This paper was presented as part of the 14th IEEE Global Internet Symposium (GI) 2011 at IEEE INFOCOM 2011

CombiHeader: Minimizing the Number of Shim Headers in Redundancy Elimination Systems

Sumanta Saha, Andrey Lukyanenko, Antti Ylä-Jääski
Aalto University, School of Science, Finland

***Abstract*—Redundancy elimination has been used in many places to improve network performance. The algorithms for doing this typically split data into chunks, fingerprint them, and compare the fingerprint with cache to identify similar chunks. Then these chunks are removed from the data and headers are inserted instead of them. However, this approach presents us with two crucial shortcomings. Depending on the size of chunks, either many headers need to be inserted, or probability of missing similar regions is increased. Algorithms that try to overcome missed similarity detection by expanding chunk boundary suffers from excessive memory access due to byte-by-byte comparison. This situation leads us to propose a novel algorithm, CombiHeader, that allows near maximum similarity detection using smaller chunks sizes while using chunk aggregation technique to transmit very few headers with few memory accesses. CombiHeader uses a specialized directed graph to track and merge adjacent popular chunks. By generating different generations of CombiNodes, CombiHeader can detect different lengths of similarity region, and uses the smallest number of headers possible.**

Experiments show that CombiHeader uses less than 25% headers than general elimination algorithms, and this number improves with the number of hits. The required memory access to detect maximal similarity region is in the range of 1%-5% of comparable algorithms for certain situations. CombiHeader is implemented as a pluggable module, which can be used with any existing redundancy elimination algorithm.

KEYWORDS: redundancy elimination, RE, optimization

I. Introduction

From commercial products [1–3] to academic research works [4–6], redundancy elimination (RE) in network traffic at a level lower than conventional object-level caches has gained a lot of attention in recent years. Application independent RE allows to detect similarity within the network traffic at a much granular level than that of conventional object level caches. Thus, it opens a window of opportunity to detect intrinsic similarity between two completely different objects. While mostly used in localized single point gateways such as WAN access points, recent works [6–8] have shown that it has wider scope of use in different scenarios and deployments of network. This discovery has triggered the invention of several new algorithms for redundancy elimination at different parts of the network.

A typical RE algorithm inspects the contents of an IP packet and generates content dependent chunks from it. This chunking is done mostly using the famous Rabin fingerprinting [9] method. These chunks are then fingerprinted for easy searching and retrieval. At a later time, the incoming packets are also fingerprinted and checked for containment in the existing cache. If some parts of the new packets are found in the cache, they are stripped off and a shim header is inserted in place. These stripped packet are then transported as far as possible before finally reconstructing them at a downstream network element with the help of the shim header.

One very important parameter in any RE algorithm is the average chunk size. While smaller chunk sizes allow better similarity detection, bigger chunk sizes allow inserting fewer shim headers in the outgoing packet resulting in better compression rate. Most of the previous work [6, 7, 10] just use a small chunk size to report high similarity detection. However, they fail to mention the increasing number of headers to transmit, and the added complexity in the operation. While the importance of choosing the right chunk size is well understood and analyzed in [11] and [8], no work so far has proposed any solution to combine the advantages of using small chunk size (better similarity detection) with that of bigger ones (better compression rate). Works in [11, 12] offer multiresolution chunks, however, do not allow adaptive change of chunk sizes. Similar works were done for content syncing [13, 14], which operate only on offline disk files and does not apply to real time traffic. In this paper, we propose an algorithm, CombiHeader, which allows any RE algorithm to use smaller chunk sizes and thus detect maximal similarity while transmitting much fewer shim headers to downstream elements by adapting the chunk size in real time based on popularity index.

