Chunk and Object Level Deduplication for Web Optimization: A Hybrid Approach

Ioannis Papapanagiotou, *Student Member, IEEE,* Robert D. Callaway, *Member, IEEE,* and Michael Devetsikiotis, *Fellow, IEEE*

***Abstract*—Proxy caches or Redundancy Elimination (RE) systems have been used to remove redundant bytes in WAN links. However, they come with some inherited deficiencies. Proxy caches provide less savings than RE systems, and RE systems have limitations related to speed, memory and storage overhead.**

In this paper we advocate the use of a hybrid approach, in which each type of cache acts as a module in a system with shared memory and storage space. A static scheduler precedes the cache modules and determines what types of traffic should be forwarded to which module. We also propose several optimizations for each of the modules, such that the storage and memory overhead are minimized. We evaluate the proposed system by performing a trace driven emulation. Our results indicate that a hybrid system is able to provide better savings than a proxy cache, or a standalone RE system. The hybrid system requires less memory, less disk space and provides a speed-up ratio equal to three compared to an RE system.

***Index Terms*—WAN optimization, Traffic deduplication, Redundancy Elimination, Hybrid Cache**

I. Introduction

The exponential growth of mobile data traffic has led service providers to implement data deduplication systems. Data deduplication can remove repetitive patterns in traffic streams, and decrease response times for time sensitive applications. The most widely implemented technique in wired networks were proxy caches [1]. Although proxy caches remove redundancy at the object level, there is a vast amount of web data that is uncacheable per RFC 2616 [2].

Some recent studies [3], [4], [5] have advocated the benefits of protocol-independent Redundancy Elimination techniques (also called byte caches). They can remove redundancy at the chunk level, which is a much smaller granularity than at the object level. Hence even if an object or a file is partially modified prior to being transferred for a second time, the unchanged parts would still benefit from the optimization. Initially, chunks were identified inside each packet. However, in [6] the authors proposed a WAN optimization system that removes duplicate chunks on top of the TCP layer. TCP based chunks are bigger than packet based chunks, but smaller than objects. The advantage of this approach is that it can identify the redundant bytes even if they span over many packets.

An RE implementation requires the installation of two middleware boxes; one closer to the server (encoder) and one closer to the client (decoder). As data flows from the server to the client, it passes through the boxes and is broken into chunks. The chunks are stored on the persistent storage of each box. For each chunk, a representing fingerprint (hash) that maps to the actual chunk is generated and stored in the memory (e.g. a 1KB stream can be represented by a collision-free 20B hash). The two boxes communicate through an out-of-band TCP connection such that the data are delivered in order. Since both boxes contain the same data, they are *synchronized*. A second reference to a chunk would mean that the encoding box would send the hash value instead of the actual bytes [3].

In RE systems, fingerprinting is performed based on the Rabin fingerprinting algorithm [7]. A sliding window moves byte and byte, generating the fingerprints. Each one of them is compared with a global constant to derive the boundaries of each chunk. However, performing these steps over all of the data may create a bottleneck in higher bandwidth links [8]. Moreover, the hash overhead per object in proxy caches is much smaller than the hash overhead per chunk in RE systems.

In this paper, we propose a hybrid redundancy elimination technique. The proposed system consists of a scheduler, an RE module and a proxy cache module. The decision on which module the byte stream should flow through is determined by the scheduler. The benefits of such an approach can be summarized as follows:

- Fewer hash computations are performed, therefore allowing our hybrid system to be deployed within higher bandwidth links.
- An application layer compression scheme, instead of the standard IP packet-based RE, should lead to better savings and storage overhead.
- A reduction in the memory overhead compared to a standalone RE system. As some byte streams flow through the proxy cache module, they do not need to be broken in chunks and unnecessary hash generation and storage can be avoided.
- The proposed approach can be implemented on the encoding box of a WAN optimization system without significant architectural modifications.

