Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation

Anindya Datta
Georgia Institute of Technology
Atlanta, GA 30332
adatta@cc.gatech.edu

Kaushik Dutta
Georgia Institute of Technology
Atlanta, GA
gte314q@prism.gatech.edu

Helen Thomas
Carnegie Mellon University
Pittsburgh, PA
hthomas@andrew.cmu.edu

Debra VanderMeer
Georgia Institute of Technology
Atlanta, GA
deb@cc.gatech.edu

Suresha
Database Systems Lab
Indian Institute of Science
Bangalore, India
suresha@csa.iisc.ernet.in

Krithi Ramamritham
Indian Institute of Technology-Bombay
Powai, Mumbai, India
krithi@iitb.ac.in

ABSTRACT

As Internet traffic continues to grow and web sites become increasingly complex, performance and scalability are major issues for web sites. Web sites are increasingly relying on dynamic content generation applications to provide web site visitors with dynamic, interactive, and personalized experiences. However, dynamic content generation comes at a cost – each request requires computation as well as communication across multiple components.

To address these issues, various dynamic content caching approaches have been proposed. Proxy-based caching approaches store content at various locations outside the site infrastructure and can improve Web site performance by reducing content generation delays, firewall processing delays, and bandwidth requirements. However, existing proxy-based caching approaches either (a) cache at the page level, which does not guarantee that correct pages are served and provides very limited reusability, or (b) cache at the fragment level, which requires the use of pre-defined page layouts. To address these issues, several back end caching approaches have been proposed, including query result caching and fragment level caching. While back end approaches guarantee the correctness of results and offer the advantages of fine-grained caching, they neither address firewall delays nor reduce bandwidth requirements.

In this paper, we present an approach and an implementation of a dynamic proxy caching technique which combines the benefits of both proxy-based and back end caching approaches, yet does not suffer from their above-mentioned limitations. Our dynamic proxy caching technique allows granular, proxy-based caching where both the content and layout can be dynamic. Our analysis of the performance of our approach indicates that it is capable of providing significant reductions in bandwidth. We have also deployed our proposed dynamic proxy caching technique at a major financial institution. The results of this implementation indicate that our technique is capable of providing order-of-magnitude reductions in bandwidth and response times in real-world dynamic Web applications.

ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA

Categories and Subject Descriptors

H.3.4 [**Information Systems**]: Systems and Software—*Distributed systems, Performance evaluation (efficiency and effectiveness*; H.3.5 [**Information Systems**]: Online Information Services—*Web-based services*

General Terms

Design, Performance

Keywords

Edge Caching, Dynamic Content, Proxy-based Caching

1. INTRODUCTION

To provide visitors with dynamic, interactive, and personalized experiences, web sites are increasingly relying on dynamic content generation applications, which build Web pages on the fly based on the run-time state of the Web site and the user session on the site. But, these benefits come at a cost – each request for a dynamic page requires computation as well as communication across multiple components inside the server-side infrastructure.

Caching is a widely-used approach to mitigate the performance degradations due to WWW content distribution and

delivery. Here, content generated for one user is saved, and used to serve subsequent requests for the same content.

In general, there are two basic approaches: *back-end caching* and *proxy-based caching*, which we discuss in detail in Section 3. Back-end caches typically reside within a site, and cache at the granularity of a *fragment*, i.e., a portion of a Web page. This type of cache works with a dynamic content application to reduce the computational and communication resources required to build the page on the site, thus reducing server-side delays. As we will describe in detail in Section 3, back-end caching solutions do not rely on URLs to identify cached content (as is the case with proxy-based solutions), and thus guarantee correctness of the contents in a generated page. However, this type of solution does not reduce the bandwidth needed to connect to the server to obtain content. In contrast, proxy-based caches typically store content at the granularity of full web pages [1], and reside outside the site's infrastructure. As we will explain in detail in Section 3, this type of caching can provide significant bandwidth savings, both in the site's infrastructure as well as on the WWW infrastructure; however, it suffers from two major drawbacks: (1) full-page dynamically generated HMTL files generally have little reusability, leading to low hit ratios; and (2) cache hits are determined based on a request's URL, which does not necessarily uniquely identify the page content, leading to the possibility of serving incorrect pages from cache.

In this paper, we explore whether it is possible to *achieve the benefits of both approaches, without the limitations*. The holy grail of dynamic content caching is the ability to cache dynamic content at finer granularities outside the site's infrastructure. Such an approach would provide the benefits of caching finer granularities of content (e.g., greater reusability), while simultaneously achieving the benefits associated with proxy-based caching (e.g., reduced bandwidth, reduced firewall processing). In this paper, *we propose an approach for caching granular proxy-based dynamic content that combines the benefits of both approaches, while suffering the drawbacks of neither*. We describe our approach for caching granular proxy-based dynamic content in Section 4. Specifically, we describe an architecture for such a system, as well as the data structures and algorithms needed to make it work. Based on this approach, we describe the implementation of a dynamic content caching system, which is currently in commercial deployment at a major financial institution.

We show the effectiveness of our system by studying its performance analytically and experimentally. Section 5 describes our analysis, and the corresponding results, which indicate that our approach is capable of providing significant reductions in bandwidth on the site infrastructure: more than 70% savings in bytes transmitted through the network. In Section 6, we present experimental results which validate our analytical findings.

This paper outlines the science behind, as well as an implementation of, a true proxy-based dynamic content caching system. Specifically, our architecture and implementation describe a dynamic content caching system operating in *reverse proxy mode*, providing significant bandwidth savings within the site infrastructure. The next step, moving the proxy out to the edge of the network in *forward proxy mode* would provide bandwidth savings beyond the site infrastructure, between the Web site and the edge of the network. By placing content on forward proxies, end users would also see substantial response time improvements, since content would be delivered from points close to them on the network. However, there are significant technical challenges associated with moving dynamic content to forward proxies. Section 7 outlines the issues that remain open in order to take dynamically generated content to the edge.

[1]One class of proxy-based solutions which cache content at finer granularities is *dynamic page assembly* solutions. We discuss dynamic page assembly in detail in Section 3.

2. DYNAMIC CONTENT GENERATION: BACKGROUND AND PRELIMINARIES

Over the past few years, Web sites have transitioned from a *static* content model, where content is served from ready-made files, to a *dynamic* content model, where content is generated on demand. Dynamic content generation allows sites to offer a wider variety of services and content. For instance, a Web page can be customized according to an individual's preferences, perhaps displaying the user's preferred stock quotes and a personal greeting.

A broad range of technologies are available to support such dynamic content generation. For instance, application servers (e.g., BEA's WebLogic [30] and IBM's WebSphere [17]) are commonly used to handle page generation tasks and manage connections to back-end services, such as DBMSs and content management systems (CMSs). Application servers run *dynamic scripts* or programs to generate Web pages. These scripts can be written in a number of languages including Sun's Java Servlets and Java Server Pages (JSP) [22], the Active Server Pages (ASP) family from Microsoft [21], and Perl [24].

At a high level, dynamic scripting works as follows. A user request maps to an invocation of a script. This script executes the necessary logic to generate the requested page, which involves contacting various resources (e.g., database systems) to retrieve, process, and format the requested content into a user deliverable HTML page.

