

Improving Packet Caching Scalability Through the Concept of an Explicit End of Data Marker

Xiaolong Li, David Salyers and Aaron Striegel
 Dept. of Computer Science and Engineering
 University of Notre Dame
 Notre Dame, IN 46556 USA
 E-mail: *xli5, dsalyers, striegel@nd.edu*

Abstract—Over the past few years, the web has witnessed an explosion of dynamic content generation to provide web users with an interactive and personalized experience. While traditional web caching techniques work well when redundancy occurs on an object-level basis (page, image, etc.), the use of dynamic content presents unique challenges. Although past work has addressed mechanisms for detecting redundancy despite dynamic content, the scalability of such techniques is limited. In this paper, we present a technique for explicit packet boundary delineation to enable scalable and highly efficient packet caching in the network. Our approach, Explicit End of Data (EEOD), does not require client-side modification and requires only minimal server-side modifications. We demonstrate through experimental studies on an Apache web server 25% and 30% relative improvements in terms of bandwidth efficiency and retrieval time over current approaches in the literature.

I. INTRODUCTION

Significant work has been invested by the research community in improving the efficiency and hence scalability of the web [1]–[3]. However, the emergence of dynamic content (blogs with commentary, user preferences, etc.) creates significant difficulties for traditional caching mechanisms. To that end, various mechanisms [4]–[8] have been proposed to explicitly separate dynamic content into cache-friendly objects. As a result, dynamic content can still be cached in the existing object-based cache infrastructure. Unfortunately, such mechanisms often involve significant and non-trivial site re-design.

In contrast, recent works in [9]–[11] adapt the caching mechanism to detect redundancy in dynamic content without site modifications. Whole packet caching [9], while lightweight computationally, performs poorly when cacheable content is not aligned on a packet basis. Although the techniques in [10], [11] do not require packet-wise alignment, the techniques pay a significant computational price in order to dynamically infer cacheable content boundaries.

In essence, the approaches to dealing with the efficiency of dynamic content can be grouped into two categories: exceptional accuracy with heavyweight site modifications or complete avoidance of site modifications with significant in-band computational expense. It is the premise of this paper that a middle ground can be reached by providing a lightweight mechanism for accurate boundary demarcation of cacheable content.

In this paper, we introduce the concept of an Explicit End of Data (EEOD) marker. With minimal effort, the content provider can force packet separation of cacheable and non-cacheable content to enable highly scalable whole packet caching. Specifically, the contributions of our paper include:

- *EEOD Concept*: The paper proposes the notion of an Explicit End of Data (EEOD) marker to facilitate the efficient and accurate separation at the packet level of cacheable and non-cacheable content.
- *Improved whole packet caching*: The paper introduces an improved whole packet caching model that uses hints from EEOD combined with a novel windowed aggregation scheme.
- *Prototype evaluation*: The paper completes extensive experimental studies contrasting EEOD versus existing schemes. Notably, the paper demonstrates a 25% relative improvement in terms of bandwidth savings in addition to significantly improved scaling properties in terms of retrieval time.
- *Rabin fingerprinting efficiency*: This paper is the first to highlight the poor performance of Rabin fingerprinting when minimal cacheable content exists.

The rest of the paper is organized as follows. In Section II, we present the fundamentals of the EEOD approach. In Section III, we present the integration of EEOD and the Apache web server. Experimental studies are presented in Section IV. In Section V, we discuss related work and address concerns about EEOD. Finally, we present several conclusions and discuss future work in Section VI.

II. EEOD BACKGROUND & FUNDAMENTALS

To provide further background and motivation for EEOD, we discuss several immediately related technologies and highlight critical weaknesses in the efficiency of Rabin fingerprinting.

A. HTTP 1.1: Pipelining and Chunk Encoding

Beyond the improvement of persistent connections offered with HTTP 1.1 [4], two other options related to handling dynamic content are *pipelining* and *chunk encoding*.

With regards to dynamic content, pipelining can be used to overcome the overhead associated with splitting a site into multiple sub-objects for per-object caching. Ignoring

the significant costs associated with site re-design, there are considerable deployment issues to realizing true pipelined performance. First, only minimal support exists among current web browsers. Internet Explorer (IE) does not support pipelining and Mozilla/Firefox supports pipelining but disables all pipelining behavior by default [12]. Second, most caches do not support HTTP 1.1 pipelining. Notably, Squid will translate any HTTP 1.1 request to an equivalent HTTP 1.0 request. Moreover, there are no current plans to support HTTP 1.1 pipelining in Squid [13]. Thus, for the foreseeable future, HTTP 1.1 pipelining is not a reliable option to overcome the overhead associated with sub-objects.

While pipelining is only tangentially related to dynamic content, chunk encoding was directly targeted at dynamic content. Chunk encoding allows a server to send content of an unknown length to a client to reduce the perceived retrieval time with the display of partial results. Although it would appear that chunk encoding could be used to produce similar results to what we will propose with EEOD, there is one critical difference. Specifically, chunk encoding does not ensure the separation of packets, thus requiring the computationally expensive dynamic boundary detection. Moreover, when dynamic content of varying lengths is interspersed in the content, the usage of chunk encoding and its proposed caching mechanisms [14] are often precluded.

B. Rabin Fingerprint Scaling

To better illustrate why in-band cacheable content boundary detection is undesirable, we examine the performance of one of the more popular techniques, Rabin fingerprinting [15]. In short, the Rabin fingerprint employs a relatively efficient sliding window scheme to extract pattern matches. The works in [10] and [11] use Rabin fingerprinting to detect cache hits and tokenize content.

While extremely efficient when content exhibits a significant degree of redundancy, Rabin fingerprinting does not perform well when the content is primarily unique (i.e. cache misses). Figure 1 compares a block-wise approach (MD5 checksum) versus Rabin fingerprinting with variable sliding window sizes on non-cacheable content. The implementations of the algorithms were based on the prototype code of whole packet caching (MD5) [9] and partial packet caching (Rabin) [10]. The tests included the overhead of the payload/window fingerprint, performing cache lookups, and appropriately tokenizing redundant content.

The results show a significant difference in the throughput associated with Rabin fingerprinting versus an MD5 checksum. While MD5 operates in an efficient block-wise manner, the Rabin fingerprint must consistently recompute the sliding window (W) on a byte-wise basis ($O(N - W)$). Furthermore, Rabin fingerprinting uses multiple lookups to the cache as each fingerprint calculation yields multiple potential fingerprints (as noted in [10]). Hence, the Rabin fingerprint approach conducts at worst $O(K(N - W))$ lookups (K fingerprints, N bytes, window size of W bytes) while MD5 conducts a single cache lookup. This trend is especially visible in the figure as a decrease in overall packet size (1518 versus 1024,

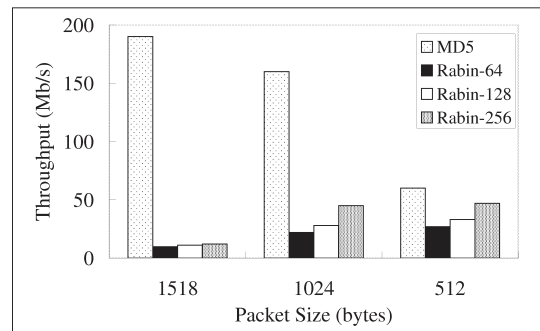


Fig. 1. Redundancy Function Performance Comparisons

etc.) shows an increase in Rabin fingerprint throughput from processing less of a data payload (layer 2-4 information can be ignored). Moreover, an increase in window size also improves performance as additional table lookups are further removed.

