted by I^k_{n+1}.

For the decoding of E^k_{n+1}, we let SyncCoding first decode I^k_{n+1} to recall the references at the decoder side. Then, we let SyncCoding create the concatenated file of R^k_{n+1} as if it was done at the encoder and compress it by LZMA. Once we get the output, we append it at the front part of E^k_{n+1} to create a compound and decode the compound by LZMA. By the nature of LZMA, this decoding guarantees the acquisition of F_{n+1} from E^k_{n+1}. The encoding and decoding procedures of SyncCoding is summarized in **Algorithm 1**. We implement SyncCoding of this procedure by modifying an open-source implementation of LZMA [23].

Algorithm 1 Encoding/Decoding Procedures of SyncCoding

Encoding:
1) Choose k useful references R^k_{n+1}, and index them by I^k_{n+1}
2) Sort the references in R^k_{n+1} in the reverse order of usefulness
3) Concatenate all the references in R^k_{n+1}
4) Append it at the front of F_{n+1} to get V^k_{n+1}
5) Encode V^k_{n+1} by LZMA and cut out the encoded file E^k_{n+1}
6) Transmit E^k_{n+1} and I^k_{n+1}

Decoding:
1) From I^k_{n+1}, restore the concatenated file made up of R^k_{n+1}
2) Compress it by LZMA
3) Append the compressed file at the front of E^k_{n+1}
4) Decode the compound by LZMA and cut out to obtain F_{n+1}

B. Comparative Analysis

We analyze SyncCoding by comparing its total number of bits for encoding, $T_{\text{SC}}(\{S\}_1^N)$, with that of LZMA. Recall that the input is again $\{S\}_1^N$, a sequence of N symbols, which was identically used for LZMA. By the analogy with the analysis of LZMA, we can view that $T_{\text{SC}}(\{S\}_1^N)$ conforms to $\sum_{i=1}^{p_{\text{SC}}} \{f(l_i) + g(d_i)\} + k \log_2 n$, where p_{SC} denotes the number of phrases to be encoded in SyncCoding. $k \log_2 n$, the overhead of SyncCoding, quantifies the number of bits to list the indexes of the references. Since SyncCoding adopts LZMA for its bit encoding, $f(\cdot)$ and $g(\cdot)$ for SyncCoding are not different from those in LZMA. Note that the number of phrases identified in SyncCoding is always smaller than or at least equal to that in LZMA mainly because the references give a more abundant source of matching phrases. Therefore, the better the reference selection, the more the gap between p_{SC} and p_{LZMA}. It is also obvious that $p^1_{\text{SC}} \leq p^1_{\text{LZMA}}$, where p^1_{SC} denotes the number of phrases of length one in SyncCoding.

We now find the condition that guarantees better compression for SyncCoding over LZMA, so $T_{\text{SC}}(\{S\}_1^N) < T_{\text{LZMA}}(\{S\}_1^N)$ is satisfied. For that, we compare the worst case bit-size of SyncCoding with the best case bit-size of LZMA. Suppose that SyncCoding reduces the number of phrases by the factor of γ as $p_{\text{SC}} = \gamma \cdot p_{\text{LZMA}}$, where γ is a constant satisfying $0 < \gamma \leq 1$. It is unlikely, but if the reference selection goes extremely wrong, it is possible to have $\gamma = 1$. Having a smaller number of phrases that is to encode a smaller number of length-distance pairs is the key factor of reducing bits to encode for SyncCoding. However, this brings a side effect that is the increment in the average phrase length. Note that the ratio between numbers of phrases in LZMA and SyncCoding, γ, affects the average phrase length because the following holds: $\bar{l}_{\text{SC}} \cdot p_{\text{SC}} = N$, where

$\bar{l}_{\mathrm{SC}}$ is the average phrase length in SyncCoding. Therefore, the average phrases length in SyncCoding increases by the factor of $1/\gamma$ compared to LZMA as in $\bar{l}_{\mathrm{SC}} = \bar{l}_{\mathrm{LZMA}}/\gamma$, where $\bar{l}_{\mathrm{LZMA}}$ is the average phrase length in LZMA. Also, there is another side effect that is the increment in the distance of a length-distance pair. This increment may request more bits to encode the distance. The largest increment in bits comes from the case when a phrase finds its match from the farthest reference (i.e., the reference appended at the very beginning). Thus, this largest bit increment is affected by the number of references and is bounded by $\log_2 k$ bits. Under this setting, we derive an upper bound of the bit-size of SyncCoding by assuming possible worst cases in combination as follows: 1) the distance to encode in each length-distance pair is either not found from any of the four lastly used distances or not of the length one, 2) the phrases to encode whose length is one are fully remove by using the references, say $p^1_{\mathrm{SC}} = 0$. The condition 1) makes each distance to be encoded by the binary encoding, so $g(d_i) = \lceil \log_2 N \rceil$ holds. The condition 2) makes a phrase always encoded by a length-distance pair instead of being encoded by the symbol space, whose bit consumption $\log_2 C$, is typically much smaller than $g(d_i) = \lceil \log_2 N \rceil$. These arguments with the Jensen's inequality[1] let us conclude that $T_{\mathrm{SC}}(\{S\}_1^N)$ is upper bounded by the following lemma.