2.1 Dynamic Layouts

Consider a Web site that caters to both *registered* users (i.e., users who have set up an account with the site) and *non-registered users* (i.e., occasional visitors). Suppose the site allows registered users to create a user profile, which specifies the user's content preferences and allows him to control the layout of the page. Here, pages contain a number of elements or *fragments*. For each request, the Web site lays out the fragments on a page in a specific default configuration for non-registered, and based on a user profile for registered users.

In general, an HTML page consists of two distinct components: *content* and *layout*. Content refers to the actual information displayed and layout refers to a set of markup tags that define the presentation (e.g., where the content appears on the page). Loosely speaking, the different fragments on a page represent content, whereas the layout determines how the fragments are presented on the user viewable page. Here, the final presentation of the page is partially determined by the order in which the markup tags appear in the page, as well as the actual markup tags themselves (e.g., $\langle HR \rangle$, which adds a horizontal rule).

The foregoing discussion highlights two important characteristics of dynamically generated content. First, *not only*

is the content of many sites dynamic, but also the page layout. In other words, the precise organization of a page is often determined at run-time. Second, and most important, *the same request URL can produce different content and/or different layouts.* The registered and non-registered users submit the exact same URL to the site, yet they may receive very different pages. This fact is very important, and as we will show, one of the major impediments to caching dynamic pages in a proxy cache.

2.2 Performance Bottlenecks in Serving Dynamic Web Pages

Having described how a dynamic Web page request is served, we now discuss the potential bottlenecks in this process. These bottlenecks can be classified into two broad areas: (a) network latency, i.e., delays on the network between the user and the Web site, and (b) server latency, i.e., delays at the Web site itself.

2.2.1 Network Latency

Typically, long distances separate users and Web sites. Furthermore, content that is delivered over the Internet must go through an extensive network of transmission and switching devices (e.g., routers, switches). Each such device is a potential source of delay. The larger the size of the content, the greater the network delay. Various caching solutions have beeen proposed to mitigate network delays, which will be discussed in Section 3.

2.2.2 Server Latency

After a user's request traverses the Internet and arrives at the Web site, a number of Web site infrastructure delays can occur, and these delays can be significant. Delays at the Web server can be broadly classified into two categories: (1) *session processing delays*, and (2) *dynamic content generation delays*. Web server session processing delays occur because once a request arrives at the Web site, it must traverse several hardware and software layers, a router, a firewall and a switch, before reaching the Web server. Forcing a user's request through these devices, each of which has a finite throughput, can expose network performance bottlenecks. With today's Web pages containing an average of 10-20 objects, the sheer number of trips through the Web site's infrastructure creates significant latency [26]. Furthermore, as more and more users try to access the same content, the redundant load on the firewalls and switches for the same objects increases dramatically. Caching solutions have also been proposed to address server delays, which will be discussed in Section 3.

Content generation delays occur as a result of the work required to generate a Web page. In the case of static Web sites, content generation involved accessing the appropriate response file from a file system. Thus, generation delays are negligible in this case. However, in the case of dynamic sites, the story is completely different. As mentioned previously, dynamic site requests are processed by an *application layer* consisting of application servers and other back end system components such as DBMSs. Due to the complexity of modern Web site application layers, sites are increasingly employing a layered or *n-tier* application architecture, which partitions the application into multiple layers. For instance, the *presentation layer* is responsible for the display of information to users and includes formatting and

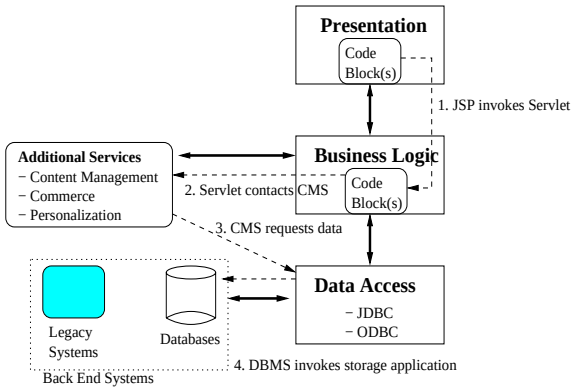


Figure 1: Example of Workflow Required to Generate Dynamic Page

transformation tasks. Presentation layer tasks are typically handled by dynamic scripts (e.g., ASP, JSP). The *business logic layer* is responsible for executing the business logic, and is typically implemented using component technologies such as Enterprise Java Beans (EJB). The *data access layer* is responsible for enabling connectivity to back-end system resources (e.g., DBMSs), and is typically provided by standard interfaces such as JDBC or ODBC. Such multi-layered architectures have become widely accepted. For instance, most Java-based Web applications follow the *Model View Controller* (MVC) [16] design paradigm. In this paradigm, presentation logic is handled by JSPs, and business logic is controlled by Servlets, which in turn invoke the appropriate business components (EJBs).

To better illustrate how multi-layered architectures serve requests, consider Figure 1, which illustrates how part of this request may be served. As this figure shows, the following steps are required to serve a request:

1. The application server executes the **JSP** script. The **JSP**, running in the presentation layer, invokes a Java **Servlet** in the business logic layer.
2. The **Servlet** contacts a **content management system (CMS)** (e.g., Vignette [10]) to run personalization logic.
3. The **CMS** requests data from the **DBMS**. This data may include, for example, the names, descriptions, and images associated with the *Fiction* category, as well as user profile information (assuming that the user has registered with the site). This request invokes **connectivity software** (e.g., JDBC), in the data access layer, which waits for a connection to the **DBMS**.
4. The **DBMS** invokes **storage applications**. These **storage applications** may, in turn, make calls to a file system (not shown).

As the above example illustrates, serving a request for a dynamically generated page typically involves nested task invocations across multiple application layers. This process can incur several types of delays, including:

- **Computational Delays**. This type of delay is the result of executing various types of logic (e.g., query processing). Note that this delay can occur at multiple layers.

- **Interaction Bottlenecks**. This type of delay occurs when a request must wait for a resource, such as a connection to a DBMS.
- **Cross-tier Communication**. This type of delay occurs as a result of the network communication required between application components. For each invocation, communication between the two components requires network protocol support in the connectivity software (e.g., JDBC), which traverses a network protocol stack (e.g., TCP/IP).
- **Object Creation and Destruction**. This type of delay is common in object-oriented applications, which must repeatedly create and destroy objects.
- **Content Conversion**. This type of delay is a result of data transformations (e.g., XML-to-HTML) and/or formatting tasks.

Each of these content generation delays contributes to the end-to-end latency in delivering a Web page. As user load on a site increases, the site infrastructure is often unable to serve requests fast enough. The end result is increased response times for end users.

In summary, the performance bottlenecks in serving dynamic content are of two main types: network latency and server latency. Server latency is further composed of session processing delays and dynamic content generation delays. In the next section, we will discuss existing caching approaches that attempt to mitigate all of these types of delays.

3. EXISTING APPROACHES AND THEIR LIMITATIONS

A widely used existing approach to address WWW performance problems is based on the notion of content caching. A variety of such methods exist. For a treatment of Web caching techniques, refer to [23] - this work includes many references, which we do not repeat in this paper. These caching approaches can be classified as *back end* and *proxy-based* caching solutions. We discuss each in turn.

