Towards Efficient Delivery of Dynamic Web Content

A Thesis
Presented to
The Academic Faculty

by

Lakshmish Macheeri Ramaswamy

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
December 2005

Towards Efficient Delivery of Dynamic Web Content

Approved by:

Dr. Ling Liu, Advisor
College of Computing
Georgia Institute of Technology

Dr. Calton Pu
College of Computing
Georgia Institute of Technology

Dr. Arun Iyengar
IBM T.J. Watson Research Center

Dr. Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Dr. H. Venkateswaran
College of Computing
Georgia Institute of Technology

Date approved: August 24, 2005

Dedicated to my parents, my wife and my sister for their constant love and support

ACKNOWLEDGEMENTS

It is a matter of great joy, besides being my duty, to thank various people without whose help and support the research presented in this thesis would not have been possible. First and foremost, my advisor Prof. Ling Liu has been a constant source of inspiration, encouragement and support over the past five years. In addition to the most valuable guidance and help that I have received from her, she also gave me enough freedom to explore various research ideas. She was always there when I needed her. I cannot thank her enough for all the advice, support, and encouragement that I received from her.

My Ph.D. thesis committee members, Dr. Arun Iyengar, Prof. Calton Pu, Prof. Mustaque Ahamad and Prof. H. Venkateswaran gave very useful and insightful comments, which have enhanced the quality of this dissertation. I want to express my heartfelt gratitude to all of them.

The distributed data-intensive systems lab (DiSL) has been a great place to do research. Apart from the many stimulating academic discussions, I also had countless enjoyable experiences whose sweet memories I will cherish for the rest of my life. I thank the former and the current DiSL group members for making my stay at Georgia Tech a memorable one. Many thanks to the CERCS staff members Jennifer Chisholm, Deborah Mitchell, and Susie McClain for their help with various administrative issues. I thank my many friends at Georgia Tech whose company has added color and joy to my life as a Ph.D. student at Georgia Tech. I thank Prabir Mehta for being a great roommate during the first three and half years of my stay in Atlanta.

My parents have sacrificed a lot in every conceivable way to see me succeed in life. I owe every bit of what I am today to their selfless sacrifices over the years. No words can adequately express the depths of my gratitude towards them. My wife Seema has been extremely supportive in this endeavor apart from being a great life-partner. I thank her with all my heart for her constant love and support. Last, but certainly not the least,

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In recent years the dynamic content on the World Wide Web has grown at a rapid pace, introducing new challenges to the scalability and performance of web-based services. Traditional web caching technologies that have been successful for efficient delivery of static web pages, have not been very effective for delivering dynamic web content due to their frequent changing nature and their diversified freshness requirements. Hence, there is a growing demand for scalable and efficient mechanisms for delivering dynamic web content to the end users.

This dissertation is devoted to exploring novel architectures, techniques, and mechanisms for enhancing the scalability and the performance of dynamic web content delivery. In this thesis we study the challenges to efficient dynamic content delivery, and propose various schemes and techniques to address the challenges. This dissertation makes two main technical contributions.

As the first major technical contribution we design cooperative edge cache grid (cooperative EC grid, for short) - a large-scale edge cache network that supports low-cost cooperation among its caches. The cooperative EC grid is, to our knowledge, the first cooperative edge cache network for efficiently delivering highly dynamic web content with varying server update frequencies, whose design focuses on the reliability and scalability of dynamic content delivery in addition to cache hit rates.

Motivated by the philosophy of placing data and possibly parts of application closer to the clients, caching on the edge of the Internet has emerged as a popular mechanism for delivering dynamic content. However, many of the present-day edge cache networks do not provide adequate support for cooperation among the edge caches. The few edge cache networks that incorporate some kind of cooperation among their caches do not support server-driven stronger document consistency mechanisms. Hence these edge networks cannot effectively cache highly dynamic documents that are updated frequently. It is our

contention that cooperation among the individual edge caches coupled with scalable server-driven document consistency mechanisms can significantly enhance the benefits provided by the edge cache network.

In this thesis we study the major challenges involved in designing a cooperative edge cache network. Our design of the cooperative EC grid includes several unique techniques and algorithms that have been specifically designed to address these challenges.

1. We introduce the concept of cache clouds as the fundamental framework of cooperation in the Cooperative EC grid. A cache cloud is a group of edge caches located in close vicinity within the Internet, which cooperate among one another for maintaining document freshness and for serving client requests.

2. The cooperative EC grid scheme incorporates a novel technique for cache cloud formation called the selective landmarks-based distance sensitive clustering mechanism. This technique utilizes the concept of Internet landmarks for creating cache clouds such that the cooperation in the EC grid is both efficient and effective.

3. The architectural design of cache clouds includes distributed and failure resilient mechanism called the dynamic hashing scheme for object lookups and updates. This technique balances the lookup and update loads among the caches belonging to a cloud, and can gracefully tolerate the failures of individual caches.

4. A utility-based scheme has been designed for placing documents within the cache clouds. This scheme ensures optimal utilization of the available resources by estimating the costs and benefits of storing a document in a particular edge cache, and storing the document at that cache only if the benefit to cost ratio is favorable.

The second main contribution of this dissertation is an automatic scheme for detecting cache-effective fragments in dynamic web pages. This is the first automatic fragment detection scheme reported in the literature. The objective of our scheme is to detect and flag fragments that maximize the benefits of fragment-based caching and delivery of dynamic web content.

Constructing dynamic web pages from fragments has been shown to provide significant benefits to the delivery of dynamic web content. It has also been successfully commercialized in recent years. However, for a web site to harness the complete benefits of fragment-based schemes, good methods are needed for fragmenting the web pages. Most fragment-based dynamic content delivery techniques rely upon manual fragmentation of web pages, which is expensive, error prone and does not scale well.

The automatic fragment detection scheme presented in this thesis analyses the dynamic web pages with respect to their information sharing behavior, personalization characteristics and change patterns, and detects fragments that are cost-effective cache units. Our automatic fragment detection scheme has several novel features:

- The automatic fragment detection framework includes a hierarchical and fragment-aware model for dynamic web pages and a compact and effective data structure, called the augmented fragment tree (AF tree) for fragment detection.

- Second, we present an efficient algorithm to detect maximal fragments that are shared among multiple documents.

- Third, we develop a practical algorithm that effectively detects fragments based on their lifetime and personalization characteristics.

The fragments detected through our scheme can be utilized in any fragment-based scheme for dynamic content delivery.

This dissertation reports the extensive experiments that we have conducted to study the benefits and costs of the proposed techniques, and their effects on the dynamic content delivery process. The results of the experiments show that the proposed schemes can significantly enhance the scalability and performance of the dynamic content delivery process.

While the specific focus of this thesis is the efficient delivery of dynamic web content, many of the techniques and mechanisms that we have explored in this thesis have wider applicability, and they can also be applied to improve scalability and performance in various types of distributed systems.

CHAPTER I

INTRODUCTION

The Internet and the World Wide Web continue to grow at a tremendous pace, both in terms of the user populations, and the amounts and the varieties of the contents and services being offered to their users. Several periodic surveys have clearly demonstrated the phenomenal growth of the Internet and its diffusion into modern-day societies [8, 9, 10, 11]. The survey from Internet World Statistics [9] estimates that the number of Internet users increased from 361 million in the year 2000 to more than 682 million in 2003, which amounts to a growth of 89% over a period of three years.

The content and the services available on the Web have also experienced a tremendous growth. A survey from the Zooknic Internet geography project [15] shows that the total number of top level domains grew more than 10 times from 3.2 million in 1998 to 37.5 million in 2001.

In addition to the growths in user population and content/service providers, the Web has also grown in terms of the variety of the contents and services being offered through it. In contrast to the early days, when the contents available on the Web were limited to simple texts and images, today's Web is a source of multimedia-rich information. Further, during the initial years, most Web applications were information-dissemination in nature. Today, the World Wide Web forms the foundation on which advanced applications such as E-commerce and Online games are hosted.

An important technological development that has tremendously contributed to the growth and popularity of the Web was the advent of *Dynamic Web Content*. It has transformed the Web from being a passive information dissemination medium to a platform for applications that can not only react to user actions, but also can adapt to the profile of the user utilizing their services.

1.1 Dynamic Web Content

The web pages that are generated on the fly when a user request arrives at the web server infrastructure are known as *dynamic web pages*. In contrast to static HTML documents (static web pages), the dynamic web pages are not pre-composed. When the request for a dynamic web page arrives at the service provider infrastructure an appropriate application program is invoked which generates the document to be sent to the client. The application programs are also referred to as *scripts*. These application-programs may access various resources at the origin site such as back-end databases, html segments and text files. The framework within which the application programs are executed is known as the *application server*. Several application servers are available in the market including Oracle's 10g-application server [12], IBM's Websphere [6], and BEA's WebLogic [2].

Ever since its advent, the dynamic content on the web is experiencing a tremendous growth in popularity. Today, many web content providers publish their content through dynamic web pages. The popularity of dynamic web pages can be attributed to its three distinguishing features. First, the programs generating the dynamic web pages can be designed to take-in user specified query parameters, and to generate web pages based on these input parameters. Second, the generated web pages may incorporate results of queries to back-end databases, or other data sources. Therefore, the application programs can generate web pages that reflect the current state of back-end information sources. Third, the application programs can also incorporate organization-specific business logic into the web page composition process. These three features of the dynamic web pages empower a web site to provide personalized and up-to-date information to its clients. As an example, let us consider the web page from a weather service website such as `http://www.weather.com`. Here, the user specifies the Zip code and the date for which he needs the weather information. Depending upon the parameters, the application program retrieves appropriate weather information from one or more databases and composes the web page. Further, the application programs might place appropriate advertisements depending on the business policies of the website. It would have been virtually impossible to host a service of this nature using only pre-composed static web pages.

While dynamic web content has enabled various services on the web, it has also introduced new challenges to the efficiency and the scalability of Web-based services. Further, the very features of the dynamic web pages that have made them popular are also at the root of the challenges involved in efficiently delivering them to the clients.

As mentioned earlier, the dynamic web pages are generated on the fly when the user request arrives at the origin server, in contrast to the static web pages, which are precomposed and stored at the origin server. The process of generating a dynamic web page usually involves multiple costs. First, when the client-request arrives at the origin web site, the application server has to invoke the corresponding application program, which has an associated cost commonly referred to as application invocation cost. Second, the application server might need to query one or more back-end databases, which would result in additional costs known as the database access costs. Further, there are costs associated with operations such as parsing URL to retrieve the query parameters, communication across different tiers of the website infrastructure, web page composition, and object creation and destruction. Notice that in contrast to the web page generation process, serving static content is simple and significantly cheaper.

Due to the multiple costs, the process of dynamic web page generation places significant loads on the system resources such as CPU, memory, hard disks, and I/O bus, which results in two specific problems. First, the throughput of the web server - application server infrastructure, in terms of the number of client requests processed per unit time, is adversely impacted. Second, the average time required to process client requests increases. Mendes and Almeida [70] study the effect of dynamic content on these two key parameters. Their results show that the latencies of requests to dynamic web pages might be 2.5 to 3 times the latencies of requests to static web pages of similar sizes, and a web server handling dynamic content experiences 50% - 78% reduction in the number of maximum connections they can handle per unit time, in comparison with their counterparts which serve pure static content.