Most of the current solutions either uses a fixed size chunk—inserting many shim headers to the outgoing packet, or uses a byte-by-byte comparison to maximize the matching region—generating numerous memory accesses. On the other hand, CombiHeader utilizes a highly optimized data structure to adaptively decide on the chunk size. For traffic flows with around 70% similarity, CombiHeader reduces shim headers upto 80% compared to general RE solutions, thus reducing the encoding and decoding processing. CombiHeader also uses very few memory accesses to enlarge the chunks to its maximum matching size. While a byte-by-byte comparison to expand a 1024-byte chunk to a 1374-byte similarity region takes around 351 memory access and one shim header, CombiHeader running with 256-byte chunk size requires only nine memory accesses and one shim header, in the best case. This is around 2% of the actual number of accesses. CombiHeader also provides an efficient parameter to tune and control the memory usage of the algorithm.

978-1-4244-9920-5/11/$26.00 ©2011 IEEE

II. Background and Motivation

With the goal of improving the achievable compression rate of an RE algorithm, we will discuss in slightly more detail the present redundancy elimination techniques and the motivation behind the development of CombiHeader.

A. Chunk Size

As discussed briefly in §I, the idea behind traffic RE techniques is to remove repeated arrays of bytes from a network stream at a much finer level than objects or even packets. The superiority of packet RE over traditional object caches stems from the fact that RE is mostly application unaware and can eliminate redundancy irrespective of which application the flow is a part of. One of the most important parameters of any RE algorithms is the average chunk size. This size is decided by tuning a parameter in the Rabin fingerprinting algorithm [9]. The compression rate of a RE algorithm heavily depend on this parameter: the bigger the chunk size is, the fewer shim headers the algorithm has to transmit. However, having bigger chunk size results is worse similarity detection, diminishing the fewer-header advantage.

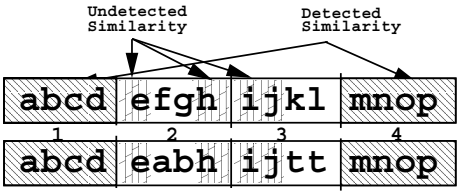

Fig. 1: Matching regions that cannot be detected by chunk-level RE, but can be detected by maximal-match RE with many memory accesses

B. Chunk Area Expansion

The matching of incoming traffic with the chunk store is typically done in a chunk level. However, matching regions can extend beyond the chunk boundary. To detect this similarity, two different approaches has been taken thus far: 1. Chunk level RE and 2. Maximal Match RE. In Chunk level RE, comparison is done between representative fingerprints of whole chunks, which causes partial chunk matches not to be detected (similar regions in chunk pairs 2 and 3 are not detected in Figure 1). This problem renders the case of bigger chunks less useful, which, according to §II-A, was the better choice. On the other hand, Maximal Match RE performs rigorous byte-by-byte matching to extend a matching chunk to its maximum. This allows maximum matching with the cost of more memory accesses.

C. Performance Parameters

While maximal RE seems to be a good idea to detect as much matching payload as possible, it requires many memory accesses and creates a bottleneck in high speed network elements. On the other hand, chunk level RE, which only depends on fingerprint comparison, is free from this flaw but requires a large number of shim headers to be inserted. Moreover, it can only detect chunk level similarity and the detection granularity suffers increasingly as bigger chunk sizes are used. For example, in Figure 1, two series of chunks are compared. With chunk level RE, we can detect similar chunks pairs 1 and 2 with only two memory comparisons. However, partially matching chunk pairs 2 and 3 remain undetected. Maximal match RE, on the other hand, is able to at least detect the similarity in chunk 2 which is adjacent to chunk 1 with the expense of extra memory accesses.

If the chunk size is ω, matching regions are X_i, number of matching regions m, total number of chunks is κ, and total matching chunks are n:

Lemma II.1. *Number of memory access for maximal match RE, μ_{mm}, is proportional to that of chunk level RE, μ_{cl} with ω^2 multiplier.*

Proof: Lower bound for μ_{mm}, when each chunk grows only to the right, and partial matches within chunks ($X_i \bmod \omega \mid i = 1 \ldots m$) are uniformly distributed in $[0, (\omega - 1)]$, then, $\mu_{mm} = \kappa + \sum_{i=1}^{m}(X_i \bmod \omega + 1) = \frac{\omega+1}{2} \cdot m + \kappa \geq \frac{\alpha}{2}(\omega+1) \cdot \omega \cdot \kappa + \kappa$, where $0 < \alpha < 1$. While memory access required for chunk level RE, $\mu_{cl} = \kappa$. Therefore, $\frac{\mu_{mm}}{\mu_{cl}} = 1 + \frac{\alpha}{2}\omega + \frac{\alpha}{2}\omega^2$. Thus μ_{mm} is proportional to μ_{cl} with ω^2 multiplier, as stated. ■

