EndRE: An End-System Redundancy Elimination Service for Enterprises

Bhavish Aggarwal⋆, Aditya Akella†, Ashok Anand[1]†, Athula Balachandran[1]‡,
Pushkar Chitnis⋆, Chitra Muthukrishnan[1]†, Ramachandran Ramjee⋆ and George Varghese[2]§

⋆ Microsoft Research India; † University of Wisconsin-Madison; ‡ CMU; §UCSD

Abstract

In many enterprises today, middleboxes called WAN optimizers are being deployed across WAN access links in order to eliminate redundancy in network traffic and reduce WAN access costs. In this paper, we present the design and implementation of EndRE, an alternate approach where redundancy elimination is provided as *an end system service*. Unlike middleboxes, such an approach benefits both end-to-end encrypted traffic as well as traffic on last-hop wireless links to mobile devices.

EndRE needs to be fast, adaptive and parsimonious in memory usage in order to opportunistically leverage resources on end hosts. Thus, we design a new fingerprinting scheme called SampleByte that is much faster than Rabin fingerprinting while delivering similar compression gains. Unlike Rabin, SampleByte can also adapt its CPU usage depending on server load. Further, we introduce optimizations to reduce server memory footprint by 33-75% compared to prior approaches. Using several terabytes of network traffic traces from 11 enterprise sites, testbed experiments and a pilot deployment, we show that EndRE delivers 26% bandwidth savings on average, processes payloads at speeds of 1.5-4Gbps, reduces end-to-end latencies by up to 30%, and translates bandwidth savings into equivalent energy savings on mobile smartphone.

1 Introduction

With the advent of globalization, networked services have a global audience, both in the consumer and enterprise spaces. For example, a large corporation today may have branch offices at dozens of cities around the globe. In such a setting, the corporation's IT admins and network planners face a dilemma. On the one hand, they could centralize or concentrate the servers that power the corporation's IT services (e.g., email and file servers) at one or a small number of locations. This would keep administration costs low but may drive up network costs and also hurt performance, because, for instance, what would have normally been LAN traffic becomes WAN traffic. On the other hand, the servers and services could be distributed to be closer to clients. However, this would likely drive up the complexity and cost of developing and administering the services.

This paper arises from the quest to have the best of both worlds, specifically, having the operational benefits of centralization along with the performance benefits of distribution. In recent years, protocol-independent redundancy elimination (RE) [21] has emerged as a powerful technique to help bridge the gap by making WAN communication more efficient through elimination of redundancy in traffic. Such compression is typically applied at the IP or TCP layer, for instance, using a pair of middleboxes placed at either end of a WAN link connecting a corporation's data center and a branch office. Each box stores the payload from any flow traversing the link between them in a cache, irrespective of the application or protocol. When one box detects chunks of data that match entries in its cache (by computing "fingerprints" of incoming data and matching them against cached data), it encodes the matched data with tokens. The box at the far end reconstructs the original data using its own cache and the encoded tokens. Recently, this approach has seen increasing commercial deployment as part of a suite of optimizations in middleboxes called WAN optimizers [2].

Middlebox-based solutions have two key drawbacks that impact their overall usefulness in the long term. With the standardization of SSL, SSH, and IPSec, there is a growing shift toward end-to-end encryption of data. Unfortunately, middleboxes do not cope well with traffic encrypted end-to-end, and many leave such data uncompressed (e.g., [1]). A small fraction of middleboxes (e.g., [6]) employ tricks such as connection termination and sharing of encryption keys to accommodate SSL and SSH traffic, but these weaken end-to-end semantics of enterprise transactions considerably. The second drawback is that in-network middleboxes do nothing to improve performance over last-hop links of mobile and wireless devices; these devices are beginning to overrun the enterprise workplace.

If end-to-end encryption does become ubiquitous, and the adoption of resource constrained mobile and wireless devices continues its upward trend, then RE will eventu-

[1]A part of this work was done while the authors were interns at Microsoft Research India.

[2]The author was a visiting researcher at Microsoft Research India during the course of this work.

ally be *forced out* of middleboxes and directly *into* end-host stacks. Motivated by this, we explore a new point in the design space of RE proposals — an *end-system redundancy elimination service* called EndRE. The EndRE service could either supplement or supplant middlebox-based techniques and help bridge the fundamental drawbacks of middleboxes. While we cannot anticipate future trends, our paper examines the ramifications of an end-host based design and its costs and benefits for both clients and servers, whose primary purpose (unlike middleboxes) is end-user computation.

Effective protocol-independent end-host RE requires looking for small redundant chunks of the order of 32-64 bytes (because most transfers involve just a few packets each [17]). The standard algorithms (e.g. [21]) for such fine scale redundancy are very expensive in memory and processing especially on resource constrained clients such as smart phones. Hence, we adopt a novel *asymmetric* design that systematically offloads as much of processing and memory to servers as possible, requiring clients to do no more than perform basic FIFO queue management of a small amount of memory and do simple pointer lookups to decode compressed data sent by the server.

While client processing and memory are paramount, servers in EndRE need to do other things as well (unlike middleboxes). This means that server CPU and memory are also crucial bottlenecks in our asymmetric design. For server processing, we propose a new fingerprinting scheme called SampleByte that is much faster than Rabin Fingerprinting used in traditional RE approaches while delivering similar compression. In fact, SampleByte can be up to 10X faster, delivering compression speeds of 1.5-4Gbps. SampleByte is also *tunable* in that it has a payload sampling parameter that can be adjusted to reduce server processing if the server is busy, at the cost of reduced compression gains.

For server storage, we devise a suite of highly-optimized data structures for managing meta-data and cached payloads. For example, our Max-Match variant of EndRE (§5.2.2) requires 33% lower memory compared to [21]. Our Chunk-Match variant (§5.2.1) cuts down the aggregate memory requirements at the server by 4X compared to [21], while sacrificing a small amount of redundancy.

We conduct a thorough evaluation of EndRE. We analyze several terabytes of network traffic traces from 11 different enterprise sites and show that EndRE can deliver significant bandwidth savings (26% average savings) on enterprise WAN links. We also show significant latency and energy savings from using the EndRE service. Using a testbed over which we replay enterprise HTTP traffic, we show that latency savings of up to 30% are possible from using the EndRE service, since it operates above TCP, thereby reducing the number of roundtrips needed for data transfer. Similarly, on mobile smartphones, we show that the low decoding overhead on clients can help translate bandwidth savings into an equivalent amount of energy savings compared to no compression. Finally, we also report results from a small-scale deployment of EndRE in our lab.

The benefits of EndRE come at the cost of memory and CPU resources on end systems. We show that a median EndRE client needs only 60MB of memory and negligible amount of CPU. At the server, since EndRE is adaptive, it can opportunistically trade-off CPU/memory for compression savings.

In summary, we make the following contributions:

(1) We present the design of EndRE, an end host based redundancy elimination service (§4).

(2) We present new asymmetric RE algorithms and optimized data structures that limit client processing and memory requirements, and reduce server memory usage by up to 33-75processing by up to 10X compared to [21] while delivering similar or slightly lower bandwidth savings (§5).

(3) We present an implementation of EndRE as part of the Windows Server/7/Vista as well as on Windows Mobile 6 operating systems (§6).

(4) Based on extensive analysis using several terabytes of network traffic traces from 11 enterprise sites, testbed experiments and a small-scale deployment, we quantify the benefits and costs of EndRE (§7 - §9)

2 Related Work

Over the years, enterprise networks have used a variety of mechanisms to suppress duplicate data from their network transfers.

Classical approaches: The simplest RE approach is to compress objects end-to-end. It is also the least effective because it does not exploit redundancy due to repeated access to similar content. Object caches (e.g. Web caches) can help eliminate redundancies due to repeated access, but they are largely ineffective today. A key reason is their protocol-specific nature. Our measurements indicate that nearly 50-70% of enterprise traffic is composed of protocols other than HTTP, and that this traffic has substantial redundancy (20-50%) [9]. A second reason, applying specifically to Web caches, is that an increasing amount of data is dynamically generated and hence not cacheable. In our enterprise traces, we found that a majority of Web objects are not cacheable and deploying a Web proxy would only yield 5% bandwidth savings for HTTP. Delta encoding has also been used for eliminating redundancy of one Web object with respect to another, often an earlier version of the object with the same name [15]. It was later extended to pairs of files that do not share an explicit versioning relationships [13]. Being application-specific, it suffers from the same drawbacks as Web caches.