The overhead of dynamically deriving redundancy boundaries is quite significant in the worst case which occurs any time a non-cacheable or new but not yet cached packet crosses the cache, a non-trivial portion of the underlying traffic. Importantly, this experiment highlights why it is desirable to utilize whole packet caching instead of partial packet caching. As will be shown later in our experimental results, this overhead of Rabin translates to a non-trivial delay even with only 10-20% non-cacheable content.

C. Motivating EEOD

While it is desirable to use whole packet caching, separation of cacheable and non-cacheable content is not easily achieved in the current environment. Informally, the underlying transport mechanism (TCP) is tasked with reliably moving a block of data from Point A (source) to Point B (receiver). For each packet containing information in the block, a certain amount of overhead is incurred by both the IP and TCP header. The goal of the TCP stack is to maximize efficiency (minimize overhead). Thus, it is only natural to minimize the total number of packets used to send the data. Hence, there is a tendency for the size of TCP data packets to approach the network Maximum Transmission Unit (MTU) size.

To TCP, the concept of not using the full MTU (i.e. separating content) would be anathema to the goal of maximizing efficiency. However, at the initial introduction of TCP, the concept of caching at the object level, or more importantly at the packet level, had not been considered. Moreover, while disabling Nagle's algorithm (automatically disabled by Apache) implies support for the separation of packets, our experiments have found that under a reasonable system load, separation is not guaranteed. Similarly, the usage of the TCP PUSH flag does not ensure packet separation either.

If the content provider could explicitly signal boundaries between cacheable and non-cacheable content which in turn enforces packet separation between the two types of content, whole packet caching could be enabled. The end result is that in-band hardware can be kept considerably simpler and more scalable, leaving intelligence at the edge (input + server) where it is best suited. Importantly, the packet splitting in and of itself does not change the normal TCP client operation.

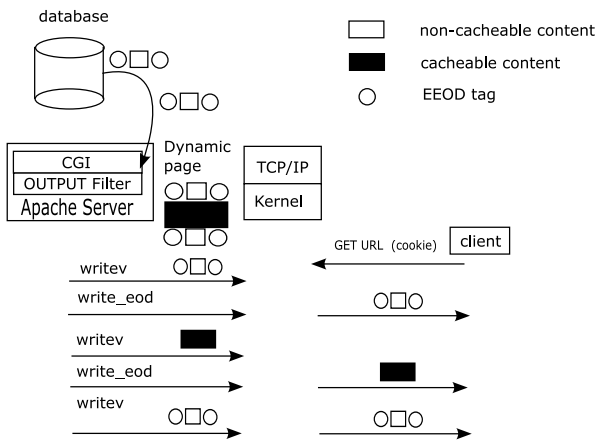


Fig. 2. EEOD Operation Flow

Unlike mechanisms which require significant site re-design to accommodate sub-objects, the boundaries between cacheable and non-cacheable content are often quite straightforward. Whether it be calls to a database or script variables, finding such locations and adding explicit markers to the script output at those locations is trivial.

D. EEOD Overview

The key components of EEOD include:

- *Network stack extension*: Through the addition of a single system call, an application can separate packets in a non-blocking manner. We describe a new system call and the minimal modifications to the Linux kernel to support the extension.
- *Improved packet cache*: Unlike previous work which operates on a single packet [9], our novel improvement allows the cache to examine a limited width window of packets to aggregate cacheable tokens for further bandwidth and computation savings.
- *EEOD demarcation*: A simple demarcation mechanism embedded in HTML content is used to separate cacheable and non-cacheable content without the need for client modification.
- *Apache Modification*: We present modifications to the Apache web server to recognize the EEOD marker and to appropriately call the new EEOD system call.

Before describing the EEOD stack modifications in more detail, an example scenario employing EEOD is discussed. Consider a website similar to Slashdot¹ consisting of a large degree of non-cacheable / unique content scattered amongst redundant data (see Figure 2). The flow of events is as follows:

- 1) The web request is received at the server. The web server calls the appropriate web page generation script with the user cookies or user identification (such as from a login screen).
- 2) The web page generation script queries the content repository (database, locally cached content, etc.) and demarcates the boundaries between cacheable and non-cacheable content with an EEOD marker.

- 3) The web server receives the content from the script. The web server parses the content for the EEOD marker(s). At each marker, the EEOD system call is invoked.
- 4) The EEOD-friendly network stack receives the data and ensures that packets are separated as specified by EEOD system calls.
- 5) The server transmits the TCP data packets towards the client.
- 6) The in-network packet cache receives the data packets. Cache hits are determined on a whole packet basis. Packets are either transmitted onwards with no modifications (cache miss) or tokenized and sent onwards to the downstream packet cache. Fingerprinting of packets is done using MD5 or other efficient algorithms.
- 7) The downstream cache receives the packet. If the packet is tokenized, the original packet is reconstructed and sent onwards. Otherwise, the packet is simply forwarded onwards.

In keeping with the idea of packet caching proposed in [9], cache replacement is governed by the following rules:

- Upon the first transfer, both the parent (cache closer to server) and child cache (closer to client) calculate and store same dictionary token calculated by the fingerprint algorithm (MD5, etc.).
- For subsequent packets with same payload, the payload is replaced with a dictionary token.
- In the event that the child cache receives an invalid token, the payload is requested from the parent cache by the child cache.
- Upon receiving the packet with a token, the child cache replaces the token with the requisite data and forwards the packet to end client.

III. DESIGN AND IMPLEMENTATION

The goal of EEOD is to generate whole packet cache-friendly packet flows, in which the payload of a packet either belongs to cacheable content or belongs to unique content (i.e. non-cacheable).

A. EEOD System Call

The signaling API for the application to create packet separation is listed below:

```
ssize_t sys_write_eod(unsigned int fd,
                    size_t flag);
```

Unlike the normal *sys_write* function, the new function only signals to the modified stack that the preceding block of information passed cannot share the same packet as future writes, instead of passing data to the stack. In order to implement the EEOD system call, one could employ one of two methods, a fully non-blocking system call or application-level waiting/flushing. Without some form of modification, there is no guarantee that multiple calls to the normal *sys_write* will result in separate packets (stream vs. datagram packets).

For example, successive writes of 200, 200, and 400 bytes under *sys_write* could result in a single packet of 800 bytes, two packets of 400 bytes, or the desired behavior of 200,

¹www.slashdot.org

200, and 400 bytes. While enabling the *TCP_NODELAY* option for Linux (disable Nagle’s algorithm) purports to allow for packet separation, a heavy system load exhibits similar behavior. In fact, our experiments validated that the single packet behavior is most often the case without appropriate delays between successive *sys_write* calls. These results were noted through Apache which by default enables *TCP_NODELAY*.

While wait functions could be used to ensure a flush of the socket between successive writes, the timing for the wait depends upon the network and system load which could vary dramatically over time. Unlike RTSP implementations over TCP (typically employed to ensure packets are not blocked by firewalls) which has only periodic writes to the stream and hence a natural temporal separation, web content is burst out in its entirety for performance.

Thus, rather than imposing a complicated sequence of blocking wait events on the application, the operation of *sys_write_eod* is straightforward. Once the kernel receives a *sys_write_eod* call, it simply flushes out the current socket buffer and then uses a new socket buffer for future data. Our test shows that the *sys_write_eod* call is completed in less than 80 μ s. Moreover, by making the EEOD call non-blocking, the expected behavior of stream writes is preserved.

B. Effects on TCP - Local and Global

It is important to note that EEOD does not fundamentally change the TCP protocol itself. While EEOD does change the grouping of the data to be transmitted by TCP, the critical TCP characteristics such as slow start, congestion reaction, etc. remain unmodified². From the perspective of the client, there is no discernable difference nor software to deploy beyond minor changes in packet size distributions.