The remainder of our paper is organized as follows: In Section II, we describe the proposed system design. In Section III, we briefly describe the emulation environment and the dataset that has been used to validate the suggested implementation. In Section IV, we showcase results on how effective is the proposed system, under various parameters and criteria. Finally, in Section V we conclude with our remarks.

The authors are with the Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, 27695-7911 USA and with the IBM WebSphere Technology Institute, RTP, NC, USA. Emails:(ipapapa, mdevets)@ncsu.edu and rcallawa@us.ibm.com.

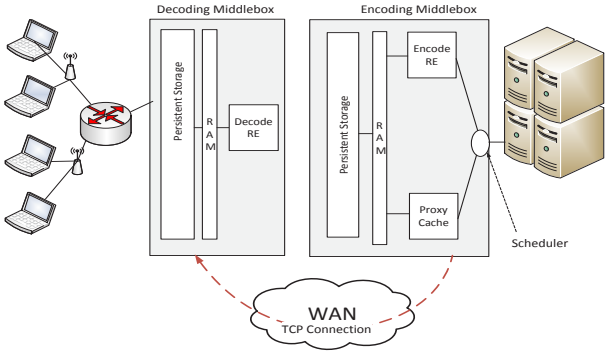

Fig. 1. Proposed Architectural Approach of a Hybrid Cache

II. System Design

We initially assume a standard architectural approach of an RE system [3]. The decoding box is located closer to the access network and the encoding box closer to the server. The encoding box performs fingerprinting of the byte stream, indexing and lookup, and data storage. The task of data reconstruction is performed at the decoding box. The proposed implementation is shown in Fig. 1. In the encoding middleware we assume three separate entities (a) the scheduler, (b) the RE cache (or chunk based cache), and (c) the proxy cache. The memory and the persistent storage are shared among the caching modules.

A. Scheduler

The scheduler precedes the two caching modules, and is responsible to decide whether to forward the flow to the encoding RE or to the proxy cache or to send the data unprocessed. The scheduler operates at multiple layers. It first checks the IP header to identify the protocol. All TCP traffic is passed to the higher layers for further examination.

For every TCP connection that uses destination port 80, we look into the bytes of the TCP buffer to determine whether they contain web data. Once we have performed the application classification, we process the HTTP headers. Note that HTTP headers may span multiple packets or may be delivered in the application layer in different TCP buffer reads. By processing the HTTP request headers the scheduler determines whether the object is cacheable or not, based on RFC 2616 [2]. All cacheable content is directed towards the proxy cache module. All non-cacheable content is directed towards the RE module, since there would be no benefit from flowing through the proxy cache module.

For the remaining TCP and UDP traffic, the scheduler decides if they will flow through the RE module or they will not be processed by the hybrid system. However, unless mentioned specifically, we will assume that the scheduler decides to send the data unprocessed. We follow this approach because, for non-web traffic, the RE module could show additional savings, whereas the proxy cache would not do that. Therefore for fairness in comparison we will focus on web traffic optimization. In the last section we will show further results on other types of traffic.

B. Module Design Principles

1) Proxy Cache Module: In the HTTP proxy cache module a hash value of the object's URL (host name and URI), along with some metadata, is stored in the non-volatile memory. The object's content is stored locally on the persistent storage. When an object is still valid in the proxy cache, it is retrieved from the cache, otherwise it is fetched from the end server. The proposed web proxy module is fully compliant with RFC 2616 [2]. This RFC defines several rules and leaves several others open for implementation. For the latter, we follow the most recent version of Squid [9] with LRU replacement policy. However, there are some cases for which we use an optimized approach compared to Squid.

First, since recent works [10] have indicated that several users tend to partially download an object (e.g. watch a small number of seconds from a video and then jump to another one), we assume a *partial downloading* policy. In other words, the proxy cache only stores the portion of the object that has been already requested by the users (which is the optimum policy in terms of cache overhead and savings). This is different from the default policies in Squid, in which an object is cached only if an $x\%$ of the object has been downloaded (this configurable argument in squid is called *range offset limit*).

