CoRE: Cooperative End-to-End Traffic Redundancy Elimination for Reducing Cloud Bandwidth Cost

Lei Yu, Haiying Shen, Karan Sapra, Lin Ye and Zhipeng Cai

Abstract—The pay-as-you-go service model impels cloud customers to reduce the usage cost of bandwidth. Traffic Redundancy Elimination (TRE) has been shown to be an effective solution for reducing bandwidth costs, and thus has recently captured significant attention in the cloud environment. By studying the TRE techniques in a trace driven approach, we found that both short-term (time span of seconds) and long-term (time span of hours or days) data redundancy can concurrently appear in the traffic, and solely using either sender-based TRE or receiver-based TRE cannot simultaneously capture both types of traffic redundancy. Also, the efficiency of existing receiver-based TRE solution is susceptible to the data changes compared to the historical data in the cache. In this paper, we propose a Cooperative end-to-end TRE solution (CoRE) that can detect and remove both short-term and long-term redundancy through a two-layer TRE design with cooperative operations between layers. An adaptive prediction algorithm is further proposed to improve TRE efficiency through dynamically adjusting the prediction window size based on the hit ratio of historical predictions. Besides, we enhance CoRE to adapt to different traffic redundancy characteristics of cloud applications to improve its operation cost. Extensive evaluation with several real traces show that CoRE is capable of effectively identifying both short-term and long-term redundancy with low additional cost while ensuring TRE efficiency from data changes.

Index Terms—traffic redundancy elimination, cloud computing, network bandwidth, bandwidth cost, end-to-end

1 Introduction

Cloud computing is an emerging IT paradigm that provides utility computing by a pay-as-you-go service model [1]. More and more organizations are moving their businesses to the cloud to provide services like video streaming, Web and social networking. With modern virtualization technologies, the cloud provides resource elasticity [2], which enables the capacity of cloud applications to scale up and down on demand to adapt to the changes in workloads. However, the deployment of web and video services in cloud drive increasing egress bandwidth demands due to data access from a large amount of users. A well-known example is Netflix [3], which has hosted its video-streaming service in the cloud with Amazon Web Services [4] since 2009. The cost of cloud hosting services can vastly increase due to the usage-based bandwidth pricing. Amazon EC2 charges the data transfer out to Internet can be $0.155 per GB for first 10TB/month [4], and nowadays it is easy for a cloud server to have 1TB monthly data transfer. Thus, the bandwidth cost has become a serious concern for the cloud application deployment and received a lot of attention [5], [6].

In order to reduce the bandwidth cost for data transfer from the cloud, Traffic Redundancy Elimination (TRE) technologies have being exploited [7] to reduce the bandwidth usage by eliminating the transmission of duplicate information. In traditional TRE solutions [8], [9], the sender detects the duplicate content by comparing the outgoing data with its local cache which stores previously transmitted packets, and then sends the reference of duplicate data to the receiver instead of raw data. The receiver fetches the data from its local cache by the reference lookup. These methods need to maintain fully synchronized caches at both the sender and receiver in order to store previously transmitted/received packet payload. However, the workload distribution and migration for elasticity in the cloud can lead to frequent changes of service points for the clients, which makes cache synchronization difficult and may result in degraded TRE efficiency. Thus, traditional TRE solutions are ill-suited to the cloud environment, as noted in [7]. Besides, considering the usage-based resource pricing of the cloud, the usage of computation, storage and bandwidth resources for running TRE software may eradicate the bandwidth cost savings obtained by TRE. Without cloud elasticity, the load incurred by TRE at the servers may negatively affect the performance of the cloud applications. With the elastic capacity scaling in cloud, the overhead of TRE may trigger the scaling up of the application's capacity through adding more servers, which increases the operation cost of the application. Thus, ensuring low resource expense is necessary for TRE in the cloud environment.

To address the above issues of TRE in cloud, a receiver-based TRE solution named PACK [7], [10] has been proposed. In PACK, once a client receives a data chunk that already exists in its local cache, it is expected that the future incoming data would also be duplicate. Thus, the client predicts the future coming data chunks and notifies the cloud server with the signatures of the predicted chunks. The server compares the predicted signatures received from the client with the signatures of outgoing chunks and confirms the correctly predicted chunks, which then do not need to be transferred. Without maintaining the clients' status at the server, PACK effectively handles the cloud elasticity. By

- *A preliminary version of this paper appears in the proceedings of IEEE ICNP 2012.*
- *Lei Yu is with the School of Computer Science, Georgia Institute of Technology, Atlanta, GA, 30332. E-mail: lyu79@gatech.edu*
- *Zhipeng Cai is with the Department of Computer Science, Georgia State University, Atlanta, GA, 30303. E-mail: zcai@gsu.edu*
- *Haiying Shen and Karan Sapra are with the Department of Electrical and Computer Engineering, Clemson University, Clemson, SC, 29631. E-mail: {shenh, ksapra}@clemson.edu*
- *Lin Ye is with the Department of Computer Science and Technology, Harbin Institute of Technology, Harbin, China, 150001. E-mail: yelin@nis.hit.edu.cn*

predicting future traffic redundancy at clients, PACK offloads most computation and storage cost to clients, and thus greatly reduces the TRE cost in cloud servers.

On the other hand, previous studies on network traffic redundancy [11] indicate two types of redundancy, referred to as *short-term traffic redundancy* (repetition in minutes) and *long-term traffic redundancy* (repetition in hours or days) according to the time scale of repetition occurrence. It is found that vast majority of data matches between the traffic and the cache occur for data less than 150 bytes and have high degree of temporal locality (e.g., 60% within 100 seconds), while popular chunks can recur with a time difference as large as 24 hours [11]. By studying real traces, we demonstrate that the short-term and long-term redundancy may co-exist in the network traffic, and PACK can effectively capture long-term redundancy in the traffic between a server and a client but fails to capture short-term redundancy. Because the clients can cache a large amount of historical packet data on large-size persistent storages (e.g., disks) in the long term, PACK is able to eliminate the traffic content repeated in a long period with the clients' traffic redundancy predictions. However, because PACK matches data chunks at average size of 8KB, it cannot detect the short-term redundancy that appears at fine-granularity (e.g., the data size about 150 bytes). Since a large portion of redundancy is found in short-term time scale, PACK cannot exploit the full redundancy in the network traffic. However, using a small chunk size in PACK is not an acceptable solution to the problem, since otherwise the bandwidth cost for transmitting chunk predictions would eradicate the bandwidth savings from eliminating chunk transmissions.

In this paper, we aim to design an efficient TRE solution for services that are deployed in the cloud and dominated by the data transfer from the cloud servers to the clients. We propose a Cooperative end-to-end TRE solution, named as CoRE, with capability of removing both short-term and long-term redundancy such that traffic redundancy can be eliminated to the highest degree. CoRE involves two layers of cooperative TRE operations. The first-layer TRE performs *prediction-based Chunk-Match* similar to PACK [7] to capture long-term traffic redundancy. The second-layer TRE identifies maximal duplicate substrings within an outgoing chunk compared with the previously transmitted chunks in a local chunk cache at the sender, referred to as *In-Chunk Max-Match*. If the redundancy detection at the first-layer fails, CoRE turns to the second-layer to identify finer-granularity redundancy, i.e., short-term redundancy, inside chunks.

With the consideration of the requirements of cloud environment for TRE, CoRE incorporates several careful designs and optimizations to reduce CoRE's operation cost and improve its TRE efficiency. First, CoRE uses a temporary small chunk cache for each client to reduce the storage cost of In-Chunk Max-Match at a server. It requires cache synchronization between the sender and receiver, but it only detects the traffic redundancy in short-term and thus cloud elasticity does not have much effect on the TRE efficiency. Second, a single-pass scanning algorithm is used in CoRE to determine chunk boundaries in the TCP stream while at the same time obtaining the fingerprints within chunks used to find maximal duplicate substrings in chunks. It efficiently integrates two layers of TRE operations. Third, existing prediction-based design in PACK requires that the predicted chunk exactly appears at the expected position of TCP stream. Even a small offset of the outgoing data at the sender, compared with the data cached at the receiver, can invalidate the predictions and greatly degrade TRE efficiency. To ensure prediction efficiency against data changes, we propose an improved prediction-based TRE. In CoRE, the sender divides the outgoing data into chunks in the same way as the receiver, and compares the signatures of outgoing chunks with all chunk predictions recently received from the receiver regardless of their expected positions in the TCP stream. At the receiver, an adaptive prediction algorithm is proposed, which dynamically decides the prediction window size, i.e., the number of subsequent chunks to predict based on the hit ratio of historical chunk predictions. In this way, CoRE gains better TRE efficiency than PACK.

The performance of CoRE is evaluated with several traffic traces of the real-world applications. Our trace-driven evaluation results show that the distribution of traffic redundancy at long-term and short-term time scales varies with different applications, which further causes different CPU costs for CoRE at the cloud servers. Some applications may have dominant long-term redundancy in traffic, while some have dominant short-term redundancy. In CoRE, detecting short-term traffic redundancy has higher computation cost than detecting long-term traffic redundancy. Thus, in order to improve the benefit-cost ratio between bandwidth savings and TRE operation costs, we further propose an adaptive solution to discover the dominant redundancy in the traffic and intelligently decide which layer of TRE in CoRE to be enabled.

In summary, the contributions of this paper are listed as follows:

1. By a real trace driven study, we identify the limitations of existing end-to-end TRE solutions for capturing short-term and long-term redundancy of data traffic.

2. We propose a two-layer TRE scheme, named CoRE, in order to effectively detect both long-term and short-term traffic redundancy.

3. Several optimizations are proposed in CoRE to reduce the operating cost, improve the prediction efficiency and increase the benefit-cost ratio of TRE, respectively.

4. We implement CoRE and quantify its benefits and costs based on extensive experiments by using several real-world network traffic traces.

The rest of the paper is organized as follows. Section 2 describes existing TRE solutions. Section 3 discusses TRE solutions for cloud and their limitations. Section 4 presents our end-to-end TRE solution CoRE in detail. Section 6 presents our implementation. In Section 7, we evaluate CoRE and compare it with PACK by extensive experiments using several traffic traces.