Content-based naming: The basic idea underlying EndRE is that of *content-based naming* [16, 21], where an object is divided into chunks and indexed by computing hashes over chunks. Rabin Fingerprinting [19] is typically used to identify chunk boundaries.

In file systems such as LBFS [16] and Shark [10], content-based naming is used to identify similarities between files and versions of the same file. Only unique chunks along with (smaller) hashes of repeated chunks are transmitted between file servers and clients, thereby resulting in lower bandwidth utilization and lower file read/write latencies. A similar idea is used in Value-based Web Caching [20], albeit between a Web server and its client. Our chunk-based EndRE design is patterned after this approach, with key modifications for efficiency as detailed in §5.

Generalizing these systems, DOT [22] proposes a "transfer service" as an interface between applications and network layers. Applications pass the object they want to send to the transfer service that then ensures delivery of objects to receivers. Objects are split into chunks and the sender sends chunk hashes to the receiver. The receiver maintains a cache of earlier received chunks and requests the sender only for chunks that were not found in its cache or its neighbors' caches. Thus, DOT can leverage TBs of cache in the disks of an end host and its peers to eliminate redundancy. Similarly, SET [18] exploits chunk-level similarity in downloading related large files. DOT and SET are built essentially to benefit large transfers with average chunk sizes of 2KB and extra round trips that can only be amortized over large transfers. In contrast, EndRE identifies redundancy across chunk sizes of 32 bytes but is limited to cache sizes of 1-10MB per pair of hosts (§5). Thus, we believe that EndRE and DOT complement each other.

Protocol-independent WAN optimizers. To overcome the limitations of the "classical" approaches, enterprises have moved increasingly toward protocol independent RE techniques, used in WAN optimizers. These WAN optimizers can be of two types, depending on which network layer they operate at, namely, IP layer devices [21, 4] or higher-layer devices [1, 6].

In either case, special middleboxes are deployed at either end of a WAN link to index all content exchanged across the link, and identify and remove partial redundancies on the fly. Rabin fingerprinting [19] is used to index content and compute overlap (similar to [21, 16]). Both sets of techniques are highly effective at reducing the utilization of WAN links. However, as mentioned earlier, they suffer from two key limitations, namely, lack of support for end-to-end security and resource-constrained mobile devices.

3 Motivation

In exploring an end-point based RE service, one of the main issues we hope to address is whether such a service can offer bandwidth savings approaching that of WAN optimization middleboxes. To motivate the likely benefits of an end-point based RE service, we briefly review two key findings from our earlier study [9] of a middlebox-based IP-layer WAN optimizer [8], that was based on several terabytes of enterprise network packet traces (detailed in §7).

First, we seek to identify the origins of redundancy in a WAN optimizer. Specifically, we classify the contribution of redundant byte matches to bandwidth savings as either *intra-host* (current and matched packet in cache has identical source-dest IP addresses) or *inter-host* (current and matched packet differ in at least one of source-dest IP addresses). We were limited to a 250MB optimizer cache size given the large amount of meta-data necessary for this analysis, though we saw similar compression savings for cache sizes up to 2GB. Surprisingly, *our study revealed that over 75% of savings were from intra-host matches*. This implies that a pure end-to-end solution could potentially deliver a significant share of the savings obtained by a WAN optimizer middlebox, since the contribution due to inter-host matches is small. However, this finding holds good only if end systems operate with similar (large) cache sizes as middleboxes, which is impractical. This brings us to the second key finding. Examining the temporal characteristics of redundancy, we found that the redundant matches in the WAN optimizer cache displayed a high degree of temporal locality with *60-80% of middlebox savings due to matches with packets in the most recent 10% of the cache*. This implies that smaller caches should capture bulk of the savings of a large cache.

Taken together, these two findings suggest that an end point-based RE system with a small cache size can indeed deliver a significant portion of the savings of a WAN optimizer, thus, motivating the design of EndRE.

Finally, note that, the focus of comparison in this section is between an IP-layer WAN optimizer with an in-memory cache (O(GB)) and an end-system solution. The first finding is not as surprising once we realize that the in-memory cache gets recycled frequently (on the order of tens of minutes) during peak hours on our enterprise traces, limiting the possibility for inter-host matches. A WAN optimizer typically also has a much larger on-disk cache (O(TB)) which may see a large fraction of inter-host matches; an end-system disk cache-based solution such as DOT [22] could capture analogous savings.

4 Design Goals

EndRE is designed to optimize data transfers in the direction from servers in a remote data center to clients in the enterprise, since this captures a majority of enterprise traffic. We now list five design goals for EndRE — the first two design goals are shared to some extent by prior approaches, but the latter three are unique to EndRE.

1. Transparent operation: For ease of deploy-ability, the EndRE service should require no changes to existing applications run within the data center or on clients.

2. Fine-grained operation: Prior work has shown that many enterprise network transfers involve just a few packets [17]. To improve the latencies and provide bandwidth savings for such short flows, our approach must work at fine granularity's, suppressing duplicated byte strings as small as 32-64B. This is similar to the operation in [21], but different from earlier proposals for file-systems [16] and Web caches [20] where the sizes of redundancies identified is 2-4KB.

3. Simple decoding at clients: EndRE's target client set includes battery-and CPU-constrained devices such as smart-phones. While working on fine granularity's can help identify greater amount of redundancy, it can also impose significant computation and decoding overhead, making the system impractical for these devices. Thus, a unique goal is to design algorithms that limit the overhead on clients by *offloading* all computation intensive actions *to servers*.

4. Fast and adaptive encoding at servers: Since EndRE is designed to opportunistically leverage CPU resources on end hosts, redundancy suppression must be fast. Furthermore, CPU on end-hosts may be used by other applications. Thus, unlike commercial WAN optimizers and prior RE approaches [21], a second unique goal for EndRE is that it must be able to *adapt* its use of CPU based on the load at the server.

5. Limited memory footprint at servers and clients: EndRE relies on data caches to eliminate redundancy in traffic. However, the amount of memory on servers and clients could be limited, and may be actively used by other applications. Thus, another goal is that EndRE use as minimal memory on end-hosts as possible through the use of optimized data structures.

5 EndRE Design

In this section, we describe how EndRE's design meets the above goals.

EndRE introduces redundancy elimination modules into the network stacks of clients and remote servers. Since we wish to be transparent to applications, EndRE could be implemented either at the IP-layer or at the socket layer (above TCP). As we argue in §6, we believe that the socket layer is the right place to implement EndRE. Doing so offers key performance benefits over an IP-layer approach, and more importantly, shields EndRE from network-level events (e.g., packet losses and reordering), making it simpler to implement.

There are two sets of modules in EndRE, those belonging on servers and those on clients. The server side module is responsible for identifying redundancy in network data by comparing against a cache of prior data, and encoding the redundant data with shorter meta-data. The meta-data is essentially a set of (offset, length) tuples that are computed with respect to the client-side cache. The client-side module consists of a fixed-size circular FIFO log of packets (the client packet cache) and simple logic to decode the meta-data by "de-referencing" the offsets sent by the server.

Thus, the complexity in redundancy elimination is mainly on the server side EndRE module. In particular, when a new data block is received by the server module, it needs to efficiently identify and encode contiguous strings of repeated content in the data block with respect to a cache of prior data. Since this process has to be fast, this is typically accomplished [21, 8] by the following two steps:

• *Fingerprinting:* Selecting a few "representative regions" for the current block of data handed down by application(s). We describe four fingerprinting algorithms in Section 5.1 that differ in the trade-off between the *computational overhead* imposed on the server and the *effectiveness* of redundancy elimination

• *Matching and Encoding:* Once the representative regions are identified, we examine two approaches for identification of redundant content in Section 5.2: (1) Identifying chunks of representative regions that repeat in full across data blocks, called Chunk-Match and (2) Identifying maximal matches around the Representative regions that are repeated across data blocks, called Max-Match. These two approaches differ in the trade-off between the *memory overhead* imposed on the server and the *effectiveness* of redundancy elimination.

Next, we describe EndRE's design in detail, starting with selection of representative regions, and moving on to identifying and removing redundancy.