Second, in many occasions content is provided in fragments (e.g. smartphone mobile players). The range of each fragment is specified in the HTTP headers, through the *Range Offset* field (this technique is sometimes referred as *adaptive streaming*). The client requests for a specific range, and the server replies with an HTTP *206 Partial Content* response. In these occasions, the URL of the different fragments that correspond to the same file will be the same. While default caching policies would not cache fragmented objects, our proxy cache module is able to parse the *Range Offset* and uniquely identify the objects. Hence, it can cache them and determine redundancy based on each fragment.

Lastly, the proxy cache module incorporates video optimizations (such as uniquely identifying videos even if their URLs contain user-related information), similar to the video cache Squid plugin [11]. For example, accessing Youtube service is done in two phases: 1) content look-up, 2) content download and playback. The first phase is performed on the web browser (or a custom application that embeds Youtube video) by selecting a video uniquely identified by an 11 byte *videoID*. In the second phase, the Youtube video contacts the Youtube CDN to collect a copy of the video object solely identified by a 16 byte *cacheID*. While the PC-player and Mobile-player may have some differences [10], our proxy cache module is able to handle those cases from Youtube and from several other video service providers.

2) RE Cache Module: Our RE system follows the *Chunk-Match* principles from [5] with some major differences. In [5] the RE was performed on a per packet basis. A Rabin fingerprint [7] is generated for every substring of length β. The base of the modular arithmetic, for generating the fingerprints, was set to 2^{60}, such that the collisions among hashes are minimized. When a fingerprint matches a specified constant γ ($hash \bmod \gamma = 0$), the fingerprint constitutes a boundary. In

Constants	
Fingerprint window size	β
Module operator for chunk boundaries	γ
Minimum chunk size	δ
Temporal Parameters	
Size of circular TCP buffer	M
Chunk size	L
Hash size	N

TABLE I
RE CACHE MODULE CONSTANTS AND VARIABLES DEFINITION

our case, the fingerprint generation is performed per web object. One of the main advantages of this approach is the higher upper bounds of compression. Assuming a perfect match, the upper bound of the packet level compression is $1-\beta/MTU$, whereas in our approach it may reach $1-\beta/object\_size$.

In the legacy RE approach [3] determining the boundaries was relatively easy, since packets that contain the same data are usually of the same size. However, in the proposed approach, the RE module receives the byte stream through the TCP socket buffer. In a naive approach, the whole object would be allocated in RAM, and then processed byte-by-byte. However, this has problems related to unnecessary memory allocation (especially for some large objects) and the process of loading the whole object in the socket buffer and then fingerprinting the data is serial.

To tackle this issue we use an application layer circular buffer per TCP connection. For every application call the data is copied from the socket buffer to circular buffer and the fingerprints are computed over this byte stream. Assuming a temporal size M of the circular buffer, the total number of fingerprints per buffer read is $M-\beta+1$. Since the Rabin algorithm determines the fingerprints in a sliding window manner, we are aware of the fingerprints from the start of the circular buffer and up to β bytes before the end. We remove all processed bytes, and keep the last $\beta-1$ bytes in the circular buffer. Once a new byte stream is available for this connection, we allocate it after the $\beta-1$ bytes and perform the fingerprint generation. Hence, the size of the circular buffer per socket connection must be at least the size of the fingerprint window size. For fairness in comparison, we have implemented an LRU policy for the RE module. All hashes are stored in RAM, and the content in the persistent storage.

Finally for the RE system, we assume a minimum chunk size, namely δ. This is because: (a) We want to avoid very small chunk sizes because the overhead is high. (b) Smaller chunks tend to contribute much less than bigger chunks in savings. (c) A smaller δ may decrease the CPU utilization as the modulo operations, for chunk boundary determination, are computed only when $L>\delta$. (d) For security reasons, since an attacker can flood the RE cache with very small or very big chunks[1]. The proper selection for the value of δ is shown in section IV.