From a larger network perspective, EEOD can have a discernable effect. If RED queues are also deployed in the network which detect congestion by using the queue packet count, the use of EEOD will result in a faster throttling of flows. However, we propose an improved packet caching technique that overcomes this penalty and also allows for further bandwidth conservation.

C. Improved Packet Caching

As noted earlier, the core packet caching architecture in EEOD is a derivative of the work in [9]. The work in [9] analyzes individual packets and appropriately tokenizes or forwards packets towards the client or downstream cache. While bandwidth savings are achieved whenever tokenization takes place, there exists a 1:1 relationship between input and output packets ($P | Data \rightarrow P | Data$ or $P | Token$). Thus, in the absence of other mechanisms, EEOD injects additional packets into the network and hence suffers additional overhead (IP, TCP) over the bottleneck links.

In contrast, VBWC, proposed in [11], applied Rabin fingerprinting on the entire block at the server, which offers

²Section V discusses the behavioral impacts of EEOD on TCP in more detail.

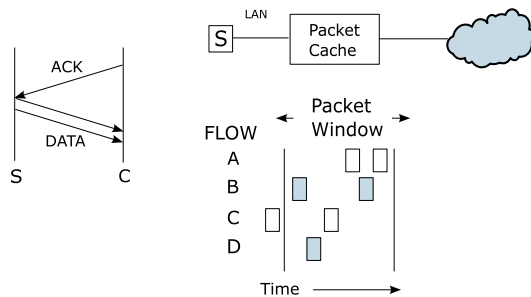


Fig. 3. Packet Aggregation

an interesting proposition: is it possible to tokenize over a larger scale? To explore this further, consider the underlying dynamics of TCP and the likely placement of the upstream packet cache as being close to the server. Put simply, as ACKs arrive, a close temporal burst of data packets from congestion window growth will occur ($ACK \rightarrow Data, Data$). Since the packet cache is close to the source, it is likely that cacheable content will be located in close temporal proximity.

If a minimal, fixed delay window of N packets is introduced, there is a high probability that multiple packets from the same flow will be present in that window (see Figure 3). With a view of a window of N packets, multiple cacheable payloads can be tokenized in the same outgoing packet ($P_1 | Data, P_2 | Data \rightarrow P_X | T_1, T_2$). Thus, the overhead introduced by separating packets is eliminated by aggregating multiple contiguous cacheable packets and saving the respective TCP/IP headers through delta encoding.

Moreover, provided that N is kept small, $N < 10$, issues with downstream cache shaping are contained ($P_X | T_1, T_2 \rightarrow P_1 | Data, P_2 | Data$) and the delay is kept to a minimum. For example, a window size of 10 would imply a delay of 0.12 ms on Gigabit Ethernet. In addition, dispatch times could be further improved if hints are given regarding the cacheability or non-cacheability of content. To that end, packets leaving the EEOD-enabled web server are marked using a bit from the IP ID field (MSb). The receipt of a non-cacheable packet or non-contiguous packet is sufficient to force an early dispatch. At a minimum, the inclusion of the notion of a cacheable bit allows the cache to ignore non-cacheable content, thus further improving cache performance.

D. Application: HTML Content

Given that the script writer can easily determine where the boundaries of cacheable and non-cacheable content occur, the next step is to derive a method whereby the script can signal the web server of the boundary.

In order to achieve the simplicity desired while remaining transparent to the client, we make the EEOD marker a specially formatted HTML comment string. While a proper HTML tag may be in order in the future, the use of a comment field, however unwieldy, accomplishes the goal of end client transparency and simplistic string location identification. The pair of EEOD tags is defined as:

```
<!--EEOD-->
```

```

<HTML><body>
.....
<!--EEOD-->
<B>Hello, Xiaolong Li</B>
.....
<td class="middleoffonleft"><div align="center">
<a href="http://www.amazon.com/exec/obidos/tg/
stores/your/store-home/-/0/ref=pd_ysl_gw_gw_2/
103-8422194-7641468"> Xiaolong's <br /> Store</a>
</div></td>
<!--EEOD-->
.....
</body></HTML>

```

Fig. 4. Tagging the output of a page

```
<!--/EEOD-->
```

where the contents between the bounding EEOD tags are assumed to be non-cacheable content. In short, the identification of either the front or rear EEOD tag offers a suggestion to the web server that the data on either side of the EEOD marker should not appear in the same packet. Figure 4 shows what an EEOD tagged web page would look like.

It is the responsibility of the web page script designer to embed the EEOD tags into web pages. Unlike conversion to Ajax or the usage of sub-objects, the modification for EEOD is extremely trivial. We do not feel this is an unnecessary burden as the appropriate calls for unique content already provide a clear signal of where to demarcate data.

In addition, it is worth noting that for a web page with multiple pieces of non-cacheable data scattered around the page, simply applying EEOD to each segment may be unwise. If the length of the content included by successive EEOD markers is too small, say, 10 bytes (10 bytes cacheable, J bytes non-cacheable, 10 bytes cacheable, K bytes non-cacheable), the usage of EEOD will cause a reduction in bandwidth conservation performance.

E. Application: Web Server

Upon receiving a request, a web server renders the output web page and places the content into a buffer for the kernel using a normal system write call. EEOD adds an additional level of functionality in the EEOD-friendly Apache server, where the server scans the content for EEOD markers to determine if splitting is necessary.

Since the string for EEOD is well-defined, string searching algorithms such as the ones employed by Snort [16] can be used to isolate EEOD tags. With the beginning and end of an EEOD pair, successive calls are made to the `sys_write_eod` function. Packet delineation between cacheable and non-cacheable content is provided, thus optimizing the packet flow for the in-band packet cache to conduct whole packet caching types of operations.

F. Implementation: Apache Web Server, Red Hat Linux

We implemented the EEOD scheme through modifications to the Apache 2 web server (version 2.0.54). Kernel enhancements included the `sys_write_eod` system call and supporting code that were applied to RedHat Linux Fedora Core 3 (version 2.6.11). In-network packet caching was based on code

provided by the authors of [10] and freely available code on the web [17].

When serving content to clients, Apache2 uses two system calls, `writew` and `sendfile`. The `sendfile` function is used to send an existing file directly requested by the client. With respect to the OS, we added the new system call `sys_write_eod` to the Linux kernel. From the user perspective (i.e. Apache), the call is `write_eod`. Modifications to the kernel included the addition of a new member variable `eodflag` to the socket buffer data structure and `sk_buff`, which indicates a socket buffer is to be closed. The new system call is a relatively minor modification to the kernel. Most importantly, the modified kernel does not change normal TCP behavior, only how buffers are dispatched for transmission by the TCP stack.

Although `sys_write_eod` guarantees packet separation between cacheable and non-cacheable content, EEOD does not guarantee that contiguous content will be packetized in the same fashion (i.e. a large cacheable block). In such a case, system load or other network activities at the server may cause misalignment which we define as identical contiguous blocks yielding different network packetizations. To account for this, we modified the Apache server to dispatch buffers at multiples of MSS and hold if possible where packet misalignment may occur. Section V-C comments further on the prevalence of misalignment with large file transfers and the relevance of EEOD for other network services.

IV. EXPERIMENTAL STUDIES

In this section, we present experiments comparing the performance of EEOD and related approaches. Figure 5 shows the experimental setup. Internet emulation is provided through a single machine running NISTNet [18] between the upstream and downstream packet caches. The web server (Apache) is running on Red Hat Fedora Core 3 with the extensions outlined earlier. Experiments with EEOD utilize an EEOD-friendly Apache server while other experiments use the standard Apache server. The web server itself uses a CGI Perl script to retrieve content from a database running concurrently with the server. Each of the experiments lasted 300 seconds with new web pages being cycled every 5 seconds. Non-cacheable content is consistently changed over the course of the experiment.