Lemma 2 *$T_{SC}(\{S\}_1^N)$ is upper bounded by the following maximal total number of bits:*

$$T_{SC}(\{S\}_1^N) \le k\lceil \log_2 n \rceil + \gamma \cdot p_{LZMA} \cdot (\lceil \log_2 N \rceil + \lceil \log_2 k \rceil) + \gamma \cdot p_{LZMA} \cdot \Big(2\lceil \log_2 \lceil \log_2 (\bar{l}_{LZMA}/\gamma + 1) \rceil \rceil + \lceil \log_2 (\bar{l}_{LZMA}/\gamma + 1) \rceil \Big).$$

By using the lemmas 1 and 2, the condition, $T_{\mathrm{SC}}(\{S\}_1^N) < T_{\mathrm{LZMA}}(\{S\}_1^N)$, gives the following theorem.

Theorem 1 *If $h(\gamma) > 0$ is satisfied for the following definition of $h(\gamma)$, SyncCoding always compresses the same sequence of symbols more than LZMA.*

$$h(\gamma) = \alpha - \gamma \cdot p_{LZMA} \cdot \Big(\beta + \lceil \log_2 (\bar{l}_{LZMA}/\gamma + 1) \rceil + 2\lceil \log_2 \lceil \log_2 (\bar{l}_{LZMA}/\gamma + 1) \rceil \rceil \Big), \quad (1)$$

where α and β denote $\sum_{i=1,|l_i|\neq 1}^{p_{LZMA}} \big(2\lceil \log_2 \lceil \log_2 (l_i + 1) \rceil \rceil + \lceil \log_2 (l_i + 1) \rceil \big) + p^1_{LZMA} \cdot \lceil \log_2 C \rceil + 4(p_{LZMA} - p^1_{LZMA}) - k\lceil \log_2 n \rceil$ and $\lceil \log_2 N \rceil + \lceil \log_2 k \rceil$, respectively.

It is complex to find the solution for γ that guarantees $h(\gamma) > 0$, but it is not difficult to show numerically that there exists $\gamma < 1$ satisfying $h(\gamma) > 0$. Also, it is trivial that $h(\gamma) > 0$ if γ approaches to zero. This implies that selecting references that effectively reduces the number of phrases to encode is the key for SyncCoding to be superior than LZMA.

C. Questions on SyncCoding

As it is revealed by the analysis, the efficacy of SyncCoding over LZMA highly depends on how much SyncCoding can reduce the amount of length-distance pairs to encode. The ratio of reduction, γ, is the outcome of the reference selection. The question on which selection of a set of references from the synchronized data whose volume may be huge is the most efficient selection, brings the subsequent questions: 1) *which data in the synchronized data helps the most?*, 2) what is the size of the set of references that leads to the best compression?, and 3) *how long does it take for SyncCoding to encode and to decode a file with the chosen references (i.e., encoding and decoding complexity)?*

It is essential to answer these questions to make SyncCoding viable in general data synchronization services, but answering each of the questions is of a challenge. By the complexity involved in the symbol tree construction in LZMA and also by the correlated nature of symbols in the input sequence of symbols (e.g., language characteristics and intrinsic data correlation), none of the three questions can be tackled analytically. In the next section, we empirically characterize SyncCoding and give heuristic answers to these questions.

V. Characterization of SyncCoding

A. Reference Selection

We first tackle the question on the reference selection. As it was intuitively explained in the system design, it is obvious that a file containing high similarity with the target file to encode is preferred to be included in the set of references. However, given that SyncCoding as well as LZMA tries to minimize the number of length-distance pairs to encode by seeking for the longest matching subsequence of symbols, it is unclear how this similarity between files in the context of encoding can be defined. One definition rooted from the usefulness as a reference can be *the total length of matching subsequences included in the reference give a target file to encode.* The more the matching subsequences and the longer the matching subsequences, this definition gives a higher similarity value. However, this definition is practically impaired as its measurement itself takes time as much as the encoding process takes, so it is not so different from quantifying how much additional compression is obtained in SyncCoding by having the reference afterwards.

In order to secure the practicality, we need a much lighter similarity measure that can quickly investigate the individual usefulness of all the previously synchronized files with respect to the target file to encode. For this, we borrow the concept of document similarity, which has been widely used in the machine learning field with various implementations such as cosine similarity [24] and Kullback–Leibler divergence [25]. Based on such similarity measures, we propose a modified cosine similarity measure. Our modified cosine similarity denoted by $\mathrm{sim}(A, T)$ between two files, a reference candidate A and the target file to encode T, is formally defined as follows:

$$\mathrm{sim}(A, T) \triangleq \frac{\sum_{i \in S(T)} f(t_i^A) f(t_i^T)}{\sqrt{\sum_{i \in S(T)} f(t_i^A)^2} \sqrt{\sum_{i \in S(T)} f(t_i^T)^2}}, \quad (2)$$

[1] For a random variable X and a concave function g, $\mathbb{E}[g(X)] \le g(\mathbb{E}[X])$ holds. Such g includes $\log_2$ function.