3.1 Back-end Caching Approaches

Back end caching approaches have been proposed to accelerate dynamically generated content. These approaches are based on the idea of caching content at the various layers within the site architecture. For instance, various types of database caching have been suggested, including caching the results of database queries [20] and caching database tables in main memory [28, 9]. Database caching approaches can reduce some of the delays associated with query processing operations. Another back end approach is *presentation layer caching*, which caches HTML fragments. Many application servers provide this type of caching capability (e.g., WebLogic from BEA Systems [30]), which can mitigate delays due to presentation layer tasks. Solutions from vendors such as SpiderCache [29] take the approach of caching dynamically generated pages within the site infrastructure. These solutions are similar to reverse proxy caches, except that they operate within the site infrastructure, and are typically implemented as plug-ins to the Web server. A recent work proposes a back end caching system that caches content at various levels, such as database queries, HTML fragments, and pages [34]. Another more general back-end caching approach is *component level caching* [14, 13], which caches arbitrary objects, including HTML fragments and programmatic objects. This approach addresses delays due to computation as well as delays due to communication between different modules, and is available commercially as a software solution from Chutney Technologies [32].

All of the above-mentioned back end caching approaches can help reduce the delays associated with generating content. Also, since they reside at the back-end, these solutions do guarantee the correctness of the output, unlike proxy-based caches. Finally, by caching at finer granularities, these solutions achieve greater reuse of content and allow fine-grained invalidation. A limitation of back-end caching solutions, however, is that they deliver all content from the dynamic content application itself, and thus do not address network-related delays, i.e., delays resulting from the need to transmit high-bandwidth content through the site and WWW infrastructures (e.g., firewall processing delays, routing delays).

3.2 Proxy-based Caching Approaches

Proxy-based caching approaches are based on caching content outside the site's infrastructure. Such content can include static content such as media files (pictures, audio, video) or dynamically generated HTML pages. The utility of using proxies to cache static content is well-known and is not the focus of this paper. Rather, we wish to study the usefulness of using proxies to cache the output of dynamic Web sites. Two broad approaches exist in using proxies to cache dynamic pages: *page-level caching* and *dynamic page assembly*.

3.2.1 Page-level Caching

In this approach, the proxy caches full page outputs of dynamic sites. This approach has been considered in the literature, e.g., [6, 5] propose page-level caching techniques. A number of commercially available solutions are based on this approach. Some operate in *reverse proxy* mode, sitting between the site and the Internet cloud (usually just outside the site firewall – a reverse proxy resides between the site firewall and the Internet cloud.), and relieving the site infrastructure from the work required to push responses through the site. Software solutions include Inktomi's Traffic Server [8] and Internet Security and Acceleration (ISA) Server from Microsoft [27], while hardware solutions are available from vendors such as CacheFlow [4] and Network Appliance [3]. Other solutions are deployed in *forward proxy* mode, i.e., in distributed caching architectures located at numerous points around the Internet. These solutions are based largely on the fundamental body of work that addresses distributed proxy caching, e.g., [15, 25]. These solutions, also known as *Content Delivery Networking* (CDN) solutions, are primarily service-based solutions offered by vendors such as Akamai [31] and Digital Island [1].

In general, page-level caches can improve web site performance by reducing (a) delays associated with generating the content, (b) delays associated with packet filtering and other firewall-related delays, and (c) the bandwidth required to transmit the content from the back end application to the proxy-based cache.

There are, however, three major limitations associated with using page-level caching solutions to cache dynamic pages. First and foremost, page level caching solutions must

rely on the request URL to identify pages in cache. When pages are dynamically generated, different invocations of a given script, even with the same input parameters, are not guaranteed to produce the same page. Consider a Web site that greets registered users on each page. Suppose a registered user, whom we will call Bob, requests a page. The page displayed to Bob will include a "Hello, Bob" greeting. Suppose a subsequent user, whom we will call Alice, requests the same page (using the same request URL). Alice is not a registered user on the site, so she should receive the page without a greeting. However, if the site is using a proxy cache, Alice will be served the page that was just served to Bob, since this page matches the request URL. Thus, as this example illustrates, proxy-based caches may serve incorrect pages. This problem, traditionally, has prevented the use of proxies in caching dynamic pages.

Another limitation of page level solutions is that there is often very little reusability of full HTML pages. Sites that serve highly personalized pages, for instance, may include a customer greeting on every page, thus making every page instance unique, and reusable only if the same user makes the same request. This can lead to low hit ratios, in which case few, if any, of the benefits of caching are actually realized.

Furthermore, caching at the page level causes unnecessary invalidation. If only one or a few elements on a page become invalid, then the entire page becomes invalid. Consider, for example, a stock quote page on an online brokerage site. Suppose that, given a ticker symbol as input, the output page consists of three basic content elements: a current price quote, a set of recent headlines for the company, and historical research data (e.g., price to earnings ratio). Clearly, price quotes become invalid relatively quickly (perhaps within seconds), while news headlines might be updated every thirty minutes and historical data on a monthly basis. In this scenario, the page cache would invalidate cached pages as the price quotes become invalid, thus leading to the regeneration of the news headlines and historical content elements at a much greater frequency than the frequency at which they change.

3.2.2 Dynamic Page Assembly

Dynamic page assembly is an approach popularized by Akamai [31] as part of the Edge Side Includes (ESI) initiative [7], and found in other products, such as IBM's trigger monitor feature (available as part of WebSphere Edge Server version 2.0 [11]). This approach entails establishing a template for each dynamically generated page. The template specifies the content and layout of the page using a set of markup tags. Essentially, each page is factored into a number of fragments (specifically, separate dynamic scripts) that are used to assemble the page at a network cache when the page is requested. Content generated from templates and factored fragments are cacheable as separate HTML files on distributed caching architectures; here, responses are actually assembled at these distributed caching locations around the Internet, rather than hitting the origin server. By moving the dynamic content closer to the user, many of the same benefits of page-level caching accrue, with the additional benefit of further reduced response times and network bandwidth requirements (since content need not be delivered from the origin Web site).

There are two major limitations associated with the dynamic page assembly approach. A key drawback is the requirement that a site follow a specified page design paradigm, specifically, the use of templates which in turn call separate dynamic scripts for each dynamically generated fragment. This requires that page layout be known in advance. The problem with this scenario is that these caches base response decisions on the requested URL; once the template and fragments for one of the two results in this example are present in the cache, every request for a particular URL will be served from cache, regardless of whether the template in cache would produce the same output page as the dynamic scripts on the Web site. Thus, sites supporting dynamic layouts (most sites) will not be able to take advantage of dynamic page assembly. In addition, the use of templates and fragments is a major departure from the standard Model-View-Controller design paradigm used in many Web sites; thus, utilizing this new design paradigm may require redesigning and rebuilding a Web site from the ground up.

Another, equally significant drawback of the dynamic page assembly approach is that it cannot be used in the context of pages with semantically interdependent fragments. Indeed, it turns out that if there exist dependencies among the fragments of a page, it may be difficult, if not impossible, to factor the page into fragments. For example, a dynamic script may contain the following execution sequence: (1) query the user profile repository based on the visitor's userId obtained at login and generate a *user profile object*; (2) based on the user profile object, generate a **Personal Greeting** fragment; and (3) based on the user profile object, generate a **Recommended Products** fragment for the user. Here,*two of these fragments, the Personal Greeting and the Recommended Products fragments, are dependent on the same user profile object.* If these two fragments were processed in sequence in the script, it might be possible to combine them into a single fragment (i.e., generated by a single script in dynamic page assembly). However, other processing may occur between the generation of these two fragments, processing which may or may not result in another cacheable fragment. In either case, factoring this script into fragments would require the same call to the user profile repository to be repeated for both the **Personal Greeting** and **Recommended Products** fragments. Clearly, the repetition of the same call in generating a single page is redundant. As a result, using a dynamic page assembly approach to cache this page would result in significant repetition of work on the site. The corollary is that these approaches are optimal only for pages that can be easily decomposed into a small number of static, independent fragments, and where the overall layout of the page does not change.