Client emulation is provided through a bank of clients using `wget` for HTTP 1.0 accesses and the base `libwww` client from the W3C for HTTP 1.1 accesses. Session performance was noted to millisecond accuracy levels by customized versions of `wget` and `libwww`. Two in-network devices on COTS hardware are used to serve as the packet caching mechanisms and the prime monitoring point for bandwidth efficiency measurements. In order to mimic a WAN environment, a machine running NISTNet [18] was placed between the two caches.

Specifically, our experiments analyze performance using two metrics:

- **Bandwidth Efficiency:** Bandwidth efficiency is defined as the ratio of bandwidth consumed over the cache link versus the non-caching case. An efficiency of 10% implies that the approach consumes 10% of the bandwidth of the non-caching approach.

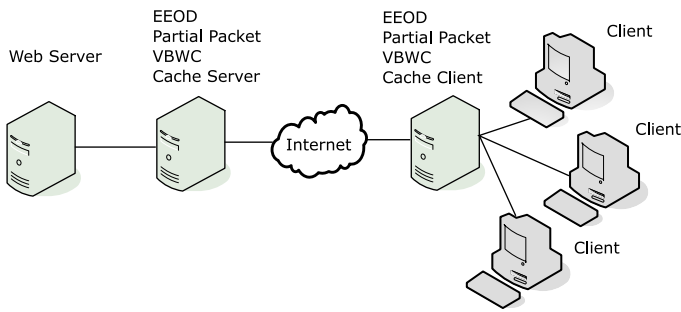


Fig. 5. Experiment Environment

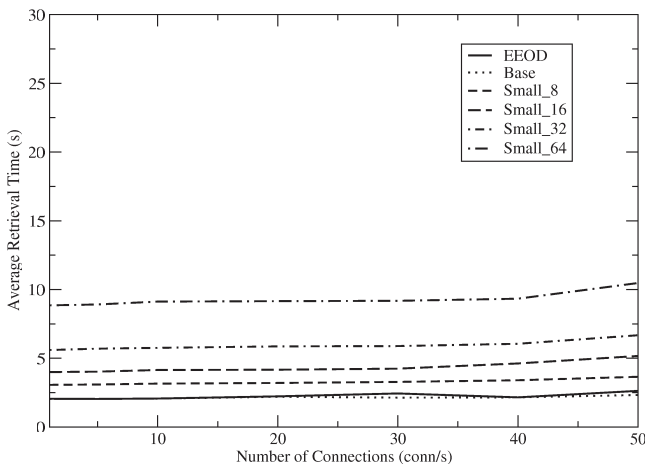


Fig. 6. Effect of the number of the objects of web pages on retrieval time by connections per second. (No HTTP pipelining, no squid cache)

- *Average Retrieval Time*: The average retrieval time is the time to retrieve the entire content of the referenced page. The average retrieval time measures the latency introduced by the approach in terms of both computation and download dynamics.

A. Effect of Sub-Object Usage and Squid

To begin, we first present experiments that show the effect on performance of prolific sub-object usage. The base webpage contains a Slashdot-like page with an initial page and 14 small images. Figure 6 shows the performance with multiple levels of sub-objects and HTTP 1.1 via the *libwww* client. For example, the *Small_32* denotes the usage of 32 additional objects over the core webpage. In each case, the total content to display is kept the same. Packet caching is disabled but EEOD is included to reflect the overhead of EEOD modifications (system call, EEOD tag scan).

Note from the figure that the usage of sub-objects imposes severe penalties due to the fact that the full object must arrive before the next object can be transmitted. While the penalty associated with TCP slow start is removed by the use of HTTP 1.1, a significant penalty exists nonetheless. Figure 7 shows the effect of enabling pipelining which enables a single request to retrieve multiple objects at the same time. While pipelining significantly reduces the delay penalty for sub-objects, limitations to the number of objects that can be pipelined still cannot overcome the over-zealous use of sub-

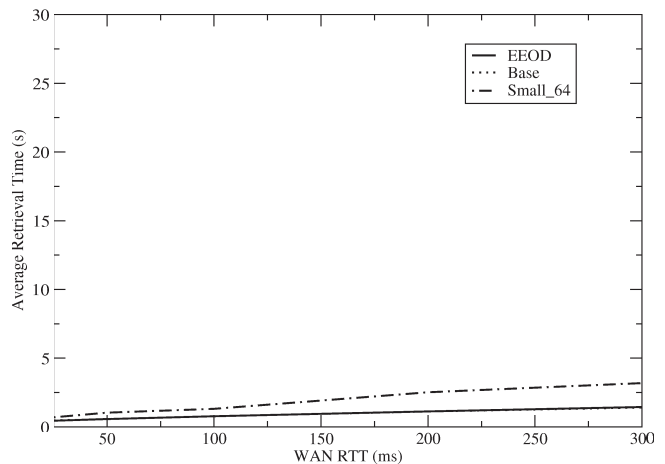


Fig. 7. Effect of the number of the objects of web pages on retrieval time by RTT. (With HTTP pipelining, no squid cache)

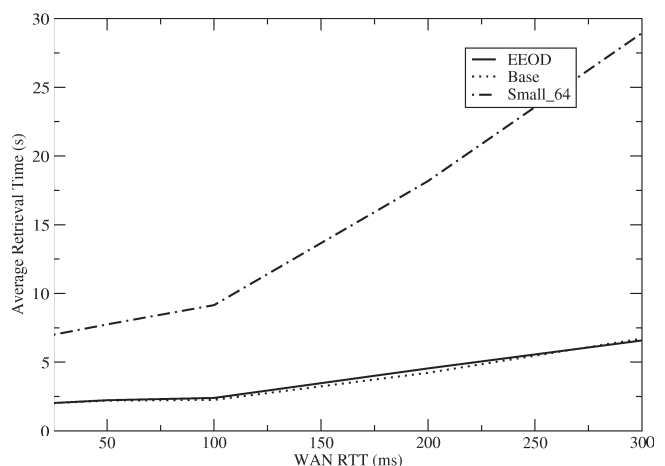


Fig. 8. Effect of the number of the objects of web pages on retrieval time by RTT. (No HTTP pipelining, no squid cache)

objects. As noted earlier, the pipeline feature is minimally available in the current Internet.

Figure 8 shows that as the RTT is increased, the performance of using sub-objects decreases significantly. While the usage of a Squid cache as noted in Figures 9 and 10 reduces the delay penalty, the fact that many sub-objects are likely to be non-cacheable nullifies the gains from a Squid cache. While Squid is able to handle pipelined requests from the client, rather than forwarding pipelined request to origin server, it only sends HTTP/1.0 request to origin server [13], which is verified in our experiments. Since the combination of HTTP pipelining + Squid does not make much difference to the performance, we do not include the results associated with HTTP pipelining + Squid.

B. Synthetic Experimental Studies

In this subsection, we present experiments based on synthetically created web content in order to better isolate the performance of EEOD versus existing packet caching schemes. Specifically, we are interested in characteristics such as efficiency and the effect of distributions of cacheable vs. non-cacheable content (i.e. what effect does the dynamic content have on performance).