Because of Lemma II.1, we will exclude maximal match from further discussion. Chunk level RE, while outperforming maximal match in memory access, misses all the regions of data which does not belong to a matching chunk, but belongs to a matching region. Number of missed matching bytes:

$$\Delta_u = \sum_{i=1}^{m} X_i \bmod \omega \tag{1}$$

Another parameter, number of transmitted shim headers τ_h, depends on the selected ω. Smaller ω allows better similarity detection, but causes larger τ_h. This τ_h can be expressed as:

$$\tau_h = n \tag{2}$$

Lemma II.2. *Δ_u and τ_h are inversely related.*

Proof: From Equation 1 and 2, $\tau_h = n = \sum_{i=1}^{m} X_i \div \omega = \sum_{i=1}^{m} \frac{X_i - X_i \bmod \omega}{\omega} = \frac{1}{\omega}(\sum_{i=1}^{m} X_i - (\sum_{i=1}^{m} X_i \bmod \omega)) = \frac{1}{\omega}(\sum_{i=1}^{m} X_i - \Delta_u)$, and the lemma follows. ■

The target of any RE algorithm should be to reduce all the three parameters (μ_{cl}, Δ_u and τ_h) simultaneously–which is non-trivial because of the inverse relationship of Δ_u with τ_h and μ_{cl} (Lemma II.1 and II.2). CombiHeader aims to achieve this target in an RE-algorithm independent way, i.e., a pluggable way which can be used with any RE algorithm.

III. CombiHeader

From the discussion above, it is apparent that a solution that can remove as much similar content from a packet as possible without having to insert too many chunk headers and without slowing down the operation with many memory accesses is needed for an efficient RE system. This is where CombiHeader comes into picture. CombiHeader employs an efficient algorithm to track consecutive popular chunks and

merges them together without losing the ability to detect basic chunk level hits.

The idea of CombiHeader is based on the presumption of spatial locality. We assume that, if two consecutive chunks are highly popular, then the probability of them always appearing together is higher. Therefore, if we can keep track of adjacent popular chunks and their probability of appearing together, it is possible to merge them together and transmit only one header for more than one chunk. However, while merging smaller chunks together to generate bigger chunks sounds exciting with the promise of transmitting less shim headers, with merging we lose the possibility of detecting smaller chunks.

Algorithm 1 Pseudocode of CombiHeader algorithm

```
if current chunk is cache miss then
    if miss streak = 1 then
        Update last seen Combi and Elementary node
        Create new complex CombiNode if link hit > θ
    else
        Clear last CombiNode and transmit it
        Update last seen elementary node
    end if
    Transmit full text of current chunk
else
    if miss streak = 1 then
        Start fresh, clear last seen CombiNode
        Do not transmit anything
        Buffer current chunk as last seen elementary
        miss streak← 0
    else
        Update last seen elementary node
        Create new CombiNode with (Last CombiNode, current Node) if link hit > θ
        if Try failed then
            Transmit and clear last CombiNode
        else
            Do not transmit anything
            Complex CombiNode is accumulating chunks to transmit
        end if
    end if
end if
```

CombiHeader employs a specialized data structure to tackle this problem. With this data structure, both the elementary and the merged chunk fingerprints are stored in such a way that it is possible to gradually pile up chunk hits to a maximal region and then transmit in a burst. For each adjacent chunk pair, the algorithm maintains a link, and updates the link hit count whenever those two chunks appear one after another in a stream of data. Further, based on a threshold (θ) of link hits, a new merged chunk (CombiNode) is generated from the two elementary chunks, and is attached to the link. After creating the CombiNode based on hitcount of the elementary nodes, each time one of the constituent elementary nodes are hit, no header is inserted in lieu of that, rather the algorithm waits for the next chunk and matches it up with the linked elementary fingerprint. In case of a match, it inserts only the CombiHeader (Header created from a CombiNode) that has been created earlier instead of two elementary headers. The algorithm, thus, works in two phases: 1. CombiNode Creation, and 2. CombiHeader Insertion. To better understand these two operations, an example scenario is described with the help of a pseudo-code description and a figure.