Researchers have explored various approaches to address the performance and scalability challenges caused by the dynamic web content. These approaches can be broadly classified into two categories: (1) efficient dynamic content generation; and (2) efficient caching

of dynamic content. The first category include approaches such as improving inter-tier communication process, distributed and parallel application server architectures, smarter request-scheduling techniques, and admission control policies. The work presented in this thesis focuses on the second approach, which is to cache the dynamic content in order to ensure its efficient and timely delivery.

1.2   Caching-based Strategies for Dynamic Content Delivery

Encouraged by the success of proxy caching in improving the performance of static content delivery, researchers began exploring caching as a technique for efficient delivery of dynamic web content. The fundamental idea of caching is to re-use the generated content. This mitigates the need for re-generation of the same content, thereby reducing the load on the origin site infrastructure. Further, depending upon where the caches are located, it can also reduce load on the network. Hence, caching dynamic content can improve both the scalability and the efficiency of dynamic content delivery.

However, it was soon realized that the traditional proxy caching techniques are not very effective for delivering dynamic web pages. Their dynamic nature introduces new challenges that have to be addressed satisfactorily in order to make caching an effective tool for dynamic content delivery.

1.2.1   Challenges for Caching Dynamic Web Content

Dynamic web content exhibits many unique characteristics, which introduce additional complexities and challenges to designing effective cache strategies. In this section we discuss some of the important challenges for designing dynamic content caches.

Personalized Content in Dynamic Pages

Many application programs are designed to generate web pages that are personalized to the clients issuing the requests. For example, the web pages generated by a news web site may contain a welcome bar indicating the login-name of the person accessing the web site. These web pages contain varying amounts of personalized information in them. This raises

the issues of privacy and security of the data when stored at caches, thereby prompting the web page designers to mark the web pages as uncacheable. Further, even when the web pages are not marked as uncacheable, their personalized nature might forbid the caches from re-using them for serving future requests from other users. This has not only caused a drastic reduction in the web content that can be stored at caches, but has also affected the utility of cached content for serving future requests.

Reusability of Cached Dynamic Content

Generally, the contents of dynamic web pages are dependent on one or more user-specified query parameters. Therefore, a single base URL could lead to many different web pages based on the parameters. Thus a caching scheme storing the web pages indexed by its URL and parameter values is not likely to have high content reusability. However, studies have shown that the web pages that are generated by the same application program, but with different parameter values are likely to share significant amount of content. Also the web pages from the same web site often have a common template and hence are likely to share content as well [43, 42]. Therefore, in order to improve the reusability of dynamic content, smarter caching schemes that can exploit the information-sharing characteristics of dynamic web pages need to be designed.

Document Freshness and Consistency

One of the most challenging problems for caching dynamic content is to ensure the freshness and consistency of the data stored at various caches. This is a very important problem involving several key issues. Dynamic web pages often contain results of queries to back-end databases, or other information sources. Therefore, the freshness of a dynamic web page is dependent on the state of the databases whose query-results were used to generate the page. For example, if a tuple T in a database D is updated, then all the web pages which were generated by incorporating the results of a query containing the tuple T would no longer be fresh. The backend databases of an origin website may undergo frequent changes. Further, it is very hard to accurately predict when a database would be updated. Hence,

not only are the dynamic web pages likely to undergo frequent changes, but it is also very difficult to estimate how long they are likely to remain fresh.

The frequent and unpredictable change patterns of dynamic web pages have many important implications. First, the caches that store dynamic web content have to support stronger consistency schemes than the traditional time-to-live mechanism. Most client-side caches still rely upon the time-to-live mechanism for maintaining freshness of cached documents, which prevents them from effectively caching dynamic web content. Thus the utility of client-side caches for efficiently delivering dynamic content is further reduced.

Second, the consistency requirements of the dynamic web pages can vary widely based on the semantics of the application program generating them. For example, some applications might require that any change to a dynamic web page be immediately reflected at all caches, whereas other applications might tolerate some relaxation in consistency, but may need guarantees with respect to worst-case staleness.

Third, stronger consistency mechanisms require active support from the origin site, and hence, they are commonly referred to as server-driven consistency mechanisms. Further, these mechanisms also require close interaction between the caches and the origin server. Implementing stronger consistency mechanisms entails significant costs in terms of the loads imposed on the origin servers, the caches, and the network.

- When a database table is modified, the origin server has to compute the web pages that are affected by the modification. Dynamic web pages might include results of complex queries that involve multiple database tables. In these scenarios, accurately computing the set of web pages that are affected by a database update is computationally hard. Further, in order to compute the set of web pages affected by an update, the origin server has to maintain the mapping between query instances and the generated web pages, which imposes additional load on the origin server.

- After computing the set of web pages that are affected by a database update, the origin server has to send update/invalidate messages to the caches that are currently storing copies of those web pages. There are two possible solutions to the update/invalidate

message propagation problem. The server might send update/invalidate messages to all the caches in the system irrespective of whether they are currently storing a copy of the particular document, or the server might send update/invalidate messages to only those caches that contain are currently storing a copy of the document. While the first solution involves unnecessary message communication and processing overheads, the second solution requires the origin server to maintain the information about the documents stored in various caches. When the documents are modified frequently, both solutions impose heavy loads on the server, network and the caches. If documents are updated at the rate of $UpdRate$ per unit time and each document is stored at $CCount$ caches on average, then the origin server has to send out $UpdRate \times CCount$ messages per unit time. Hence, we note that server-driven consistency mechanisms do not scale well either with respect to the number of caches or with respect to the update rates of documents.

Due to the above costs, the server-driven consistency mechanisms do not scale well both with respect to the number of caches in the system as well as the rate at which the dynamic content is modified.

Failure Resilience and Recovery

In addition to the loads placed on the servers, caches and the network, stronger consistency requirements of dynamic web content impacts other aspects of caching. An example would be the failure resilience and crash recovery mechanisms for caches. In static document caching scenario, wherein time-to-live mechanism suffices for maintaining document consistency, the failure resilience and recovery process are simple. If a cache fails, all the client-requests are redirected to other caches, or to the server. When the cache recovers, it checks whether any the TTL of any documents have expired. If so, those documents are refreshed either immediately, or when requested by clients. Similarly if the server becomes un-reachable, the cache checks whether the document requested by a client has expired TTL. If so the cache sends appropriate error messages to the clients, else the cache supplied the requested documents.

In dynamic content caches supporting server-driven consistency schemes, the problem becomes complicated. First, a reliable communication mechanism is needed to ensure that either the update/invalidate messages reach all the intended caches, or the server knows which caches did not receive the messages, so that it can attempt to re-send the message. Second, suppose a cache recovers from a crash, it does not know whether any of the documents were updated in the time duration when it had crashed. Hence, all the documents in the cache have to be validated with the server, which imposes heavy loads on the server as well as the cache. Similarly, if the server becomes unreachable due to network partitions and the cache receives a client-request for a document that is locally available, there is not straightforward way for the cache to determine whether the locally available copy is still fresh, or whether it has been updated at the server.

Additional Challenges

In addition to the above challenges, which are specific to dynamic web pages, dynamic content cache designers also encounter problems that are general to both static and dynamic web content. Some such problems are:

1. **Designing document placement and replacement schemes:** When a document is cached, the cache pays a cost in terms of the resources utilized to retrieve and store the document locally. Examples of resources include disk-space and bandwidth. It is important to design document placement and replacement strategies which can maximize the benefits while the costs are kept low.

2. *Reducing the message processing overheads at the caches:* The web caches have to process large numbers of messages from the server and the clients. Hence, it is important to optimize the communication stack, so that the overheads of processing each message are minimized.

3. *Reducing the disk I/O costs:* Caches read from and write documents into disks frequently. Therefore, it is necessary to optimize the disk I/O operations.

1.2.2 State of the Art in Dynamic Content Caching

Researchers have investigated various mechanisms to address the challenges of caching dynamic content. Research efforts in this area include augmenting proxy caching schemes to support dynamic content, designing new caching architectures for delivering dynamic content, techniques for increasing cacheable web content, fragment-based web content publication, improving the re-usability of cached content, and engineering low-cost mechanisms for supporting stronger consistency requirements.

Many schemes have been proposed to cache dynamic data at different stages of its generation and delivery process. While *browser caches* and *client caches* are located at each end-client, large organizations employ dedicated *proxy caches* to serve requests from multiple clients. Motivated by the idea of moving the data closer to the clients, *edge caching* or *content delivery network caching (CDN caching)* schemes store the contents at various geographically distributed caches. The *reverse proxy caches* or *front-end caches* are typically located within an origin site's infrastructure. These caches store dynamic web pages (or parts of web pages) that were dispatched from the web server. The *mid-tier* caches are logically located between the application servers and the database servers, and they store the results produced by the database servers in response to the queries issued by the application server. Aiming to increase the quantity of cacheable web content and to improve the reuse of cached data, researchers have also explored caching dynamic web content at various granularities such as whole-page caching, fragment caching, and query caching.

1.3 Contributions of the Thesis

Although the field of dynamic web content delivery has received considerable attention from the research community, there are several open problems that are yet to be addressed satisfactorily. This thesis is devoted to exploring novel cache-based architectures, techniques, and mechanisms for enhancing the scalability and the performance of dynamic web content delivery. The contributions of this thesis are epitomized by the following two innovations:

- We design a novel cooperative edge cache architecture, called *cooperative edge cache grid (cooperative EC grid, for short)*, for scalable and efficient dynamic web content delivery. The cooperative EC grid is distinguished by its two unique features. First, it incorporates scalable server-driven document consistency mechanisms, thereby ensuring efficient delivery of highly dynamic web content with varying server update frequencies. Second, the design of the cooperative EC grid focuses on the scalability and reliability of dynamic content delivery in addition to cache hit rates. Through the cooperative EC grid we demonstrate that low-cost cooperation can considerably enhance the benefits provided by the edge cache networks.

- We develop a novel framework for automatically detecting cache-effective fragments in dynamic web pages. Our automatic fragment detection scheme has three unique features. First, we identify two kinds of fragments, namely shared fragments and lifetime-personalization-based fragments, which benefit dynamic content delivery in two very distinct ways. Second, we design a novel fragment-aware data structure for representing dynamic web pages. Third, we provide two algorithms that can effectively detect the two kinds of fragments. Through this work we demonstrate the feasibility of designing automatic schemes for accurately fragmenting dynamic web pages.

These two major contributions encapsulate several techniques and algorithms, which we discuss briefly in the next two sections.

While the central focus of this dissertation is the efficient delivery of dynamic web content, many of the techniques and mechanisms explored in this thesis have wider applicability, and they can be used to improve scalability and performance in various kinds of distributed systems.