5.1 Fingerprinting: Balancing Server Computation with Effectiveness

In this section, we outline four approaches for identifying the representative payload regions at the server that vary in the way they trade-off between computational overhead and the effectiveness of redundancy elimination. Furthermore, in some of the approaches, the computational overhead can be *adaptively tuned* based on server CPU load and the effectiveness of RE varies accordingly. Although three of the four approaches were proposed earlier, the issue of their computational overhead has not received enough attention. Since this issue is paramount for EndRE, we consider it in great depth here. We also propose a new approach, SAMPLEBYTE, that combines the salient aspects of prior approaches.

Notation and terminology: We first introduce some notation and terminology to help explain the approaches. Restating from above, a "data block" or simply a "block" is a certain amount of data handed down by an application to the EndRE module at the socket layer. Each data block can range from a few bytes to tens of kilobytes in size.

Let w represent the size of the minimum redundant string (contiguous bytes) that we would like to identify. For a data block of size S bytes, $S \geq w$, a total of $S - w + 1$ strings of size w are potential candidates for finding a match. Typical values for w range from $12 - 64$ bytes. Based on our findings of redundant match length distribution in [9], we choose a default value of $w = 32$ bytes to maximize effectiveness of redundancy elimination. Since $S >> w$, the number of such candidate strings is on the order of the number of bytes in the data block/cache. Since it is impractical to match/store all possible candidates, a fraction $1/p$ "representative" candidates are chosen.

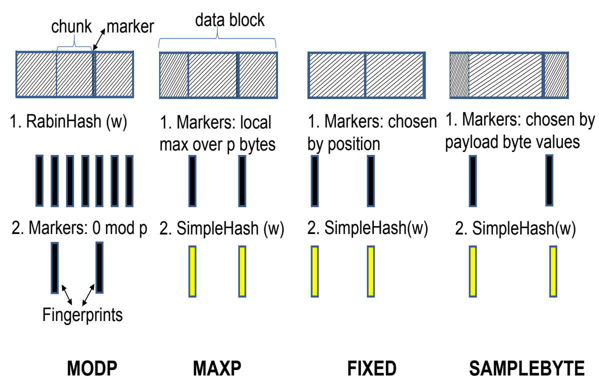


Figure 1: Fingerprinting algorithms with chunks, markers and fingerprints; chunk-hashes, not shown, can be derived from chunks

Let us define *markers* as the first byte of these chosen candidate strings and *chunks* as the string of bytes between two markers. Let *fingerprints* be a pseudo-random *hash of fixed w-byte strings* beginning at each marker and *chunk-hashes* be *hashes of the variable sized chunks*. Note that two fingerprints may have overlapping bytes; however, by definition, chunks are disjoint. The different algorithms, depicted in Figure 1 and discussed below, primarily vary in the manner in which they choose the markers, from which one can derive chunks, fingerprints, and chunk-hashes. As we discuss later in Section 5.2, the Chunk-Match approach uses chunk-hashes while Max-Match uses fingerprints.

5.1.1 MODP

```
1  //Let w = 32; p = 32; Assume len ≥ w;
2  //RabinHash() computes RABIN hash over a w byte window
3  MODP(data, len)
4      for(i = 0; i < w − 1; i++)
5          fingerprint = RabinHash(data[i]);
6      for(i = w − 1; i < len; i++)
7          fingerprint = RabinHash(data[i]);
8          if (fingerprint % p == 0) //MOD
9              marker = i − w + 1;
10             store marker, fingerprint in table;
```

Figure 2: MODP Fingerprinting Algorithm

In the "classical" redundancy elimination approaches [21, 8, 16], the set of fingerprints are chosen by first computing Rabin-Karp hash [19] over sliding windows of w contiguous bytes of the data block. A fraction $1/p$ are chosen whose fingerprint value is *0 mod p*. Choosing fingerprints in this manner has the advantage that the set of representative fingerprints for a block remains mostly the same despite small amount of insertions/deletions/reorderings since the markers/fingerprints are chosen based on content rather than position.

However, note that two distinct operations — marker identification and fingerprinting — are both handled by the same hash function. While this *appears* elegant, it has a cost. Specifically, the per block computational cost is independent of the sampling period, p (lines 4–7 in Figure 2). Thus, this approach *cannot* adapt to server CPU load conditions (e.g., by varying p). Note that, while the authors of [21] report some impact of p on processing speed, this impact is attributed to the overhead of managing meta-data (line 10). We devise techniques in Section 5.2 to significantly reduce the overhead of managing meta-data, thus, making fingerprint computation the main bottleneck.

5.1.2 MAXP

Apart from the conflation of marker identification and fingerprinting, another shortcoming of the MODP approach is that the fingerprints/markers are chosen based on a *global* property, i.e., fingerprints have to take certain pre-determined values to be chosen. The markers for a given block may be clustered and there may be large intervals without any markers, thus, limiting redundancy identification opportunities.

In order to guarantee that an adequate number of fingerprints/markers are chosen uniformly from each block, markers can be chosen as those bytes that are the *local-maxima over each region of p bytes* of the data block [9]. Once the marker byte is chosen, an efficient hash function such as Jenkins Hash [3] can be used for computing the fingerprint. Finally, by increasing p, fewer maxima-based markers need to be identified, thereby reducing CPU overhead.

5.1.3 FIXED

While markers in both MODP and MAXP are chosen based on content of the data block, the computation of Rabin hashes and local maxima can be expensive. A simpler approach is to be content-agnostic and simply select *every pth byte as a marker*. Since markers are simply chosen by position, marker identification incurs no computational cost. Once markers are chosen, S/p fingerprints are computed using Jenkins Hash as in MAXP. Thus, while this technique is very efficient, its effectiveness in redundancy elimination is not clear given that this approach is not robust to small changes in content. While work in file systems such as [16], where cache sizes are large (O(TB)), argue against this approach, it is not clear how ineffective FIXED will be in identifying redundancy in EndRE where cache sizes are small (O(MB)).

5.1.4 SAMPLEBYTE

```
1  //Let w = 32; p = 32; Assume len ≥ w;
2  //SAMPLETABLE[i] maps byte i to either 0 or 1
3  //Jenkinshash() computes hash over a w byte window
4  SAMPLEBYTE(data, len)
5        for(i = 0; i < len − w; i++)
6            if (SAMPLETABLE[data[i]] == 1)
7                marker = i;
8                fingerprint = JenkinsHash(data + i);
9                store marker, fingerprint in table;
10               i = i + p/2;
```

Figure 3: SAMPLEBYTE Fingerprinting Algorithm

MAXP and MODP are content-based and thus robust to small changes in content, while FIXED is content-agnostic but computationally efficient. We designed SAMPLEBYTE (Figure 3) to combine the robustness of a content-based approach with the computational efficiency of FIXED. It uses a 256-entry lookup table with a few predefined positions set. As the data block is scanned byte-by-byte (line 5), *a byte is chosen as a marker if the corresponding entry in the lookup table is set* (line 6–7). Once a marker are chosen, a fingerprint is computed using Jenkins Hash (line 8) and $p/2$ bytes of content are skipped (line 10) before the process repeats. Thus, SAMPLEBYTE is content-based, albeit based on a single byte, while retaining the content-skipping and, thus, the computational characteristics of the FIXED approach.

One clear concern is whether such a naive marker identification approach will do badly and cause the algorithm to either over-sample or under-sample. First, note that MODP with 32-64 byte rolling hashes was originally used in file systems [16] where chunk sizes were large (2-4KB). Given that we are interested in sampling as frequent as every 32-64 bytes, sampling chunk boundaries based on 1-byte content values is not as radical as it might first seem. Also, note that if the x entries of the 256-entry lookup table is randomly set (where $256/x = p$), then the expected sampling frequency is indeed $1/p$. In addition, SAMPLEBYTE skips $p/2$ bytes after each marker selection to avoid oversampling when the content bytes of data block are not uniformly distributed (e.g., the same content byte is repeated contiguously). Finally, while a purely random selection of $256/x$ entries does indeed perform well in our traces, we use a lookup table derived based on the heuristic described below. This approach outperformed the random approach and we have found it effective after extensive testing on traces (see §8).

Since the number of unique lookup tables is large (2^{256}), we use an offline, greedy approach to generate this table. Using network traces from one of the enterprise sites we study (site 11 in Table 2) as training data, we first run MAXP to identify redundant content and then sort the characters by decreasing order of their presence in the identified redundant content. We then add these characters one at a time, setting the corresponding entries in the lookup table to 1, and stop this process when we see diminishing gains in compression. The intuition behind this approach is that we would like to increase the probability of being selected as markers to those characters that are more likely to be part of redundant content. The characters selected in this process from our training data were 0, 32, 48, 101, 105, 115, 116, 255. While the current approach results in a static lookup table, we are looking at online dynamic adaptation of the table as part of future work.