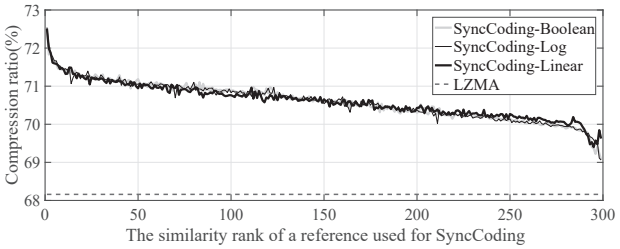


Fig. 3. The compression ratios of LZMA and SyncCoding with one reference whose modified cosine similarity is ranked by either of Boolean, Log, and Linear. Overall, SyncCoding with a single reference shows higher compression ratios than LZMA and a reference of a higher rank achieves a better compression ratio.

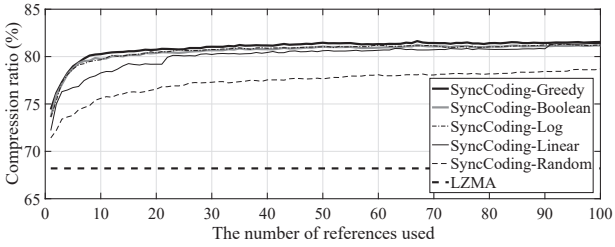


Fig. 4. The compression ratios of SyncCoding with increasing number of references that are selected randomly, by a greedy search, or by the modified cosine similarity rank with either of Boolean, Log, or Linear. In this figure, the overhead for reference indexing is not considered to focus on understanding the impact of reference selection.

where $S(T)$ is the set of distinct symbols $\{t_i\}$, observable from T, t_i^A and t_i^T are the frequencies of observing the symbol t_i in the file A and T, and $f(\cdot)$ is a transformation function. By definition, $t_i^T \geq 1$ and $t_i^A \geq 0$ hold.

In order to validate the efficacy of the proposed similarity measure, we randomly chose and downloaded a research paper from Google scholar [26] with a keyword "wireless networking", which is arbitrarily chosen. To imitate the database of previously synchronized data for the chosen document, we have also downloaded three hundreds of research papers that came up with the same keyword as candidate references. With this sample data set, we rank the candidate references with the modified cosine similarity of three different $f(\cdot)$ transformation functions for the chosen document: 1) Linear: $f(t_i) = t_i$, 2) Log: $f(t_i) = \log(t_i + 1)$, and 3) Boolean: $f(t_i) = 1$ for $t_i > 0$ and $f(t_i) = 0$ for $t_i = 0$. We depict the compression ratio of SyncCoding with different $f(\cdot)$ for each candidate reference sorted by its similarity rank in Fig. 3 in comparison with LZMA that uses no reference. Note that the compression ratio is the fraction of the compressed amount over the size of the original file, where the compressed amount is the difference between the size of the original file and the compressed file. Fig. 3 shows that SyncCoding with either of three functions compresses the chosen document more than LZMA. Especially with the reference candidate of the highest similarity rank, SyncCoding-Boolean achieves about 72.6% compression ratio meaning that the compressed size is only 27.4% of the original size. Comparing this result with that of LZMA which achieves the compression ratio of 68.2% and results in the compressed file whose size is 31.8% of the original, SyncCoding reduces the size of the compressed file by about 13.9% only with one well-chosen reference. Also, as Fig. 3 shows, SyncCoding with either of three functions maintains non-decreasing tendency over the reference candidates sorted by the rank. This implies that it is acceptable to use the modified cosine similarity rank for a quicker selection of a reference.

Fig. 4, where we increasingly add references for SyncCoding by the similarity rank measured by either of three functions, further investigates the efficacy of using the modified cosine similarity in the reference selection. In Fig. 4, we also include, for comparison, the compression ratios from a greedy search where the reference that maximally improves the compression ratio out of all remaining references is added to the existing set of references and from a random addition. Note that Fig. 4 only takes the size of the compressed amount into account when evaluating the compression ratio and does not consider the overhead of indexing the references, which will be discussed in the next subsection. As shown in Fig. 4, SyncCoding-Boolean performs better than others at least slightly and achieves the closest performance to the greedy search. Given that the computational complexities of the greedy search and SyncCoding-Boolean are $O(N^2)$ and $O(N)$, respectively[2], it is reasonable to conclude that SyncCoding-Boolean is a viable solution to the reference selection problem. Throughout this paper, we use SyncCoding-Boolean as our default SyncCoding implementation.

B. Maximum Compression Efficiency

We now tackle the second question on the maximum compression advantage of SyncCoding over LZMA. It is of particular interest in the cases where the network bandwidth to deliver the compressed data is severely limited. The cases not only include extreme situations such as deep sea communication, inter-planet communication, but also include networks with high link cost such as satellite communication while being at an ocean cruise or at an airplane. In a different perspective, it is also of strong interest in the cases where even a small amount of additional compression gives huge benefit. A nice example is found in inter-data center synchronization in which tens of terabytes are easily added daily and need to be synchronized (e.g., 24 terabytes of new videos are added to YouTube daily [27]).

