IMPROVING PACKET CACHING SCALABILITY THROUGH THE CONCEPT OF AN EXPLICIT END OF DATA MARKER

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science
in Computer Science and Engineering

by

Xiaolong Li, B.S., M.S.

Aaron Striegel, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
July 2006
Improving Packet Caching Scalability Through the Concept of an Explicit End of Data Marker

Abstract
by
Xiaolong Li

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 thesis, an effective and highly scalable approach, Explicit End of Data (EEOD) is presented, which allows the content designer to easily signal boundaries between cacheable and non-cacheable content. EEOD provides application-to-stack mechanisms to guarantee separation of packets with the end goal of simplifying packet-level caching mechanisms. Most importantly, EEOD does not require client-side modifications and can function in a variety of server-side/network deployment modes. Additionally, experimental studies are presented, showing EEOD offers 25% and 30% relative improvements in terms of bandwidth efficiency and retrieval time over current approaches in the literature.
I dedicate this work to my family and loved ones. This is for Xiaoyang, who has been a great inspiration for her strength and spirit. This is for my parents, who have always believed their son could do anything.
## CONTENTS

<table>
<thead>
<tr>
<th>FIGURES</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFACE</td>
<td>vii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>viii</td>
</tr>
</tbody>
</table>

### CHAPTER 1: INTRODUCTION 1

1.1 Overview 1

1.1.1 Web Performance 1

1.2 Web Caching Proxy 4

1.2.1 Cacheable vs. Non-Cacheable Content 7

1.2.2 Caching Non-Cacheable Content 9

1.3 Motivation 10

1.4 Contributions 10

1.5 Thesis Outline 11

### CHAPTER 2: EEOD BACKGROUND AND FUNDAMENTALS 13

2.1 HTTP 1.1: Pipelining and Chunk Encoding 13

2.2 AJAX 14

2.3 Base Instance Caching and Template Caching 15

2.4 Packet Caching and Rabin Fingerprinting Algorithm 16

2.4.1 Packet Caching Schemes 17

2.4.2 MD5 vs. Rabin Fingerprinting 18

2.4.3 Rabin Fingerprinting Efficiency 20

2.5 Further Motivating EEOD 22

### CHAPTER 3: EEOD 24

3.1 EEOD Overview 24

3.2 Design and Implementation 27

3.2.1 EEOD System Call 27

3.2.2 Effects on TCP - Local and Global 29
3.2.3 Improved Packet Caching ........................................ 30
3.2.3.1 Holding and Dispatching of Cacheable Packets .......... 32
3.2.4 Application: HTML Content .................................... 33
3.2.4.1 EEOD Placement ............................................. 36
3.2.5 Application: Web Server ........................................ 37
3.2.6 Implementation: Apache Web Server, Red Hat Linux ...... 37
3.2.6.1 System Write Eod ........................................... 38
3.2.6.2 Apache Web Server ......................................... 38

CHAPTER 4: EXPERIMENTAL RESULTS ................................. 42
4.1 Effect of Sub-Object Usage and Squid .......................... 44
4.2 Synthetic Experimental Studies ................................... 46
4.2.1 Web Content Generation: ..................................... 48
4.2.2 Effect of the Number of Clients: ............................. 49
4.2.3 Effect of Web Page Layout: .................................... 50
4.2.4 Effect of Web Page Size: ...................................... 51
4.3 Realistic Content Experiments ..................................... 53

CHAPTER 5: EEOD CONSIDERATIONS AND RELATED WORK ...... 57
5.1 EEOD Considerations .............................................. 57
5.2 Related Work ..................................................... 58
5.3 Beyond Web Services .............................................. 60
5.4 Prevalence of Misalignment ...................................... 60

CHAPTER 6: SUMMARY AND FUTURE WORK .......................... 63
6.1 Summary .......................................................... 63
6.2 Future Work ...................................................... 64

BIBLIOGRAPHY ......................................................... 65
1.1 The Web Client-Server Model .................................. 2
1.2 Basic Scenario of Web Cache ................................ 5
1.3 Dynamically generated web page .......................... 6
2.1 Scenario of whole packet caching ......................... 17
2.2 An example of how the input affects the output of packet caching schemes ........................................... 19
2.3 Redundancy Function Performance Comparisons .......... 21
3.1 EEOD Operation Flow .................................... 25
3.2 Packet Aggregation ...................................... 31
3.3 Algorithm performed to determine how to dispatch aggregation window .................................................. 34
3.4 Tagging the output of a page ............................. 35
3.5 Algorithm for writev all (apr_socket_t *s, struct iovec *vec, int nvec, apr_size_t len, apr_size_t *nbytes). .......................... 40
3.6 Algorithm for writev all (apr_socket_t *s, struct iovec *vec, int nvec, apr_size_t len, apr_size_t *nbytes) with Holding-tail Mechanism. ........................................ 41
4.1 Experiment Environment .................................. 43
4.2 Effect of the number of the objects of web pages on retrieval time by connections per second. (No HTTP pipelining, no squid cache) 43
4.3 Effect of the number of the objects of web pages on retrieval time by connections per second. (With HTTP pipelining, no squid cache) 45
4.4 Effect of the number of the objects of web pages on retrieval time by RTT. (No HTTP pipelining, no squid cache). 46
4.5 Effect of the number of the objects of web pages on retrieval time by connections per second. (No HTTP pipelining, with squid cache) 47
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6</td>
<td>Effect of the number of the objects of web pages on retrieval time by RTT. (No HTTP pipelining, with squid cache)</td>
<td>48</td>
</tr>
<tr>
<td>4.7</td>
<td>Effect of the number of connections on bandwidth efficiency</td>
<td>50</td>
</tr>
<tr>
<td>4.8</td>
<td>Effect of the number of connections on retrieval time</td>
<td>51</td>
</tr>
<tr>
<td>4.9</td>
<td>Effect of the layout of web pages on bandwidth efficiency</td>
<td>52</td>
</tr>
<tr>
<td>4.10</td>
<td>Effect of the layout of web pages on retrieval time</td>
<td>52</td>
</tr>
<tr>
<td>4.11</td>
<td>Effect of the page size on bandwidth efficiency</td>
<td>53</td>
</tr>
<tr>
<td>4.12</td>
<td>Effect of page size on retrieval time</td>
<td>54</td>
</tr>
<tr>
<td>4.13</td>
<td>Bandwidth Efficiency for (A) CNN.com (B) Amazon.com (C) Slashdot webpages</td>
<td>55</td>
</tr>
<tr>
<td>4.14</td>
<td>Retrieval time for (A) CNN.com (B) Amazon.com (C) Slashdot webpages</td>
<td>56</td>
</tr>
<tr>
<td>5.1</td>
<td>Probabilities of the Occurrence of the Misalignment in 10 Consecutive File Transfers</td>
<td>62</td>
</tr>
<tr>
<td>5.2</td>
<td>Percentage of the Cacheable Content Range of Different Files</td>
<td>62</td>
</tr>
</tbody>
</table>
PREFACE