Finally, since SAMPLEBYTE skips $p/2$ bytes after every marker selection, the fraction of markers chosen is upper-bounded by $2/p$, irrespective of the number of entries set in the table. By increasing p, fewer markers/fingerprints are chosen, resulting in reduced computational overhead.

5.2 Matching and Encoding: Optimizing Storage and Client Computation

Once the markers and fingerprints are identified, identification of redundant content can be accomplished in two ways: (1) Identifying *chunks* of data that repeat in full across data blocks, called Chunk-Match and (2) Identifying *maximal matches around fingerprints* that are repeated across data blocks, called Max-Match. Both techniques were proposed in prior work. Chunk-Match was proposed in the context of file systems [16] and Web object compression [20]. Max-Match was proposed in the context of IP-layer RE middleboxes [21]. However, prior proposals impose significant storage and computation overhead.

In what follows we describe how the overhead impacts both servers and clients in either approach, and the two sets of techniques we employ to help address these overheads. The first technique is to leverage the *asymmetry* between servers and clients. We propose that clients offload most of the computationally intensive operations (e.g. hash computations) and memory management tasks to the server. The second technique is to exploit the inherent *structure* within the data maintained at servers and clients to optimize memory usage.

5.2.1 Chunk-Match

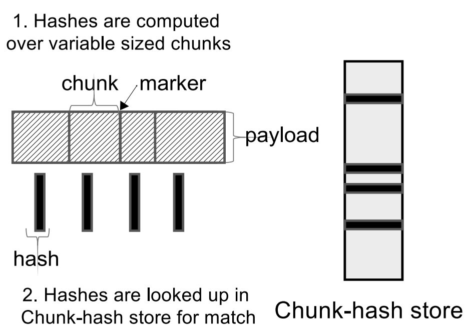


Figure 4: Chunk-Match: only chunk-hashes stored

This approach (Figure 4) stores hashes of the chunks in a data block in a "Chunk-hash store". Chunk-hashes from payloads of future data blocks are looked up in the Chunk-hash store to identify if one or more chunks have been encountered earlier. Once matching chunks are identified, they are replaced by smaller-sized meta-data.

Although similar approaches were used in prior systems, they impose significantly higher overhead if employed directly in EndRE. For example, in LBFS [16], clients have to update their local caches with mappings between new content-chunks and corresponding content-hashes. This requires expensive SHA-1 hash computation at the client. Value-based web caching [20] avoids the cost of hash computation at the client by having the server send the hash with each chunk. However, the client still needs to store the hashes, which is a significant overhead for small chunk sizes. Also, sending hashes over the network adds significant overhead given that hash sizes (20 bytes) are comparable to average chunk sizes in EndRE (32-64 bytes).

EndRE optimizations: We employ two ideas to improve the overhead on clients and servers.

(1) Our design carefully *offloads all storage management and computation functionality to servers*. A client simply maintains a fixed-size circular FIFO log of data blocks. The server emulates the client's cache behavior on a *per-client basis*, and maintains within its chunk-hash store a mapping of each chunk hash to the start memory addresses of the chunk in a client's log along with the length of the chunk. For each matching chunk, the server simply encodes and sends a four-byte (offset, length) tuple of the chunk in the client's cache. The client simply "de-references" the offsets sent by the server and reconstructs the compressed regions from local cache. This approach avoids the cost of storing and computing hashes at the client, as well as the overhead of sending hashes over the network, at the cost of slightly higher processing and storage at the server end.

(2) In traditional Chunk-Match approaches, the server maintains a log of the chunks locally. We observe that the server only needs to maintain an up-to-date chunk-hash store, but *it does not need to store the chunks themselves* as long as the chunk hash function is collision resistant. Thus, when a server computes chunks for a new data block and finds that some of the chunks are not at the client by looking up the chunk-hash store, it inserts mappings between the new chunk hashes and their expected locations in the client cache.

In our implementation, we use the SHA-1 algorithm to compute 160 bit hashes, which has good collision-resistant properties. Let us now compute the storage requirements for Chunk-Match assuming a sampling period of 64 bytes and a cache size of 16MB. The offset to the 16MB cache can be encoded in 24 bits and the length encoded in 8 bits assuming the maximum length of a chunk is limited to 256 bytes (recall that chunks are variable sized). Thus, the server meta-data storage is 24 bytes per 64-byte chunk, comprising 4-bytes for the (offset, length) tuple and 20-bytes for SHA-1 hash. This implies that the memory requirement at the server is about 38% of the client cache size.

5.2.2 Max-Match

A drawback of the Chunk-Match approach is that it can only detect exact matches in the chunks computed for a data block. It could miss redundancies that, for instance, span contiguous portions of neighboring chunks or redundancies that only span portions of chunks. An alternate approach, called Max-Match, proposed for middlebox-based RE in [21, 8] and depicted in Figure 5, can identify such redundancies , albeit at a higher memory cost at the server. In Max-Match, fingerprints computed for a data block serve as random "hooks" into the data block payload around which more redundancies can be identified.

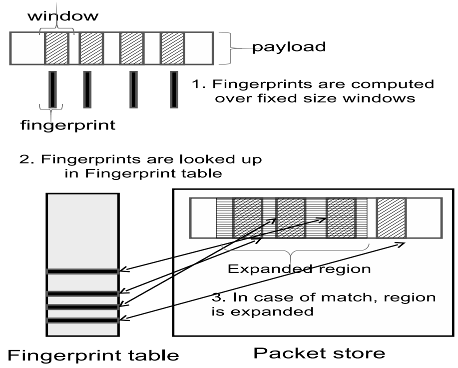


Figure 5: Max-Match: matched region is expanded

The computed fingerprints for a data block are compared with a "fingerprint store" that holds fingerprints of all the past data blocks. For each fingerprint of the data block that is matched against the fingerprint store, the matching previous data block is retrieved from the cache and the matching region is expanded byte-by-byte in both directions to obtain the *maximal region* of redundant bytes (Figure 5). Matched regions are then encoded with (offset, length) tuples identifying the matching region in the client's cache.

EndRE optimizations: We employ two simple ideas to improve the server computation and storage overhead.

First, since Max-Match relies on a byte-by-byte comparison between the bytes in the data block and those in the cache, fingerprint collisions are not costly; any collisions will be recovered via an extra memory lookup. This allows us to significantly *limit fingerprint store maintenance overhead for all four fingerprint algorithms* since fingerprint values are simply overwritten without separate bookkeeping for deletion. Further, a relatively simple hash function that generates a few bytes of hash value as a fingerprint (such as Jenkins hash [3]) is sufficient.

Second, we design an optimized representation of the fingerprint hash table that reduces storage needs significantly. Since the mapping is from a fingerprint (hash value) to an offset value, the fingerprint itself need not be stored in the table, at least in its entirety. The index into the fingerprint table can implicitly represent part of the fingerprint and only remaining bits, if any, of the fingerprint that is not covered by the index can be stored in the table. In the extreme case, the fingerprint table is simply a contiguous set of offset values, indexed by the fingerprint hash value.

Table 1 illustrates the fingerprint store for a cache size of 16MB and a sampling period, p, of 64. In this case, the number of fingerprints to index the entire cache is simply $2^{24}/64$ or 2^{18}. Using a table size of 2^{18} implies that 18 bits of the fingerprint is implicitly stored as the index of the table. The offset size necessary to represent the entire cache is 24 bits. Assuming we store an additional 8 bits of the fingerprint as part of the table, the entire fingerprint table can be compactly stored in a table of size $2^{18} * 4$ bytes, or 6% of the cache size. A sampling period of 32 would double this to 12% of cache size. This representation is a significant reduction in fingerprint meta-data size compared to the 67% indexing overhead in [21] or the 50% indexing overhead used in [8].

index (implicit fingerprint, 18 bits)	fingerprint remainder (8 bits)	offset (24 bits)
0	...	...
...	...	...
$2^{18}-1$	...	...

Table 1: 1MB Fingerprint store for 16MB cache

These two optimizations are not possible in the case of the Chunk-Match approach due to the more stringent requirements on the collision-resistance of the chunk hashes. However, memory requirements for Chunk-Match at the server is only 38% of cache size, which is still significantly lower than 106% of the cache size (cache + fingerprint store) needed for Max-Match.

