An Inter-data Encoding Technique that Exploits Synchronized Data for Network Applications

Wooseung Nam, *Student Member, IEEE,* Joohyun Lee, *Member, IEEE,* Ness B. Shroff, *Fellow, IEEE,* and Kyunghan Lee, *Member, IEEE*

Abstract—In a variety of network applications, there exists significant amount of shared data between two end hosts. Examples include data synchronization services that replicate data from one node to another. Given that shared data may have high correlation with new data to transmit, we question how such shared data can be best utilized to improve the efficiency of data transmission. To answer this, we develop an inter-data encoding technique, SyncCoding, that effectively replaces bit sequences of the data to be transmitted with the pointers to their matching bit sequences in the shared data so called references. By doing so, SyncCoding can reduce data traffic, speed up data transmission, and save energy consumption for transmission. Our evaluations of SyncCoding implemented in Linux show that it outperforms existing popular encoding techniques, Brotli, LZMA, Deflate, and Deduplication. The gains of SyncCoding over those techniques in the perspective of data size after compression in a cloud storage scenario are about 12.5%, 20.8%, 30.1%, and 66.1%, and are about 78.4%, 80.3%, 84.3%, and 94.3% in a web browsing scenario, respectively.

Index Terms—Source coding; Data compression; Encoding; Data synchronization; Shared data; Reference selection

1 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. These new services now become indispensable and occupy a large portion of Internet bandwidth. Given the rise of data synchronization services in which significant amount of shared data exists between servers and clients (i.e., end hosts), we raise the following question: *"how can the previously synchronized data between the end hosts be best exploited for the delivery of new data between them?"*

We find that this question is not only important to synchronization services but also to general network applications including web browsing and data streaming because data transfer between servers and clients essentially lets them have the same synchronized data in the end. Unfortunately, this question has been only partially addressed in the literature and in practical systems.

Index coding [1] first suggested the concept of encoding blocks of data to be broadcasted most efficiently to a group of receivers holding different sets of blocks. The problem setting of Index coding is related to ours, but it focuses on mixing blocks for optimal broadcasting by mostly using XOR operations and does not pay attention to exploiting the similarity among the blocks. *Deduplication (dedup)* [2] and *RE (redundancy elimination)* [3], which have been studied and developed intensively for storage and network systems, are capable of exploiting previously stored or delivered data for storing or transmitting new data. However, they mostly work at the level of files or chunks of a fixed size (e.g., 4MB in Dropbox, 8kB in Neptune [4]), which significantly limit the potential of synchronized data. Even with state-of-the-art deduplication techniques that can find chunk boundaries in a flexible manner from using CDC (contents-defined chunking) techniques [4], [5], [6], [7], the synchronized data is not fully exploited due to their chunk-to-chunk operations. There exist many computation acceleration techniques for deduplication and RE such as bloom filter [8], stream-informed locality [9], rolling hash [10], and hardware accelerators [11], but these do not improve the efficiency of encoding (i.e., the size of encoded data).

In this paper, we try to answer the question by proposing an inter-data encoding technique called *SyncCoding* that is a new framework of exploiting shared data for encoding in two steps: 1) given data to encode, selecting *references* for encoding which hold high similarity with the data to encode from the pool of previously synchronized data, 2) encoding the data with the chosen references, which allows bit sequences of flexible lengths in the data to encode to be referenced efficiently from multiple references. This framework enables a long matching bit sequence much larger than the size of a chunk in a reference to be referred by a single pointer (i.e., the position in the reference and the matching length) and enables a group of short matching bit sequences toward multiple references to be referred from multiple references instead of them being simply delta-coded over a certain chunk.

While deduplication and RE for chunk-level redundancy elimination are known to be effective in leveraging the series of files originated from a single source file, which are mostly the same and only partially different, SyncCoding can potentially benefit from more diverse files (e.g., including data that are created on similar topics, by similar authoring

- *Corresponding authors: K. Lee and J. Lee.*
- *W. Nam and K. Lee are with the School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology, Ulsan 44919, South Korea (e-mail: {wsnam, khlee}@unist.ac.kr).*
- *J. Lee is with the Division of Electrical Engineering, Hanyang University (e-mail: joohyunlee@hanyang.ac.kr).*
- *N. B. Shroff holds a joint appointment in both the Department of ECE and the Department of CSE at The Ohio State University (e-mail: shroff.11@osu.edu).*

styles, or in similar formats), irrespective of whether they are originated from a file or not.

How to most efficiently realize these two steps in SyncCoding is not straightforward to answer and is an open problem. In this paper, we start tackling the problem by providing an initial implementation of SyncCoding utilizing 1) modified cosine similarity for selecting references and 2) modified LZMA (Lempel-Ziv-Markov chain algorithm) [12] for efficient bit-sequence referencing from multiple references. We find it especially interesting that a non-trivial portion of data in network applications such as documents or program codes stored in cloud storages or web servers fall into the category where the current implementation of SyncCoding is highly effective. Nonetheless, we note that the framework of SyncCoding is not limited to a specific reference selection or a compression algorithm. This framework can be easily extended to use other similarity measures for ensuring more efficient applicability to general files (e.g., images, videos) and to use more advanced compression algorithms such as PAQ [13] and AV1 [14] for improving the efficiency.

In order to design, validate, and evaluate the initial implementation of SyncCoding, we take the following steps.

1) We revisit the algorithm of LZMA, the core of *7-zip* compression format [15] that is known as one of the most popular data encoding techniques and reveal how it works in detail.
2) We design the framework of SyncCoding with LZMA and provide a way for LZMA to encode data using references.
3) We analyze the conditions under which SyncCoding outperforms the original LZMA with no reference in the size of compressed data and suggest practical heuristic algorithms to select references from the pool of synchronized data (i.e., reference candidates) in order to meet the conditions.
4) We implement SyncCoding in a Linux system and evaluate its compression characteristics. We also implement it in an Android system and study its energy consumption characteristics. We further demonstrate the benefits of using SyncCoding in realistic use cases of cloud data sharing and web browsing.
5) We study the performance of SyncCoding for encrypted data and discuss an implementation guideline for SyncCoding.

Our evaluation of SyncCoding in the cloud data sharing scenario with a dataset of RFC (Request For Comments) technical documents reveals that on average SyncCoding compresses documents about 10.8%, 20.8%, and 66.1% more compared to LZMA after Deduplication, LZMA without Deduplication, and Deduplication only[1], respectively. Another evaluation in the same scenario with a dataset of PDF and image files shows that SyncCoding compresses 18.5% and 40.0% more for PDF files and 1.2% and 3.0% more for image files compared to LZMA and Neptune. It confirms that SyncCoding is still beneficial even for PDF and image files, but at the same time it reveals the need for a future work that can work more efficiently especially for image files, for instance, by allowing partially matching bit sequences to be referenced with delta-encoded bits or by allowing arbitrary two-dimensional blocks (i.e., sets of non-continuous bits in the original bit stream) of bits to be referenced.

Further evaluation of SyncCoding in the web browsing scenario shows that SyncCoding outperforms commercial web speed-up algorithms, Brotli [16] from Google with and without deduplication by 81.3% and 84.3%, and Deflate [17] with and without deduplication, by 85.6% and 87.2%, respectively, in the size of compressed webpages of CNN. We find that this substantial gain observed for SyncCoding comes from the similar programming style maintained over the webpages in the same website and confirm that the gain is persistent over various websites such as NY Times and Yahoo.

1. The chunk size for deduplication here is chosen as a small value, 8 bytes to demonstrate the maximum potential of deduplication

2 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 original bit sequence can be perfectly recovered from the encoded bit sequence, it is called *lossless* compression which is the focus of this work. 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 [18] tells us that a symbol-by-symbol encoding becomes optimal when symbol i that appears with probability p_i in the symbol sequence is encoded by $-\log_2 p_i$ bits. It is well known that Huffman coding [19] is an optimal encoding for each symbol but is not for a symbol sequence. Arithmetic coding [20] 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 unclear how to find the optimal block sizes that give the smallest encoded bits. Therefore, finding the real optimal encoding for an arbitrary bit sequence becomes NP-hard [21] due to the exponential complexity involved in testing the combinations of the block sizes.

LZ77 [22], 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.

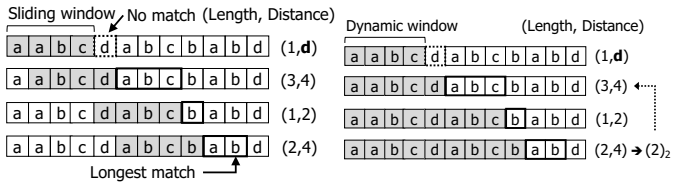
Deflate [17] combines LZ77 and Huffman coding. It replaces matching blocks of symbols with length-distance pairs similarly to LZ77 and then further compresses those pairs using Huffman coding. LZ78 and LZMA are variants of LZ77, of which their encoding methods for length-distance pairs are improved. 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 3.