If there is no overhead of listing the indexes of the references used for encoding, it is obvious that adding a new reference keeps improving the compression ratio of SyncCoding although the gain achieved by each addition may keep diminishing as shown in Fig. 4. However, SyncCoding requires the indexes to be independently encoded and transmitted

[2]SyncCoding-Boolean incurs the complexity of evaluating N reference candidates linearly, where as the greedy search incurs the complexity of $\sum_{j=N}^{N-k+1} j$ in order to find out the most helpful reference at every addition. The optimal can be obtained by a full search, but incurs $O(N!)$.

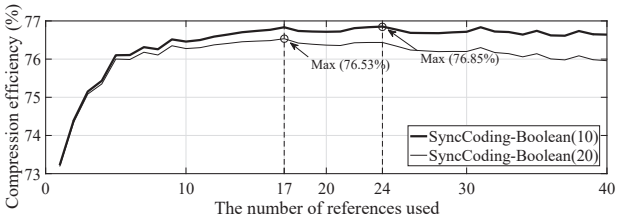


Fig. 5. The compression efficiency of SyncCoding for an increasing number of references. The per-index overhead is chosen as either of 10 or 20 bytes.

along with the main data. We simply let SyncCoding use the address space of ten bytes, that is of 80 bits. This size of address (i.e., index size) gives a pointer that can specify a file from a database with about 10^8 files. It is relatively a large number for a personal use, but in the case of a global data center, it can be extended to twenty bytes (160 bits) or more to index the files with active accesses. To characterize the impact of the overhead from the indexes in SyncCoding, we depict SyncCoding-Boolean with two index sizes, considering the overhead added to the size of the compressed file in Fig. 5. To avoid confusion, we define *compression efficiency* as the compression ratio evaluated with the compressed amount including the overhead, i.e., the ratio between the compressed amount plus overhead and the original file size. For simplicity, we quantify the overhead by the address space size multiplied with the number of references used, meaning that no additional encoding is applied for the indexes. As shown in Fig. 5, with 10 and 20 bytes overhead per index, SyncCoding achieves about 76.85% and 76.53% as its maximum compression efficiency for the chosen document, respectively. The number of references that achieves the maximum compression efficiency is 24 and 17, confirming the intuition that a larger per-index overhead makes the compression efficiency saturated earlier with respect to the number of references used. However, even with a larger per-index overhead, the maximum compression efficiency achieved does not change much. This is because the referencing happens mostly from a small number of highly similar files. With 10 bytes per-index overhead, we further obtain the results about the maximum compression efficiency by repeatedly applying SyncCoding to a randomly selected document from our research paper dataset with all the unselected documents used as reference candidates for one hundred times. In Fig. 6, we depict the maximum compression efficiency from 100 input documents in the ascending order and for the same input document we also plot the compression efficiency achieved with 24 references, the optimal number of references in Fig. 5. As shown in Fig. 5, SyncCoding works well enough with only 24 references and the gain of using more references is less than 1.5%. Therefore, we opt to use $k^* = 24$ as out default number of references for all the following experiments, otherwise it is specified.

C. Encoding Time and Decoding Time of SyncCoding

We tackle the last question on the the encoding and the decoding time of SyncCoding by performing experiments. We

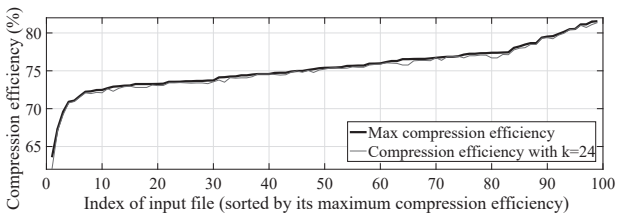


Fig. 6. Maximum compression efficiencies of SyncCoding obtained from 100 randomly chosen documents are compared with the compression efficiencies of SyncCoding that use only 24 references. Only little gap exists.

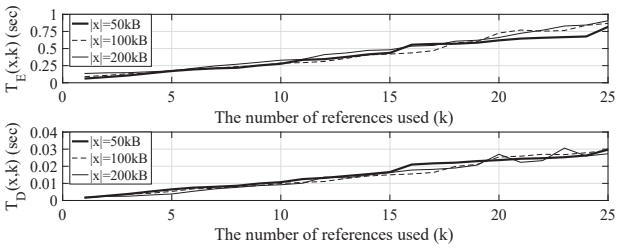


Fig. 7. Experimental evaluation of $T_E(x,k)$ (top) and $T_D(x,k)$ (bottom), the time durations to encode and to decode a file x with k references under Intel i7-3770 CPU (3.40 GHz).

