Caching Techniques for Dynamic Web Servers

A Thesis

Submitted for the Degree of

Doctor of Philosophy

in the Faculty of Engineering

by

Suresha

Department of Computer Science and Automation

Indian Institute of Science

Bangalore 560 012 INDIA

June 2007

Publications based on this Thesis

- **A Key-Mapping Technique for Proxy-Based Dynamic Web Cache Consistency,** *Inter Research Institute Student Seminar* **(IRISS-2002), IISc, Bangalore, (Awarded second prize),** *March 2002***.**
- **Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation,** *Proceedings of ACM SIGMOD International Conference on Management of Data (***SIGMOD***),* **Madison, Wisconsin, USA, [pgs. 97-108],** *June 2002***.**
- **A Proxy-Based Approach for Dynamic Content Acceleration on the WWW,** *Proceedings of 4th IEEE International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems (***WECWIS***),* **Newport Beach, California, USA, [pgs. 159-164],** *June 2002***.**
- **An Integrated Approach for Reducing Dynamic Web Page Construction Time,** *Proceedings of 10th National Conference on Communications(***NCC***),* **Bangalore, India, [pgs. 489-493],** *January 2004***.**
- **On Reducing Dynamic Web Page Construction Times,** *Proceedings of 6th Asia Pacific Web Conference (***APWEB***),* **Hangzhou, China, [pgs.722-731],** *April 2004,* **published as** *Advanced Web Technologies and Applications, Springer, Lecture Notes in Computer Science (LNCS) 3007, Eds. Jeffrey Xu Yu, Xuemin Lin, Hongjun Lu, Yanchun Zhang.*
- **Integrating Code and Fragment Caching to Speedup Dynamic Webpage Construction,** *Proceedings of 12th International Conference on Management of Data (***COMAD***),* **Hyderabad, India, [pgs. 112-121],** *December 2005***.**

Acknowledgements

I am really thankful to my advisors Prof. Jayant Haritsa and Prof. R.C. Hansdah, who trained me to do focused, systematic and scientific research. Their dedication towards research is inspiring. My research experience with them is really unforgettable for the rest of my life. I don't have words to thank them for the support they extended for everything.

My special thanks to all faculties and staffs of CSA and SERC. I also thank all the DSLlites, CSA students, SERC students and other friends, who have helped me a lot during my research and stay at IISc, directly and indirectly.

I thank all those who disliked me, because of them I took this task as a challenge and there by I could get my PhD.

Abstract

Many websites incorporate *d*ynamic web-pages in order to deliver customized contents to their users. Websites are shifting from a static model to a dynamic model, in order to deliver their users 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 within the website and across the Internet. In fact, dynamic pages are constructed on the fly, "on demand". Hence dynamic pages, due to their construction overheads and non-cacheability, result in substantially increased user response times, server load, and increased bandwidth consumption, as compared to traditional static pages. Due to continuous growth of Internet traffic and websites becoming increasingly complex, performance and scalability are becoming major issues for dynamic websites.

This thesis presents some novel ways of integrating existing solutions to address performance and scalability issues. Specifically, it aims at achieving reduced bandwidth consumption from web infrastructure perspective, and reduced page construction times from user perspective. To address performance and scalability issues, various dynamic content caching approaches have been proposed in the literature. These approaches can be broadly classified into two categories: *proxy-based caching approaches* and *server-side caching approaches* (also called as *back-end caching approaches*).

Proxy-based caching approaches store content at various locations outside the site infrastructure and can improve website performance by reducing content generation delays, firewall processing delays, and bandwidth requirements. However, existing proxy-based

caching approaches either cache at the page-level or at the fragment-level. The proxy-based page-level caching does not guarantee that correct and fresh pages are served and provides very limited reusability. The proxy-based fragment-level caching requires the use of pre-defined page layouts. To address these issues, several *server-side caching approaches* have been proposed. While server-side caching 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 thesis, we consider mechanisms for reducing bandwidth consumption and dynamic page construction overheads by integrating fragment caching with various techniques such as proxy-based caching of dynamic contents, caching codes and pre-generating pages. The proposed mechanisms can be classified into two categories, namely *proxy-based* and *server-side*. In the proxy-based mechanisms, we concentrate on reducing the bandwidth consumption due to dynamic web pages. In the server-side mechanisms, firstly, we concentrate on reducing the page construction time by resorting to a hybrid technique of fragment caching and page pre-generation. Secondly, we make use of the fragment caching to further reduce execution time of the scripts integrating with code caching and optionally augmented with page pre-generation.

In summary, this thesis presents some novel ways of integrating existing solutions for serving dynamic web pages with the goal of achieving reduced bandwidth consumption, from web infrastructure perspective, and reduced page construction times from user perspective.

Keywords

Proxy-Based Caching
Edge Caching
Rich Contents
Dynamic Content
Dynamic Web Server
Static Web Page
Dynamic Web Page
Fragment Caching
Forward Proxy
Reverse Proxy
Performance
Scalability
Prefetching
Web Log Mining
Code Caching
Freshness
Correctness
Time To Live
Back-End Caching
Reusability
Database Caching
Caching Webviews

Contents

List of Tables

List of Figures

Notation and Abbreviations

Abbreviation	Stands for
ESI	*Edge Side Include*
ASP	*Application Server Process*
JSP	*Java Server Pages*
Pure_FC	*Pure Fragment Caching*
Pure_PG	*Pure Page Pre-generation*
Hybrid	*Both Fragment Caching and Page Pre-generation*
NO_FC_PG	*Neither Fragment Caching nor Page Pre-generation*
Pure_CC	*Pure Code Caching*
Integrated	*Both Fragment Caching and Code Caching*
NO_FC_CC	*Neither Fragment Caching nor Code Caching*
DPC	*Dynamic Proxy Cache*
CDN	*Content Delivery Network*
CMS	*Content Management System*
XML	*Extensible Markup Language*
HTML	*HyperText Markup Language*
MVC	*Model View Controller*
JDBC	*Java Database Connectivity*
ODBC	*Open DataBase Connectivity*
CGI	*Common Gateway Interface*
TTL	*Time To Live*

Chapter 1

Introduction

Traditionally, websites started with serving static web pages, by we mean the web pages served from files, which are constructed manually and readily available. But in recent times, websites started serving dynamic web pages, by we mean web pages constructed "on the fly" by running scripts. Now, websites are shifting from a static web page service model to a *dynamic* web page service model in order to facilitate delivery of custom content to users [19]. Increasingly, e-business sites employ dynamic web pages since they provide a much wider range of interactions than static pages. A website can generate dynamic web pages at run time using dynamic page generation technologies to significantly increase its flexibility in customizing page contents. While dynamic web pages enable much richer interactions than static pages, these benefits are obtained *at the cost of significantly increased user response times and bandwidth consumption*, due to the on-demand page construction and non-cacheability unlike static web pages. Dynamic web pages also seriously reduce the performance of the web server due to the load incurred by the page generation process. In fact, it has been recently estimated that server-side latency accounts for as much as 40 percent of the total page delivery time experienced by end-users [30]. Delays can be detrimental for websites, as users tend to leave a site if the response time is too long. A widely cited study [76] has helped to quantify this phenomenon – it finds that users will abandon requests with exponentially increasing probability as response time grows: at a response time of 7 seconds, 12% of users will

abandon their requests; at 8 seconds this number grows to 30%, and so on. Moreover, a recent study indicates that even this thumb-rule is too conservative, i.e., users are now expecting even faster response times [40]. Hence, *performance* and *scalability* are becoming major issues for dynamic websites.

In the context of dynamic web servers, we present in this thesis some novel ways of integrating existing solutions to address performance and scalability issues. Specifically, this thesis aims at achieving reduced bandwidth consumption from web infrastructure perspective, and reduced page construction times from user perspective.

1.1 Architecture of a Typical Dynamic Content Generation Process

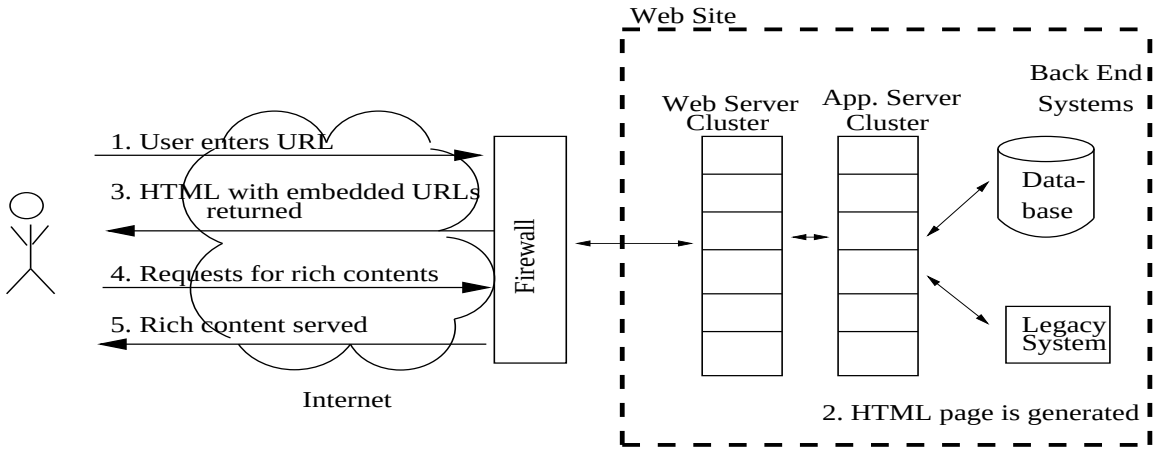


Figure 1.1: **Typical Dynamic Content Generation Application Architecture**

Consider an online book store as a running example. Figure 1.1 illustrates architecture of a website employing dynamic content generation technologies. We use this figure to describe the process by which a request is served in our running example. When a user first requests a page, his browser enters the URL (http://www.booksOnline.com/category.jsp? categoryID=Fiction) for the location of the script that will generate the requested page (shown as Step 1 in Figure 1.1). The request is passed from the web server to the application server, which executes the script

that generates the page (shown as Step 2 in Figure 1.1). Referring back to our running example, the application server would execute the category.jsp script. This script may access a database to retrieve the content associated with the Fiction category, create several objects, and format the content for display. This step often requires significant work, especially for sites running complex business logic.

The HTML that is output from Step 2 is sent back to the user (shown as Step 3 in Figure 1.1). The HTML typically contains several embedded references to rich content objects, such as images. These objects must be retrieved separately (shown as Steps 4 and 5 in Figure 1.1). These objects may be located on the origin server, though this is often not the case.

1.2 Performance Bottlenecks for 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:

- **network latency:** i.e., delays on the network between the user and the website.
- **server latency:** i.e., delays at the website itself, due to on-demand dynamic web page generation.

Network Latency: In the Internet, users and websites are typically separated by long distances. 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 been proposed to mitigate network delays, which will be discussed in Section 2.3.

Server Latency: After a user's request traverses the Internet and arrives at the website, a number of website 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 website, 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 [47]. 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 been proposed to address also server delays, which will be discussed in Section 2.4.

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 websites, the story is completely different. As mentioned previously, dynamic website 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 website 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 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) [29] 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).

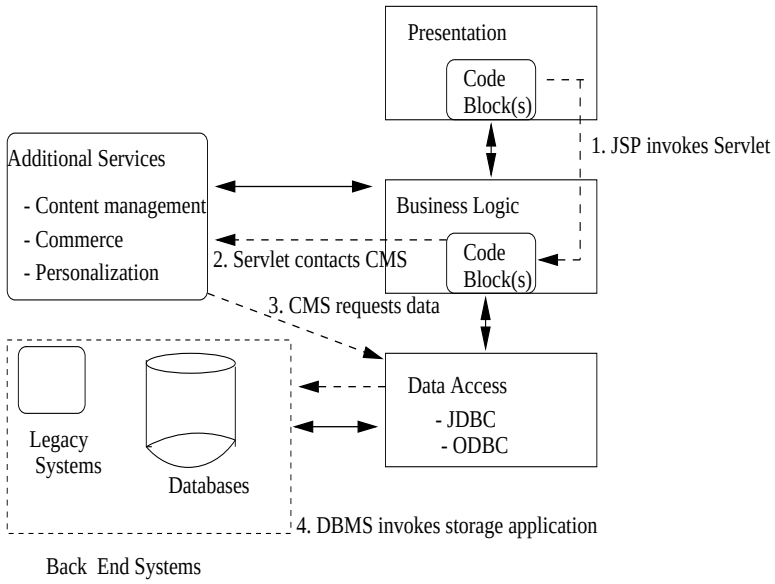


Figure 1.2: **Example of Workflow Required to Generate Dynamic Page**

To better illustrate how multi-layered architectures serve requests, consider Figure 1.2, 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 [67] 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 the 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, which leads to performance bottlenecks of dynamic web servers.

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.

1.3 Optimization Techniques

To address *performance* and *scalability* issues of dynamic websites, a variety of optimization techniques have been developed in the recent literature. Here, we consider the techniques for improving user response time, since response time is more severe from user perspective. These solutions can be broadly classified into two groups: namely, *caching* and *predictive* techniques. Caching is a widely-used approach to mitigate the performance degradation due to WWW content distribution and delivery [46]. Here, the content generated for one user is saved, and used to serve subsequent requests for the same content. Some significant caching techniques are dynamic content-aware full-page caching, content acceleration, database caching, fragment caching, active query caching, and code caching [13, 19, 20, 34, 39, 43]. Predictive techniques are alternative techniques to caching. Here, the web contents are fetched or generated based on some prediction. For example, client-side prefetching is a predictive technique [25].

1.3.1 Caching Techniques

Active query caching for database web servers is proposed in [43], which is basically a collaboration scheme between active web proxies and database web servers. The goal here is to enable web proxies to share the workload of database web servers as much as possible at a low overhead. The active proxy answers from the cache, all queries which it has seen before, as well as for those queries whose answers are contained in the answers of previously cached queries (for simple selection queries, which are a special case). Active query caching can improve system scalability as well as reduce wide area network traffic.

In *page-level caching*, the proxy caches full page outputs of dynamic sites. Page-level caching has been considered in [9, 34]. Page-level caches can improve website performance by reducing delays associated with page generation, as well as reduce bandwidth requirements. However, there are some major limitations associated with using page-level caching. First, since page-level caching solutions rely on the request URL to identify pages in cache, they may serve *incorrect pages* [21]. Second, there is often

very little reusability of full HTML pages. That is, in page-level caching, there will be unnecessary invalidation because even if one or a few elements on a page become invalid, then the entire page becomes invalid.

Dynamic page assembly is an approach popularized by Akamai [1]. 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. A drawback of this approach is the requirement that a site follows a specified page design paradigm, specifically, the use of templates which in turn call separate dynamic scripts for each dynamically generated fragment, forcing the page layout to be specified in advance.

Fragment caching technique is proposed in [19, 20]. This technique involves caching fragments (components) of dynamically generated pages. A fragment is a part of a web page, which is common across one or more web pages. The success of fragment caching is based on the notion that many fragments of dynamic pages do not change for extended periods of time and hence can be cached for future requests. A cached fragment can be invalidated whenever the underlying data source corresponding to the cached fragment changes. A cached fragment is reusable across all users. Thus cached fragments are more reusable compared to cached pages. In fragment caching, the page skeleton is generated for each user's request and hence serves the correct page. Fragment caching achieves reductions in dynamic page construction time by making use of cached fragments, instead of computing them afresh each time.

Code caching is a technique that has been proposed and implemented in the context of the PHP Accelerator project [39]. Before executing a script, the engine reads, parses, and compiles the script into code ready for execution. Since, in practice, the scripts rarely change, this pre-processing is targeted for elimination in [39]. The compiled code can also be optimized before being put into execution to improve its efficiency and reduce its footprint. High performance can be achieved by using a shared memory cache from which the compiled code can be executed directly. The code caching technique only concentrates on reducing the script execution time. But, it does not eliminate the execution itself. The attractiveness of code caching is that it can be integrated with any

other optimization technique.

Among caching techniques described above, *fragment caching*, which reduces dynamic page construction time by caching dynamic fragments, is particularly attractive since it provides the following desirable guarantees [19, 20]: Firstly, it ensures the *freshness* of the page contents by maintaining an association between the cached dynamic fragments and the underlying data sources. Secondly, it ensures the *correctness* of the page contents by freshly generating the page skeleton each time the dynamic page is requested.

On the down side, however, fragment caching has some limitations: First, its utility is predicated on having a significant portion of dynamic fragments to be cacheable – however, such cacheability may not always be found in practice. Second, even when most fragments are cacheable, dynamic page construction begins only upon receiving the request for the page – therefore, the server latency may still turn out to be considerable. Finally, it does not address the problem of bandwidth requirements.

1.3.2 Predictive Techniques

Page pre-generation is an alternative to caching to mitigate page construction time at the server. A page can be pre-generated for one or more users based on a prediction technique. There are several page access prediction models that have been proposed in the literature [24, 35, 59, 63, 70, 77] based on information gained from mining web logs. The accuracy of these models has been found to be high enough to justify the pre-generation of dynamic content [59]. If a pre-generated page is requested by one or more users, then the page construction time for that pre-generated page is zero from the user's perspective. This is what one can expect at the most. But, on the other hand, an incorrectly predicted page will consume the server resources to pre-generate that page, which eventually turns out be a waste. This problem becomes severe when the system is heavily loaded.

Prefetching is another prediction technique, at the client side, that attempts to predict the object most likely to be accessed next by the user and fetches it into the browser cache in advance. By prefetching, the latency associated with objects that are currently

not in the cache, but most likely to be accessed in the near future, can be reduced. Prefetching complements the web caching. For those web objects, which are correctly predicted and fetched into the cache, the delay becomes zero, which is the best one can expect. Prefetching of web objects poses several problems. Firstly, it is difficult to predict the object which is likely to be accessed next, since web logs are not available to clients. Secondly, unsuccessful prefetching increases the network bandwidth consumption and the load on the server.

1.4 Motivation

It is clear in the context of dynamic web servers, performance and scalability are critically important issues. More over, internet traffic is exponentially increasing [71]. There are various solutions to address these issues. Many of these solutions are good in their own context, but have not been analyzed in an integrated fashion. In this thesis, we have carried out a study of combining a variety of these solutions and analyzed their performance, to address the above issues. We are impressed by the attractive features of fragment caching, since it ensures both correctness and freshness of page contents. Our solutions are based on combining fragment caching with various other techniques. We are motivated by the fact that the problem has to be tackled at various levels. For example, in the Internet and Intranet, it is the bandwidth consumption, whereas at the website, it is dynamic page construction overheads. The performance goals in this thesis are at two locations, namely network within the site infrastructure and dynamic web server.

The contributions of this thesis are shown in Figure1.3, in the context of a typical dynamic web server. Our contributions are marked with rounded rectangles and labeled as I, II, III, which are described below. The overall thesis structure is as shown in Figure 1.4 [1].

[1]At dynamic proxy caching, the work is carried out with the actual implementation. Since, fragment caching and code caching are available only in the form of proprietary softwares, we resorted to simulation for the results shown in server-side.

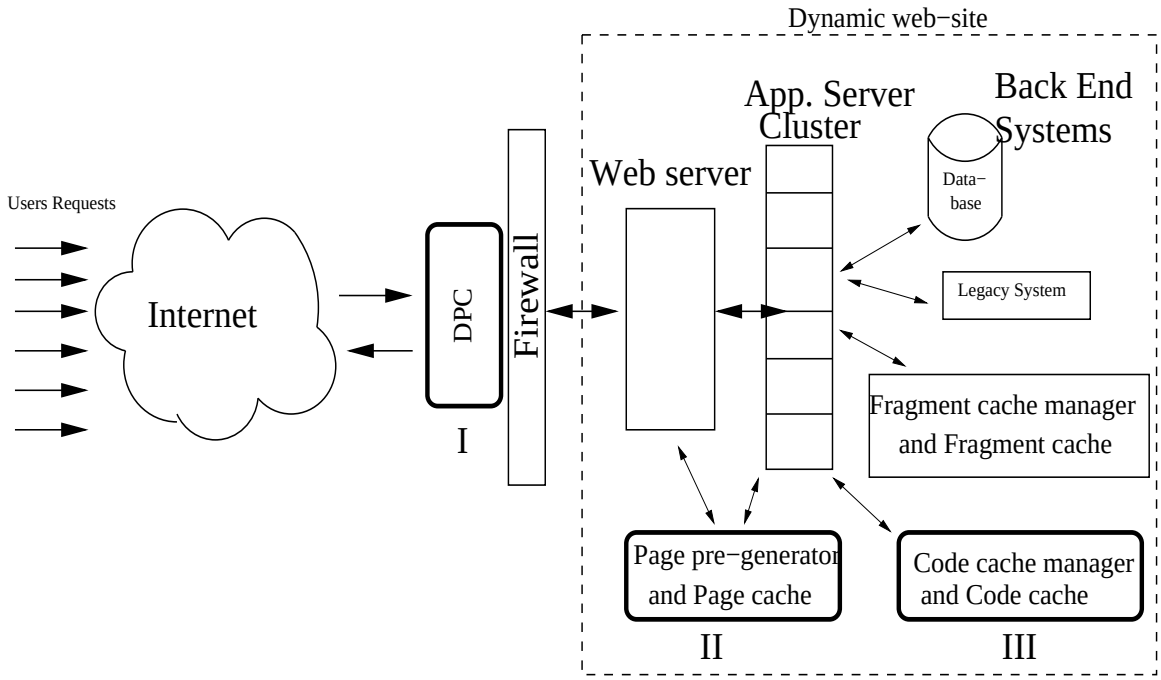


Figure 1.3: **Thesis Contributions**

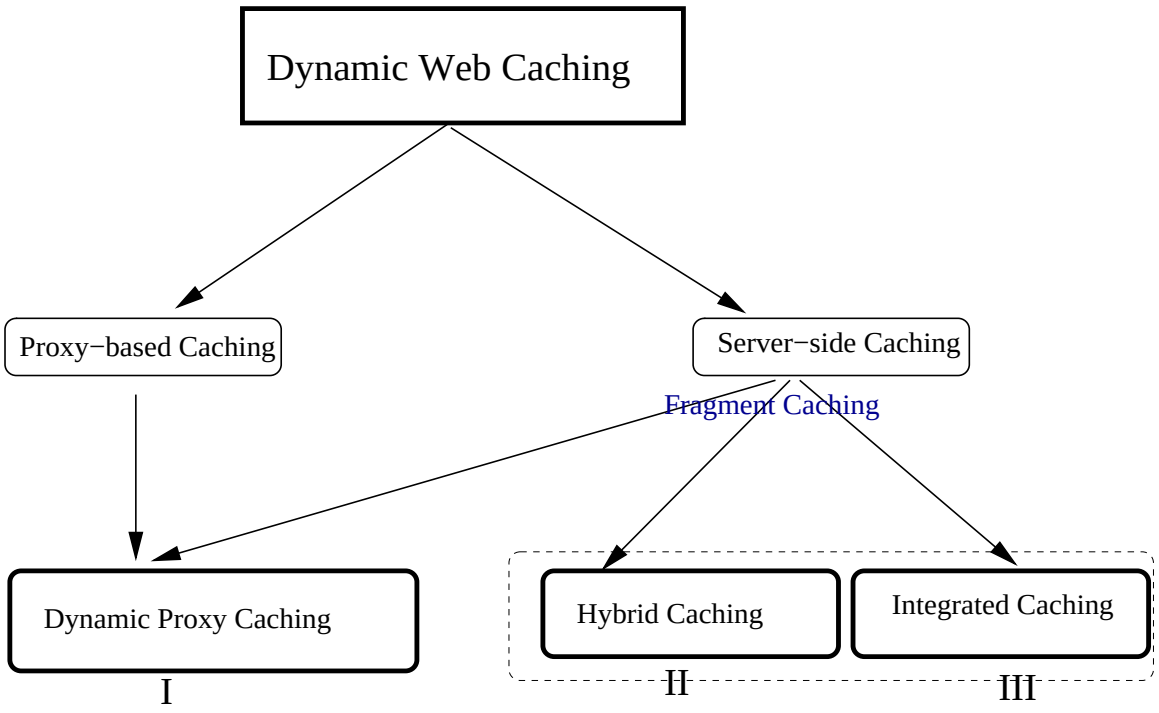


Figure 1.4: **Thesis Structure**

Contribution I: In the network, we try to reduce the network bandwidth consumption and related delays (like firewall processing delays, switching delays)[2]. We specifically address this problem by combining fragment caching with proxy caching. By taking the dynamic contents to proxy, we are able to achieve reduction in bandwidth consumption. We move the bulky fragment responses to the proxy cache and generate the page layout and any fresh fragment at the origin server. The dynamic page is completed at the dynamic proxy cache after filling the remaining fragment responses. The dynamic proxy acts as forward proxy, whereas the fragment cache remains inside the origin server.