Unlike the aforementioned compression algorithms, there exist several techniques that include external information in addition to the source data for encoding. There are simpler ways of exploiting external information such as Star encoding (*-encoding) [23] that uses an external static dictionary shared between a server and its client. A similar yet more efficient approach has been made using Length Index Preserving Transform (LIPT) [24] with an English dictionary having about 60,000 words. *Brotli* [16], one of the latest encoding techniques, 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% compression gain over Deflate in the encoding of webpages in a web browser [25]. Exploiting a static shared dictionary is useful in general, but its efficacy is limited as each replacement is bounded by the length of words.

Deduplication [2] is a practical repetition elimination technique for duplicate data, which is widely studied and developed for storage systems. It essentially replaces repeated data chunks of a file with the matching chunks of other files in the storage by which it enables the concept of SIS (single instance storage). A similar idea called *RE* (redundancy elimination) eliminates duplicate packets in network traffic originally at the network switches [3] and later in the end hosts as in EndRE [26] to avoid its impact being reduced by encrypted packets with TLS (transport layer security). Deduplication is especially effective for secondary storage systems in which periodic system back-ups that are highly redundant to each other occupy a large portion. It is also effective in cloud storage services such as Google Drive [27] and Dropbox [28] because there exist many subscribers who store popular files such as music, image, video, and PDF files in their storage spaces. As long as these popular files are unmodified, the files can be easily deduplicated in the cloud storage system. However, when there exist slight modifications, the original deduplication with FSC (fixed-size chunking) fails to work due to so called *boundary shift* problem.

To tackle this problem, CDC (contents-defined chunking) [10] is proposed with byte-level fingerprints (i.e., rolling hash values such as Rabin fingerprints [10] and Gear hashes [29]) in which chunk boundaries are not deterministically defined by the size but defined adaptively by a pre-defined hash pattern. CDC makes chunk sizes variable and requires much more computation than FSC, but it effectively identifies modified chunks that are subject to delta encoding [30] over certain chunks and also detects right boundaries to extract unmodified chunks. For efficient delta encoding for an unmatched chunk, Neptune [4] and a WAN optimization technique [31] leverage a sketch of that chunk, which is nothing but some characteristic values obtained from fingerprints, to find a similar chunk as a basis for delta encoding. QuickSync [32] utilizes the idea of Neptune for mobile devices and optimizes it for energy saving by adapting the average chunk size of CDC to the network bandwidth and by bundling packet transmissions for delta-encoded chunks. In another line of research, a number of acceleration techniques for deduplication in practical systems are proposed such as bloom filter [8], stream-informed locality [9], and hardware accelerators using GPU [11], but they do not fundamentally improve the compression efficiency of deduplication.



(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.

3 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 static window size in LZ77 may cause inefficiencies. For example, when the window size is small, the number 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 is not suffering 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 [33].

The optimality of LZ77 was proved earlier by Ziv and Lempel [34] 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 [34].

Our interest lies whether SyncCoding uses less or more bits than LZMA. 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 the 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 [34] 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 [34], 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 above equations, 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 $T_{\text{LZMA}}(\{S\}_1^N)$.

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., $l_i = 1$).

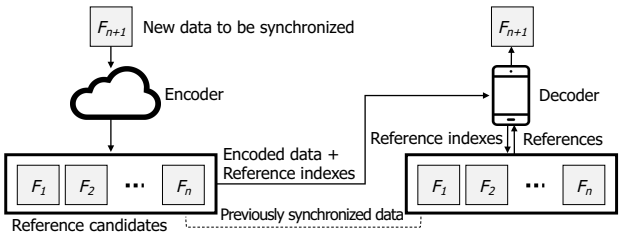


Fig. 2. The concept and basic operations of SyncCoding.

4 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.

4.1 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 the $(n+1)$-st file F_{n+1}, from the server to the client, our problem is to answer how should F_{n+1} 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 decoding 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_{n+1}^k 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_{n+1}^k. 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 [35].

Algorithm 1 Encoding/Decoding Procedures of SyncCoding

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

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

4.2 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 $p_{\text{SC}}^1 \le p_{\text{LZMA}}^1$, where p_{SC}^1 denotes the number of phrases of length one in SyncCoding.

We now find the condition that guarantees better compression for SyncCoding over LZMA, so that $T_{\text{SC}}(\{S\}_1^N) < T_{\text{LZMA}}(\{S\}_1^N)$ 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 $0 < \gamma \le 1$. 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, which is to increase 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 $\bar{l}_{\text{SC}}$ 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 $\bar{l}_{\text{SC}} = \bar{l}_{\text{LZMA}}/\gamma$, where $\bar{l}_{\text{LZMA}}$ 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 removed by using the references, say $p_{\text{SC}}^1 = 0$. 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[2] let us conclude that $T_{\text{SC}}(\{S\}_1^N)$ 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, $T_{\text{SC}}(\{S\}_1^N) < T_{\text{LZMA}}(\{S\}_1^N)$, gives the following theorem.

Theorem 1 *If $h(\gamma) > 0$ is satisfied for the following definition of $h(\gamma)$, the total number of bits of SyncCoding is less than that of LZMA, i.e., $T_{SC}(\{S\}_1^N) < T_{LZMA}(\{S\}_1^N)$.*

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

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

4.3 Questions on SyncCoding

As revealed by the analysis, the efficacy of SyncCoding over LZMA depends highly on how much SyncCoding can reduce the number of length-distance pairs to encode. The ratio of reduction, γ, is the outcome of the reference

2. 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.

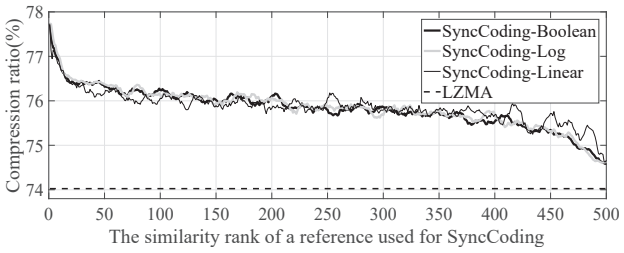


Fig. 3. The compression ratios of LZMA and SyncCoding with one reference whose modified cosine similarity is ranked by either of Boolean, Log, and Linear. Overall, SyncCoding with a single reference shows higher compression ratios than LZMA and a reference of a higher rank achieves a better compression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

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?*, 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)?*, and 4) *How much energy does SyncCoding consume in downloading and decoding a file in mobile devices?*

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

5 Characterization of SyncCoding

5.1 Reference Selection

We first tackle the question on 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 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 given 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 as much time 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 ensure 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 [36] and Kullback–Leibler divergence [37]. Based on such similarity measures, we propose a modified cosine similarity measure. Our modified cosine similarity denoted by $\text{sim}(A, T)$ between two files, a reference candidate A and the target file to encode T, is formally defined as follows:

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

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 build a dataset by randomly downloading 8,000 RFC documents from the IETF database [38]. We use one hundred documents as new data for synchronization and use the others to imitate the database of previously synchronized data (i.e., reference candidates). With this sample database, we rank the reference candidates 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 reference candidate 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 77.7% compression ratio meaning that the compressed size is only 22.3% of the original size. Comparing this result with that of LZMA which achieves the compression ratio of 74.0% and results in the compressed file whose size is 26.0% of the original, SyncCoding reduces the size of the compressed file by about 14.3% 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

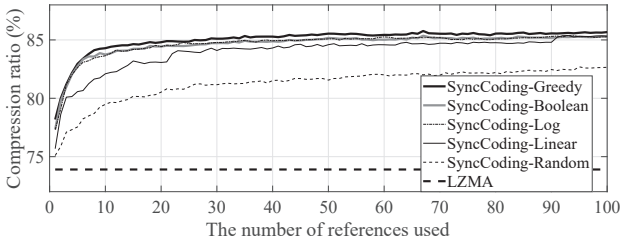


Fig. 4. The compression ratios of SyncCoding with increasing number of references that are selected randomly, by a greedy search, or by the modified cosine similarity rank with either of Boolean, Log, or Linear. In this figure, the overhead for reference indexing is not considered to focus on understanding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

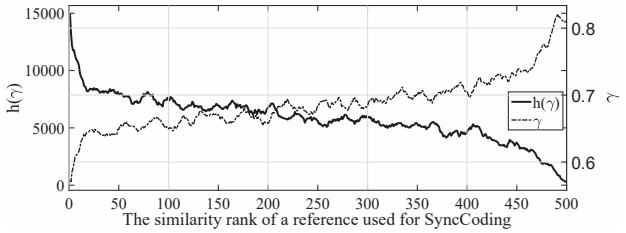


Fig. 5. The $h(\gamma)$ and γ of SyncCoding over LZMA with one reference whose modified cosine similarity is ranked by the Boolean transformation function.