[1]The Rabin fingerprint determines the boundaries based on the content, any long sequence of *00* would potentially generate a chunk boundary. Another potential attack is when the intruder is aware of the γ; then he can generate unnecessary fingerprints.

C. Memory Overhead

The memory overhead is the amount of bytes that need to be stored in memory. Those bytes are related to the hashes that point to the disk location, where the actual content is stored.

A hash of the URL along with some metadata are stored in memory, and the object itself on the persistent storage. The index entry in Squid is 80B. An optimized indexing methodology [12] would not change the overhead considerably.

For the RE system, the chunks are stored on the persistent storage. For every chunk the representing hash value is stored in memory. Let us assume that all chunks are very small, e.g., $L=8B$. A hash value of size $N=4B$ would produce 4.2 billion unique entries for 8B chunks, viz. 24TB of data on the HDD. The 4B hashes are stored in a doubly linked LRU list, for which a 2x3B virtual memory pointers (forward and backward connection pointers) would suffice for an 134GB hash memory space. The chunk size, which is also stored in memory, is 2B; the log generation output is 1B and the disk number 1B (in case a disk array is implemented). The total overhead per chunk is therefore 14B.

D. Transmission Overhead

The transmission overhead is the amount of bytes that will be sent instead of the actual data. It is expressed in terms of percentages of the actual data. For example, if each 60B chunk/object is encoded with a unique 6B hash, the transmission overhead is 10% (or N/L). Therefore, we define:

- **Savings** *as the amount of bytes the provider would not transmit due to the implementation of a redundancy elimination technique.*
- **Redundancy** *is the amount of bytes that a technology would determine as redundant.*

Note that the difference between savings and redundancy is the transmission overhead.

For the proxy cache, once an object is found in the cache and is fresh, it does not have be downloaded from the server. As described above, the implemented partial downloading policy does not add any extra overhead.

In the RE module, there are several cases in which the redundancy spans over more than one chunks. In our implementation, the encoding cache sends the location of the chunk in memory (4B are enough to map a memory of 4TB) and also 2B that indicate the maximum matching region. Thus, the transmission overhead has a lower bound of 6B per chunk.

III. IMPLEMENTATION AND DATA SET

To understand the tradeoffs of each type of cache (standalone web cache, standalone chunk based cache, hybrid cache) we have developed an emulator for each one. The emulators read packet-level traces using the *Libpcap* library [13] and perform TCP reassembly using the *Libnids* library [14], which emulates a Linux Kernel 2.0.x IP stack. Object reconstruction and each of the caching techniques are developed as separate libraries in C.

We captured two wireless packet traces in an aggregation router of an educational institution, one in a low bandwidth link (Trace A) and one in a high bandwidth link (Trace B). The details of the traces are shown in Table II. We did not

	Trace A	Trace B
Length	11am-2pm (3h)	7am-2pm (7h)
Dates	11 Jan 2011	10 April 2011
Size (GB)	19GB	64GB
Packets (Mil)	36	104
Unique IPs	651	1103

TABLE II
TRACE FILE DETAILS

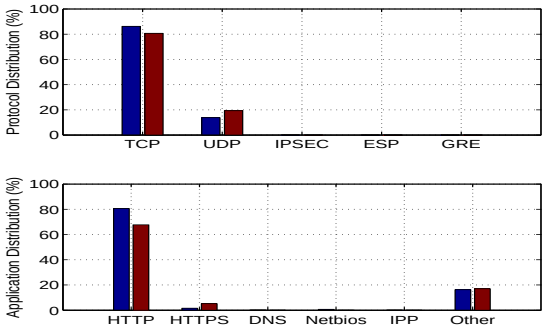

Fig. 2. Protocol and application distribution of bytes in the traces

process the TCP connections, if only one half-stream appears in the traces. This is important because one stream may contain the HTTP request and the other the HTTP response. In Fig. 2, we showcase the protocol and application distribution in the captured traces. We may observe that the majority of the generated traffic is TCP and a smaller portion of UDP traffic. From the application analysis, we may see that HTTP is the dominant application, with a small portion of HTTPS, and 15% of other types of traffic. As we will show in the following section byte caching other types of traffic may provide additional savings.