6 Implementation

We now discuss our implementation of EndRE. We first discuss the benefits of implementing the RE service at the socket layer above TCP and then present our in-kernel socket-layer implementation of EndRE.

We discuss the benefits of implementing RE at the socket layer above TCP in terms of the overall performance offered and end-to-end benefits.

6.1 Performance benefits

Bandwidth: In the socket-layer approach, the redundancy elimination algorithm operates at the size of socket writes which are typically larger than IP layer MTUs. While Max-Match and Chunk-Match do not benefit from these larger sized writes since they operate at a granularity of 32bytes, the large size helps produce higher savings if a compression algorithm like GZIP is *additionally* applied at the EndRE, as evaluated in §9.1.

Latency: The socket-layer approach will result in fewer packets transiting between the server and the clients, as opposed to the IP layer approach which merely compresses packets without reducing their number. This is particularly useful in lowering connection completion times for short flows, as evaluated in §9.2.

6.2 End-to-end benefits

Encryption: When using socket-layer RE, payload encrypted in SSL can be compressed before encryption, providing RE benefits to protocols such as HTTPS. IP-layer RE will leave SSL traffic uncompressed.

Cache Synchronization: Recall that both Max-match and Chunk match-based RE algorithms require caches to be synchronized between client and server. One of the advantages of implementing the RE service above TCP is that TCP ensures reliable in-order delivery of payload between client and server, which can help with maintaining cache synchronization. However, there are still two issues that must be addressed.

First, multiple simultaneous TCP connections may be operating between a client and a server, resulting in ordering of data across connections not being preserved. We implement a simple sequence number-based *reordering mechanism* as part of the RE service.

Second, TCP connections may get reset in the middle of a transfer. Thus, packets written to the cache at server end may not even reach the client, leading to cache inconsistency. One could take a *pessimistic* or *optimistic* approach to maintaining consistency in this situation. In the pessimistic approach, writes to the server cache are performed only after TCP ACKs for the corresponding segments are received at the server. The server needs to monitor TCP state, detect ACKs, and then perform writes to its cache and notify the client to do the same. In the optimistic approach, the server writes to the cache but monitors TCP only for reset events. In case of connection reset (receipt of a TCP RST from client or a local TCP timeout), the server simply notifies the client of the last sequence number that was written to the cache for the corresponding TCP connection. The responsibility is then on the client to detect missing packets, if any, and recover these from the server. We adopt the *optimistic approach of cache writing* for two reasons: (1) Our redundancy analysis [9] indicated that there is high temporal locality of matches; a pessimistic approach over a high bandwidth-delay product link can negatively impact compression savings; (2) The optimistic approach is easier to implement since one needs to monitor only for reset events rather than every TCP-level acknowledgment.

6.3 Implementation

We have implemented the EndRE service above TCP in Windows Server/Vista/7. Our default fingerprinting algorithm §5.1 is SAMPLEBYTE with a sampling period, $p = 32$. Our packet cache is a circular buffer which maintains a configurable 1-16MB of payload history between two IP address pairs. Our fingerprint store is also allocated bounded memory based on the values presented earlier. We use a simple resequencing buffer with a priority queue to handle re-ordering, if any, across multiple parallel TCP streams. At the client side, we maintain a fixed size circular cache and the decoding process simply involves lookups of specified data segments in the cache.

Trace Name (Site #)	**Unique Client IPs**	**Dates (Total Days)**	**Size (TB)**
Small Enterprise (Sites 1-2)	29-39	07/28/08 - 08/08/08 (11) 11/07/08 - 12/10/08 (33)	0.5
Medium Enterprise (Sites 3-6)	62-91	07/28/08 - 08/08/08 (11) 11/07/08 - 12/10/08 (33)	1.5
Large Enterprise (Sites 7-10)	101-210	07/28/08 - 08/08/08 (11) 11/07/08 - 12/10/08 (33)	3
Large Research Lab (Site 11, training trace)	125	06/23/08 - 07/03/08 (11)	1

Table 2: Data trace characteristics (11 sites)

In order to enable protocol independent RE, we transparently capture application payloads on the server side and TCP payloads at the client side at the TCP stream layer, that lies between the application layer and the TCP transport layer. We achieve this by implementing a kernel level filter driver based on Windows Filtering Platform (WFP) [7]. This implementation allows seamless integration of EndRE service with all application protocols that use TCP, with no modification to application binaries or protocols. We also have a management interface that can be used to restrict EndRE service only to specific applications. This is achieved by predicate-based filtering in WFP, where predicates can be application IDs, source and/or destination IP addresses/ports.

Finally, we have also implemented the client-side of EndRE on mobile smartphones running the Windows Mobile 6 OS. However, since Windows Mobile 6 does not support Windows Filtering Platform, we have implemented the functionality as a user-level proxy.

7 Evaluation approach

We use a combination of trace-based and testbed-based evaluation to benchmark the performance of EndRE on servers and clients and to quantify its benefits. In particular, we quantify bandwidth savings and evaluate scalability aspects of EndRE using enterprise network traffic traces; we use a testbed to quantify processing speed and evaluate latency and energy savings. We also report results from a small pilot deployment in our lab over 1 week on 15 desktops.

Traces: Our trace-based evaluation is based on full packet traces collected continuously at the WAN access link of 11 corporate enterprise locations (44 days in total for sites 1-10). The key characteristics of our traces are shown in Table 2. We classify the enterprises as small, medium or large based on the number of internal host IP addresses seen (less than 50, 50-100, and 100-250, respectively) in the entire trace at each of these sites. While this classification is somewhat arbitrary, we use this division to study if the benefits vary depending on the size of an enterprise. Note that the total amount of traffic in each trace is also approximately correlated to the number of host IP addresses, though there is a large amount of variation from day to day. Typical incoming traffic numbers for small enterprises

Max-Match	Fingerprint		InlineMatch		Admin	
$p \rightarrow$	32	512	32	512	32	512
MODP	526.7	496.7	9.6	6.8	4.8	0.6
MAXP	306.3	118.8	10.1	7.7	5.2	0.5
FIXED	69.4	14.2	7.1	4.7	4.7	0.4
SAMPLEBYTE(SB)	76.8	20.2	9.5	6.1	3.0	0.7

Table 3: CPU Time(s) for different algorithms

varied from 0.3-10GB/day, for medium enterprises from 2-12GB/day and for large enterprises from 7-50GB/day. The access link capacities at these sites varied from a few Mbps to several tens of Mbps. Finally, the total size (including inbound/outbound and headers) of enterprise network traffic collected and analyzed is about 6TB.

Testbed: Our testbed consists of a desktop server connected to a client through a router. In wireline experiments, the router is a dual-homed PC capable of emulating links of pre-specified bandwidth and latency. In wireless experiments, the router is a WiFi access point. The server is a desktop PC running Windows Server 2008. The client is a desktop PC running Windows Vista or Windows 7 in the wireline experiments and a Samsung mobile smartphone running Windows Mobile 6 in the wireless experiments.

8 Costs

In this section, we quantify the CPU and memory costs of our implementation of EndRE. Though our evaluation focus largely on the the Max-Match approach, we also provide a brief analysis of Chunk-Match.

8.1 CPU Costs

Micro-benchmarks: We first focus on micro-benchmarks for the different fingerprinting algorithms using Max-Match for a cache size of 10MB (we examine cache size issue in detail in Section 8.2). Table 3 presents a profiler-based analysis of the costs of the three key processing steps on a single large packet trace as measured on a 2GHz 64-bit Intel Xeon processor. The fingerprinting step is responsible for identifying the markers/fingerprints; the InlineMatch function is called as fingerprints are generated; and the Admin function is used for updating the fingerprint store. Of these three steps, only the first step, fingerprinting, is distinct for the algorithms and is the most expensive step of the three.

One can clearly see that fingerprinting is expensive for MODP and is largely unaffected as p is increased from 32 to 512. Fingerprinting for MAXP is also expensive but we see that as p is increased, the cost of fingerprinting comes down. In the case of FIXED and SAMPLEBYTE, as expected, fingerprinting cost is low, with significant reductions as p is increased.

Finally, note that the optimizations detailed earlier for updating the fingerprint store in Max-Match results in low cost for the Admin function in all the algorithms. Since matching is interleaved with fingerprinting [21], the cost of fingerprinting and matching functions, and hence total processing speed, depend on the redundancy of a particular trace. We next compute *average* processing speed for the different algorithms over a large set of traces.