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[3], it is reasonable to conclude that SyncCoding-Boolean is a viable solution to the reference selection problem.

We lastly check whether modified cosine similarity with Boolean transformation function can be used as a substitute for $h(\gamma)$ of Theorem 1 or not. We explain in Section 4 that quantifying $h(\gamma)$ is difficult because computing $h(\gamma)$ requires the actual number of length-distance pairs in a target file and the references used. This means that obtaining the value of $h(\gamma)$ is as difficult as compressing the target file by SyncCoding using the chosen references. Therefore, having a good proxy of $h(\gamma)$ is important. Fig. 5 shows that our similarity measure has an increasing tendency with $h(\gamma)$ and a decreasing tendency with γ over reference candidates, confirming that our measure can predict the rank of $h(\gamma)$ for reference candidates. Throughout this paper, we use SyncCoding-Boolean as our default SyncCoding implementation.

5.2 Maximum Compression Efficiency

We now tackle the second question on the maximum compression advantage of SyncCoding over LZMA. It is of particular interest in 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

3. 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 $O(N!)$.

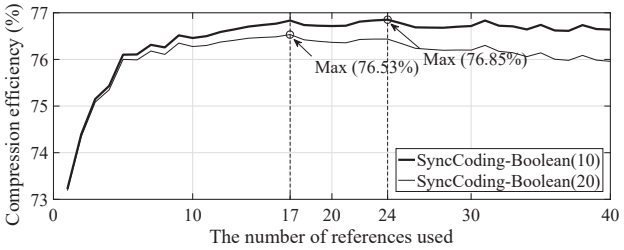


Fig. 6. The compression efficiency of SyncCoding for an increasing number of references. The per-reference overhead is chosen as either of 10 or 20 bytes.

networks with high link cost such as satellite communication, while being at an ocean cruise or at an airplane. From 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 [39]).

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 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^{24} 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. 6. 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, assuming that no additional encoding is applied for the indexes. We discuss about the overhead optimization in the next subsection.

As shown in Fig. 6, with 10 and 20 bytes overhead per reference, SyncCoding achieves about 76.85% and 76.53% as its maximum compression efficiency for the chosen document, respectively. The number of references that achieve the maximum compression efficiency in this test are 24 and 17, confirming the intuition that a larger per-reference overhead makes the compression efficiency saturated earlier with respect to the number of references used. However, even with a larger per-reference 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.

We now provide a practical method of choosing the number of references that approximately obtains the maximum compression efficiency for a given file to encode. Note that finding the real optimal number of references to use,

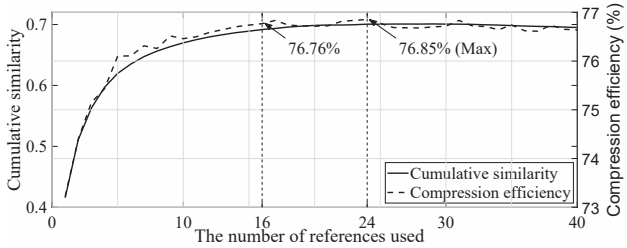


Fig. 7. The cumulative similarity and compression efficiency of SyncCoding for an increasing number of references. The per-reference overhead is chosen as 10 bytes.

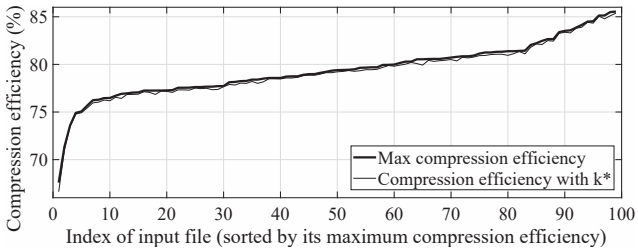


Fig. 8. Maximum compression efficiencies of SyncCoding obtained from 100 randomly chosen documents are compared with the compression efficiencies of SyncCoding that use only k^* references. The difference between them is marginal.

say k^*, is impractical as it requires to find the maximum from ${}_nP_k$ ($= \frac{n!}{(n-k)!k!}$) permutations given k ranging from 1 to n, where k and n are the number of references to use and the number of reference candidates. To avoid such high complexity, we again use our similarity measure to approximate the compression result as follows: 1) We first sort each reference candidate $A \in R$ for the file to encode T by its similarity, $\text{sim}(A, T)$, where R is the set of reference candidates (i.e., $|R| = n$). 2) We then add them one by one to the set of references while measuring the cumulative similarity at each addition as $\text{sim}(A_k, T)$, where A_k denotes the set of k most similar reference candidates from R. 3) We finally find the saturation point of the cumulative similarity by testing if the moving average of $\Delta_k = \text{sim}(A_k, T) - \text{sim}(A_{k-1}, T)$ is below a certain threshold Δ_{th} (e.g., Δ_{th} = 0.01). We use k at this saturation point as k^*.

Fig. 7 shows the cumulative similarity with 10 bytes per-reference overhead and k^* obtained from the aforementioned method for the same data and setting as in Fig. 6. As shown in Fig. 7, our method finds $k^* = 16$ and gives the compression efficiency of 76.76% which is not much different from the actual maximum 76.85% with $k = 24$ as in Fig. 6. We finally test 100 randomly chosen documents with our method for choosing k^* and compare their compression efficiency with actual maximum compression efficiency obtained from huge computation cost under 10 bytes per-reference overhead in Fig. 8. Fig. 8 shows that the performance gap is within 0.96%, confirming that our method of approximately finding k^* works effectively.

5.3 Referencing Overhead Optimization

The overhead of referencing files by fixed-length indexes can be further optimized by a variable length coding such as Huffman coding [19]. Especially when there are multiple

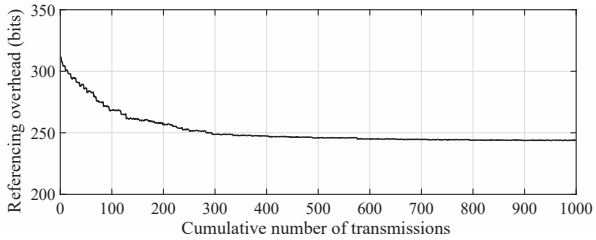


Fig. 9. The referencing overhead by indexing references with Huffman coding when referencing frequencies of reference candidates are updated over transmissions.

files to exchange between end hosts which already have many synchronized files, for instance file exchange between data centers, indexing with a variable length coding can help. In such a case, instead of indexing N files equally assigned with $\log_2 N$ bits, assigning less bits for more frequently referenced files is possible. Huffman coding in principle assigns $-\log_2 w_i$ bits to index file i that is referenced by f_i times thus having its relative weight $w_i = f_i / \sum_i f_i$. Similarly, at every file transfer between end hosts, w_i can be updated for all synchronized files and the referencing indexes can be reassigned accordingly. The more the files are referenced, the less the bits are reassigned. Hence, referencing overhead reduces as file transfers continue.

In order to demonstrate this idea, we reuse our 8,000 RFC documents dataset in Section 5.1 and build a synchronization database between end hosts, in which indexing a reference needs 13 bits for equal bit assignment. We test the total referencing overhead for each file transfer where a thousand randomly chosen files from the database are transferred sequentially each with 24 references, and the indexes are updated as aforementioned. In a test, all other unchosen files are considered previously synchronized and are used as reference candidates. The results in Fig. 9 are averages from 200 repetitions of this test. As shown in the figure, the overhead starts from 312 bits (13×24) and reduces gradually over transmissions to about 245 bits. It is hard to know what the actual reduction in overhead will be because it depends on how frequencies of chosen references change over transmissions, but it is always possible to optimize overhead in this way. In particular, data centers with a huge number of synchronized files can benefit from this more. However, in the remaining sections, we use fixed-length indexes to characterize the performance of SyncCoding the most conservatively.

5.4 Encoding Time and Decoding Time of SyncCoding

We tackle the third question on the encoding and the decoding time of SyncCoding by performing experiments. We let $T_E(x, k)$ and $T_D(x, k)$ denote the time duration 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. 10, a measurement on Linux (Kernel 2.6.18-238.el5) over Intel i7-3770 CPU (3.40 GHz) for three kinds of documents of on average about 50, 100, and 200kB, a hundred for each kind, randomly chosen from the aforementioned dataset in Section 5.1, confirms that the average encoding

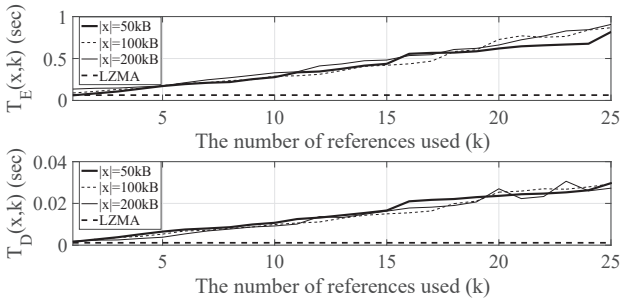


Fig. 10. Experimental evaluation of $T_E(x,k)$ (top) and $T_D(x,k)$ (bottom), the time duration to encode and to decode a file x with k references under Intel i7-3770 CPU (3.40 GHz). For comparison, the line for LZMA denotes the time consumed from LZMA for $|x| = 200$kB.