The research material in this thesis represents joint research done with my advisor, Dr. Aaron Striegel. Funding for this work was provided by NSF grant CNS03-47392.
ACKNOWLEDGMENTS

I would like to thank my advisor, Aaron, Striegel, for all of his continual support. Without his advice, feedback, and ideas, this thesis would not have been possible. Lastly, I would like to thank my family for their constant support throughout my life, especially throughout my years at ND.
CHAPTER 1

INTRODUCTION

1.1 Overview

Fundamentally, the web is a very simple client-server model as Figure 1.1 shows. The information saved at the server is organized as Web Objects, each of which has a globally unique name, Uniform Resource Locator (URL) [3], that web clients can use to retrieve it. Web clients communicate with the web server by the HTTP [10] protocol. It is this simple model that enables the web to be the most prevailing platform over the Internet. On the other hand, however, it is its simplicity that produces significant performance issues for the Internet.

1.1.1 Web Performance

Web performance usually refers to scalability and reliability that can be evaluated by two metrics: bandwidth consumption and user perceived delay, also referred to as the retrieval time. Specifically, the web performance is influenced by the following issues:

- **Network Efficiency.** First, web performance depends on the underlying network efficiency. However, the initial web model was designed for accessibility not efficiency. Specifically, as each web request reaches the origin server, both the load of the server and the load of the network are increased.
Due to its unicast nature, the web can create significant quantities of redundant data \(^1\) over the network. Those data transfers reduce the network efficiency that in turn degrade the web performance.

- **Transport Protocol.** The web typically uses TCP [15] as underlying transport protocol, which had profound effect on the web performance. Initially, HTTP [10] transfers require multiple TCP connections to retrieve a web page with multiple sub-objects. As TCP uses a slow start algorithm to avoid network congestion, each connection requires several RTTs to fully open the TCP send window. Ironically, the fact that web objects usually are of small size implies that the connection most likely be closed before the TCP send window is fully opened [20]. Hence, TCP performs poorly in terms of network efficiency and thus, significantly degrades the performance of the web. The persistent connection was introduced in HTTP 1.1 [11] to remove the overhead for multiple TCP connection establishment by maintaining only a single TCP connection for each URL. However, the client only sends the

\(^1\)Redundant data refers to the same data transferred multiple times over the same network path.
next request after obtaining the response to the previous one, which creates delays in transmission of the entire web page.

- **Synchronous Web Model.** With the classic synchronous web model, most user actions in the interface (e.g. browser) trigger an HTTP request to a web server. While the server is processing the request (retrieving data, crunching numbers, talking to various legacy systems etc.) the user has nothing to do but face a blank window and an hourglass icon. In most cases, this significantly frustrates the user’s interests in the web page.

- **User Access Pattern.** Usually, the time one user stays at one web page is less than 3 seconds, and almost 30% of connections are aborted in the middle of the transferring due to the user’s impatience [7]. To some extent, such a request-transfer-abort behavior exacerbates the network efficiency since it not only puts more pressure on the server but also creates useless traffic. Technically, the impatient access patterns of the user arose from the long retrieval time of the web page, which could disappear if other web performance issues could be solved or mitigated.

Over the past several years, various approaches have been proposed to address the web performance or scalability issues, which directly or indirectly target one or more issues noted above. Table 1.1 summarizes these approaches as well as their targeted issues.

Fundamentally, all of the performance issues are deeply rooted at both the network efficiency and the transport protocol issues, by impacting the web performance together, which directly or indirectly create problems. Thus, a solution to

---

\[2\] HTTP1.1 represents persistent connection, pipeline and parallel connections in the context of this table.
TABLE 1.1: ISSUES AFFECT WEB PERFORMANCE AND APPROACHES TO IMPROVE THEM. X: BENEFIT; O: NO BENEFIT.

<table>
<thead>
<tr>
<th></th>
<th>Web Caching</th>
<th>AJAX</th>
<th>HTTP1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Efficiency</td>
<td>x</td>
<td>x</td>
<td>o</td>
</tr>
<tr>
<td>Transport Protocol</td>
<td>x</td>
<td>o</td>
<td>x</td>
</tr>
<tr>
<td>Synchronous Web Application</td>
<td>o</td>
<td>x</td>
<td>o</td>
</tr>
<tr>
<td>User Access Pattern</td>
<td>x(indirectly)</td>
<td>x(indirectly)</td>
<td>x(indirectly)</td>
</tr>
</tbody>
</table>

improve the web performance is to improve the network efficiency and the transport protocol issues, which could be translated into improving *bandwidth efficiency* and *retrieval time*. Among a variety of approaches, the caching proxy approaches turn out to be the most prevailing and the most effective means to scale the web.

1.2 Web Caching Proxy

Caching refers to the idea of storing a copy of the information in accessing in an easily accessible place if it might be used again in the near future. In the context of web caching [1, 4, 5, 23, 28, 31], the cache is dealing with the web object, which may change over time and likely contain references to other objects such as images and advertising content, namely, sub-objects. The object as well as all its sub-objects can be stored in the cache and indexed by their *URL*. A web cache usually operates between the web client and the web server and can be deployed by an enterprise or its Internet service provider (ISP) that provides Internet connectivity to this enterprise. Figure 1.2 shows a basic scenario of a web caching proxy.

In Figure 1.2, two enterprise proxies serve two enterprise networks respectively and a client ISP proxy is deployed. A browser first attempts to satisfy requests from its local cache, then sends unresolved requests (URLs) to the enterprise
proxy, which is responsible for sending its misses to the ISP proxy, which finally forwards unresolved requests to their intended web servers.

Proxy caching offers important benefits to all parties on the web-users, ISPs, and origin web servers as well as the Internet at large. For the user, a nearby cache can often satisfy requests faster than a remote origin server. For the ISP of the user, the cache can reduce the amount of external bandwidth exchanged with other ISPs since requests satisfied by an ISP cache terminate within the ISP network. Depending on the relationship of the ISP to its peer ISPs, such bandwidth reduction can directly translate into cost savings. For the origin servers, a cache reduces the load the servers have to handle, since the servers do not have to process requests satisfied by the cache. Finally for the Internet at large, a cache offers the potential to improve scalability by reducing overall load on the network,
because a cache that satisfies a request from a nearby client leaves the rest of the Internet free to carry traffic for others.