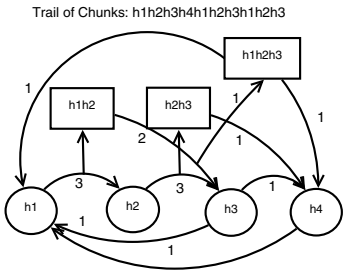

Fig. 2: Creation of CombiNodes based on a trail of chunks

CombiNode Creation: The pseudo code of Algorithm 1 and the simple demonstration of how CombiNodes are created for a sample stream of chunks is depicted in Figure 2. If we follow Figure 2 according to the pseudo code, the following operations occur (The threshold $\theta = 1$):

- When chunk $h2$ arrives after chunk $h1$, a link is established between them to indicate the adjacency. The hit count of the link is set to 1. No CombiNode will be created for this link because the hit count is not over θ.
- $h3$ arrives and a link with $h2$ is created similar to the previous operation. The case for $h4$ is similar to $h3$.
- The chunk $h1$ arrives again and this is a cache hit. So, no new node is created, just a new link from $h4$ to $h1$ is established with a hit count of 1.
- Next comes $h2$ and the already existing link between $h1h2$'s hitcount is updated to 2. Now that the hitcount is greater than θ, a new CombiNode is created for the link which contains the fingerprint of the combined chunk $h1h2$. All the children of $h2$ (e.g. $h3$) are added as children of $h1h2$. Similar operations occur when $h3$ arrives and a new CombiNode $h2h3$ is created. The CombiNode $h1h2h3$ is not created now, as the hitcount between $h1h2$ and $h3$ is not over 1.
- Next comes $h1$, and just a new link between $h3$ and $h1$ is established.
- After $h1$, arrives $h2$ and $h3$, which along with increasing the hitcount of links $h1 \rightarrow h2$ and $h2 \rightarrow h3$, also increases hitcount of $h1h2 \rightarrow h3$, and triggers the creation of $h1h2h3$—a second generation CombiNode.

CombiNode Insertion: From Algorithm 1 we can notice that the algorithm keeps track of the last elementary node and the last CombiNode it has created. Based on this knowledge and on whether the current chunk is a cache hit or miss, CombiHeader algorithm decides what to insert into the packet scheduled for transmission. For an example trail similar to Figure 2, let's take a look at how the algorithm generates packets to transmit.

```
Trail: h1h2h3h4h1h2h3h1h2h3h5
Last elementary: h1    h2    h3    h4    h1    h2    h3    h1    h2    h3    h5
Last CombiNode : -     -     -     -     -     h1h2  -     -     h1h2  h1h2h3 -
Cache hit/miss : M     M     M     M     H     H     H     H     H     H     M
Insert in trans: F(h1) F(h2) F(h3) F(h4) -     -     h1h2  h3    -     -     h1h2h3
                                                                             & F(h5)

   F(x) = Full payload for chunk x
   x    = Elementary header for chunk x
   xy   = CombiHeader for combined chunks x and y
```

Following the above with the help of the pseudo-code should be self-explanatory. For the first instances of $h1$, $h2$, $h3$, and $h4$ we are in a miss streak and we insert the full text

always in the transmit packet. After that, when chunks $h1$ and $h2$ are received again, nothing is transmitted as the algorithm enters in a hit streak and starts to search for CombiNodes in the CombiGraph. This buffering stops once the algorithm cannot find any ready made CombiNodes in the graph. So, when $h3$ arrives, the algorithm cannot find $h1h2h3$ in the graph, and it then inserts the last seen CombiNode, e.g., $h1h2$ in the transmit buffer, and clears last seen CombiNode to indicate that nothing is anymore buffered for transmission. However, $h3$ itself if not transmitted yet, it's buffered to look for further aggregation possibility with the next chunks. When $h1$ appears next, $h3$ is immediately released, as there is no CombiNode opportunity. However, with the incoming $h2$ and $h3$, a new CombiNode $h1h2h3$ is now created, and is then transmitted when a cache miss with $h5$ occurs. Full text of $h5$ is also transmitted with that.