1.3.1 Cooperative Edge Cache Grid

Caching on the edge of the Internet has been a very popular technique for dynamic web content delivery. The fundamental philosophy of edge computing is to move data, and possibly parts of application closer to the clients. Motivated by this philosophy companies like Akamai [1] and Speedera [13] have designed and implemented massive edge cache

networks. These networks have large numbers of geographically distributed edge caches. Unlike proxy caching, the origin server has the knowledge about the edge caches. Hence, the edge cache networks can support stronger consistency requirements and can implement origin site-specific policies to achieve optimal performance.

Although the edge caching technology has been successfully commercialized, most of the current systems do not harness the capabilities of the edge caching technique completely. Many commercial edge cache networks [1, 13] still find the cost of server-driven consistency mechanisms to be very high, and they use the traditional time-to-live mechanism for maintaining document freshness. Hence, the edge caches in these systems cannot effectively cache highly dynamic documents that are updated frequently. We contend that cooperation among the individual edge caches can considerably reduce the overheads of server-driven consistency mechanisms, and can significantly enhance the benefits provided by the edge cache network. However, supporting effective low-cost cooperation in edge cache networks poses many problems. In this thesis we have identified four major challenges involved in designing a cooperative edge cache network.

These challenges motivate us to design the *cooperative edge cache grid* (or cooperative EC grid, for short) - a large-scale cooperative edge cache network based on the concept of cache clouds [89]. The cooperative EC grid is, to our knowledge, the first cooperative edge cache network for efficiently delivering dynamic web content with varying server update frequencies, whose design focuses on the reliability and scalability of dynamic content delivery in addition to cache hit rates. The goal of our research is to utilize the power of flexible and low cost cooperation in designing techniques and system level facilities for efficient delivery of dynamic web content in large-scale edge cache networks.

While the previous works on cooperative web caching utilize the concept of cache cooperation mainly for handling misses of caches, the edge caches of the cooperative EC grid cooperate for multiple purposes, such as:

1. **Cooperative miss handling:** When a cache in the cooperative EC grid suffers a miss, it tries to retrieve the document from a nearby cache rather than immediately contacting the origin server.

11

2. **Cooperative document consistency maintenance:** The edge caches cooperate with one another to reduce the overheads of server-driven document consistency maintenance mechanisms. In the cooperative EC grid, the server is communicate document updates to a few caches. The caches cooperatively distribute the update message to all the appropriate caches. item **Cooperative cache management:** The cache management policies adopted by an individual cache such as the document placement and the document replacement schemes are not only sensitive to the statuses of the other caches with which it is cooperating, but also to the utility of the documents to the other cooperating caches.

3. **Cooperative failure resilience and recovery:** The failures of individual caches are gracefully handled, with other caches sharing the load of the failed cache. Similarly, the caches in the EC grid aid in the recovery of failed caches, so that the cache that is recovering from its failure need not repeatedly contact the origin server for re-validating its documents.

Our design of the cooperative EC grid is based on a careful analysis of the various costs involved in delivering dynamic web content to the clients. The unique features of our work can be summarized as follows:

1. We introduce the concept of *cache clouds*, which forms the fundamental framework for cooperation in the cooperative EC grid. Cache clouds are a group of caches located in close network proximity which cooperate among one another to serve client requests and to maintain consistency of documents.

2. We have designed an Internet landmarks-based mechanism called the *selective landmarks-based server distance sensitive clustering scheme* (SDS scheme for short) for creating cache clouds in a cooperative edge cache grid such that the cooperation is both efficient and effective.

3. We present the architectural design of cache clouds, including a distributed and failure

resilient mechanism called the *dynamic hashing scheme* for object lookups and updates. The dynamic hashing technique balances the lookup and update loads among the caches belonging to a cloud, and can gracefully handle the failures of individual caches. The central idea of dynamic hashing has wider applicability. It can be utilized to design distributed, efficient, and failure-resilient lookup and update mechanisms in many large-scale distributed systems such as peer-to-peer networks.

4. We have developed a *utility-based scheme* for placing documents within the cache clouds, so that the available resources within the clouds are optimally utilized. This document placement scheme estimates the costs and benefits of storing a document in a particular edge cache, and stores the document at that cache only if the benefit to cost ratio is favorable. This document placement scheme consists of four components, each of which quantifies one aspect of the interplay between the benefits and costs.

The performance of the proposed architecture and techniques are studied both through analytical modeling and by experimental evaluation. Our study shows that the cache clouds-based architecture supports low-cost cooperation among the caches of the cooperative edge cache grid. Our experiments indicate that the SDS clustering scheme accurately clusters edge caches of the EC grid into cooperative clouds, thereby improving the performance of the EC grid. The experiments also demonstrate that the dynamic hashing scheme for distributed lookup and update protocols ensures high availability of the lookup information at low message costs. The dynamic hashing scheme also achieves good balancing of the lookup and the update loads among the caches belonging to the cloud. Further, we also show that the utility-based document placement scheme contributes positively towards the performance of the cooperative EC grid, by reducing the network load, and improving the hit rates of the cache clouds.

1.3.2 Automatic Fragment Detection in Dynamic Web Pages

Fragment-based publishing and caching of dynamic web pages is another popular technique for efficient delivery of dynamic web content. Several research groups have explored this area from different perspectives. It has also been successfully commercialized in recent

years. A fragment is a portion of a web page which has a distinct theme or functionality. The central idea of fragment-based techniques is to store the fragments independently at the server and at the caches. The web pages are composed on the fly using these fragments. Publishing and caching dynamic content at the granularity of fragments provides several advantages. It increases the cacheable content of the web sites, reduces the amount of data that gets invalidated at caches, and improves the disk-space utilization at the caches.

Most fragment-based dynamic content delivery techniques rely upon web pages that have been manually fragmented at their respective web sites. However, manual fragmentation of web pages is expensive, error prone, and unscalable. In this thesis we propose a novel scheme to automatically detect and flag fragments that are cost-effective cache units in web sites serving dynamic content [85, 86, 87]. Our approach analyzes web pages with respect to their information sharing behavior, personalization characteristics and change patterns, and detects fragments which are shared among multiple documents, or which have distinct lifetime or personalization characteristics. The proposed scheme has three unique features.

1. First, we propose a framework for fragment detection, which includes a hierarchical and fragment-aware model for dynamic web pages and a compact data structure for fragment detection.

2. Second, we present an efficient algorithm to detect maximal fragments that are shared among multiple documents.

3. Third, we develop a practical algorithm that effectively detects fragments based on their lifetime and personalization characteristics.

We have evaluated the proposed scheme through a series of experiments, showing the benefits and costs of the algorithms. Further, we also study the impact of using the fragments detected by our system on key parameters such as disk space utilization, network bandwidth consumption, and load on the origin servers. The experiments show that the fragments detected through the proposed scheme substantially improves the performance of the dynamic content delivery by effectively reducing data invalidations at the caches, and by improving their disk-space utilization.

1.4 Organization of Thesis

The rest of the thesis is organized as follows. Chapter 2 discusses the need for cooperation in edge cache networks and the challenges in designing large-scale edge cache networks. We also present a high-level architecture of the cooperative EC grid, and explain its design rationale by comparing it with other possible architectures.

In Chapter 3 we discuss the process of cache cloud formation in the cooperative EC grid. We explain the concept of Internet landmarks for quantifying the relative positions of the nodes in the Internet, and motivate the need for taking server distance into consideration while forming cache clouds. We then outline the selective landmarks-based distance sensitive clustering scheme (SDS scheme) for creating cache clouds. We also discuss a set of experiments that measure the accuracy of the SDS scheme, and its impact on the overall performance of the cooperative EC grid.

Chapter 4 describes the design architecture of individual cache clouds in detail. We explain the dynamic hashing-based mechanisms for document lookups and updates in cache clouds, and two techniques for making the lookups and updates resilient to failures of individual caches. Further, we also discuss the utility-based scheme for placing documents in cache clouds. We report several experiments to quantify the costs and benefits of the proposed techniques.

Chapter 5 describes our scheme for automatic fragment detection. We present a novel, fragment-aware data structure for modeling dynamic web pages called augmented fragment tree (AF tree for short). We explain two fragment detection algorithms, one for detecting fragments that are shared among multiple pages, and the other for detecting fragments having distinct lifetime and personalization characteristics. We also present the experimental evaluation of the proposed fragment detection scheme and discuss the impact of detected fragments on the performance of fragment-based web caches.

Chapter 6 discusses the prior related work in the area of dynamic web content delivery. We conclude the thesis in Chapter 7 by presenting an overview of the future work that we intend to pursue.

CHAPTER II

COOPERATIVE EDGE CACHE GRID

In recent years edge computing has emerged as a popular technique to address the challenges posed by the tremendous growth of dynamic web content to the scalability and performance of the World Wide Web. In addition to the various research projects being pursued in this area, edge computing has also been successfully commercialized.

The motivating philosophy of edge computing is to move data, and possibly some parts of the application, closer to the user. The vision of edge computing is to place the data/service such that any client request can reach the document or service it needs by traversing very few hops within the Internet. Towards realizing this vision, large content providers would maintain several caches at various locations on the edge of the Internet. Along with these edge caches the content providers also have one or more origin servers. The origin servers *offload* data and parts of the application to these edge caches. At the most basic-level the caches hold copies of documents generated by the origin servers. In more sophisticated implementations, parts of applications are also offloaded to these edge caches. An example of application offloading is fragment-based edge caching, wherein the functionality of composing the web pages from the fragments is offloaded to the edge caches.

Although the caches of edge cache network are geographically distributed, the origin servers have knowledge about the individual caches and they may maintain various kinds of information about the caches. This enables the content provider to serve dynamic web content from these edge caches while ensuring that any stringent freshness requirements on them are met. For example, the cached copies of a document may be explicitly invalidated or updated by the origin server when the document is modified at the origin site. Thus edge caching can support stronger consistency requirements than the traditional *time-to-live* mechanism. Further, the edge caches can adopt origin site-specific policies and schemes to optimize the performance of content delivery.

Edge computing provides significant performance and scalability benefits. First, as most client requests are serviced at the edge of the network, it reduces the load on the core backbone of the Internet. Second, the load on the origin server is substantially reduced in two distinct ways. The caches reuse the dynamic data generated by the origin server to serve future client requests, thus decreasing the amount of duplicate computations at the origin server. If parts of applications are offloaded to the edge caches, the load on the origin server is further reduced. Finally, the latencies experienced by the clients are also reduced since most requests are served at nearby caches.

2.1 Need for Cooperation in Edge Cache Networks

Many edge-caching schemes treat individual edge caches as independent entities. In these schemes the edge caches rarely communicate with one another, and they may not even be aware of other edge caches in the network.

Inspired by the work on cooperative proxy caching, a few edge cache networks like Akamai [1] have attempted to support cooperation among its caches. However, the cooperation in these edge cache networks is limited to serving client requests. Further, these systems employ the traditional time-to-live mechanism for maintaining consistency of documents, and they do not support server-driven consistency schemes that can provide stronger document consistency guarantees.

Designing efficient cache cooperation mechanisms in edge cache networks employing server-driven document consistency schemes is very attractive considering the potential benefits it can provide.