IV. RESULTS

A. Optimal Parameter Selection

The three main parameters of the RE module are the fingerprint size β, the average chunk size (which is a derivative of the modulo parameter γ) and the minimum chunk size δ. We have performed multiple runs of the emulator to determine the optimal combination of those parameters. We used the biggest trace (Trace B), and modified the scheduler to send all data to the RE module (contrary to the default configuration of sending only the uncacheable content), such that the highest possible amount of data flow through the RE module.

We estimated the savings and memory overhead using a Response Surface Methodology (RSM) [15]. We varied the three parameters from $8 - 64B$ with increments in powers of 2. $f_1(\beta, \gamma, \delta)$ represents the 3-dimensional response for the savings and $f_2(\beta, \gamma, \delta)$ the corresponding one for the memory overhead. Their responses are shown on Fig. 3. Note that in the graphs, the red values (higher) are better for the savings and the blue values (lower) are better for the memory overhead. We can observe that the vector $\{\beta, \gamma, \delta\} = \{8, 32, 32\}$ provides the best savings, and the smallest memory overhead. A $\gamma = 32B$ would produce chunk size of average length around 64B. For the rest of the results we are going to use these values. The best savings for web data using an RE module is 21%

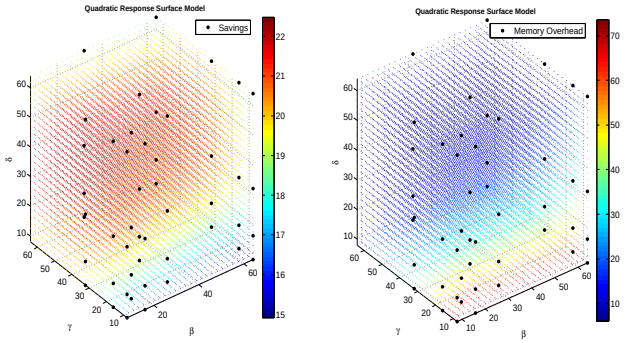

Fig. 3. Savings and memory overhead as a function of the input parameters.

and this is attained with a memory overhead of 22%. In other words, for every 1TB of data in the disk, 220MB needs to be stored in memory.

The parameters of the statistical regression are shown below and have been estimated with a squared coefficient of multiple correlations $R_1^2 = 0.83$ and $R_2^2 = 0.92$.

$$f_1 = 12.7 - .01\beta + .217\gamma + .273\delta - .001\gamma\delta - .002\gamma^2 - .003\delta^2$$
$$f_2 = 103 - 0.03\beta - 2.04\gamma - 2.05\delta + .016\gamma\delta + 0.01\gamma^2 + 0.01\delta^2$$

For presentation purposes, we have omitted the combinations that had minimal contribution (second most significant digit). The parameters of the polynomials provide an indication of the effects of each of the system parameters to the savings and the memory overhead. Indicatively, the fingerprint size β has a smaller contribution compared to the other parameters, and the minimum chunk size δ has the highest contribution.

B. The Hybrid Approach

In this subsection the scheduler forwards the non-cacheable content to the RE module and the cacheable to the proxy caching module. This was derived from the observation that this non-cacheable content can be more than 35% of the web content. For example, we found that the non-cacheable content is 49% for trace A and 37% for trace B.

In Fig. 4, we showcase the performance of a proxy standalone system, an RE standalone system and the hybrid system that we propose. Obviously the savings from an RE standalone system would be higher than a proxy cache system. In our traces we find that the RE provides 33% higher savings compared to a proxy cache. However, the most interesting result is that a hybrid system provides higher savings than a standalone RE. Given that the RE has higher redundancy compared to the hybrid system in all traces (figure 5), we may conclude that the higher savings for the hybrid system is an artifact of the lower transmission overhead to encode and send the chunks to the decoding side.