Most importantly, caching operates in a manner that is typically transparent to the end client and/or server. At a minimum, the majority of techniques do not require modifications to the intermediate network transport (IP), thus allowing for simple deployment. Furthermore, such techniques are often incrementally deployable with an immediate and tangible benefit.

In keeping with the spirit of simple interoperability by mimicking client interactions, the majority of web caching techniques employ object-level caching [5, 28, 31]. With object-level caching, the entire object (web page, image, etc.) is cached and a cache hit is determined on an object-wise basis. When the majority of a web site content is static, an object-level cache can achieve excellent performance.

Nevertheless, proxy caching is far from perfect. For example, once a cache miss occurs, the relay of the content increases the retrieval time. For a web site with most non-cacheable content, the proxy caching is not only negated but
even introduces a penalty. Moreover, research [9] reveals that the usefulness of proxy caching is limited by low hit rates due to the fact that up to 43% of web requests are for “non-cacheable” content, which occurs for both technical and non-technical reasons [23]. In one case, content providers are reluctant to make their content cacheable due to the copyright and concerns of security. In order to prevent unauthorized accesses, such content usually is set as non-cacheable. Moreover, the web model has gradually shifted from the static model to a dynamic model for providing users with an interactive and personalized experience. In the context of a dynamic web model, content is dynamically generated at run-time or personalized according to the preference of the user. Notably, this prevents a traditional object-level cache from being effective and may produce potentially wrong responses to users in some cases. Thus, given the trend of an increasing proportion of dynamic web traffic, it is essential to deal with non-cacheable content efficiently.

1.2.1 Cacheable vs. Non-Cacheable Content

Typically, non-cacheable content comes from both static content and dynamic content. The statically non-cacheable content results from resourcing modifications and aliases. While the former refers to identical URLs can yield different reply payloads, the latter means different URLs can yield identical payloads.

The dynamically non-cacheable content is typically created by applications (database, scripts etc.) running on servers that depend on the user’s input as well as the application’s run-time status. As the Internet has grown, web sites have been increasingly serving dynamic content, which accounts for 85% of all the web traffic [16]. Currently, dynamic content is typically created for:
• **Personalization**: Rather than a one-size-fits-all approach, web sites offer dynamic/adaptive content customized to the personal preferences of the user. Through the use of *cookies* or even direct logins (portals), web sites are able to identify the preferences of an individual user and tailor the content appropriately. For example, consider the popular computing website Slashdot (slashdot.org - see Figure 1.3). While the content displayed to the end user is relatively similar, the content varies considerably on a per-object level. Notably, the addition of commentary and end user generated content (comments, journals) can create small intra-object differences between multiple web accesses. Furthermore, registration with the site allows for user-specific content (greeting, notable messages, removal of ads, etc.).

• **Information on Demand (IOD)**: IOD informally refers to getting the right information to the right people at the right time. As people are increasingly relying on Internet to get information such as news, stock information, web sites providing news and stock portals tend to update their content as soon as the information becomes available. For example, CNN.com falls into this category.

• **Advanced Searching**: In order for users to locate their interested things quickly and conveniently, sites like amazon.com often provide effective search on the site’s content to users.

The direct effect of this inclusion of non-cacheable content is a reduction in the effectiveness of caching. In general, the presence of the dynamic content in a page makes the whole page non-cacheable as the web object is no longer static. Traditional object-level cache techniques perform poorly since significant
redundancy is missed. Since non-cacheable content is only increasing, it is essential to improve web cache efficiency despite dynamic content.

1.2.2 Caching Non-Cacheable Content

One way to meet this requirement is to design the web site in a cache-friendly way for the conventional object-level caching system. An intuitive way is to encapsulate dynamic content as objects embedded in the main object. While this is useful in many situations, it falls short in two kinds of pages. For example, given a page with multiple bits and pieces of dynamic data scattered around the page, including all dynamic items in a single embedded object would make this object encompass most of the page, defeating the purpose of the technique. On the other hand, encapsulating each dynamic item into a separate object would entail multiple downloads of small objects from the server, which would increase the server load and may therefore be unacceptable. The experimental results in Chapter 4 verify this conclusion.

Therefore, simply modifying web sites to cater to the requirement of the traditional object level cache is not only painstaking, but also less effective. In essence, the pervasion of the caching proxy results from its property of being transparent, i.e. no changes to either the server or the clients. Thus, to maintain transparency but capture intra-object redundancy, techniques aimed at caching at the packet level were introduced [27, 29, 30]. Packet caching attempts to detect redundancy in an application-agnostic fashion in the packet itself rather than at an object-level. Although whole packet caching [27] is extremely scalable, it can miss significant redundancy when coupled with intra-object dynamic content. In
contrast, while dynamic boundary detection techniques [29, 30] offer the largest bandwidth savings, the techniques come with serious scalability concerns.

1.3 Motivation

It is the performance tradeoff of packet caching (scalability versus bandwidth efficiency) that introduces the motivation for this thesis. 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 thesis that a middle ground can be reached by providing a lightweight mechanism for accurate boundary demarcation of cacheable content. Specifically, this thesis posits the following questions:

- Is it possible to simply delineate dynamic content boundaries, thus improving the effectiveness of whole packet caching?

- Can one exploit bursts of contiguous redundancy to offer improved bandwidth savings and client-side performance?

1.4 Contributions

In this thesis, the concept of an Explicit End of Data (EEOD) marker is introduced. 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 this thesis include:

- EEOD Concept: The thesis 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. By means of EEOD, the whole packet caching can achieve very good bandwidth savings while keeping its scalability advantage.

- **Improved whole packet caching:** The thesis introduces an improved whole packet caching model that uses hints from EEOD combined with a novel windowed aggregation scheme. Unlike traditional packet caching which operates on a single packet, the scheme allows the cache to examine a limited width window of packets to aggregate cacheable tokens for further bandwidth and computation savings.

- **Prototype evaluation:** The thesis 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. Finally, the potentials of the EEOD in other applications is also studied.

- **Rabin fingerprinting efficiency:** This thesis is the first to highlight the poor performance of Rabin fingerprinting [22] when minimal cacheable content exists. Although Rabin fingerprinting is extremely efficient in identifying intra-packet redundancy, its computational overhead as well as cache lookup overhead make it perform poorly.