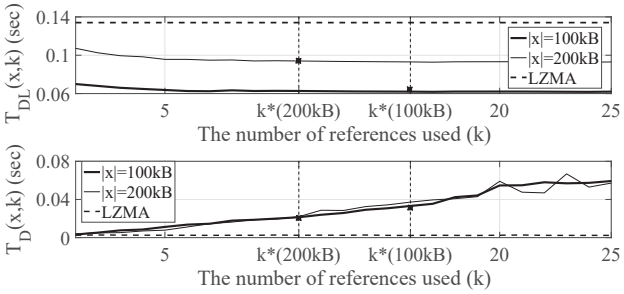


Fig. 11. Experimental evaluation of $T_{DL}(x,k)$ (top) and $T_D(x,k)$ (bottom), the time duration to download and to decode a file x with k references under Galaxy Note 5 (Exynos 7 Octa 7420) connected to an LTE network with the downlink speed of about 30 Mbps. For comparison, the line for LZMA denotes the time consumed for $|x| = 200$kB.

time as well as the average decoding time from one hundred trials increases almost linearly to k. Fig. 10 also confirms that the size of x has little impact on the times because the size of the data to encode is smaller than the total size of the references. The decoding time is on the scale of milliseconds and is relatively negligible compared to the encoding time which is on the scale of a second.

We further evaluate end-to-end time performance in the perspective of a mobile device, which is from downloading to decoding. We let $T_{DL}(x,k)$ denote the time duration taken for downloading a document x encoded with k references in the aforementioned Linux platform. The average downlink speed of the LTE network connected on a mobile device, Galaxy Note 5, we use during the experiment is about 30 Mbps. We repeat the experiments 100 times for two different average sizes of the tested documents, 100kB and 200kB. As shown in Fig. 11, the downloading time of SyncCoding decreases and the decoding time increases as the number of references increases. As a result, the end-to-end time in the mobile device is minimized when using k^*= 16 for 100kB and $k^* = 11$ for 200kB, and the reduction in end-to-end time is 2.3% for 100kB and 15.1% for 200kB compared to only LZMA, respectively. One important thing to note here is that the encoding time can often be hidden to users due to 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 6.

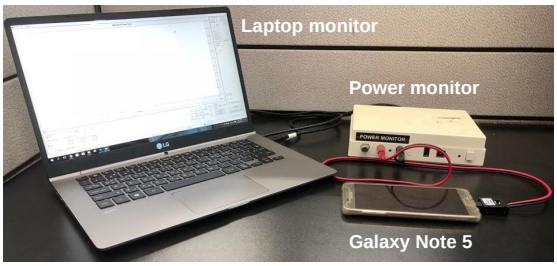
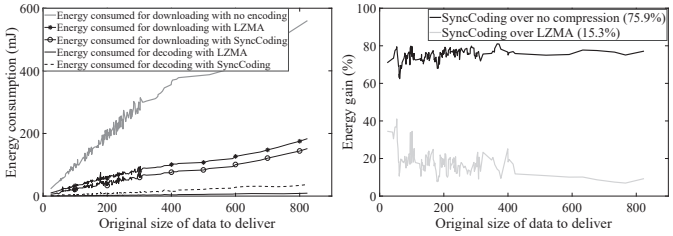


Fig. 12. Our test environment for measuring energy consumption of receiving data with or without SyncCoding in an Android device. Measurements are conducted by a digital power monitor from Monsoon [40].



(a) The energy consumption for downloading and decoding data of variable sizes with SyncCoding and LZMA.

(b) The energy gain of SyncCoding over no compression and LZMA. Average gains are presented in the bracket.

Fig. 13. The energy consumption measurement results on Galaxy Note 5 smartphone for downloading and decoding data of variable sizes with SyncCoding and LZMA.

5.5 Mobile Energy Consumption of SyncCoding

We answer the last question on the mobile energy consumption of SyncCoding in this subsection. Since mobile devices are more sensitive to high energy consumption than power-corded desktops or data centers, it is important to know how much energy that SyncCoding consumes for receiving data in a mobile device. We define the energy consumption with a compression algorithm for receiving data to be the total energy consumption from the start of downloading data to the end of decoding data. When no compression is applied, no energy is consumed for decoding. We experiment the energy consumption from SyncCoding in comparison with LZMA and no compression in two perspectives: 1) energy saving by downloading compressed data of smaller sizes and 2) extra energy spending for decoding. For this experiment, we reuse the RFC documents of the aforementioned dataset in Section 5.1, whose average file size is about 250kB. We randomly choose one input file (i.e., the target paper to compress) from this dataset and use k^* reference files chosen out of all other papers. The experiment is carried out on an Android device, Galaxy Note 5 connected to an LTE network. The average downlink speed of the LTE network we use during the experiment is about 30 Mbps. Our setup for the energy measurement is depicted in Fig. 12.

Fig. 13 (a) shows the mobile energy consumption with SyncCoding, LZMA, and with no compression for receiving data of variable sizes. When receiving data with a

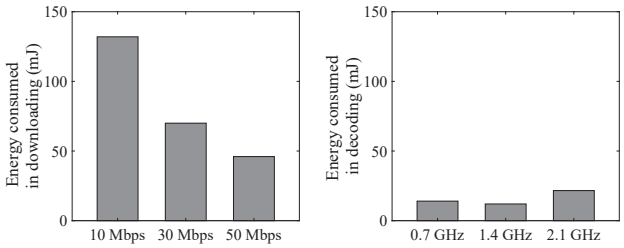


Fig. 14. The energy consumption of SyncCoding for downloading with various downlink speeds which are 10, 30, and 50 Mbps (left) and for decoding with different computational efficiencies which are 0.7, 1.4, and 2.1 GHz (right) on a Galaxy Note 5 smartphone.

compression method, the downloading size is reduced, but the additional decoding process is needed. SyncCoding on average saves 26.7% and 69.2% of energy over LZMA and no compression for the downloading part but overspends 69.4% of energy over LZMA for the decoding part. The energy consumption for downloading as well as decoding increases nearly linearly to the input file sizes. This is reasonable because the compressed data size is highly correlated with the input file size and the decoding process that reads through the compressed data to progressively recover the original bit sequences from the references is also highly correlated with the compressed data size. Since the amount of saved energy in downloading is greater than the energy overspent for decoding, SyncCoding can overall outperform no compression as well as LZMA in terms of total mobile energy consumption. Fig. 13 (b) summarizes the energy gain of SyncCoding from receiving data over LZMA and no compression, which are shown to be on average 15.3% and 75.9%, respectively[4].

Fig. 14 further shows the impact of various network bandwidths and CPU clock frequencies (i.e., energy efficiencies of computation) to mobile energy consumption when using one hundred documents of on average about 200kB encoded with k^* references. For the evaluation, we controlled the bandwidth of our server from which our mobile device downloads data through LTE networks and adjusted the CPU clock of our mobile device using DVFS (dynamic voltage and frequency scaling) API of Android platform. As shown in Fig. 14, network bandwidth gives nearly linear impact to mobile energy consumption while CPU clock gives rather flat impact due to computation time increase for clock reduction. However, note that these results can be more fickle when using different LTE chipsets and CPU models with different architectures. Therefore, one should consider these conditions when applying SyncCoding for mobile energy saving.

We lastly evaluate the total energy consumption of SyncCoding for the different number of references, k, used in Fig. 15 to find the optimal k that minimizes the energy consumption. As shown in Fig. 15, for smaller k, the energy consumption for decoding is less, while the energy consumption for downloading is more. It is vice-versa for larger k. We find that the optimal k^* in terms of mobile energy

4. The results fluctuate for small original file sizes because of the randomness in experiments (e.g., difference in contents for encoding, LTE channel variation, computational load variation from system services), but it becomes more stable as the original size increases.

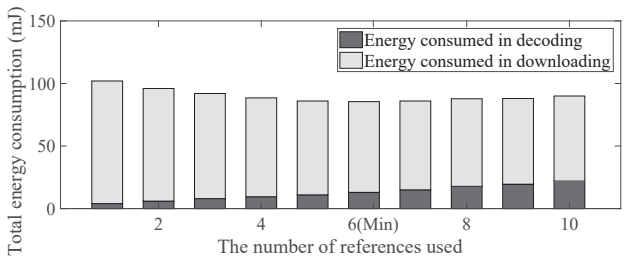


Fig. 15. The total energy consumption of SyncCoding for the different number of references used, k, which consists of the energy consumed in downloading and decoding.

consumption in this experiment is 6, which is different from the optimal k^* of 9 that obtains the shortest mobile end-to-end time. We leave the mobile energy modeling for downloading and decoding for different k as future work.