An additional advantage of the hybrid system is shown in the memory overhead graph of figure 6. We can see that the memory overhead for the RE system ranges from $15 - 22\%$. The hybrid system though requires half or one third of the memory compared to a standalone RE implementation.

The final micro-benchmark that we performed is based on the bandwidth that the system can support, or else the amount

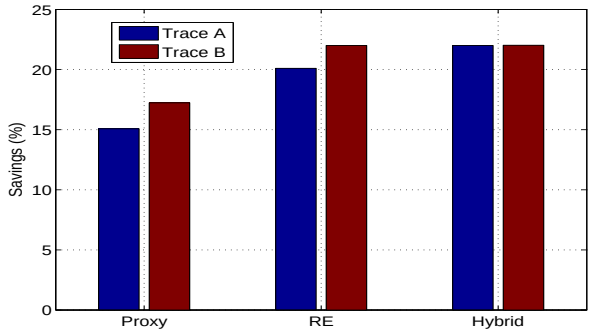

Fig. 4. Savings for proxy cache standalone, RE standalone and hybrid system.

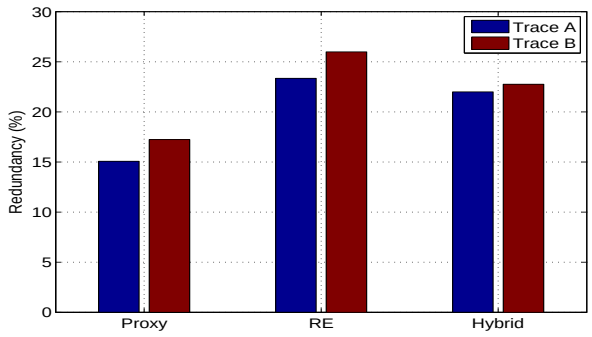

Fig. 5. Redundancy for proxy cache standalone, RE standalone and hybrid system.

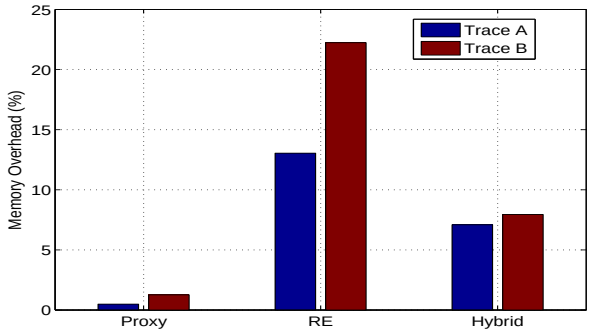

Fig. 6. Memory requirement to represent data for proxy cache standalone, RE standalone and hybrid system.

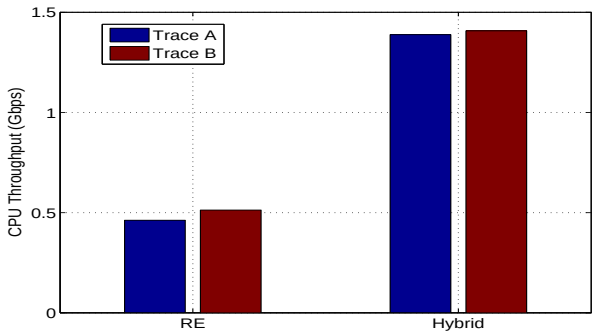

Fig. 7. CPU throughput for an RE standalone and the hybrid system.

of time that the RE module requires to process the web data. We used a 16-core Intel Xeon X5560 with CPU 2.8GHz and 16GB of RAM on a 64bit Linux OS. By using the GNU profiler *gprof* [16], we isolated the cumulative time per trace to perform the chunk based RE functions. These functions are related to hash generation and fingerprinting. We observed that these functions consume almost 95% of the CPU time, and the majority of the time was spent for generating the Rabin fingerprints ($\sim$2.31 msec/call).