2 RELATED WORK

Since significant redundancy has been found in the network traffic [8], [11], due to repeated accesses to the same or similar data objects from the Internet end-users, several TRE techniques have been proposed to suppress duplicate data from the network transfers to reduce the bandwidth usage. A protocol-independent packet-level TRE solution was first proposed in [8]. In this work, the sender/receiver maintains a local cache respectively which stores recently transferred/received packets. The sender computes the *Rabin fingerprints* [12] for each packet by applying a hash function to each 64 byte sub-string of the packet content, and selects a subset of representative fingerprints as to the packet content. For an outgoing packet, the sender checks whether its representative fingerprints have appeared in earlier cached packets. If yes, the sender identifies the maximal duplicate region around each matched fingerprint and replaces the region with a fixed-size

pointer into the cache to compress the packet. To decode compressed data, the receiver replaces the pointer with the corresponding referred data in its local cache. Several commercial vendors have developed such protocol-independent TRE algorithms into their "WAN optimization" middle-boxes [13], [14], [15] placed at either end of a WAN link. The successful deployment of TRE solutions in enterprise networks motivated the exploration of TRE deployment at routers across the entire Internet [16], [17]. In [16], redundancy-aware intra- and inter-domain routing algorithms are proposed to further enhance network-wide TRE benefits. In [17], an architecture for network-wide TRE is proposed, which allocates encoding and decoding operations across network elements and perform redundancy elimination in a coordinated manner.

The previous work on traffic redundancy [11] found that over 75% of redundancy was from intra-host traffic, which implies that an end-to-end solution is very feasible for redundancy elimination. Accordingly, a sender-based end-to-end TRE [9], named EndRE, was proposed for the enterprise networks. By maintaining a fully synchronized cache with each client at the server, EndRE offloads most processing effort and memory cost to servers and leaves the clients only simple pointer lookup operations. It suppresses duplicate byte strings at the order of 32-64B.

A receiver-based end-to-end TRE, PACK, is proposed for cloud environment [7], [10]. At the receiver, the incoming TCP data stream is divided into chunks. The chunks are linked in sequence, which forms a *chain*, and stored into a local chunk store. The receiver compares each incoming chunk to the chunk store. Once finding a matching chunk on a chain, it retrieves a number of subsequent chunks along the chain as the predicted chunks in the future incoming data. The signatures of the retrieved chunks and their expected offsets in the incoming data stream are sent in a PRED message to the sender as a prediction for the sender's subsequent outgoing data. Figure 1 briefly describes the PACK algorithm. Once finding a match with an incoming chunk $[XYZA]$ in the chunk store, the receiver sends to the sender the triples of signature, expected offset and length of following chunks $[ABCD]$, $[BCFA]$ and $[HIJK]$ in the same chain, i.e., $(Sign.1, n, 4)$, $(Sign.2, n+4, 4)$ and $(Sign.3, n+8, 4)$ as predictions. To match data with a prediction, the sender computes SHA-1 over the outgoing data at the expected offset with the length given by the prediction, and compares the result with the signature in the prediction. Upon a signature match, the sender sends a PRED-ACK message to the receiver to tell it to copy the matched data from its local storage. In this way, PACK offloads the computational effort of TRE from the cloud servers to the clients, and thus avoids the additional computational and storage costs incurred by TRE at the cloud to outweigh the bandwidth saving gains.

The efficiency of PACK for capturing the long-term redundancy is susceptible to data changes, which can happen frequently in various cloud applications involving frequent data update such as collaborative development [18] and data storage [19]. In PACK, the sender determines the data chunk to match based on the position and the length both given by the prediction. The prediction is true only if the predicted chunk exactly appears at the expected position in the byte stream. Thus, even a small position offset in the sender's outgoing data due to data insertion or deletion can invalidate all the following predictions. For example, in Figure 1, 'E' is inserted into the sender's data. For the outgoing data $[ABCDBCEFAHIJK]$, the PACK sender matches $Sign.1$ with the data chunk at the expected offset n with given length 4, which is $[ABCD]$. Similarly, it will match $Sign.2$ with the data chunk at the expected offset $n+4$ with length 4 that is $[BCEF]$, which causes

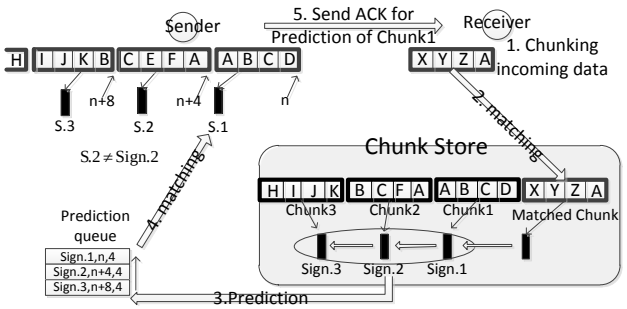


Fig. 1. A brief description for PACK algorithm. $s.\#$ and $sign.\#$ are chunk signatures. $n+\#$ is the expected offset of a predicted chunk in TCP stream.

a matching failure. Therefore, in PACK once a matching failure occurs, the following predictions are abandoned, even though the data chunk $[HIJK]$ can match with the third prediction. In CoRE, an improved prediction-based TRE is proposed. The CoRE sender performs the content-based chunking as the receiver, and finds a match by computing the signatures of the chunks and looking up them in a prediction store that keeps recently received predictions. In this way, CoRE achieves resiliency against data changes. The further explanation can be found in Section 4.3.

Note a hybrid mode of sender-based and receiver-based TRE is also proposed for PACK [7], but it is completely different from two-layer design of CoRE. The hybrid solution simply deploys two separate TRE schemes together and switches from the receiver-based TRE to the sender-based TRE if data changes make predictions inefficient. Since at any time only one scheme works, the hybrid solution cannot capture the long-term and short-term redundancy simultaneously. It also cannot adaptively and flexibly distribute TRE effort among two separate schemes according to the characteristics of traffic redundancy. In contrast, with joint work of two layers of TRE, CoRE is able to simultaneously and adaptively capture the long-term and short-term redundancy.

Besides the applications in wired networks, TRE also has been explored in wireless and cellular networks [20]. REfactor [20] aims to remove the traffic redundancy at the sub-packet level by IP-layer RE with content overhearing. It leverages overheard data by estimating the overhearing probability of data for the receiver and removing the chunks that highly likely have been overheard. Celleration [21] is a gateway-to-mobile TRE system for the data-intensive cellular networks. The gateway predicts the future chunks to a mobile device and eliminates the transmission of the chunks whose predictions are confirmed by the mobile devices, which indicates the chunks are cached on the device. The measurements in [22] shows that TCP-level TRE saves significant bandwidth consumption for 3G traffic. In [23], Zhang et al. surveyed the existing protocol-independent TRE techniques and systems developed for wired and wireless networks respectively. In this paper, we focus on the TRE for wired networks in the context of the cloud computing.

3 TRE SOLUTIONS FOR CLOUD AND LIMITATIONS

In this section, with a trace-driven approach, we analyze the limitations of the existing end-to-end TRE solutions (i.e., received-based [7] and sender-based TRE [9]) for cloud in capturing long-term and short-term traffic redundancy, and propose our design goals for TRE in cloud.

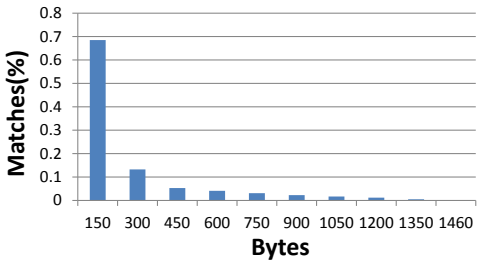


Fig. 2. Length distribution of matched data.

3.1 Limitations in Capturing Short-term and Long-term Traffic Redundancy

3.1.1 Short-term redundancy

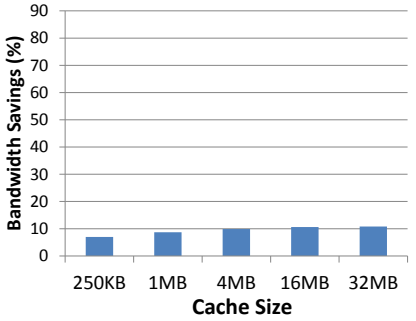
The real Internet traffic trace we used was captured from an access link from a large university to the backbone. The trace is 120 seconds long and contains 1.9GB HTTP traffic between 8277 host pairs including payloads. The detailed description is given in Table 1. For every host pair of sender and receiver, we detect redundancy in the traffic from server to client, by received-based approach PACK [7] and a sender-based TRE similar to EndRE's Max-Match [9], respectively. Our experimental results show that PACK detected little redundancy, totaling 1.4MB in 1.9GB traffic. In contrast, the sender-based TRE using a small cache size of 250KB detected 5% redundancy, which amounts to 114MB.

Figure 2 shows the length distribution of matched data found by the sender-based TRE in our traffic trace. We can see that about 70% of the matches have a size no more than 150 bytes, while occurring in 120 seconds long trace. Such results confirm that significant redundancy appears in short-term time scale and mostly with size at the order of hundred bytes. The large difference on TRE efficiency between PACK and the sender-based TRE is due to their different ability of capturing short-term redundancy in the traffic; PACK cannot capture short-term redundancy while sender-based TRE can. The reason is that PACK uses a large chunk size of 8KB for efficiency, which causes it unable to identify finer-granularity content repetitions and hence misses the short-term redundancy. For example, in Figure 1, the chunk $[CEFA]$ would be sent without any compression due to the false prediction. However, $[CEFA]$ has overlaps on 'A' and 'C' with the previous sent chunk $[ABCD]$. PACK misses such short-term repetition at fine-granularity.

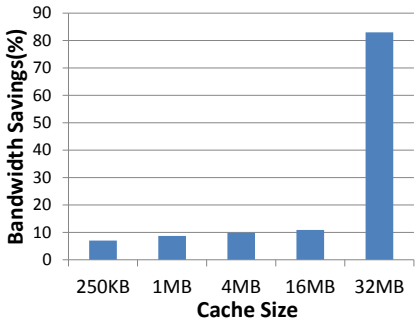
3.1.2 Long-term redundancy

Eyal et al. [7] have found significant long-term redundancy in YouTube traffic, online social network service traffic and some real-life workloads, where data repetition can occur over 24 hours or months. To study the efficiency of existing TRE solutions for capturing long-term redundancy, we use the same approach as in [7] to synthesize a real-life workload traffic with long-term redundancy, by downloading 40 Linux source files one by one in their release order.