A recent work that can be considered a dynamic assembly approach is [19]. This work proposes a proxy cache that stores query templates, along with query results, which are used to manage the cache. While this approach can mitigate some delays associated with query processing, it does not address the numerous other delays associated with dynamic content generation.

3.2.3 Summary of Proxy-based Caching Approaches

In summary, proxy caching approaches, whether based on full-page caching or dynamic page assembly, when they work, are able to generate significant bandwidth savings by serving the request from the proxy rather than having to route it through the origin Web site infrastructure. How-

ever, due to the many limitations discussed above, their applicabilty in caching dynamic pages is rather limited and their primary use is in caching static content or fixed layout content that can be factored into some combination of static fragments.

3.3 Summary of Existing Approaches

In summary, we note a strong dichotomy between the two above-mentioned approaches. Proxy-based approaches, when effective, can provide tremendous bandwidth savings by serving content from points outside the site's infrastructure. However, these approaches suffer from several severe limitations in the context of dynamic sites which limit their effectiveness in practice, leading to the loss of the benefits of caching. Back end approaches mitigate these issues, yet cannot provide any bandwidth benefits. The "Holy Grail" of content caching has been a solution that can provide *both bandwidth savings, like proxy-based approaches, as well as server-side acceleration, like the back-end approaches discussed above.*

In this paper, we propose an approach for granular proxy-based caching of dynamic content that combines the benefits of proxy-based caching with those of back-end caching techniques, while attempting to minimize their limitations. Unlike existing dynamic page assembly techniques, our approach supports dynamic page layouts and thus does not require that a particular site design be enforced.

4. DYNAMIC PROXY-BASED CACHING APPROACH

In this section, we describe our proposed approach for granular proxy-based caching of dynamic content. We first discuss the intuition behind our approach, followed by the architecture and technical details.

4.1 Intuition

Our objective is to deliver dynamic pages from proxy caches. Recall that dynamic pages are "dynamic" across two dimensions: they posess dynamic content and dynamic layout. Any dynamic content caching system must account for both - in fact, the primary weakness of existing proxy caching schemes arises from their inability to map a URL to the appropriate content and layout. To mitigate this weakness, our essential intuition may be summarized as follows: we will cache dynamic content fragments in the proxy caches, but the layout information would be determined, on demand, from the source site infrastructure. In other words, we propose to respond to a dynamic page request, R_i, as follows. We will route R_i through a dynamic proxy, D_i, to the site infrastructure. Upon reaching the site infrastructure, R_i will cause the appropriate dynamic script to run. A back end module will observe the running of this script and determine the layout of the page to be generated (the actual process is much more complicated, and will be described in greater detail subsequently in this section). This layout, which will be much smaller than the actual page output, will be routed to the proxy D_i. The proxy will fill in the content from its cache and route it to the requestor.

For example, consider a request for a `Category` page on an e-commerce site. The request passes through the dynamic proxy cache, which routes it to the origin site. At the origin site, the application server executes the `category.jsp` script to serve this request. A monitor at the back end observes the application processing and generates the page layout accordingly. This layout will contain "holes" to indicate where the cached fragments should be inserted. This layout is sent to the dynamic proxy cache, which fills in the "holes" with the appropriate fragments from its cache. The resulting page is then delivered to the user.

This high-level example raises many questions about the details of our dynamic proxy caching system. For instance, how does the monitor at the back end determine the page layout? How does the monitor know which fragments are in the dynamic proxy cache? How is the dynamic proxy cache managed? In the remainder of this section, we answer these types of questions by explaining the details of this system.

4.2 System Architecture

The *Dynamic Proxy Cache* (DPC) stores dynamic fragments outside the site infrastructure and assembles these fragments in response to user requests. Note that the DPC can also cache other types of content as well (e.g., rich content, static fragments). However, in this paper, we focus on the novel aspects of our approach – the ability to handle dynamic content and dynamic layouts. The ability to support dynamic layouts is enabled by the *Back End Monitor* (BEM). The BEM resides at the back end and generates the layout for each request. This layout is passed back to the DPC, which assembles the page that is returned to the requesting user. As we will soon show, this approach enables significant reductions in bandwidth requirements, since only the page layouts and perhaps some content, are transmitted from the back end to the DPC.

The DPC can reside either (a) at the origin site (in a reverse proxy configuration), or (b) at the network edge (in a forward proxy configuration). In the former case, the primary benefit is the reduction in the number of bytes transferred through the site infrastructure for each request.

In the latter case, the forward proxy configuration (similar to that of present-day CDNs), the benefits are even greater – the reduction in bytes transferred for each request is realized not only within the site infrastructure, but also across the Internet. The basic underlying technical issues are the same for both the reverse proxy as well as the forward proxy configurations. The main difference between the two is that a forward proxy configuration typically would mandate a distributed cache architecture, whereas a reverse proxy configuration is a logically single unit. Thus, two issues arise in the forward proxy case that are not present in the reverse proxy case: (1) *request routing*, and (2) *cache coherency*.

Request routing refers to the problem of determining which dynamic proxy should service an incoming request. This problem has been studied extensively in the context of CDNs, which focus primarily on routing requests for static files (e.g., image files), where a file is uniquely identified by its URL. A key difference between request routing for CDNs and for our system is the nature of the content. Our system must address the issue of routing requests for dynamic fragments. Clearly, routing that is based on URL is not applicable in our case since page fragments cannot be determined from the URL. Given that multiple copies of fragments may exist in the dynamic proxies, the issue of *cache coherency* arises. When changes to source data cause a fragment to become invalid, some mechanism must be in place to ensure that all dynamic proxies are aware of this change so that all

serve the correct version of the fragment.

Our commercially implemented solution incorporates techniques which address the above-mentioned issues and thus can be configured either as a forward proxy or as a reverse proxy. Due to the conciseness and space requirements of this paper, our subsequent treatment will assume a reverse proxy configuration. Note however, that all the technical issues would apply, virtually unchanged, to the forward proxy case.

Having described the system architecture, we now delve into the technical details.

4.3 Technical Details

Our dynamic proxy caching system consists of two main phases: (a) system initialization, and (b) run-time operation. In this section, we discuss these two phases, followed by an in depth discussion of the system components.

4.3.1 System Initialization Phase

A prerequisite of our dynamic proxy caching system is that the cacheable fragments be identified and marked. This is an initialization activity which we refer to as *tagging*. The tagging process enables page layouts to be determined dynamically at run-time.

Once the cacheable fragments are identified, each of the corresponding code blocks in the script is tagged. Tagging essentially means marking a code block as cacheable. This is done by inserting APIs around the code block, enabling the output of the code block to be cached at run-time. The tagging process assigns a unique identifier to each cacheable fragment, along with the appropriate metadata (e.g., time-to-live).

4.3.2 Run-Time Operation

At run-time, a user submits a request to the site. This request, e.g., `http://www.booksOnline.com/catalog.jsp?categoryID=Fiction` is passed through to the application server. This causes the `catalog.jsp` script to be invoked with the parameter name-value pair `categoryID-Fiction`. The application logic in the script runs as usual, until a tagged code block is encountered. When such a code block is encountered, a check is made to see whether the fragment produced by that code block exists in the DPC. This is done by looking up the fragmentID in the BEM's *cache directory*. The cache directory will be described in detail in the next section. For now, it is sufficient to know that the cache directory contains the fragmentIDs and additional metadata for each fragment in the DPC.