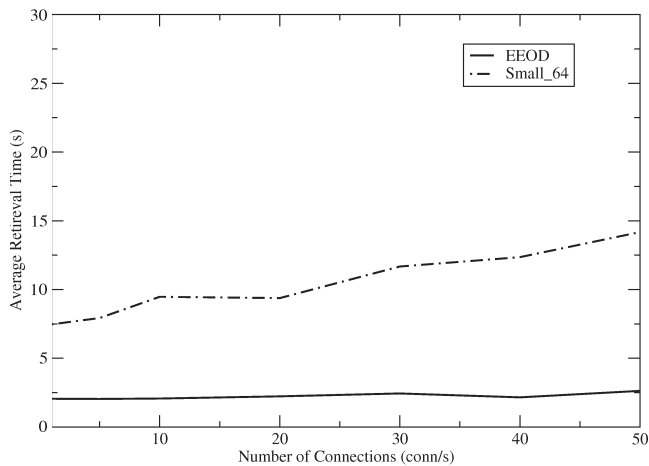


Fig. 9. Effect of the number of the objects of web pages on retrieval time by connections per second. (No HTTP pipelining, with squid cache)

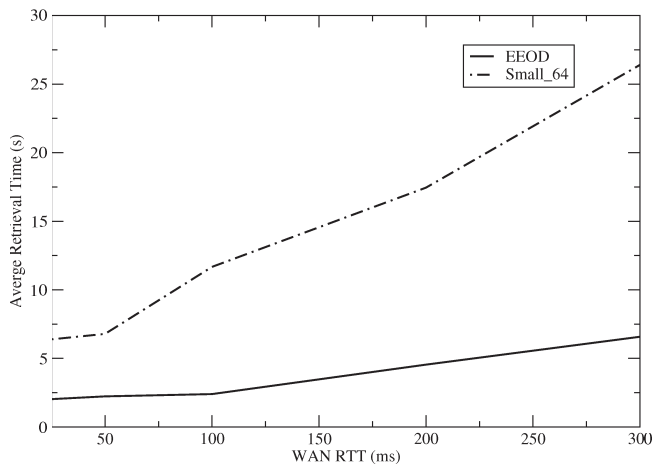


Fig. 10. Effect of the number of the objects of web pages on retrieval time by RTT. (No HTTP pipelining, with squid cache)

In our experiments, we compared three schemes:

- *Partial Packet Cache (PPC)*: The partial packet cache is based on the work in [10]. The partial packet cache isolates partial packet redundancy through a small window Rabin fingerprint (window size = 64 bytes).
- *Explicit End of Data (EEOD)*: The source script employs the EEOD marker to demarcate cacheable versus non-cacheable content. Packet caching in the network is provided through the improved packet caching mechanism proposed in the paper.
- *No Cache*: Results of the two schemes were compared using a non-caching as a baseline for bandwidth efficiency.

Performance metrics were the same as the metrics outlined earlier. In addition, we also conducted experimental studies with prototype code for Value Based Web Caching (VBWC) [11]. In VBWC, the initial server completes redundancy detection using small window Rabin fingerprints over the entirety of the data block. However, the VBWC prototype provided by the authors of [11] was able to handle only a limited connection load, thus presenting an unfair comparison of VBWC. Hence, we do not include results of VBWC in the performance graphs but offer commentary on VBWC from our observations with lighter system loads.

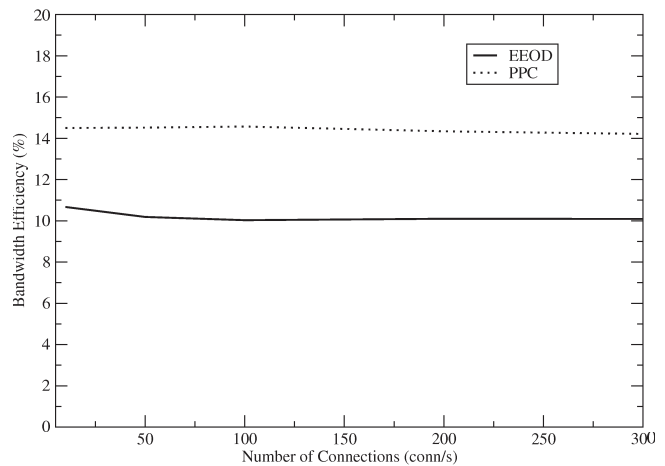


Fig. 11. Effect of the number of connections on bandwidth efficiency

Web Content Generation: To provide a source for our experiments, we created a database of synthetic web pages. Each webpage contained a mixture of content with a single 30 kB image. For the text portion of the web page, the page reduced to blocks of data that were either cacheable or non-cacheable. Cacheable content is repeated until the web content changes while non-cacheable content is unique to each client session. Web page dynamics are mimicked through periodic changes of the distribution of data blocks.

Our synthetically generated data and clients had the following properties:

- 1) *Page Size*: Total non-image content retrieved (default is 64k³).
- 2) *Non-Cacheable Content*: Percentage of the content that was non-cacheable (10% based on [5]).
- 3) *Cacheable Gap Size*: The gap between two successive cacheable blocks that is made up of non-cacheable content (default is 300 bytes).
- 4) *Client connection rate*: The average number of clients per second that connect to the server (default is 100 clients/s).
- 5) *RTT*: The round trip time between the client and server provided by NISTNet (default is 100 ms, 10 ms standard deviation).
- 6) *Update period*: The frequency at which the entire content of the web page changes (5s).

Effect of the Number of Clients: Figures 11 and 12 show the performance as the client connection rate is varied from 5/s to 300/s. Over the entire spectrum, sufficient capacity exists in both the network and server to handle all clients. The initial brief improvement in performance represents cache hits becoming more effective sufficient client hits occur between content changes. Figure 11 demonstrates that although EEOD injects more packets into the network, our proposed aggregation scheme offsets the penalty of additional packets by reasonable amounts.

Most importantly, Figure 12 illustrates the key claim of the paper, namely that dynamic boundary detection does not

³Following CNN.com

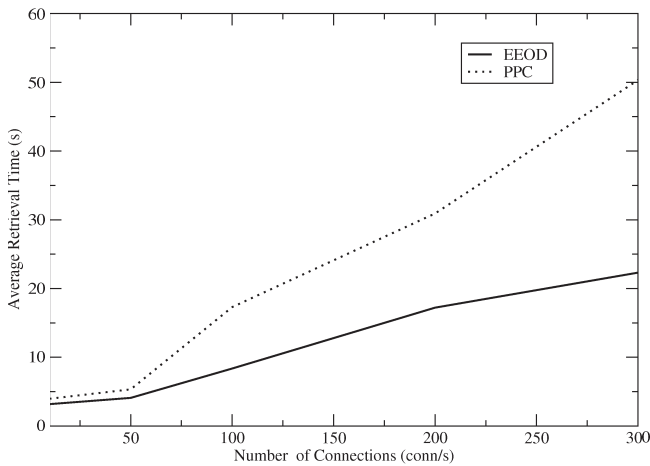


Fig. 12. Effect of the number of connections on retrieval time

scale well. As noted by our scalability experiments earlier in the paper, the computation overhead of dynamic boundary detection (Rabin fingerprinting) comes at a significant cost. At 50 connections per second, the computation overhead of PPC begins to dominate, rapidly accelerating as the number of client increases. In contrast, EEOD stays relatively flat, exhibiting significantly better scaling properties despite the introduction of the limited delay window for aggregation. We believe this offers credible support to the notion that the simpler whole packet cache detection mechanism will be more scalable than dynamically determining redundancy boundaries.

Effect of Web Page Layout: Figure 13 shows the performance of the schemes as the percentage of non-cacheable content is varied. As the percentage of non-cacheable content increases, the chances that EEOD will split packets and incur additional overhead increases as well. Both schemes follow a similar trend in terms of bandwidth efficiency.

However, Figure 14 further explains the performance of PPC. While EEOD relatively slowly with delay being consumed by the new non-cacheable content, the PPC curve suffers significantly in terms of delay. Put simply, as the amount of non-cacheable increases, the number of table lookups for Rabin fingerprinting increases proportionately. If a block is non-cacheable, Rabin fingerprinting must scan each individual byte and compute multiple table lookups. In contrast, a cacheable block will yield a cache hit and simpler *memcmp* calls to find the boundary. Hence, the additional table lookups impose a severe penalty despite the relative speed of the algorithm itself as evidenced in the graph and despite the usage of extremely efficient table structures.