To capture long-term traffic redundancy, the sender-based TRE needs to synchronously maintain a large persistent cache at both the sender and receiver. To verify this point, we investigate the TRE efficiency of two types of sender-based TRE using temporary cache and persistent cache, respectively. In the persistent cache based approach, the client maintains a cache which keeps the most recently received packets during the entire period of downloading 40 files, in order to detect data repetition across successively downloaded files. In the temporary cache based approach, for each file downloading, the client will allocate a new cache which just temporarily exists during the period of downloading one file. When a file download completes, the cache is released and no historical information is stored. Thus, such approach can only capture the redundancy within each file itself.



(a) Temporary cache.



(b) Persistent cache.

Fig. 3. Detected redundancy in Linux source traffic by the sender-based TRE.

Figure 3 presents the bandwidth savings, i.e., the percentage of total redundant bytes in the workload traffic detected by the sender-based TRE with various cache sizes. Figure 3(a) shows the redundancy detected by the temporal cache based approach. With a cache size of 250KB, about 7% redundancy can be detected inside each file. Increasing cache size only yields diminishing returns, which indicates that most data repetitions occur within the traffic of 250KB window and they actually compose the short-term traffic redundancy at the time scale of 250KB/(downloading rate), less than 2 seconds in our experiment.

Figure 3(b) shows the redundancy detected by the persistent cache based approach. Comparing Figure 3(b) with Figure 3(a), we can see a significant difference between the temporary cache and persistent cache approaches. As opposed to 10% redundancy detected by using 32MB temporary cache, using 32MB persistent cache can detect about 80% traffic redundancy for every downloaded file except the first one. The reason is that, since the sizes of Linux source files are between 21MB and 31MB, the 32MB cache can store a whole file previously downloaded such that data repetition across successive files can be completely detected and thus more redundancy is eliminated. The redundancy across different versions of Linux source actually is a long-term redundancy, because the download of a source file most likely occurs when a new version is released, usually in days or months. Therefore, persistently keeping past transferred data is essential for capturing long-term redundancy. Besides, the steep rise of detected redundancy at 32MB in Figure 3(b) also indicates that the cache needs to be large enough for effectively capturing long-term traffic redundancy.

In our experiment, PACK detected 54% traffic redundancy in the synthesized workload, less than 80% detected by sender-based TRE with 32MB persistent cache but much higher than 10% obtained with smaller persistent cache size. Unlike the sender-based TRE, PACK cannot detect the significant short-term redundancy at fine-granularity. Thus, the sender-based TRE with 32MB persistent cache obtains much more redundancy than PACK. However, because the cloud server has limited resources and usually serves a large number of users, it is not practical for the server to run sender-based TRE with a large persistent cache for every client. But using smaller cache size, the sender-based TRE detects much less redundancy than PACK. As a result, the sender-based TRE is not suitable to efficiently remove the long-term traffic redundancy for cloud environment. In contrast, PACK can efficiently capture the long-term redundancy for cloud by offloading large size caching and computation effort from the server to clients.

3.2 Design goals

Based on the above analysis, we conclude that solely using either sender-based or receiver-based solution would fail to eliminate a large amount of redundancy. With the advance of various applications and services in clouds, significant short-term and long-term redundancy can concurrently appear in the network traffic. Thus, in this paper, we aim to design a TRE scheme, which is able to eliminate more traffic redundancy than PACK by capturing both long-term and short-term redundancy. To be apt for the cloud environment, our scheme also needs to ensure low computation and storage cost at servers and the changes of service points in cloud should not have much effect on the TRE efficiency.

4 CoRE DESIGN

In this section, we describe the design of CoRE in detail and explain how it achieves our design goals.

4.1 Overview

CoRE has two TRE modules each for capturing short-term redundancy and long-term redundancy respectively. Two TRE modules are integrated into a two-layer redundancy detection system. For any outbound traffic from the server, CoRE first detects the long-term redundancy by the first-layer TRE module. If no redundancy is found, it turns to the second-layer TRE module to search for short-term redundancy at finer granularity.

The first-layer TRE module detects long-term redundancy by a prediction-based *Chunk-Match* approach like PACK [7]. Considering the inefficiency of PACK described in Section 2, we propose an improved prediction algorithm to efficiently handle data changes so that a small data change will not affect the TRE efficiency. Specifically, the sender in CoRE performs the same chunking algorithm as the receiver to divide the outgoing data into chunks. Then, for each chunk, the sender computes its signature (e.g., its SHA-1 hash value) and looks up the signature in the *prediction store* that keeps all predictions recently received from the receiver. If a matching signature is found, the sender sends a prediction confirmation PRED-ACK message to the receiver instead of the outgoing chunk, no matter whether the chunk has the expected offset in TCP stream which is specified in the prediction. This is in contrast to PACK. In this way, our prediction matching does not involve data position in TCP stream, which makes CoRE resilient against data changes.

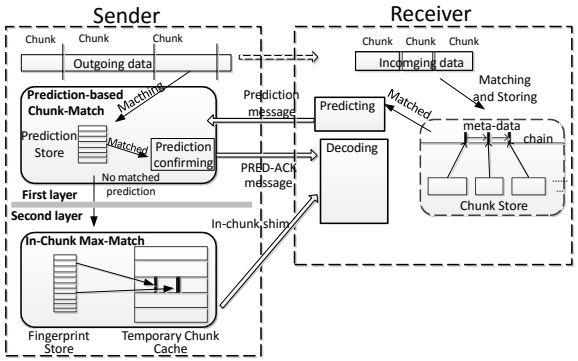


Fig. 4. Overview of CoRE

In the second-layer TRE module, both the server and client maintain a temporary small local chunk cache to store most recently transmitted and received chunks during their communication respectively. The server matches outgoing data with the local cache at fine granularity to detect short-term redundancy. In particular, the sender stores its recently transferred chunks in its local *chunk cache*. For every chunk in the cache, the sender computes a set of representative fingerprints, each of which is the hash value of a data window of size w in the chunk. Every representative fingerprint (along with a pointer to the corresponding chunk in the cache) is stored into a *fingerprint store*. To detect the redundancy inside an outgoing chunk, the sender performs *In-Chunk Max-Match* to identify maximal sub-strings in the chunk which have duplicates in the chunk cache. Specifically, the sender compares each representative fingerprint of the outgoing chunk against the fingerprint store to check whether a matching fingerprint exists. A matching fingerprint indicates that the outgoing chunk has a data window of size w which also appears in a previously transmitted chunk in the cache. If a matching fingerprint is found in the fingerprint store, the in-cache chunk which it points to is retrieved. The data window corresponding to the matching fingerprint in the in-cache chunk is expanded byte-by-byte in both directions and compared with the outgoing chunk, in order to identify the maximal overlap substring between the in-cache chunk and the outgoing chunk. Then, the sender encodes the maximal matched substring in the outgoing chunk with an *in-chunk shim* which contains the signature of the corresponding in-cache chunk, the offset and length of the matched substring.

For any incoming packet, the receiver first decodes in-chunk shims if any. To decode an in-chunk shim, the receiver retrieves the chunk in its local cache which has the same signature as that in the shim, and replace the shim by the substring of the in-cache chunk according to the offset and length specified by the shim. If the packet is a PRED-ACK message, the receiver checks the confirmed prediction in its prediction store and retrieves the corresponding predicted chunk in its chunk store. Then, it copies the chunk to its TCP input buffer according to its offset in TCP stream specified by the PRED-ACK message. An overview of CoRE is shown in Figure 4. Next, we explain the details of CoRE.

4.2 Chunking and Fingerprinting

As described above, CoRE coordinately uses prediction based Chunk-Match and In-Chunk Max-Match to implement two-layer TRE in order to maximally detect redundancy by capturing both long-term and short-term redundancy. Chunk-Match identifies the repeated chunks in a TCP stream while Max-Match identifies the maximal repeated substrings in a chunk. Similar to PACK [7],

CoRE uses a large chunk size, i.e., at the order of several KB, for Chunk-Match. A larger chunk size can reduce the total number of chunks in a TCP stream, which further results in fewer expensive SHA-1 operations for computing signatures of chunks, less storage cost for storing the meta-data of the chunk store at the receiver and also fewer prediction transmissions from the receiver. With a large chunk size, prediction based Chunk-Match identifies long-term redundancy with low operating cost. With computing the representative fingerprints at average interval of 32-64 bytes for any chunk, which is referred to as *fingerprinting*, In-Chunk Max-Match is able to identify redundancy inside chunks at a small granularity. In this way, when the transmission of an outgoing chunk cannot be eliminated by prediction based Chunk-Match, In-Chunk Max-Match can capture the redundancy inside the chunk to improve bandwidth savings. As we can see, chucking and fingerprinting are basic operations in CoRE and their efficiency is desired for CoRE to achieve low computation cost.

Chunking algorithms in previous works [7], [9], [11] determine chunk boundaries based on either byte or fingerprint. Byte based algorithms, such as MAXP [11] and Samplebyte [9], choose a byte that satisfies a given condition as chunk boundary. The fingerprint-based algorithms, such as Rabin fingerprint based [9] and PACK chunking [7], compute fingerprints by applying a pseudo-random hash function to sliding windows of w contiguous bytes in a data stream and select a subset of fingerprints with a given sampling frequency. The first byte in the window of each chosen fingerprint forms the boundaries of the chunks. As opposed to previous works [7], [8], [9], [11], [24] which use Chunk-Match and Max-Match exclusively, CoRE needs the sender to perform both chunking and fingerprinting to every chunk. A straight approach for this is to first use a chunking algorithm to divide a data stream into chunks and then computes the fingerprints for each chunk. However, this approach needs scanning the byte string twice for chunking and fingerprinting respectively, which increases the computation cost of the servers. To save the cost, we can utilize the chunking algorithm based on the fingerprint to generate chunks and fingerprints of chunks within one single-pass scanning. Specifically, based on XOR-based hash in the chunking algorithm of PACK [7], our algorithm computes fingerprints over the byte stream and chooses a subset of them as chunk boundaries with the remaining fingerprints as the fingerprints of chunks, as shown in Figure 5.