- First, when an edge cache receives a request for a document that is not available locally (i.e. the request is a local miss), it can try to retrieve the document from nearby caches rather than immediately contacting the remote server. Retrieving document from a nearby cache can significantly reduce the latency of a local miss. It also reduces the number of requests reaching the remote servers, thereby reducing the load on them.

- The second benefit of cooperation among edge caches is the reduction in the load induced by the document consistency maintenance on the origin servers. When the

edge caches are treated as independent entities, the origin server has to communicate document updates/

invalidations to each edge cache. In contrast, when the caches are organized as cooperative groups, the server can communicate the update message to a few caches, which in turn distribute the message to other edge caches that are currently holding a copy of the document.

- Third, when the caches of an edge network cooperate with one another, smarter cache management policies, which are sensitive to the cooperation among the edge caches, can be designed such that the performance of the edge cache network is further enhanced. An example of these cache management policies would be document placement and replacement schemes which consider the availability of the documents in other nearby caches.

Considering these performance benefits, in this thesis, we study the problem of designing cooperative edge cache networks that can support server-driven consistency mechanisms.

2.2 Challenges for Constructing Cooperative Edge Networks

Designing efficient cooperative edge networks poses several research challenges. In this section we discuss a few important challenges.

1. First, an effective mechanism is needed to decide the appropriate number of edge caches needed for the edge network. Further, it is also necessary to decide where these edge caches have to be located. In this thesis, we use the term *cache placement problems* to refer to these two related challenges. Having too few edge caches increases the load on each edge cache, as well as the average latency experienced by clients. On the other, increasing the number of caches increases both the infrastructure setup costs and the management costs of the system. The challenge is to optimize the setup and maintenance costs of the network, while ensuring that the client latency is within desirable limits. Similarly, the locations of the edge caches are likely to have profound impact on the overall performance of the edge cache network, since the locations of

the caches determine the number of Internet hops the client requests have to traverse in order to access the required data or service.

2. The second challenge in constructing a cooperative edge cache network is to decide the caches that should cooperate with one another. In other words how to group the caches into cooperative structures? On one end of the spectrum, we can design an edge cache network, wherein all the caches of the network cooperate with one another. At the other end of the spectrum, would be the scenario where none of the caches cooperate with one another, in which case the scheme is reduced to the edge cache network with no cooperation among the caches. The primary criterion that guides the decision in this regard would be optimize the both effectiveness and the efficiency of cooperation.

3. The third challenge is to design the architecture of the cooperative cache groups. In other words, the caches belonging to a cooperative group needs to be organized into logical structures. Further, highly efficient and failure-resilient mechanisms have to be designed for cooperation among the caches of a cooperative group. Two crucial components of cooperation are (a) document lookup protocol and (b) document update protocol. A cache suffering a miss on a client-request tries to locate a document copy within the cache-group. Similarly, when a document has to be updated all the copies of the document within the cache group have to be located. Therefore, edge cache networks serving frequently changing information requires highly efficient document lookup and update protocols.

4. The fourth challenge for edge cache network is to design cache management policies such that various resources of the cache group are optimally utilized. Two such cache management policies are the document placement policy and the document replacement policy. Most schemes only consider factors that are local to the cache and are oblivious to the fact that the caches are in a cooperative group. However, designing policies that are sensitive towards the cooperation in addition to considering the local factors can further enhance the performance of the edge cache network.

In this thesis we present the design of cooperative edge cache grid (Cooperative EC Grid for short), through which we attempt to address the second, third and fourth challenges listed above.

In this work, we assume that the scale of the network in terms of the number of caches it contains, and the appropriate locations of these edge caches are pre-decided. Previously, researchers in the area of content delivery networks (CDN) have proposed techniques to address similar cache placement problems. Most proposed approaches in the CDN area view the Internet as an undirected graph, where the data storage/access points (such as routers, servers, clients, edge caches, etc.) form the vertices of the graph, and the physical network connections form its edges. The cache placement problem itself is modeled as that of computing appropriate vertices for placing the caches, such that the cumulative sum of the access costs to all the cached data items is minimized. . Qiu et al. [81] and Jamin et al. [56] show that the cache placement problem is actually a variant of the minimum K-median problem in graphs, which has been proven to be NP-complete. These papers propose approximation algorithms based on techniques such as greedy strategy and heuristics to address the cache placement problems.

Although, the cache placement problems in edge cache network appear to be similar to the analogous problems in the area of CDNs, we think that a detailed study is required in order to determine whether the proposed solutions, or their variants, can adequately address the cache placement problem in edge cache networks, or whether it is necessary to explore very different approaches. We intend to undertake this study as a part of our future work.

2.3   Cooperative Edge Cache Grid

In this section we give a broad overview of the design architecture of the cooperative edge cache grid (cooperative EC grid), and discuss the salient features of our design. As we have mentioned before, the cooperative EC grid is a large-scale cooperative edge cache network.

Our design of the cooperative EC grid is based on the concept of *cache clouds*, which forms the fundamental framework of cooperation among the edge caches. Conceptually,

a cache cloud is a group of edge caches that are located in close network proximity. The caches belonging to a cache cloud cooperate with one another for three basic purposes.

- *Cooperative Freshness Maintenance:* When a documents that is cached in one or more caches of a cache cloud is modified at the origin site, the origin server is required to send a single update message to the cache cloud. The caches of the cloud cooperate to ensure that the update message reaches all the caches that are holding a copy of the document.

- *Collaborative miss handling:* When an edge cache within a cloud suffers a local miss, it attempts to retrieve the document from another cache in its cache cloud, rather than immediately contacting the origin site.

- *Cooperative cache management:* The management policies adopted by a cache, such as document placement and document replacement schemes, are sensitive to the state of the other caches in its cache cloud in terms of the resource and document availabilities at these caches.

The cooperative EC grid typically contains several cache clouds. The appropriate number of cache cloud within a cooperative EC grid, and the average size of each cloud depend on various factors including the scale of the edge cache grid, the capacity of the origin server, and the document request and update patterns. In addition to the edge caches which are organized into cache clouds, cooperative edge cache grid also contains an origin site which hosts the infrastructure for generating the dynamic web pages such as the web server, the application server, and the back-end databases. The origin site is also responsible for initiating the updates/invalidation messages when the documents undergo modifications. Figure 1 illustrates the high-level design architecture of the cooperative edge cache grid. The cooperative EC grid indicated in the figure has 22 edge caches, which are organized into 4 cache clouds.

The caches within a cache cloud interact with one another in a peer-to-peer fashion. In other words, the individual cache clouds have a flat structure with no hierarchy. In our cache cloud-design, all the caches belonging to a cloud share the responsibilities of lookups
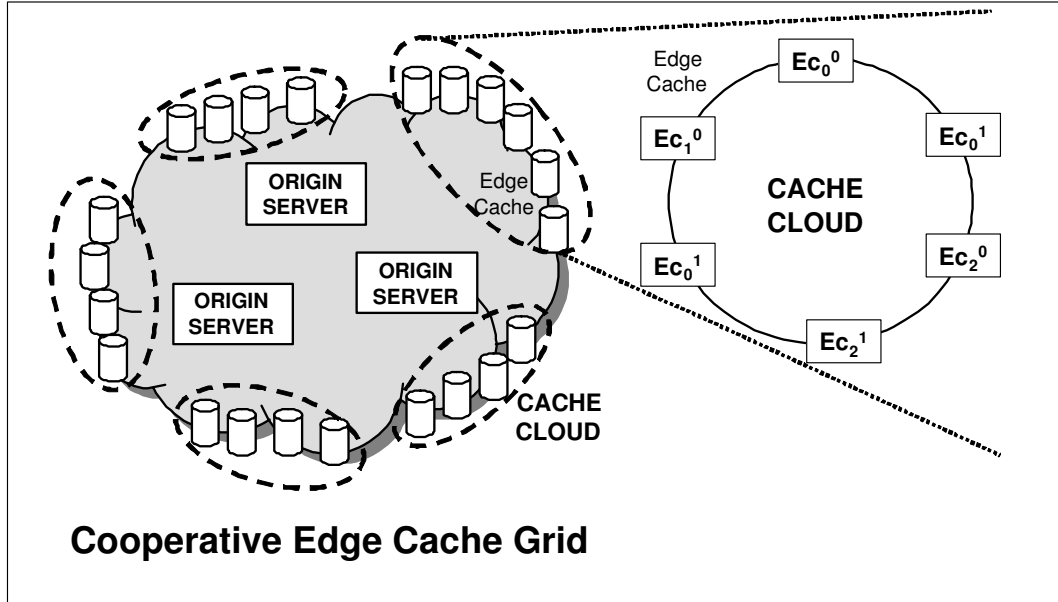
Figure 1: Architecture of Cooperative Edge Cache Grid

and updates of documents that are cached in the cloud. A cache may communicate with any other cache in its cloud, or with the origin server. Similarly the origin server may send messages directly to any cache in any cache cloud. The design of individual cache clouds are explained in detail in Chapter 4.

2.4 Comparison with Other Architectures

In this section we explain the rationale of our cooperative EC grid by comparing with two possible alternatives, namely the hierarchical architecture [37] and the completely distributed architecture [98]. First, we give a brief overview of these architectures.

As the name suggests, the hierarchical architecture for cooperation organizes the caches into hierarchical structures. The most commonly adopted hierarchical structure is the tree structure. Here, each cache is placed at a certain level of the tree-hierarchy, and is assigned a parent, and possibly one or more children. The origin server forms the root of the tree. Generally, the caches that are located closer to the origin server are placed at higher levels of the tree-hierarchy and vice-versa. The depth and the width of the trees may be varied,
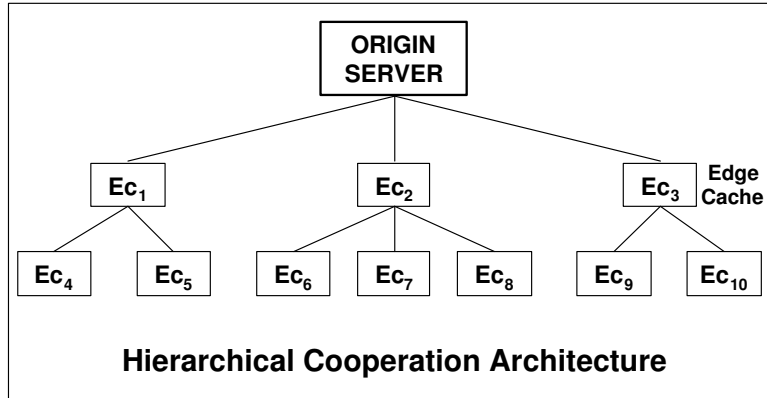
Figure 2: Hierarchical Cooperation Architecture

according to the needs. Figure 2 illustrates a tree-based hierarchical organization of 10 caches, with the tree-depth being 3.

In the hierarchical cooperation architecture, each cache communicates only with its parent and its children. Only the caches at the highest level of hierarchy can communicate with the origin server. In this architecture, the cache-miss and the document-update are handled as follows. When a cache Ec_l suffers a miss, it contacts it parent cache as well as its children caches. These caches check whether they or their descendants have the requested document. If the document is available at one of these caches, it is sent to Ec_l. Otherwise, the request is propagated to the cache located in the next higher-level of hierarchy. This process of request-propagation continues until the document is found in one of the caches receiving the request, or until the request reaches the origin server.