When a request is made, there are two general cases possible:

1. **The fragmentID is not in cache or is in cache but invalid.** In this case, an entry is inserted into the cache directory for this fragment, the content is generated, and a `SET` instruction is written to the page template. This instruction will insert the fragment into the DPC.

2. **The fragmentID is in cache and is valid.** In this case, a `GET` instruction is written to the page template. This instruction will retrieve the fragment from the DPC.

At run time, a cache directory lookup is done for the `nbKey` fragmentID. If the fragmentID is not found or is invalid, an entry is inserted into the cache directory. Details of this process will be described in the next section. For now, it is sufficient to know that the BEM assigns a key that is used by the DPC. The corresponding code block executes to generate the content, which is then written to the template, along with a `SET` instruction. If the fragmentID is found in cache, only the key and a `GET` instruction are written to the template. Similar processing would be done for the remaining cacheable code blocks.

For the first request for a given page, none of the fragments will be in cache, so the layout will consist of `SET` instructions, along with the generated content. For subsequent requests, the cacheable fragments will likely be cached, assuming that they have not been invalidated. In this case, the layout will consist mostly of `GET` instructions and hence will be much smaller. Having described the run-time operation of our system, we are now ready to discuss the system components in greater detail.

4.3.3 System Components

In this section, we provide a detailed explanation of the two main components of our dynamic proxy caching system, the Dynamic Proxy Cache (DPC) and the Back End Monitor (BEM).

The DPC is a proxy cache that stores dynamic fragments and assembles these fragments on demand using run-time page layout instructions. The DPC assembles pages by following the instructions provided by the BEM (to be described in more detail later in this section). All cache management functionality for the DPC is handled by the BEM as well. The structure of the DPC cache is straightforward: it is implemented as an in-memory array of pointers to cached fragments, where the `DpcKey` serves as the array index.

The BEM resides at the back end and has two primary functions: (1) managing the cache for the DPC, and (2) caching intermediate objects. We proceed to describe each of these functions.

Managing the DPC cache is a critical function of the BEM. This function is enabled by the *cache directory*, a critical data structure contained in the BEM. The cache directory keeps track of the fragments in the DPC and their respective metadata. The cache directory has the following basic structure:

`fragmentID`	unique fragment identifier (`name+parameterList`)
`dpcKey`	unique fragment identifer within the DPC
`isValid`	flag to indicate validity of fragment
`ttl`	time-to-live value for fragment

The `dpcKey` is a unique integer identifier associated with each fragment that serves as a common key for both the BEM and the DPC. There are two reasons why we use this `dpcKey`. First, it reduces the tag size. The fragmentIDs described in the previous section are typically quite long, especially those that include a list of parameters. By assigning an integer, we are able to reduce the size of the page templates that are sent to the DPC. Second, as we will soon show, assigning a common key eliminates the need for explicit communication between the BEM and the DPC.

There are two basic ways in which fragments can become invalid: (a) an invalidation policy determines that a fragment is invalid, or (b) a replacement policy determines that a fragment should be evicted from cache. A cache invalida-

tion manager monitors fragments to determine when they become invalid. Fragments may become invalid due to, for instance, expiration of the `ttl` or updates to the underlying data sources. A cache replacement manager monitors the size of the cache directory and selects fragments for replacement when the directory size exceeds some specified threshold.

In any case, the fragment's `isValid` flag will be set to `FALSE` to indicate that it is no longer valid. When this occurs, the `dpcKey` for the fragment is inserted at the end of a list of available `dpcKeys`. This technique ensures that a subsequent request for the fragment will be generated and served fresh.

Note that the size of the list of available keys should be at least as large as the maximum cache size. This is due to the fact that invalid fragments are not explicitly removed from the DPC. Rather, the slots corresponding to these fragments simply remain unused until they are subsequently assigned to a new fragment by the BEM. For example, suppose a fragment F_i, having `dpcKey` 2, becomes invalid. It is marked as such by the BEM and "2" is inserted back into the freeList. No action is taken by the DPC. Eventually, `dpcKey` 2 will be assigned to a fragment (either F_i or a new fragment) by the BEM, at which time the appropriate content will be inserted into the corresponding slot in the DPC.

Having described the technical details of our proposed approach, we now examine the benefits of this approach. In the next section, we present an analysis that attempts to quantify these benefits.

5. ANALYTICAL RESULTS

There are two types of benefits that accrue in our model: (a) performance and scalability of the server side, and (b) bandwidth savings. In this section, we analyze these benefits. Due to space limitations, we only present the results of our bandwidth savings analysis. Table 1 contains the notation to be used throughout this section.

Symbol	Description
$\mathcal{E} = \{e_1, e_2, \ldots, e_m\}$	set of fragments
$\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$	set of pages
$E_i = \{e_j : e_j \in c_i\}$	set of fragments corresponding to page c_i
s_{e_j}	average size of fragment e_j (bytes)
g	average size of tag (bytes)
f	average size of header (bytes)
h	hit ratio, i.e., fraction of fragments found in cache
R	total number of requests during observation period

Table 1: Notation

In our analysis, we wish to compare the bandwidth savings for two cases: (a) with the dynamic proxy cache and (b) without. We next describe our assumptions and derive a generic expression for the number of outbound bytes served by a given Web site infrastructure, i.e., the number of bytes transmitted between the back end and the DPC during a given time period. We then derive specific expressions for each of the two cases.

Recall from our discussion in Section 2 that a dynamic script generates pages. For the purposes of this analysis, we model a given Web application as a set of such pages $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$. Each page is created by running a script (as described in Section 2), and the resulting page consists of a set of fragments, drawn from the set of all possible fragments, $\mathcal{E} = \{e_1, e_2, \ldots, e_m\}$. We let E_i, $E_i \subseteq \mathcal{E}$, be the set of fragments corresponding to page c_i. There exists a many-to-many mapping between $\mathcal{C}$ and $\mathcal{E}$, i.e., a page can have many fragments and a fragment can be associated with many pages. We are interested in the size of a page, which depends on the size of its constituent fragments. The exact size of a fragment cannot be determined a priori, since it will depend upon a variety of run-time factors (e.g., query selectivity). Thus, we use the average size of a fragment e_j, which we denote by s_{e_j}. Each page also has f bytes of header information associated with it. Header information includes HTTP headers, such as `Server, Content-type`.

We define *expected bytes served*, $\overline{B}$, as the average number of bytes served by the Web site that is hosting the application during some time interval. In other words, $\overline{B}$ is the number of bytes transferred between the back end and the DPC during some period of interest. To compute $\overline{B}$, we need to know the size of each *response* and the number of times the page is accessed during the time interval. A response refers to the content that is generated by the application server to represent the requested page. In our analysis, we are interested in capturing the impact of our system on the bandwidth requirements for the dynamic content that is served.