In our algorithm, a XOR-based rolling hash function is used to compute a 64-bit pseudo-random hash value over each sliding window of w bytes of the byte stream. As shown in [7], the XOR-based rolling hash function achieves higher speed than Rabin fingerprinting algorithm for computing fingerprints. Therefore, we choose the XOR-based rolling hash function to generate fingerprints and chunk boundaries. Given k bit-positions in a 64 bit string, denoted by $P_f = \{b_1, b_2, ..., b_k\}$, if the 64-bit pseudo-random hash has '1' value at all these positions, it is chosen as a fingerprint. For each fingerprint, given a set of n bit-positions P_c in a 64-bit string such that $|P_c| = n$ and $P_f \subset P_c$, if the fingerprint has '1' value at all positions in P_c, it is chosen as a chunk boundary. As a result, the average sampling interval for fingerprints is 2^k bytes and the average chunk size is 2^n bytes. Algorithm 1 shows the pseudo-code of our chunking and fingerprinting algorithm with $w = 48, n = 13, k = 6$, which gives the average fingerprint sampling interval of 64B and the average chunk size of 8KB. Note that in CoRE only the sender needs to run both chunking and fingerprinting over the outgoing data stream while the receiver just runs chunking over the received data.

Fig. 5. Chunking and fingerprinting

Algorithm 1 Chunking and Fingerprinting

```
1: cmask ← 0x00008A3110583080; //13 1-bits, 8KB chunks
2: fmask ← 0x0000000000383080; //6 1-bits, 64B fingerprint
   sampling interval
3: longval ← 0; //64-bit
4: for all byte∈ stream do
5:    shift left longval by 1 bit;
6:    longval ← longval XOR byte;
7:    if processed at least 48 bytes and
      (longval AND fmask) == fmask then
8:       found a fingerprint f;
9:       if (longval AND cmask) == cmask then
10:         f is a chunk boundary;
11:      end if
12:   end if
13: end for
```

4.3 CoRE Sender Algorithm

The sender uses a prediction store to cache the most recent predictions received from the receiver. Each prediction contains the SHA-1 signature of a predicted chunk and its expected offset, i.e., TCP sequence number in the TCP byte stream. The sender also has a chunk cache to store chunks that have been recently sent. A fingerprint store holds the meta-data of every representative fingerprint for each cached chunk, which includes the fingerprint value, the address of the chunk in the cache referred by the fingerprint, and the byte offset of the window in the chunk over which the fingerprint is computed.

When receiving new data from the upper layer application, the sender performs the chunking and fingerprinting algorithm. For each outgoing chunk, if the prediction store is not empty, the sender first computes the SHA-1 signature of the chunk and then looks up it in the prediction store. If a matching signature is found in a prediction p, the sender replaces the outgoing chunk with a PRED-ACK confirmation message which carries a tuple $< \text{offset}_p, \text{offset}_s >$ where offset_p is the expected offset in prediction p and offset_s is the actual offset of the outgoing chunk in the TCP stream. As we explain later in Section 4.4, each prediction is uniquely identified at the receiver by its expected offset.

In PACK, the sender simply discards previously received predictions which predict chunks before the position of current outgoing data in TCP stream. The chunk signature of a prediction is compared with the outgoing data chunk which is in the TCP sequence interval uniquely determined by the offset and length in the prediction. However, as we pointed earlier in Section 2, PACK suffers degraded TRE efficiency if the data offset changes due to disperse insertions and deletions of data. With consideration of possible data changes, the predicted chunks may actually appear in the near future after the expected offset. Thus, we improve the prediction-based algorithm by allowing the sender to use the historical predictions regardless of their expected chunk offset in TCP stream. To this end, the CoRE sender needs to divide the outgoing data into chunks in the same way as the receiver

does with a content-based chunking algorithm, instead of dividing according to the chunk offset and length specified by the receiver in the prediction. Each outgoing chunk is then compared with all entries in the prediction store regardless of their expected chunk positions (i.e., TCP sequences), such that the chunk can match a prediction having an inconsistent TCP sequence but the same signature. As a result, CoRE can achieve resiliency against data changes and be able to leverage useful predictions from the receiver as much as possible. As for the example in Figure 1, because the content around chunk boundaries does not change, CoRE divides the outgoing data at the sender into chunks [*ABCD*] [*BCEFA*] [*HIJK*], given the same chunking algorithm as at the receiver that determines chunk boundaries based on data content. The signature of chunk [*HIJK*] is already received as a prediction from the receiver. Then, CoRE would find a match and compress the chunk. By contrast, PACK misses this opportunity as mentioned before in Section 2.

If the prediction store is empty or the prediction-based Chunk-Match does not find a matching prediction, the sender then performs In-Chunk Max-Match. It checks the fingerprints of the outgoing chunk against the fingerprint store. If a matching fingerprint is found, the corresponding chunk is retrieved from the chunk cache. Considering the possible collision in the fingerprint namespace due to the same hash values for different data, the window corresponding to the matching fingerprint in the chunk is compared with the outgoing chunk byte-by-byte. Then, if the window has the same data, it is expanded byte-by-byte in both directions to obtain the maximal overlapped substring. Each matching substring in the outgoing chunk is encoded with a shim $<sign_c, length, offset_c, offset_s>$ where $sign_c$ is the signature of the in-cache chunk, *length* is the length of the matching substring, $offset_c$ is the offset of the matching substring in the in-cache chunk, and $offset_s$ is the offset of the matching substring in the outgoing TCP stream. If an in-cache chunk has multiple matching substrings with an outgoing chunk, all corresponding shims can be compressed together in the form of $<sign_c, length_1, offset_{c1}, offset_{s1}, length_2, offset_{c2}, offset_{s2}, \ldots>$ where each matching substring i is described by $length_i$, $offset_{ci}$ and $offset_{si}$.

After the sender sends out the chunk, the sender updates the chunk cache and the fingerprint store with this chunk. Figure 6 describes the sender algorithm for processing outgoing data by state machines. The details for the maintenance of local data structures at the sender are described as follows.

Chunk Cache. The chunk cache is a fixed-size circular FIFO buffer. The server maintains a chunk cache for every client. If no further request is received from the client within a period, the cache is released. Once receiving the request from a new client, the server allocates a chunk cache for the client. The chunk cache stores recently transferred chunks from the server to the client, via either one TCP connection or multiple TCP connections. Once the chunk cache is full, the earliest chunk is evicted and the fingerprints pointing to it are invalidated. Each entry in the chunk cache is a tuple < *data*, *signature* >, where *data* field is the chunk data and *signature* field is its SHA-1 hash. Note that the signature is not always computed for all chunks. The *signature* field is filled for a chunk in two cases. First, the outgoing chunk's signature has to be computed for prediction matching when the prediction store is not empty, so in this case the chunk is stored into the cache with its signature. Second, during In-Chunk Max-Match, if the in-cache chunk where the maximal matching substring is obtained has empty the *signature* field, its signature is computed and filled. In this way, the expensive signature computation is only performed on demand, which avoids unnecessary SHA-1 computations to reduce the server's computation cost.

Since the chunk cache is used to detect short-term traffic redundancy, a small cache size can be sufficient. As indicated in Section 3, even with small cache size like 250KB, significant short-term redundancy is detected by Max-Match of the sender-based TRE. In CoRE, the server and the client also need to maintain synchronized chunk caches for In-Chunk Max-Match. The changes of service point in the cloud may cause the loss of cache synchronization at the server. However, since the small chunk cache only stores short-term historical traffic, the changes of service point just result in the missing of traffic redundancy in a short term, and its impact on the TRE efficiency is limited. Besides using small chunk cache, CoRE releases the cache if no further service request is received from the client within a period, which further reduces the storage cost of CoRE at the server.

Prediction Store. As opposed to that a chunk cache is shared by all TCP connections to the same client which the cache is allocated for, a prediction store is allocated by the server for each TCP connection with a client. The expected chunk offset of a prediction is represented by the TCP sequence number of the predicted chunk (i.e., the TCP sequence number of the first byte in the chunk) in the TCP stream. Once receiving a new prediction from the receiver, the sender inserts it into its prediction store associated with the corresponding TCP connection. The prediction store holds the most recent predictions. Outdated predictions need to be identified and evicted to limit the size of the prediction store and ensure the efficiency of prediction matching. In CoRE we define the elapsed time of a prediction p in the store as:

$$E_p = \begin{cases} 0, & Seq_c \leq_{seq} Seq_p \\ (Seq_c - Seq_p) \bmod 2^{32}, & Seq_c >_{seq} Seq_p \end{cases} \quad (1)$$

where Seq_c is the TCP sequence number of the chunk currently to be sent, and Seq_p is the expected TCP sequence number in the prediction p. $\leq_{seq}$ and $>_{seq}$ are comparisons of TCP sequence number with modulo 2^{32} [25]. We use TTL_{pred} to denote the maximum elapsed time of a prediction, a system parameter. At the time when CoRE compares the outgoing chunk having the TCP sequence number Seq_c against the prediction store, the predictions satisfying $E_p > TTL_{pred}$ are removed from the prediction store. In this way, we use the difference of TCP sequence number between a prediction and current outgoing data to measure the timeliness of the prediction, given that a larger difference implies the less relevance of the prediction with the outgoing data. By keeping all the predictions within TTL_{pred} rather than only the latest prediction as in PACK, CoRE has higher chance to find the matches since the chunks predicted by these predictions may appear in the near future due to their delayed appearance or repetitive appearance in the TCP stream.