5.6 Energy Consumption of SyncCoding in Extreme Situations

As a special case of mobile applications of SyncCoding, we further characterize SyncCoding in extreme situations such as deep sea communications and planetary communications in which devices are often resource-constrained and their communication bandwidth are mostly extremely limited. In such an environment, the benefit of applying SyncCoding for data exchange is directly related to the trade-off between the energy saving from compressing the size of data to transmit or receive and the energy expenditure from encoding or decoding of SyncCoding. We here focus on the encoding and transmission case for simplicity. We find that data communications in such extreme situations often demand very high power consumption especially in the transmission part (e.g., tens of watts in deep see acoustic data transmission [41], tens of watts in the outer planetary data transmission of Voyager 1 and 2 [42], and tens of watts in the Iridium satellite data transmission [43]), but they only achieve very low data rates such as tens of kilo bps. In such a case, reducing the size of data to transmit, thus reducing the transmission time can save a substantial amount of energy. Therefore, although SyncCoding takes extra energy for data encoding, there exist possibilities for energy saving.

In order to experimentally characterize such behaviors, we emulate the energy consumption of SyncCoding in extreme situation by lowering the CPU clock frequency and bandwidth. For one hundred documents to transmit whose average size before encoding is about 200 kB [5], we measured average time to encode in SyncCoding and LZMA under 700, 1400, and 2100 MHz CPU frequency under i7-3770 CPU by controlling the clock and voltage with CPUfreq module [44], one of DVFS APIs on Linux platform. The resulting encoding times are 2.24, 1.07, and 0.71 seconds for SyncCoding and 0.45, 0.22, and 0.14 seconds for LZMA. We also measured the transmission time between two Linux machines by limiting the network bandwidth using netem [45] as 10, 50, and 100 kbps. The results are 25.55, 4.99, and 2.40 seconds for SyncCoding and 43.83, 8.38, and 4.32 seconds for LZMA. Overall, SyncCoding spends

5. Each document is encoded in SyncCoding with k^* references chosen for the document.

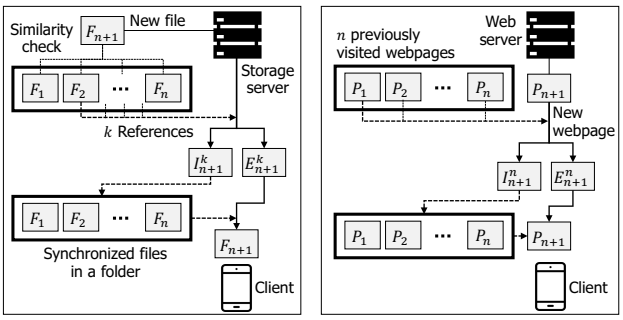


Fig. 16. Overview of the evaluation scenarios: 1) cloud data sharing (left) and 2) web browsing (right).

more time in encoding and spends less time in transmission. Converting these results to energy consumption for the case of 10 kbps and 700 MHz with typical 10 watts power consumption in the transmitters for extreme situations and 11.4 watts power consumption measured by PowerTOP Linux tool, we get 41.7% energy saving (i.e., 182.8 J) in transmission and 79.9% energy overspending (i.e., 20.4 J) in encoding from SyncCoding over LZMA. In total, SyncCoding gives 36.6% energy saving (i.e., 162.4 J) in this test scenario. We note that this result can vary depending on the types of communication chipsets, processor models with different architectures, the size of data to transmit, and link speeds of the networks. Therefore, in order to apply SyncCoding to extreme situations, it is necessary to study these conditions judiciously.

6 EVALUATION

We evaluate the efficacy of SyncCoding in two real scenarios: 1) cloud data sharing and 2) web browsing. The scenario we consider for 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 files relevant to the new file. The use case we consider for web browsing is to browse webpages of a website at a user device given that the webpages 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. 16.

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 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 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.

6.1 Settings

SyncCoding: We use SyncCoding-Boolean with k^* references chosen by the proposed algorithm in Section 5.2 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 [35] with the parameters from the maximum compression option.
Deflate: For the evaluation of Deflate, we use [46], 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 [47], which is embedded in Google Chrome web browser [48]. We also use its maximum compression option.
Deduplication: For the evaluation of Deduplication, we modify OpenDedup [49] so as to investigate its ideal Deduplication performance for documents. We reduce the lower bound of the chunk size (i.e., 1kB in OpenDedup [49]) to be arbitrarily small.

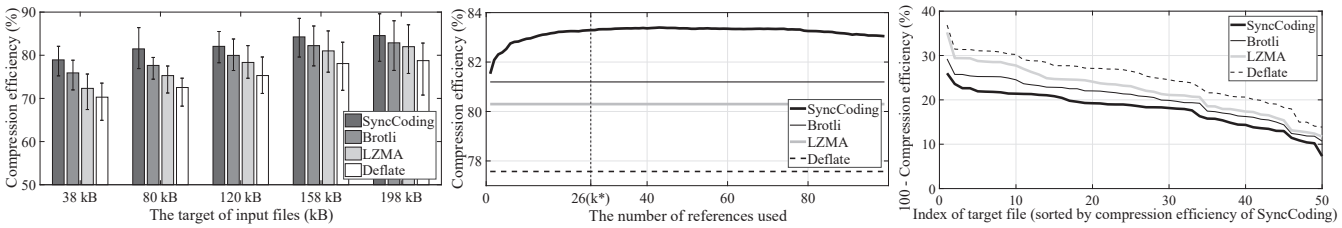
6.2 Use Case 1: Cloud Data Sharing

We emulate a folder of a cloud storage (e.g., Dropbox) by creating a folder with files of similar attributes. In this folder, we put three types of data: 1) 8,000 RFC documents as in Section 5.1, 2) 300 image files collected from Google image for a few search keywords (e.g., Eiffel tower), 3) 300 PDF files of presentation materials collected from Google for a few research topics (e.g., wireless networking).

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. For the RFC documents, 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. 17 summarizes the results of these tests. Fig. 17 (a) shows the average compression efficiency 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 12.5%, 20.8%, and 30.1% less than Brotli, LZMA, and Deflate. Fig. 17 (b) shows that SyncCoding achieves nearly the maximum compression efficiency at around the chosen number of references, $k^* = 26$ [6]. Fig. 17 (c) comparing the compressed sizes of 50 randomly chosen documents confirms that SyncCoding gives consistent saving over Brotli, LZMA, and Deflate of about 11.5%, 18.3%, and 29.5%.

We separately test the performance of Deduplication from a randomly chosen target file with one hundred reference files for various chunk sizes from 4 to 4096 bytes.

6. SyncCoding inarguably requires more computation. We note that Brotli and Deflate show 14% and 98% faster encoding and 72% and 67% faster decoding compared to SyncCoding in this experiment.



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

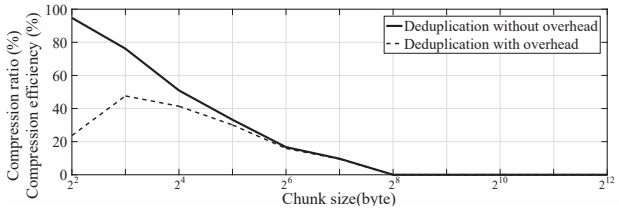


Fig. 18. Compression ratio and compression efficiency of Deduplication without overhead and with overhead for various chunk sizes in the cloud data sharing scenario with 100 reference files.

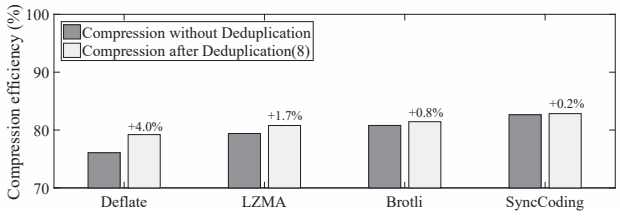
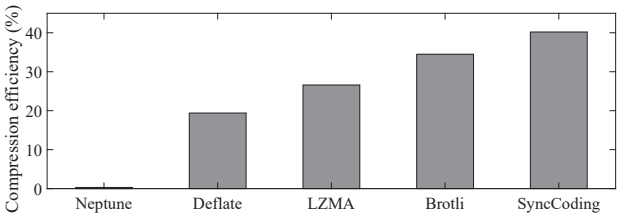


Fig. 19. The performance of compression techniques when combined with Deduplication whose chunk size is 8 bytes.

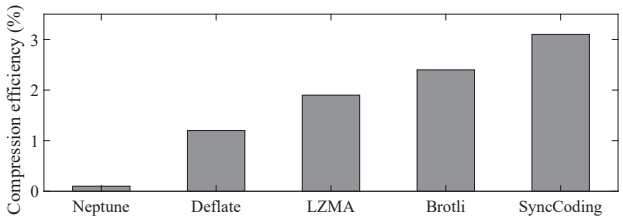
Fig. 18 shows the compression ratio and efficiency with and without overhead. As Fig. 18 shows, Deduplication achieves its maximum of about 47.60% when the chunk size is 8 bytes, but this is far lower than 82.26% from SyncCoding.