When the dynamic proxy cache is not used, the response size is simply the page size. However, when the dynamic proxy cache is used, the size of the response will be different from the page size due to the inclusion of the tags and the exclusion of the cacheable content. Let S_{c_i} be the size of the response corresponding to page c_i as delivered by the hosting site, and $n_i(t)$ be the number of times the page c_i is accessed during the specified time interval. Then the general form of $\overline{B}$ over a given time interval is given by: $\Sigma_{i=1}^{n} S_{c_i} \times n_i(t)$. Note that S_{c_i} will be different for the no cache and dynamic proxy cache cases, while $n_i(t)$ will be the same. We now proceed to derive expressions for $n_i(t)$ and S_{c_i}.

In deriving an expression for $n_i(t)$, we need to characterize the access rate for a given page, e.g., the probability that the *Fiction* category page is requested, and the arrival rate of requests to the `www.books.com` site. Let $\mathcal{P}(i)$ be the probability that page c_i is accessed for a given request and $f(t)$ be the probability density function (pdf) that describes the arrival rate of requests. Then the number of times page c_i is accessed during the interval (t_1, t_2) is $\mathcal{P}(i) \int_{t_1}^{t_2} f(t)dt$.

We assume that $\mathcal{P}(i)$ is governed by the Zipfian distribution, which has been shown to describe Web page requests with reasonable accuracy [2, 12].

We now derive expressions for response size S_{c_i} for the two cases. For the no cache case, the size of the response for page c_i, denoted as $S_{c_i}^{NC}$, is given by $\Sigma_{\forall e_j \in c_i} s_{e_j} + f$, which is the sum of the sizes of all the fragments on the page and the header information.

For the dynamic proxy cache case, we must consider first whether a given fragment is considered to be cacheable. Let X_j be an indicator variable defined as follows:

$$\forall e_j \in E, X_j = \begin{cases} 1 & \text{if fragment } e_j \text{ is cacheable} \\ 0 & \text{otherwise} \end{cases}$$

We assume that the cacheability of each fragment is determined at design time. At run-time, we are interested in the fraction of fragments found in cache, which we denote as h. Then, the size of the response $S_{c_i}^C$, is given by $\Sigma_{\forall e_j \in c_i}[X_j[(h \times g) + (1-h)(s_{e_j} + 2g)] + (1 - X_j)(s_{e_j}) + f]$.

We have compared the expected bytes served for the two cases using the baseline parameter values shown in Table 2. Our choice of 0.8 as the baseline hit ratio is driven largely by the numerous studies that have shown that Web requests often exhibit locality [2, 12]. Furthermore, our experience with several large enterprise Web applications indicates that such hit ratios are easily achievable in practice.

Parameter	Value
hit ratio (h)	0.8
fragment size (s_e)	1K bytes
number of fragments per page	4
number of pages	10
average size of header information (f)	500 bytes
tag size (g)	10 bytes
cacheability factor	0.6
number of requests during interval (R)	1 million

Table 2: Baseline Parameter Settings for Analysis

In this comparison, we plot the ratio $\frac{\overline{B}^C}{\overline{B}^{NC}}$. Figure 2(a) shows the results of this comparison as fragment size (s_e) is varied. As this figure shows, this ratio decreases as fragment size increases. For small fragments sizes (e.g., less than 1 KB), the ratio exhibits a steep drop. This drop can be explained as follows: For small fragment sizes, the size of the tags is large with respect to the fragment size, decreasing the savings in bytes served for the dynamic proxy cache. This is why the ratio is greater than 1 as the fragment size approaches 0. As these results indicate, our dynamic proxy caching technique has a greater impact for larger fragment sizes (e.g., greater than 1 KB).

We now examine the sensitivity of expected bytes served to changes in key parameter values. We begin by varying hit ratio (h), while holding all other parameter values constant. Figure 2(b) shows the percentage savings in expected bytes served as the hit ratio is varied from 0 to 1. In the case where no fragments are served from cache (i.e., $h = 0$), we see that the savings is negative. In other words, there is a cost to use the dynamic proxy cache in this case because it adds tags to the responses, thereby increasing the response sizes. This effect holds up to the point where $h = 0.01$. Thus, as long as 1% or more fragments are served from cache, using the dynamic proxy cache will reduce the expected bytes served. Clearly, the greatest savings occurs when all fragments are served from cache (i.e., $h = 1$).

The foregoing results indicate that the dynamic proxy cache is indeed beneficial in terms of reducing the expected number of bytes transferred. The dynamic proxy cache, however, incurs a cost. In particular, assembly of the page at the dynamic proxy cache requires that each response be scanned for the tags. A logical question that arises is: *does the savings in bytes transferred offset the cost to scan?* We now provide a comparative analysis in an attempt to answer this question. More specifically, we compare the savings in expected bytes served to the scan cost. Note that regardless of whether the dynamic proxy cache is used, each packet is scanned by the firewall. Let y be the cost for the firewall to scan a byte. Then the cost to scan in the case where no cache is used is given by:

$$scanCost^{NC} = \overline{B}^{NC} \times y \quad (1)$$

Let z be the scan cost per byte for the dynamic proxy cache. Then the cost to scan in the case where the dynamic proxy cache is used is $scanCost^C = \overline{B}^C(y + z)$. Both the firewall and the dynamic proxy cache scan a given string of bytes. Since string matching algorithms (e.g., KMP [18]) are linear-time algorithms, we can consider the scanning costs for the firewall and the dynamic proxy cache to be of the same order. Thus, we assume that $z \approx y$. Making this substitution, our expression for the scan cost per byte for the dynamic proxy cache becomes:

$$scanCost^C = \overline{B}^C \times 2y \quad (2)$$

In comparing our expressions for the cost to scan in both cases, (1) and (2), we expect the dynamic proxy cache to provide better performance when the following condition holds: $\overline{B}^{NC} > 2\overline{B}^C$. Thus, we can conclude the following result:

RESULT 1. *It is preferable to use the dynamic proxy cache when the expected bytes served with no cache are more than twice the expected bytes served with cache.*

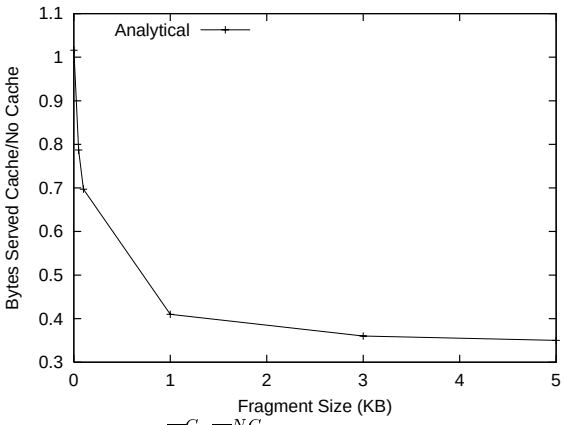
Figure 3(a) shows a comparison of (1) and (2) as the *cacheability factor* is varied (using again the parameter settings in Table 2). The cacheability factor is the percentage of all fragments that are cacheable for a given application. This figure shows two plots: (a) the savings in expected bytes served, and (b) the savings in bytes scanned (both expressed as percentages). The upper curve shows the savings in bytes served. As expected, this savings increases as the cacheability ratio increases. Note that this savings is positive over the entire range, indicating that employing the dynamic proxy cache will always decrease the bytes served. The lower curve shows the savings in the scan cost. The savings in this case also increases as the cacheability ratio increases. An important difference in this curve is that the savings in bytes does not always offset the scan cost, as indicated by the negative range. More specifically, using the parameters we have selected, if the cacheability ratio is less than about 50%, then it is not worth caching since the scan cost is greater than the savings in bytes served.