The content kept at dynamic proxy is invalidated by a novel key-mapping technique. This technique does not carry any invalidation information for any cached contents at dynamic proxy. Instead, when the content is known to have been changed at the origin server, the content gets generated when it is next requested in any page construction at the origin server. A copy of it is sent to dynamic proxy in a new page request along with a unique number and also cached at the dynamic proxy for future use. In fact, at the back-end, we maintain just the fragment-name and a key associated with it in the back-end cache, whereas the actual contents are kept at the dynamic proxy outside the site infrastructure.

Contribution II: At the server-side, during normal loading, we reduce the dynamic page construction time by resorting to a hybrid technique of fragment caching and anticipatory page pre-generation, by utilizing the excess capacity with which websites are typically provisioned to handle peak loads. During heavy loads, a load-sensitive feed-back mechanism is used to turn off page pre-generation. We have shown that over fifty percent page construction time can be reduced compared to pure fragment caching, during normal loading.

Contribution III: Further, at the server-side, we show that the integration of fragment caching with code caching reduces page construction times during both normal

[2]This work was carried out at Chutney Technologies, USA, during a summer internship.

loading and peak loading. Here, we retain the benefits of fragment caching. At the time of script execution, we try to avoid the time-consuming repeated script parsing and interpretation by caching the compiled script code. In a nutshell the page construction time is reduced at the fragment cache level, if the responses are readily available in the fragment cache. If not, the compiled script codes are used to carry out the execution. This reduction in page construction time is achieved at the time of script execution. The net effect will be further reduced page construction time.

1.5 Thesis Contributions

The main contributions of this thesis are threefold:

- Reduced bandwidth consumption while retaining the benefits of fragment caching.
- Reduced page construction times during normal loading, by a hybrid technique of fragment caching and anticipatory page pre-generation, using excess capacity with which web servers are typically provisioned to handle peak loads.
- Reduced page construction times during both normal loading and peak loading, by integrating fragment caching and code caching, optionally augmented with anticipatory page pre-generation.

The above contributions are cumulative. We can achieve all the above benefits together. That is, both *reduced bandwidth consumption achieved at proxy-side*, and *reduced page construction time achieved at server-side*.

1.6 Organization of this Thesis

The reminder of this thesis is organized in the following manner: Chapter 2 gives a overview of the background material and the literature related to this thesis. Chapter 3 outlines our strategy for dynamic proxy implementation and combined reduced

response time and network bandwidth. Chapter 4 outlines our new approach to reduce dynamic page construction time by integrating fragment-caching with anticipatory page pre-generation, during normal loading, by utilizing the excess capacity with which websites are typically provisioned. Chapter 5 outlines our new approach to reduce dynamic page construction time by integrating fragment-caching with code-caching, optionally augmented with page pre-generation. Finally, Chapter 6 concludes the thesis, with avenues open for further research, in extending the problem or solution methodologies.

Chapter 2

Related Work

In this chapter, we start with the background necessary for our work followed by a review of the literature related to the main contributions in this thesis – namely, proxy-side caching, which we call dynamic proxy caching, and server-side caching, by integrating fragment-caching with anticipatory page pre-generation, and integrating fragment-caching with code-caching, optionally augmented with anticipatory page pre-generation. Before discussing the related work, we give a brief background about the material related to this thesis and we also distinguish between the two types of web pages, namely static and dynamic pages, and the way they are served.

2.1 Background

A web-site is a collection of web pages, typically common to a particular domain name on the World Wide Web (WWW) on the Internet. A web page is a HyperText Markup Language (HTML) document accessible generally via HyperText Transfer Protocol (HTTP). A website provides information about an organization. All publically accessible websites are seen as constituting a mammoth "World Wide Web" of information. A website can be used for various purposes like educational use, disseminating information about an organization, for doing business, for searching information on the WWW, etc.

A computer system that delivers web pages is called a *web server*. A web server

serves web pages in response to requests from web browsers. Every web server has an Internet Protocol (IP) address and possibly a domain name. A computer system can be made as a web server by installing web server software and connecting the machine to the Internet. A web server processes HTTP requests by responding with HTML pages. HTTP requests are generated by clients through their browsers.

A *static web page* contains content/information that does not change frequently. And it is displayed the same way each time the page is requested by any user. A static web page is designed in HTML manually. Static web pages can contain HTML tags and text, as well as other elements suitable for the page, such as images and animation. A website which serves static web pages is called a static website. Static websites deliver the same page contents to all visitors to the website. In case any changes are needed to static web pages, someone must edit each page whenever a change is required. Furthermore, if the content that needs changes appears on multiple web pages, each of them will have to be edited individually. As a result, websites that use static content are expensive to maintain. Static web pages are also inflexible. For example, we cannot personalize the content on a static web page.

A *dynamic web page* contains content/information that is generated when the user requests the page. This content is generated upon receiving the request by executing a script (program) file that contains the instructions to build the page based on the current state of the website. This content is extracted from databases or other data sources, allowing the web page to present the most current information. Common examples of simple dynamic pages are those that display the current date and time, or a visitor counter. A website which serves dynamic web pages is called a dynamic website. Dynamic websites may present differing information to different visitors to the same page in the website. In a dynamic web page, the display logic is maintained separately from the content and the content is stored in a database repository or obtained from other data sources instead of a HTML file. Each dynamic web page consists of a template that provides the *look and feel* plus *code* that assembles the page by retrieving the appropriate content. Both the *template* and *content* can be updated independently. This results in

lower maintenance costs and increased flexibility.

A static website comprises a set of related HTML pages (HTML files) hosted on a computer running a web server. A page request is generated when a visitor clicks a link on a web page, selects a bookmark in a browser, or enters a URL in a browser's address text box. When the web server receives a request for a static page, the server reads the request, finds the page, and sends it to the requesting browser. The final content of a static web page is determined by the page designer and doesn't change when the page is requested. Every line of the page's HTML code is written by the designer before the page is placed on the server.

When a web server receives a request for a static web page, the server sends the page directly to the requesting browser. When the web server receives a request for a dynamic page, however, it handles it differently: it maps the request to the corresponding script and passes it to a server called *application server* responsible for generating the page. The application server reads the code on the script, and constructs the page according to the instructions in the code. The result is a static page that the application server passes back to the web server, which then sends the page to the requesting browser. All that a requesting browser gets when the response for the page's request arrives is a pure HTML page.

Static pages are normally ready even before the request for it appears at the website. Unlike static web pages, dynamic web pages are constructed on the fly after receiving a request for the page and the page content is generated *on-demand*. A dynamic web page has two components, the *layout* and the *content*. The layout refers to how a page looks like, whereas the content refers to what a page contains. For a highly personalized dynamic web page, both the layout and content can be different. Two different requests for the same dynamic web page may turn out to provide different pages depending on the user and the state of the website. We say that a dynamic web page is correct if it is same as intended for a particular user request. We assume that a dynamic web page is fresh if all the components in it are fresh at the time of page assembly.

2.1.1 Serving Static Web Pages

For serving static web pages, there are web content caches often placed between end-users and origin servers as a means to reduce server load, network usage, and ultimately, user-perceived latency. Cached static contents typically have associated Time-To-Live (TTL) value, after which the content has to be validated with a remote server (origin or another cache), before they can be served to the end-user. To serve static web contents, there are a variety of well established solutions that are implemented practically and effectively at various levels in the Internet [7, 10, 16, 17, 72]. In contrast, the solutions to address the dynamic contents are still effectively in a development stage.

2.1.2 Serving Dynamic Web Pages

Unlike static web pages, the content and the layout of dynamic web pages are not ready before the request arrives at the origin server and the dynamic pages are constructed on the fly after receiving a request for the page and the page content is generated on-demand. To support dynamic content generation, a range of technologies are available. A dynamic website maintains a program corresponding to a dynamic web page, called a dynamic script (a program). A dynamic website may have a set of application servers. These application servers run dynamic scripts to generate dynamic web pages. A user request for a dynamic web page is mapped to a dynamic script at the website. The web server selects an application server and instructs it to execute the corresponding script and to produce the dynamic web page. The selected application server carries out script execution with necessary logic to generate the requested page, which involves contacting various resources to retrieve, process, and format the requested content into a user deliverable dynamic web page (which is in fact a HTML page). It is clear from this discussion that serving dynamic web pages involves a lot of resources and computation, leading to performance and scalability issues. These problems will still remain even with increased processor speed, since the Internet traffic for dynamic contents is also growing.

2.2 Dynamic Content Generation Process

Over the past few years, websites have transitioned from a static content model to a dynamic content model. Dynamic content generation allows web sites to offer a wider variety of services and contents. A broad range of technologies are available to support such dynamic content generation. For instance, application servers (e.g., BEA's WebLogic [6] and IBM's Web- Sphere [31]) 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 (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) [64], the Active Server Pages (ASP) family from Microsoft [44], Hypertext Preprocessor (PHP) [53], and Perl [52].

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.

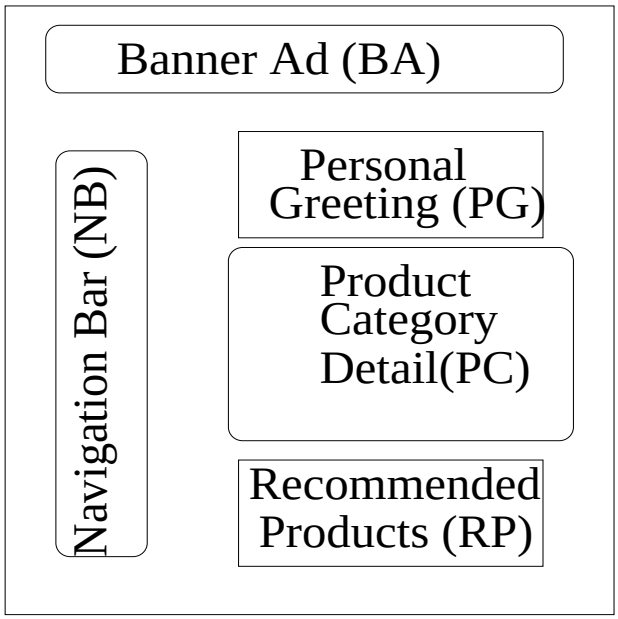
2.2.1 Dynamic Layouts

Consider an online book store that caters to both regular and casual visitors (we call them as registered users and non-registered users respectively). Assume a registered user is on the entry page of the site, which presents a list of category links that the user can choose to navigate. Suppose the user clicks on the Fiction category, which results in the following request arriving at the site's web server:
http://www.booksOnline.com/category.jsp?categoryID=Fiction. An application server at the site will execute the program *category.jsp*, which takes the categoryID input parameter to retrieve the content associated with the Fiction category. Suppose this request produces a page having the layout shown in Figure 2.1(a). This page contains a number of fragments. The **Banner Ad** fragment contains an advertisement that is retrieved

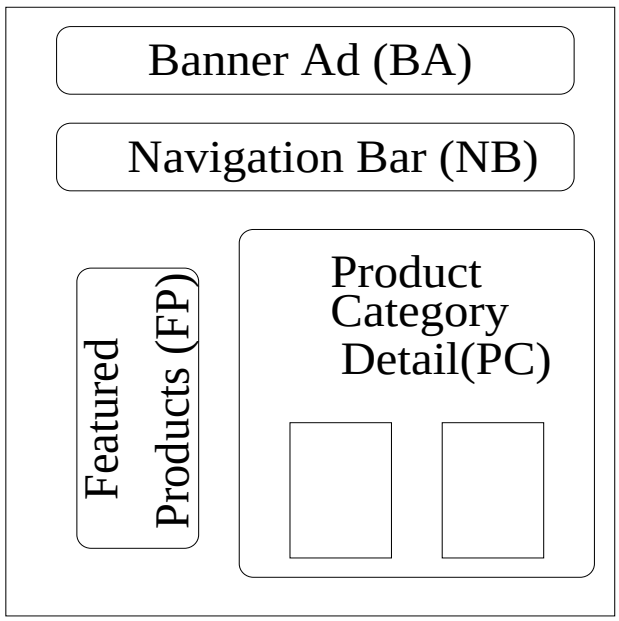
from an ad server based on the user's referring URL and the current time. The **Personal Greeting** fragment consists of a greeting for the user including the user's name and the current time. The **Product Category Detail** fragment displays the names, descriptions, and images associated with the products in the Fiction category. This information is obtained by querying a content database. The **Navigation Bar fragment** displays the navigation selections available. Finally, the **Recommended Products** fragment contains a list of recommended products which are retrieved from a personalization server based on the current category and user profile information.

Clearly, some of the fragments of the page shown in Figure 2.1(a) are based on user profile information, which would only be present for registered users. As mentioned earlier, the site also caters to non-registered users. Suppose that a non-registered user, from the home page, clicks on the Fiction category link, which results in exactly the same URL request arriving at the site server (i.e., http://www.booksOnline.com/category.jsp?categoryID=Fiction). However, for this user, the site provides a different page as shown in Figure 2.1(b). There are two key differences between the two pages shown in Figures 2.1(a) and 2.1(b). First, some of the contents are different – the page for non-registered users contains a **Featured Products** fragment rather than the **Personal Greeting** and **Recommended Products** fragments. Second, the page layouts are different – the **Navigation Bar** appears in a different place on the non-registered user's page and the **Product Category Detail** is displayed in two-column format rather than single-column.

In general, a 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, with respect to Figure 2.1(a), the different fragments (e.g., the **Banner Ad**) represent content, whereas the layout determines how the fragments are presented on the user viewable page. Examples of layout includes any HTML tags (e.g., <TABLE>, <TITLE>). Figure 2.2(a) shows a schematic of the layout for the registered user's page. In this figure, each $< L_i >$ represents the layout for a particular section of the page.



(a) For Registered User



(b) For Non-registered User

Figure 2.1: **Page Layouts:**

Each $< L_i >$ would include a string of markup tags, such as $< TABLEWIDTH =''$ $100\%'' >< TR >< TD >$ .... The remaining elements represent the content, e.g., $< BA >$ denotes the **Banner Ad**. Figure 2.2(b) shows the layout for the non-registered user's page for comparison purposes. 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., $< HR >$, which adds a horizontal rule).

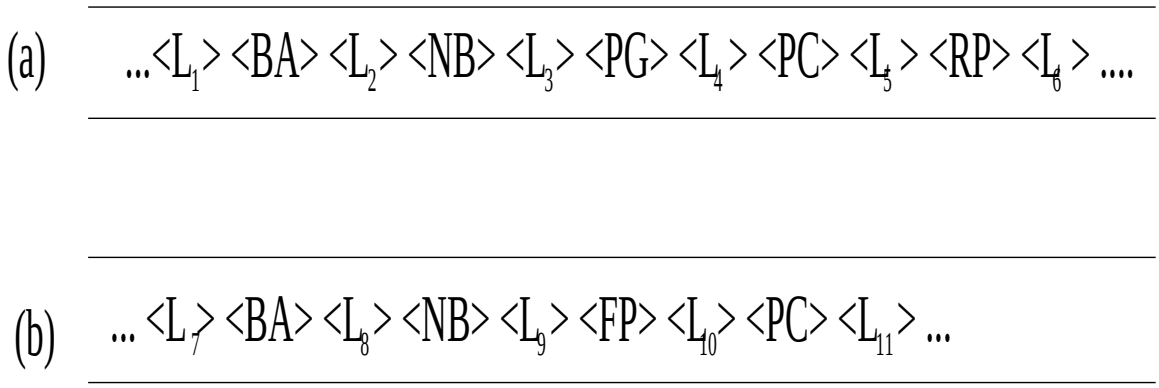


Figure 2.2: **Comparison of Dynamic Page Layouts**

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 runtime. 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 one of the major impediments to caching dynamic pages in a proxy cache.

Another important issue one can think of is the layout caching in addition to content caching. Layouts are normally specific user request and inputs. This may lead to huge number of layouts to be cached, with limited reusability. Hence, layout caching is not attractive.

2.3 Proxy-Based Caching Solutions

Two broad approaches exist in using proxies to cache dynamic pages, namely page-level caching and dynamic page assembly. In these approaches the dynamic contents are cached outside the site's infrastructure.

In page-level caching approach [9, 11], the proxy caches full page outputs of dynamic sites. Some solutions are deployed in forward proxy mode, in distributed caching architecture located at numerous points around the Internet [28, 54]. Page-level caching can improve website performance by reducing delays associated with page generation, as well as reducing bandwidth consumption. However, there are some major limitations associated with using page-level caching. Since, the page level caching solutions rely on the request URL to identify pages in the cache, they may serve incorrect pages. When pages are generated dynamically, different invocations of a given script are not guaranteed to produce the same page, even with the same input parameters. For example, consider a website that serves both registered (regular) and non-registered (casual) users. Let us say Bob is a registered user and Alice is a non-registered user. When Bob makes the request to the site, the site will put relevant personalization contents to the page served to Bob. Whereas there may not be any personalization for Alice. For Bob, the page served may include a "Hello Bob" greeting. Suppose this page is cached, then suppose a subsequent user, say Alice, requests the same page (using the same request URL). Now Alice, will receive a greeting "Hello Bob", which she is not supposed to receive, since she is not a registered user and she is not Bob. But, 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. In fact, this problem has prevented the use of proxies in caching dynamic pages. Another problem with page level caching is the unnecessary invalidation, even if one or a few elements on a page become invalid, then the entire page becomes invalid. This means that there is often very little reusability of full HTML pages.

Dynamic page assembly is an approach popularized by Akamai [1] as part of the Edge Side Includes (ESI) initiative [27]. 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. A drawback of this approach is the requirement that a site follows a specified page design paradigm, specifically, the use of templates which in turn call separate dynamic scripts for each dynamically generated fragment, forcing that page layout to be known in advance. Thus, sites supporting dynamic layouts will not be able to take the advantage of dynamic page assembly. Another drawback of the dynamic page assembly approach is that it cannot be used in the context of pages with semantically interdependent fragments. The work in [42] can be considered as a dynamic page assembly approach. This work proposes a proxy cache that stores query templates, along with query results, which are used to manage the cache. This approach can only mitigate some delays associated with query processing, but it does not address the other delays associated with dynamic web page generation.

The proxy caching approaches are able to achieve significant bandwidth savings. However, their applicability in caching dynamic pages is rather limited and their primary use is in caching fixed layout content.

Web caching is an effective way to alleviate server bottlenecks, network traffic and the delay in fetching web objects. Web caching only enables reduction of the latency of accessing web objects that are in the cache, which are already accessed and retained for future use. An alternative technique to web caching is prefetching, where by predicting the object which is most likely to be accessed by the user and fetching it into the cache. By prefetching, the latency associated with objects that are not in the cache, but most likely to be accessed in the near future, can be reduced. Prefetching complements the web caching technique. It predicts the web objects most likely to be accessed by the user and fetches them into cache. For those web objects, which are correctly predicted and fetched into the cache, the delay becomes zero, which is the best that could be hoped for from the user perspective.

Prefetching can be applied in three ways with respect to WWW [69]. Namely, between clients and web servers, between proxies and web servers, and between browser clients and proxies. The prefetching of rich contents is the simplest prefetching scheme. The

benefits of prefetching them is imperative, since they are mostly likely to be requested. Other than this, prefetching can be done when the users/clients surf the websites with a pattern and not randomly. Such schemes typically use Markov models for predicting the next object that is accessed.

Prefetching of web objects poses some serious problems. Firstly, the prefetching increases the network bandwidth consumption, due to prefetching of objects which are not used. Secondly, it increases the server's load, by sending requests for all predicted objects that may not be subsequently used at all. Also most pages on any website are accessed only once (one-time accesses) and a lot of users access only one page from a particular website before moving on to a new website. Such behaviour increases the difficulty of predicting and prefetching the object that is likely to be accessed next.

The ever changing nature of the web, in particular the website, also tends to affect the performance of prefetching. In particular, users surfing pattern changes as the popularity of different web objects change over a period of time. For example, in a news web-server, such as *www.indianexpress.com*, the most recent and hot news items, and the related web objects, have higher popularity. However, the popularity of a specific object may fade with time. Also, as newer pages are added to the web, newer surfing patterns emerge. It is important to take these aspects of the web and individual web sites into account while designing a prefetching algorithm.

2.4 Server-Side Caching Solutions

The server-side caching solutions are based on the idea of caching dynamic content within the site architecture at various levels, to accelerate dynamically generated content. The server-side caching approaches can help to reduce the delays associated with dynamic web page generation. These solutions do guarantee the freshness of the output. By caching at the finer granularity, these solutions also achieve greater reuse of content and allow fine-grained invalidation. However, they do not address network-related problems due to dynamic web pages.

Some solutions take the approach of caching at the page-level of dynamically generated pages (e.g., [34, 62, 75]). The page-level caches can improve website performance by reducing delays associated with page generation. However, there are some major limitations associated with using page-level caching as discussed in Section 2.3.

Various types of database caching have been suggested, including caching the results of database queries [13, 42], caching WebViews [37], caching database tables [4] and caching database tables in main memory [50]. These solutions, of course, can address the delays associated with databases, but not other delays in dynamic web page generation and distribution.

In caching WebViews [37], an adaptive algorithm for the Online View Selection problem is presented. This helps at run time to determine which views should be materialized (cache and refresh immediately on updates) and which ones should just be cached and re-used as necessary. This solution concentrates on database updates and freshness of dynamic contents, but does not address the problem in general with dynamic web pages.

The work in [73] proposes a caching system that caches content at various levels. This work introduces the Weave management system developed at INRIA. Weave concentrates more on the declarative specification of websites and offers a number of tools for the easy implementation, deployment and monitoring of the specified site. Weave features a customizable cache system that implements the data materialization strategy according to the website's specifications: it can cache database data, XML fragments and HTML files. A limitation of this approach is that it requires the site to be designed using a particular declarative website specification language.