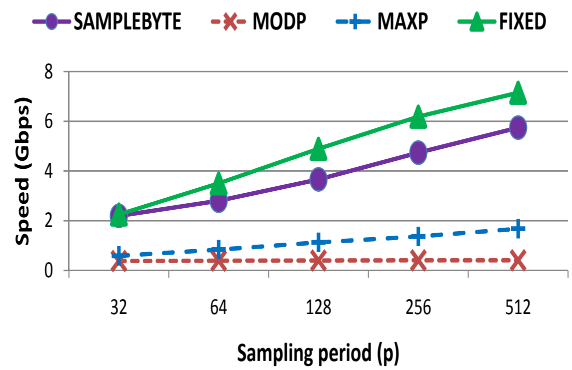


Figure 6: Max-Match Processing Speed

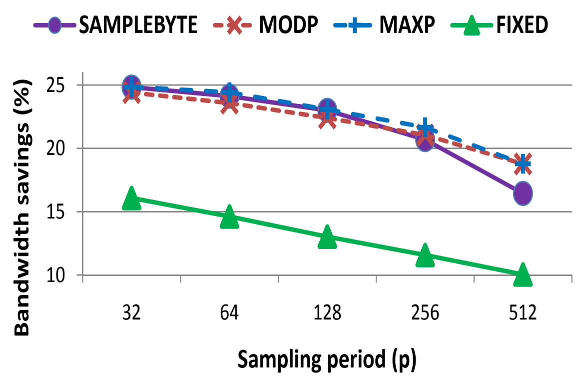


Figure 7: Max-Match Bandwidth Savings

Trace analysis: Figure 6 plots the average processing speed in Gbps at the server for the Max-Match approach for different fingerprinting algorithms, while Figure 7 plots the average bandwidth savings, assuming a packet cache size of 10MB, using the 11-day traces for enterprise sites 1-10 in Table 2.

We make a number of observations from these figures. First, the processing speed of MODP is about 0.4Gbps and, as discussed in Section 5, is largely unaffected by p. Processing speed for MAXP ranges from $0.6-1.7$Gbps, indicating that the CPU overhead can be decreased by increasing p. As expected, FIXED delivers the highest processing speed, ranging from $2.3-7.1$Gbps since it incurs no cost for marker identification. Finally, SAMPLEBYTE delivers performance close to FIXED, ranging from $2.2-5.8$Gbps, indicating that the cost of identification based on a single byte is low. Second, examining the compression savings, the curves for MODP, MAXP, and SAMPLEBYTE in Figure 7 closely overlap for the most part with SAMPLEBYTE underperforming the other two only when the sampling period is high (at $p = 512$, it appears that the choice of markers based on a single-byte may start to lose effectiveness). On the other hand, FIXED significantly underperforms the other three algorithms in terms of compression savings, though in absolute terms, the compression savings of FIXED is surprisingly high.

While the above results was based on a cache size of 10MB, typical for EndRE, a server is likely to have multiple simultaneous such connections in operation. Thus, it is unlikely to benefit from having beneficial CPU cache effects that the numbers above portray. We thus conducted experiments with large cache sizes (1-2GB) and found that processing speed indeed slows by about 30% for the algorithms. Taking this overhead into account, SAMPLEBYTE provides server processing speeds of $1.5 - 4$Gbps. To summarize, *SAMPLEBYTE provides just enough randomization for identification of chunk markers that allows it to deliver the compression savings of MODP/MAXP while simultaneously being inexpensive enough to deliver processing performance, similar to FIXED, of* $1.5 - 4Gbps$.

In the case of the Chunk-Match algorithm, the processing speed (not shown) is only 0.1-0.2Gbps. This is mainly due to expensive SHA1 hash computation (§5.2.1) and the inability to use the fingerprint store optimizations of Max-Match (§5.2.2). We are examining if a cheaper hash function with an additional mechanism to detect collision and recover payload through retransmissions in the Chunk-Match implementation will improve performance without impacting latency significantly.

Client Decompression: The processing cost for decompression at the end host *client* is negligible since EndRE decoding is primarily a memory lookup in the client's cache; our decompression speed is 10Gbps. We examine the impact of this in greater detail when we evaluate end-system energy savings from EndRE in §9.3.

8.2 Memory Costs

Since EndRE requires a cache per communicating client-server pair, quantifying the memory costs at both clients and servers is critical to estimating the scalability of the EndRE system. In the next two sections, we answer the following two key questions: 1) what cache size limit do we provision for the EndRE service between a single client-server pair? 2) Given the cache size limit for one pair, what is the cumulative memory requirements of the EndRE service on clients and servers?

8.2.1 Cache Size versus Savings

To estimate the cache size requirements for the EndRE service, we first need to understand the trade-off between cache sizes and bandwidth savings. We can then identify a good operating point for the cache size limit to be that cache size value where increasing the cache size provides diminishing gains in bandwidth savings.

For the following discussion, unless otherwise stated, by cache size, we refer to the client cache size limit for EndRE service with a given server. The server cache size can be estimated from this value depending on whether Max-Match or Chunk-Match is used. For example, assuming a sampling period of 64 bytes, the total server memory requirement can be as much as 106% of the client's cache size in the Max-Match approach and 38% of the client's cache size in the Chunk-Match approach (§5). Further, while one could provision different cache size limits for each client-server pair, for administrative simplicity, we assume that cache size limits are identical for all EndRE nodes.

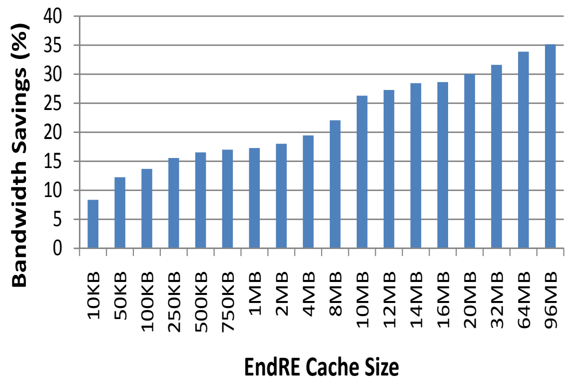


Figure 8: Cache Size vs Overall Bandwidth Savings

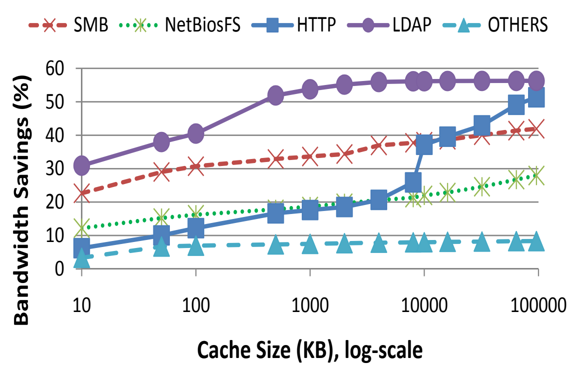


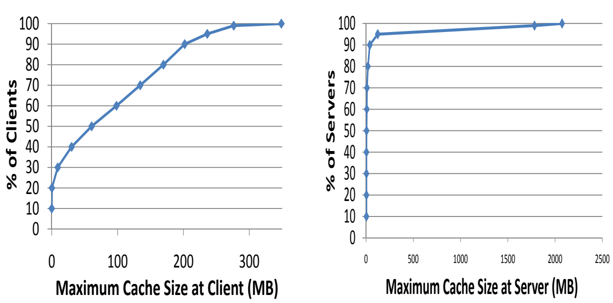
Figure 9: Cache Size vs Protocol Bandwidth Savings

Using the 11-day (1TB) enterprise traces over all ten enterprise sites, Figure 8 presents the overall bandwidth savings versus cache sizes for the EndRE service using the Max-Match approach. Although not shown, the trends are similar for the Chunk-Match approach. Based on the figure, a good operating point for EndRE is at the knee of the curve corresponding to 10MB of cache, allowing for a good trade-off between memory resource constraints and bandwidth savings.