6. EXPERIMENTAL RESULTS

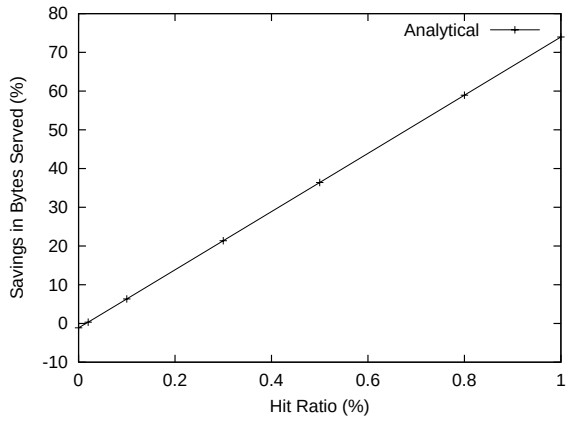
In this section, we attempt to validate our analytical results obtained in Section 5 with a set of experimental results.

We have implemented our dynamic proxy caching system. Both the DPC and the BEM are written in C++. The DPC is built on top of Microsoft's ISA Server [27] so that we can take advantage of ISA Server's proxy caching features. The page assembly code is implemented as an ISAPI filter that runs within ISA Server.

Our experiments were run in a test environment that attempts to simulate the conditions described in Section 5. Thus, we have incorporated the parameter settings in Table 2. The test site is an ASP-based site which retrieves content from a site content repository.

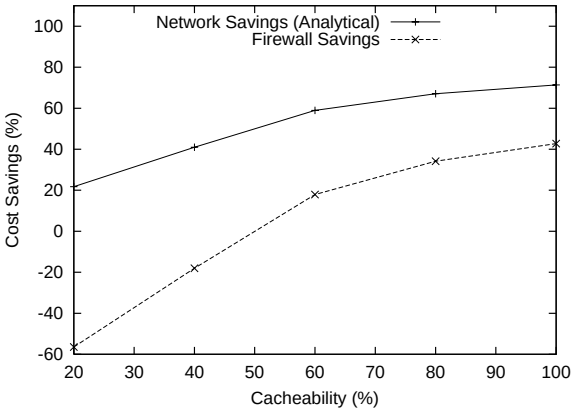


(a) $\overline{B}^C/\overline{B}^{NC}$ vs. Fragment Size

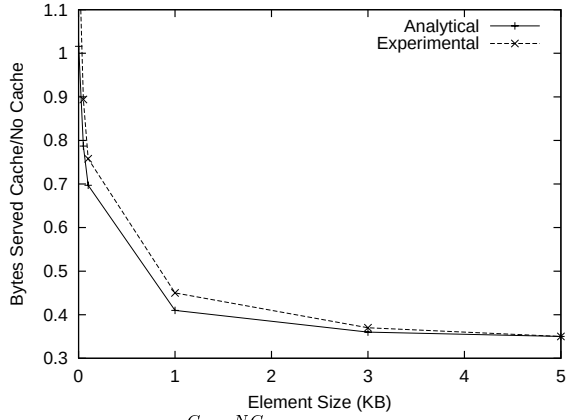


(b) Expected Bytes Served (%) vs. Hit Ratio

Figure 2: Analytical Results - Expected Bytes Served



(a) Comparison of Cost Savings



(b) $\overline{B}^C/\overline{B}^{NC}$ vs. Fragment Size

Figure 3: Analytical and Experimental Results

The basic test configuration consists of a Web server (Microsoft IIS), a site content repository (Oracle 8.1.6), a firewall/proxy cache (ISA Server), and a cluster of clients. The client machines run WebLoad, which sends requests to the Web server. For the dynamic proxy cache case, the DPC runs on the ISA Server machine, and the BEM runs on the IIS machine. Communication between all software modules is via sockets over a local area network. Figure 4 shows the test configuration. This figure attempts to show both the logical and physical test configurations. The origin site components (Web server, DBMS, and BEM) run on one machine (labeled *Origin Site*), while the components that reside outside the site infrastructure (firewall, proxy cache, and DPC) run on another machine (labeled *External*).

The number of bytes served is obtained by measuring bandwidth using the Sniffer network monitoring tool [33]. More precisely, the bandwidth measurement is taken between the Origin Site machine and the External machine in Figure 4. In these experiments, we are interested in capturing the impact of our system on the bandwidth requirements for the dynamic content that is served. Based on our earlier discussion regarding static content, the static content in these experiments is cacheable in the ISA Server proxy cache. Thus, in steady-state, static content will be served

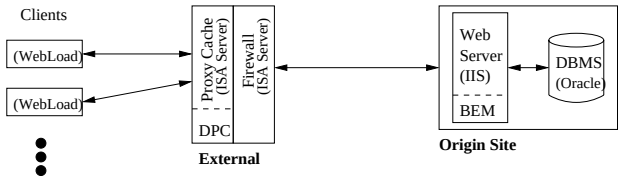


Figure 4: Test Configuration

from the ISA Server proxy cache and therefore will not impact bandwidth requirements between the Web server and the DPC.

Figure 3(b) shows the ratio $\frac{\overline{B}^C}{\overline{B}^{NC}}$ as fragment size is varied. Our results from Section 5 are repeated here (the curve labeled "Analytical") for comparison purposes. As this figure shows, our experimental results follow our analytical results closely. Interestingly, the analytical curve falls below the experimental curve. This difference can be explained by the network protocol headers (e.g., TCP/IP headers) that are included in the responses, which the Sniffer tool captures in its bandwidth measurements. However, we do not account for these headers in our analytical expressions. Thus, for

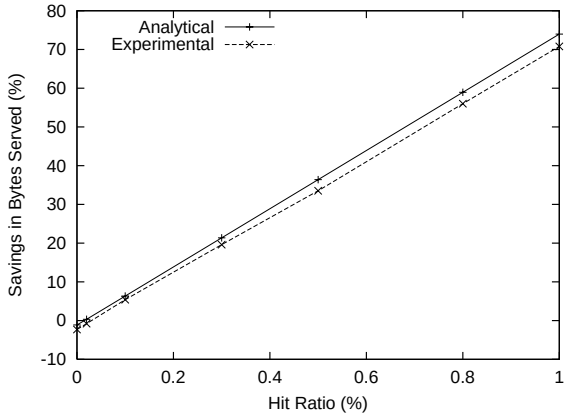


Figure 5: Expected Bytes Served (%) vs. Hit Ratio

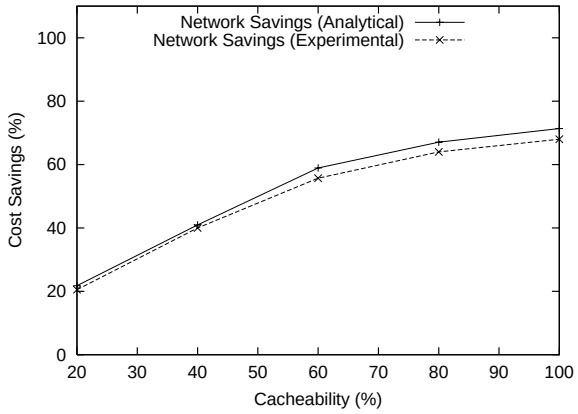


Figure 6: Validation of Cost Savings

every response, there is some network protocol messaging overhead. The smaller the response, the greater this overhead is. This is why the difference between the analytical and experimental curves is higher for smaller fragment sizes than it is for larger fragment sizes.