1.5 Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2, the background and fundamentals of the EEOD approach are presented. Chapter 3 presents the integration of EEOD and the Apache web server. Experimental studies are pre-
presented in Chapter 4. Finally, in addition to related work, Chapter 6 presents several conclusions and discusses future work as well as concerns about EEOD.
CHAPTER 2

EEOD BACKGROUND AND FUNDAMENTALS

To provide further background and motivation for EEOD, this chapter discusses several immediately related technologies and highlights critical weaknesses in the efficiency of Rabin fingerprinting.

2.1 HTTP 1.1: Pipelining and Chunk Encoding

Beyond the improvement of persistent connections offered with HTTP 1.1 [11], 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 [18]. 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 [21]. 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 this thesis 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 [25] are often precluded.

2.2 AJAX

Originally, Asynchronous JavaScript + XML (AJAX) [13] was proposed to improve the user’s experience. By introducing an intermediary, “AJAX engine”, between the user and the server, AJAX avoids the start-stop-start-stop nature of the interaction on the web. By enabling the asynchronous user’s interaction with the application - independent of the communication with the server\(^1\), the user is never staring at a blank browser window and an hourglass icon, waiting for the server’s response [13].

Not only does AJAX reduce the server’s load and bandwidth consumption by isolating most of user requests into local AJAX engine, the network pressure is also relieved by avoiding the establishment of extra TCP connections to the server.

---

\(^1\)Any HTTP request caused by user action takes the form of a JavaScript call to the AJAX engine, which locally handles the request that does not require a communication to the server such as simple data validation. Otherwise, if the user is submitting data for processing, or retrieving new data, the engine makes those requests asynchronously, usually using XML, without halting a user’s interaction with the application.
in some cases, which significantly improves user's experience in web applications. While AJAX could be adapted to extract the dynamic content of a web page in the context of dynamic content, AJAX requires significant change to a web page itself and hence cannot make an immediate benefit.

2.3 Base Instance Caching and Template Caching

Present dynamic web pages have much non-cacheable content scattered all over the page, as noted in section 1.2.2, which makes the encapsulation technique useless. To overcome this, two major methods, base instance approach [14] and template caching [8], are used to cache the static page portions.

Base Instance Caching [14] actually is a delta-encoding approach, by allowing the server to designate a certain base instance of the dynamic page and arrange that every client has the same base page instance, only one base instance needs to be stored at the server for delta-encoding future responses. In contrast, template caching [8] separates dynamic and static portions of the page explicitly. The static portion is augmented with macro-instructions for inserting dynamic information. Prior to rendering the page, the client expands the template according to the macro-instructions and using the bindings that are downloaded from the server for each access. The rationale behind template caching is that the client caches the template and downloads only the bindings for every access instead of the entire page.

Although these approaches significantly improve the web performance, in essence, they still fall in the category of the object-level cache. Moreover, both of them need either the support from clients or changes to server applications, which are not transparent approaches and therefore suffer from deployment issues. Thus,
approaches [17, 27, 29, 30] based on content itself have emerged, which employ the fingerprint of the content as the index of the cache instead of URLs of objects. In this way, dynamic content can be identified effectively in finer granularity, and thus achieving further improvement of web performance. Among these approaches, packet caching schemes turn out to be the most promising because of their nature of being transparent, i.e. zero-change to both the server and the client and no protocol-wise requirement.

2.4 Packet Caching and Rabin Fingerprinting Algorithm

Generally, packet caching schemes share the common idea, replacing the cacheable content in a packet (whole/partial) with a short dictionary token computed by fingerprinting algorithms between a child and a parent proxy. Figure 2.1 shows a base scenario of a packet cache, in which two clients are visiting a server and two packet caches sit in the middle of the server and the clients. When the client A accesses, both caches compute the fingerprints of the packets respectively and store the original packets as well as their fingerprints. When the packets for the client B arrive at the parent cache, two packets are replaced by two tokens computed from the last access, which are converted back to original packets at the child cache and sent to the client B. In this way, although more accurate redundancy identification can be achieved, the computation overhead introduced for tokenizing packets cannot be ignored. Furthermore, the selection of fingerprint algorithms has profound effect on the packet caching performance. This section first illustrates how the input data interacts with packet caches, and then briefly introduces two well-adopted fingerprinting algorithms. Finally, a performance comparison and analysis between fingerprint algorithms are presented as well.
2.4.1 Packet Caching Schemes

As is shown in Figure 1.3, the difference between pages for different users can be small. From the point of view of a data stream, one user’s page can be seen as an alternative version of another user’s page by slightly shifting and shuffling or removing and inserting bytes of a small size (e.g. user’s name). In this context, while the cacheable region in a packet remains unchanged, a simple change, even only one byte insertion, may prevent the whole packet caching from functioning effectively. In contrast, Rabin fingerprinting algorithm is able to identify small repeated portions in a packet. Figure 2.2 illustrates an example of the outputs of a whole packet cache and a partial packet cache for different users, in which the same content is served except the user’s name in the greeting words.

In Figure 2.2, four users consecutively access a dynamic generated web page with their name on it. At the first user’s access, a compulsory cache miss occurs, and thus both caches output the untokenized content. When another user accesses, since the length of the new user’s name is the same as the first user’s, the packets sequence remains the same except the packet containing the user’s name. In this
case, both caches achieve bandwidth savings with more savings achieved by the partial cache as it shortens the packet that contains greeting words. When the pages are created for the third and the fourth users, while the partial cache still performs fine, no bandwidth savings achieved at the whole packet cache as the input packets are entirely different from those it already cached due to the slight shift at the beginning and the modification occurred in the middle of the content. However, while the partial caching performs more effectively than the whole packet cache in terms of bandwidth savings, the gain is achieved at the expense of the scalability.

2.4.2 MD5 vs. Rabin Fingerprinting

While MD5/SHA is used for the whole packet caching scheme, with the whole packet replaced by a small token, Rabin fingerprinting is used for the partial-packet caching scheme, allowing to effectively identify the redundant portions in a packet. As MD5 is a well-known algorithm, only Rabin fingerprinting is introduced.