When a document is modified at the origin site, the origin server sends the update/invalidate message to the caches at the highest level of the tree-hierarchy. These caches in-turn communicate the message to their children and so on, until the message reaches all the caches holding a copy of the document.

The architecture that can be regarded as being diametrically opposite to the hierarchical organization is the pure distributed architecture. Here the cooperative structure is completely flat. An edge cache can communicate with any other edge cache in the edge

cache network, and with the origin server. In this architecture, when a cache Ec_l suffers a miss, it sends a message to all the caches in the edge cache network to check whether any of them have the document. If any of the caches have the required document, Ec_l retrieves the document from them. Otherwise, the cache Ec_l contacts the origin server to obtain the document. When a document is modified at the origin site, the origin server sends update message to all the caches in the edge cache network, or those caches that are currently holding a copy of the document.

A minor variant of the distributed architecture would be the scheme wherein the document lookup information is maintained within the edge cache network. In this variant, which we call as the distributed architecture with lookup information, each cache maintains the lookup information of a set of documents. The lookup information of a document indicates which caches in the entire edge cache network have a copy of the document. A cache Ec_l suffering a miss for a document, contacts the cache maintaining the document's lookup information, obtains the list of caches currently holding a copy of the document, and retrieves the document from one of those caches, or from the origin server, if none of the caches in the edge cache network has the document. Similarly, if a document is modified at the origin site, the origin server sends a *single* update message to the cache maintaining the document's lookup information, which communicates the message to all the caches in the edge cache network that are currently holding a copy of the document. We note that this variant is actually a special case of the cooperative EC grid architecture proposed by us, in which the cache grid always contains a single cache cloud.

We will now discuss why these alternative architectures are not very well suited for designing a large-scale cooperative caching scheme for dynamic content delivery. The hierarchical architecture has two major disadvantages. When a cache experiences a miss, the request to resolve the miss might have to traverse multiple hops, reaching the cache's ancestors at multiple levels of the hierarchy. Hence, in the hierarchical scheme the latency of handling a local miss is likely to be very high. The second disadvantage of the hierarchical scheme is that the caches located at higher levels of the hierarchy encounter very high document update loads. The caches located at the top most level of the tree hierarchy have

to handle every update message sent by the origin server. A related disadvantage is the load imbalance among the caches that are situated at different tree levels.

The distributed architecture is also not a suitable design choice. In the distributed architecture, processing a local miss is very costly in terms of the number of lookup messages circulated in the edge cache network. If the edge cache network has $CacheCount$ number of edge caches, each local miss at any of the edge caches causes $CacheCount - 1$ lookup messages to be circulated within the cache cloud. Similarly, when a document is modified at the origin site, the origin server has to send update messages to each cache that is currently holding the document, which places a heavy load on the origin server. Further, in the distributed architecture, an edge cache might have to frequently communicate with caches that are located far away in terms of their network distances. All these factors undermine the benefits of cooperation, and negatively impact the performance of the entire edge cache network.

Although, the distributed architecture with lookup information alleviates the first problem of high costs of miss processing, it fails to address the other two drawbacks of the distributed architecture. When a document undergoes modification at the origin server, the cache maintaining the document's lookup information has to potentially send $CacheCount-$ 1 messages. Hence, document updates are costly in this variant too. Finally, as in the distributed architecture, the edge caches may have to frequently communicate with other caches which might be situated quite far-off, thus affecting the efficiency of cooperation.

The cooperative EC grid architecture proposed by us does not suffer from the above drawbacks. First, the process of handling local miss is very efficient. When a cache Ec_l suffers a local miss, it contacts the document's beacon point to obtain the lookup information. Depending upon the response from the beacon point, Ec_l retrieves the document either from another cache within the cache cloud, or from the origin server. Therefore, the number of lookup messages circulated to handle a local miss is always 2. Further, unlike the hierarchical architecture the lookup message in the cooperative EC grid, has to traverse a single hop, which makes the lookup operations very efficient. Second, the number of messages sent-out by the origin server when a document is updated is never greater than

the number of clouds in the grid. Hence, the document update load on the origin server is very low. Third, as we explain in Chapter 4, all the caches in the cloud share the document update and the lookup costs. Therefore, any single cache in the cooperative EC grid is not overloaded with update and lookup operations. Finally, the edge caches in any cache cloud are located in close network proximity. Therefore, the communications among the caches belonging to a cloud are cheap, thus enhancing the efficiency of cooperation in EC grid.

We validate the above arguments through simulation-based experiments. For this purpose we consider an edge cache network consisting of an origin server, and 120 caches, which is simulated on top of an Internet topology generated through GT-ITM. GT-ITM is a network topology simulator developed at Georgia Tech. In our experiments the topologies were generated according to the hierarchical transit-stub model with 10 transit domains at the top level, each containing 5 routers on average. The transit routers have 10 stub domains attached to them on average. The stub domains in turn have an average of 10 routers each. The configuration-settings that we have used for generating the topologies has been adopted by several previous research project [31, 111]. The origin server and the caches are attached to randomly selected routers in the generated network topologies.

Using the caches of the edge network, we have simulated hierarchical cooperation architecture, distributed cooperation architecture, distributed cooperation architecture with lookup information, and the cooperative EC grid architecture. The caches and the origin server in each of the architectures are driven by request and update traces derived from a major IBM sporting event website [1](henceforth referred to as Sydney). The request logs contained client requests received in a 24 hour time period during the event and the update logs contained updates generated at the origin server during the same time period. The logs contained 52527 unique documents with the average size of each document being 65.3 Kilobytes. The requests were segregated based on their client-ids, and the requests from a few clients were combined to generate the request-logs, which were used to drive the edge caches. Clients were randomly assigned to the edge caches. In case the number of caches in the simulation exceeded the number of traces, extra traces were generated such that the

---

[1]The Sydney 2000 Olympic Games Website

statistical properties of the new traces are identical to one of the original traces.

The configuration details of the simulated architectures are as follows. For the hierarchical architecture, we construct a tree of depth 4, with origin server as the root of the tree. Each internal node in tree including the root has three children. We simulate the distributed architecture by organizing all the 120 caches in a single flat structure. In this architecture a flooding-based approach such as Internet Cache Protocol (ICP) [53] is adopted for document lookups and updates. In the distributed architecture with lookup information, the lookup data is distributed among all the caches, which share the document lookup and update loads. Finally, we also simulate a cooperative EC grid architecture with 10 clouds, wherein each cloud contains 12 caches. In the experiments reported in this chapter the disk-space at each cache in the simulation is set to 10% of the sum of sizes of all documents in the respective trace.

Table 1: Performance of Various Architectures for Cooperative Edge Networks

Performance Parameter	Hierarchical Architecture	Distributed Architecture	Distributed with Lookup Information	Cooperative EC Grid
Max Hops of Lookup Msgs.	7	2	2	2
Avg Hops of Lookup Msgs.	5.77	2.00	2.00	2.00
Max Msgs Circulated per Lookup	39	120	3.00	3.00
Avg Msgs Circulated Per Lookup	21.42	120.00	2.68	2.42
Max Msgs Circulated per Update	120.00	120.00	120.00	120.00
Avg Msgs Circulated per Update	120.00	120.00	63.44	68.06
Update Msgs Sent by Server per Unit Time	585.00	23400	195.00	1950.00
Max Update Msgs Sent by Cache per Unit Time	585.00	0.00	182.52	114.77
Max Comm Cost between Cooperating Caches	1127.6	1829.97	1127.6	683.3
Avg Comm Cost between Cooperating Caches	576.4	689.4	689.4	262.1

We provide a synopsis of the results of the experiments in Table 1. The experimental

results indicate that the average number of hops a lookup message traverses, and the average number of messages needed per lookup is the lowest for the cooperative EC grid architecture. Similarly, the number of messages circulated per object update in the cooperative EC grid, although slightly higher than that of the distributed architecture with lookup information, is still very low. The load induced by the update operations on individual caches is also the lowest for the cooperative EC grid. Finally, the cooperative EC grid yields the lowest values for both maximum and average communication costs between cooperating caches. The communication cost between cooperating caches is a very important factor which determines the overall benefits of cooperation. Hence, we note that cooperative EC grid architecture supports efficient cooperation among the caches of an edge network, and has the potential to harness the maximum benefits of cooperative caching of dynamic documents.

In summary, we note that the results in the Table 1 substantiate our discussions regarding the merits and demerits of each architecture.

2.5   Conclusions

Caching on the edge of the Internet has been a popular mechanism for serving dynamic web content to the end users. Most edge caching schemes to-date regard individual caches of the edge cache network as completely independent entities. However, cooperation among the edge caches can significantly enhance the performance and scalability of edge cache networks.

In this chapter we have studied the important research challenges in designing efficient cooperative edge networks. We have presented the architecture of cooperative EC grid, which has been designed with the objective of promoting low-cost cooperation among the caches of an edge cache network. The proposed cooperative EC grid architecture is based on the concept of cache clouds, which forms the basic framework of cooperation among the edge caches. We have compared the design of cooperative EC grid with other possible architectures, discussing the pros and cons of each approach. Our study indicates that the proposed cooperative EC grid architecture is very well suited for designing large-scale cooperative edge cache networks.

CHAPTER III

CACHE CLOUDS FORMATION IN COOPERATIVE EC GRID

In this chapter we address the problem of cache cloud formation in the cooperative EC grid. In other words, the problem we consider in this chapter is that of deciding which caches in the cooperative EC grid should cooperate with one another. As we discussed in Chapter 2, the caches belonging to a cache cloud are likely to interact with one another very frequently for the purposes of cooperatively serving misses and for maintaining consistency of cached documents. Therefore, the issue of cache cloud formation is a crucial one, and the manner in which the cache clouds are constructed can have profound implications on the overall performance of EC grid.

Although it is very commonly believed in the caching community that caches that are cooperating with one another should be located in close network proximity, the problem of accurately constructing cache clouds has not received much attention from the research community. To our knowledge, there are no techniques available for the system designers to group the caches in an edge cache network into cooperative clusters. Most cooperative caching solutions to-date relies upon network-administrators' sense of geographical vicinity for forming cooperative cache groups. We contend that this ad-hoc way of forming cache clouds adversely affects the cooperation efficiency, thereby leading to sub-optimal performance of the edge cache network.

In this chapter we systematically study the various aspects of this problem, and we present *selective landmarks-based server distance sensitive clustering scheme (SDS scheme for short)*, which is a novel technique that we have designed to address this vital challenge. Our technique uses the concept of *Internet landmarks* to accurately quantify the relative positions of the caches within the Internet, and constructs cache clouds based on the network proximity among the caches, and the network distances between caches and the origin server.

The main technical contributions of this chapter are along three directions:

- First, we show how the concept of landmarks can be utilized for cache-cloud formation in cooperative edge cache grid, and present an efficient strategy for selecting high quality landmarks such that the relative positions of the caches and the server are accurately quantified.