As in Section 5, we now examine the sensitivity of expected bytes served to changes in hit ratio. Figure 5 shows a comparison of this sensitivity for the analytical (curve repeated from Figure 2(b)) and experimental cases. Here again, our experimental results closely follow our experimental results. In this case, the analytical curve is slightly higher than the experimental curve, and the difference increases as the hit ratio increases. This is again a result of the network protocol headers that are included in the bandwidth measurements. Specifically, as more content is served from cache, response size decreases, yet the network protocol message size remains constant. Thus, the message overhead increases with respect to the response size as hit ratio increases, causing the savings to be smaller in the experimental case.

Figure 6 shows a comparison of the sensitivity of expected bytes served to changes in cacheability. The analytical curve is repeated from Figure 3(a) (the upper curve). Once again, the experimental results follow our analytical results closely. We also observe again the effects of the network protocol headers that are included in the experimental results, which cause the analytical curve to be higher than the experimental curve.

7. LIMITATIONS OF THIS WORK

The approach considered in this paper assumes a reverse proxy configuration, in which a single instance of the cache sits between a Web site's firewall and the Internet, serving cached dynamic content from outside the site's infrastructure. This provides significant bandwidth savings within the site's infrastructure, but does not impact bandwidth usage between the site and the end user.

An ideal approach would place the dynamic proxy cache on the edge, serving dynamic content from a point close to the end user and providing bandwidth savings not only within the site infrastructure, but also between the site and the forward proxy cache. In this approach, a number of forward proxies would be placed at strategic points around the network to provide optimal coverage. Since content would be served from the edge of the network, end users would see dramatic improvements in response time.

There are, however, a number of technical challenges associated with approach.

- *Request Routing:* With multiple dynamic proxy caches out on the network, how can we route requests for dynamic content optimally across the cache set? Most work in this area addresses the problem of routing static content identified by a URL. However, we are interested in routing *fragments* of dynamic content rather than full pages, which cannot be identified with a URL. Another complication within this area is the issue of handling proxy failure. Here, requests routed to a given dynamic proxy cache must failover seamlessly and transparently (from the user's point of view) to another proxy cache.

- *Cache Coherency:* How do we handle issues of cache coherency across multiple distributed caches? Here, multiple copies of a particular fragment may reside on different dynamic proxy caches distributed across the network. Some mechanism must be in place to ensure that correct responses are served to end users from the caching system.

- *Cache Management:* How do we manage the content of multiple caches? Changes to the data source on a site cause fragments to become invalid. The dynamic proxy caches distributed across the network need some means of obtaining notice of such changes.

- *Scalability:* Clearly, a system comprised of muliple caches distributed across the network and addressing the issues noted above must contain some complexity within its protocols. However, this system must be capable of serving heavy traffic loads in real time. In other words, the data structures and algorithms underlying the system must scale, both in time and space requirements.

8. CONCLUSION

In this paper, we have proposed an approach for granular, proxy-based caching of dynamic content. The novelty in our approach is that it allows both the content and layout of Web pages to be dynamic, a critical requirement for modern Web

applications. Our approach combines the benefits of existing proxy-based and back end caching techniques, without their respective limitations. We have presented the results of an analytical evaluation of our proposed system, which indicates that it is capable of providing significant reductions in bandwidth on the site infrastructure. Furthermore, we have described an implementation of our system and presented a case study which details the performance results of this system on a major real-world dynamic Web application. Our implementation results demonstrate that our system is not only capable of providing order-of-magnitude reductions in bandwidth requirements, but also order-of-magnitude reductions in end-to-end response times.

9. ACKNOWLEDGEMENTS

The authors would like to thank Jayant Haritsa for his insightful comments during the preparation of this manuscript.

10. REFERENCES

[1] Digital Island (a Cable & Wireless Company). Digital island 2 way web services. http://www.digitalisland.net.

[2] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the www. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS '96)*, 1996.

[3] Network Appliance. http://www.netapp.com.

[4] CacheFlow. Accelerating e-commerce with cacheflow internet caching appliances (a cacheflow white paper, October 1999.

[5] K. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proceedings of the ACM SIGMOD 2001 Conference*, pages 532 – 543, 2001.

[6] J. Challenger, P. Dantzig, and A. Iyengar. A scalable system for consistently caching dynamic web data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, 1999.

[7] ESI Consortium. Edge side includes. http://www.esi.org, 2001.

[8] Inktomi Corp. Inktomi network products. http://www.inktomi.com/products/network/.

[9] Oracle Corp. Oracle 9ias database cache. http://www.oracle.com/ip/deploy/ias/db_cache_fov.html.

[10] Vignette Corp. Vignette content suite. http://www.vignette.com.

[11] IBM. Corporation. Ibm websphere edge server version 2.0 (product documentation), 2001.

[12] C. Cunha, Bestavros, and M. Crovella. Characteristics of www client-based traces. Technical Report Technical Report TR-95-010, Boston University Computer Science Department, 1995.

[13] A. Datta, K. Dutta, D. Fishman, K. Ramamritham, H. Thomas, and D. VanderMeer. A comparative study of alternative middle tier caching solutions. In *Proceedings of the 2001 VLDB Conference*, Rome, Italy, September 2001.

[14] A. Datta, K. Dutta, K. Ramamritham, H. Thomas, and D. VanderMeer. Dynamic content acceleration: A caching solution to enable scalable dynamic web page generation. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, May 2001.

[15] S. Gadde, M. Rabinovich, and J. Chase. Reduce, reuse, recycle: An approach to building large internet caches. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 1997.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vliss. *Design Pattern Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series, October 1994.

[17] IBM. Websphere application server. http://www.ibm.com.

[18] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, June 1977.

[19] Q. Luo and J. Naughton. Form-based proxy caching for database-backed web sites. In *Proceedings of the 2001 VLDB Conference*, pages 191–200, September 2001.

[20] Q. Luo, J. Naughton, R. Krishnamurthy, P. Cao, and Y. Li. Active query caching for database web servers. In *Proceedings of WebDB 2000*, 2000.

[21] Microsoft. Asp, c#, vbscript, and asp+. http://www.microsoft.com.

[22] Sun Microsystems. Java servlets and jsp. http://java.sun.com.

[23] C. Mohan. Tutorial: Caching technologies for web applications. Tutorial at 2001 VLDB Conference, September 2001.

[24] Perl. http://www.perl.org.

[25] M. Rabinovich and A. Aggarwal. Radar: A scalable architecture for a global web hosting service. *WWW8 / Computer Networks*, 31(11-16):1545–1561, 1999.

[26] Morgan Stanley Dean Witter Analyst Report. *The Internet Evolution - Content Delivery Networks*. Morgan Stanley Dean Witter, November 2000.

[27] Microsoft ISA Server. http://www.microsoft.com/isaserver.

[28] TimesTen Software. http://www.timesten.com.

[29] SpiderCache. http://www.spidercache.com.

[30] BEA Systems. Weblogic application server. http://www.bea.com/products/weblogic/index.html.

[31] Akamai Technologies. http://www.akamai.com.

[32] Chutney Technologies. http://www.chutneytech.com.

[33] Sniffer Technologies. Sniffer basic. http://www.sniffer.com/products/sniffer-basic/.

[34] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching strategies for data intensive web sites. In *Proceedings of the VLDB 2000 Conference*, pages 188–199, 2000.