In a partial packet cache, the representative fingerprints for a packet are generated by computing a Rabin fingerprint for every $W$ length substring of the packet, and selecting a deterministic subset of these fingerprints. A Rabin fingerprint for a sequence of bytes $b_1, b_2, b_3, ... b_W$, of length $W$ is given by the following expression, where $p$ and $M$ are constant integers:

$$RF(b_1, b_2, b_3, ... b_W) = (b_1p^W + b_2p^{W-1} + b_{W-1}p + b_W) \mod M$$

The form of this expression makes fingerprinting each $W$ length substring $\{b_1, b_2, b_3, ... b_W\}, \{b_2, b_3, b_4, ... b_{W+1}\}, etc.$ computationally efficient. If computing the fingerprints of a window of size of $W$ over the packet from beginning to
Figure 2.2. An example of how the input affects the output of packet caching schemes
end, then at each step, the next fingerprint can be defined in terms of the previous one:

$$RF(b_{i+1, \ldots b_{W+i}}) = (RF(b_i, \ldots b_{W+i-1})) - b_i \times p + b_{W+i} \mod M$$

For fast execution, $b_i \times p^W$ can be pre-computed and stored in a table. Since $W$ and $p$ are constant, this table has 256 entries. Rather than generating a new fingerprint from scratch, advancing the fingerprint in this manner requires a subtraction, a multiplication, an addition and a mask (by $M - 1$ to perform the mod $M$ operation, where $M$ is a power of two). In essence, such a computation for every $W$ length substring of the packet is a computation on the basis of a fixed-size ($W$) sliding window.

2.4.3 Rabin Fingerprinting Efficiency

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 2.3 compares a block-wise approach (MD5 checksum) versus Rabin fingerprinting with variable sliding window sizes on non-cacheable content. In Fig. 2.3, unique packets were sent to the different algorithms, MD5 and Rabin. For the MD5 algorithm the MD5 was calculated over the whole packet. Rabin, on the other hand, does a partial match based on Windows size. The window sizes used for the performance test were 64, 128, and 256 bytes. The implementations of the algorithms are based on the prototype code of [27] and [30] and were not optimized for the underlying architecture\(^2\). Multiple runs of the tests were conducted, which included the overhead of computing

\(^2\)HP Itanium2 900 MHz workstation. Similar results were observed on other platforms.
the payload/window fingerprint, performing a cache lookup, 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 [30]). 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, 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 the experimental results, this overhead of Rabin translates to a non-trivial delay even with only 10-20% non-cacheable content.

2.5 Further Motivating EEOD

While it is desirable to use whole packet caching for scalability, separation of cacheable and non-cacheable 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, the 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.

Thus, this thesis proposes the EEOD concept that allows the content designer to easily signal boundaries between cacheable and non-cacheable content. By providing application-to-stack mechanisms, EEOD is able to guarantee the separation of packets, which enables highly scalable whole packet caching mechanism. Most importantly, EEOD does not require client-side modifications and can function in a variety of server-side/network deployment modes.

The target application for EEOD is large-scale web sites that provide dynamic, interactive, and personalized content. It is important to note that the target performance of the web server is distinctly different from that of a streaming media server (ex. RTSP over TCP or UDP) in that a web server transfers data as fast as possible in a block rather than as a stream over time. Although not specifically addressed in this thesis, EEOD could also be applied to other services with mixtures of cacheable/non-cacheable content such as e-mail. Furthermore, as will be noted later, EEOD could be directly incorporated into several of the previously noted schemes to significantly improve performance.
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. In addition to the introduction of the major components of EEOD with an overview of EEOD operations, the details of the design and the implementation of EEOD are presented in this chapter as well.

3.1 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. A new system call and the minimal modifications to the Linux kernel are proposed to support the extension.

- Improved packet cache: Unlike previous work which operates on a single packet [27], the novel improvement allows the cache to examine a limited width window of packets to aggregate cacheable tokens for further bandwidth and computation savings.
Figure 3.1. EEOD Operation Flow

- **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**: 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. A website similar to Slashdot (as noted in Figure 1.3) consisting of a large degree of non-cacheable / unique content scattered amongst redundant data (see Figure 3.1) is considered. 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’s cookies/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 string matching using string search techniques [2] for the EEOD marker. At each marker, the EEOD system call is invoked.

4. The EEOD-friendly TCP/IP 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 (no hit) 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 [27], 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.

3.2 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 infrequently changed content (i.e. cacheable) or belongs to dynamic content (i.e. non-cacheable). This section addresses the following questions:

- **What is the API for using EEOD?**
- **What effect does EEOD have on TCP?**
- **What modifications are required of the Linux kernel?**
- **How is an EEOD Tag embedded in HTML file?**
- **What are the minimum components for EEOD?**

3.2.1 EEOD System Call

The signaling API for the application to create packet separation is listed below:
// @ fd: File descriptor.
//@ flag: An indicator for cacheable/non-cacheable content

void sys_write_eod(unsigned int fd,
                    u8 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, 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, the 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 to the OS for performance.

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

### 3.2.2 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 unmodified. From the perspective of the client, there is no discernable difference nor software to deploy beyond minor changes in packet size distributions that will occur. The issue of avoiding client-side deployment is critical to making EEOD practical for a content provider to adopt.

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. As EEOD potentially generates more packets and thus imposes more risk to EEOD-generated data stream of being dropped by RED. Hence, a new packet caching technique is introduced that overcomes this penalty, but also allows for further bandwidth conservation.

---

1Section 5.1 discusses the behavioral impacts of EEOD on TCP in more detail.
3.2.3 Improved Packet Caching

As noted earlier, the core packet caching architecture is a derivative of the work in [27]. The work in [27] 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 → P|Data or P|Token). In contrast, EEOD changes such characteristics of packet cache by actively separating data stream according to application writing sequence. Thus, given the fact that the size of non-cacheable content block is likely less than the MSS of TCP, 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 [29], applied Rabin fingerprinting on the entire block at the server. While this approach does not scale well beyond the target audience of dial-up users as noted in the paper and by the experiments, it offers 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 → Data, Data). Since the packet cache is close to the source, there is a high likelihood 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 3.2). With a view of a window of $N$ packets, multiple cacheable
payloads can be tokenized in the same outgoing packet ($P_1 | Data, P_2 | Data \rightarrow P_X | T_1, T_2$ or $P_X | T_{(1,2)}$). Thus, the overhead introduced by separating packets is eliminated by aggregating multiple contiguous cacheable packets and savings the respective TCP/IP headers.

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.