Another approach is presentation layer caching, which caches HTML fragments. Many application servers provide this type of caching capability (e.g., WebLogic from BEA Systems [6] and WebSphere from IBM [31, 41]), which can mitigate delays due to presentation layer tasks. But, it does not address other delays in dynamic web page construction.

An efficient technique to compose web pages from fragments for web based publications is given in [12]. Complex web pages are constructed from simpler fragments.

Fragment-based web publication allows parts of dynamic web pages to be cached. This work also presents algorithms for detecting and updating all affected web pages after one or more fragments change. It presents an automatic feature for publishing system for automatically and consistently publishing dynamic contents. It has been deployed at several popular websites, including the 2000 Sydney Olympics Games website [49]. This technique is a good solution for publishing dynamic content. The cached web page has to be regenerated even if one or more fragment become invalid and hence this solution has limited reusability. Since, cached dynamic web pages are identified by requesting URL, this technique may serve incorrect pages, if implemented in case of highly personalized websites. Since two requests for the same URL may turn out to provide two different dynamic web pages. Hence, this technique is useful for dynamic web sites with fixed page layouts.

Fragment caching technique is proposed in [19, 20]. This technique involves caching fragments of dynamically generated pages. A fragment is a part of a web page, which is common across one or more web pages. The success of fragment caching is based on the notion that many fragments of dynamic pages do not change for extended periods of time and hence can be cached for future requests. A cached fragment can be invalidated whenever the underlying data source corresponding to the cached fragment changes. A cached fragment is reusable across all users. In fragment caching, the page skeleton is generated for each user's request and hence serves the correct page. Fragment caching achieves reduction in dynamic page construction time by making use of cached fragments, instead of computing them afresh each time. On the down side, however, fragment caching has some limitations as discussed in Section 1.3.1.

Code caching is a recent technique that has been proposed and implemented in the context of the PHP Accelerator project [39]. Before executing a script, the engine reads, parses, and compiles the script into code ready for execution. Since, in practice, the scripts rarely change, this pre-processing is targeted for elimination in [39].

In PHPA, it is assumed that the compiled code uses instructions from a virtual

instruction set that is platform independent and once compiled, the code is executed by a virtual machine that interprets the instructions. In our context, it is also possible to conceive that since dynamic websites are usually meant for a specific task, the script can be directly compiled to the native machine, dispensing with platform independence. This compiled code can also be optimized before being put into execution to improve its efficiency and reduce its footprint. We assume that the code caching technique can be extended to the level of individual code blocks, where, instead of caching the entire compiled code of a script, only the compiled code corresponding to an executable code block is cached. High performance can be achieved by using a shared memory cache from which the compiled code can be executed directly. The code caching technique only concentrates on reducing the script execution time. But, it does not eliminate the execution itself.

Page pre-generation is a server-side alternative to dynamic web caching. Here, dynamic pages are generated based on some prediction prior to the request for the page. There are several page access prediction models that have been proposed in the literature [24, 35, 59, 63, 70, 77], based on information gained from mining web logs. These models can be classified into two categories: *point-based* and *path-based*. The point-based models predict the user's next request solely based on the current request being served for the user. On the other hand, the path-based prediction models are built on entire request paths followed by users. The path-based predictions use a path profile, which is a set of pairs, each of which contains a path and the number of times that path occurs over the period of the profile. The profiles can be generated from standard HTTP server logs and the accuracy of these models has been found to be high enough to justify the pre-generation of dynamic content [59] – in the rest of this thesis, we assume the use of such a path prediction model.

The server-side caching approaches briefly described above can help to reduce the delays associated with content generation. Since they reside at the origin server, these

solutions do guarantee the freshness of the content. Specifically, fragment caching solution also ensures correctness of the dynamic web pages, since the page skeleton is generated for each user request, after receiving the page request. However, they deliver all contents from the dynamic website and do not address network-related delays. These solutions still have considerable page construction times, which can perhaps be reduced further by integrating with various other solutions. We have carried out a variety of such integrations in this thesis.

2.4.1 Fragment Caching

Now, we explain in detail the fragment caching technique proposed in [19, 20]. This technique involves caching fragments or components of dynamically generated web pages. To understand fragments in a dynamically generated web page, consider a dynamically generated page in an application that provides TV listings. Such a page might have the following, easily reusable fragments: site navigation, local TV listings, editorial content, and external content. While the local TV listings fragment probably can be reused for all users in the same geographical location, the site navigation, editorial content, and external content will likely be reusable across all users. Thus, these fragments can be cached to serve future requests.

Based on this notion, a dynamic content accelerator (DCA) has been built [19, 20]. DCA is a server-side caching engine that caches such dynamic web page fragments to reduce page-generation processing delays. A set of prediction and observation-based techniques are used for cache replacement and invalidation. The substantially reduced processing load on the web application server lets it handle significantly higher user loads. The dynamic content accelerator (DCA) has two components: fragment-level caching and a set of intelligent cache management strategies based on prediction-based cache replacement and observation-based cache consistency.

Caching Dynamic Page Fragments

To address page-level caching issues, the DCA employs a fragment-level caching approach that focuses on reusing HTML fragments of dynamic web pages. To show how fragment-level caching works, let us first examine dynamic scripts in more detail. A dynamic script typically consists of a number of code blocks, each performing some work required to generate the page such as retrieving or formatting content and produces a HTML fragment as output. A *write to out* statement, which follows each code block, places the resulting HTML fragment in a buffer. When a dynamic script runs, each code block executes and the resulting HTML fragment goes into the buffer. Once all code blocks

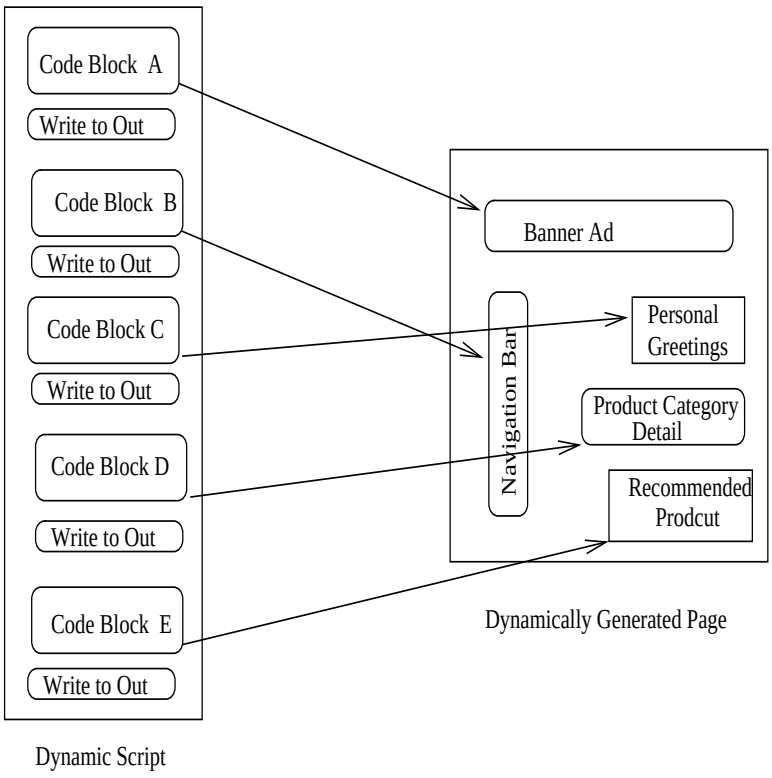


Figure 2.3: **A Template of Dynamic Script to Generate a Dynamic Web Page**

have been executed, the entire HTML page is transmitted as a stream to the user. The resulting page generated by the scripting process thus consists of a set of fragments. Figure 2.3 gives a high-level depiction of the dynamic scripting process. A particular code block in the dynamic script generates a corresponding component (for example,

code block A generates the "Banner Ad" component, code block B generates the "site navigation" component, and so on). When the script executes, each code block writes the HTML corresponding to its component into an output buffer, which then goes to the user for viewing.

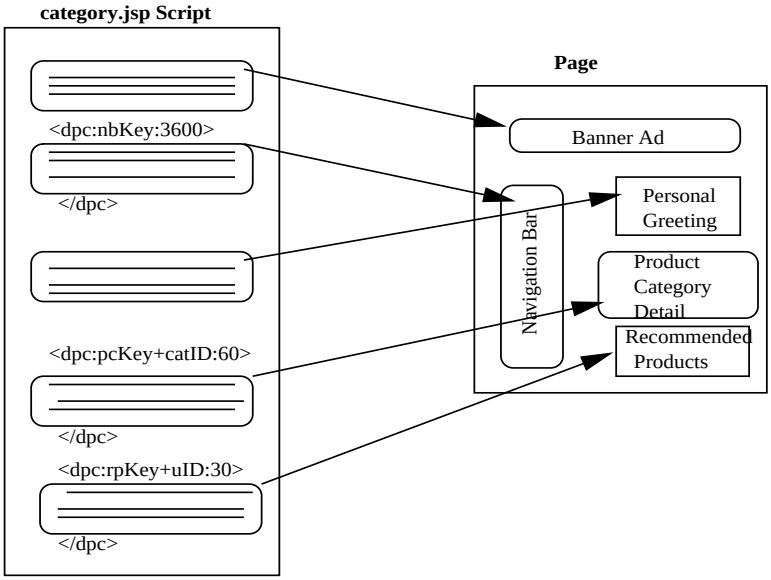


Figure 2.4: **A Template of Dynamic Script with Tagged Code Blocks for Caching**

Given the significant potential for reusing the page's components, each cacheable component is identified, which the application developer accomplishes by marking or tagging the corresponding code blocks in the script. For instance, Figure 2.4 depicts the tagging of a code block. Placing tags around a code block indicates that its output is cacheable. Thus, when the script executes, the tags instruct the application server to first check the DCA cache before executing the code block. If the server finds the requested fragment (corresponding to a tagged component) in the fragment cache and finds that it is fresh, it returns the fragment directly from the cache, bypassing the code block logic. If the server does not find the requested fragment in cache or finds that it is stale, the code block executes and the server generates the requested fragment and subsequently places it in the cache for future use, if it is tagged for caching.

Clearly, fragment caching solution requires identifying cacheable components. The

site designer ultimately has to make this decision. As part of the initial identification process, the designer has to assign each cacheable component a cache key. This key, along with the actual fragment content, resides in the cache. Other meta-data can also reside with each cache entry, such as the Time-To-Live (TTL) value. The fragment cache is implemented as a hash table, thus providing constant lookup time. Figure 2.5 shows a simplified depiction of an end-to-end website architecture with the DCA cache.

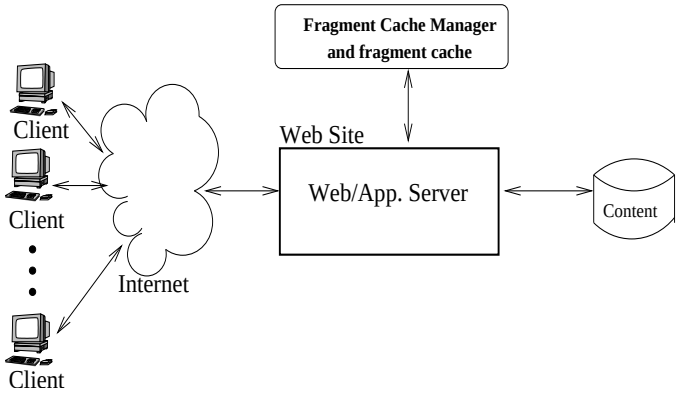


Figure 2.5: **The Existing Fragment Caching**

Prediction-Based Cache Replacement

Key to the DCA is a predictive cache management policy called Least-Likely-to-be-Used (LLU). When choosing a replacement victim, the policy considers not only how recently a cached item has been referenced, but also whether any user is likely to need it in the near future.

LLU cache replacement policy

The LLU cache replacement policy requires three types of information from a site: the site graph, current behavior of users on the site, and aggregated past behavior of users on the site. At startup, the cache is empty. As fragment sets are accessed, corresponding cacheable fragments are stored in the cache. When the total size of the cached fragments reaches the cache limit, cache replacement starts. For each new fragment set stored in the cache, the DCA cache manager must evict a (different) fragment set from the cache.

The fragment set chosen for replacement is the one with the lowest probability of access – the fragment set that is least likely to be used. If there are multiple fragment sets with this same probability value, the DCA cache manager chooses the fragment set with the oldest access time stamp.

Observation-Based Cache Invalidation

The DCA provides a number of intelligent cache invalidation strategies that keep cached content fresh as the underlying source data changes. Developers can combine these strategies as necessary to best suit a particular site. Keeping cached content fresh requires intelligent decisions regarding when to invalidate items and which items to invalidate. Existing solutions generally rely on three techniques to determine when to invalidate cache elements:

- **Time-based:** Each cache element is assigned a fixed TTL.
- **Event-based:** An update to some data source (such as a database system) invalidates an item in the cache. An invalidation message goes from the data source to the cache.
- **On-demand:** The site administrator manually invalidates either the entire cache or specific items.

The DCA has adapted these techniques to work in the context of DCA's component-level cache. In addition, the DCA has used a new invalidation scheme called observation-based invalidation. This technique is similar to event-based invalidation, except that the updates can be observed within the scripts. This is typically the case for sites where updates are made via the web application (such as auction or online trading sites). This approach is non-intrusive, requiring no communication between the data sources and the cache. Rather, invalidation messages go from the web application server to the cache.

Deciding which items to invalidate is another important decision. Sites might have data dependencies or other criteria that determine which items in the cache should be

invalidated. For instance, consider an online retail site that displays product information such as description and price. The site might have the following dependency: If price changes, then product information changes. System designers can specify such dependencies and use them to invalidate the appropriate items in the cache.

2.5 Comparison of Optimization Techniques

Here, we give a comparison of various optimization techniques for dynamic web contents. The advantages and disadvantages are summarized in Table 2.1.

2.6 Our Research

As mentioned above, research efforts on performance and scalability of dynamic web contents have gained momentum in the last few years. In our research work, we have concentrated on integrating some of these solution techniques to address the scalability and performance issues. We specifically integrate fragment caching technique with others, to address both *network consumption* and *dynamic web page generation time* problems.

2.6.1 Delays addressed by our techniques

Our proposed *dynamic proxy caching* technique addresses network bandwidth consumption and related delays (like firewall processing delays, switching delays). We achieve this by combining fragment caching with proxy caching. We move the bulky fragment responses to the proxy cache and generate the page layout and any fresh fragment at the origin server. By taking the dynamic contents to proxy, we are able to achieve reduction in bandwidth consumption. The dynamic page is completed at the dynamic proxy cache after filling the remaining fragment responses.

In our hybrid technique of fragment caching and anticipatory page pre-generation, we address server-side latency. In this technique, during normal loading, we pre-generate a

most expected page for the current user, so that for subsequent user request pre-generated page can be served in case of correct prediction. In this case, the page construction time is zero, which is the best that could be hoped for from the user perspective.

The proposed integration of fragment caching with code caching reduces page construction times during both normal loading and peak loading. In this technique, while assembling a dynamic page from fragment responses (either taking fragment response from fragment cache or freshly generating by executing fragment script), the code caching is used to cache the compiled codes. So that, for subsequent code execution, the time consuming code interpretation part can be avoided by executing the compiled code directly from code cache. From a broad perspective, by integrated caching we are achieving the long-term benefits whenever the fragments or their associated compiled codes are reused in course of time.

Our thesis contributions are cumulative. We can achieve all the above benefits together. That is, both *reduced bandwidth consumption achieved at proxy-side*, and *reduced page construction time achieved at server-side*. In the next three chapters we present our research work.

Table 2.1: **Comparison of Optimization Techniques**

Techniques	Advantages	Disadvantages
Proxy-Based Caching	• Reduce response time • Reduce bandwidth requirements	• May serve stale content • May serve incorrect pages.
i) Page-level caching	• Reduce response time • Reduce bandwidth requirements	• May serve stale content • May serve incorrect pages • Have limited reusability
ii) Dynamic page assembly	• Reduce response time • Reduce bandwidth requirements	• A site requires a specific page design paradigm • Not applicable to sites supporting dynamic layouts
Prefetching	• Reduce response time	• Consume bandwidth • Increase spurious load on the server
Server-Side Caching	• Guarantee freshness	• Do not address bandwidth requirements
i) Page-level caching	• Reduce response time	• May serve incorrect pages • Have limited reusability
ii) Fragment caching	• Reduce response time • Serve correct pages • Achiev greater reusability	• Depend upon cacheability • Do not address bandwidth requirements • Page construction starts only after receiving page request
iii) Database caching	• Address delays associated with databases	• Do not address other delays in dynamic web page generation and distribution
iv) Weave management system	• Cache at various levels	• Site to be designed using a particular declarative website specification language
v) Fragment based web publications	• Automatically and consistently publishes dynamic contents	• Limited reusability • May serve incorrect pages
vi) Code Caching	• Reduce script execution time	• Do not eliminate execution itself
Page pre-generation	• Reduce response time	• Increase load on the server, if not controlled

Chapter 3

Dynamic Proxy Caching

In this chapter, we present an approach and an implementation of a Dynamic Proxy Caching (DPC)[1] technique which combines the benefits of both proxy-based caching and server-side caching approaches, yet does not suffer from their individual limitations. Our dynamic proxy caching technique allows granular, proxy-based caching, where both the content and layout can be dynamic. Specifically, we describe an architecture for such a system, as well as the data structures and algorithms needed to make it work. We show the effectiveness of our system by studying its performance analytically and experimentally. A performance analysis of our approach indicates that it is capable of providing significant reductions in bandwidth consumption. The work presented in this chapter refers to proxy-side caching, shown as *Dynamic Proxy Caching* in Figure 1.4, in Section 1.4, with a coupling between contents stored at proxy-cache and data origin, without extra communication cost to refresh cache contents. Our experimental results validate our analytical findings.

Organization

The remainder of this chapter is organized as follows: In Section 3.1, we provide a detailed description of the architecture of proposed dynamic proxy. In Section 3.2, the analytical results are shown. The experimental results are presented in Section 3.3. A case study

[1]This work was carried out during a summer internship at Chutney Technologies, USA.

is discussed in Section 3.4. Extensions to this work are discussed in Section 3.5. Finally, we conclude in Section 3.6.

3.1 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.

3.1.1 Intuition

Our objective is to deliver dynamic pages from proxy caches. Recall that dynamic pages are "dynamic" across two dimensions: they possess 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, (much like what is depicted in Figure 2.2), 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 requester.

Figure 3.1 illustrates this process and helps to clarify the intuition. Consider again a request for the fiction category page, as described in Section 2.2. The request (Step 1 in Figure 3.1) passes through the dynamic proxy cache, which routes it to the origin

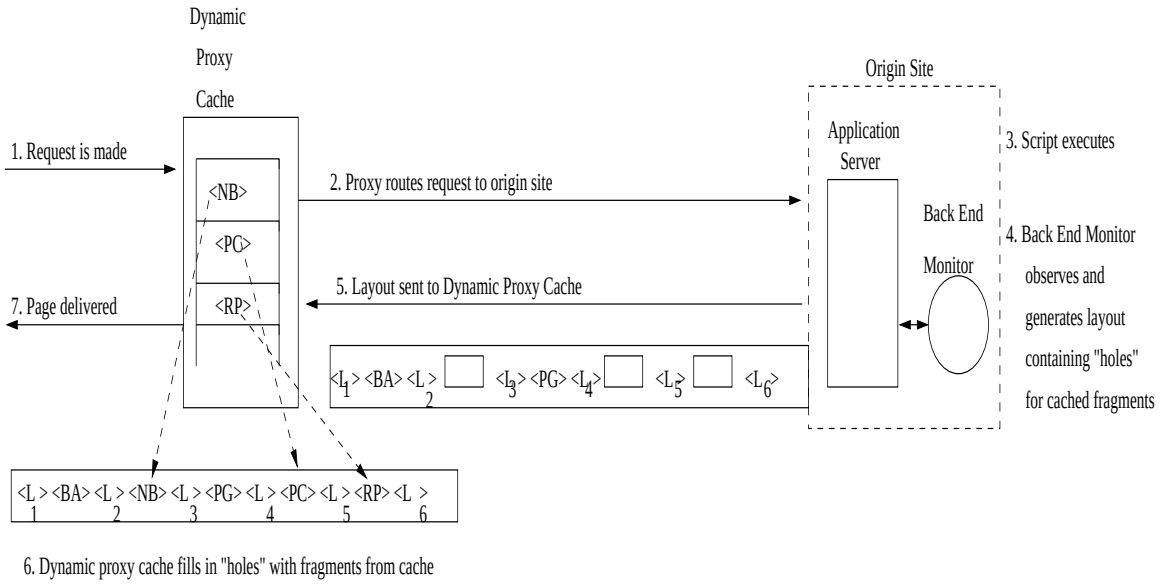


Figure 3.1: **Overview of Dynamic Proxy Caching System**

site (Step 2). At the origin site, the application server executes the category.jsp script to serve this request (Step 3). A monitor at the back-end observes the application processing and generates the page layout accordingly (Step 4). This layout will contain "holes" to indicate where the cached fragments should be inserted. In Figure 3.1, these "holes" are shown as the empty boxes in the layout. This layout is sent to the dynamic proxy cache (Step 5), which fills the "holes" with the appropriate fragments from its cache (Step 6). The resulting page is then delivered to the user (Step 7).

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.

3.1.2 System Architecture

Figure 3.2 illustrates the architecture of our system. Note that this architecture is similar to the one shown in Figure 1.1, with two additional modules (the shaded modules): the *Dynamic Proxy Cache* (the shaded squares) and the *Back-End Monitor* (the shaded

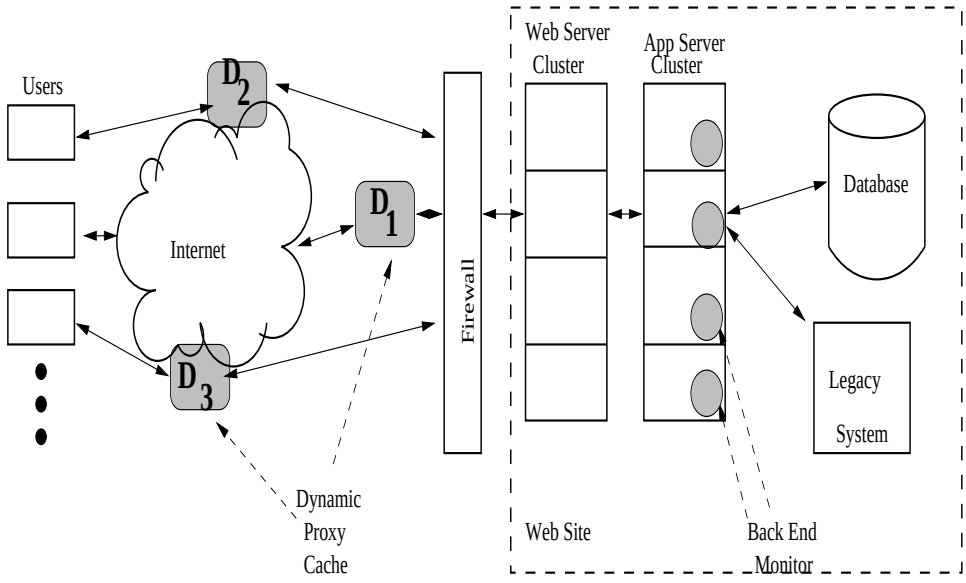


Figure 3.2: **Dynamic Proxy Cache Architecture**

circles). 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 chapter, 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 contents, are transmitted from the back-end to the DPC.