By dividing the bytes that would flow from each module with the cumulative time spend to process the data, we could derive the bandwidth of the system. Figure 7 indicates that hybrid approach can support up to 1.5 Gbps, viz. a 3x speedup compared to an RE only module. This speedup comes from the fact that the RE module in the hybrid case needs to process much less traffic (only the non-cacheable portion), whereas the RE standalone cache needs to process all web data. Since a hybrid cache can have higher or close to the RE standalone savings such a speedup is highly beneficial. Note that the hash computation for the proxy cache is minimal, thus it is not shown on Fig. 7.

Finally, we need to point that this speedup does not include the disk I/O operations. Since newer disks can support close to Gbps of throughput, an RE only system would potentially create a CPU bottleneck, which the hybrid cache resolves.

C. Finite Sized Cache System

In this subsection we used our biggest trace (Trace B). We performed two emulations of the hybrid system, one with a disk capacity of 1GB and another one with 10GB. The two emulations are plotted in Fig. 8a and 8b. Note that the x-axis represents the amount of bytes allocated to each module, in the format $\{$RE module size - Proxy module size$\}$. Those two figures depict two things: (A) They showcase the percentage of disk space allocation for each functionality in order to get the optimal savings. This is important as the memory and disk space are shared entities. (B) They present the difference in the savings between a proxy cache standalone system and a hybrid system (red and blue line). This is because the savings of the proxy cache module do not change when implemented as part of the hybrid system.

Therefore, from Fig. 8a, we may observe that the maximum attainable savings, for the 1GB hybrid system, are 19%. Whereas for the 10GB hybrid system the maximum attainable savings are 22%. Since an infinite sized cache[2] provides savings close to 22%, we may conclude the total storage space of a hybrid system needs to be 10GB. Fig. 8a and 8b present similar savings for the RE module. This indicates that the RE module has diminishing returns only after 400MB have been used. Given that the non-cacheable content in the trace is much bigger, we may conclude that a 400MB allocated to the RE module would be enough to provide the optimal savings. Apparently the decrease in the savings for the hybrid system with size 1GB, compared to the one with 10GB space, is an artifact of the proxy cache. Hence the proxy cache module

[2]Infinite size cache is a cache whose storage space is higher than the size of the trace, therefore no content replacing is taking place.

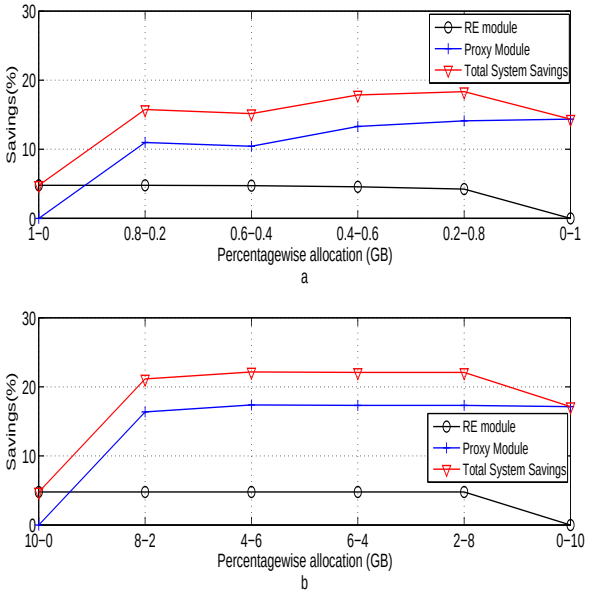

Fig. 8. Disk Capacity Allocation for each module for an 1GB (a) and 10GB (b) hybrid system (RE module size - Proxy module size)

needs to be at least 10GB to get the maximum savings. This shows that the RE module requires only 4% of additional storage space to provide an increase of 28% in the savings.

We also modified the scheduler to send all web data to the RE module (emulating again an RE standalone system). We observed a difference of 3% between a 1GB and 10GB disk storage. This indicates that the reason that only 400MB of RE module are required in the hybrid system is because the non-cacheable content is not as repetitive as the cacheable content. Non-cacheable content tends to include user specific information, which may not be the same across different users.