Enabling the cacheable bit requires a signal to the stack from the application to indicate if the coming block is cacheable or not, which can be achieved by
simply reusing the EEOD system call. Recall the definition of the new system call:

```c
void sys_write_eod(unsigned int fd,
        u8 flag);
```

which carries two arguments. Since each calling of this system call implies the change of the type of the successive blocks, it is able to carry more information rather than only an indication of separation. By borrowing the `flag` argument (0: cacheable, 1: non-cacheable), the type of the block is able to be forwarded to TCP/IP stack and in turn the IP `ID` field could be set appropriately.

3.2.3.1 Holding and Dispatching of Cacheable Packets

Packets aggregation requires holding and dispatching operations to incoming packets. Only can cacheable packets be held for aggregation and only those cacheable packets belonging to the same connection could be aggregated together. Specifically, a dispatch is triggered by any one of the following four events:

- **AGG\_THRESH**: The maximum number of packets from the same connection held in the window. Given a connection, once the new coming cacheable packet makes the number of the held cacheable packets beyond the `AGG\_THRESH`, the packets of this connection are dispatched immediately.

- **MHT**: Maximum Holding Time, which is defined to prevent the packets from being held too long. The `MHT` timer is set to the half of the default TCP timeout time (15ms) in order to avoid influencing normal TCP behavior due to the holding.
• **FULL**: The window is full. When the window is full, the connection that has the first/earliest packet in the window would be dispatched.

• **NC**: Non-cacheable packet. Whenever a non-cacheable packet comes in, a dispatch is executed immediately.

Figure 3.3 demonstrates the algorithm. Notably, the performance of this scheme is sensitive to the holding window size. If the packets from multiple connections are well interleaved, i.e. no packet is from the same connection in the packet window, this scheme is useless since all dispatches are triggered by **FULL** events. Therefore, adaptive adjustment to the window size is proposed. Given the dispatching history, consecutive 3 dispatches aroused from **FULL** events imply that the window size is not big enough and thus the window size is doubled. Otherwise, if consecutive 3 dispatches displays that the window is occupied less than half of the size, the window size is decreased to the half of the present size. Furthermore, if most dispatches are caused by **NC** and **MHT**, the **AGG.THRESH** is set too high. Similarly, if **AGG.THRESH** causes most dispatches, it is too small. As a result, the **AGG.THRESH** is reset with an increment/decrement of 1. Finally, each change to any of such parameters flushes the history.

3.2.4 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. Notably, the method must be simple to detect but yet invisible to the client if EEOD support at the server or OS does not exist.
1: TYPE ← Pkt− > type
2: numOfPktOfSameConn ← getNumOfPkt(Pkt− > conn) {Get the number of packets that belong to the same connection in the aggregation window}
3: if TYPE == NONCACHEABLE then
4: if numOfPktOfSameConn ≠ 0 then
5: Disable MHT timer of this connection.
6: Do the aggregation.
7: Send the aggregated packet out.
8: Re-organize the packet window.
9: end if
10: Send the Pkt out.
11: else if Pkt is the last packet of the present object then
12: Disable MHT timer of this connection.
13: Do the aggregation.
14: Send the aggregated packet out.
15: Re-organize the packet window.
16: else if The window is full then
17: {Dispatch the connection that occupies the first packet in the window.}
18: Disable the MHT timer of the selected connection.
19: Do the aggregation.
20: Send the aggregated packet out.
21: Re-organize the packet window.
22: else if numOfPktOfSameConn + 1 == AGGTHRESH then
23: {Hit the threshold}
24: Disable the MHT timer of this connection.
25: Do the aggregation.
26: Send the aggregated packet out.
27: Re-organize the packet window.
28: else
29: Add this packet into the window and if no packet from the same connection exists in the window, start the MHT timer.
30: end if

Figure 3.3. Algorithm performed to determine how to dispatch aggregation window.
In order to achieve the simplicity desired while remaining transparent to the client, the EEOD marker is nothing more than 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:

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

<!--/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 3.4 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 the conversion to AJAX or the usage of sub-objects, the modifications to EEOD are extremely trivial (bound dynamic content inclusion). Notably, this is not an unnecessary burden as the appropriate calls to the database...
or cache for unique content provides a clear signal of where to demarcate data. Moreover, since the designer is in the best position to easily identify the boundary of content, why not take advantage with a trivial one-time setup cost?

3.2.4.1 EEOD Placement

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.

Since EEOD breaks data stream into small blocks and tends to generate more packets, it produces the extra transportation overhead. Given a HTTP payload of size of $N$, in which, the size of cacheable content is $C$ bytes and the size of non-cacheable content is $NC$ bytes. Suppose the physical layer header size is $PH$, the TCP/IP header size is $TH$ and the maximum payload size is $MP$. Thus, the size of the packet generated by regular server would be:

$$PH + TH + N_{bytes}, \text{assuming } N < MP.$$  
Suppose this payload is broken into 2 blocks, one is cacheable and another is non-cacheable. The total bytes generated by EEOD would be:

$$PH + TH + 16 + PH + TH + NC$$  
bytes due to the fact that the MD5 will generate 16 bytes digest. Only if:

$$PH + TH + 16 + PH + TH + NC < PH + TH + N,$$

could EEOD make benefits, which means:

$$N - NC = C > PH + TH + 16.$$  
This turns out a rule for a web designer that only if the length of cacheable contents is larger than $PH+TH+16$ bytes, should EEOD be applied to it. In case
of the Ethernet environment, the minimum size of cacheable contents block would be 70 bytes.

3.2.5 Application: Web Server

Upon receiving a request, a web server renders the output web page and places the content into a buffer for kernel using the normal system write call. However, 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 [2] 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-network packet cache to conduct whole packet caching types of operations.

3.2.6 Implementation: Apache Web Server, Red Hat Linux

The EEOD scheme was implemented through modifications to the Apache 2 web server (version 2.0.54). Kernel enhancements to include the `sys_write_eod` system call and its supporting code 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 [30] and freely available code on the web [24].
3.2.6.1 System Write Eod

When serving content to clients, Apache2 uses two system calls, *writev* and *sendfile*. The *sendfile* function is used to send an existing file directly requested by the client. Since it is less likely that an existing file contains dynamically generated content, as noted earlier, this system call is not included in EEOD. With respect to the OS, this thesis 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, *sk_buff*, by which to set a socket buffer 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 blocks 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 is defined as identical contiguous blocks yielding different network packetizations. To account for this, the Apache server was modified to dispatch buffers at multiples of *MSS* and hold if possible where packet misalignment may occur. Section 5.4 comments further on the prevalence of misalignment with large file transfers and the relevance of EEOD for other network services.