As Figure 3.2 shows, the DPC can reside either (a) at the origin site (in a reverse proxy configuration as in the case for the DPC labeled D_1), or (b) at the network edge (in a forward proxy configuration, as in the case for DPCs D_2 and D_3). 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 Content Delivery Networking (CDN)), 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. Having described the system architecture, we now delve into the technical details.

3.1.3 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.

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 as described in Section 2.4.1. Figure 2.4 shows the category.jsp script with the tags inserted.

As this figure shows, each tag contains a *begin* and *end* tag which surround a particular code block.

Now, we describe the format of these tags. A begin tag has the following basic format: <dpc:fragmentID:ttl>, where dpc is a constant which indicates the start of a tag, *fragmentID* is a unique string to identify the fragment, and *ttl* is an optional time-to-live value. The *fragmentID* can be further decomposed into two elements: name and parameter-List, where name is a name assigned to the fragment and parameterList is an optional list of run-time parameters. For example, as shown in Figure 2.4, the **Navigation Bar** code blocks has been assigned the *fragmentID* nbKey, consisting of only the fragment name and a *ttl* of 1 day (3600 minutes). The **Product Category Detail** fragment has been assigned a *fragmentID* consisting of pcKey (the name) and the catID parameter (the parameterList), delimited by +. Thus, at run-time, in our running example, the request would result in this *fragmentID* being instantiated as pcKey+catID=Fiction. The end tag is constant and has the format:</dpc>. Note that these tags can be easily extended to incorporate additional functionality (e.g., keywords to support keyword-based invalidation).

Run-Time Operation

At run-time, a user submits a request to the site. This request, e.g., http://www.booksOnline.com/category.jsp?categoryID=Fiction is passed through to the application server. This causes the category.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 meta-data 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.

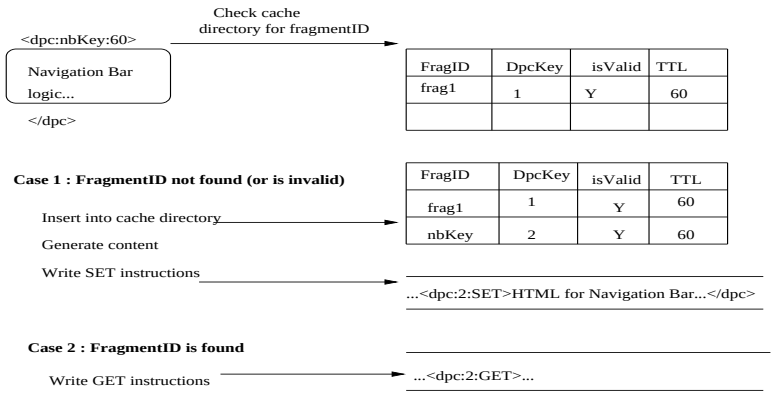


Figure 3.3: **Serving Requests**

Figure 3.3 illustrates this run-time operation for the **Navigation Bar** code block in our running example. As this figure shows, a cache directory lookup is done for the nbKey *fragmentID*. If the *fragmentID* is not found or is invalid (Case 1 in Figure 3.3), 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 (labeled as DpcKey in the figure). The **Navigation Bar** 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 (Case 2 in Figure 3.3), 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. A mockup of the first and second requests for the example is shown in Figure 3.4 for comparison, assuming that all fragments are found in cache for the second request. Note that the size of the layout for the second request is smaller than that of the first request, even for this simple example.

Template for First Request	**Template for Subsequent Reuqest**
<HTML>...HTML for Banner Ad... <dpc:1:set>HTML for Navigation Bar...</dpc> ...HTML for Personal Greeting <dpc:2:set>HTML for Product Category Detail...</dpc> <dpc:3:set>HTML for Product Recommendations...</dpc> </HTML>	<HTML>...HTML for Banner Ad... <dpc:1:get> ....HTML for Personal Greeting... <dpc:2:get> <dpc:3:get> </HTML>

Figure 3.4: **Example Templates**

Having described the run-time operation of our system, we are now ready to discuss the system components in greater detail.

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 meta-data. The cache directory has the following basic structure:

fragmentID	unique fragment identifier (name+parameterList)
dpcKey	unique fragment identifier assigned by Key Mapping algorithm
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.

The *dpcKey* is assigned at run-time using a *Key Mapping algorithm*, which we now describe. The *dpcKey* is an integer value drawn from some pool of integers (1,2,...,N), allocated at system initialization. The maximum key value, N, is chosen such that it provides an upper bound on the number of cacheable fragments. Typically, N is composed by dividing available memory by the average size of a fragment. This pool of integers is maintained as queue, which we refer to as the freeList. When a run-time request is made, a check is done to see whether the *fragmentID* exists in the cache directory. If the *fragmentID* does not exist or exist but *isValid* is FALSE, i.e., a SET operation is required, a new entry is inserted into the cache directory as follows: the *fragmentID* and *ttl* are instantiated based on the system initialization configuration described previously, and *isValid* is set to TRUE. If the freeList is not empty, the *dpcKey* for the fragment is

assigned by taking the next available integer from the head of the freeList, otherwise, the *dpcKey* value is one more than the maximum key value used so far. This *dpcKey* (obtained through mapping) is inserted into the page template and used as the key in the DPC. This procedure is shown as the SET procedure in Algorithm 1. The content obtained for the fragment is also sent to DPC along with *dpcKey*. This content will be cached at the DPC and identified with corresponding *dpcKey*. If the *fragmentID* exist in the BEM cache directory and *isValid* is TRUE, then the corresponding *dpcKey* is taken from the cache directory and the *dpcKey* is inserted into the page template with GET instruction to DPC, so that the content for the *dpcKey* should be obtained from the DPC cache. The DPC cache is implicitly flushed, whenever a mapped number is reused in course of time.

Algorithm 1: Key Mapping Algorithm

Note: Maximum size of freeList $\geq$ maximum cache size/average-fragment-size.

Input:

freeList having maximum size n

maxUsed: number of occupied slots in freeList

```
1: INIT
2: freeList.maxUsed=0
3: SET
4: if fragmentID is not mapped then
5:     if freeList is empty then
6:       dpcKey = maxUsed
7:       increment maxUsed
8:     else
9:       dpcKey = freeList.pop
10: REMOVE
```

```
11: if fragmentID is mapped then
12:     freeList.insert(dpcKey)
```

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 invalidation 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 FALSE to indicate that it is no longer valid. When this occurs, the *dpcKey* for the fragment is inserted at the freeList. This technique ensures that a subsequent request for the fragment will be generated and served fresh. This procedure is shown as the REMOVE procedure in Algorithm 1. It should be emphasized that our Key Mapping Technique is independent of the invalidation and replacement policy utilized.

Note that the size of the freeList should be at least as large as the maximum cache size/average-fragment-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 the **Navigation Bar** fragment, 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 the **Navigation Bar** fragment or a new fragment) by the BEM, at which time the appropriate content will be inserted into the corresponding slot in the DPC.

In addition to managing the **DPC**, the **BEM** can also cache certain types of objects if needed. In some cases, it may be beneficial to cache intermediate objects rather than user deliverable fragments. For instance, consider our earlier example in Section 3.1, where a user profile object is used to generate both the **Personal Greeting** and the

Recommended Products fragments. The site may choose to cache the intermediate user profile object so that it can be used across multiple requests by that user. The BEM supports caching of such objects. In fact, the BEM is capable of caching any arbitrary object that is serializable. Objects that are cacheable are tagged in a manner similar to the tagging method described in Section 2.4.1. However, these objects are given a special identifier to indicate that they are to be cached in the BEM and not sent to the DPC (since the DPC is not able to manipulate arbitrary objects).

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.

3.2 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. Table 3.1 contains the notation to be used throughout this section.

Table 3.1: **Notations**

Symbol	Description
$\varepsilon = \{e_1, e_2, ..., e_m\}$	set of fragments
$\mathcal{C} = \{c_1, c_2, ..., c_n\}$	set of pages
$E_i = \{e_j : e_j \epsilon c_i\}$	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 information (bytes)
h	hit ratio, i.e., fraction of fragments found in cache
R	total number of requests during observation period

In our analysis, we wish to compare the bandwidth savings for two cases: (a) with the dynamic proxy cache and (b) without the dynamic proxy cache. We next describe our assumptions and derive a generic expression for the number of out-bound bytes served by a given website 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 2.2 that a dynamic script generates pages. For the purposes of this analysis, we model a given web application as a set of such pages $\mathcal{C} = \{c_1, c_2, ..., c_n\}$. Each page is created by running a script (as described in Section 3.1.3), and the resulting page consists of a set of fragments, drawn from the set of all possible fragments, $\varepsilon = \{e_1, e_2, ..., e_m\}$. We let E_i, $E_i \subseteq \varepsilon$, be the set of fragments corresponding to page c_i. Referring back to our running example, the page shown in Figure 2.1(a), i.e., the Fiction category page consists of several fragments, e.g., **Banner ad, Navigation Bar**. There exists a many-to-many mapping between $\mathcal{C}$ and ε, 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*, $\bar{B}$ as the average number of bytes served by the website that is hosting the application during some time interval. In other words, $\bar{B}$ is the number of bytes transferred between the back-end and the DPC during some period of interest. To compute $\bar{B}$, 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. One question that may arise is the impact of static content on bandwidth requirements. In most modern websites, static content (e.g., images, HTML files) is served from a proxy cache outside the site infrastructure. Thus, in steady-state, static content will be served from a proxy cache and therefore will not impact bandwidth requirements between the web server and the proxy cache. For this reason, we do not incorporate the bandwidth requirements for static content in this

analysis.

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 accessed during the specified time interval. Then the general form of $\bar{B}$ 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.booksOnline.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. Our general expression for expected bytes served then becomes:

$$\bar{B} = \Sigma_{i=1}^{n} S_{c_i} \times \mathcal{P}(i) \times \int_{t_1}^{t_2} f(t)dt \tag{3.1}$$

We assume that $\mathcal{P}$(i) is governed by the Zipfian distribution, which has been shown to describe web page requests with reasonable accuracy [3, 18]. Thus, the probability that page c_i is accessed for a given request, $\mathcal{P}$(i), is given by 1/(i $\times\Sigma_{i=1}^{n}\frac{1}{i}$).

To characterize the arrival rate of requests, $\int_{t_1}^{t_2}$f(t)dt, we do not assume any particular distribution. Rather, any distribution can be used in Equation 3.1. For the purposes of this analysis, we consider the number of requests R that occur during some period of interest. For instance, we may wish to determine the bandwidth requirements during a period when there are R = 1,000,000 requests that arrive to www.booksOnline.com.

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 $\Sigma_{\forall_{e_j}\epsilon c_i} s_{e_j} + f$, which is

the sum of the sizes of all the fragments on the page and the header information. Putting this all together, the expected bytes served for the no cache case can be expressed as:

$$\bar{B}^{NC} = R \times \Sigma_{i=1}^{n} \mathcal{P}(i) \times S_{c_j}^{NC} \tag{3.2}$$

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:

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

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 $\Sigma_{\forall_{e_j} \epsilon c_i}[X_j[(h \times g) + (1-h)(s_{e_j} + 2g)] + (1 - X_j)(s_{e_j}) + f]$. The first term,$X_j(h \times g) + (1-h)(S_{e_j} + 2g)$, represents the case where the jth fragment is cacheable. In this case, there are two possible outcomes at run-time (a) the fragment is found in cache, in which case the size is the size of the tag, g, or (b) the fragment is not found in cache, in which case the size includes the size of the fragment as well as the size of the two tags required (the start and end tags). The second term in this expression, $(1 - X_j)(s_{e_j})$, represent the case where the fragment e_j is not cacheable and hence, the size simply the size of the fragment.

Putting this all together yields the following expression for expected bytes served in the case where the dynamic proxy cache is used:

$$\bar{B}^{C} = R \times \Sigma_{i=1}^{n} \mathcal{P}(i) \Sigma_{\forall e_j \epsilon c_i}[X_j[(h \times g) + (1-h)(s_{e_j} + 2g)] + (1 - X_j)(s_{e_j}) + f] \tag{3.3}$$

We have compared the expected bytes served for the two cases using the baseline parameter values shown in Table 3.2. 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 [3, 18]. Furthermore, our experience with several large enterprise web applications indicates that such hit ratios are easily achievable in practice.

Table 3.2: **Baseline Parameter settings for Analysis**

Parameter	Value
hit ratio (h)	0.8
fragment size (s_e)	1Kbytes
number of fragments per page	9
number of page instances	250,000
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

In this comparison, we plot the ratio $\frac{\bar{B}^C}{\bar{B}^{NC}}$. Figure 3.5 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). This can also be used as a guideline to implement fragment caching. If most of the fragments in a dynamic web site are smaller in size and take very less time to compute, then better to turn off the fragment caching.

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 3.6 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).

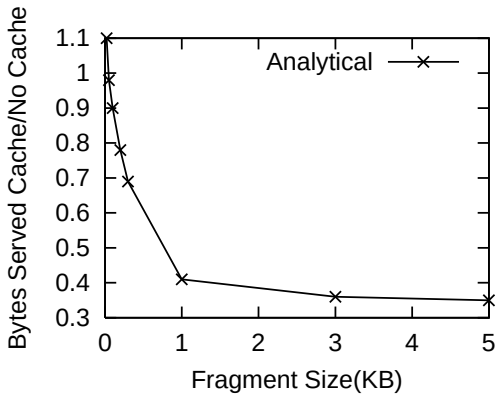


Figure 3.5: **Analytical Results - Expected Bytes Served : $\bar{B}^C/\bar{B}^{NC}$ vs. Fragment Size**

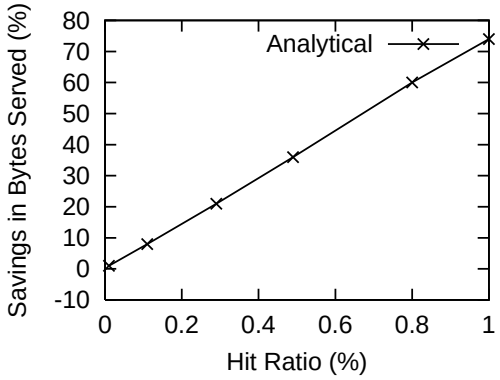


Figure 3.6: **Analytical Results - Expected Bytes Served : Expected Bytes Served (%) vs. Hit Ratio**

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:

$$scanCost^{NC} = \bar{B}^{NC} \times y \tag{3.4}$$

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 $scanCost^{C} = \bar{B}^{C}(y + z)$. Both the firewall and the dynamic proxy cache scan a given string of bytes. Since string matching algorithms (e.g., KMP [36]) are linear-time algorithms, we can consider the scanning cost for the firewall and the dynamic proxy cache to be of the same order. Thus, we assume that z ≈ y. Making this substitution, our expression for the scan cost per byte for the dynamic proxy cache becomes:

$$scanCost^{C} = \bar{B}^{C} \times 2y \tag{3.5}$$

In comparing our expressions for the cost to scan in both Equations (3.4) and (3.5), we expect the dynamic proxy cache to provide better performance when the following condition holds: $\bar{B}^{NC} > 2\bar{B}^{C}$. 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.*

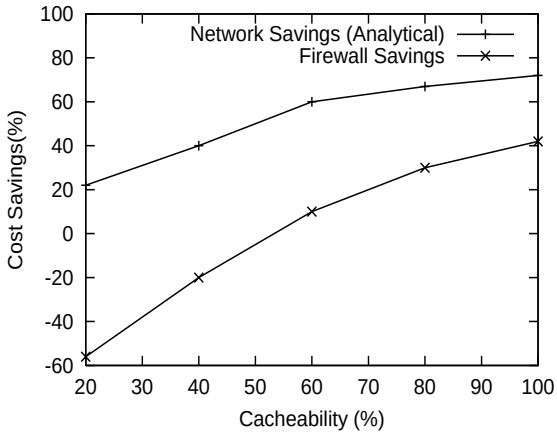


Figure 3.7: **Analytical Results : Comparison of Cost Savings**

Figure 3.7 shows a comparison of Equations (3.4) and (3.5) as the *cacheability factor*

is varied (using again the parameter settings in Table 3.2). The cacheability factor is the percentage of all fragments that are cacheable for a given application. Thus, the 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 saving 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.

3.3 Experimental Results

In this section, we attempt to validate our analytical results obtained in Section 3.2 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 [45] 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 as described in Section 3.2. Thus, we have incorporated the parameter settings in Table 3.2. The test site is an ASP-based site which retrieves content from a site content repository.

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 3.8 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).

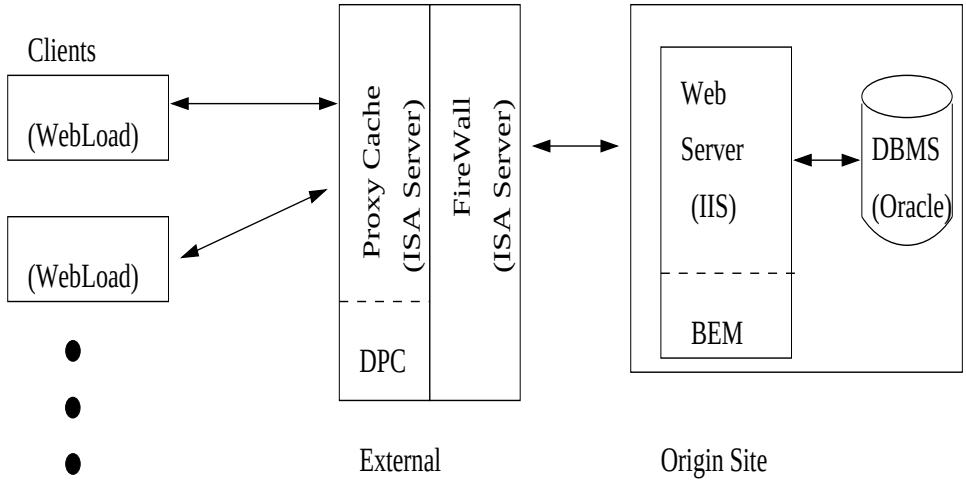


Figure 3.8: **Test Configuration**

The number of bytes served is obtained by measuring bandwidth using the Sniffer network monitoring tool [61]. More precisely, the bandwidth measurement is taken between the Origin Site machine and the External machine in Figure 3.8. 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 from the ISA server proxy cache and therefore will not impact bandwidth requirements between the web server and the DPC.

Figure 3.9 shows the ratio $\frac{\bar{B}^C}{\bar{B}^{NC}}$ as fragment size is varied. Our results from Section 3.2 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,

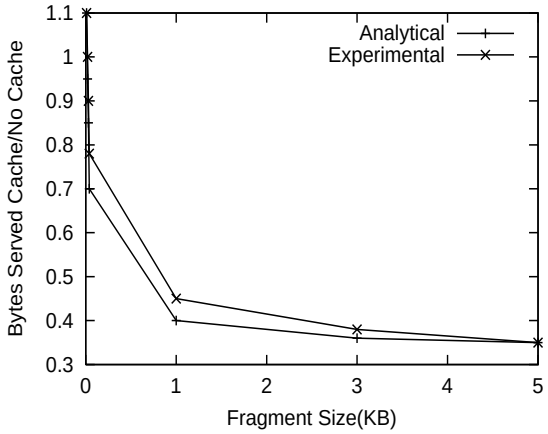


Figure 3.9: **Analytical and Experimental Results : $\bar{B}^C/\bar{B}^{NC}$ vs. Fragment Size**

which the Sniffer tool captures in its bandwidth measurements. However, we do not account for these headers in our analytical expressions. Thus, for 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 3.2, we now examine the sensitivity of expected bytes served to changes in hit ratio. Figure 3.10 shows a comparison of this sensitivity for the analytical (curve repeated from Figure 3.6) and experimental cases. Here again, our experimental results closely follow our analytical 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 3.11 shows a comparison of the sensitivity of expected bytes served to changes in cacheability. The analytical curve is repeated from Figure 3.7 (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,

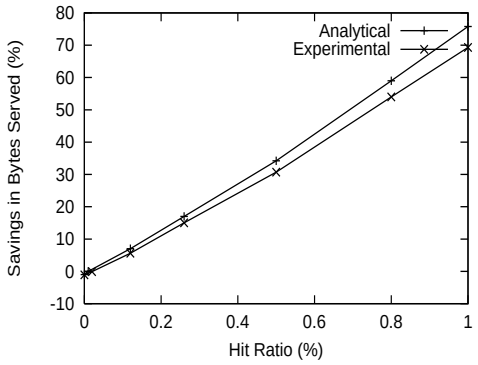


Figure 3.10: **Validation of Analytical Results : Expected Bytes Served (%) vs. Hit Ratio**

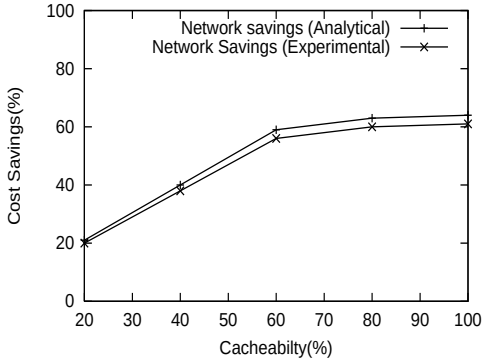


Figure 3.11: **Validation of Analytical Results : Validation of Cost Savings**

which cause the analytical curve to be higher than the experimental curve.

3.4 Case Study

In this section, we present a case study which describes the use of our dynamic proxy caching system in a large enterprise. We have deployed our system on a web application at a large financial institution. The application provides a variety of financial information to end users by retrieving data from a number of data sources such as database systems, content management systems, and remote web servers. Pages in the site are generated using ASP scripts.

We performed a set of experiments using this application to compare the performance of our proxy-based caching system with that of a system without any caching. Each page

selected for the experiments consists of 9 fragments, including **Market Index, Market Forecast, Research News**, and **Financial Resources.** Each fragment is generated from a different data source. The average size of each fragment is about 1 KB. For the dynamic proxy cache, all of the 9 fragments are tagged as cacheable. The **Market Index** fragment has a *ttl* of 5 minutes, while all other fragments have ttls of 30 minutes. An important aspect of this application, like many existing dynamic web applications, is that both the page layout and content are dynamic. The page layout and content depend upon the user's profile and the time of day. For instance, a user who requests a page in the morning gets a page containing today's **Market forecast** at the top of the page, the **Market Index** on the left side of the page, **Research News** and **Financial Resources** in the center of the page, and the remaining fragments on the bottom portion of the page. When the same user request this page at the end of the day, the page contains today's **Market Summary** and **Market Highlights** at the top of the page, tomorrow's **Market Forecast** on the left of the page, and so on.