D. Addition Savings and Future Work

In Section II-B2 we proposed an RE module with a single circular application level buffer. Nonetheless, we have observed that $16\% - 20\%$ of the traffic is UDP. Therefore, we implemented an additional static buffering approach with a capacity equal to the maximum MTU (at least 9KB to fit Jumbo frames). The scheduler was modified to forward UDP traffic to this static sized buffer, such that the RE operations can be performed on a per packet basis in this buffer. Effectively, the modified system is able to perform the RE module functionalities on a per TCP connection basis and on a per packet basis. Our results indicate that a dual-buffered system will add 2% in savings for trace A and 2.6% in trace B. The difference in the savings between the hybrid cache and the proxy cache would now increase to 10%.

Finally, we need to note that such an additional application buffering may potentially create other issues, which require further investigation, e.g. processing encrypted/SSL traffic. We plan to extend our emulator to a prototype and identify further challenges.

V. Conclusion

In this work we proposed and emulated a redundancy elimination system that uses both object and chunk based deduplication techniques. Each of those techniques, if run independently, have significant drawbacks. For example, a proxy cache does not process uncacheable content and determines redundancy at the object level; therefore it offers limited savings. The RE cache determines redundancy at a smaller granularity, but has resource limitations such as storage overhead and CPU processing.

The proposed *hybrid* system incorporates both those techniques into an in-the-box solution with a static scheduler. The scheduler forwards uncacheable content to the RE module, which performs chunk based caching on a per object basis. The proxy cache module is able to handle partial content and optimized video service delivery related to smartphone devices. We showcase that the system provides two times more savings than a standalone proxy cache, and better savings than a standalone RE system. It requires three times less memory storage, and provides a speedup equal to 3 compared to an RE only approach.

References

[1] P. Barford, A. Bestavros, A. Bradley, and M. Crovella, "Changes in Web client access patterns: Characteristics and caching implications," *World Wide Web*, vol. 2, no. 1, pp. 15–28, 1999.

[2] "RFC 2616 hypertext transfer protocol – HTTP/1.1," http://www.ietf.org/rfc/rfc2616.txt, June 1999.

[3] N. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *SIGCOMM*. ACM, 2000.

[4] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee, "Redundancy in network traffic: Findings and implications," in *ACM SIGMETRICS/IFIP PERFORMANCE*. ACM, 2009.

[5] B. Aggarwal, 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 *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010.

[6] S. Ihm, K. Park, and V. Pai, "Wide-area network acceleration for the developing world," in *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, Jun. 2010.

[7] M. Rabin, *Fingerprinting by Random Polynomials*. Center for Research in Computing Tech., Aiken Computation Laboratory, Harvard Univ., 1981.

[8] M. Martynov, "Experimental study of protocol-independent redundancy elimination algorithms," in *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. ACM, 2010.

[9] The Squid Project, "Web proxy caching," www.squid-cache.org.

[10] A. Finamore, M. Mellia, M. Munafo, R. Torres, and S. Rao, "Youtube everywhere: Impact of device and infrastructure synergies on user experience," *Purdue Research Report*, 2011.

[11] "Video cache squid plugin," cachevideos.com/.

[12] A. Badam, K. Park, V. Pai, and L. Peterson, "Hashcache: Cache storage for the next billion," in *6th Network Systems Design and Implementation (NSDI)*. USENIX Association, 2009, pp. 123–136.

[13] "Libpcap, packet capture library," www.tcpdump.org.

[14] "Libnids: E-component of network intrusion detection system," libnids.sourceforge.net.

[15] G. Box and N. Draper, *Response surfaces, mixtures, and ridge analyses*. Wiley-Interscience, 2007, vol. 527.

[16] J. Fenlason and R. Stallman, "The GNU profiler." [Online]. Available: http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html