- Second, it is widely believed in the caching community that the caches that cooperate with one another should be in close network proximity. However, we experimentally illustrate the drawbacks of the strategies which exclusively consider the network distances between the caches as the sole factor for cache cloud creation. We discuss the reasons for these drawbacks, and thereby motivate the need for incorporating the network distances between the server and the caches as a factor in creating cache clouds.

- Third, we present the design of the SDS clustering scheme, which incorporates the distances between caches and the server together with the network proximities among the caches in creating cache clouds, thereby overcoming the drawbacks.

We have performed a series of experiments to study the various aspects of accuracy and performance of the techniques we have proposed. This chapter includes the description of the experiments we have performed and a discussion about the results obtained. Our experiments indicate that our techniques can significantly improve the clustering accuracy, and the performance of the edge cache network.

3.1   Problem Formulation

Let us consider an edge cache grid with an origin server (represented as Os) and $CacheCount$ number of edge caches. Let the set of edge caches be represented as $EcSet = \{Ec_0, Ec_1, \ldots, Ec_{CacheCount-1}\}$, where Ec_l, $0 \leq l < CacheCount$ denotes an arbitrary edge cache in the grid. As mentioned in Chapter 2, we assume that the decisions regarding where to place the edge caches have already been made, which means that the relative positions of the origin server and the edge caches are known to us. Specifically, we assume that the network

distances between any two nodes may be determined, if necessary. Given such an edge cache grid, the problem is to group the caches into *CloudCount* number of cooperative cache clouds, represented as $\{CCloud_0, CCloud_2, \ldots, Ccloud_{CloudCount-1}\}$, such that the performance of the entire cache grid is optimized.

Before attempting to design a solution to this problem, we enlist the factors that can influence the costs and benefits of cooperation, and hence need to be considered while creating cache clouds. The first factor influencing cache cooperation is whether the clients of the caches have overlapping interests, or in other words, whether there are any similarities among the request patterns at various caches, and if so to what extents they are similar. The second factor influencing the overheads of cooperation is the relative positions of the caches within the Internet. Clearly, cooperation is not beneficial if the caches serve clients with little commonality of interests, and hence have very different request patterns. In this thesis we assume that the caches have statistically similar request patterns. The fact that these edge caches belong to the same edge cache network implies that their clients have common interests, and hence we believe that it is reasonable to assume that these caches have statistically similar request patterns.

Let us now consider the performance criteria that need to be optimized while creating cache clouds. Two important performance criteria of cooperation are:

- **Effectiveness of Cooperation:** When designing cooperative edge cache grid we would want to reap the maximum possible benefits of cooperation in the sense that most client requests are satisfied by the cache receiving the request, or at least within the cloud to which the cache belongs, so that very few client requests need to be sent to the origin server. One way to quantify the effectives of cooperation would be in terms of the improvements in the hit rates of the individual caches achieved by cooperating with other caches in the cache cloud. A larger improvement in hit rates signifies higher effectiveness of cooperation, and vice-versa.

- **Efficiency of Cooperation:** One of the premises that motivates the idea of co-operation among caches of an edge network is that it is significantly less costly to

retrieve documents from caches within the cloud rather than obtaining them from the origin server. Efficiency of cooperation refers to cost savings obtained by fetching the documents from other caches in the cloud rather than obtaining them from the server. Therefore, an important parameter that has direct impact on the cooperation efficiency is the average cost of obtaining a document from another cache within the same cache cloud. One of the common ways to quantify cost is in terms of the latency involved. We refer to the average cost (latency) of retrieving a document from another cache in the same cache cloud as *intra-cloud interaction cost*. As the edge caches belonging to a cache cloud interact with one another very frequently for the purposes of document lookups, document retrievals, and document updates, the lower the intra-cloud interaction costs, the better it is for the performance of the EC grid.

It is important to consider both these factors while constructing cooperative clouds within an edge cache network. In other words, we have to create cache clouds such that interaction cost between any two caches of a cache cloud is minimized, while ensuring that the cumulative hit rates of all the cache clouds are reasonably high. However, the above factors are not always complementary, and there is a tradeoff between the two factors. Further, as we discuss later in the chapter this tradeoff affects the performance of cache clouds in different ways and extents depending upon the network distances between the clouds and the origin server.

As a first step towards the solution to cache cloud construction problem, we attempt to design a scheme that exclusively considers the intra-cloud interaction costs when grouping caches into cooperative clouds. An effective way of reducing the intra-cloud interaction cost is to construct cache clouds in such a manner that the caches belonging to one cache cloud are located in close vicinity within the Internet. This ensures that the network latencies of the communications between the caches of a cloud are minimized, thereby reducing the intra-cloud interaction costs. In other words, this scheme is based exclusively on the network proximities of the caches in the grid.

3.2 Selective Landmarks Scheme

In this section we discuss a cache cloud formation scheme, called the *selective landmarks scheme (SL scheme)* that is exclusively based on the network proximity of the caches. When we want to design such a scheme, two key questions need to be addressed at the outset:

1. How can the network distance between two nodes in the Internet be precisely quantified?

2. How can the network distance quantifications be utilized to accurately cluster the caches of an edge cache grid into cooperative clouds?

We use roundtrip time values (RTT values) for quantifying network distances between the nodes of an edge cache grid. RTT values have been utilized by many researchers in the fields of network measurement and modeling for quantifying the network distance between Internet nodes because it is easy to measure the RTT values between arbitrary pair of Internet nodes, and they provide precise quantification of the network distance. However, although we have used RTT values in our experiments, we believe that the techniques proposed by us for construction of cache clouds would work equally well with any reasonable measure of network distance (such as the minimum number of network hops between nodes).

We address the problem of utilizing the network proximity measurements to accurately group the edge caches into cooperative clouds by adopting the concept of *Internet landmarks* [40, 76, 96]. The concept of Internet landmarks originated to address the need for a positioning system to specify the locations of nodes within the Internet. Conceptually, Internet landmarks are a set of few key Internet hosts that serve as a frame of reference for determining the relative position of any other node on the Internet. An arbitrary host H_i measures the round trip time to each of these landmarks, and uses these values to determine its relative location in the Internet. The idea of landmarks has been adopted by researchers to assign coordinates to the nodes of wide area network (WAN) applications such as multicast overlay networks and peer-to-peer systems in order to improve their performance and scalability [111, 112]. For the purpose of brevity, in the rest of the thesis we refer to the Internet landmarks as just landmarks.

The selective landmarks (SL) scheme that we have designed works in three phases:

1. Choosing high quality landmark set

2. Probing landmarks and constructing feature vectors

3. Creating clouds through clustering edge caches

Generally the origin server coordinates the execution of the above three phases. However, any cache in the edge cache network might be entrusted with this responsibility. We refer to the node that coordinates the execution of the scheme as the *Cache Clustering-Coordinator* or the *CC-Coordinator* for short. We now explain each of these three phases of the SL scheme.

3.2.1   Choosing High Quality Landmark set

The first step in constructing the cache clouds is to choose a set of nodes to serve as the landmarks. These nodes serve as the frame of reference, and the nodes (caches and the origin server) of the cooperative EC grid repeatedly measure their network distance to these landmark nodes in order to determine their relative positions within the Internet. One approach to choosing landmarks would be to use some of the popular Internet hosts (such as machines hosting government or university web servers) as landmarks. However, this approach might raise administrative issues, since the nodes of the EC grid repeatedly send messages to the landmark nodes in order to determine their relative positions thus generating significant traffic. Hence, we choose all the landmarks from the nodes belonging to cache grid. Formally, if $LmSet$ denotes the set of landmarks, then $LmSet \subseteq (EcSet \cup Os)$ and $Os \in LmSet$ (recall that $ECSet$ represents the set of edge caches in the EC grid and Os represents the origin server).

Since the nodes in the landmark set serve as the frame of reference for specifying the locations of the server and the caches, the quality of the landmark set is likely to have a significant impact on the quality of the clouds that are generated by the scheme. Therefore, the edge caches that are included in the landmark set have to be chosen carefully. A simple solution to this problem would be to randomly select a few caches from the edge cache

34

network and include them in the landmark set. However, this simple approach may not always yield good quality landmarks set. Consider the case when two or more nodes that are chosen as landmarks lie in very close vicinity of one another. In this scenario the network distances from any other node to all of these landmarks would approximately be the same. Therefore, the position information conveyed collectively by all of these landmarks would not be significantly better than the position information conveyed by any one of them.

Before formulating an approach to address this problem, we first investigate the properties that the landmark set should satisfy in order to be considered as a high quality landmark set. The most important property that a good landmark set has to satisfy is that the landmarks have to be well dispersed among the set of nodes in the edge caches. If the landmarks are well distributed, then the position information obtained by using them as landmarks is better, hence improving the cluster quality.

The question that now arises is: *how to ensure that the landmarks are well dispersed within the edge cache grid?* One approach would be to choose the landmarks such that the distance between any two points in the landmarks-set is maximized. In other words, this approach attempts to maximize the minimum distance between any two landmark nodes.

Formally, let $MinRTT(LmSet)$ represent the minimum of the RTT values between any two nodes in the landmark set, i.e. $MinRTT(LmSet) = Min(RTT(Lp_i, Lp_j)) \, \forall \{Lp_i, Lp_j\} \in LmSet$. Suppose we decide to have $LandMarkCount$ landmark points, then $(LandMarkCount - 1)$ edge caches have to be chosen from the $EcSet$ such that the MinRTT value of the landmark set formed by those $(LandMarkCount-1)$ edge caches together with the origin server Os is maximized.

However, constructing the $LmSet$ that satisfies the above criterion of maximizing the $MinRTT$ value requires that we know the RTT values between every pair of edge caches, which results in significant measurement overheads for edge cache networks, especially for those networks with large number of edge caches. Further, even if the RTT values between every pair of edge caches were known, constructing the landmark set maximizing the $MinRTT$ value is equivalent to a discrete optimization problem, which is known to be NP-complete. Hence, we need to devise a computationally efficient strategy that can yield

landmark sets with $MinRTT$ values that are close to optimal

Considering the above requirements, we have designed an approximation-based greedy strategy for building the $LmSet$. Our approach for selecting the landmark set works as follows: First, the origin server is included in the $LmSet$. Therefore, we need to select $(LandMarkCount - 1)$ edge caches to be included in the landmark set. Initially, the CC-coordinator randomly selects $MFactor \times (LandmarkCount - 1)$ edge caches as *potential landmark points*, where $MFactor$ is a configurable parameter such that $MFactor \times (LandmarkCount - 1) \leq CacheCount$. We call this set of edge caches as *potential landmark set*, and denote it as $PLSet$. The CC-coordinator then sends a message to all potential landmark points informing them about the nodes that are included in the $PLSet$. The potential landmark points now probes the origin server and all the other caches in $PLSet$ multiple times, and determines its network distances to each of them.