IV. Theoretical Analysis

From the discussion of §II-B, it is evident that the two parameters where CombiHeader algorithm needs to improve is memory Access, and the number of headers transmitted. We will analyze a simple case here to generate a model of various aspects of the algorithm. CombiNodes are generated in layers, and depend on θ. In this case, we assume a stretch of chunks $c_1, c_2, \ldots, c_l$, which are repeated exactly t times, separated by unique symbols. We express the maximum depth of CombiNode layer as δ, total number of chunks as l, total number of nodes in the CombiGraph as Ω, and total headers transmitted as H.

TABLE I: CombiHeader Theoretical Parameters

Layer	Layer generation constraint	Number of nodes	Tx. Headers
1	$0 < i \leq \theta$	l	$l \cdot \theta$
2	$\theta < i \leq 2\theta$	$l-1$	$\frac{l}{2} \cdot \theta$
3	$2\theta < i \leq 3\theta$	$l-2$	$\frac{l}{3} \cdot \theta$
…	…	…	…

Table I lists the restrictions on CombiNode layer generation as per the algorithm, where i is the repetition count of the chunks (details excluded for brevity). Therefore, after t repetitions, maximum depth should be: $\frac{t}{\theta}$. However, if total number of chunks is less than $\frac{t}{\theta}$, then the generation ends prematurely. So the final value is:

$$\delta \approx min(\frac{t}{\theta}, l) \tag{3}$$

The total number of nodes created in the graph depends on the maximum depth, and nodes created on each layer is listed on Table I. Therefore, total number of nodes is:

$$\Omega = \sum_{i=1}^{\delta}(l - i + 1) = \delta(l+1) - \frac{\delta(\delta+1)}{2} \tag{4}$$

To get the number of headers transmitted for a particular round of repetition, we have to analyze the algorithm. For the first θ repetition, only the first layer is generated, and l headers are inserted. For the next θ repetitions, second layer is generated, and $\frac{l}{2}$ headers are transmitted. This pattern continues as shown in Table I. Therefore, total number of headers transmitted after t repetitions are:

$$H_c = \sum_{i=1}^{\delta}(\theta \cdot \frac{l}{i}) = \theta l \sum_{i=1}^{\delta}(\frac{l}{i}) \rightarrow \theta l(\ln \delta + c) \tag{5}$$

where c is a constant. As any other general RE algorithms have to transmit at least $H_r = l \cdot t$ headers for the same stretch of chunks, the ratio is [from (5)]:

$$\frac{H_c}{H_r} = \frac{\theta(\ln \delta + c)}{t} = \frac{\theta(\ln(min(\frac{t}{\theta}, l)) + c)}{t} \tag{6}$$

which gets really small as t increases.

V. Implementation

The implementation of the solution is divided into two major parts: the chunking or RE engine, and the CombiHeader algorithm.

The chunking engine is used for generating chunks out of IP packet payloads. The engine is implemented in pure C, and uses the well known Rabin fingerprinting [9] to detect content dependent hooks in the payload of each IP packet that goes through it. We exclude a detailed description of the chunking algorithm for brevity, however, interested readers can always refer to previous works [5, 10] which discuss about content dependent chunking in general. A configurable parameter in the engine decides whether chunking is done in IP layer or in TCP layer. Another parameter of the engine determines the average chunk size. For calculating the fingerprint we have used SHA-1 hashing algorithm. As for the length of the hash value, its tunable and depends of the chunksize used. The reader is requested to refer to [11] for a detailed analysis of the optimal fingerprint size for any chunksize.