4.4 CoRE Receiver Algorithm

For each TCP connection, the receiver divides the incoming data stream into chunks and maintains a local prediction store which caches recent predictions for the TCP connection. To reconstruct the raw data stream, the receiver processes incoming TCP segments according to their different types. There are three types of TCP segments from the sender to the receiver: PRED-ACK message, the message encoded with shims and raw data. Upon receiving a PRED-ACK message containing $<offset_p, offset_s>$, the receiver first checks the corresponding prediction store to find the prediction p which has the expected offset $offset_p$. Then, the receiver retrieves the chunk predicted by prediction

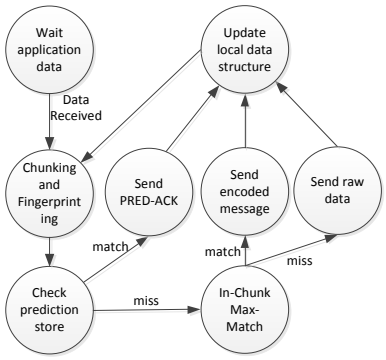


Fig. 6. Sender algorithm

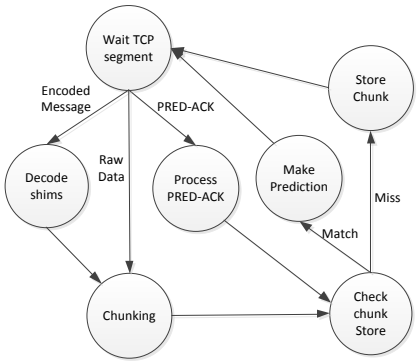


Fig. 7. Receiver algorithm

p from the chunk store and place it to the input buffer of the corresponding TCP connection with the offset $offset_s$ in the TCP stream specified by the sender. If receiving the message containing shim <$sign_c$, $offset_c$, $length$, $offset_s$>, the receiver finds the chunk having signature $sign_c$ from its chunk store. The matching substring in the chunk indicated by $offset_c$ and $length$ is copied to the receiver's TCP input buffer at the position $offset_s$ in TCP stream.

The chunking algorithm at the receiver is the same as Algorithm 1 without the fingerprint identification with $fmask$. The receiver maintains a local chunk store as in PACK [7], which stores chunks from all its TCP connections. The chunks from the same TCP connection are linked together in the order they are received and then stored into the chunk store. If the signature of an incoming chunk is found in the chunk store, the receiver makes one or multiple chunk predictions for a number of subsequent chunks from the sender. Each chunk prediction consists of the signature of the predicted chunk and its expected offset in the TCP stream. For any chunk prediction, if the predicted chunk has overlapped TCP sequence range with any previous predictions in the corresponding prediction store, the prediction is discarded. Accordingly, within a TCP connection, the predictions from the receiver to the sender have no overlap with each other on the TCP sequence range for their predicted chunks. The chunk predictions are sent within a PRED message and in the order of their expected offsets. Figure 7 shows the receiver algorithm, in which the receiver enters different processing procedures according to the types of incoming messages.

Since in a TCP connection the chunk predictions have no overlapped TCP sequence range for their predicted chunks, each prediction for the same TCP stream can be uniquely identified by the expected offset in it. Then, each entry in the prediction store consists of an expected TCP sequence number and a pointer to the corresponding predicted chunk. The receiver also periodically removes the outdated predictions from the prediction store in the similar way as the sender does. Each time when the receiver receives a chunk having the TCP sequence number Seq_c, any prediction p with its elapsed time E_p (computed by (1)) greater than TTL_{pred} is removed from the prediction store before the receiver tries to make any new predictions.

5 ADAPTIVE SOLUTIONS TO TRAFFIC REDUNDANCY

We have described the basic operations in CoRE in previous sections. In this section, we propose adaptive enhancements for CoRE to improve its TRE efficiency according to the characteristics of traffic redundancy.

5.1 Adaptive Prediction For Long-term Redundancy

The prediction based Chunk-Match TRE allows the receiver to locally obtain the sender's data instead of through the network when the chunk prediction is successful. When the traffic has significant long-term redundancy and most predictions would be successful, a lot amount of data would be obtained locally by the receiver. In this case, the rate of data reception at the receiver can be mainly determined by the transmission rate of chunk predictions from the receiver. The data reception rate can be increased by making more chunk predictions each time when the receiver finds a matching chunk in the chunk store. On the other hand, however, it may cause significant and unnecessary prediction transmission cost for the receiver, especially when the prediction fails often. Thus, an adaptive prediction method is desired to dynamically determine the number of chunks to predict each time with the goal to increase hit ratio of predictions.

PACK [7] proposes a *virtual window* scheme to control the aggregated number of bytes of all chunks predicted in the pending predictions. The virtual window size is doubled with each successful prediction. Upon a mismatch, the window is reset to TCP receiver window. The reason for such design is that in PACK the chunk chain for prediction is constructed from the TCP stream. The virtual window starts from the matching chunk, and moves forward along the chain upon each prediction match with the window size being doubled to include more subsequent chunks for prediction, which increase the prediction transmission rate. However, since CoRE does matching regardless of the prediction positions in the chain, the position of the matching chunk and thus the virtual window may move forward and backward on the chain. Therefore, doubling the window size upon each prediction success in CoRE would not obtain the same efficiency as in PACK. Accordingly, a new virtual window scheme is required for CoRE. Furthermore, PACK's virtual window scheme is sensitive to data changes. Scattered data changes, compared against the cached chain, would frequently cause the window reset and force the sender to revert to raw data transmission. Here we propose an adaptive prediction algorithm, which adjusts the virtual window according to the hit ratio of previous chunk predictions, to ensure the prediction efficiency from the data changes and handle the out-of-order prediction matching in CoRE.

For an incoming TCP stream, we assume that the receiver finds a matching chunk M in a chunk chain. Then, the receiver uses

a sequence of chunks following M in the chain for prediction, denoted by $P_M = \{p_1^M, p_2^M, \ldots, p_k^M\}$ where each p_i^M denotes a predicted chunk. The size of chunk p_i^M is denoted by $S(p_i^M)$. Suppose T_{P_M} be the set of chunks in P_M that are successfully predicted. Then, the hit ratio of prediction P_M is computed by

$$H(P_M) = \frac{\sum_{p_i^M \in T_{P_M}} S(p_i^M)}{\sum_{i=1}^{k} S(p_i^M)} \quad (2)$$

Let N_M be the next TCP sequence number after the chunk that is matched with M. For any TCP connections, the prediction algorithm records N_M, P_M and $\sum_{i=1}^{k-1} S(p_i^M)$ where M is the most recent matching chunk for the incoming TCP stream. These parameters are updated when a new matching chunk appears and new predictions are made based on it. When a new chunk is received, the prediction algorithm tries to match it for prediction only if this chunk has a sequence number larger than $N_M + \sum_{i=1}^{k-1} S(p_i^M)$. That is, the chunk used for prediction should not be in the previous prediction range. The last chunk p_k^M is excluded from the previous prediction range, since we allow the algorithm to make new predictions when the incoming chunk is matched with p_k^M.

When making predictions with a new matching chunk, say M', the algorithm first computes the hit ratio of chunk predictions in P_M, and then accordingly adjusts the virtual window W. Suppose W_o be the initial prediction window size. Then, we set $W = W_o$ if (1) $N_{M'} - (N_M + \sum_{i=1}^{k-1} S(p_i^M)) \geq d_T$ or (2) $H(P_M) < H_T$, where d_T and H_T are two thresholds to determine whether to use the historical hit ratio for the next prediction. The first condition $N_{M'} - (N_M + \sum_{i=1}^{k-1} S(p_i^M)) \geq d_T$ is given with the consideration that, if the data to be predicted with M' is far away from the data predicted by the last prediction P_M, the hit ratio of P_M may be outdated for M' and thus we set W to the initial size; otherwise, it could be a good estimate for the hit ratio of the prediction $P_{M'}$ since the data predicted with M' closely follows the data predicted by P_M and we increase W. For the condition (2), if the hit ratio $H(P_M)$ is lower than the minimum requirement H_T for increasing the window size, W also is set to the initial size. For example, we can let $H_T = 0.5$ which indicates that the virtual window is increased only if the historical hit ratio is at least 50%. If $N_{M'} - (N_M + \sum_{i=1}^{k-1} S(p_i^M)) < d_T$ and $H(P_M) > H_T$, we let $W = W_o + H(P_M)W_{max}$ such that W increases with the hit ratio. Here W_{max} is a constant which decides the maximum virtual window size $W_o + W_{max}$.

5.2 Adaptive TRE Operations For Different Redundancy Distribution

The distribution of traffic redundancy over short-term and long-term time scales varies with different kinds of cloud applications. According to our previous work [26], some applications may have dominant long-term redundancy in traffic, while some have dominant short-term redundancy. In this case, the TRE operations on both layers to simultaneously detect both short-term and long-term redundancy cause unnecessary cost. CoRE's first-layer TRE incurs the prediction transmission cost, which may decrease the bandwidth saving attained from redundancy removal if the predictions fail a lot due to little long-term redundancy. CoRE's second-layer TRE incurs a lot of computation cost for In-Chunk Max-match. Thus, an adaptive solution to this issue is to discover the layer which detects the dominant redundancy and disable another layer TRE operation to save the TRE operation costs.

To identify the dominant redundancy, CoRE records the aggregate redundant bytes detected by each layer after running for a period with two-layer TRE for a cloud application. The aggregate redundant bytes detected by the first-layer and second-layer of CoRE, are denoted by R_L and R_S respectively. Given a threshold α (for example, $\alpha = 0.95$), if $\frac{R_L}{R_L+R_S} \geq \alpha$, we say R_L is dominant and the second-layer TRE can be disabled. If $\frac{R_L}{R_L+R_S} \leq 1-\alpha$, R_S is dominant and the first-layer TRE can be disabled. If $1-\alpha < \frac{R_L}{R_L+R_S} < \alpha$, we say R_L and R_S are comparable, and both layers can be enabled.

On the other hand, since TRE incurs additional load on servers, the bandwidth savings may be eradicated by the increased operation cost. To avoid that, we can measure the percentage of redundant traffic for the targeted cloud application during the runtime profiling. If its percentage is lower than some threshold given by the cloud users, TRE service can be shut down for that application.

6 IMPLEMENTATION

We implemented CoRE in JAVA based on PACK implementation [27]. The protocol is embedded in the TCP options field. The prototype runs on Linux with Netfilter Queue [28].