We further test the compression efficiency of SyncCoding and other techniques over the outcomes of Deduplication with one hundred reference files and its best chunk size in Fig. 19. This mimics a mixed Deduplication and compression method proposed in [50]. Our experiment verifies that Deduplication indeed helps other encoding techniques by about 2.17% on average but helps SyncCoding by only about 0.19%. This limited improvement over SyncCoding implies that SyncCoding already eliminates most redundancy that Deduplication targets to eliminate.

We lastly test the impact of SyncCoding over JPEG image files and PDF files, which are known to occupy top two portions, 19.6% and 14.5%, in cloud storage services [51]. As these files are relatively larger than RFC documents (on average 400kB for JPEG images and 330kB for PDF files), we compare SyncCoding with one of the state-of-the-art Deduplication techniques, Neptune [4] that exploits CDC with the average chunk size of 8kB in conjunction with delta encoding for unmatched chunks. We note that in order to find similar files for PDF and image files, we apply the same modified cosine similarity as in Section 5.1 with the definition of a word being simply replaced by a block of 8 bytes, because such files do not have the notion of words. Figs. 20 (a) and (b) compare the compression efficiencies for PDF and JPEG image files. As shown in the figures, SyncCoding compresses 8.71%, 18.53%, 25.81%, and 40.02% more for PDF files and 0.72%, 1.22%, 1.94%, and 3.01% more for JPEG files compared to Brotli, LZMA, Deflate, and Neptune. We observe that the current implementation of SyncCoding still compresses a non-negligible amount of data for both formats, but we also observe that it is our important future work to develop a more efficient compression technique as a substitute of Brotli or LZMA especially for compressed image files.



(a) The performance of the compression techniques for PDF files.



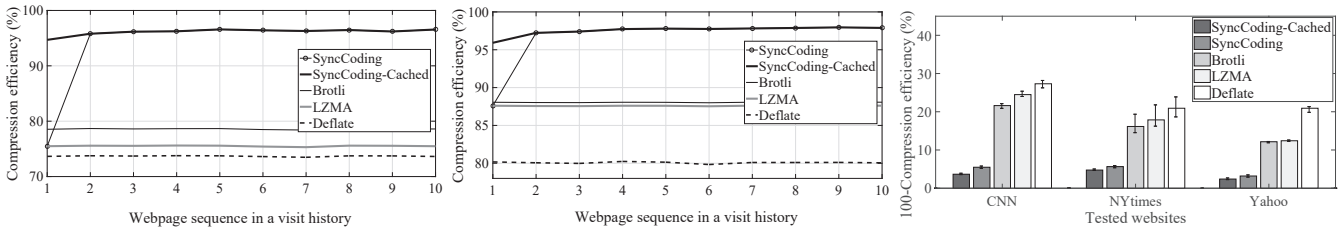
(b) The performance of the compression techniques for JPEG images.

Fig. 20. The compression efficiencies of SyncCoding, Brotli, LZMA, Deflate, and Neptune with the average chunk size of 8kB for PDF and JPEG image files.

6.3 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



(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. 21. Compression efficiencies of SyncCoding, SyncCoding-Cached and three other encoding techniques for the webpages 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.

newly invoked 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. 21 (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. 21 (a) shows that SyncCoding does not show any advantage over LZMA when there is no previous webpage to use, i.e., for the first webpage. However, from the second webpage onward, 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. 21 (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. 21 (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 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. 21 (c). The figure confirms that from the perspective of the compressed size, the improvement of SyncCoding-Cached over Brotli, LZMA, and Deflate are on average 78.4%, 80.3%, and 84.3% 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. 22 shows the compression ratio and efficiency

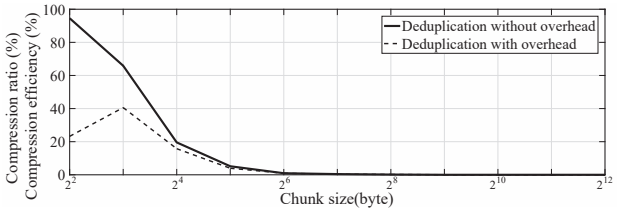


Fig. 22. Compression ratio and compression efficiency of Deduplication without and with overhead for various chunk sizes on a CNN webpage with 10 reference pages.

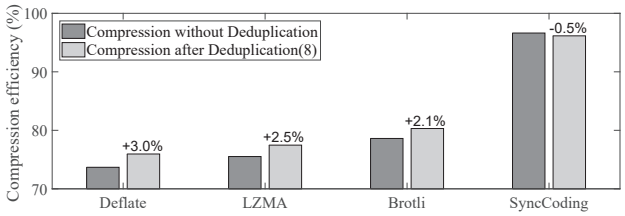


Fig. 23. The performance of compression techniques when combined with Deduplication whose chunk size is 8 bytes.

with and without referencing overhead. Fig. 22 shows that Deduplication achieves its maximum compression efficiency of about 40.59% when the chunk size is 8 bytes, but this is still far below 96.63% from SyncCoding.

We also test the compression efficiencies of SyncCoding and other techniques with ten reference pages and its best chunk size in Fig. 23. It shows Deduplication helps other encoding techniques by about 2.53% on average but makes SyncCoding even worse by about 0.5%. 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.

Motivated by this web browsing use case in which program codes are shown to be a good application of SyncCoding, we further evaluate the efficacy of SyncCoding for the cloud code-hosting scenario. For the evaluation, we chose Github, one of the most popular cloud code-hosting services, and downloaded files from three popular Github projects, Keras [52], Tensorflow [53], and Pytorch [54] [7] The total sizes of files downloaded from Keras, Tensorflow, and Pytorch projects are 110MB, 880MB, and 730MB, re-

7. The downloaded release of Keras, Tensorflow, and Pytorch are v2.2.4, v1.12.3, and v1.1.0, respectively.

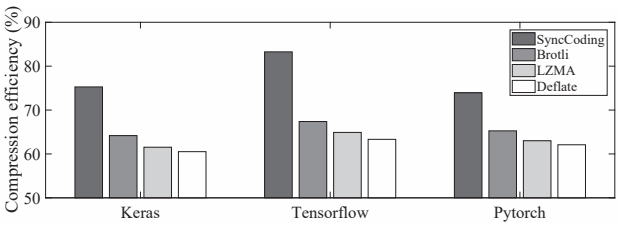


Fig. 24. Compression efficiency for program codes of three open source projects on Github, Keras, Tensorflow, and Pytorch

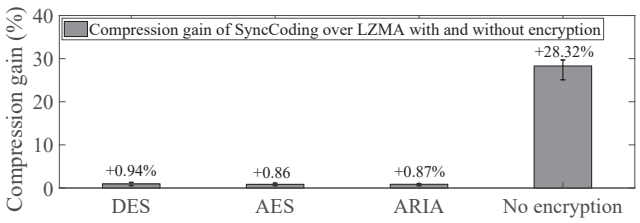


Fig. 25. The compression gain of SyncCoding over LZMA with 90% confidence intervals for three encryption algorithms and for no data encryption.

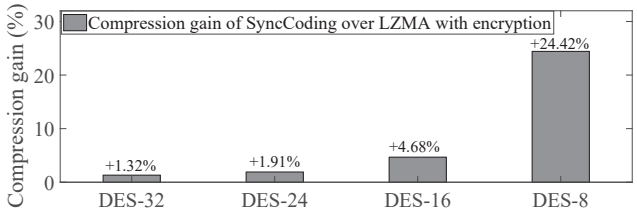


Fig. 26. The compression gain of SyncCoding over LZMA for DES with different block sizes.

spectively. Each project has several repositories, and each repository consists of various program codes written in many kinds of programming languages such as Java, C++, and Python. They also include images, docs, and PDF files. In our evaluation, SyncCoding is assumed to utilize all the files included in the older half of the repositories in each project to compress the files in the remaining half of the repositories. This emulates a scenario that the programmers participating in a project exploit SyncCoding with the previously shared or stored repositories as references when they make a new repository to share and store their program codes in the project. For these three projects, as shown in Fig. 24, SyncCoding on average compresses more about 33.2%, 37.5%, and 38.1% than Brotli, LZMA, and Deflate in the perspective of the compressed size, respectively. Since the codes would be written in a similar style by the programmers involved in a project and each repository may be related to each other in terms of the contents, as expected, SyncCoding shows superior performance over other compression techniques.

7 Discussion

In this section, we discuss two potential performance issues: 1) the performance of SyncCoding when it is applied to encrypted data, and 2) the operational overhead of SyncCoding when it is applied to commercialized synchronization services such as Dropbox.