The CombiHeader algorithm is implemented as a directed graph where each node contains a pointer to a corresponding entry in the chunkstore. The sole purpose of the node graph is to keep track of the sequence of chunks passed through the gateway. Additionally, the node graph is used for generating and keeping track of merged nodes. The node graph heavily uses pointers to access memory at random and is optimized for low memory access with controlled memory usage. For generated CombiNodes, the chunkstore does not store the payload. Rather it stores references to the elementary nodes that were used to create the CombiNode. As any CombiNode can be a result of unforeseen number of elementary chunks, storing references to all those elementary chunks would make the payload arbitrarily large, thus creating problem in memory management. Therefore, each entry for CombiNode in the chunkstore have only two references to another CombiNode or elementary Node. It's then traversed recursively to resolve the whole payload of the CombiNode.

VI. Evaluation

In this section, the proposed algorithm is evaluated on the light of the targets specified in previous sections. The

evaluation primarily focuses on three major areas: how CombiHeader algorithm influences redundancy elimination process with different chunk sizes, the effect of primary tunable parameters on the result, and the cost of the algorithm in terms of memory access.

Test Setup: As is already shown in previous works [6, 15], the Internet traffic has enough redundancy that could be removed by applying redundancy elimination techniques. This paper does not try to reinforce that premise again. Rather, we show how the CombiHeader algorithm affects the outcome of any RE algorithm with representative traffic. For most of our experiments, a set of five video files with same video track and different audio tracks is used. This was an informed choice because the audio track is interleaved within video frames and thus presents with a scenario where similar content are interleaved with dissimilarities. We transmit the files one after another through the RE system that we developed and measure various parameters of it.

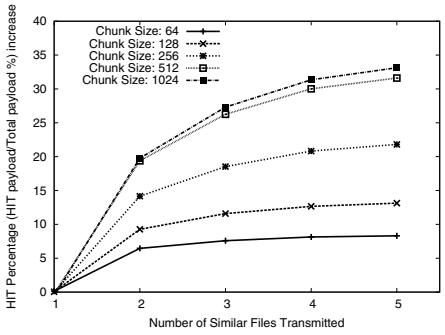

Fig. 3: Increment in hit percentage for flow based chunking compared to packet based chunking

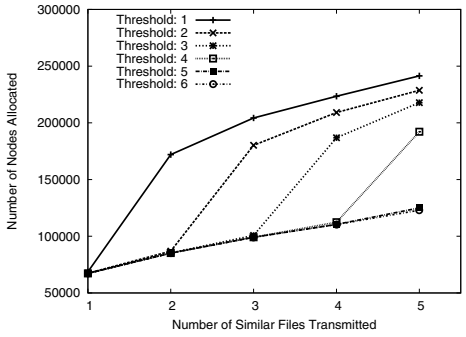

Fig. 4: Effect of θ parameter over memory usage

Packet-level vs. Flow-level chunking: While performing content based chunking over network traffic, one decision every RE algorithm needs to take is whether to use packet level or flow level chunking. In packet level chunking, the payload of each packet is treated separately and no flow-tracking is applied, while in flow level chunking, each flow is tracked individually to reconstruct the actual transport level payload. While flow level chunking requires extra processing power and memory for flow tracking, it is much more efficient in detecting similarity between two flows. In packet level chunking each packet has a leftover chunk which does not meet the chunking criterion. However, in flow level chunking we have only one leftover at the end of the flow.

This observation is backed up by Figure 3 where percentage gain for chunk hit is depicted for different chunk sizes. As can be seen from the figure that there is always a positive gain for flow based chunking, and additionally, the increase is more for bigger chunk sizes. This proves the "leftover" hypothesis we made in the previous paragraph.