The implementation of the sender component largely follows the discussion in Section 4.3. The implementation of the prediction store at the sender must support quick identification of matching predictions, and efficient identification and deletion of outdated predictions. Considering that the arrivals of predictions are in the increasing order of the expected offset, we use LinkedHashMap because its iteration ordering is normally the order in which keys are inserted. The chunk cache is implemented as a circular FIFO with a maximum of M fixed-size entries. The fingerprint store is implemented as HashMap. When the old chunk is evicted from the chunk cache, the associated fingerprints should be invalidated. To efficiently perform this task, we adopted the method in [16]. We maintain a counter $MaxChunkID$ (4 bytes) that increases by one before a new chunk is stored. Each chunk has an unique identifier $ChunkID$ which is set to the value of $MaxChunkID$ when it is stored. The position of a chunk in the store is computed by $ChunkID\%M$, thus we store $ChunkID$ instead of the position of the chunk in meta-data of the fingerprint store. For each entry in the fingerprint store, if $ChunkID < MaxChunkID - T$, the corresponding chunk has been evicted and thus the fingerprint is invalid. For the implementation of receiver component, we add the in-chunk shim decoding algorithm, and the prediction store is also implemented as LinkedHashMap.

Parameter Settings. In default, we use an average chunk size of 8KB and fingerprint sampling interval of 64B by setting $n = 13$ and $k = 6$ for the chunking and fingerprinting algorithm. We set $TTL_{pred} = 2^{21}$ for the prediction store. The chunk cache in the second-layer TRE can hold up to 64 chunks, about 512KB at default. For the adaptive prediction algorithm, we let $d_T = 32KB$, $H_T = 0.6$, $W_o = 32KB$ and $W_{max} = 56KB$.

7 EVALUATION

In this section, we evaluate CoRE and compare it with PACK and the sender-based scheme by both trace-based and testbed approach. Our trace-based evaluation is based on two genuine data set of real-life workloads and two Internet HTTP traffic traces. We deploy CoRE implementation onto our testbed consisting of a server and client(s). Our server is 2 Ghz Dual-Core with 2 GB RAM running Ubuntu, and clients have 2.67GHz Intel Core i5 with 4 GB RAM.

7.1 Traffic Traces

Our evaluation uses the following trace data sets:

Linux Source Workload (2.0.x): Forty tar files of Linux kernel source code. These files sum up to 1 GB and were released over a period of two year. The traffic is generated by downloading all these tar files in their release order from 2.0.1 to 2.0.40.

IMAP trace: 710MB of IMAP Gmail for past 1 year containing 7500 email messages. These emails consist of personal emails as well as regular mail subscriptions.

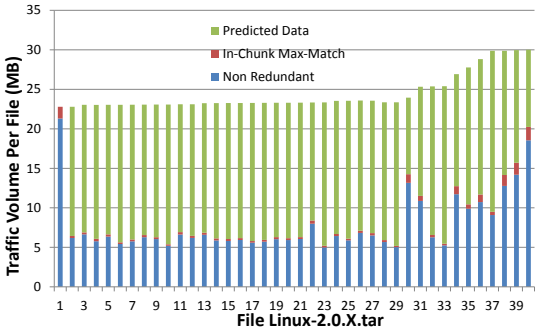
Internet traffic traces: We monitored one of the access links from a large university. The link has a 1Gbps full-duplex connection to the backbone and serves roughly 30,000 users. We captured entire packets (including payloads) going in either direction on the link. Due to limited disk volume compared with huge traffic volume in our measurement architecture, we could not store all kinds of traffic. Thus, without loss of generality, we decided to focus on the web applications as the target traffic by using port 80 to filter them out. However, the HTTP traffic still consume about 1.6GB-2.9GB per minute, which is not suitable for long-term monitoring. In order to study the redundancy in traffic in a long-term period, we adapted our capturing rules for one of the most popular social network (SN) website. As a result, our real Internet traffic traces include one full HTTP trace (1.9GB in 120 seconds) and one SN trace (1.3GB in 2 hours). The detailed information is shown in Table 1.

7.2 TRE efficiency of CoRE

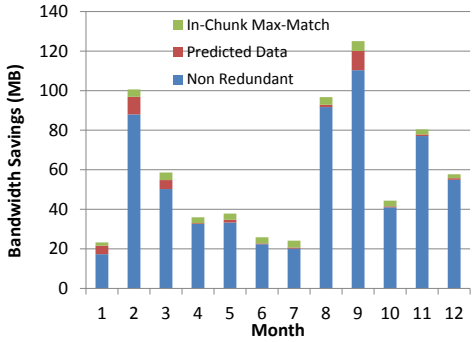
In this section, we first evaluate TRE efficiency of CoRE with the traffics of Linux source files and Gmail messages as in PACK [7], and then compare bandwidth savings of CoRE compared with PACK and sender-based solution. The traffic of Linux source files are resulted from the downloads of these files in the their release order. For the traffic of Gmail, all the email messages are grouped by month, and downloaded by their issue dates. The CoRE sender uses a 4MB chunk cache, a size of the maximum capacity of 512 chunks with an 8KB average chunk size.

Figure 8(a) and 8(b) show the redundancy detected by the prediction-based Chunk-Match and In-Chunk Max-Match of CoRE, respectively. In Figure 8(a), 29 files have more than 70% redundancy and the remaining files have at least 38% redundancy. Total redundancy amounts to 68% in the whole traffic volume of 40 Linux source files, about 664MB. Such a large amount of redundancy results from the high similarity between an earlier version and a subsequent version. When the file "Linux-2.0.1.tar" is downloaded for the first time, there is no similar data that is cached, so that there are no successful predictions and CoRE only found a small redundancy inside the file itself by In-Chunk Max-Match. As we can see, each file contains a little redundancy inside itself and a large amount of redundancy exists across files. This suggests that in such real-world workloads long-term redundancy is significantly more than short-term redundancy. Because a download happens only after the release of a new version and the cache required to detect long-term redundancy has at least the size of a file, the sender-based TRE with small temporary cache cannot obtain bandwidth savings.

In Figure 8(b), the email traffic in each month has much less redundancy than the Linux source files. The reason is that most of emails are distinct and have less duplicate content. The total redundancy detected by CoRE amounts to 11% in our Gmail traffic volume of 12 months, about 71MB, which is less than 31.6% redundancy shown in the Gmail traffic used by PACK [7].



(a) Detected redundancy of 40 different Linux Kernel versions



(b) Detected redundancy of 1-year Gmail account by month

Fig. 8. Detected Redundancy by CoRE

Data traffic Name	CoRE	PACK
	% savings	
Linux source	68	54
Email	11	3.6
Univ-HTTP	4.8	0
Univ-SN	8.3	0.3

TABLE 2
Percentage bandwidth savings of CoRE and other solutions

The authors of PACK found that the redundancy arises from large attachments from multiple sources in their Gmail messages. However, our Gmail messages contain few attachments, which causes less redundancy than PACK's Gmail trace. Little redundancy is detected by prediction in the Gmail traffic of several months from April to August and from October to December, which indicates little long-term redundancy in the Gmail trace and confirms the distinction of our Gmail messages. We also find that in our detected redundancy, 5% is contributed by prediction and the remaining 6% is contributed by In-Chunk Max-Match. As we can see, the redundancy detected by In-Chunk Max-Match is significant compared to the redundancy detected by prediction, which is sharp contrast to the results of Linux source files. This indicates that Gmail traffic has a significant short-term redundancy, and the receiver-based TRE only obtains less than half of bandwidth savings in our Gmail traffics compared to CoRE. The results verify the necessity of cooperative operations between sender-based TRE and receiver-based TRE in two layers.

Table 2 compares the bandwidth savings of CoRE, PACK and sender-based TRE with different data traffics. Each entry shows the percentage of bandwidth savings in the total volume of the corresponding traffic trace. From this table, we see that CoRE performs better than PACK because CoRE captures short-term redundancy and is also resilient to data changes. In particular, for two HTTP traffic traces Univ-HTTP and Univ-SN, PACK detects

Trace Name	Description	Dates/ Start Times	Duration	Total Volume(MB)	IP Pairs	Servers(IP)	Clients(IP)
Univ-HTTP	Inbound/outbound http	10am on 11/05/11	120s	1900	8277	3183	1931
Univ-SN	Inbound/outbound for SN	3pm on 11/08/11	2h	1300	3237	6	894

TABLE 1
Characteristics of Internet traffic traces

little redundancy, but CoRE detects 4.8% and 8.3% redundant traffic respectively. This indicates that these two traces exhibit sone short-term traffic redundancy but have little long-term traffic redundancy.

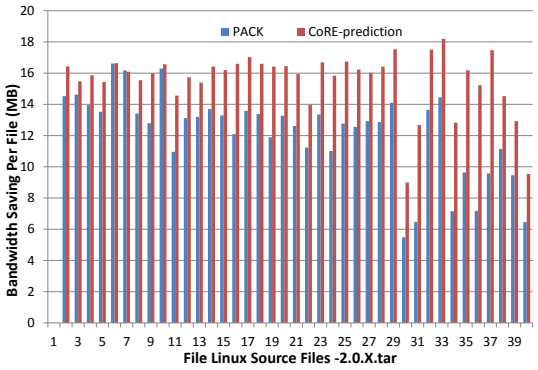
Based on the evaluation and comparison results of CORE and PACK on TRE efficiency, we can conclude the traffic traces we used have different traffic redundancy characteristics. They exhibit different distributions of traffic redundancy over short-term and long-term time scales. Linux source workload has dominant long-term traffic redundancy, Gmail IMAP trace has comparable long-term and short-term traffic redundancy, and Internet traffic traces have little long-term traffic redundancy but relatively high short-term traffic redundancy. Such results indicate the necessity of an adaptive TRE solution for different cloud applications.

7.3 Performance Of CoRE Prediction

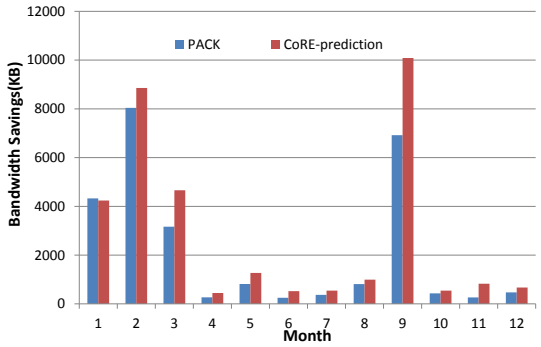
The difference of the prediction designs between CoRE and PACK is that the CoRE sender performs the same chunking operation as the receiver so that it is able to compare chunks with predictions regardless of predicted data offsets in the data stream. As a result, CoRE can ensure TRE efficiency from the changes of outgoing data compared with the original data cached at the receiver. To verify this, we disabled the low layer of In-Chunk Max-Match in CoRE and only measured the redundancy detected by its prediction. We compared the CoRE's prediction solution with PACK's by using Linux source and Gmail traffics, with an average chunk size 8KB for both CoRE and PACK.