The test configuration used here is the same basic configuration shown in Figure 3.8. Performance metrics used are *bandwidth*, in bytes per second, and *average response time*, the end-to-end delay in delivering an HTML page. In our experiments, we vary load, measured in transactions per second (TPS). As in our experiments in Section 3.3, bandwidth is measured between the web server and the DPC using Sniffer. Also, as in our previous experiments, static content is served from the ISA Server proxy cache and thus does not impact our bandwidth measurements. Average response time is measured using the WebLoad load testing tool [55].

Figure 3.12 shows the performance results in terms of bandwidth as load is varied. Note that the y-axis is shown on a log scale. In both the no cache and cache cases, bandwidth increases as load increases. However, the no cache curve is much higher than the cache curve. As load increases beyond about 10 TPS, the dynamic proxy cache results in an order-of-magnitude reduction in bandwidth.

Given that the proxy-based cache requires additional scanning of the page templates,

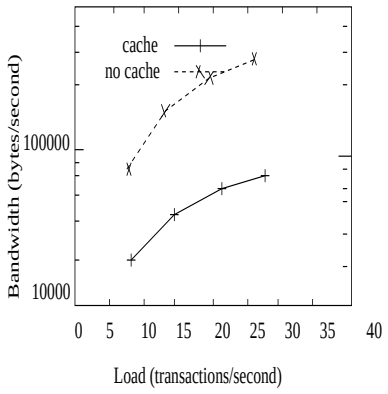


Figure 3.12: **Experimental Results : Comparison of Bandwidth**

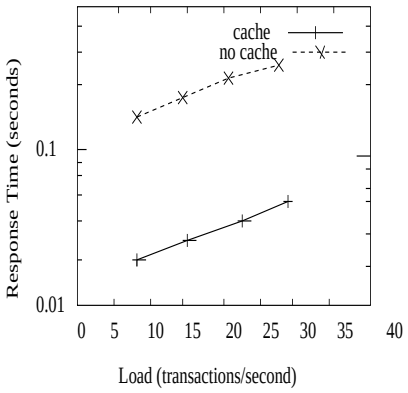


Figure 3.13: **Experimental Results : Comparison of Response Times**

it is of interest to examine the impact on response times. Figure 3.13 shows the performance results in terms of average response time as load is varied. As this figure shows, the cache case significantly outperforms the no cache case under the conditions used in our experiments. In fact, the cache case provides an order-of-magnitude reduction in average response times over the entire range shown.

3.5 Extensions to this Work

The approach considered in this chapter assumes a reverse proxy configuration, in which a single instance of the cache sits between a website'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 this 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 fail-over 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 multiple 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.

- Storage Vs. Bandwidth tradeoff: When we extend the present model from single proxy to multiple proxies, we have to consider the tradeoff bewteen the storage requirements and bandwidth savings.

Some of these issues are addressed in the recent work reported in [22].

3.6 Conclusions

In this chapter, we have proposed an approach for granular, proxy-based caching of dynamic content, a proxy-side caching with a coupling between contents stored at proxy-cache and data origin, without extra communication cost to refresh cache contents. 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 server-side 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.

Chapter 4

Hybrid Caching

4.1 Overview

The previous chapter dealt with proxy-based caching. We now turn our attention to server-side caching. Server-side caching is particularly attractive for dynamic websites, since it can guarantee freshness of contents. In this chapter, we propose an architecture to address the dynamic page generation delays by a hybrid technique of fragment caching and anticipatory page pre-generation. The work presented in this chapter refers to server-side caching, shown as *Hybrid Caching* in Figure 1.4, in Section 1.4.

We consider here mechanisms for reducing the server latency, during normal loading, by utilizing the excess capacity with which web servers are typically provisioned [59]. We propose a hybrid architecture of fragment caching and anticipatory page pre-generation. Specifically, we present a caching technique that integrates fragment caching with anticipatory page pre-generation in order to deliver dynamic pages faster during normal operating situations. A feedback mechanism is used to tune the page pre-generation process to match the current system load. The experimental results from a detailed simulation study of our technique indicate that, given a fixed cache budget, page construction speedups of more than fifty percent can be consistently achieved as compared to a pure fragment caching approach, during normal loading.

Here, we consider the possibility of achieving significant reductions in server latencies,

and thereby user response times, by resorting to dynamic page *pre-generation*, in conjunction with fragment caching. The page pre-generation is based on having a statistical prediction mechanism for estimating the next page that would be accessed by a user during a session. The page pre-generation is executed during the time period between sending out the response to the user's current request and the receipt of her subsequent request. Note that in the case where the page prediction turns out to be right, the page pre-generation effectively reduces the server latency to *zero*, which is the best that could be hoped for from the user perspective.

An unsuccessful page pre-generation on the other hand represents wasted effort on the part of the server. This may not be an issue for web-servers that are under normal operation since these systems are usually over-provisioned in order to handle peak loads [59], and therefore some wastage of the excess capacity is not of consequence. But, during peak loads, the additional effort may further exacerbate the system performance. To address this problem, we incorporate a simple linear feedback mechanism that scales down the degree of page pre-generation to match the current system load. Here, for feedback mechanism, we consider only the load on the application servers, assuming that the back-end servers are optimized with techniques like table caching, query caching and so on.

A related design issue is that we need to allocate space in the server cache to store the pre-generated pages. That is, the cache has to be *partitioned* into a fragment space and a page space, and the relative sizing of these partitions has to be determined.

Our hybrid approach of combining fragment caching with page pre-generation ensures the freshness of content through either fresh computation or by accessing fragments from the fragment cache. Further, it ensures the correctness of pages by pre-generating pages specific to individual user. In a nutshell, our approach achieves both the *long-term benefit through fragment caching* and the *immediate benefit through anticipatory page pre-generation*.

Using a detailed simulation model of a dynamic web-server, we study the performance of our hybrid approach in terms of reducing dynamic page construction times,

as compared to pure fragment caching and pure page pre-generation approaches. Our evaluation is conducted over a range of fragment caching levels and prediction accuracies, for a given cache budget. The results show that under normal loads, we are able to achieve reductions in server latency by over fifty percent on average as compared to pure fragment caching, whereas under heavy loads, we do no worse. Further, the number of pages delivered with *zero server latency* is proportional to the prediction accuracy.

Contributions

In summary, the contributions of this chapter are the following:

1. We propose a hybrid caching approach of combining fragment caching with page pre-generation to reduce dynamic webpage construction times.

2. We demonstrate that robust settings exist for the relative sizing of the cache partitions for pre-generated pages and fragments, respectively.

3. We incorporate a simple linear feedback mechanism to ensure that the system performance is always as good or better than that of pure fragment caching.

4. Our experimental results show that significant improvements in page generation times can be achieved through the hybrid caching approach as compared to pure fragment caching.

Organization

The remainder of this chapter is organized as follows: In Section 4.2, we discuss the hybrid architecture-namely fragment caching with anticipatory page pre-generation along with a load-thresholding feedback system. Section 4.3 describes an analytical model. The simulation model to evaluate the various alternatives is described in Section 4.4. The experimental results are highlighted in Section 4.5. Finally, in Section 4.6, we summarize our contributions and outline future research avenues.

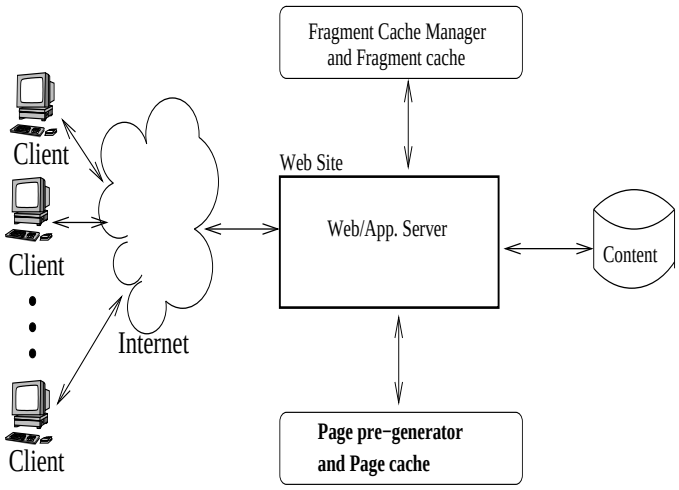


Figure 4.1: **The Proposed Hybrid Model**

4.2 A Hybrid Approach to Dynamic Page Construction

In this section, we describe in detail our proposed hybrid caching architecture. This architecture is based on fragment caching (described briefly in Section 2.4.1) and page pre-generation (described briefly in Section 2.4).

4.2.1 Combining Page Pre-Generation and Fragment Caching

Our proposed hybrid model is an integration of anticipatory page pre-generation and fragment caching. A high level representation of the proposed hybrid architecture is given in Figure 4.1.

In a typical dynamic web server, there are various components in the system leading to server latencies. Application servers interact with back-end servers like database servers and file systems. There are various solutions to address latencies due back-end servers like caching the results of database queries, caching database tables, caching database tables in main memory and so on [13, 42, 4, 50]. In our simulation, we assume that the back-end servers are already optimized and they are not the bottlenecks in the system. Hence, for feedback mechanism, we consider only the load on the application servers.

Here, for each individual user session, when a response for a request leaves the web server, the web server decides whether or not to pre-generate the next most expected page for the associated user, based on considerations such as the system's current load, the benefit of pre-generating a page, the type of user and so on. If the system decides to pre-generate a page for a particular user, it requests the page pre-generator to carry out the page pre-generation for this user. When the web server receives the next page request from this user, it checks whether the page is already available with the page pre-generator. If the page is available, the page is immediately served. If not, the page is freshly computed by the web/application server as usual. The page pre-generator retains only the pre-generated pages of current active users.

Note that the user response leaving the web server will take some time to reach the user and the user will then take some time to click the next page. We expect that under normal operating conditions, this time delay is sufficient for the page pre-generator to complete the page pre-generation process before the arrival of the next request of the same user. The implication is that in case of a correct prediction, the server latency in terms of page construction time is *brought down to zero*.

Further, note that the proposed solution is guaranteed to serve *fresh* content, since it is associated with the origin server. Moreover, it also ensures serving *correct* pages, since the page pre-generation is specific to the user session and is not generic across users. From a broad perspective, by fragment caching we are achieving the long-term benefit whenever the fragment is reused in course of time, whereas by page pre-generation we are achieving the immediate benefit for the current user.

4.2.2 Server Cache Management

In a pure fragment caching approach, the server cache can be used solely for hosting these fragments. However, in our hybrid approach, we need to allocate space for hosting pre-generated pages as well. Therefore, we partition the cache into a *fragment cache* and a *page cache* (here, a page we mean HTML page, but not memory page).

We consider two separate caches for fragments and pages, since they differ in both in

replacement policy and reusability. Fragments can be reused across different pages and they are managed by a replacement cache policy. Whereas, cached pages are specific to individual user, hence no-reusability. Since, cached pages are also short lived do not require any replacement policy.

Cache Partition Sizing:

An immediate issue that arises here is determining the relative sizes of the fragment and page cache partitions. This issue is investigated in detail in our experimental study presented in Section 4.5 – our results there indicate that a 50-50 partitioning works well across a range of page prediction accuracies and fragment cacheability levels.

Cache Replacement Policies:

With regard to the fragment cache, we are not aware of any web logs that are available to track the reference patterns for fragments. This restricts us to the use of simple techniques like Least Recently Used (LRU) for managing the fragment cache.

With regard to the page cache, we do not expect to require an explicit replacement policy since the utility of pages in the cache is typically short-lived – that is, until the arrival of the next request by the user – after this arrival, the page is immediately vacated from the cache. However, to address those uncommon cases where the page cache is completely filled with active pre-generated pages, we adopt the simple mechanism of blocking further page pre-generations until some of the existing pages expire.

An association between the fragments in the fragment cache and the pre-generated pages in the page cache is maintained by the page pre-generator. Whenever a fragment is invalidated, all the pre-generated pages associated with it are marked invalid.

4.2.3 Server Load Management

While page pre-generation is useful for reducing response times, it also involves expense of computational resources. This is acceptable under normal operating conditions, even if the page prediction accuracy is not good, since web-servers are typically over-provisioned

in order to be able to handle peak load conditions [59], and we are only using this excess capacity. But, when the system is under peak load conditions, the wasted resources due to the mistakes made by the page pre-generation process may actually exacerbate the situation, driving the system into *a worse condition*. To address this issue, we implement a simple linear feedback mechanism that modulates the pre-generation process to suit the current loading condition. Specifically, we periodically measure the system load, and if it exceeds a threshold value, the role of the page pre-generator is restricted in proportion to the excess load. For feedback mechanism, we consider only the load on the application servers, assuming that the back-end servers are optimized with techniques like database table caching, query caching and so on.

We have applied a simple linear feedback mechanism in our hybrid model to control the role played by the page pre-generator during the peak loads. Specifically, for each outgoing page response, the web server allows the page pre-generator to generate pages with probability *prob_gen* set as follows:

$prob\_gen = 1$ if $(current\_load \ < \ threshold\_load)$

$prob\_gen = \frac{100 - current\_load}{100 - threshold\_load}$ otherwise.

When the page pre-generator is restricted, its assigned *cache partition* may become underutilized – therefore the size of the fragment cache is dynamically enlarged to cover the underutilization of the page cache.

4.3 Analytical Results

The benefit that we expect from our hybrid caching architecture is *reduced dynamic web page construction time*. In this section, we analyze this benefit. Table 4.1 contains the notations used in this section.

In our analysis, we wish to compare the the dynamic web page construction times for four cases : (a) with no optimization; (b) with only fragment caching in place; (c) with only page pre-generation in place; and (d) the proposed hybrid technique of fragment caching and page pre-generation.

Table 4.1: **Notations for Hybrid Caching Analysis**

Symbol	Description
$\varepsilon = \{e_1, e_2, ..., e_m\}$	set of fragments
$\mathcal{C} = \{c_1, c_2, ..., c_n\}$	set of pages
$E_i = \{e_j : e_j \epsilon c_i\}$	set of fragments corresponding to page c_i
s_{e_j}	average size of fragment e_j (bytes)
W_Q	average waiting time in queue

We next describe our assumptions and derive a generic expression for the average time taken to generate a dynamic web page by a given web site infrastructure, when no optimization is in place. We then derive specific expressions for each of the remaining three cases.

Recall from our discussion in Section 2.2 that a dynamic script generates pages. For the purposes of this analysis, we model a given web application as a set of such pages $\mathcal{C} = \{c_1, c_2, ..., c_n\}$. Each page is created by running a script (as described in Section 3.1.3), and the resulting page consists of a set of fragments, drawn from the set of all possible fragments, $\varepsilon = \{e_1, e_2, ..., e_m\}$. We let E_i, $E_i \subseteq \varepsilon$, be the set of fragments corresponding to page c_i.

In a dynamic web server, normally, there will be a number of applications servers. Basically, a dynamic web server receives the dynamic page requests from different browsers. The web server simply selects one of the available application servers to generate the required page. Here, the dynamic web-server is nothing more than a FCFS queue, since all it does is to queue the dynamic web page request and submit it to one of the available application servers. To the best of our knowledge, there are no web logs which logs the fragment script execution times. The web log analysis reported in [2], states that according to an analysis of real world environment, the distribution of the response times, obtained from the log file, is generally characterized by a very high number of values with very low response times and a few values with very high response times, that, it is a Zipfian distribution. Taken in conjunction with the arrival process at web server, which is usually modelled as Poisson [5], we characterize our system as an M/M/k queue.

The parameter of our interest is the average page construction time, which is essentially the average waiting time of page requests in the queue (W_Q) and the average of the actual page construction time (script execution time). For M/M/k queuing model, the average waiting time in queue W_Q is given by the below equation [68]:

$$W_Q = \frac{\rho(k\rho)^k \pi_0}{k!\lambda(1-\rho)^2} \tag{4.1}$$

where λ is the arrival rate, k is the number of servers in the system, ρ is defined as: $\rho = \frac{\lambda}{k\mu}$, and μ is the service rate.

Suite of Algorithms

To put the performance of our approach in proper perspective, we compare it against the following four yardstick algorithms:

Hybrid: This is our new algorithm in which page pre-generation and fragment caching are simultaneously used, and the cache is partitioned into a page cache and a fragment cache.

Pure_FC: This algorithm implements pure fragment caching (with no page pre-generation).

Pure_PG: This algorithm implements pure page pre-generation (with no fragment caching).

NO_FC_PG: Neither fragment caching nor page pre-generation is used here, and the cache does not come into play at all.

We define $T^{NO\_FC\_PG}$ as the average time taken by the website that is hosting the application to generate dynamic web pages without any optimization technique in place. We also define T^{Pure_FC} as the average time taken by the website when only fragment caching is used, T^{Pure_PG} as the average time taken by the website when only page pre-generation is used, and T^{Hybrid} as the average time taken by the website when both fragment caching and page pre-generation used, i.e., in our proposed hybrid caching architecture.

Now, we proceed to derive expressions for the above four cases. We obtain the expression for the base case where no optimization is employed and the dynamic web page is constructed from the scratch after receiving the request by the web site. In this case,

$$T^{NO\_FC\_PG} = W_Q + \Sigma t_{e_i} \tag{4.2}$$

where W_Q is the waiting time of the page request, given by Equation 4.1 and t_{e_i} is the time taken by a fragment e_i for its generation.

We now obtain the expression for the case where only fragment caching is used. In this case,

$$T^{Pure\_FC} = W_Q + n^{'} \times cst + (1 - n^{'}) \times \Sigma t_{e_i} \tag{4.3}$$

where $n^{'}$ is the fraction of fragments used from cache, cst is the cache search time for fragments used from the fragment cache and $(1-n^{'})$ is the fraction of fragments generated fresh and W_Q is same as above.

We now obtain the expression for the case where only page pre-generation is used. In this case,

$$T^{Pure\_PG} = W_Q + (\Sigma t_{e_i}) \times (1 - prediction factor) \tag{4.4}$$

where predictionfactor is the page prediction accuracy used and W_Q is same as above.

We now obtain the expression for our proposed hybrid cache, where both fragment caching and page pre-generation are used. In this case,

$$T^{Hybrid} = W_Q + (n^{'} \times cst + (1 - n^{'}) \times \Sigma t_{e_i}) \times (1 - prediction factor) \tag{4.5}$$

We have compared the expected time taken for four cases using the baseline parameter values shown in Table 4.2, the basis for parameter values is described in Section 4.4. It is very difficult to obtain exact time of a fragment computation, since it depends upon various factors, like the time of computation, the load on the system, and so on. Hence we assume a representative average value. This is not a problem, since our primary focus is comparing *relative* performance. The analytical results are shown in Figure 4.2, which

Table 4.2: **Baseline Parameter settings for Analysis (Hybrid Caching)**

Parameter	Value
average fragment size (s_{e_j})	2Kbytes
average number of fragments per page	9
average page size (s_e)	20Kbytes
cacheability	50%
page predictability	50%
number of application servers	4
average fragment generation time	20ms

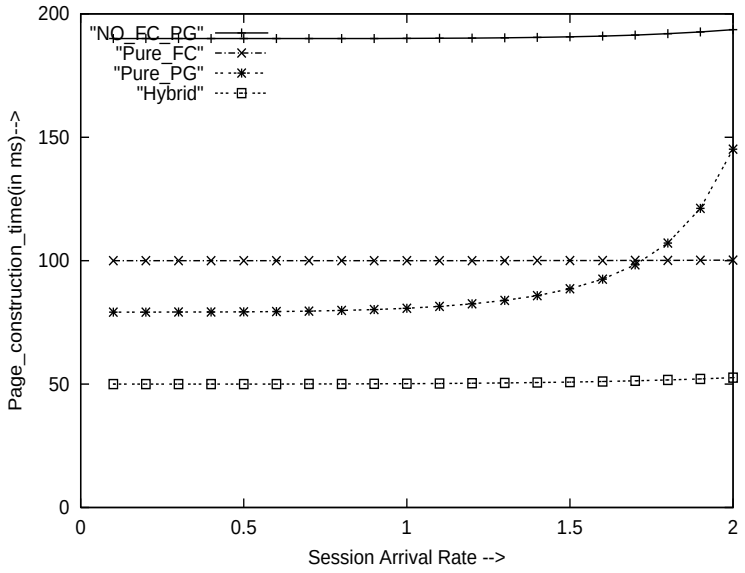


Figure 4.2: **Analytical results: Hybrid Caching**

gives the relative performance of the four dynamic webpage construction algorithms. We see here that the Hybrid approach (labeled as Hybrid in the figure) performs the best compared to all others and further, it requires less than the half the time to construct pages as compared to fragment caching (labeled as Pure_FC in the figure), the policy that has been advocated in the recent literature.

The foregoing results indicate that the hybrid caching is indeed beneficial in terms of reducing the dynamic web page construction time, during normal loading.

4.4 Simulation Model

To evaluate the performance of the proposed hybrid model, we have developed a detailed simulator of a web-server supplying dynamic pages to users. Table 4.3 gives the default values of the parameters used in our simulator – these values are chosen to be indicative of typical current websites, with some degree of scaling to ensure manageable simulation run-times. Some of these parameters are worked out from the web log statistics reported in [60]. For example, we have taken average page size 20KB and averagae session length 10 based on this statistics. Fragment size taken as 2KB based on the observation reported in [56]. The fanout is fixed based on a conservative value from [38] The average user think time between requests is set to 5 seconds based on [58]. Some of these parameters values have been set larger to account for worst case. Cache size is set to 2MB, so that ratio between cache size and fragment space is 1:8. If we consider bigger cache, we can achieve even better performance. Page prediction is varied from 20 percent to 80 percent and similarly fragments cacheability is varied from 20 percent to 80 percent. We also vary the cache given to fragment cache and page cache from zero percent to 100 percent, so that we can determine best partition size. Some of the parameters are set based on our experience in analyzing some web pages.

4.4.1 Web-site Model:

The website is modeled as a directed graph. Each node in the graph represents a dynamic webpage. Each edge represents a link from one page to another page. A node may be connected to a number of other nodes. The website graph is generated in the following manner: We start with a node called the root node, at level zero, and an initial fanout $FanOut$. Then, at each level l, for all nodes of that level, the next level nodes are created and linked, with a uniform random fanout ranging between $(0, FanOut - l)$. When a fanout of 0 is chosen at a node, the generation process at that node is terminated. In order to model "back-links", we permit, in the process of linking a node to other nodes, even the *previously generated nodes* of the prior levels to be candidates. The percentage

of back links is determined by the *BackLinks* parameter.

4.4.2 Web-page Model:

Each dynamic web page consists of a static part and a collection of identifiable dynamic fragments. A fraction *FragCacheable* of these dynamic fragments are cacheable, while the remaining are not. The number of fragments in a page are uniformly distributed over the range *(MinFragNum, MaxFragNum)* and are selected randomly from the *FragPopulation* fragments. The cost of producing a fragment is taken to be exponentially distributed with mean time for fragment generation given by *FragCost*.

4.4.3 User Model:

The website receives requests from the sessions of different users. The creation of sessions is assumed to be Poisson distributed [5] with rate *ArrRate*. Each session generates one or more page requests, in a sequential manner. The number of pages in a session are uniformly distributed over the range *(MinSessionPage, MaxSessionPage)*. Between the page requests of a session, a uniformly distributed user think time over the range *(MinThinkTime, MaxThinkTime)* is modeled.