Now we adopt a greedy strategy to choose $(LandMarkCount - 1)$ edge caches from the $PLSet$ which along with the server forms the landmark set. Now we adopt a greedy strategy to choose $(LandMarkCount - 1)$ edge caches from the $PLSet$ which along with the server forms the landmark set. Specifically, for each cache Ec_l in the $PLSet$, we examine the RTT values between Ec_l and the current members of $LmSet$, and assign the minimum of those values as the Ec_l's *inclusion potential* for the current iteration. We choose the cache in the $PLSet$ that has the highest inclusion potential for the current iteration, and include it in the $LmSet$. This cache is then removed from $PLSet$. At the end of $(LandmarkCount - 1)$ iterations we obtain a $LmSet$ whose $MinRTT$ value is close to the optimal $MinRTT$ value achievable for the set of edge caches in the edge cache network. At the end of this step, the CC-Coordinator sends the $LmSet$ to all the edge caches in the network.

Figure 3-C shows the execution of this phase of the algorithm. Here the $LandMarkCount$ is set to 3. Hence, the scheme has to select 2 edge caches, which along with the origin server forms the 3 landmarks. In this example the parameter $MFactor$ is set to 2. Therefore, $2 \times (3 - 1) = 4$ edge caches are randomly chosen to be included in the potential landmark set. In the example, the caches $\{Ec_0, Ec_1, Ec_3, Ec_4\}$ form the potential landmark set. These caches probe one another, and the origin server to obtain the RTT values between

A: Edge Cache Network

B: RTT Matrix

	Os	Ec$_0$	Ec$_1$	Ec$_2$	Ec$_3$	Ec$_4$	Ec$_5$
OS	0.0						
Ec$_0$	12.0	0.0					
Ec$_1$	8.0	4.0	0.0				
Ec$_2$	12.0	17.0	14.4	0.0			
Ec$_3$	8.0	14.4	11.3	4.0	0.0		
Ec$_4$	12.0	17.0	14.4	17.0	14.4	0.0	
Ec$_5$	8.0	14.4	11.3	14.4	11.3	4.0	0.0

C: Choosing the Landmarks

LandMarkCount = 3 m = 2 Potential Landmark Set = {Ec $_0$, Ec$_1$, Ec$_3$, Ec$_4$}

Initialization: LmSet = {Os}

Iteration 1: LmSet = {Os, Ec$_0$}   MinRTT(LmSet) = 12.0

Iteration 2: LmSet = {Os, Ec$_0$, Ec$_4$}   MinRTT(LmSet) = 12.0

Chosen Landmarks = {Os, Ec $_0$, Ec$_4$}

Figure 3: Choosing Landmarks for Cloud Constructions

them. The greedy phase of the algorithm starts with $LmSet = \{Os\}$. In the first iteration inclusion potentials of Ec_0, Ec_1, Ec_3 and Ec_4 are 12, 8, 8, 12 respectively. In this iteration Ec_0 is included into the $LmSet$, as it has one of the maximum inclusion potentials. For the second iteration the inclusion potentials are 4, 8, 12 for caches Ec_1, Ec_3 and Ec_4 respectively. Therefore, in this iteration Ec_4 is included in the $LmSet$. Now we have the final $LmSet$ consisting of Os, Ec_1, and Ec_4.

We now briefly analyze the worst-case complexity of this step of the cloud formation scheme. At the jth iteration of the greedy scheme, the algorithm computes the network distances between each of the $(MFactor \times LandMarkCountj + 1)$ remaining potential landmark points and the new landmark node that selected in the previous iteration. Hence the total number of distance computations in the jth iteration is $MFactor \times LandMarkCountj+1$. The algorithm terminates in $LandMarkCount-1$ iterations. Hence, the worst-case complexity of this phase of the algorithm is $O(MFactor \times LandMarkCount^2)$.

3.2.2 Probing Landmarks and Constructing Feature Vectors

The second step of our scheme is to construct the feature vectors of the edge caches. The feature vector of a cache Ec_j, represented as $FtVector_{Ec_j}$, specify the relative position of Ec_j within the edge cache network. The feature vectors can be constructed in a variety of ways depending upon the specific information they contain. For example works such as Global Network Positioning (GNP) [76] and Vivaldi [40] map the nodes into an N-dimensional Euclidean space. In these systems the feature vectors are the coordinates of the nodes in the N-dimensional space. Zhang et al. [111] used a binary string called landmark signature to encode the relative ordering of the elements in a measured landmark feature vector. The network proximities of nodes are captured by the similarity of their landmark identifiers.

In contrast to the above, our approach uses a simple and straightforward feature vector representation wherein the feature vector of a cache Ec_j contains the network distance values between the cache Ec_j and various landmark points. Specifically $FtVector_{Ec_j}[l]$ holds the average network distance value between edge cache Ec_j and the lth landmark point. Our experiments in Section 3.4 indicate that the cloud construction accuracy obtained by this simple feature vector representation is very similar to, and in some cases better than, the accuracy yielded by a more complex scheme such as GNP.

In order to construct the feature vectors, the caches probe the landmark points multiple times, and measure the roundtrip times of each message. The cache Ec_j builds its feature vector $FtVectorEc_j$, by recording the average of the RTT values of its messages to the lth landmark point at $FtVectorEc_j[l]$. Figure 4-A shows this phase of the SL scheme. The cache Ec_1 is pinging the three landmarks Os, Ec_0 and Ec_4. Similarly all caches probe the landmarks and build up their feature vectors. The figure also shows the feature vectors of all the edge caches in our example network.

3.2.3 Creating Clouds through Clustering

In the third step of the SL scheme, the edge caches are clustered based on their feature vectors to create the required number of cache clouds. Researchers in the fields of data management and pattern analysis have proposed various algorithms for data clustering
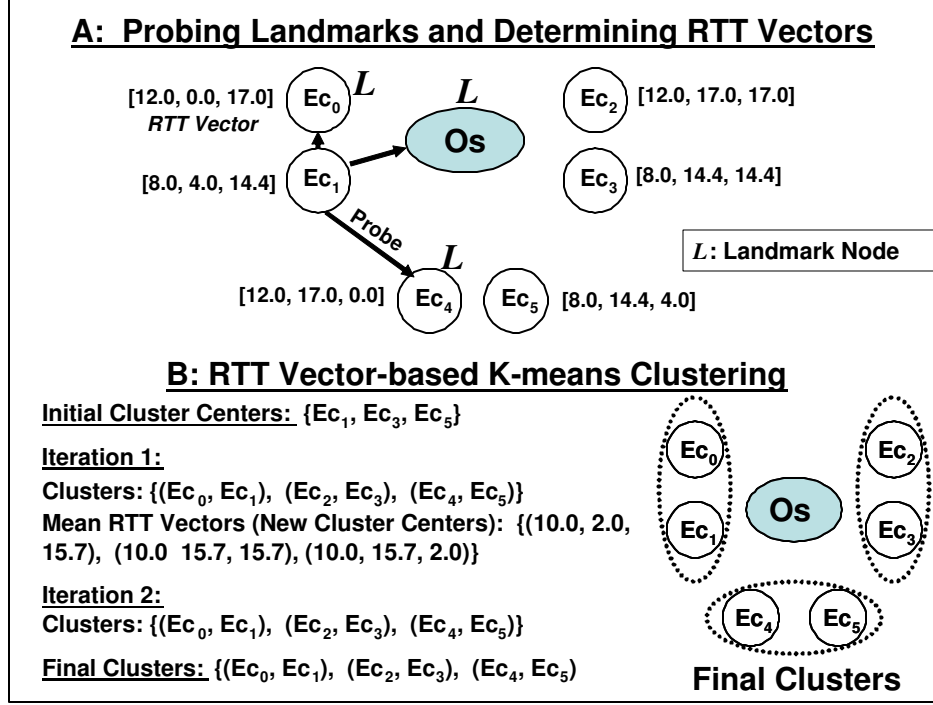
Figure 4: Feature Vector Determination and K-Means Cache Clustering

such as agglomerative clustering, K-means clustering, and nearest neighbor clustering [55]. In our work, we have used the K-means clustering algorithm, because it is known to yield accurate clusters, and it is computationally efficient. However, the techniques described in this paper are quite general, and can be adapted to any chosen clustering algorithm.

The K-means algorithm is a popular data-clustering scheme that clusters the data points into pre-specified number of groups. The clustering is based on the dissimilarities of the data-point feature vectors. The dissimilarity between two feature vectors may be quantified through any reasonable metric such as L_2 distance, L_∞ distance, or the cosine distance. In our scenario, each edge cache is represented by its feature vector (constructed in Step 2), and we use the L_2 distance between two feature vectors to measure the dissimilarity between the corresponding edge caches. If the feature vectors FTV_1 and FTV_2 are the feature vectors of edge cache Ec_1 and Ec_2 respectively, then the L_2 distance between them represented as $D_2(Ec_1, Ec_2)$ is given by:

$$D_2(Ec_1, Ec_2) = (\sum_{l=0}^{LandMarkCount-1} (FTV_1[l] - FTV_2[l])^2)^{\frac{1}{2}} \tag{1}$$

Although we have not experimented with other dissimilarity measures, we believe that substituting the L_2 distance with L_∞ distance, or the cosine distance does not adversely affect the accuracy and performance of the scheme.

We will now briefly outline the K-means clustering algorithm as applied to the current scenario. Suppose we decide to create $CloudCount$ number of clouds from the caches of the edge cache network. The K-means clustering scheme is an iterative algorithm, which works as follows:

1. First, the algorithm randomly chooses $CloudCount$ edge caches and designates them as *cluster centers*.

2. In the second step, each edge cache is assigned to its closest cluster center. Here the distance between an arbitrary edge cache and a cluster center is quantified as the L_2 distance between their feature vectors. At the end of this step we will have $CloudCount$ clusters with each cluster containing one or more edge caches.

3. In the third step , for each cluster, the algorithm computes the cluster's *mean vector*. The mean vector of a cluster is computed as the mean of the feature vectors of all caches belonging to that cluster. Suppose caches $\{Ec_1, Ec_2 \ldots Ec_p\}$ belong to cluster CCl_1, then the mean vector of CCl_1, represented as $MV(CCl_1)$ is calculated as $MV(CCl_1)[i] = \frac{\sum_{j=1}^{j=p} FTV_j[i]}{(p)}$, $0 \le i \le LandmarkCount - 1$. We now have $CloudCount$ mean vectors. These mean vectors are designated as the new cluster centers.

4. Now, for each edge cache the algorithm calculates its L_2 distances from the new cluster centers, and based on these distances checks whether the cache has to remain in the same cluster, or should be re-assigned to a different cluster (whose mean vector would be nearer than the mean vectors of all the other clusters including the cluster to which the cache belongs). Similar to Step 3, the caches are assigned to their closest cluster centers.

5. The process of cluster center computation (Step 3) and subsequent reassignment of

caches to clusters proceeds until the number of caches that are reassigned in the current iteration becomes minimal. At this stage the generated clusters are regarded as stable. Each cluster is assigned a cluster-ID. The algorithm outputs the cluster-IDs, and the caches belonging to those clusters.