Effect of Web Page Size: Figures 15 and 16 plot the performance of EEOD versus PPC as the average page size is varied. As the page size increases, both approaches suffer slightly decreased performance as the total volume of non-cacheable content also increases proportionately. The mode change in EEOD at 32 kB is important to note as at this point, the effect of aggregation begins to have a significant effect. With additional aggregation EEOD is able to offer increasing performance benefits as an increased congestion window allows for larger bursts of contiguous redundant

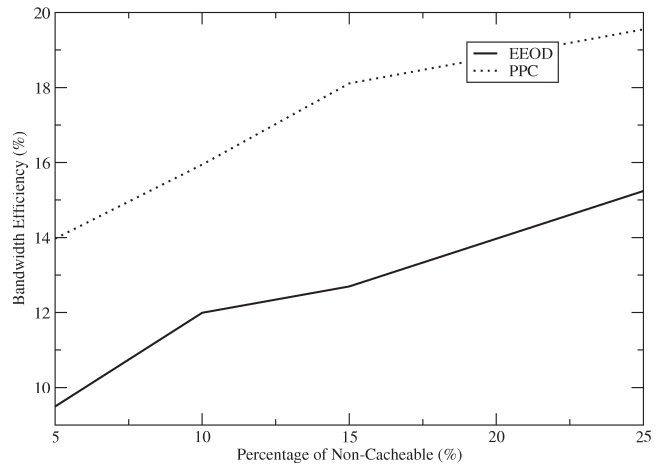


Fig. 13. Effect of the layout of web pages on bandwidth efficiency

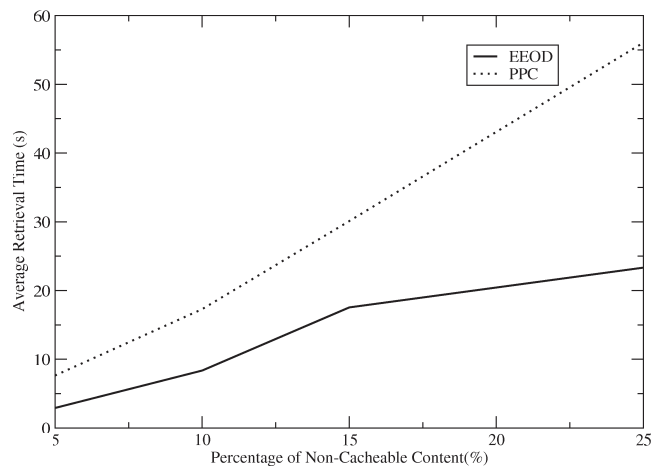


Fig. 14. Effect of the layout of web pages on retrieval time

content. The savings of packet aggregation is also noticeable in Figure 16 where the delay begins to taper off despite increasing page sizes. Table I shows how the number of packets increases with EEOD but yet significantly decreases after aggregation is applied.

C. Realistic Content Experiments

While synthetic content generation allows for variations along specific axes (amount of dynamic content, etc.), we also conducted experiments using source content from popular websites (CNN, Amazon, Slashdot) in order to better validate the comparisons between EEOD and PPC.

For the source content, we gathered pages from the three target websites across multiple clients at thirty second intervals over the period of a single day. The web pages were logged to disk and served via our own Apache server. For EEOD marker generation, we utilized the *diff* tool to isolate where dynamic content occurs.

Figures 17 and 18 show the performance of PPC and EEOD under the three respective website sets of content. To better emphasize the dynamicity of the content, content was set to change every 5 seconds (accelerating from the 30 second interval it was recorded).

TABLE I
AVERAGE NUMBER OF PACKETS PER CONNECTION FOR DIFFERENT PAGE SIZE

Approaches	Avg. Page Size	4kB	8kB	16kB	32kB	64kB
EEOD	Avg. Packet Size (bytes)	1376	1420	1423	1374	1283
	Avg. # of Pkts/Conn	31.23	33.75	40.59	53.84	84.93
	Avg. # of Pkts/Conn. with Pkt Aggregation	8.13	8.27	9.37	15.71	30.52
PPC	Avg. Packet Size (bytes)	1468	1491	1498	1484	1455
	Avg. # of Pkts/Conn	29.73	30.1	37.19	49.62	79.41

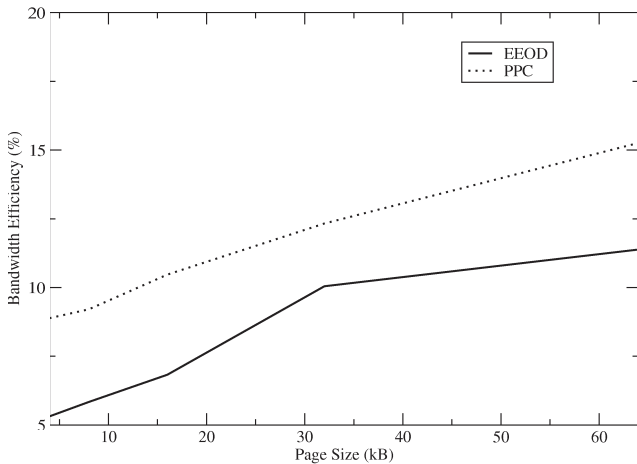


Fig. 15. Effect of the page size on bandwidth efficiency

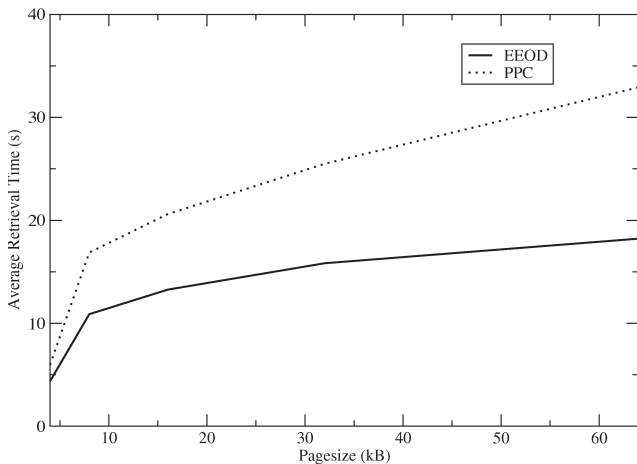


Fig. 16. Effect of page size on retrieval time

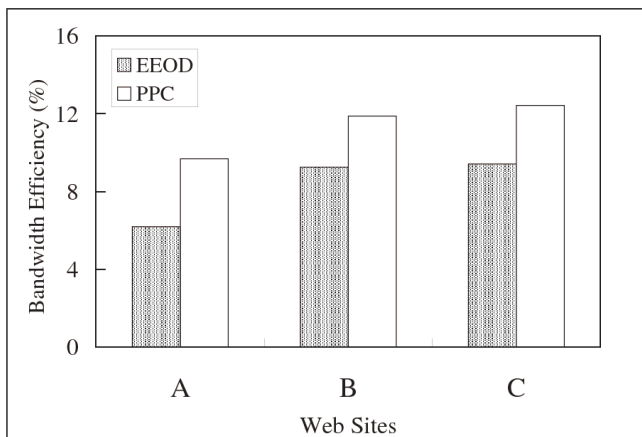


Fig. 17. Bandwidth Efficiency for (A) CNN.com (B) Amazon.com (C) Slashdot webpages