let $T_E(x,k)$ and $T_D(x,k)$ denote the time durations taken for encoding and decoding a file x with k references. Because the encoding and the decoding complexities of SyncCoding with k references are not largely different from the complexity of LZMA repeated by k times, it is expected that $T_E(x,k)$ and $T_D(x,k)$ may increase linearly as k increases for a given x. Fig. 7, a measurement on Linux (Kernel 2.6.18-238.el5) over Intel i7-3770 CPU (3.40 GHz) for three kinds of research papers of about 50, 100, and 200 kB, randomly chosen from the aforementioned dataset, confirms that the average encoding time as well as the average decoding time from one hundred trials increases almost linearly to k. Fig. 7 also confirms that the size of x has little impact to the times because the size of the data to encode is relatively smaller than the total size of the references. The decoding time is in the scale of milliseconds and is relatively negligible compared to the encoding time which is in the scale of a second. One important thing to note here is that the encoding time can often be hidden to users by the following reasons: 1) the existence of a powerful encoding server, 2) the parallelism between the encoding process and the network transmission process, and 3) preprocessing of SyncCoding in the server. We will explain more about the applicability of the preprocessing of SyncCoding to practical use cases in Section VI.

VI. EVALUATION

We evaluate the efficacy of SyncCoding in two real data synchronization services: 1) cloud data sharing and 2) web browsing. The scenario we consider for the cloud data sharing is to synchronize a new file of an existing folder from the storage server to a user device, given that the folder already includes about a hundred of files relevant to the new file. The use case we consider for the web browsing is to browse webpages of a website at a user device given that the webpages

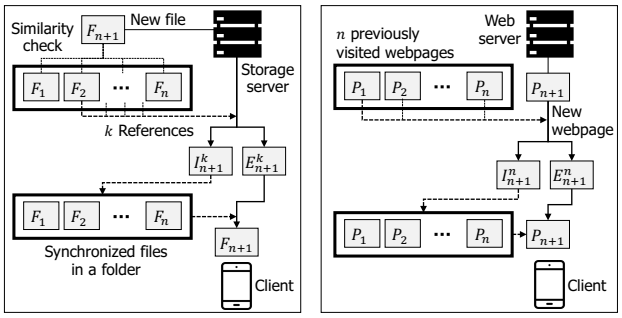


Fig. 8. Overview of the evaluation scenarios of two use cases: 1) cloud data sharing (left) and 2) web browsing (right).

visited up to a moment are all cached in the device, so the web server can exploit those cached pages for encoding a new page. The overview of these scenarios is depicted in Fig. 8.

We experiment both scenarios and statistically compare the compression efficiency of SyncCoding with existing encoding techniques, Brotli, Deflate, LZMA, and Deduplication, whose settings are described in the next subsection. Here we focus on the compression efficiency without being concerned about the encoding and the decoding time, in order to give our focus to the reduction of network data traffic. As it is discussed in the previous section, applying SyncCoding on the fly takes time. Therefore, SyncCoding may not be effective in speeding up web browsing experiences especially for the web servers with insufficient computational capability. However, SyncCoding is still useful to the users who would like to browse web pages with minimal cellular data cost. In the case of cloud data sharing where the users are less sensitive to the synchronization delay, the processing time for the SyncCoding can be successfully hidden to its users.

A. Settings for the Encoding Techniques in Comparison

SyncCoding: We use SyncCoding-Boolean with $k^* = 24$ references unless it is specified and use the per-index overhead of 10 bytes. For the parameters inherited from LZMA implementation, we adopt the values from LZMA with its maximum compression option.

LZMA: For the evaluation of LZMA, we use its SDK (Software Development Kit) provided in [23] with the parameters from the maximum compression option.

Deflate: For the evaluation of Deflate, we use [28], a popular open source library including Deflate with all the parameters from the maximum compression option.

Brotli: For Brotli, we use an open source implementation of Brotli [29], which is embedded in Google Chrome web browser [30]. We also use its maximum compression option.

Deduplication: For the evaluation of Deduplication, we modify OpenDedup [19] so as to investigate its ideal Deduplication performance for documents. We reduce the lower bound of the chunk size (i.e., 1kB in OpenDedup) to be arbitrarily small.

B. Use Case 1: Cloud Data Sharing

We emulate a folder of a cloud storage (e.g., Dropbox) by creating a folder with files of a similar attribute. To fill the folder, we randomly downloaded two hundreds RFC documents from [31], which are all in the format of TXT.

For the evaluation of SyncCoding and other encoding techniques except Deduplication, we regard a randomly chosen file from the folder as the target file to encode for synchronization and assume that all other files in the folder are reference candidates. We perform the following three tests and evaluate the compression efficiencies of SyncCoding and other techniques: 1) Tests for the target documents of various sizes with k^* references, 2) Tests for a randomly chosen document with various numbers of references, 3) Tests for 50 randomly chosen target documents with k^* references. In test 1), for each size of the target document, we select and test 20 documents whose size ranges from 90% to 110% of the given size. Fig. 9 summarizes the results of these tests. Fig. 9 (a) shows the average compression efficiencies with 90% confidence intervals for different sizes of documents to encode and reveals that SyncCoding persistently outperforms others. With respect to the compressed size (i.e., 100% - compression efficiency), SyncCoding makes the size on average 12.4%, 20.1%, and 29.9% less than Brotli, LZMA, and Deflate. Fig. 9 (b) shows that SyncCoding achieves nearly the maximum compression efficiency at around $k^* = 24$ number of references, which was our rule of thumb for practical use. Fig. 9 (c) comparing the compressed sizes of 50 randomly chosen documents confirms that SyncCoding gives consistent saving over Brotli, LZMA, and Deflate of about 11.2%, 17.9%, and 29.2%.