3.2.6.2 Apache Web Server

In order for a minimum extension and effect on the origin Apache server, the modification points of the Apache code are carefully selected. Since the processes
of Apache following a incoming GET request are quite complicated, to avoid delving into the detail of the Apache implementation, the modifications are applied right before the point where the writev is invoked for forwarding an application’s data to the kernel. Thus, the modifications are mostly made in the function core_output_filter:

```c
static apr_status_t core_output_filter(ap_filter_t*f, apr_bucket_brigade *b);
```

which is the final egress point of the data generated by the Apache server. In this function, data is collected from the input and organized into vectors:

```c
struct iovec {
    void __user *iov_base;
    __kernel_size_t iov_len;
};
```

Next, the data vectors are written to the kernel by the function call:

```c
static apr_status_t writev_it_all(apr_socket_t *s,
        struct iovec *vec, int nvec,
        apr_size_t len, apr_size_t *nbytes)
```

As it is responsible for writing all vectors to the kernel, the best place for accommodating the modifications is in this function. In the context of EEOD, whenever a server hits an EEOD tag, the previous read data before the tag needs to be forwarded to the kernel followed by a write_eod call to notify the kernel the end of data. In other words, EEOD requires OS support to do the partial_write. Since
```plaintext
1: CurVec ← vec, CurNvec ← nvec
2: for i = 0 to CurNvec do
3:   Search string "<! --EEOD -->".
4:   if Find then
5:     write eod {signal the kernel the boundary}
6:     writev_it_all eod {write i + 1 vectors from CurVec}
7:     CurVec ← CurVec + i {Re-organize vectors}
8:   else
9:     writev_it_all eod
10: end if
11: end for
```

Figure 3.5. Algorithm for writev_it_all (apr_socket_t *s, struct iovec *vec, int nvec, apr_size_t len, apr_size_t *nbytes).

writev_it_all is able to deal with the situation in which the data cannot be written completely, i.e., ability to re-organize vectors and calculate the size of remained data, it is the ideal location for the purpose, repeating search-partial_write-return. For keeping a uniform interface to the original Apache server, all modifications were encapsulated in this function and the original writev_it_all was cloned as writev_it_all_eod. The algorithm is described as Figure 3.5.

However, in dynamic web, with the attempt of reducing the user’s perceived delay, the server could respond the client with the partial content, once enough data has been collected, which means one call of core_output_filter does not necessarily contain all the data for a request. For example, for a block of 8k data, the writing behavior could be one write for 2k followed by another write for the rest 6k or only a single write for 8k. In the former case, as noted earlier, such a simple scanning-writing mechanism may create misalignments. To account for
if strlen(eodbuf)! = NULL then
    newvec ← eodbuf + vec
    CurVec ← newvec, CurNvec ← nvec + 1
end if
for i = 0 to CurNvec do
    Search string "<!−−EEOD−−>".
    if Find then
        write_eod {signal the kernel the boundary}
        writexit_all_eod {write i + 1 vectors from CurVec}
        CurVec ← CurVec + i {Re-organize vectors}
    else
        if < /html > is found then
            writexit_all_eod
        else
            remained_len ← len_rest {length of the rest data}
            writexit_all_eod {only write (remained_data/MSS) * MSS data}
            eodbuf ← remained_data {remained_data%MSS}
        end if
    end if
end for

Figure 3.6. Algorithm for writexit_all (apr_socket_t *s, struct iovec *vec, int nvec, apr_size_t len, apr_size_t *nbytes) with Holding-tail Mechanism.

This, a holding-tail mechanism is proposed to allow Apache only to dispatch data at multiples of MSS unless the end of the page arrives. The algorithm is described as Figure 3.6.
CHAPTER 4

EXPERIMENTAL RESULTS

In this chapter, experiments comparing the performance of EEOD and related approaches are presented. Figure 4.1 shows the experimental setup. Internet emulation is provided through a single machine running NISTNet [6] 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. 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.
Figure 4.1. Experiment Environment

Figure 4.2. Effect of the number of the objects of web pages on retrieval time by connections per second. (No HTTP pipelining, no squid cache)
Specifically, the experiments analyze performance using two metrics:

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

- **Average Retrieval Time**: This thesis computes the average retrieval 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.

### 4.1 Effect of Sub-Object Usage and Squid

To begin, this section first presents experiments that show the effect on performance of prolific sub-object usage. The base webpage contains a Slashdot-esque page with the base page and 14 small images. Figure 4.2 shows the performance with multiple levels of sub-objects and HTTP 1.1 enabled by 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 and hence the bandwidth efficiency is not shown. EEOD is included to reflect the overhead of EEOD modifications (system call, EEOD tag scan). Note 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 4.3 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 over-zealous use of sub-objects. Although HTTP 1.1 includes support for the ability to pipeline multiple object requests, the pipeline feature is minimally used in the current Internet. The Mozilla webpage notes several issues with pipeline including unknown interactions, detection of pipelining support, user-perceived rendering delays [18]. Pipelining is disabled by default on most major web servers (Apache, IIS).

Moreover, each of the previous figures use a RTT of 100 ms. As the RTT is increased (see Figure 4.4), the performance of using sub-objects decreases significantly. While the usage of a Squid cache as noted in Figures 4.5 and 4.6 reduces the delay penalty, the fact that many sub-objects are likely to be non-cacheable nullifies the gains from a Squid cache.
4.2 Synthetic Experimental Studies

This section presents experiments based on synthetically created web content in order to better isolate the performance of EEOD versus existing packet caching schemes. Specifically, this thesis is interested in characteristics such as scalability and the effect of distributions of cacheable vs. non-cacheable content (i.e. what effect does the dynamic content have on performance).

In the experiments, this thesis compared three schemes:

- **Partial Packet Cache (PPC):** The partial packet cache is based on the work in [30]. 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
Figure 4.5. Effect of the number of the objects of web pages on retrieval time by connections per second. (No HTTP pipelining, with squid cache)

the network is provided through the improved packet caching mechanism proposed in the thesis.

- **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, this thesis also conducted experimental studies with prototype code for Value Based Web Caching (VBWC) [29]. 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 [29] was able to handle only a limited connection load, thus presenting an unfair comparison of VBWC. Hence, this thesis do not include results of VBWC in the
Figure 4.6. Effect of the number of the objects of web pages on retrieval time by RTT. (No HTTP pipelining, with squid cache)

performance graphs but offer commentary on VBWC from the observations with lighter system loads.

4.2.1 Web Content Generation:

To provide a source for the experiments, a database of synthetic web pages was created. 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.