4.4.4 System Model:

We assume that the web server has a cache for dynamic page construction, of size *CacheSize*. The fraction of the cache given to the Page-Cache is given by *PageCacheFraction*, with the remainder assigned to the Fragment-Cache. The search times in the page and fragment caches are determined by the *CacheSearchTime* parameter. The accuracy of page access prediction is determined by the *PagePredict* parameter. The fragments in the fragment cache are modeled to be invalidated randomly by the data source with an invalidation rate set by *InvalidRate*. The threshold load at which the feedback control mechanism kicks in is set by the *ThresholdLoad* parameter.

Table 4.3: **Simulation parameter settings (Hybrid Caching)**

MinSessionPage	1	MaxSessionPage	19
MinPageSize	10KB	MaxPageSize	30KB
MinFragNum	1	MaxFragNum	19
MinThinkTime	1 second	MaxThinkTime	9 seconds
MinFragSize	1KB	MaxFragSize	3KB
FragPopulation	8000	CacheSize	2MB
PageCacheFraction	0 to 100 percent	FragCost	20 ms
FanOut	10	BackLinks	20 percent
ArrRate	0 to 5 sessions per second	InvalidRate	1/ms
PagePredict	20, 50, 80 percent	CacheSearchTime	0.1 ms
FragCacheable	20, 50, 80 percent	ThresholdLoad	75 percent

4.5 Experiments and Results

Using the simulation model described in Section 4.4, we conducted a variety of experiments, the highlights of which are described here. The performance metric used in all our experiments is the average dynamic page construction time, evaluated for various settings: *LOW (20%), MEDIUM (50%)* and *HIGH (80%)* of the page prediction accuracy and the cacheability of the dynamic fragments, as a function of the session arrival rate and the fraction of the cache assigned to page pre-generation. Covering these variety of values permits the modeling of a range of real-life website environments. Also, the arrival rates are set so as to model both normal loading conditions as well as peak load scenarios. The simulator was written in C++SIM [8], an object-oriented simulation testbed. The experiments were conducted on Ultra-Sparc/Solaris 2.6 workstations.

4.5.1 Experiment 1: Page Construction Times (Normal Load)

In our first experiment, we evaluate the dynamic web page construction times under normal loading conditions. Here, both the fragment cacheability level and the page prediction accuracy are set to MEDIUM (50 percent), and the cache memory is *equally* partitioned between the page cache and the fragment cache. For this scenario, Figure 4.3 gives the relative performance of the four dynamic webpage construction algorithms as

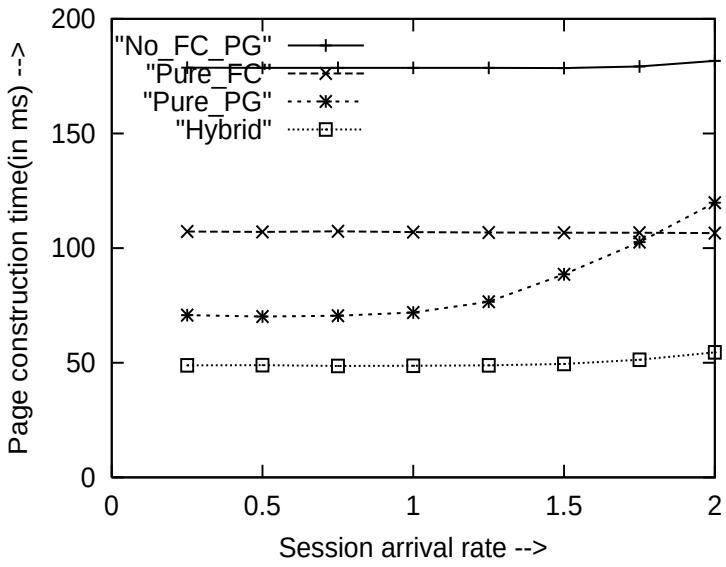


Figure 4.3: **Page Construction Times : Normal Load**

a function of the session arrival rate. We see here that:

- The HYBRID approach performs the best across the entire normal loading range. Further, it requires less than *half the time* to construct pages as compared to fragment caching, the policy that has been advocated in recent literature.
- The utility of caching and page pre-generation are indicated by the significant improvement in performance that are provided by HYBRID, Pure_PG and Pure_FC, as compared to No_FC_PG which is completely impervious to caching/page pre-generation.
- While the performance of HYBRID, Pure_FC and No_FC_PG is flat across the loading range, the Pure_PG approach begins to progressively do worse as the load is increased. This is because of the extra load that is imposed by the pre-generation process. In contrast, HYBRID, while also incorporating page pre-generation, does not suffer from the problem because of its fragment caching component.
- Our experimental results follow our analytical results closely, derived in Section 4.3.

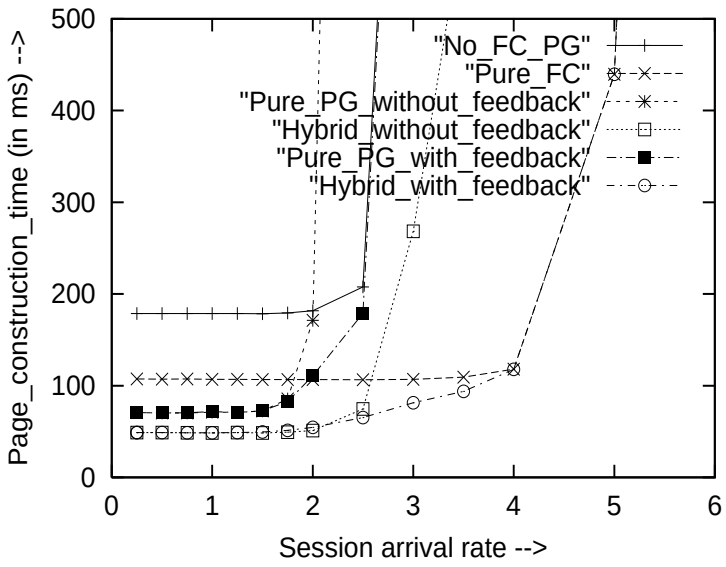


Figure 4.4: **Page Construction Times : Peak Load**

4.5.2 Experiment 2: Peak Load Performance

We now evaluate the performance under transient peak load situations which all web-servers experience from time to time. For this experiment, we present the performance of the HYBRID and Pure_PG approaches, both with and without the feedback mechanism, to evaluate the effectiveness of this mechanism. The page construction performance for this experiment is shown in Figure 4.4. We see here that:

- The HYBRID-with-feedback approach performs the best across the entire loading range. As the load moves into the peak-loading region, this approach progressively reduces the role of page pre-generation, finally winding up eliminating it completely and becoming identical to Pure_FC.
- The benefits of feedback are clearly shown by comparing the with-feedback and without-feedback versions of HYBRID and Pure_PG.

4.5.3 Experiment 3: Cache Partitioning

We now investigate the performance impact on HYBRID of different cache partitionings – this is done over the entire range of fragment cacheability levels (Low,Medium and

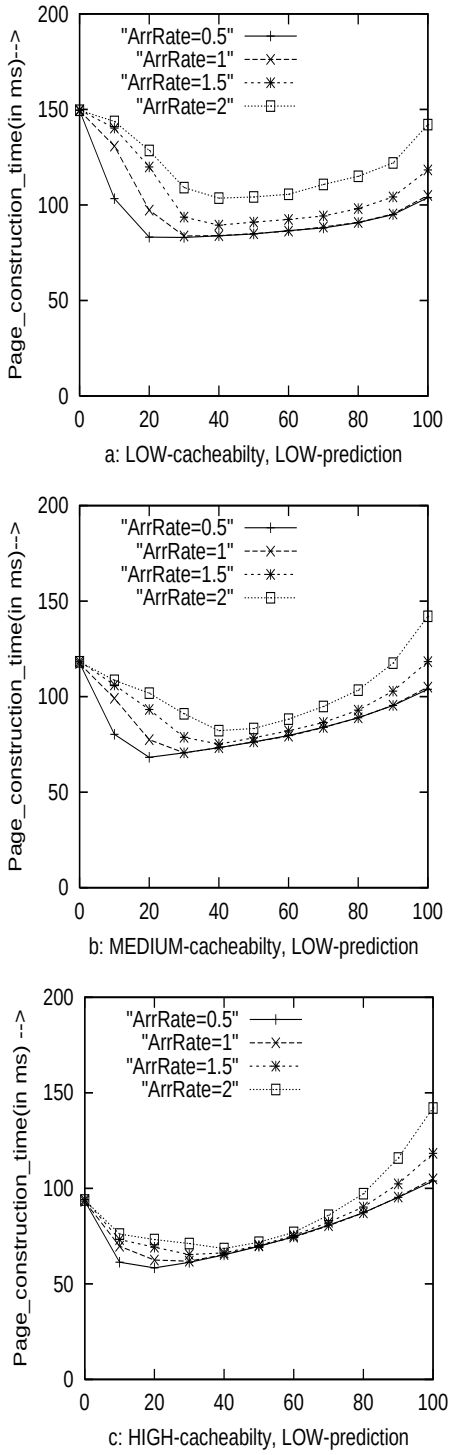


Figure 4.5: **Cache Partitioning (LOW Prediction)**

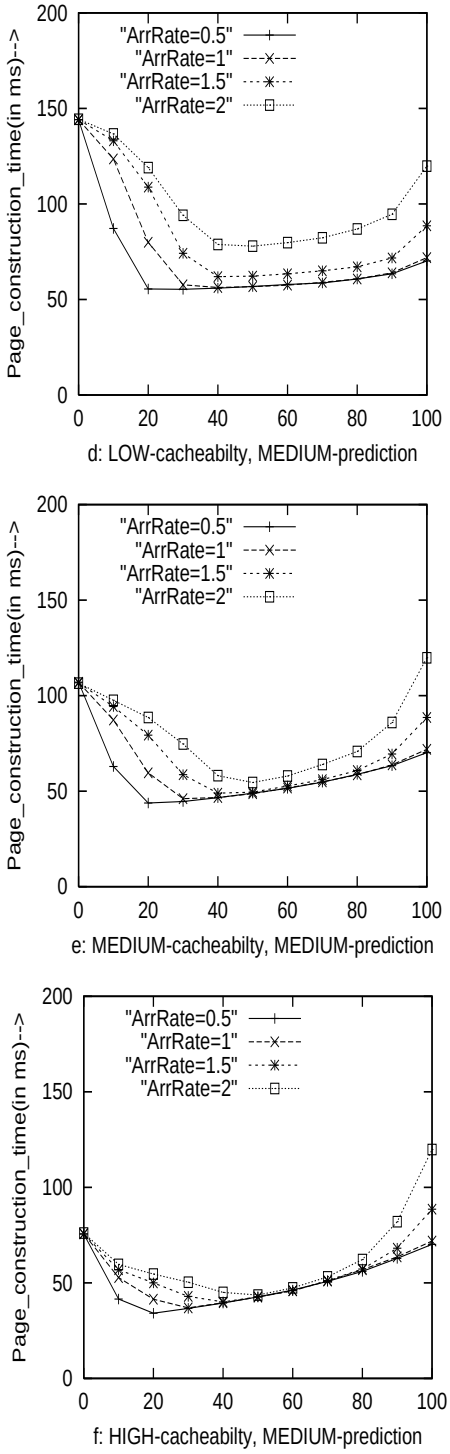


Figure 4.6: **Cache Partitioning (MEDIUM Prediction)**

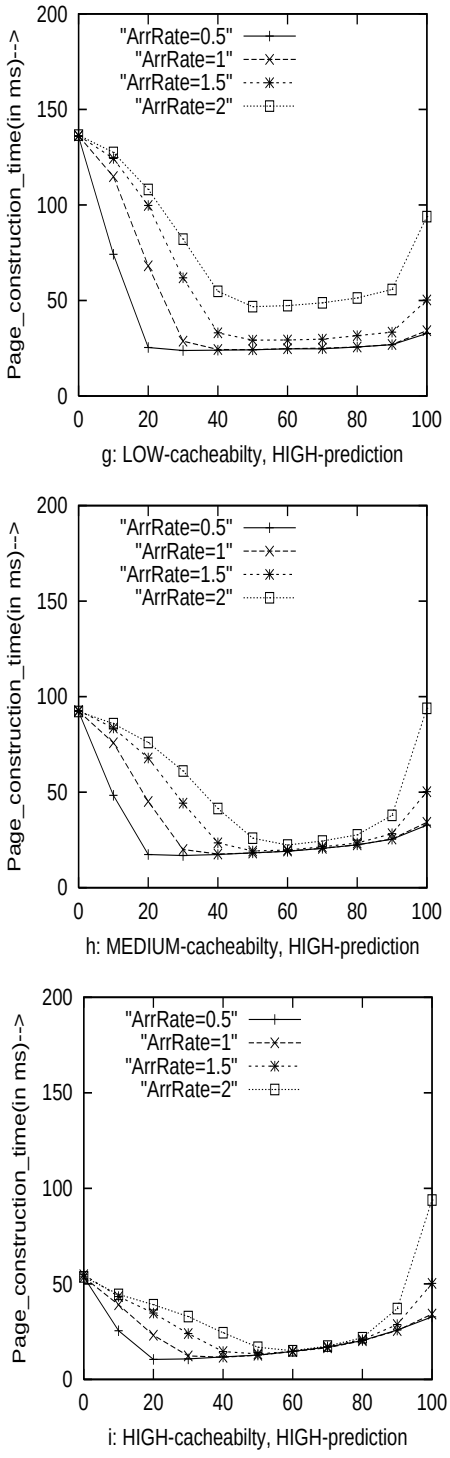


Figure 4.7: **Cache Partitioning (HIGH Prediction)**

High) and page prediction accuracies (Low, Medium and High), resulting in nine different combinations. The results for all these combinations are shown in Figures 4.5 through 4.7, where we observe the following:

- All of them have a "cup shape" with the highest construction times being at the extremes (0% page cache and 100% page cache), and the lowest somewhere in between.
- For the LOW prediction scenario (Figure 4.5), the best overall partitioning is about 40 percent page cache, while for the MEDIUM page prediction scenario (Figure 4.6), the best partitioning is 50 percent page cache and for the HIGH page prediction scenario (Figure 4.7), the best partitioning is 60 percent page cache.
- While the best partitionings are a function of the prediction accuracy as mentioned above, using a value of 50 percent page cache is very *close to the best in all the graphs*. That is, with this setting we are assured almost-optimal performance across the entire range of web-server scenarios.
- Note that the setting of 0 percent page cache is equivalent to a *Pure_FC* approach. We observe that the performance of Pure_FC is strongly dependent on the fragment cacheability level.

4.6 Conclusions

In this chapter, we have proposed a hybrid caching approach to reduce dynamic web-page construction times by integrating fragment caching with anticipatory page pre-generation, utilizing the spare capacity with which web servers are typically provisioned. Through the use of a simple linear feedback mechanism, we ensure that the peak load performance is no worse than that of pure fragment caching.

We made a detailed study of the hybrid approach over a range of cacheability levels and prediction accuracies, for a given cache budget. Our experimental results show that an even 50-50 partitioning between the page cache and the fragment cache works very

well across all environments. With this partitioning, we are able to achieve over fifty percent reduction in server latencies as compared to fragment caching. In summary, our approach achieves both the long-term benefit through fragment caching and the immediate benefit through anticipatory page pre-generation.

Currently, we restrict the pre-generation to the single most likely page. It would be interesting to investigate the performance effects of pre-generating a set of pages, rather than just a single page.

Chapter 5

Integrated Caching

5.1 Overview

In this chapter, we propose another server-side caching approach, by integrating two caching techniques – fragment-caching [19, 20] and code-caching [39]. The experimental results from a detailed simulation study of our techniques indicate that, given a fixed cache budget, the proposed integrated caching performs significantly better than the caching techniques in isolation. We also consider augmenting integrated caching with anticipatory page pre-generation in order to deliver dynamic web-pages faster during normal operating situations, by utilizing the excess capacity with which web-servers are typically provisioned [59]. The work presented in this chapter refers to a server-side caching, shown as *Integrated Caching* in Figure 1.4, in Section 1.4.

In summary, we investigate in this chapter the possibility of achieving significant reductions in server latencies, and thereby user response times, by a combination of fragment-caching and code-caching, optionally augmented with anticipatory page pre-generation.

Our approach ensures the *freshness* of content through either fresh fragment computation or by accessing fragments from the fragment cache, and the *correctness* of the page contents by newly generating the page skeleton each time the dynamic web-page

is requested. Overall, our goal is to achieve the *long-term benefits through the integrated fragment and code-caching*, and the *immediate benefit through anticipatory page pre-generation*.

Using a detailed simulation model of a dynamic web-server, we study the performance of our integrated caching approach in terms of reducing dynamic web-page construction times, as compared to pure fragment-caching and pure code-caching approaches. Our evaluation is conducted over a range of fragment-caching levels for a given cache budget. The results show that the integrated caching approach is able to achieve significantly better reductions in server latency as compared to pure fragment-caching and pure code-caching approaches. Our experimental results also show that a hybrid technique of integrated caching and anticipatory page pre-generation, reduces response times even further during normal loading, and does no worse during peak loading if augmented with a load-thresholding feedback mechanism.

Contributions

In summary, the contributions of this chapter are the following:

1. We propose an integrated fragment-cum-code-caching approach to reduce dynamic web-page construction times. Our experimental results show that significant improvements in page generation times can be achieved through this integrated approach as compared to pure fragment caching and pure code-caching approaches.

2. We demonstrate that, given a fixed cache budget, robust settings exist for the relative sizing of the cache partitions for fragments and compiled codes.

3. We extend the integrated fragment-cum-code-caching with anticipatory page pre-generation. Our experimental results show that significant improvements in response times can be achieved during normal loading through this process. The feedback process ensures that no adverse affects are caused during sporadic heavy load situations.

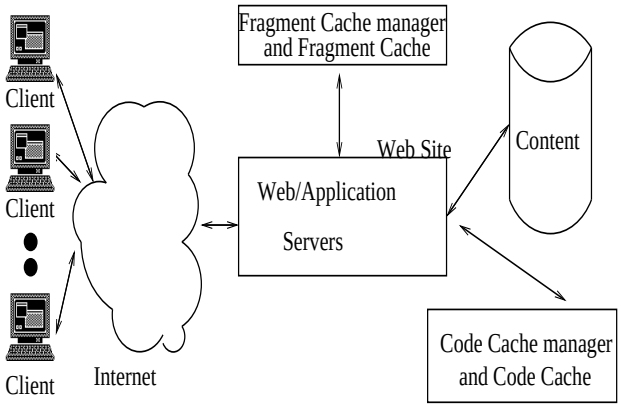


Figure 5.1: **The Proposed Integrated Caching Model**

Organization

The remainder of this chapter is organized as follows: In Section 5.2, we discuss the integrated fragment and code-caching technique. The incorporation of page pre-generation along with a load-thresholding feedback system is discussed in Section 5.3. Section 5.4 describes an analytical model. The simulation model to evaluate the various alternatives is described in Section 5.5. The experimental results are highlighted in Section 5.6. Finally, in Section 5.7, we summarize our contributions and outline future research avenues.

5.2 Integrated Fragment and Code Caching

In this section, we describe in detail our proposed integrated caching architecture. An associated problem is cache partitioning between fragment cache and code cache. We study this problem by varying code cache size. Our proposed integrated caching model is a simple combination of fragment-caching and code-caching. A high level representation of the proposed integrated caching architecture is given in in Figure 5.1.

Here, a request for a dynamic page triggers the execution of the corresponding script. While executing the script, for all those fragments of the script for which the outputs are

already available in the fragment cache, the execution of the fragment is bypassed, and we save on the fragment execution time. For those fragments for which the outputs are currently not available in the fragment cache, or the output is invalid, or the fragment is not cacheable, the fragment code must be executed from scratch.

While executing the fragment code, the web-server must first interpret and compile the fragment script and then execute it. In our proposed integrated caching approach, we cache the compiled fragment code in the code cache, so that any future request for the execution of the same fragment code will save on the compilation overheads, if the compiled fragment code is still available in the cache.

In principle, it is possible to cache both the output and the code of a cacheable fragment, but this appears to be an overkill since if the output is long-lived, then its associated code will be accessed only rarely. Therefore, it appears better to cache only the codes of *uncacheable* fragments, since these are the codes that will be frequently utilized.

It is important to note that the above solution is guaranteed to serve *fresh* content, since it is associated with the origin server. Moreover, it also ensures serving *correct* pages, since the page is specific to the user request. From a broad perspective, by integrated caching we are achieving the long-term benefits whenever the fragments or their associated compiled codes are reused in course of time.

5.2.1 Server Cache Management

In a pure fragment-caching approach, the server cache is used solely for hosting fragments. Similarly, in a pure code caching approach, the server cache can be used solely for hosting compiled codes. However, in our integrated caching approach, we need to allocate cache space for hosting both fragments and compiled codes. Therefore, we partition the cache into a *fragment cache* and a *code cache*.

Cache Partition Sizing

An immediate issue that arises here is determining the relative sizes of the fragment cache and the code cache partitions. This issue is investigated in detail in our experimental study presented in Section 5.6 – our results there indicate that the size of the code-cache must be in accordance with Equation 5.3, given in Section 5.6.3. Which signifies that the size of the code cache must be related to cacheability in such a way that if the cacheability is high, the cache space given to code cache must be less, otherwise the cache space given to the code cache must be high.

Cache Replacement Policies

With regard to the fragment cache, we are not aware of any web logs that are available to track the reference patterns for fragment access. This restricts us to the use of simple techniques like Least Recently Used (LRU) for managing the fragment cache. A similar technique can be applied to code-caching as well.

An association between the fragments in the fragment cache and the data in the origin server is maintained. Whenever a fragment is invalidated, it is marked invalid in the fragment cache, so that the further use of such a fragment is avoided.

In general, no such association is required for the compiled codes cached in the code cache since their source scripts rarely change, and further, these changes are usually made manually, in which case, the system administrator can forcibly flush the code-cache. But in an environment where script changes are frequent and automatically done, a similar association between scripts and their cached code fragments can be maintained. To the best of our knowledge, we are not aware of any web server where script changes are done automatically.

5.3 Augmentation with Page Pre-generation

As mentioned in the Introduction Chapter 1, caching helps to reduce page construction time after a request has been received, that is, *post-facto*. However, if it were possible to

anticipate a forthcoming request and have the page generated *apriori*, then the response would be instantaneous.

Here, we assume the use of a path-based prediction model, described in Section 2.4. Specifically, for an outgoing user response at the web-server, the web-server decides to generate the most expected next page for the user, based on many considerations such as current system load, the type of user, the benefit of pre-generating a page and so on. When the web server receives the next page request from the same user, it checks whether it has pre-generated the page the user is requesting now. If so, the page is served immediately. If not, the page request is treated as a normal page request and the page is constructed freshly and served.

5.3.1 Server Load Management

While page pre-generation is useful for reducing response times, it also involves expense of computational resources. This is acceptable under normal operating conditions, even if the page prediction accuracy is not good, since web-servers are typically over-provisioned in order to be able to handle peak load conditions [59], and we are only using this excess capacity. But, when the system is under peak load conditions, the wasted resources due to the mistakes made by the pre-generation process may actually exacerbate the situation, driving the system into *a worse condition*. To address this issue, a simple linear feedback mechanism that modulates the pre-generation process to suit the current loading condition is implemented in [65]. We use this technique in the work reported here also.