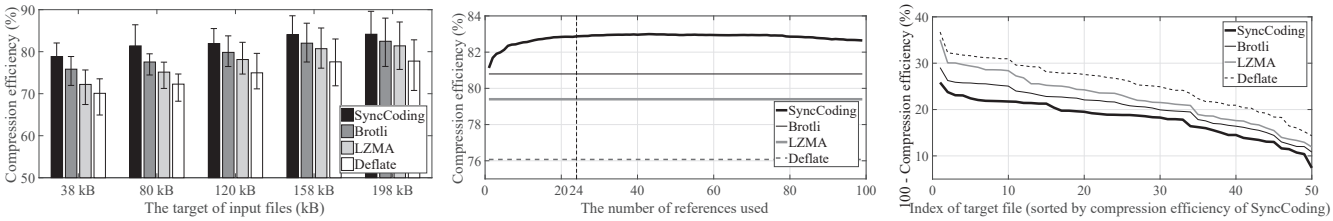
We separately test the performance of Deduplication from a randomly chosen target file with one hundred reference files for various chunk sizes ranging from 4 to 4096 bytes. Fig. 10 shows the compression ratio and efficiency, where the overhead is excluded and included. As Fig. 10 shows, Deduplication achieves its maximum compression efficiency of about 40.59% when the chunk size is 8 bytes. This performance is far lower than 82.65% that from SyncCoding.

We further test the compression efficiencies of SyncCoding and other techniques over the outcomes of Deduplication with one hundred reference files and its best chunk size in Fig. 11. This mimics a mixed Deduplication and compression method proposed in [32]. Our experiment verifies that Deduplication indeed helps other encoding techniques by about 2.17% on average but helps SyncCoding only about 0.22%. This limited improvement in the case of SyncCoding implies that the essence of Deduplication is already embedded well in the redundancy elimination mechanism of SyncCoding.

C. Use Case 2: Web Browsing

To evaluate the efficacy of SyncCoding in web browsing, for a given website, we recorded webpage visit histories of a user and cached all the resources relevant to the webpages (e.g., HTML files, Java scripts, and CSS files) in the visit histories by an off-the-shelf web browser, Google Chrome.

For a given sequence of webpages in a history, we let encoding techniques in comparison compress each webpage when it is invoked. SyncCoding and Deduplication are assumed to utilize all the previous webpages to the newly invoked



(a) Compression efficiency for a target document of various sizes. The error bars indicate 90% confidence intervals.

(b) Compression efficiency for various numbers of references.

(c) Compressed size comparison for 50 target documents sorted by the value of SyncCoding.

Fig. 9. Compression efficiencies of SyncCoding and other techniques (a) for various sizes of the document to encode, (b) for various numbers of references. (c) A comparison of the compressed sizes of 50 target documents when k^* references are used.

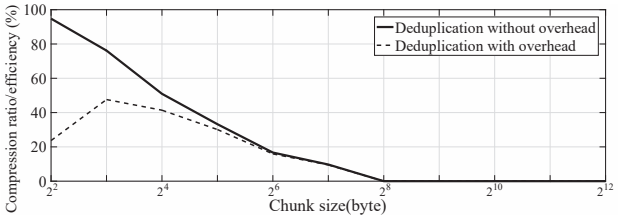


Fig. 10. Compression ratio and efficiency of Deduplication without overhead and with overhead for various chunk sizes in the cloud data sharing scenario with 100 reference files

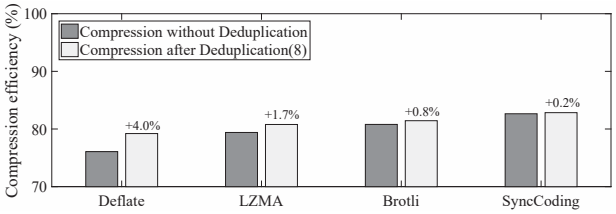


Fig. 11. The performance of compression techniques when combined with Deduplication whose chunk size is 8 bytes.

webpage and Brotli is assumed to exploit its pre-defined static dictionary, that is delivered in advance, between the server and the client. LZMA and Deflate do not use additional resources.