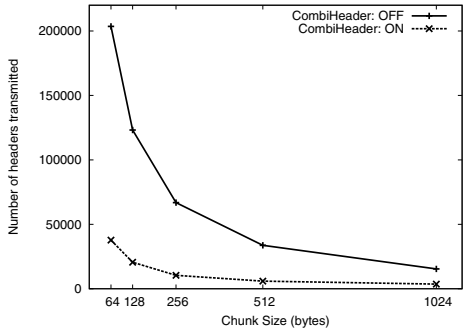

Fig. 5: Effect of CombiHeader algorithm on header transmission savings

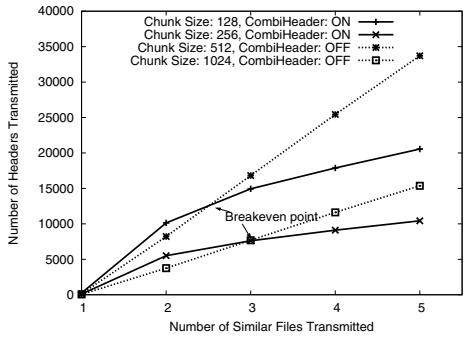

Fig. 6: CombiHeader allowing smaller chunk sizes to be as beneficial as bigger ones

Tunable Memory Requirements: While the algorithm presented in this paper promises wonders in terms of transmitting less bytes to the wire, there is a scope of considerable discussion about how much memory it requires. The algorithm offers a parameter represented by θ to address this. θ controls when two elementary nodes are merged together to generate a CombiNode. Thus by changing the value of this parameter, the memory requirement of the algorithm can be tuned. In Figure 4 the number of allocated nodes are shown. The experiment has been done for 64-byte chunks with different θ value while transmitting the five video files under consideration. The experiment shows that the explosion in node creation always starts when the number of files transferred (and thus the number of hit on similar chunks) reaches θ. This allows us to decide two characteristics of the algorithm: 1. The increase in elementary nodes is linear for similar files because elementary nodes are only generated for missed chunks, 2. The explosion of CombiNodes, and thus memory usage can be conveniently tuned using the θ value.

Merging reduces total transmitted headers: One of the

features that CombiNode promises is to reduce the number of headers to be transmitted. The dramatic nature of this feature is expressed in Figure 5. Five video files are transmitted through the engine twice: once with the combiHeader algorithm turned on and then once with it turned off. Figure 5 shows the number of shim header to be transmitted for both the scenarios. As can be seen, for smaller chunk sizes CombiHeader algorithm requires less than 20% of total headers to be transmitted, while for bigger chunk sizes it requires around 25% of the total.

CombiHeader allows best of both worlds: Continuing from the previous discussion, we will now show that how we can get the best of both worlds (better similarity detection with less header transmission) using CombiHeader. In Figure 6, we show a superimposed graph where four different chunk sizes are used, two bigger ones without CombiHeader and smaller two with CombiHeader. Intuitively, the number of chunk headers inserted for transmission grows linearly without CombiHeader algorithm. While for smaller chunk sizes, they start with a bit higher chunk header insertion as expected but quickly recovers and surpasses the bigger chunk sizes in savings. Both 128-byte and 256-byte chunks cross 512 and 1024 byte chunks sizes respectively before the third similar file is transmitted. This occurs due to more and more complex CombiNode creation and thus lesser shim header transmission.

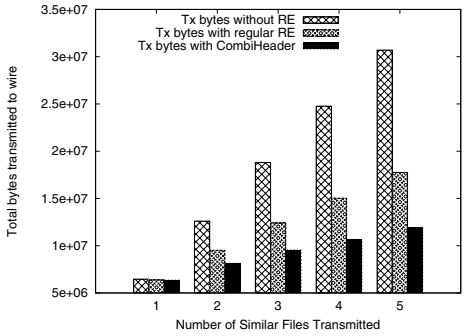

Fig. 7: CombiHeader reduces total bytes transmitted to wire

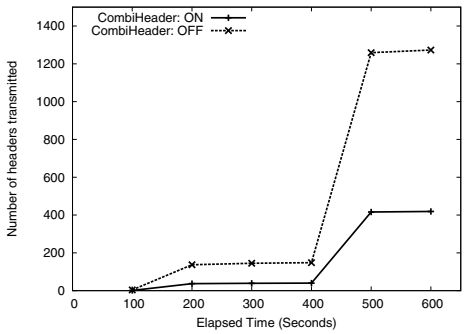

Fig. 8: Shim headers to be transmitted: generic HTTP traffic

CombiHeader's affect on actual bytes transmission: Thus far we have demonstrated how CombiHeader allows transmission of dramatically low number of shim headers. Figure 7 shows how this reduction positively affects the number of bytes to be transferred out to the wire. The figure compares traffic amount of normal transfer with regular RE algorithm, and with CombiHeader helped RE algorithm.