7.1 SyncCoding for Encrypted Data

Most cloud storage services such as Dropbox and Google Drive store and exchange user data with encryption due to security and privacy concerns. A natural question that arises is whether SyncCoding can compress even the encrypted data more or not? If the repeated patterns of data inherent in the file before it is encrypted can be preserved in the encrypted file, SyncCoding may still be possible to compress the encrypted data more compared to other compression algorithms. In such a case, SyncCoding encoder and decoder can be implemented in network proxies located in edge servers and can improve the efficiency of data transmission without having any modification in the existing data synchronization applications.

To evaluate the efficacy of SyncCoding over encrypted data, we test the encryption algorithms, DES, AES, and ARIA, explained as follows. DES (Data Encryption Standard) [55], [23], [56] is a symmetric encryption algorithm whose encryption and decryption keys are the same, which had been used from 1975 as an encryption standard in the US. It has relatively small key size of 56 bits. AES (Advanced Encryption Standard) [57], [58], developed by NIST (National Institute of Standard and Technology) in 2001, is also an encryption standard in the US which has been widely used for applications such as Dropbox and Google Drive. It uses variable key sizes from 128 to 256 bits and adopts Rijndael algorithm [59], which uses substitution and permutation in each block encryption round. ARIA (Academy Research Institute Agency) [60], [61] is a block encryption algorithm that also uses variable key size like AES. ARIA is an encryption standard in South Korea. Both AES and AIRA use symmetric keys. We here focus only on symmetric algorithms because asymmetric algorithms, whose encryption and decryption keys are different, are not widely used due to their much slower encryption and decryption performance.

We reuse the same RFC dataset as in the cloud data sharing scenario in Section 6.2. For the evaluation of SyncCoding, we repeatedly choose one random target document from the dataset and used k^* most similar references from the remaining documents for one hundred times. Fig. 25 shows the average compression gain of SyncCoding over LZMA with 90% confidence interval for compression over no encryption and for compression after applying three encryption algorithms. When the data is compressed after encryption, the gain of SyncCoding over LZMA for DES, AES, and ARIA in the compressed size are about 0.94%, 0.86%, and 0.87% while the compression efficiency of SyncCoding for DES, AES, and ARIA are about 0.9%, 0.64%, and 0.66%, respectively. When the data is compressed before encryption, the gain of SyncCoding over LZMA is about 28.3% while SyncCoding compresses about 78.8%. Interestingly, SyncCoding as well as LZMA merely reduce the size of the encrypted data. This is because of the features of modern encryption techniques: *Confusion* and *Diffusion*. *Confusion* makes it more difficult to guess the contents of the original data. *Diffusion* makes it harder to find the pattern of the encrypted data. By doing so, they transform original data into high-entropy data. Since the repeated patterns of original data are well hidden inside the encrypted data, compression over encryption is in general not effective.

To solve this problem, we apply and test a similar method in deduplication to improve the compression efficiency of SyncCoding for encrypted data. We first segregate data (i.e., file) into small blocks, store the concatenated encrypted blocks as the pseudo encrypted data, and then apply SyncCoding over these pseudo encrypted data. To evaluate this method, we test DES with 8, 16, 24, and 32 bytes of block sizes for the same dataset as in Fig. 25 for one hundred times. Fig. 26 shows the average compression gain of SyncCoding over LZMA for block encrypted data with DES. The SyncCoding for DES with the block sizes of 8, 16, 24, and 32 bytes on average compresses data more by 24.42%, 4.68%, 1.91%, and 1.32% with k^* references over LZMA, while LZMA itself compresses data by 29.77%, 2.78%, 1.82%, and 1.19%, respectively. On top of this block-level encryption, data in transport between servers and clients can be additionally protected by another encryption such as TLS (transport layer security). According to [62], Dropbox also stores block-encrypted data and transmit data additionally with TLS. Overall, applying SyncCoding in the network path (i.e., after TLS) is impractical, but applying SyncCoding in a block-encrypted cloud storage before data is transmitted is still a viable option.

7.2 Operational Overhead of SyncCoding

Understanding the operational overhead of SyncCoding when it is applied to commercial synchronization services such as Dropbox can be helpful in deciding how to implement and integrate SyncCoding into such services. However, it is practically not easy to quantify the overhead in such commercialized services since their databases or snapshots of databases are not accessible. Nevertheless, we can estimate the overhead logically as follows. To run SyncCoding, it first needs to index the reference candidates (e.g., up to all the files) and needs to keep the indices. This is regarded as a storage overhead. Note that assigning 10 bytes per index as in Section 5.2 can cover up to 2^{80} (roughly 10^{24}) files, so it may be sufficient for most users. Given a user who keeps 100 gigabytes of data in his or her cloud storage space with files whose average size is about 1 megabyte, the storage overhead to keep such indices becomes about 1 megabyte ($\approx 10 \times 10^5$ bytes), and the reference index overheads attached to the files with 20 references on average becomes about 20 megabytes ($\approx 10 \times 10^5 \times 20$ bytes) in total, which are both acceptable. With those indices ready, given a file SyncCoding needs to scan other files and select actual references to use for encoding. This process is subject to a trade-off between computation and storage because tagging files based on the file contents for quick classification can significantly save the scanning computation in the cost of storage overhead for tagging. If the file system is designed to tag each file with limited bytes such as 100 bytes, the storage overhead from tagging will be about 10 megabytes in total for the aforementioned use case, which is still acceptable. Note that zero file tagging that pays zero extra storage overhead is also possible, but this is never going to be a practical choice because its required computation load may be prohibitively high. Another source of overhead is from keeping track of file changes such as additions, deletions, and modifications. This operation is also subject to a trade-off between computation and storage. Immediate re-indexing of the reference candidates or re-encoding of the encoded files with the modified length-distance pairs may bring huge computation overhead, but this overhead can be intelligently mitigated by temporarily keeping copies of those untouched files in the background and by applying batch updates intermittently. This intelligence is the key to develop a practical SyncCoding-based file system and determines the major storage and computation overheads. A naive strategy is to let the file system spare a certain percentage of storage space (e.g., 1% of the cloud storage space) to keep the copies, and a more elaborated strategy may be to adjust the size of this temporary space depending on the frequency of file changes. A more radical strategy is to adopt CDC-based single instance file system as a basis for SyncCoding-based file system and redesign the length-distance pair format of SyncCoding to be compatible with CDC chunks (e.g., a distance format having a CDC chunk index with the offset bytes inside the chunk). In this strategy, re-indexing and re-encoding may substantially reduce because only the encoded files that contain chunk indices with bit changes are subject to re-computation. Designing this intelligence with high efficiency is an open question and we leave it as our future work.

8 Concluding Remarks

In this work, we propose a novel data encoding technique SyncCoding that exploits the database of previously synchronized data to improve efficiency of networking. Our experiments show that SyncCoding can reduce the energy consumption of mobile devices for data synchronization and also confirm that SyncCoding outperforms existing encoding techniques, Brotli, Deflate, and LZMA in terms of compression efficiency in two popular use cases: cloud data sharing and web browsing. SyncCoding sets up a new baseline for encoding techniques that exploit inter-data correlation.

Acknowledgements

This work was in part supported by IITP grants funded by the Korea government (MSIT) (No. 2015-0-00278, Research on Near-Zero Latency Network for 5G Immersive Service, No. 2017-0-00692, Transport-aware Streaming Technique Enabling Ultra Low-Latency AR/VR Services), NSF grants (CNS-1901057, CNS-1719371), and a grant from them Office of Naval Research (N00014-17-1-2417). *K. Lee* and *J. Lee* are the corresponding authors. A preliminary version of this work was presented at the IEEE ICNP 2017 [63].

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] S. Quinlan and S. Dorward, "Venti: A new approach to archival data storage," *USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[3] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 4, pp. 87–95, 2000.

[4] Y. Hua, X. Liu, and D. Feng, "Neptune: Efficient remote communication services for cloud backups," *IEEE INFOCOM*, 2014.

[5] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data deduplication-large scale study and system design." *USENIX Annual Technical Conference (ATC)*, 2012.

[6] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Transactions on Storage (ToS)*, vol. 7, no. 4, p. 14, 2012.

[7] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data." *USENIX Annual Technical Conference (ATC)*, 2004.

[8] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system." *USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[9] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality." *USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[10] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[11] P. Bhatotia, R. Rodrigues, and A. Verma, "Shredder: Gpu-accelerated incremental storage and computation." *USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[12] 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.

[13] M. Mahoney, "Fast text compression with neural networks," *ACM AAAI*, 2000.

[14] "Alliance for open media," https://aomedia.org/.

[15] I. Pavlov. 7z format. http://www.7-zip.org/7z.html.

[16] J. Alakuijala and Z. Szabadka, "Brotli compressed data format," Tech. Rep., 2016.

[17] P. Deutsch, "DEFLATE compressed data format specification version 1.3," Tech. Rep., 1996.