5.4 Analytical Results

The benefit that we expect in our integrated caching architecture is *reduced dynamic web page construction time*. In this section, we analyze this benefit. Table 5.1 contains the notations used in this section.

In our analysis, we wish to compare the the dynamic web page construction times

Table 5.1: **Notations for Analytical Results in Integrated Caching**

Symbol	Description
$\varepsilon = \{e_1, e_2, ..., e_m\}$	set of fragments
$\mathcal{C} = \{c_1, c_2, ..., c_n\}$	set of pages
$E_i = \{e_j : e_j \epsilon c_i\}$	set of fragments corresponding to page c_i
s_{e_j}	average size of fragment e_j (bytes)
W_Q	average waiting time in queue

for three cases: (a) with only fragment caching in place; (b) with only code caching in place; or, (c) integration of fragment caching with code caching.

We next describe our assumptions and derive a generic expression for the average time taken to generate a dynamic web page by a given web site infrastructure, when no optimization is in place. We then derive specific expressions for each of the three cases.

Recall from our discussion in Section 2.2 that a dynamic script generates pages. For the purposes of this analysis, we model a given web application as a set of such pages $\mathcal{C} = \{c_1, c_2, ..., c_n\}$. Each page is created by running a script (as described in Section 3.1.3), and the resulting page consists of a set of fragments, drawn from the set of all possible fragments, $\varepsilon = \{e_1, e_2, ..., e_m\}$. We let E_i, $E_i \subseteq \varepsilon$, be the set of fragments corresponding to page c_i.

As described in Section 4.3, we characterize our system as a M/M/k queue. For average waiting time (W_Q) of a page request in the web-server queue, we use the Equation 4.1.

Algorithms

To put the performance of our new **Integrated** approach in proper perspective, we compare it against two benchmark algorithms, **Pure_FC**, which implements pure fragment-caching on the entire cache, and **Pure_CC**, which implements pure code-caching on the entire cache, respectively. In the initial set of experiments, page pre-generation is not included, but is considered subsequently.

We define T^{Pure_FC} as the average time taken by the website when only fragment

caching is used, T^{Pure_CC} as the average time taken by the website when only code caching is used, and $T^{Integrated}$ as the average time taken by the website when both fragment caching and code caching used, i.e., in our proposed integrated caching architecture.

We proceed to derive expressions for the above three cases. In the case where only fragment caching is used, the expression is given by the Equation 4.3.

We now obtain the expression for the case where only code caching is used. In this case,

$$T^{Pure\_CC} = W_Q + (\Sigma t_{e_i})/\text{reduction-factor} \tag{5.1}$$

where reduction-factor is the execution reduction due to code caching used.

We now obtain the expression for our proposed integrated cache, where both fragment caching and code caching are used. In this case,

$$T^{Integrated} = W_Q + (n^{'} \times cst) + ((1 - n^{'}) \times \Sigma t_{e_i})/\text{reduction-factor} \tag{5.2}$$

Table 5.2: **Baseline Parameter settings for Analysis (Integrated Caching)**

Parameter	Value
average fragment size (s_e)	2Kbytes
average number of fragments per page	9
average page size (s_e)	20Kbytes
cacheability	50%
code reduction-factor	2
page predictability	50%
number of application servers	4
average fragment generation time	20ms

We have compared the expected time taken for three cases using the baseline parameter values shown in Table 5.2, the basis for parameter values is described in Section 4.4. It is very difficult to obtain exact time of a fragment computation, since it depends upon various factors, like the the time of computation, the load on the system, and so on. Hence we assume an average value [This is not a problem in our analysis, since we compare only the *relative* performance]. The analytical results are shown in Figure 5.2,

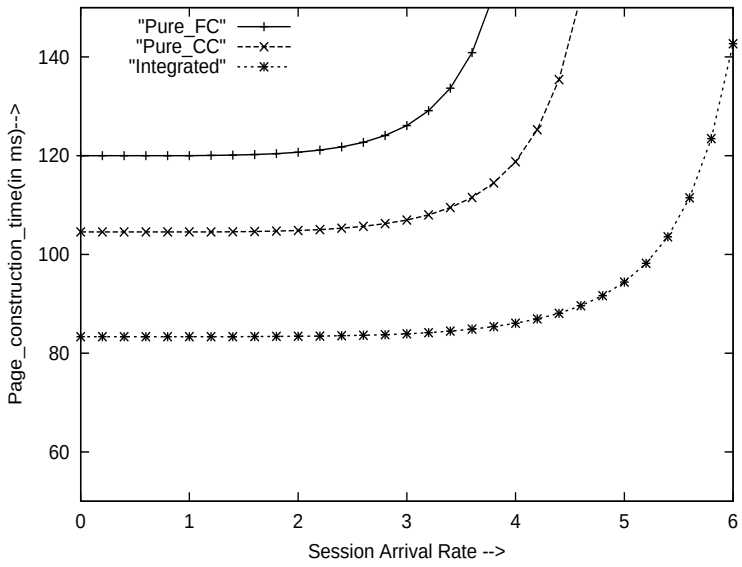


Figure 5.2: **Analytical results: Integrated Caching**

which gives the relative performance of the three dynamic webpage construction algorithms. We see here that the Integrated approach (labeled as Integrated in the figure) performs the best compared to all others and further, it performs 30% better than fragment caching (labeled as Pure_FC in the figure), the policy that has been advocated in the recent literature.

The foregoing results indicate that the integrated caching is indeed beneficial in terms of reducing the dynamic web page construction time, during both normal loading and peak loading.

5.5 Simulation Model

To evaluate the performance of the proposed integrated caching system, we have developed a detailed discrete-event simulator of a web-server supplying dynamic web-pages to users – this model is similar to that used in [65], and is extended to incorporate the code-caching feature. Table 5.3 gives the default values of the parameters used in our simulator – these values are chosen to be indicative of typical current websites, with

some degree of scaling to ensure manageable simulation run-times. Some of the parameter values setting is described in Section 4.4. Here also, we use as the same Web-site Model and User Model described in Section 4.4.1 and Section 4.4.3 respectively.

5.5.1 Web-page Model

Each dynamic web-page consists of a static part and a collection of identifiable dynamic fragments. A fraction $FragCacheable$ of these dynamic fragments are cacheable, while the remaining are not. The number of fragments in a page are uniformly distributed over the range *(MinFragNum,MaxFragNum)*. Two distributions of the choice of fragments are considered: *Uniform*, where the fragments are selected uniformly from the *FragPopulation* fragments, and *Skewed*, where a "90-10" rule applies in that 90 percent of the fragment choices are made from 10 percent of the *FragPopulation* fragments. Finally, the cost of producing a fragment is taken to be exponentially distributed with mean time for fragment generation given by *FragCost*.

The fragment source code sizes are between $MinimumSourceSize$ and $MaximumSourceSize$. When the fragment source code is compiled, the size of the compiled code is usually larger as compared to the equivalent source code size – this increase is modeled by the factor $CodeBlowupFactor$, and the value chosen is based on our analysis of a representative set of real-world scripts. The minimum execution speedup due to pre-compilation is given by $ReductionFactor$ – the value chosen is based on a conservative estimate of the speedups mentioned in [39]. Some of the parameter values chosen are conservative based on [22, 56].

Finally, the accuracy of page access prediction, used in the page pre-generation process, is determined by the $PagePredict$ parameter. The load-controlling feedback mechanism kicks in when the current load exceeds the setting of the $ThresholdLoad$ parameter.

5.5.2 Cache Model

The web-server has a cache for dynamic page construction, of size *CacheSize*. The fraction of the cache given to the Code Cache is given by *CodeCacheFraction*, with the remainder assigned to the fragment-cum-page cache. Within the fragment-cum-page cache, the space is equally divided between the fragments and pages, as per [65]. The search times in the code cache and fragment-cum-page cache are determined by the *CacheSearchTime* parameter. The fragments in the fragment cache are modeled to be invalidated randomly by the data source with an invalidation rate set by $InvalidRate$.

5.6 Experiments and Results

Using the above simulation model, we conducted a variety of experiments, the highlights of which are described here. The performance metric used in all our experiments is the average *dynamic page construction time*, evaluated for a range of fragment cacheabilities as a function of the session arrival rate and the fraction of the cache assigned to the code cache. The fragment cacheability and page prediction accuracy are evaluated for the following settings: LOW (20%), MEDIUM (50%) and HIGH (80%), covering the spectrum of real-life website environments. Also, the user arrival rates cover both normal loading conditions as well as peak load scenarios. The simulator was written in C++SIM [8], an object-oriented simulation testbed. The experiments were conducted on Ultra-Sparc/Solaris 2.6 workstations.

5.6.1 Experiment 1: Page Construction Times (Uniform)

In our first experiment, we evaluate the dynamic web-page construction times for an environment where the fragment cacheability is Medium (50 percent), the cache memory is *equally* partitioned between the code cache and the fragment cache (and no page pre-generation), and the fragment distribution is *Uniform*.

For this scenario, Figure 5.3 gives the relative performance of the various algorithms as a function of the session arrival rate. We see here first that the Integrated approach

Table 5.3: **Simulation Parameter settings (Integrated Caching)**

Parameter	**Setting**
FanOut	10
BackLinks	20 percent
FragPopulation	8000
CacheSize	2 MB
CodeCacheFraction	0 to 100 percent
FragCost	20 ms
MinPageSize	10 KB
MaxPageSize	30 KB
MinFragNum	1
MaxFragNum	19
MinFragSize	1 KB
MaxFragSize	3 KB
ArrivalRate	0 to 12 sessions per second
InvalidRate	1/ms
MinSessionPage	1
MaxSessionPage	19
MinThinkTime	1 second
MaxThinkTime	9 seconds
FragCacheable	20, 50, 80 percent
CacheSearchTime	0.1 ms
NumberofPagesUsed	2074
MinimumSourceSize	100 bytes
MaximumSourceSize	300 bytes
CodeBlowupFactor	10
ReductionFactor	2
ThreshholdLoad	75 percent
PagePredict	20, 50, 80 percent

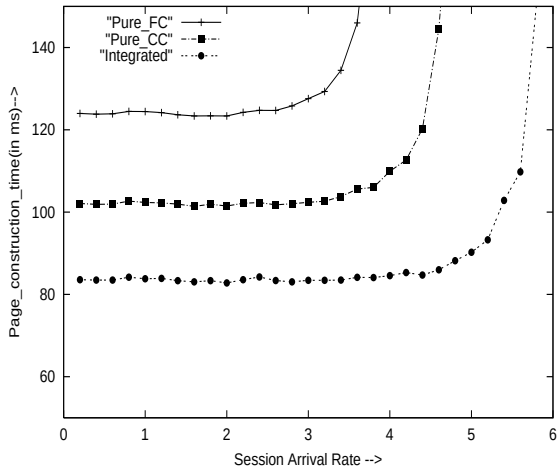


Figure 5.3: **Page Construction Times (Uniform)**

performs about 20% better than Pure_CC and 30% better than Pure_FC. Further, it can sustain *performance stability* for a higher arrival rate (upto 6 sessions per second) as compared to both Pure_CC and Pure_FC. Our experimental results follow our analytical results closely, derived in Section 5.4.

5.6.2 Experiment 2: Page Construction Times (Skewed)

When the above experiment is carried out with a *Skewed* fragment distribution, the resulting performance is as shown in Figure 5.4. We see here that the performance differences between Integrated and the baselines *substantially increase* – now, Integrated is 40 percent better than Pure_CC and 60 percent better compared to Pure_FC. In most real-world situations, fragment choices are indeed skewed, highlighting the importance of the Integrated approach to dynamic page construction.

Further, the Integrated algorithm is flatter for a longer time and sustains *performance stability* for much higher arrival rates (upto 12 sessions per second) as compared to both Pure_CC (6 sessions per second) and Pure_FC (4 sessions per second).

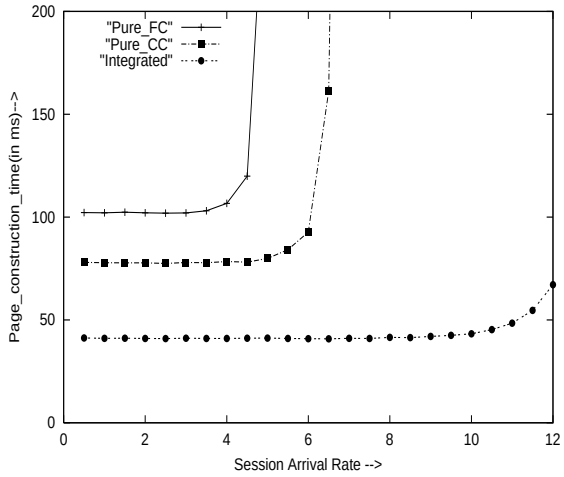


Figure 5.4: **Page Construction Times (Skewed)**

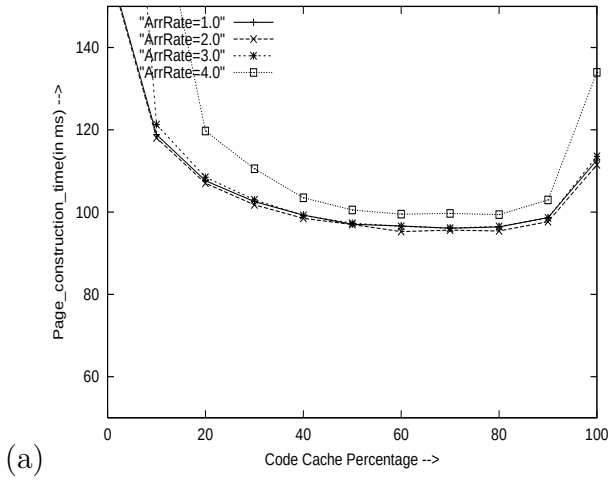
5.6.3 Experiment 3: Cache Partitioning (Uniform)

In our next experiment, we investigated the performance impact on the Integrated approach of various choices of cache partitioning sizes between the code cache and the fragment cache. This is done over the entire range of fragment-cacheability levels (Low, Medium and High), resulting in three different cases. The results for all these cases are shown in Figures 5.5(a-c) for arrival rates 1 through 4, as a function of the code-cache percentage size, and for a Uniform fragment distribution.

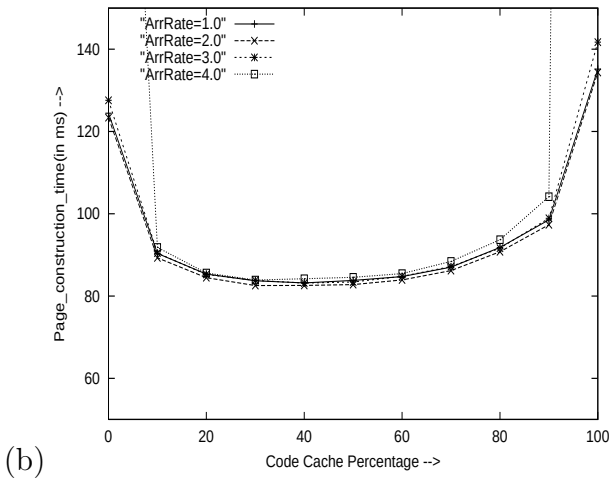
In these figures, we first observe that all the Integrated graphs have a "cup shape" with the highest construction times being at the extremes (0 percent code-cache and 100 percent code-cache), and the lowest somewhere in between. Further, we find that a simple heuristic relationship exists between the best partition, which leads to lowest page construction time, and the fragment-cacheability:

$$BestPartition = 100 - Fragment\_cacheability \tag{5.3}$$

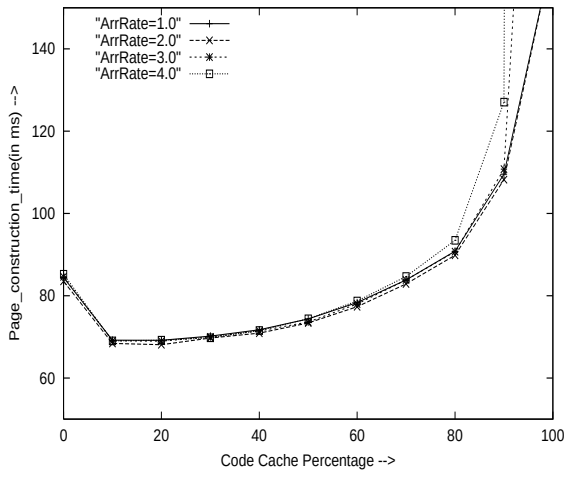
So, for example, with High (80%) fragment-cacheability, the Best Partition occurs at about 20% for the code-cache. In all our other experiments also, we found this heuristic



(a)

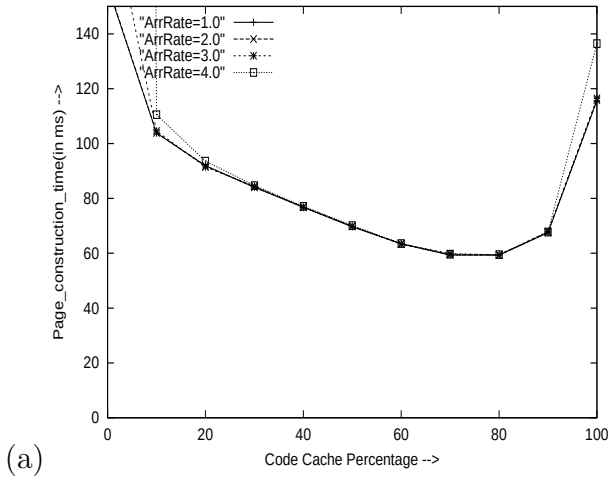


(b)

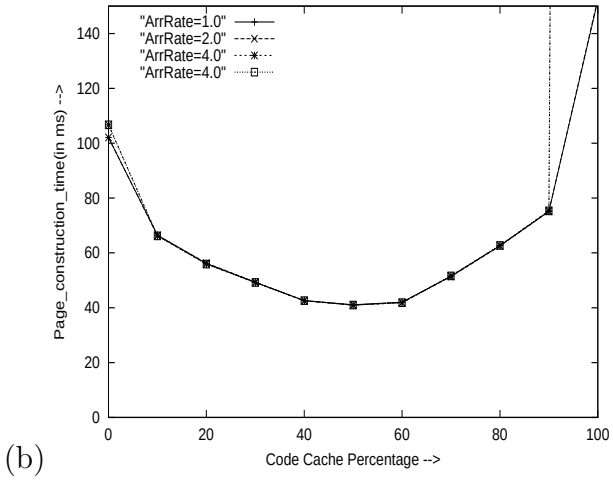


(c)

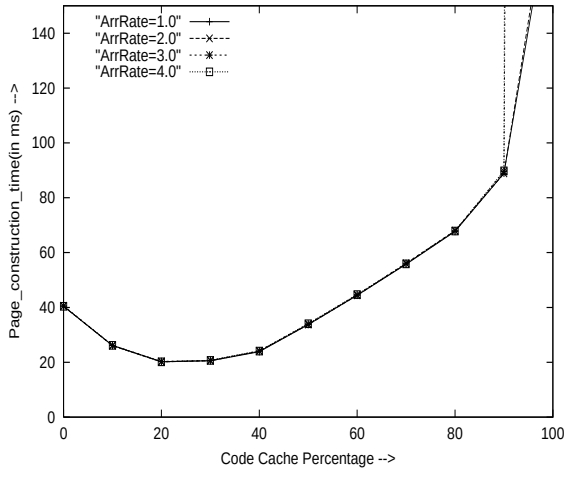
Figure 5.5: **Cache Partitioning (Uniform): (a) LOW fragment-cacheability (b) MEDIUM fragment-cacheability (c) HIGH fragment-cacheability**



(a)



(b)



(c)

Figure 5.6: **Cache Partitioning (Skewed): (a) LOW fragment-cacheability (b) MEDIUM fragment-cacheability (c) HIGH fragment-cacheability**

to approximately hold true.

An important point to note here is that the setting of 0 percent code-cache is equivalent to a *Pure_FC* approach, while 100 percent code-cache is equivalent to *Pure_CC*. We observe in the graphs that the performance of both Pure_FC and *Pure_CC* are very highly variable with regard to the fragment-cacheability level.

5.6.4 Experiment 4: Cache Partitioning (Skewed)

When the above experiment was carried out with a *Skewed* fragment distribution, the resulting performance is as shown in Figures 5.6(a-c). We see here that the cup shapes are much deeper as compared to the Uniform case, indicating that choosing the Best Partition appropriately becomes even more critical. But this is not difficult since the choice continues to be in accordance with Equation 5.3.

5.6.5 Experiment 5: Page Pre-generation (Uniform)

We now move on to investigating the impact of the optional page pre-generation on dynamic page construction times. In our first experiment here, the caching algorithms are augmented with page pre-generation for 50 percent page predictability, the rest of the parameters remaining the same as that of Experiment 1 (as mentioned earlier, the internal partitioning of the fragment-cum-page cache is always equally split between fragments and pages, as per [65]).

For this environment, the page construction times are shown in Figure 5.7 – for graph readability, we only show the performance of the basic Integrated algorithm and its PG (page pre-generation) variation. We first see here that PG_Integrated performs upto *30 percent better* than basic Integrated during normal loading (upto 2 user sessions per second), and no worse during peak loading (by virtue of the feedback-based load-control mechanism). In a nutshell, PG_Integrated provides both excellent average-case performance and stable worst-case performance.

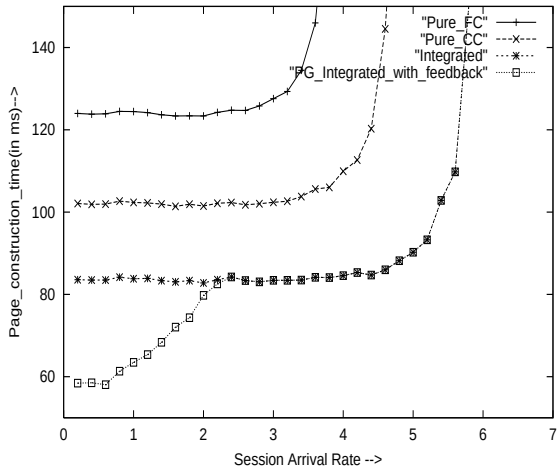


Figure 5.7: **Impact of Page Pre-generation (Uniform)**

5.6.6 Experiment 6: Page Pre-generation (Skewed)

When the above experiment was carried out with a *Skewed* fragment distribution, the resulting performance is as shown in Figure 5.8. We see here that the performance is improved only marginally (by about 10 percent at the low loads). This is because when the fragment distribution is skewed, then most of the frequent fragments are either served from the fragment cache or executed directly from the code cache and they are the ones which are frequently requested. So there is little work remaining, even when the page is constructed after receiving the request. Hence the impact of the page pre-generator is marginal in this case.