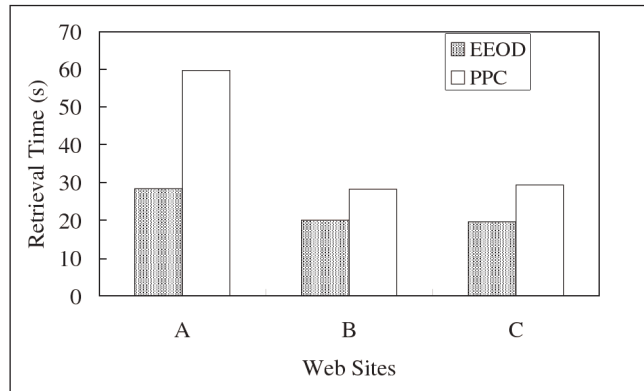


Fig. 18. Retrieval time for (A) CNN.com (B) Amazon.com (C) Slashdot webpages

In the case of CNN, EEOD offers the largest performance improvement as the dynamic content is easy to identify through the usage of the simple *diff* mechanism. In contrast, the dynamic content of Slashdot and Amazon is harder to identify. In the case of Slashdot, the table embedding creates problems with our simple *diff* mechanism, thus identifying more dynamic content than is truly present (many small EEOD blocks). The same problem occurs with Amazon, albeit on a different scale.

Similar to our synthetic experiments, EEOD significantly improves the retrieval time in each of the cases (see Figure 18). EEOD offers the best improvement over the content from CNN as the percentage of cacheable content is higher and EEOD blocks are placed in the most optimal fashion. Furthermore, we believe that with correct changes to the actual source script versus the naive use of *diff*, one could easily expand the gap in bandwidth efficiency and retrieval time for both the Slashdot and Amazon cases.

V. EEOD CONSIDERATIONS AND RELATED WORK

Although our experiments show that EEOD is an efficient and scalable cache architecture, there are several open issues and weaknesses that bear further study:

- 1) *RED interaction*: Since EEOD injects additional packets into the network, the chance that the content will arrive without a loss also decreases due to queue management mechanisms such as RED [19]. This issue is partially addressed by the usage of the packet aggregation mechanism proposed in the work.
- 2) *Wireless interactions*: Similar to the interactions with RED, if the end client uses a wireless medium for connectivity, the higher loss rate may increase web page retrieval time as both the data and ACKs must cross

the wireless medium. Simply put, EEOD increases the number of packets which in turn will increase the chance that a packet will be lost due to the lossy medium. Packet aggregation does not assist with this aspect as packets have been detokenized before arriving at the wireless medium.

- 3) *TCP fairness*: Finally, since EEOD has the potential to result in more packets at sub-MTU values, the overall fairness of the EEOD flows versus non-EEOD TCP flows is impacted. Whereas most TCP flows will transmit packets at MTU sizes (dominated by Ethernet at 1500 bytes), flows adjusted by EEOD will receive a reduced fairness when comparing on a strict packet-by-packet basis. Conversely, the ability to scalably tokenize more packets and offer significant bandwidth savings introduces an interesting tradeoff. Our current ongoing work involves conducting large scale simulation studies to assess this tradeoff further.

A. Related work

As mentioned earlier, there are several works closely related to EEOD. To the best of our knowledge, the work in [9], is the first work to introduce the concept of packet caching. In the work, packet fingerprints were calculated on a whole packet basis using a MD5 fingerprint. The extension to this work in [10] introduced the use of Rabin fingerprinting to allow for partial packet caching with the idea of detecting redundancy across multiple connections. The use of packet caching was also employed in [20] to improve file system performance.

The work in [11], Value Based Web Caching (VBWC), took the work in [10] one step further by applying Rabin fingerprinting before the message left the web server. It is important to note that VBWC was targeted at low bandwidth connections (dial-up), where computational capacity far exceeds bandwidth capabilities. VBWC requires changes to clients and VBWC proxies must explicitly track client cache state which introduces significant large scale deployment obstacles.

In these approaches, unlike the traditional object-level cache, a digest-index cache is employed in order to detect redundant transfers. While the work in DTD [21], Duplicate Transfer Detection, also uses the digest-indexed cache, it remains an object-level cache as the digest is computed only for the whole object. Furthermore, DTD requires the support from the HTTP protocol and changes to both the server and the client. Another work closely related to EEOD was the introduction of delta encoding for use in web pages as described in [5]. A more recent work is that of stealth multicast, described in [22]. Stealth multicast enqueues packets temporarily to detect close temporal redundancy (as with streaming traffic) and convert packets dynamically to/from multicast.

Unlike these approaches, base-instant caching [6] and template caching [7] encourage content providers to design website in a cache-friendly way. Although the former is transparent to web developers who create content, it adds the overhead of computing and storing the delta on the critical path of the request processing.

Although the template caching shares similar ideas with EEOD, separating cacheable and non-cacheable content, they differ in several aspects. In contrast to EEOD, while template caching requires a customized proxy or applet at the client, no client-side changes are needed for EEOD. Moreover, template caching needs complicated update mechanisms and the creation of a new template language for the website.

Another dynamic caching scheme is presented in [23]. In this work the authors develop an algorithm that allows for only changed objects to be updated to the cache, instead of the whole web page needing to be updated to the cache. It is important to note that this work deals with dynamic content that is changing in the same way for all users (an example would be live baseball box scores), not personalized content that changes on a per-user basis. This caching system was designed to alleviate the load placed on the server by allowing the server to push only the changed objects to the cache. In contrast, EEOD is not meant to alleviate the load placed on the servers, but to conserve bandwidth between the upstream and downstream caches.

For several of the related works, it is important to note that our scheme, EEOD, is quite complementary. First, we build upon the whole packet cache work in [9] with an improved packet cache with close temporal aggregation. While one could employ Rabin fingerprinting, albeit with a window size equal to the size of the packet for using the scheme in [10], our internal results indicate the speed would not be comparable to a highly optimized MD5 algorithm. Stealth multicast would benefit by having a better chance of detecting close temporal redundancy, even with TCP accesses.

Perhaps the schemes that would benefit the most would be VBWC and Delta Encoding. For VBWC, the large overhead of detecting boundaries could be easily replaced by a simple string search. While VBWC does require a proxy or changes to the client, we believe the combination of VBWC and EEOD could be quite promising. For delta encoding, the delta calculation process could be fine tuned to only calculate deltas on non-cacheable content.

B. Beyond Web Services

It is worth noting that the EEOD concept is not exclusive to web applications; it has potential in all applications (e.g. FTP, P2P, and TivoToGo) which share the following characteristics:

- Multiple transfers of identical blocks of data.
- Each transfer produces a different packet sequence.

As noted earlier, misalignments frequently occur when transferring a blocks of data over TCP. For a file transfer (FT) application, the effect is the same as with dynamic web pages in that a misalignment prevents the whole packet cache from identifying the redundant content.