The synthetically generated data and clients had the following properties:
1. **Page Size:** Total non-image content retrieved (default is 64k to follow CNN).

2. **Non-Cacheable Content:** The percentage of the content that was non-cacheable (default is 10% based on [16]).

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 NIST-Net (default is 100 ms, 10 ms standard deviation).

6. **Update period:** The frequency at which the entire content of the web page changes (5 s).

### 4.2.2 Effect of the Number of Clients:

Figures 4.7 and 4.8 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 as sufficient client hits occur between content changes. Figure 4.7 demonstrates that although EEOD injects more packets into the network, the proposed aggregation scheme offsets the penalty of additional packets by reasonable amounts.

Most importantly, Figure 4.8 illustrates the key claim of the thesis namely that dynamic boundary detection does not scale well. As noted by the scalability experiments earlier in the thesis, the computation overhead of dynamic boundary detection (Rabin fingerprinting) comes at a significant cost. At 50 connections
Figure 4.7. Effect of the number of connections on bandwidth efficiency

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. This offers credible support to the notion that the simpler whole packet cache detection mechanism will be more scalable than dynamically determining redundancy boundaries.

4.2.3 Effect of Web Page Layout:

Figure 4.9 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 4.10 further explains the performance of PPC. While EEOD
Figure 4.8. Effect of the number of connections on retrieval time

curve increases 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.

4.2.4 Effect of Web Page Size:

Figures 4.11 and 4.12 plot the performance of EEOD versus PPC as the average page size is varied. As the page size increases, both approaches suffer slightly
Figure 4.9. Effect of the layout of web pages on bandwidth efficiency

Figure 4.10. Effect of the layout of web pages on retrieval time
Figure 4.11. Effect of the page size on bandwidth efficiency

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 content. The savings of packet aggregation is also noticeable in Figure 4.12 where the delay begins to taper off despite increasing page sizes. Table 4.1 shows how the number of packets increases with EEOD but yet the average packet size is reduced as well as the effect of packet aggregation on the number of packets.

4.3 Realistic Content Experiments

While synthetic content generation allows for variations along specific axes (amount of dynamic content, etc.), this thesis also conducted experiments using
Figure 4.12. Effect of page size on retrieval time

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Avg. Page Size</th>
<th>4kB</th>
<th>8kB</th>
<th>16kB</th>
<th>32kB</th>
<th>64kB</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEOD</td>
<td>Avg. Packet Size (bytes)</td>
<td>1376</td>
<td>1420</td>
<td>1423</td>
<td>1374</td>
<td>1283</td>
</tr>
<tr>
<td></td>
<td>Avg. # of Pkts/Conn</td>
<td>31.23</td>
<td>33.75</td>
<td>40.59</td>
<td>53.84</td>
<td>84.93</td>
</tr>
<tr>
<td></td>
<td>Avg. # of Pkts/Conn. with Pkt Aggregation</td>
<td>8.13</td>
<td>8.27</td>
<td>9.37</td>
<td>15.71</td>
<td>30.52</td>
</tr>
<tr>
<td>PPC</td>
<td>Avg. Packet Size (bytes)</td>
<td>1468</td>
<td>1491</td>
<td>1498</td>
<td>1484</td>
<td>1455</td>
</tr>
<tr>
<td></td>
<td>Avg. # of Pkts/Conn</td>
<td>29.73</td>
<td>30.1</td>
<td>37.19</td>
<td>49.62</td>
<td>79.41</td>
</tr>
</tbody>
</table>
source content from popular websites (CNN, Amazon, Slashdot) in order to better validate the comparisons between EEOD and PPC.

For the source content, the experiments 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 a personalized Apache server. For EEOD marker generation, the experiments utilized the `diff` tool to isolate where dynamic content occurs.

Figures 4.13 and 4.14 show the performance of PPC and EEOD under the three respective web site 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).

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` mech-
anism. In contrast, the dynamic content of Slashdot and Amazon is harder to identify. In the case of Slashdot, the table embedding creates problems with the simple \textit{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 the synthetic experiments, EEOD significantly improves the retrieval time in each of the cases (see Figure 4.14). 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, with correct changes to the actual source script versus the naive use of \textit{diff}, one could easily expand the gap in bandwidth efficiency and retrieval time for both the Slashdot and Amazon cases.
5.1 EEOD Considerations

Although the 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 [12]. 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. The current ongoing work involves conducting large scale simulation studies to assess this tradeoff further.

5.2 Related Work

As mentioned earlier, there are several works closely related to EEOD. To the best of the knowledge, the work in [27], 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 [30] 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 [19] to improve file system performance.

The work in [29], Value Based Web Caching (VBWC), took the work in [30] 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-indexed cache is employed in order to detect redundant transfers. While the work in DTD [17], 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 [16]. A more recent work is that of stealth multicast, described in [26]. 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 [14] and template caching [8] 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.

For several of the related works, it is important to note that this scheme, EEOD, is quite complementary. First, this thesis builds upon the whole packet cache work in [27] 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 [30], the 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, 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.

5.3 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.

5.4 Prevalence of Misalignment

Using FTP, experiments were conducted to study the prevalence of misalignment in simple file transfers. The experiments used the FTP server (vsftpd) that is included with Fedora Core 3 and transferred files of 6 different sizes (1M, 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 5.1, 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 5.2). 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.
Figure 5.1. Probabilities of the Occurrence of the Misalignment in 10 Consecutive File Transfers

Figure 5.2. Percentage of the Cacheable Content Range of Different Files
6.1 Summary

In this thesis, a bandwidth conservation architecture was presented that centered on 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, this thesis 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.

The experiments on both synthetic and real web traffic demonstrated that EEOD offers significantly improved processing efficiency in addition to further bandwidth savings. Thus, it is credible that EEOD offers a compelling new approach to improving the efficiency of dynamic web content that merits future attention.
6.2 Future Work

The work includes:

- Releasing the EEOD mechanisms as open source software.

- Gigabit scale packet caching using the Intel IXP.

- Long term studies on the university tap regarding cacheable versus non-cacheable content. As the web evolves, it would be interesting to mine the characteristic of web transactions with the presence of EEOD.

- Exploring EEOD’s potentials in pervasive file sharing applications. Since the file sharing application is becoming a dominant contributor to Internet traffic, it is necessary to explore the EEOD’s potentials in this domain.
BIBLIOGRAPHY


15. Information Sciences Institute, University of Southern California. Transmission control protocol. RFC 793, 1981.