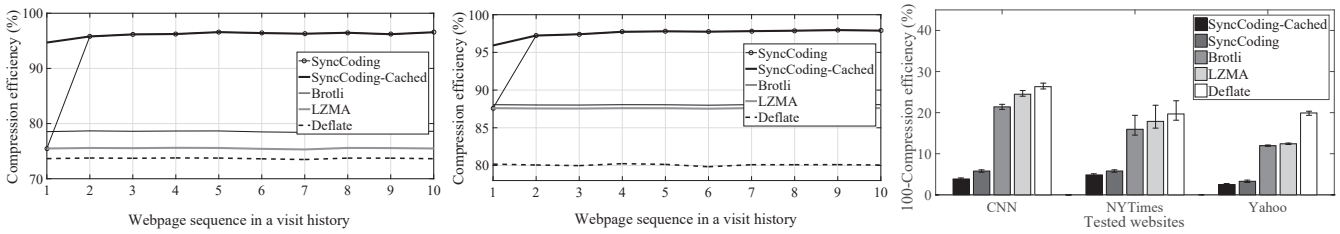
Fig. 12 (a) and (b) show the compression efficiency comparison for a sample visit history recorded inside the politics category of CNN and inside the science category of Yahoo. As expected, Fig. 12 (a) shows that SyncCoding does not show any advantage over LZMA when there is no previous webpage to use, hence for the first webpage. However, from the second webpage, SyncCoding shows significant compression efficiency improvement over LZMA, Brotli, and Deflate. The compression efficiency is nearly maximized after the third webpage and the improvement over Brotli is as much as 20% on average. The same pattern for the compression efficiency is observed for the webpages of Yahoo as shown in Fig. 12 (b). One important thing to note here is that if we allow SyncCoding to cache an old webpage of a website, for instance the main webpage of CNN or Yahoo of yesterday, to our surprise SyncCoding achieves from the first page as good compression efficiency as visiting the second page as shown in Fig. 12 (a) and (b). We denote this technique by *SyncCoding-Cached*. We wondered why this huge gain appears in SyncCoding and found the following reason by an analysis for the contents of the webpages: every webpage in a website authored by a company or a group of programmers show extremely similar programming style (e.g., programming templates), and thus a huge portion of the contents can be referenced from previous webpages in SyncCoding. Note that this gain is fundamentally not achievable when using a static pre-defined dictionary such as in Brotli. To evaluate the performance of SyncCoding for more general web browsing behaviors, we let two test users to freely visit webpages of three websites for an hour, CNN, NYTimes, and Yahoo. Using their visit histories, we perform the same test and depict the average compression efficiencies with 95% confidence intervals in Fig. 12 (c). The figure confirms that in the perspective of the compressed size, the improvement of SyncCoding-Cached over Brotli, LZMA, and Deflate are on average 78.3%, 79.6%, and 86.1% even under such general browsing behaviors. This implies that if a website is prepared to serve its webpages with SyncCoding, it can substantially enhance its user experience.

We again evaluate the performance of Deduplication on the CNN case with ten reference pages for various chunk sizes. Fig. 13 shows the compression ratio and efficiency in which the referencing overhead is excluded and included. Fig. 13 shows that Deduplication achieves its maximum compression efficiency at about 47.60% when the chunk size is 8 bytes. Although this gain is better than the case of cloud data sharing, they are still far below 96.56% which is from SyncCoding.

We also test the compression efficiencies of SyncCoding and other techniques with ten reference pages and its best chunk size in Fig. 14. It shows Deduplication helps other encoding techniques by about 2.53% on average but makes SyncCoding even worse by about 0.4%. This is because SyncCoding loses some chances to match longer subsequences after the Deduplication which twists those subsequences as mixtures of original contents and the addresses to matching chunks.

VII. Concluding Remarks

Given the prevalence of data synchronization services of various forms, we propose a novel data encoding technique called SyncCoding that exploits the database of previously synchronized data in order to achieve higher compression efficiency. Our experiments show that SyncCoding outperforms popular encoding techniques, Brotli, Deflate, and LZMA, both without and with Deduplication in two popular use cases: cloud data sharing and web browsing. We believe that this work establishes a baseline for new encoding techniques



(a) Sample compression efficiencies for the webpages in a visit history of CNN.

(b) Sample compression efficiencies for the webpages in a visit history of Yahoo.

(c) Average compressed sizes of the webpages from CNN, NYTimes, and Yahoo. The error bars indicate 95% confidence intervals.

Fig. 12. Compression efficiencies of SyncCoding, SyncCoding-Cached and three other encoding techniques for the webpages that are sequentially visited by sample visit histories obtained from (a) CNN (Politics section) and (b) Yahoo (Science section). (c) A comparison of the average compressed sizes of webpages from three websites with no section restriction.

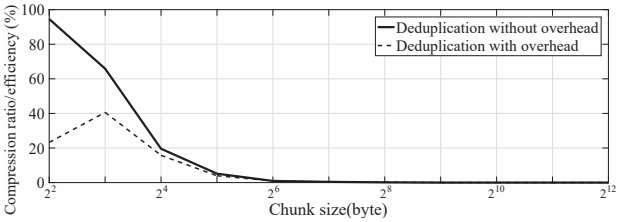


Fig. 13. Compression ratio and efficiency of Deduplication without overhead and with overhead for various chunk sizes on a CNN webpage with 10 reference pages.