C. Prevalence of Misalignment

Using FTP, we conducted experiments to study the prevalence of misalignment in simple file transfers. We conducted an FTP using the FTP server (*vsftpd*) that is included with Fedora Core 3. We transferred files of 6 different sizes (1M,

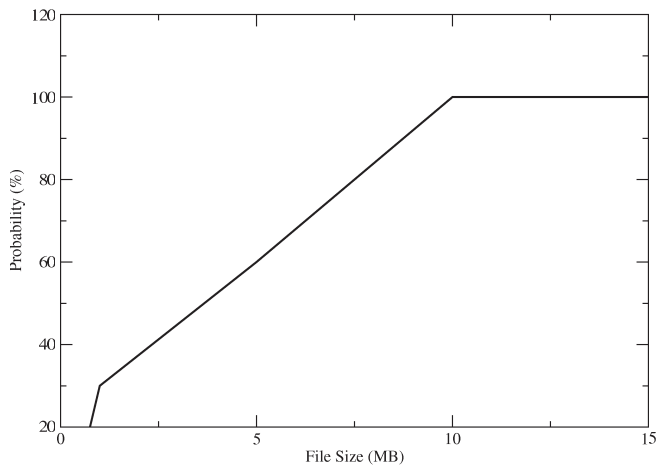


Fig. 19. Probabilities of the Occurrence of the Misalignment in 10 Consecutive File Transfers

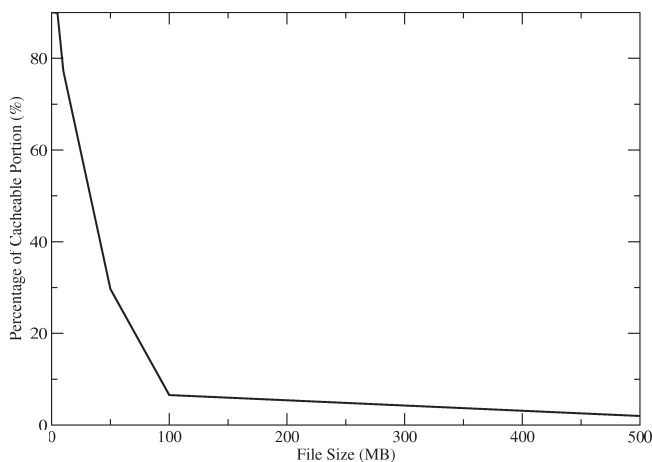


Fig. 20. Percentage of the Cacheable Content Range of Different Files

5M, 10M, 50M, 100M and 500M). The client downloaded each file 10 times and the entire packet trace was recorded by *tcpdump*. As noted in Figure 19, once the file size was beyond 10 megabytes, misalignment occurred in all transfers. Although the sub-MTU packets do not occur frequently (96 out of 345265 packets), their presence would significantly degrade the performance of whole packet caching by making subsequent packets appear to be different content.

Furthermore, even if the packet sequence gets re-synchronized, the cacheable content yielded by the re-synchronization yields only a small portion in the entire file transfer (less than 10% for files beyond 100MB - see Figure 20).

As is shown the misalignment of a file transfer is common, and can significantly degrade the performance of a packet cache. In order to avoid the occurrence of such a misalignment, an intuitive, but naïve, method is to transfer the file in block by block manner, forcing re-synchronization at the beginning of each block, something which EEOD is well suited for. As a file is entirely cacheable, the overhead of searching for EEOD tags could be avoided and a single calling of *sys_write_eod* could achieve the forcing of synchronization. Moreover, the ability to aggregate a cacheable chunk makes EEOD even more attractive in terms of bandwidth efficiency. Further work

is necessary examining the application of EEOD to other file transfer applications.

VI. SUMMARY

In this paper, we presented a bandwidth conservation architecture centered around the concept of Explicit End of Data (EEOD). Through the use of EEOD, one can easily demarcate the boundaries between cacheable and non-cacheable content. The separation of packets to allow for whole packet caching rather than partial packet caching dramatically simplifies in-band packet caching devices, which achieves 30%+ relative improvement in terms of retrieval time. The primary source of this savings occurs from eliminating the costly boundary detection associated with small window Rabin fingerprint searching. Moreover, we introduced a novel packet aggregation mechanism that trades a minimal amount of delay for introducing additional bandwidth savings (25% relative improvement versus PPC) despite the use of more packets by EEOD.

Our experiments on both synthetic and real web traffic demonstrated that EEOD offers significantly improved processing efficiency in addition to further bandwidth savings. Thus, we believe EEOD offers a compelling new approach to improving the efficiency of dynamic web content that merits future attention.

Our future work includes releasing the EEOD mechanisms as open source software, Gigabit scale packet caching using the Intel IXP, and long terms studies on the university tap regarding cacheable versus non-cacheable content.

ACKNOWLEDGMENTS

This research was supported by National Science Foundation through the grant CNS03-47392. We would like to thank Jeff Smith for his assistance in developing the initial EEOD prototype as part of a NSF REU. Thanks to David Wetherall and Neil Spring for sharing the Partial Packet Cache code. We also would like to thank Sean Rhea for sharing the VBWC prototype code.

REFERENCES

- [1] D. Wessels, "The Squid internet object cache." 1997, available at <http://squid.nlanr.net/Squid/>.
- [2] T. Schroeder, S. Goddard, and B. Ramamurthy, "Scalable web server clustering technologies," *IEEE Network*, pp. 38–45, May/June 2000.
- [3] H. Bryhni, E. Klovning, and Q. Kure, "A comparison of load balancing techniques for scalable web servers," *IEEE Network*, pp. 58–64, 2000.
- [4] R. Fielding, J. Gettys, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – (http/1.1)," in *RFC 2616*, 1999.
- [5] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta-encoding and data compression for http." in *Proceedings of the ACM SIGCOMM Conference.*, Cannes, France, Sept. 1997, pp. 181–194.
- [6] B. Housel and D. Lindquist, "WebExpress: A system for optimizing web browsing in a wireless environment." in *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, 1996, pp. 108–116.
- [7] F. Douglis, A. Haro, and M. Rabinovich, "HPP: HTML macro-preprocessing to support dynamic document caching," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997, pp. 83–94.
- [8] J. J. Garrett, "Ajax: A new approach to web applications," <http://adaptivepath.com/publications/essays/archives/000385.php>, 2005.

- [9] J. Santos and D. Wetherall, "Increasing effective link bandwidth by suppressing replicated data," in *Proceedings of the USENIX Annual Technical Conference*, New Orleans, Louisiana, June 1998.
- [10] N. T. Spring and D. Wetherall, "A protocol independent technique for eliminating redundant network traffic," in *Proceedings of the 2000 ACM SIGCOMM Conference*, Stockholm, Sweden, Aug. 2000.
- [11] S. C. Rhea, K. Liang, and E. Brewer, "Value-based web caching," in *Proceedings of the 12th international conference on World Wide Web*, Budapest, Hungary, May 2003, pp. 619–628.
- [12] Mozilla.org, "HTTP/1.1 Pipelining FAQ," available at www.mozilla.org/projects/netlib/http/pipelining-faq.html.
- [13] H. Nordstrom, "Which HTTP v1.1 features are supported - squid mailing list," <http://www.squid-cache.org/mail-archive/squid-users/200204/1030.html>.
- [14] A. Rousskov, "Range requests - squid mailing list," <http://www.squid-cache.org/mail-archive/squid-dev/199801/0005.html>.
- [15] M. Rabin, "Fingerprinting by random polynomials," Department of Computer Science, Harvard University, Tech. Rep. TR-15-81, 1981.
- [16] A. Aho and M. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–343, June 1975.
- [17] R. Rivest, "The MD5 message-digest algorithm," *IETF RFC 1321*, Apr 1992.
- [18] M. Carson and D. Santay, "NIST Net-a linux-based network emulation tool," *Computer Communication Review*, June 2003.
- [19] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 9, pp. 397–413, Aug 1993.
- [20] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proc. of ACM Symposium on Operating System Principles(SOSP)*, Oct. 2001.
- [21] J. C. Mogul, Y. M. Chan, and T. Kelly, "Design, implementation, and evaluation of duplicate transfer detection in http," in *First Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.
- [22] D. Salyers and A. Striegel, "A novel approach to transparent bandwidth conservation," in *Proc. of IFIP Networking*, Waterloo, Canada, May 2005.
- [23] J. Challenger, A. Iyengar, and P. Dantzic, "A scalable system for consistently caching dynamic Web data," in *Proceedings of IEEE Infocom 1999*, New York, NY, Mar. 1999, pp. 294–303.