To get a closer look, Figure 9 plots the bandwidth savings versus cache sizes (in log-scale for clarity) for different protocols. For this trace set, HTTP (port 80,8080) comprised 45% of all traffic, SMB (port 445) and NetBios File sharing (port 139) together comprised 26%, LDAP (port 389) was about 2.5% and a large set of protocols, labeled as OTHERS, comprised 26.5%. While different protocols see different bandwidth savings, all protocols, except for OTHERS, see savings of 20+% with LDAP seeing the highest savings of 56%. Note that OTHERS include several protocols that were encrypted (HTTPS:443, Remote Desktop:3389, SIP over SSL:5061, etc.). For this analysis, since we are estimating EndRE savings from IP-level packet traces whose payload is already encrypted, EndRE sees 0% savings. An implementation of EndRE in the socket layer would likely provide higher savings for protocols in the OTHERS category than estimated here. Finally, by examining the figure, one can see that knee-of-the-curve at different values for different protocols (10MB for HTTP, 4MB for SMB, 500KB for LDAP, etc.). This also confirms that the 10MB knee of Figure 8 is due to the 10MB knee for HTTP in Figure 9.

This analysis suggests that the cache limit could be tuned depending on the protocol(s) used between a client-server pair without significantly impacting overall bandwidth savings. Thus, *we use 10MB cache size only if HTTP traffic exists between a client-server pair, 4MB if SMB traffic exists, and a default 1MB cache size otherwise.* Finally, while this cache size limit is derived based on static analysis of the traces, we are looking at designing dynamic cache size adaptation algorithms for each client-server pair as part of future work.

8.2.2 Client and Server Memory Costs



(a) Client (b) Server

Figure 10: Cache scalability

Given the cache size limits derived in the previous section, we now address the critical question of EndRE scalability based on the *cumulative* cache needs at the client and server for all their connections. Using the entire set of network traces of ten enterprise sites (44 days, 5TB) in Table 2, we emulate the memory needs of EndRE with the above cache size limits for all clients and servers. We use a conservative memory page-out policy in the emulation, viz., if there has been no traffic for over ten hours between a client-server pair, we assume that the respective EndRE caches at the nodes are paged to disk. For each node, we then compute the maximum in-memory cache needed for EndRE over the entire 44 days.

Figure 10(a) plots the CDF of the client's maximum EndRE memory needs for all ($\approx$ 1000) clients. We find that *the median (99 percentile) EndRE client allocates a maximum cache size of 60MB (275MB)* during its operation over the entire 44-day period. We also performed an independent study of desktop memory availability by monitoring memory availability at 1 minute intervals for 110 desktops over 1 month at one of the enterprise sites. Analyzing this data, we found that the *5, 50 and 90 percentile values of unused memory, available for use, at these enterprise desktops were 245MB, 873MB, and 1994MB, respectively*. This validates our hypothesis that desktop memory resources are typically adequately provisioned in enterprises, allowing EndRE to operate on clients without significant memory installation costs.

We now examine the size of the cache needed at the server. First, we focus on the Max-Match approach and study the net size of the cache required across all active clients at the server. Using the same enterprise trace as above, we plot the CDF of server cache size for all the servers in the trace in Figure 10(b). Note that since we do not have trace access to all traffic at the server/data center links, this analysis only represents a lower bound on the cache size of the server, representing the needs of clients at ten enterprises. From the figure, we find that the maximum cache requirement is about 2GB. If it is not feasible to add extra memory to servers, say due to cost or slot limitations, the Chunk-Match approach could be adopted instead. This would reduce the maximum cache requirement of Max-Match by 3X (based on the calculations outlined earlier in §5).

9 Benefits

In this section, we characterize the various benefits of EndRE. We first investigate WAN bandwidth savings of EndRE. We then quantify latency savings of using EndRE, especially on short transactions typical of HTTP. Finally, we quantify the energy savings on mobile smartphones, contrasting EndRE with prior work on energy aware compression [12].

9.1 Bandwidth Savings

In this section, we focus on the bandwidth savings of different EndRE approaches for each of the enterprise

Site	Trace Size GB	GZIP 10ms	EndRE Max-Match 10MB				EndRE Max-Match+GZIP 10MB	EndRE Chunk-Match 10MB	EndRE Max-Match + DOT 10MB	IP WAN-Opt Max-Match 2GB	IP WAN-Opt Max-Match + DOT 2GB
		% savings									
			MODP	MAXP	FIXED	SB	SB	MODP	SB	SB	SB
1	173	9	47	47	16	47	48	46	56	71	72
2	8	14	24	25	19	24	28	19	33	33	33
3	71	17	25	26	23	26	29	22	32	34	35
4	58	17	23	24	20	24	31	21	30	45	47
5	69	15	26	27	22	27	31	21	37	39	42
6	80	12	21	21	18	22	26	17	28	34	36
7	80	14	25	25	22	26	30	21	33	31	33
8	142	14	22	23	18	22	28	19	30	34	40
9	198	9	16	16	14	16	19	15	26	44	46
10	117	13	20	21	17	21	25	17	30	27	30
Avg/site	100	13	25	26	19	26	30	22	34	39	41

Table 4: Percentage Bandwidth savings on incoming links to 10 enterprise sites over 11 day trace.

sites, and examine the gains of augmenting EndRE with GZIP and DOT [22]. We also present bandwidth savings of an emulated WAN optimizers for reference.

Table 4 compares the bandwidth savings on incoming links to ten enterprise sites for various approaches. This analysis is based on packet-level traces and while operating at packet sizes or larger buffers make little difference to the benefits of EndRE approaches, buffer size can have a significant impact on GZIP-style compression. Thus, in order to emulate the benefits of performing GZIP at the socket layer, we aggregate consecutive packet payloads for up to 10ms and use this aggregated buffer while evaluating the benefits of GZIP. For EndRE, we use the cache sizes of up to 10MB as derived from the previous section. We also emulate an IP-layer middlebox-based WAN optimizer with a 2GB cache.

We make several observations from the table. First, performing GZIP in isolation on packets aggregated for up to 10ms provides per-site average savings of 13%. Further, there are site specific variations, in particular, it performs poorly for site 1 compared to other approaches. Second, comparing the four fingerprinting algorithms (columns 3-6 in Table 4), we see that MODP, MAXP, and SAMPLEBYTE deliver similar average savings of 25-26% while FIXED under-performs. In particular, in the case of site 1, FIXED significantly under-performs the other three approaches. This again illustrates how SAMPLEBYTE captures enough content-specific characteristics to significantly outperform FIXED. Adding GZIP compression to SAMPLEBYTE improves the average savings to 30% (column 7). While the above numbers were based on Max-Match matching, using Chunk-Match instead reduces the savings to 22% (column 8), but this may be a reasonable alternative if server memory is a bottleneck.

We then examine savings when EndRE is augmented with an an approach like DOT [22]. For DOT analysis, we employ a heuristic to extract object chunks from the packet traces as follows: we combine consecutive packets of the same four-tuple flow and delineate object boundaries if there is no packet within a time window (1s). In order to ensure that the DOT analysis adds redundancy not seen by EndRE, we conservatively add only inter-host redundancy obtained by DOT to the EndRE savings. We see that (third column from right) DOT improves EndRE savings by a further 6-10%, and the per-site average bandwidth savings to 34%. For reference, the WAN optimizer with 2GB cache provides per-site savings of 39% and if DOT is additionally applied (where redundancy of matches farther away than 2GB are only counted), the average savings goes up by only 2%. Thus, it appears that half the gap between EndRE and WAN optimizer savings comes from inter-host redundancy and the other half from the larger cache used by the WAN optimizer.

Summarizing, EndRE using the Max-Match approach with the SAMPLEBYTE algorithm provides average per-site savings of 26% and delivers two-thirds of the savings of a IP-layer WAN optimizer. When DOT is applied in conjunction, the average savings of EndRE increases to 34% and can be seen to be approaching the 41% savings of the WAN optimizer with DOT.

Pilot Deployment: We now report results from a small scale deployment. EndRE service was deployed on 15 desktop/laptop clients (11 users) and one server for a period of about 1 week (09/25/09 to 10/02/09) in our lab. We also hosted a HTTP proxy at the EndRE server and users manually enabled/disabled the use of this proxy, at any given time, using a client-based software. During this period, a total of 1.7GB of HTTP traffic was delivered through the EndRE service with an average compression savings of 31.2%. A total of 159K TCP connections were serviced with 72 peak active simultaneous TCP connections and peak throughput of 18.4Mbps (WAN link was the bottleneck). The CPU utilization at the server remained within 10% including proxy processing. The number of packet re-orderings was less than 1% even in the presence of multiple simultaneous

TCP connections between client and server. We also saw a large number of TCP RSTs but, as reported in [11], these were mostly in lieu of TCP FINs and thus do not contribute to cache synchronization issues. Summarizing, even though this is a small deployment, the overall savings numbers match well with our analysis results and the ease of deployment validates the choice of implementing EndRE over TCP.