[18] C. E. Shannon, "A mathematical theory of communication," *Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[19] D. A. Huffman *et al.*, "A method for the construction of minimum-redundancy codes," *IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[20] J. Rissanen, "Generalized kraft inequality and arithmetic coding," *IBM Journal of research and development*, vol. 20, no. 3, pp. 198–203, 1976.

[21] M. Ruhl and H. Hartenstein, "Optimal fractal coding is np-hard," *IEEE Data Compression Conference (DCC)*, 1997.

[22] 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.

[23] 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*, 2000.

[24] F. S. Awan and A. Mukherjee, "LIPT: A lossless text transform to improve compression," *IEEE International Conference on Information Technology:Coding and Computing*, 2001.

[25] J. Alakuijala and Z. Szabadka. (2014) IETF Brotli compressed data format. https://tools.ietf.org/html/draft-alakuijala-brotli-01.

[26] B. Agarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese, "Endre: An end-system redundancy elimination service for enterprises." *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010.

[27] Improving the deduplication flow when uploading to Google Drive. https://gsuiteupdates.googleblog.com/2016/09/improving-deduplication-flow-when.html.

[28] Dropbox. https://www.dropbox.com/.

[29] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang, "Fastcdc: a fast and efficient content-defined chunking approach for data deduplication." *USENIX Annual Technical Conference (ATC)*, 2016.

[30] J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein, "Delta encoding in HTTP," Tech. Rep., 2001.

[31] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "Wan-optimized replication of backup datasets using stream-informed delta compression," *ACM Transactions on Storage (ToS)*, vol. 8, no. 4, p. 13, 2012.

[32] Y. Cui, Z. Lai, X. Wang, and N. Dai, "Quicksync: Improving synchronization efficiency for mobile cloud storage services," *IEEE Transactions on Mobile Computing (TMC)*, vol. 16, no. 12, pp. 3513–3526, 2017.

[33] Z. Tu and S. Zhang, "A novel implementation of jpeg 2000 lossless coding based on lzma," *IEEE International Conference on Computer and Information Technology*, 2006.

[34] 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.

[35] I. Pavlov. Lzma sdk. http://www.7-zip.org/sdk.html.

[36] N. Dehak, R. Dehak, J. Glass, D. Reynolds, and P. Kenny, "Cosine similarity scoring without score normalization techniques," *Odyssey: The Speaker and Language Recognition Workshop*, 2010.

[37] S. Kullback and R. A. Leibler, "On information and sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[38] IETF RFC Index. https://www.ietf.org/rfc.html.

[39] R. Turek. What YouTube Looks Like In A Day [Infographic]. https://beat.pexe.so/what-youtube-looks-like-in-a-day-infographic-d23f8156e599#.wbu0v1h2z.

[40] "Monsoon solutions," http://msoon.github.io/powermonitor/.

[41] M. Stojanovic and J. Preisig, "Underwater acoustic communication channels: Propagation models and statistical characterization," *IEEE Communications Magazine*, vol. 47, no. 1, pp. 84–89, January 2009.

[42] R. Ludwig and J. Taylor, "Voyager telecommunications (jet propulsion laboratory)," https://descanso.jpl.nasa.gov/DPSummary/Descanso4–Voyager_new.pdf.

[43] S. Dey, D. Mohapatra, and S. Archana, "An approach to calculate the performance and link budget of leo satellite (iridium) for communication operated at frequency range (1650-1550) mhz," *International Journal of Latest Trends in Engineering and Technology*, vol. 4, no. 4, November 2014.

[44] J. Hopper *et al.*, "Using the linux cpufreq subsystem for energy management," *IBM blueprints*, 2009.

[45] T. L. foundation, "Network emulator," https://wiki.linuxfoundation.org/networking/netem.

[46] J.-l. Gailly and M. Adler. zlib compression library. http://www.zlib.net.

[47] Brotli compression format. https://github.com/google/brotli.

[48] Google to boost compression performance. http://www.computerworld.com/article/3025456/web-browsers/google-to-boost-compression-performance-in-chrome-49.html.

[49] Opendedup – opensource dedupe to cloud and local storage. http://opendedup.org/odd/.

[50] C. Constantinescu, J. Glider, and D. Chambliss, "Mixing deduplication and compression on active data sets," *IEEE Data Compression Conference (DCC), 2011*, pp. 393–402, 2011.

[51] I. Drago, "Understanding and monitoring cloud services," *Technical report, University of Twente*, 2013.

[52] "Keras open source project on github," https://github.com/keras-team.

[53] "Tensorflow open source project on github," https://github.com/tensorflow.

[54] "Pytorch open source project on github," https://github.com/pytorch.

[55] D. E. Standard, "Data encryption standard," *Federal Information Processing Standards Publication*, 1999.

[56] "Des source code (3-des / triple des) - mbed tls (previously polarssl)," https://tls.mbed.org/des-source-code.

[57] F. P. Miller, A. F. Vandome, and J. McBrewster, "Advanced encryption standard," 2009.

[58] "Aes source code (advanced encryption standard) - mbed tls (previously polarssl)," https://tls.mbed.org/aes-source-code.

[59] T. Jamil, "The rijndael algorithm," *IEEE potentials*, vol. 23, no. 2, pp. 36–38, 2004.

[60] "Aria block encryption algorithm - kisa," https://seed.kisa.or.kr/iwt/ko/sup/EgovAriaInfo.do.

[61] ARIA source code - KISA. http://seed.kisa.or.kr/iwt/ko/index.do;jsessionid=DDB1FEE78B04222BFCEA4E93A8264485.

[62] Dropbox, "Dropbox security architecture," https://www.dropbox.com/business/trust/security/architecture.

[63] W. Nam, J. Lee, and K. Lee, "Synccoding: A compression technique exploiting references for data synchronization services," *IEEE 25th International Conference on Network Protocols (ICNP)*, 2017.

Wooseung Nam (S'17) received his B.S. and M.S. degrees in computer science and engineering from the Ulsan National Institute of Science and Technology (UNIST), Ulsan, South Korea, in 2017 and 2019, respectively. Since 2019, he is currently pursuing the Ph.D. degree at UNIST. His research interest includes designing an extremely-compressed data transfer system and a latency mobile computing platform with energy efficiency.

Joohyun Lee (S'11-M'14) received his B.S. and Ph.D. degrees in the Department of Electrical Engineering from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2008 and 2014, respectively. Since 2014, he was a postdoctoral researcher in the Department of Electrical and Computer Engineering at the Ohio State University. He is an assistant professor in the Division of Electrical Engineering at Hanyang University, Korea, since 2018. He received IEEE William R. Bennett Prize Paper Award in 2016, given to the best original paper published in IEEE/ACM Transactions on Networking in the previous three calendar years. His research interests are in the areas of context-aware networking and computing, mobility-driven cellular traffic offloading, energy-efficient mobile networking, protocol design and analysis for delay-tolerant networks, and network economics and pricing.

Ness B. Shroff (S'91-M'93-SM'01-F'07) received the Ph.D. degree in electrical engineering from Columbia University in 1994. He joined Purdue University immediately thereafter as an Assistant Professor with the School of Electrical and Computer Engineering. At Purdue, he became a Full Professor of ECE and the director of a universitywide center on wireless systems and applications in 2004. In 2007, he joined The Ohio State University, where he holds the Ohio Eminent Scholar Endowed Chair in networking and communications, in the departments of ECE and CSE. He holds or has held visiting (chaired) professor positions at Tsinghua University, Beijing, China, Shanghai Jiaotong University, Shanghai, China, and IIT Bombay, Mumbai, India. He has received numerous best paper awards for his research and is listed in Thomson Reuters' on The World's Most Influential Scientific Minds, and is noted as a Highly Cited Researcher by Thomson Reuters. He also received the IEEE INFOCOM Achievement Award for seminal contributions to scheduling and resource allocation in wireless networks. He currently serves as the steering committee chair for ACM Mobihoc and Editor at Large of the IEEE/ACM Transactions on Networking.

Kyunghan Lee (S'07-M'10) received his B.S., M.S., and Ph.D. degrees from the department of electrical engineering at KAIST (Korea Advanced Institute of Science and Technology), Daejeon, South Korea, in 2002, 2004, and 2009, respectively. He is currently an associate professor in the school of electrical and computer engineering at UNIST (Ulsan National Institute of Science and Technology), Ulsan, South Korea where he has led mobile systems and networking lab since 2012. His research interests include low-latency networking, low-power mobile computing, mobile machine learning, and context-aware networking. He received two IEEE ComSoc William R. Bennett Prize in 2013 and 2016, given to the best original paper published in the IEEE/ACM Transactions on Networking in the previous three years. He has served on the program committee of a number of leading conferences including the IEEE INFOCOM, ACM MobiSys, and ACM MobiHoc; and has also served on the organizing committee of renowned conferences such as ACM MobiSys, IEEE SECON, and IEEE WCNC.