Figure 9(a) shows the bandwidth savings, i.e., redundancy in each of the downloaded versions detected by CoRE prediction and PACK prediction, respectively. Figure 9(b) shows the redundancy in each month of the email messages. The experimental results show that CoRE prediction scheme can detect and eliminate more redundancy than PACK. The amount of redundancy detected by CoRE prediction is 21% higher than that detected by PACK in the traffic of Linux source files, and 22% higher in Gmail traffic. The reason for the higher performance of CoRE prediction is that it is common that the update of the Linux source code involves the new code insertion and old code deletion. In the Gmail traffic, the emails usually contain short-term data repetition because when people reply to an email, the content of this email is usually included to the outgoing message.

Resiliency against data changes To further verify the resiliency of CoRE prediction against data changes, we chose a 31.5MB Linux tar file and inserted one random byte to the random positions into the file. By changing the average distance between two successive insert on positions in the file, we changed the degree of data changes on the original file. We downloaded the original file first, and then downloaded the modified file to measure its redundancy that CoRE and PACK can detect given the cache of the original file at the receiver. Figure 10 shows that CoRE prediction detects more redundancy than PACK under various degrees of data changes. "No changes" in the x-axis means that the second file we downloaded is the same as the original file. In this



(a) Bandwidth savings of 40 different Linux Kernel versions



(b) Bandwidth savings of 1-year Gmail account by month

Fig. 9. Comparison of CoRE prediction and PACK

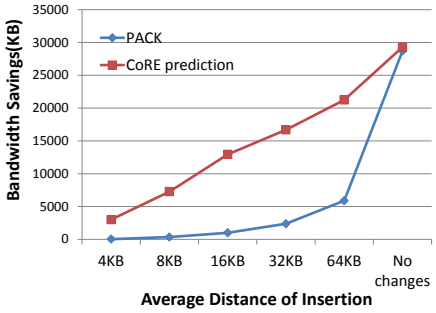


Fig. 10. Detected redundancy V.S. Insertion intensity

case, CoRE and PACK detect the same amount of redundancy. As we can see, PACK is very susceptible to data changes. It decreases exponentially as the insertion intensity increases. In contrast, CoRE is much more resilient against data changes. It decreases linearly as the insertion intensity increases. The results confirm the resiliency of CoRE prediction to data changes due to the data chunking at the sender and the matching with the historical predictions.

7.4 Cost Evaluation

To measure the operation cost of CoRE at the sender and the receiver, we monitored the average CPU utilization on the server and the client at an interval of one second. Since the TRE operation cost depends on the characteristics of traffic redundancy, we use the Linux source workload, Gmail IMAP traffic and Univ-HTTP traffic trace which have different redundancy distributions over long-term and short-term according to Table 2.

With the Linux source workload which has significant long-term redundancy, Table 3 compares CoRE's server CPU utilization ratio with PACK and sender-based approach. As we can see, CoRE has a little higher CPU utilization ratio than PACK, and both of them have less CPU utilization ratio than the sender-based solution. Because the long-term traffic redundancy is dominant in the Linux source workload, most of traffic redundancy is detected and eliminated by the prediction-based Chunk-Match and thus CoRE and PACK have closer CPU utilizations. Such results also indicate that chunking and fingerprinting in CoRE incur a low additional cost compared to PACK. As a result, CoRE is able to efficiently achieve overall gain savings when used in cloud environment.

Table 4 shows the CPU utilization with the Univ-HTTP traffic trace which only has short-term redundancy. We can see that PACK has much small CUP utilization at the server compared with Table 3. This is because that no predictions are received in this case and thus no TRE is conducted at the server in PACK. The CPU cost of CoRE is much closer to the sender-based TRE than the previous result in Table 3. This is because that most of the Univ-HTTP traffic goes through the second-layer TRE in CoRE due to little long-term redundancy in the Univ-HTTP traffic. In contrast, long-term redundancy is dominant in the Linux source workload and most redundant traffic is eliminated in the first-layer TRE. Since prediction-based Chunk-Match has less computation complexity than In-Chunk Max-Match, the difference of CPU cost between CoRE and sender-based TRE will be larger if the traffic has higher redundancy in long-term than in short-term. Table 5 shows the CPU utilization with the Gmail IMAP traffic, which has comparable long-term and short-term redundancy. Comparing Table 5 with Table 4, we can see that, PACK has higher server CPU utilization with the Gmail IMAP traffic than with Univ-HTTP traffic trace, because PACK detects and removes long-term redundancy in the Gmail IMAP traffic. The server CPU utilization of CoRE with the Gmail IMAP traffic is lower than that with Unive-HTTP trace, because the detection of long-term redundancy at the first-layer avoids the more costly operations at the second-layer TRE.

In summary, the above results indicate that different distributions of traffic redundancy at short-term and long-term time scales can cause different CPU costs for CoRE. When the short-term redundancy is dominant, CoRE's operation cost at the cloud server is closer to sender-based TRE; when the long-term redundancy is dominant, its operation cost is closer to receiver-based TRE like PACK; if the traffic has comparable short-term and long-term redundancy, the cost is between sender-based TRE and receiver-based TRE. Previous study [7] demonstrates that the sender-based TRE has higher operation cost in the cloud than PACK, with considering the elastic capacity adjustment by adding or removing servers according to the unused CPU computation power in the server pool of the cloud application. Our evaluation results suggest that the operation cost of CoRE in the cloud is between PACK and sender-based TRE. Considering the two-layer design of TRE in CoRE, if the traffic is only dominated by either long-term or short-term redundancy, the layer of In-Chunk Match or prediction-based Chunk Match, respectively, does not contribute much on the bandwidth savings but increase the operation cost. Therefore, in order to avoid unnecessary operation cost, our adaptive solution in Section 5.2 is necessary, which profiles the traffic redundancy distribution of cloud applications and intelligently decides which layer of TRE in CoRE to be enabled.

Scheme Name	Server CPU Utilization %	Client CPU Utilization %
CoRE	3.41	2.75
PACK	2.92	2.29
Sender-based	5.28	-

TABLE 3
CPU cost with the Linux source workload

Scheme Name	Server CPU Utilization %	Client CPU Utilization %
CoRE	4.33	2.93
PACK	1.10	2.17
Sender-based	4.72	-

TABLE 4
CPU cost with the Univ-HTTP traffic

Scheme Name	Server CPU Utilization %	Client CPU Utilization %
CoRE	3.80	2.84
PACK	1.52	2.23
Sender-based	4.75	-

TABLE 5
CPU cost with the Gmail IMAP traffic

Client numbers	1	2	3	4
Server CPU Utilization%	2.2	4.3	6.1	8.7

TABLE 6
CPU cost of CoRE for Gmail traffic to multiple clients

The server cost for multiple clients Our current implementation of CoRE at the client side assumes a single TCP flow from a server to every client. Because a server usually serve multiple clients concurrently, we are interested in how the server cost changes with the number of clients. From the design we can see that the server runs an instance of CoRE for each client, the cost of CoRE at the server should be additive and will increase linearly with the number of clients served by the server. To confirm this, we setup a server with Intel i7 3.40GHz and 4GB memory, and use four clients on other machines to download the Gmail workload from the server. We vary the number of clients concurrently downloading the workload from 1 to 4 and measure the corresponding average CPU utilization of CoRE at the server. Table 6 shows the results, where the average CPU utilization increases linearly with the number of clients. Note a small difference of CPU utilization with one client between Table 5 and Table 6, 3.8 and 2.2 respectively, due to the different CPU capability of servers (2.67GHz and 3.40GHz respectively) for two experiments generating these results.

7.5 The effect of chunk cache size

We also evaluate the effect of the chunk cache size on detecting short-term traffic redundancy. As we argued in section 3, because chunk caching and In-Chunk Max-Match incurs additional storage and computation cost at a server, it is critical to only use a small chunk cache to address short-term traffic redundancy. The effectiveness of a large chunk cache for capturing long-term redundancy is addressed by CoRE's first-layer prediction based TRE. Thus, here we measure the total short-term redundancy detected in the traffic of the Linux source workload with relatively

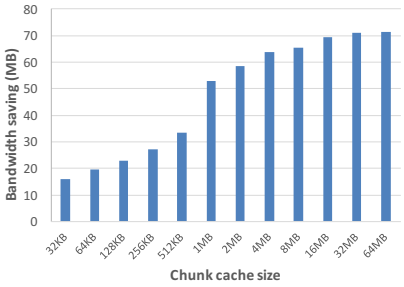


Fig. 11. The effect of chunk cache size

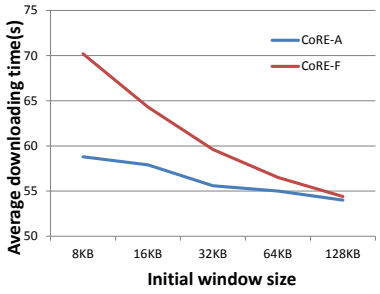


Fig. 12. Average downloading time V.S. Initial window size

small chunk cache sizes varying from 32KB to 64MB. The result is shown in Figure 11. We can see that from 512KB to 1MB the detected redundancy has significant increase but after that the increasing size yields diminishing returns. It is because that CoRE uses the chunk cache for short-term TRE, which aims at the bytes repeated within the traffic transmitted in a recent period that a small cache size is sufficient for caching. In our experiment we choose 4MB as the default chunk size, which from the figure we can see is a good trade-off between cost and saving.

At the same time, we examine the CPU utilization under different cache sizes. Since overall CoRE incurs very small CPU cost, the CPU utilization does not have large variances with these cache sizes. But still, we can tell that the CPU cost of In-Chunk Max-Match depends on the amount of detected short-term redundancy. The average CPU utilization for cache sizes from 32KB to 512KB is around 3%, close to PACK (see Table 3), because of very small short-term redundancy being detected, as shown in Figure 11. For larger cache sizes from 1MB, the average CPU utilization is around 3.5, close to CoRE with default size 4MB Table 3, because of similar amount of redundancy being detected, as shown in the above Figure.