5.6.7 Experiment 7: Integrated Caching with Page Pre-generation : Cache Partitioning (Uniform)

In our next experiment, we investigated the performance impact on the Integrated approach of various choices of cache partitioning sizes between the code cache and the fragment cache augmented with anticipatory page pre-generation. This is done over the entire range of fragment-cacheability levels (Low, Medium and High), resulting in three different cases. The results for all these cases are shown in Figures 5.9(a-c) for arrival

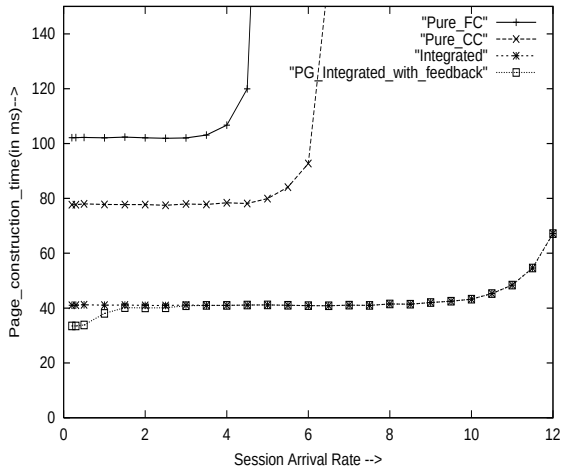


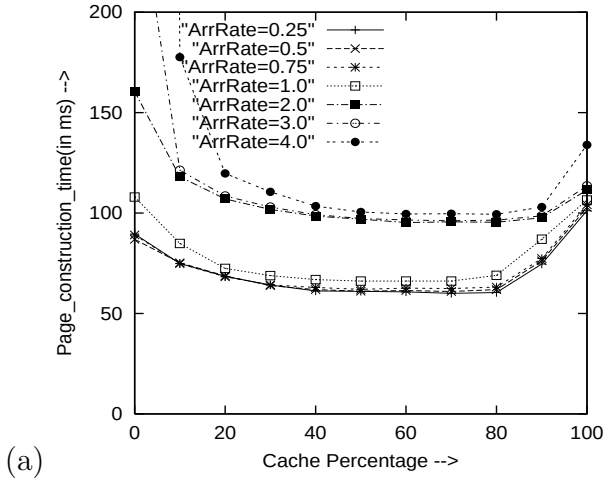
Figure 5.8: **Impact of Page Pre-generation (Skewed)**

rates 0.25, 0.5 0.75, 1 through 4, as a function of the code-cache percentage size, and for a Uniform fragment distribution. Here we consider very low arrival rate also, since at high arrival rate the page pre-generation becomes effectively off by the nature of the load-controlled feedback mechanism.

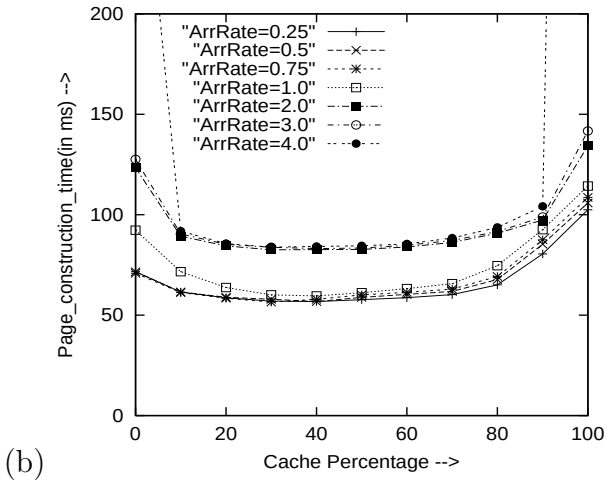
In these figures, we first observe that all the Integrated graphs have a "cup shape" with the highest construction times being at the extremes (0 percent code-cache and 100 percent code-cache), and the lowest somewhere in between. Further, we find that the simple heuristic relationship suggested in Equation 5.3 holds here also. Here, we also observe two bands of cups, the lower band corresponds to low arrival rates, in which Page pre-generation *on* and the upper band corresponds to high arrival rates, in which Page pre-generation is turned *off*.

5.6.8 Experiment 8: Integrated Caching with Page Pre-generation : Cache Partitioning (skewed)

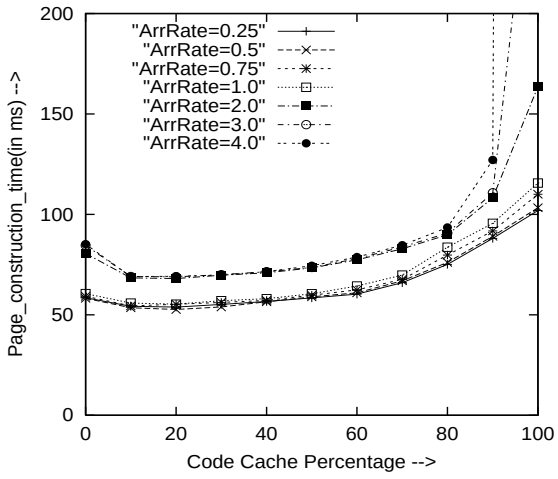
When the above experiment was carried out with a *Skewed* fragment distribution, the resulting performance is as shown in Figures 5.10(a-c). We see here also similar results and the choice continues to be in accordance with Equation 5.3. Here also, we observe



(a)



(b)



(c)

Figure 5.9: **Cache Partitioning (Uniform): (a) LOW fragment-cacheability (b) MEDIUM fragment-cacheability (c) HIGH fragment-cacheability**

two bands of cups as in the previous case.

5.6.9 Summary

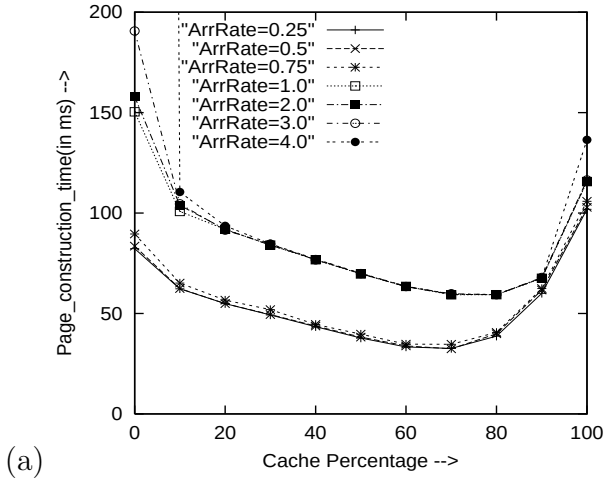
As described above, we have carried out a variety of experiments on proposed Integrated caching both with and without page pre-generation. To summarize our experimental results, we observe that the Integrated caching (without page pre-generation) performs 20 percent better than Pure_CC (the code caching) and 30 percent better than Pure_FC (the fragment caching), under Uniform fragment distribution. Under Skewed fragment distribution, it performs 40 percent better than Pure_CC and 60 percent better than Pure_FC.

The extended Integrated caching with page pre-generation (with feedback mechanism) performs better than the Integrated caching during normal loading and dose no worse during peak loading. We observe that during normal loading speedups of 10 to 30 percent can be realized, depending on fragment distribution.

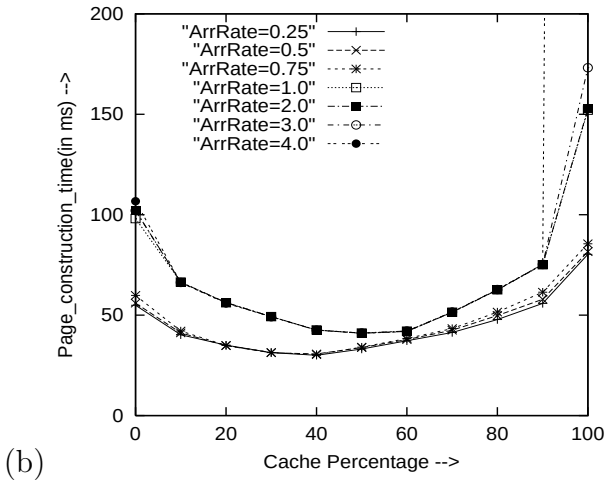
5.7 Conclusions

In this chapter, we have proposed a simple integrated caching approach to reduce dynamic web-page construction times by appropriately combining both fragment-caching and code-caching. We made a detailed evaluation of the integrated caching approach over a range of cacheability levels and fragment choice distributions. Our experimental results showed that the integrated approach can provide significant reductions in construction times, especially for skewed fragment distributions. We were also able to identify a simple heuristic for identifying the appropriate size of the code cache partition.

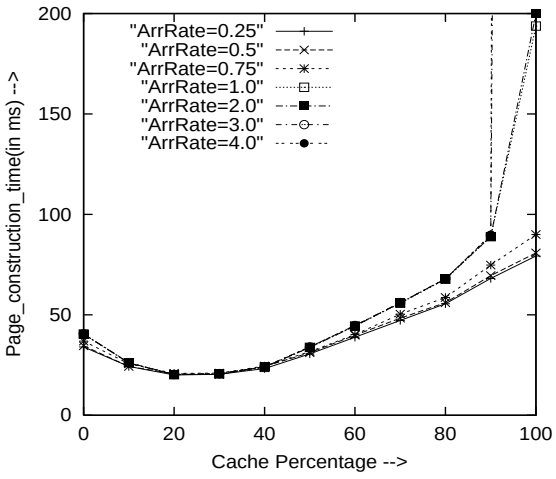
We extended our integrated caching model to incorporate anticipatory page pre-generation. The results of this integration with feedback mechanism show that the extended approach performs significantly better during normal loading and does no worse during peak loading.



(a)



(b)



(c)

Figure 5.10: **Cache Partitioning (Skewed): (a) LOW fragment-cacheability (b) MEDIUM fragment-cacheability (c) HIGH fragment-cacheability**

In summary, our approach achieves long-term benefits through fragment and code-caching and immediate benefits through page pre-generation. Currently, our work uses a LRU-based cache replacement policy. If web logs for fragment access become available, then it will be interesting to study the performance of customized cache-replacement policies.

Chapter 6

Conclusions and Future Research Avenues

6.1 Conclusions

In this thesis, we have proposed caching solutions They are broadly classified as proxy-side caching and server-side caching solutions. Under proxy-side caching, we have proposed Dynamic Proxy Caching. Under server-side caching, we have proposed Hybrid Caching, and Integrated Caching.

We have carried out a performance analysis of integrating various techniques to reduce bandwidth consumption and dynamic page construction times, by resorting to integration of fragment-caching with other techniques. A survey of existing techniques shows that many individual techniques are good in their own context, but we have shown in our results that an integration of these techniques performs significantly better when compared to the techniques in isolation.

First, we implemented a proxy-based dynamic content accelerator, which combines the best features of fragment-caching and proxy-caching, with the intention of reducing bandwidth requirement. With this, we could reduce the size of dynamic responses, leading to less bandwidth consumption and associated delays, like firewall and routing.

Next, we have proposed a hybrid caching architecture. Here, we simulated a server-side caching solution by integrating fragment-caching and anticipatory page pre-generation to reduce dynamic page construction times, during normal loading, by utilizing the excess capacity with which websites are typically provisioned to handle peak loads. We did an analysis under a fixed cache budget and shown through our simulation results that dynamic page construction times can be reduced by more than fifty percent. And further we have shown that the cache partition in the ratio 50:50 between fragment-cache and page-cache works reasonably well across different cacheability levels and page prediction accuracies.

Finally, we have proposed an integrated caching architecture. In this we simulated a server-side caching solution by integrating fragment-caching and code-caching, optionally augmented with page pre-generation. Through experiments, we have shown that proposed integrated caching performs significantly better than the individual components. We also arrived at a simple heuristic to partition the cache between the fragment-cache and the code-cache. We further extended the integrated caching architecture by optionally augmenting with anticipatory page pre-generation and showed that the results are significantly better during normal loading and no worse during peak loading.

In summary, we have presented in this thesis a performance analysis of integration of various solutions to reduce both network bandwidth consumption and page construction times of dynamic web pages, by resorting to integration of fragment-caching with various other solutions. In short, this thesis has addressed performance and scalability issues of dynamic websites.

6.2 Future Research Avenues

The work presented in this thesis can be extended in a number of ways, some of which are listed here:

1. **Predicting a set of pages:** Currently in our hybrid caching model, we restrict the page pre-generation to the single most likely page. In our future work, we plan

to investigate the performance effects of pre-generating a set of pages, rather than just a single page, in our hybrid caching model.

2. **Creating and studying web logs for fragments:** In order to effectively manage a web server, it is necessary to get feedback about the activity and performance of the server as well as any problems that may be occurring. The current web servers provide web logs at the page access level. The server access log records all requests processed by the server. But, to best of our knowledge, there are no web logs for fragment accesses. The creation of web logs for fragment access may be useful for various studies.

3. **Deploying in a real system:** Both fragment-caching and code-caching are available only in the form of proprietary software. Given public domain software for fragment-caching and code-caching, we can deploy our proposed hybrid caching and integrated caching and study their performance in real systems.

4. **Study of fragment distribution:** Since there is a lack of web logs for fragment accesses, we were restricted to cache replacement polices like LRU. If fragment access logs become available, their study may reveal many interesting phenomena and suggest better cache replacement policies, etc.

5. **Handling dynamic proxy caching failures:** In the proposed dynamic proxy caching, it is assumed that the DPC never fails. But, in practice, it may not be the case. If there is a failure in the DPC, the back-end monitor (BEM) may not be aware of this event. So, the site may become unavailable due to DPC failure. It is interesting to investigate the solutions for DPC failures.

6. **Security of dynamic contents at DPC:** In the proposed DPC, the dynamic contents should be protected. Methods to deal with the protection of dynamic contents cached at DPC have to be evolved.

Bibliography

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

[2] R. Alfano, G. Cassone and D. Gotta. Web Log Analysis for Performance Troubleshooting. TELECOM LAB ITALIA,http://www.telecomitalialab.com.

[3] V. Almeida, A. Bestavros, M. Crovella and A. de Oliveira. Characterizing reference locality in the WWW. In Proceedings of 4th International Conference on Parallel and Distributed Information Systems (PDIS), December 1996.

[4] M. Altinel, C. Bornhoevd, S. Krishnamurthy, C. Mohan, H. Pirahesh and B. Reinwald. Cache Tables: Paving the Way for an Adaptive Database Cache. In Proceedings of 29th International Conference on Very Large Data Bases (VLDB), Berlin, September 2003.

[5] M. Andersson, J. Cao, M. Kihl and C. Nyberg. Performance Modeling of an Apache Web Server with Bursty Arrival Traffic. In Proceedings of 4th International Conference on Internet Computing, June 2003.

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

[7] A. Bestavros, R. Carter, M. Crovella, C. Cunha, A. Heddaya and S. Mirdad. Application Level Document Caching in the Internet. In Proceedings of 2nd International Workshop on Services in Distributed and Networked Environments (SDNE), June 1995.

[8] C++SIM User's Guide, Public Release 1.5. Department of Computing Science, Computing Laboratory, University of Newcastle upon Tyne, 1994.

[9] 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 ACM SIGMOD International Conference on Management of Data (SIGMOD), May 2001.

[10] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In Proceedings of USENIX Symposium on Internet Technologies and Systems, December 1997.

[11] J. Challenger, A. Iyengar and P. Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. In Proceedings of 18th Annual Joint Conference of the IEEE Computer and Communications Societies, 1999.

[12] J. Challenger, A. Iyengar, K. Witting, C. Ferstat and P. Reed. A Publishing System for Efficiently Creating Dynamic Web Data. In Proceedings of IEEE INFOCOM 2000, March 2000.

[13] S. Choi, S. Huh, S. M. Kim, J. Song and Y. Lee. A Scalable Update Management Mechanism for Query Result Caching Systems at Database-driven Web sites. In Proceedings of 8th Asia Pacific Web Conference (APWEB), January 2006.

[14] W. Chan and Y. Lin. The Waiting Time Distribution for the M/M/m Queue . In Proceedings of IEE Communications, June 2003.

[15] E. Cohen and H. Kaplan. Refreshment Policies for Web Content Caches. In Proceedings of 20th Annual Joint Conference of IEEE Computer and Communications Societies (INFOCOM), April 2001.

[16] E. Cohen and H. Kaplan. Exploiting regularities in Web traffic patterns for cache replacement. In Proceedings of 31st Annual ACM Symposium on Theory of Computing (STOC), May 1999.

[17] E. Cohen, B. Krishnamurthy and J. Rexford. Evaluating server-assisted cache replacement in the Web. In Proceedings of 6th Annual European Symposium on Algorithms, Springer-Verlag, Lecture Notes in Computer Science Volume 1461, August 1998.

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

[19] A. Datta, K. Dutta, D. Fishman, K. Ramamritham, H. Thomas and D. VanderMeer. A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration. In Proceedings of 27th International Conference on Very Large Data Bases (VLDB), September 2001.

[20] A. Datta, K. Dutta, K. Ramamritham, H. M. Thomas and D. E. VanderMeer. Dynamic Content Acceleration: A Caching Solution to Enable Scalable Dynamic Web Page Generation. In Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD), May 2001.

[21] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, Suresha and K. Ramamritham. Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation. In Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD), June 2002.

[22] A. Datta, K. Dutta, H. Thomas, D. VanderMeer and K. Ramamritham. Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation. ACM Transactions on Database Systems, Vol. 29, No. 2, pp. 403-443, June 2004.

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

[24] D. Duchamp. Prefetching Hyperlinks. In Proceedings of 2nd USENIX Symposium on Internet Technologies and Systems, October 1999.

[25] A. Eden, B. Joh and T. Mudge. Web Latency Reduction via Client-Side Prefetching. In Proceedings of IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), April 2000.

[26] Ejacent White paper. Resilient Performance and Instant Scalability for Interactive Web Site Services: A New Approach to Internet Infrastructure for Dynamic Content and Services. http://www.ejacent.com.

[27] ESI Consortium. Edge side includes. http://www.esi.org.

[28] S. Gadde, M. Rabinovich and J. Chase. Reduce, Reuse, Recycle: An Approach to Building Large Internet Caches. In Proceedings of Workshop on Hot Topics in Operating Systems, May 1997.

[29] E. Gamma, R. Helm, R. Johnson and J. Vliss. Design Pattern: Elements of Reusable Object-Oriented Software. Addison Wesley, October 1994.

[30] C. Huitema. Network vs. server issues in end-to-end performance. Keynote address, Performance and Architecture of Web Servers Workshop (PAWS), June 2000.

[31] IBM. Corporation. IBM Websphere edge server version 2.0 (product documentation), 2001.

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

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

[34] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In Proceedings of Usenix Symposium on Internet Technologies and Systems, December 1997.

[35] Z. Jiang and L. Kleinrock. Prefetching Links on the WWW. In Proceedings of IEEE International Conference on Communications, June 1997.

[36] D. E. Knuth, J. H. Morris and V. H. Pratt. Fast pattern matching in strings. In SIAM Journal of Computing, 6(2):323-350, June 1977.

[37] A. Labrinidis and N. Roussopoulos. Balancing Performance and Data Freshness in Web Database Servers. In Proceedings of 29th International Conference on Very Large Data Bases (VLDB), September 2003.

[38] R. Levering and M. Cutler. The portrait of a common HTML web page. In Proceedings of the ACM symposium on Document engineering, October 2006.

[39] N. Lindridge. The PHP Accelerator 1.2. http://www.php-accelerator.co.uk/PHPA_Article.pdf., April 2002.

[40] C. Loosely, R. Gimarc and A. Spellman. E-commerce response time: A reference model, a keynote systems white paper: http://www.keynote.com/services/html/product_lib, 2000.

[41] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. Lindsay and J. Naughton. Middle-tier database caching for e-business. In Proceedings of ACM SIGMOD International Conference on Management of Data(SIGMOD), June 2002.

[42] Q. Luo and J. Naughton. Form-based proxy caching for database-backed web sites. In Proceedings of 27th International Conference on Very Large Data Bases (VLDB), September 2001.

[43] Q. Luo, J. Naughton, R. Krishnamurthy, P. Cao and Y.Li. Active query caching for database web servers. In Proceedings of 3rd International Workshop on the Web and Databases (WebDB), May 2000.

[44] Microsoft. Asp. http://www.microsoft.com.

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

[46] C. Mohan. Tutorial: Caching technologies for web applications. In Proceedings of 27th International Conference on Very Large Data Bases (VLDB), September 2001.

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

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

[49] http://www.olympics.com

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

[51] V. Panteleenko and W. Freeh. Instantaneous Offloading of Transient Web Server Load. In Proceedings of 6th International Web Caching Workshop and Content Delivery Workshop, June 2001.

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

[53] PHP. http://www.php.net.

[54] M. Rabinovich and A. Aggarwal. Radar: A scalable architecture for a global web hosting service. In Proceedings of 8th International World Wide Web Conference (WWW), May 1999.

[55] Radview. http://www.radview.com.

[56] L. Ramaswamy, A. Iyengar, L. Liu and F. Douglis. Automatic Detection of Fragments in Dynamically Generated Web Pages. In Proceedings of 13th International World Wide Web Conference (WWW), May 2004.

[57] S. Ross. Introduction to Probability Models. Seventh Edition, Academic Press, India 2001.

[58] http://rubis.objectweb.org/

[59] S. Schechter, M. Krishnan and M. Smith. Using Path Profiles to Predict HTTP Requests. In Proceedings of 7th International World Wide Web Conference, April 1998.

[60] http://www.cs.rpi.edu/ puninj/RPINFO/MAY00/APR25/USER/rpi_log_report_index.html

[61] Sniffer Technologies. Sniffer basic. http://www.networkgeneral.com/

[62] SpiderCache. http://www.warpsolutions.com/Products/ProductsSpiderCache.php

[63] Z. Su, Q. Yang, Y. Lu and H. Zhang. WhatNext: A Prediction System for Web Requests using N-gram Sequence Models. In Proceedings of 1st International Conference on Web Information System and Engineering, June 2000.

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

[65] Suresha and J. Haritsa. On Reducing Dynamic Web Page Construction Times. In Proceedings of 6th Asia Pacific Web Conference (APWEB), April 2004.

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

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

[68] N. Viswanadham and Y. Narahari. Performance Modeling of Automated Manufacturing Systems. Prentice-Hall of India, 1994.

[69] J. Wang. A Survey of Web Caching Schemes for Internet. In ACM Communications Review, October 1999.

[70] Z. Wang and J. Crowcroft. Prefetching in World Wide Web. In Proceedings of IEEE Global Telecommunications Internet Mini-Conference, November 1996.

[71] http://en.wikipedia.org/wiki/Exponential_growth.

[72] S. Williams, M. Abrams, C. R. Standbridge, G. Abdulla and E. A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In Proceedings of ACM SIGCOMM Conference, August 1996.

[73] K. Yagoub, D. Florescu, V. Issarny and P.Valduriez. Caching Strategies for Data-Intensive Web Sites. In Proceedings of 26th International Conference on Very Large Data Bases (VLDB), September 2000.

[74] Q. Yang, H. Henry Zhang and Ian T. Li. Mining Web logs for prediction models in WWW caching and prefetching. In Proceedings of 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), August 2001.

[75] H. Zhu and T. Yang. Cachuma: Class-based Cache Management for Dynamic Web Content. Technical Report TRCS00-13, Dept. of Computer Science, The University of California at Santa Barbara, June 2000.

[76] Zona Research. Quoted in Interactive Week Vol.6, No. 36, September 1996.

[77] I. Zukerman, D. Albercht and A. Nicholson. Predicting Users' Requests on WWW. In Proceedings of 7th International Conference on User Modeling, June 1999.