9.2 Latency Savings

In this section, we evaluate the latency gains from deploying EndRE. In general, latency gains are possible for a number of reasons. The obvious case is due to reduction of load on the bottleneck WAN access link of an enterprise. Even when the bottleneck link is lightly loaded, another set of latency gains arise from the choice of implementing the EndRE at the socket layer above TCP. Performing redundancy elimination above the TCP layer implies that the reduction in amount of data transferred will result in the reduction of number of packets and, thus, TCP round-trips necessary for completing the data transfer. In the case of large file transfers, since TCP would mostly be operating in the steady-state congestion avoidance phase, the percentage reduction in data transfer size translates into a commensurate reduction in file download latency. Thus, for large file transfers, say, using SMB or HTTP, one would expect latency gains similar to the average bandwidth gains seen earlier.

Latency gains in the case of short data transfers, typical of HTTP, are harder to estimate. This is because TCP would mostly be operating in slow-start phase and a given reduction in data transfer size could translate into a reduction of zero or more round trips depending on many factors including original data size and whether or not the reduction occurs uniformly over the data.

In order to quantify latency gains for short file transfers, we perform the following experiment. From the enterprise network traces, we extract HTTP traffic that we then categorize into a series of session files. Each session file consists of a set of timestamped operations starting with a connect, followed by a series of sends and receives (transactions), and finally a close.

The session files are then replayed on a testbed consisting of a client and a server connected by a PC-based router emulating a high bandwidth, long latency link, using the mechanism described in [14]. During the replay, strict timing is enforced at the start of each session based on the original trace; in the case of transactions, timing between the start of one transaction and the start of the next transaction is preserved as far as possible. The performance metric of interest is latency gain which is defined as the ratio of reduction in transaction time due to EndRE to transaction time without EndRE.

Table 5 shows the latency gain for HTTP for various transactions sorted by the number of round-trips in the original trace. For this trace, only 40% of HTTP transactions involved more than one round trip. For these transactions, latency gains on average ranged from 20% to 35%. These gains are comparable with the the average bandwidth savings due to EndRE for this trace of approximately 30%, thus demonstrating that even short HTTP transactions see latency benefits due to RE.

RTTs	**1**	**2**	**3**	**4**	**5**	> 5
Latency Gain	0	20%	23%	36%	20%	22%

Table 5: HTTP Latency gain for different RTTs

	None	ZLIB						EndRE	
	Energy uAh	Energy % savings			Byte % savings			Energy % savings	Byte % savings
	0 ms	0 ms	10 ms	100 ms	0 ms	10 ms	100 ms	0 ms	0 ms
A	5539	-59	-4	8	26	33	35	25	29
B	1743	-230	33	51	41	70	74	70	76
C	196	-25	-8	-3	12	31	32	42	47

Table 6: Energy savings on a mobile smartphone

9.3 Energy Savings

We study the energy and bandwidth savings achieved using EndRE on Windows Mobile smartphones clients and compare it against both no compression as well as prior work on energy-aware compression [12]. In [12], the authors evaluate different compression algorithms and show that ZLIB performs best in terms of energy savings on resource constrained devices for decompression. We evaluate the energy and bandwidth gains while transferring three trace files over WiFi: traces A, B, and C are 50MB, 15MB, 0.2MB in size, respectively. The first two trace files are actual enterprise HTTP traces, with trace B being more compressible than trace A. Trace C is background traffic generated by running skype and e-buddy on a mobile phone.

In ZLIB, we accumulate data by buffering for 0-100ms before compression since ZLIB works better on larger block sizes. We assume that after download and decompression, each packet is consumed in memory and not written to disk, representative of content consumed by, for example, a browser. We use a hardware-based battery power monitor [5] for our energy measurements that samples at 5KHz and is accurate to within 1mA.

Table 6 shows energy and latency gains of using ZLIB and EndRE as compared to no compression. We make the following observations from the table. First, consider ZLIB. Even though ZLIB is able to compress the trace files up to 74%, the energy savings significantly trail bandwidth savings. In fact, when ZLIB is applied on every packet, even though it saves bandwidth, it results in increased energy consumption (negative energy savings). This is due to the computational overhead of ZLIB decompression, which on these resource-constrained smartphones translate into extra energy expenditure as compared to no compression. In contrast, since decompression is a simple pointer lookup in the case of EndRE, we find that the bandwidth savings directly translate into comparable energy savings.

Thus, compared to no compression, while both ZLIB and EndRE can provide bandwidth savings, bandwidth savings of ZLIB does not translate into equivalent energy savings, and, in some cases, can even result in extra energy expenditure up to 230% while EndRE achieves energy gains comparable to the compression gains.

10 Conclusion

Using extensive traces of enterprise network traffic and testbed experiments, we show that our end host based redundancy elimination service, EndRE, provides average bandwidth gains of 26% and, in conjunction with DOT, approaches savings provided by a WAN optimizer. Further, EndRE can process payload at speeds of 1.5-4Gbps, provides latency savings of up to 30% and translates bandwidth savings into comparable energy savings on mobile smartphones.

In order to achieve these benefits, EndRE utilizes memory and CPU resources of end systems. For enterprise clients, we show that median memory requirements for EndRE is only 60MB. At the server end, we design mechanisms for working with reduced memory and adapting to CPU load. Thus, we believe that the cleaner semantics of end-to-end redundancy removal can come with considerable performance benefits and reasonable costs, enabling EndRE to be a compelling alternative to middlebox-based WAN optimizers.

Acknowledgments

We would like to thank our shepherd Sylvia Ratnasamy and the anonymous reviewers for their time and insightful comments regarding this paper. Aditya Akella, Ashok Anand, and Chitra Muthukrishnan were supported in part by an NSF FIND grant (CNS-0626889), an NSF CAREER Award (CNS-0746531), an NSF NetSE grant (CNS-0905134), and by grants from the UW-Madison Graduate School and Cisco Systems.

References

[1] Cisco Wide Area Application Acceleration Services. `http://www.cisco.com/en/US/products/ps5680/Products_Sub_Category_Home.html`.

[2] Computerworld - WAN optimization continues growth. `www.computerworld.com.au/index.php/id;1174462047`.

[3] Jenkins Hash. `http://burtleburtle.net/bob/c/lookup3.c`.

[4] Peribit Networks (Acquired by Juniper in 2005): WAN Optimization Solution. `http://www.juniper.net/`.

[5] Power Monitor, Monsoon Solutions. `http://www.msoon.com/powermonitor/powermonitor.html`.

[6] Riverbed Networks: WAN Optimization. `http://www.riverbed.com/solutions/optimize/`.

[7] Windows Filtering Platform. `http://msdn.microsoft.com/en-us/library/aa366509(V.85).aspx`.

[8] 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*, Seattle, WA, Aug. 2008.

[9] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee. Redundant in Network Traffic: Findings and Implications. In *ACM SIGMETRICS*, Seattle, WA, June 2009.

[10] S. Annapureddy, M. J. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *NSDI*, 2005.

[11] M. Arlitt and C. Williamson. An analysis of tcp reset behavior on the internet. *ACM CCR*, 35(1), 2005.

[12] K. C. Barr and K. Asanovic. Energy-aware lossless data compression. *IEEE Transactions on Computer Systems*, 24(3):250–291, Aug 2006.

[13] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *USENIX*, 2003.

[14] J. Eriksson, S. Agarwal, P. Bahl, and J. Padhye. Feasibility study of mesh networks for all-wireless offices. In *MobiSys*, 2006.

[15] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *SIGCOMM*, pages 181–194, 1997.

[16] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, 35(5), 2001.

[17] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A first look at modern enterprise traffic. In *IMC*, 2005.

[18] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *NSDI*, 2007.

[19] M. Rabin. Fingerprinting by random polynomials. Technical report, Harvard University, 1981. Technical Report, TR-15-81.

[20] S. C. Rhea, K. Liang, and E. Brewer. Value-Based Web Caching. In *12th World Wide Web Conference*, 2003.

[21] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *SIGCOMM*, pages 87–95, 2000.

[22] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *NSDI*, 2006.