7.6 Efficiency of Adaptive Prediction Window

To evaluate the adaptive prediction algorithm in Section 5.1, we study the data reception rate of two types of CoRE: one with the adaptive prediction algorithm to adjust virtual window (CoRE-A) and the other with fixed virtual window (CoRE-F). CoRE-F has fixed virtual window equal to the initial window size W_o. We still use the Linux traffic workload and compare CoRE-A and CoRE-F. We limit the bandwidth to 200KB/s and measure the average downloading time of each file with varying initial window size W_o. The results are shown in Figure 12. As we can see, CoRE-A obtained smaller average downloading time since it expands the window size adaptively according to the hit ratio of predictions. Since Linux traffic workload has significant long-term redundancy and the hit ratio is higher than 0.6 at most of time, the window size is at least $W_o + 0.6W_{max}$. The largest gap between two curves is at 8KB, in which case CoRE-F only has window size 8KB but CoRE-A has window size at least 40KB. As W_o increases, both CoRE-F and CoRE-A achieves smaller average downloading time. The gap between two curves becomes smaller, which indicates that the benefit of larger window size is diminishing. This is because when the virtual window is large enough, the transmission rate of predictions is limited by the transmission time on the path from the receiver to the sender, not by the number of predictions each time to be made.

According to Section 7.2, Linux source traffic has much more redundancy in the long-term than in the short-term, Email trace has comparable long-term and short-term redundancy, and two other traffic traces mainly have short-term redundancy. Thus, according to our adaptive solution for different redundancy distribution in Section 5.2, we can adjust TRE operations for each type of application. For Linux source traffic, the second-layer TRE is disabled and the CPU cost at the server is about the same as PACK. For Email trace, both layers are required. For two other HTTP traffic, the first-layer TRE is disabled to save unnecessary prediction transmissions, and the CPU cost at the server remains similar to the results obtained without disabling the first-layer TRE, since In-Chunk Max-Match has much higher computation cost than prediction-based Chunk-Match.

8 CONCLUSION AND FUTURE WORK

By the real trace driven study on existing end-to-end sender-side and receiver-side TRE solutions, we identify their limitations for capturing redundancy in short-term and long-term data redundancy. Thus, we propose a Cooperative end-to-end TRE CoRE, which integrates two-layer TRE efforts and several optimizations for TRE efficiency. We also propose several enhancement solutions for CoRE by adaptively adjusting the prediction window size and decide which layer of TRE is enabled according to the distribution of traffic redundancy of the cloud application. Through extensive trace-driven experiments, we show that CoRE can efficiently capture both short-term and long-term redundancy, and can eliminate much more redundancy than PACK while incurring a low additional operation cost. Our evaluation results show that different distributions of traffic redundancy at short-term and long-term time scales can cause different CPU costs for CoRE and adaptive solutions for CoRE are necessary for improving TRE efficiency and reducing the operation cost.

Note in this paper we consider the TRE for the applications following web service model which are dominated by the cost of egress bandwidth out of the cloud. Other types of cloud applications, like MapReduce, most bandwidth cost occurs in the datacenter network. According to the pricing of Amazon EC2 [4], the data transfer within the Amazon cloud is mostly free. Reducing their bandwidth cost inside the cloud does not save much expense for the cloud users, but may improve the networking performance and mitigate the bandwidth competition among cloud applications. We will investigate TRE for such applications in our future work.

ACKNOWLEDGEMENTS

This research was supported in part by U.S. NSF grants CNS-1249603, OCI-1064230, CNS-1254006, CNS-1049947, CNS-1156875, CNS-0917056 and CNS-1057530, CNS-1025652, CNS-0938189, Microsoft Research Faculty Fellowship 8300751, and U.S. Department of Energy's Oak Ridge National Laboratory including the Extreme Scale Systems Center located at ORNL and DoD 4000111689.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, April 2010.

[2] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *ICAC*. San Jose, CA: USENIX, 2013, pp. 23–27.

[3] "Netflix," www.netflix.com.

[4] "Amazon elastic compute cloud (EC2)," http://aws.amazon.com.

[5] "The hidden cost of the cloud: Bandwidth charges," http://gigaom.com/2009/07/17/the-hidden-cost-of-the-cloud-bandwidth-charges/.

[6] "The real costs of cloud computing," http://www.computerworld.com/article/2550226/data-center/the-real-costs-of-cloud-computing.html.

[7] E. Zohar, I. Cidon, and O. O. Mokryn, "The power of prediction: cloud bandwidth and cost reduction," in *ACM SIGCOMM*, 2011, pp. 86–97.

[8] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *ACM SIGCOMM*, 2000, pp. 87–95.

[9] 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," in *NSDI*, 2010, pp. 419–432.

[10] E. Zohar, I. Cidon, and O. Mokryn, "Pack: Prediction-based cloud bandwidth and cost reduction system," *IEEE/ACM Transactions on Networking*, vol. 22, no. 1, pp. 39–51, Feb 2014.

[11] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee, "Redundancy in network traffic: findings and implications," in *SIGMETRIC-S/Performance*, 2009, pp. 37–48.

[12] M.Rabin, "Fingerprinting by random polynomials," *Technical report Harvard University*, vol. TR-15-81, 1981.

[13] "Riverbed networks : Wan optimization." http://www.riverbed.com/solutions/optimize.

[14] "Juniper networks: Application acceleration." http://www.juniper.net/us/en/products-services/application-acceleration/, 1996.

[15] "Cisco wide are application acceleration services," http://www.cisco.com/en/US/products/ps5680/Products_Sub_Category_Home.html.

[16] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, "Packet caches on routers: the implications of universal redundant traffic elimination," in *ACM SIGCOMM*. ACM, 2008, pp. 219–230.

[17] A. Anand, V. Sekar, and A. Akella, "Smartre: an architecture for coordinated network-wide redundancy elimination," in *ACM SIGCOMM*. ACM, 2009, pp. 87–98.

[18] "Paas." [Online]. Available: http://www.ibm.com/cloud-computing/us/en/paas.html

[19] "Dropbox." [Online]. Available: www.dropbox.com

[20] S.-H. Shen, A. Gember, A. Anand, and A. Akella, "Refactor-ing content overhearing to improve wireless performance," in *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '11. New York, NY, USA: ACM, 2011, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/2030613.2030638

[21] E. Zohar, I. Cidon, and O. O. Mokryn, "Celleration: Loss-resilient traffic redundancy elimination for cellular data," in *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, ser. HotMobile '12. New York, NY, USA: ACM, 2012, pp. 10:1–10:6. [Online]. Available: http://doi.acm.org/10.1145/2162081.2162096

[22] S. Woo, E. Jeong, S. Park, J. Lee, S. Ihm, and K. Park, "Comparison of caching strategies in modern cellular backhaul networks," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '13. New York, NY, USA: ACM, 2013, pp. 319–332. [Online]. Available: http://doi.acm.org/10.1145/2462456.2464442

[23] Y. Zhang and N. Ansari, "On protocol-independent data redundancy elimination," *Communications Surveys Tutorials, IEEE*, vol. 16, no. 1, pp. 455–472, First 2014.

[24] S. Ihm, K. Park, and V. S. Pai, "Wide-area network acceleration for the developing world," in *USENIX annual technical conference*. USENIX Association, 2010, pp. 18–18.

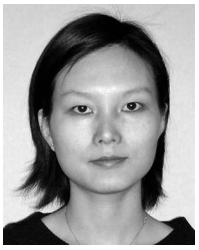
[25] G. Wright and W. Stevens., *TCP/IP Illustrated, Volume2: The Implementation.* Addison-Wesley,Massachusetts, 1995.

[26] L. Yu, K. Sapra, H. Shen, and L. Ye, "Cooperative end-to-end traffic redundancy elimination for reducing cloud bandwidth cost," in *Proc. of ICNP*, ser. ICNP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICNP.2012.6459973

[27] "Pack source code," http://www.venus-c.eu/pages/partner.aspx?id=10.

[28] "netfilter/iptables:libnetfilter_queue," http://www.netfilter.org/projects/libnetfilter_queue, Oct 2005.

Lei Yu received the PhD degree in computer science from Harbin Institute of Technology, China, in 2011. During 2011 to 2013 He was a post doctoral research fellow in the Department of Electrical and Computer Engineering at Clemson University, SC, United States. His research interests include sensor networks, wireless networks, cloud computing and network security.



Haiying Shen received the BS degree in Computer Science and Engineering from Tongji University, China in 2000, and the MS and Ph.D. degrees in Computer Engineering from Wayne State University in 2004 and 2006, respectively. She is currently an Assistant Professor in the Department of Electrical and Computer Engineering at Clemson University. Her research interests include distributed computer systems and computer networks, with an emphasis on peer-to-peer and content delivery networks, mobile computing, wireless sensor networks, and grid and cloud computing. She was the Program Co-Chair for a number of international conferences and member of the Program Committees of many leading conferences. She is a Microsoft Faculty Fellow of 2010, a senior member of the IEEE and a member of the ACM.

Karan Sapara received the BS degree in computer engineering from Clemson University, in 2011, and is currently working toward the PhD degree in the Department of Electrical and Computer Engineering at Clemson University. His research interests include P2P networks, distributed and parallel computer systems, and cloud computing. He is a student member of the IEEE.

Lin Ye received the B.S., M.S., and Ph.D. degrees in Computer Science from Harbin Institute of Technology (HIT), Harbin, China, from 2000 to 2011. In 2012, he was a visiting scholar at Temple University, Philadelphia, PA, USA. He is currently an assistant professor in School of Computer Science and Technology in HIT. His research interests include Peer-to-Peer measurement, cloud computing and information security.

Zhipeng Cai received his PhD and M.S. degree in Department of Computing Science at University of Alberta, and B.S. degree from Department of Computer Science and Engineering at Beijing Institute of Technology. Dr. Cai is currently an Assistant Professor in the Department of Computer Science at Georgia State University. Prior to joining GSU, Dr. Cai was a research faculty in the School of Electrical and Computer Engineering at Georgia Institute of Technology. Dr. Cai's research areas focus on Networking and Big data. Dr. Cai is the recipient of an NSF CAREER Award.