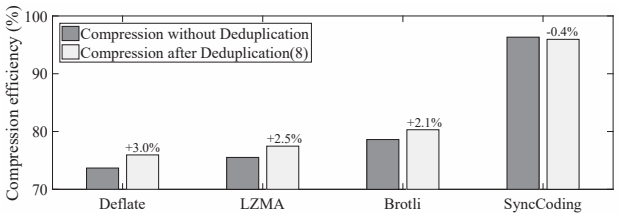


Fig. 14. The performance of compression techniques when combined with Deduplication whose chunk size is 8 bytes.

exploiting inter-file correlations. Our future work includes the extension of SyncCoding allowing its support for multimedia.

VIII. Acknowledgement

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2017-0-00562, UDP-based Ultra Low-Latency Transport Protocol with Mobility Support, No. 2017-0-00692, Transport-aware Streaming Technique Enabling Ultra Low-Latency AR/VR Services), and by Ulsan National Institute of Science and Technology grant (No. 1.170004.01). K. Lee is the corresponding author.

References

[1] Z. Bar-Yossef, Y. Birk, T. Jayram, and T. Kol, "Index coding with side information," *IEEE Transactions on Information Theory*, vol. 57, no. 3, pp. 1479–1494, 2011.
[2] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Transactions on Storage*, vol. 7, no. 4, p. 14, 2012.
[3] N. Ranganathan and S. Henriques, "High-speed VLSI designs for Lempel-Ziv-based data compression," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 40, no. 2, pp. 96–106, 1993.
[4] I. Pavlov. 7z format. http://www.7-zip.org/7z.html.
[5] J. Alakuijala and Z. Szabadka, "Brotli compressed data format," Tech. Rep., 2016.
[6] P. Deutsch, "DEFLATE compressed data format specification version 1.3," Tech. Rep., 1996.
[7] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.
[8] D. A. Huffman *et al.*, "A method for the construction of minimum-redundancy codes," *Proceedings of the I.R.E.*, vol. 40, no. 9, pp. 1098–1101, 1952.
[9] J. Rissanen, "Generalized kraft inequality and arithmetic coding," *IBM Journal of research and development*, vol. 20, no. 3, pp. 198–203, 1976.
[10] M. Ruhl and H. Hartenstein, "Optimal fractal coding is np-hard," in *IEEE Data Compression Conference (DCC)*, 1997.
[11] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.
[12] R. Franceschini, H. Kruse, N. Zhang, R. Iqbal, and A. Mukherjee, "Lossless, reversible transformations that improve text compression ratios," *Technical Report, University of Central Florida*, pp. 1–33, 2000.
[13] F. S. Awan and A. Mukherjee, "LIPT: A lossless text transform to improve compression," in *IEEE International Conference on Information Technology:Coding and Computing*, 2001.
[14] J. Alakuijala and Z. Szabadka. (2014) IETF Brotli compressed data format. https://tools.ietf.org/html/draft-alakuijala-brotli-01.
[15] Understanding Data Deduplication. https://www.druva.com/blog/understanding-data-deduplication/.
[16] Improving the deduplication flow when uploading to Google Drive. https://gsuiteupdates.googleblog.com/2016/09/improving-deduplication-flow-when.html.
[17] J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein, "Delta encoding in HTTP," Tech. Rep., 2001.
[18] File version history–Dropbox. https://www.dropbox.com/help/security/version-history-overview.
[19] Opendedup – opensource dedupe to cloud and local storage. http://opendedup.org/odd/.
[20] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao, "Quicksync: Improving synchronization efficiency for mobile cloud storage services," in *ACM International Conference on Mobile Computing and Networking*, 2015, pp. 592–603.
[21] Z. Tu and S. Zhang, "A novel implementation of jpeg 2000 lossless coding based on lzma," in *IEEE International Conference on Computer and Information Technology*, 2006.
[22] A. D. Wyner and J. Ziv, "The sliding-window Lempel-Ziv algorithm is asymptotically optimal," *Proceedings of the IEEE*, vol. 82, no. 6, pp. 872–877, 1994.
[23] I. Pavlov. Lzma sdk. http://www.7-zip.org/sdk.html.
[24] N. Dehak, R. Dehak, J. Glass, D. Reynolds, and P. Kenny, "Cosine similarity scoring without score normalization techniques," in *Odyssey: The Speaker and Language Recognition Workshop*, 2010.
[25] S. Kullback and R. A. Leibler, "On information and sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.
[26] Google scholar. http://scholar.google.com/.
[27] R. Turek. What YouTube Looks Like In A Day [Infographic]. https://beat.pexe.so/what-youtube-looks-like-in-a-day-infographic-d23f8156e599#.wbu0v1h2z.
[28] J.-l. Gailly and M. Adler. zlib compression library. http://www.zlib.net.
[29] Brotli compression format. https://github.com/google/brotli.
[30] Computerworld. Google to boost compression performance in chrome 49. http://www.computerworld.com/article/3025456/web-browsers/google-to-boost-compression-performance-in-chrome-49.html.
[31] IETF RFC Index. https://www.ietf.org/rfc.html.
[32] C. Constantinescu, J. Glider, and D. Chambliss, "Mixing deduplication and compression on active data sets," in *Data Compression Conference (DCC), 2011*, 2011, pp. 393–402.