CombiHeader with HTTP traffic: Although previous experiments have already shown the benefits of CombiHeader from different angles, to reinforce the applicability of CombiHeader, we also ran it on a trace of HTTP traffic. In Figure 8, it is clearly visible that even for a very short period of time, CombiHeader shows a definite savings on the number of transmitted headers. The experiment is done with 64-byte chunk size, with $\theta = 1$, on a trace of HTTP traffic captured from a home access point used by several users.

VII. Conclusion and Future Works

The applicability of RE in different scenarios has already been proved by previous works. However, none of them offer an efficient method to reduce the number of shim headers to be transmitted. Moreover, none of the previous works has efficiently solved the tradeoff between chunk size and maximum similarity detection. This paper proposes and evaluates a new algorithm named CombiHeader which tracks the produced chunks according to their popularity and adjacency index, and based on that merges adjacent chunk opportunistically with the premise that if several adjacent chunks are popular, there is a high possibility that they will appear together again in the future. Thus it reduces the number of shim headers and improves compression ratio with very few memory accesses.

Due to the highly dynamic nature of CombiHeader, cache knowledge synchronization among the neighbours to make informed decision about routing is imperative. We keep it as a future work to invent a highly optimized way to sync the cache states among the neighbors. Overall, CombiHeader shows exciting promise to further increase the performance of RE algorithms while providing flexible parameters for adapting it to different usage scenarios.

References

[1] (2010) Bluecoat: WAN optimization. [Online]. Available: http://www.bluecoat.com

[2] (2010) Cisco WAN optimization and application acceleration. [Online]. Available: http://www.cisco.com/en/US/products/ps5680/Products_Sub_Category_Home.html

[3] (2010) Riverbed: WAN optimization. [Online]. Available: http://www.riverbed.com/us/solutions/wan_optimization/index.php

[4] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proc. of SOSP*. ACM, 2001, pp. 174–187.

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

[6] A. Anand *et al.*, "SmartRE: an architecture for coordinated network-wide redundancy elimination," in *Proc. of SIGCOMM*. ACM, 2009, pp. 87–98.

[7] Y. Song, K. Guo, and L. Gao, "Redundancy-Aware Routing with Limited Resources," in *Proc. of 19th IEEE ICCCN*. IEEE, 2010, pp. 1–6.

[8] E. Halepovic, C. Williamson, and M. Ghaderi, "Exploiting Non-Uniformities in Redundant Traffic Elimination," *University of Calgary*, 2010.

[9] M. Rabin, *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ. Harvard, 1981.

[10] A. Broder, "On the resemblance and containment of documents," in *Proc. Compression and Complexity of Sequences*. IEEE, 2002, pp. 21–29.

[11] K. Tangwongsan *et al.*, "Efficient similarity estimation for systems exploiting data redundancy," in *Proc. of IEEE INFOCOM*. IEEE, 2010, pp. 1–9.

[12] S. Ihm, K. Park, and V. Pai, "Wide-area network acceleration for the developing world," in *Proc. of the 2010 USENIX ATC*. USENIX Association, 2010, p. 18.

[13] N. Bjørner *et al.*, "Content-dependent chunking for differential compression, the local maximum approach," *Journal of Computer and System Sciences*, vol. 76, no. 3-4, pp. 154–203, 2010.

[14] M. Ajtai *et al.*, "Compactly encoding unstructured inputs with differential compression," *Journal of the ACM (JACM)*, vol. 49, no. 3, pp. 318–367, 2002.

[15] A. Anand *et al.*, "Redundancy in network traffic: findings and implications," in *Proc of SIGMETRICS*. ACM, 2009, pp. 37–48.