Figure 4-B demonstrates this phase of the algorithm. In this example the caches Ec_1, Ec_3 and Ec_5 are chosen as the initial cluster centers. In the first iteration, the caches are assigned to their nearest cluster-centers. For example, the distances between the cache Ec_0 to the cluster centers Ec_1, Ec_3 and Ec_5 are 6.23, 15.17 and 19.80 respectively. So Ec_0 is assigned to cluster center Ec_1. At the end of this iteration we obtain the clusters $\{(Ec_1, Ec_0), (Ec_3, Ec_2), (Ec_5, Ec_4)\}$. The mean RTT vectors of these clusters are $(10.0, 2.0, 15.7)$, $(10.0, 15.7, 15.7)$, and $(10.0, 15.7, 2.0)$ respectively. These mean RTT vectors are used as the new cluster centers for iteration 2. In iteration 2, the distances between the caches and the new cluster centers are computed. A cache is re-assigned to a different cluster if that cluster's center is closer to the cache than its current cluster's center. In this example, all the caches remain in the same clusters as in the previous iteration. Therefore, the K-means clustering algorithm terminates with (Ec_0, Ec_1), (Ec_2, Ec_3) and (Ec_4, Ec_5) as the final clusters.

At the end of K-means clustering algorithm each edge cache is assigned to the cluster whose center is closest to it in terms of the L_2 distance metric. The feature vectors of the caches belonging to the same cluster can be expected to be "similar" to one another in the L_2 distance sense, which means that the caches belonging to various clusters produced by the K-means algorithm are in close network proximity, and the communication costs among the caches belonging to a cloud can be expected to be minimal. Thus, the SL scheme achieves the goal of constructing cache clouds such that the intra-cloud interaction costs are minimized.

3.2.4 Drawbacks of the SL Scheme

Although the SL scheme successfully minimizes the intra-cloud interaction costs, creating cache clouds purely based on the caches' network proximity is not sufficient to ensure optimal

cooperation among the caches of an edge cache network. In order to better understand why creating clouds just based on their mutual network proximity fails to yield optimal performance, we performed some simple experiments to study the performance of the SL scheme on critical parameters like intra-cloud interaction cost, cumulative hit rates of cache clouds, and average client latencies.

Before we explain our observations regarding the results of the experiments, we briefly outline the experimental setup and the evaluation methodology. Detailed discussion about the experimental methodology is presented in Section 3.4. For our experiments we use GT-ITM package [109] to generate the topology of the underlying wide area network. The value settings that we use for the configurable parameters of the GT-ITM package are similar to those used in the past by many researchers [31, 111], and are explained in Section 3.4. The origin server and the edge caches are placed at random locations on the generated network topology to yield the edge cache network. All the experiments reported in this section are on an edge cache grid consisting of 500 caches.

The caches of the experimental edge cache network are clustered through SL scheme to form cooperative cache clouds. The caches within a cloud cooperate with one another for serving misses, and for maintaining consistency of documents. The number of clouds in the edge cache network (represented as *CloudCount*) can be varied, and we study the performance of the SL scheme at various *CloudCount* values.
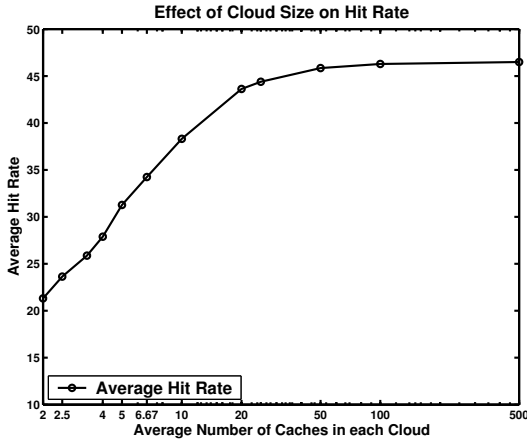
Figure 5: Effect of Cloud Size on Hit Rates

The graph in Figure 5 indicates the average cumulative hit-rate of the cache clouds in the edge cache network as the average size of the cache clouds in the edge cache network is varied. As indicated in the graph, when the average number of caches in the cache clouds increases, the cumulative hit rate also increases. This phenomenon can be explained as follows: increase in the size of a cloud implies that any cache in the cloud has larger number of peers with which it can cooperate. This leads to an increase in the cumulative resources (such as the total disk-space) of the cache cloud together with an increase in the cloud's client population, resulting in higher hit rates.
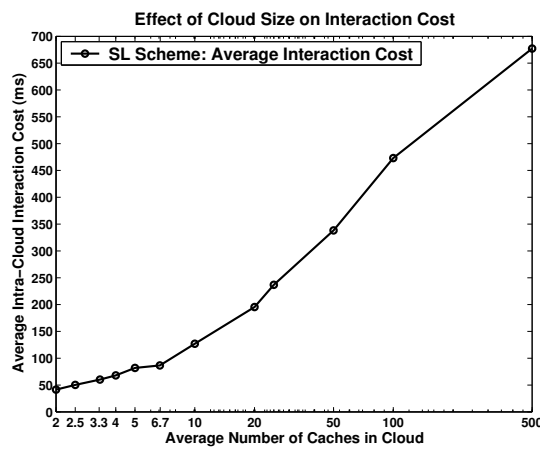
Figure 6: Effect of Cloud Size on Interaction Costs

Based on the results in Figure 5, one might conclude that creating larger cache clouds results in better performance of the edge cache network. To see why this is not always true, let us now consider the effect of cache cloud size on the intra-cloud interaction cost. The graph in Figure 6 shows the average cost of interaction between any two caches in the cache cloud as the size of the cache cloud varies. It can be seen that as the clouds get bigger, the average intra-cloud interaction cost increases. This is because, as a cache cloud grows larger in size, the caches belonging to it are more spread out in the network. Therefore, a cache in the cloud could now cooperate with a cache which may not be located very close to itself (in network-proximity sense), thus causing an increase in the average intra-cloud interaction cost.

From the graphs in Figures 5 and 6, we observe that larger cache clouds improve the effectiveness of cooperation (as indicated by the higher hit rates), but at the same time they also increase the cost of cooperation, thereby affecting the efficiency of cooperation. Thus, we observe that there is a trade-off between the effectiveness of cooperation (aggregate hit rates) and its efficiency (cooperation cost), with respect to the size of the cache cloud.

We study the effect of these counteracting phenomena on the performance of the edge cache network from clients' perspective by plotting the average client latency of the edge cache network at various cache cloud sizes (Figure 7). In order to better explain the short-comings of the SL scheme, along with the average client latency of the entire network, we indicate the average client latency of the 50 caches that are nearest (in the network proximity sense) to the origin server, and the average client latency of the 50 caches that are located farthest from the origin server. Initially all the three latencies (entire network, 50
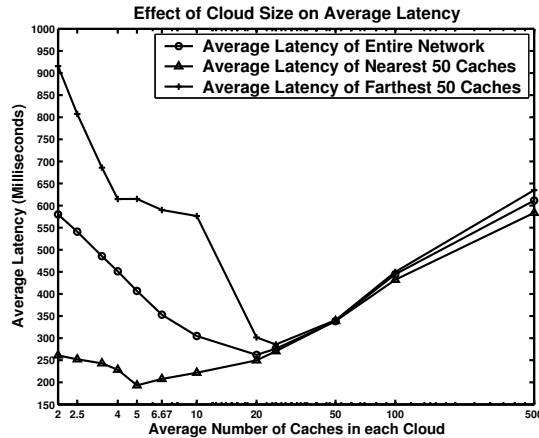
Figure 7: Effect of Cloud Size on Latency

nearest caches, and the 50 farthest caches) start decreasing as the cache clouds starts getting bigger. However, after reaching minimum values, all the three latencies start to increase, when the sizes of the cache clouds are further increased. This observation can be explained as follows. In the initial phase, the increase in the hit rates of the cache clouds (due to increasing cloud size) is the dominating factor influencing the average client latency. However, when the sizes of the cache clouds increases beyond a certain point the improvements

44

in the cumulative cloud hit rates start diminishing, and the growing intra-cloud interaction costs become the dominating factor affecting the client latency, thereby leading to higher average client latencies.

Another very important fact which emerges from the graph in Figure 7 is that the interplay between the cooperation efficiency (intra-cloud interaction costs) and the cooperation effectiveness (cumulative hit rates) influences the average client latencies of the cache clouds to different extents depending on their relative locations with respect to the origin server. This is evident when we observe the minimum values of the average client latencies for the entire edge cache network, 50 nearest caches to the origin server, and the 50 farthest caches from the origin server. If we consider the average client latency of the entire network, its minimum value occurs when the cache clouds contain around 20 edge caches on an average. On the other hand the average client latency of the 50 nearest caches to the origin server reaches its minimum value when cache clouds contain around 5 caches on an average, whereas the minimum value of the client latencies of the 50 farthest caches occurs when clouds are much larger (containing about 25 caches on an average).

The main shortcoming of the SL scheme is that it cannot adapt these variations in the interplay between the intra-cloud interaction costs and cumulative hit rates when clustering the edge caches to form cache clouds. For example, if use K-means algorithm to cluster the caches into 100 clouds (i.e. each cloud contains 5 cache on average), the 50 nearest caches yield very good latency values, whereas the performance of the 50 farthest caches are far from being optimal. Alternatively, if we create 20 clouds (with each containing 25 caches) in order to optimize the latencies of the 50 farthest caches, the caches that are located closer to the origin suffer in terms of their average latencies. Thus, irrespective of the number of clouds created in the edge cache network using the SL scheme, a considerable fraction of the caches in the network would be yielding sub-optimal performance.

Before attempting to overcome this drawback of the SL scheme, we need to locate which aspect of the SL scheme is causing the drawback. Clearly, the first step (selecting the landmarks) and the second step (determining the feature vectors) are just determining the relative positions of the caches and the server of the edge cache network, and have

little influence on the trade-offs between the effectiveness and efficiency of cooperation. Therefore, the shortcoming of the SL scheme is due to the K-means clustering algorithm (Step 3), which clusters the caches into cooperative groups exclusively based on the network proximity of the caches.

Next, we discuss why the interplay between the intra-cloud interaction costs and the cumulative hit rates affects the performance of different clouds in different ways and to different extents. The primary reason for this observed behavior is the variations in the costs of processing cloud-wide misses at different edge caches. A cloud-wide miss occurs when none of the caches in the cloud contain the requested document, in which case the document has to be retrieved from the origin server. The cost of retrieving documents from the server varies among caches depending upon their relative network distance from the server. For caches that are located faraway from the origin server, the cost of a cloud-wide miss is likely to be very high, whereas for the caches that are situated closer to the origin server, this cost would be relatively low.

The K-means clustering algorithm used in the SL scheme does not consider these variations in the trade-off between the efficiency and effectiveness of cooperation at different cache clouds, which results in the above drawback.

3.3   The SDS Scheme

In this section, we describe the selective landmarks-based server distance sensitive clustering scheme (SDS scheme), which has been specifically designed to address the drawback of the SL scheme. As we discussed in the preceding section, the drawback of the SL scheme was caused by the K-means clustering algorithm used in the third step. Hence, the first and the second steps of the SL scheme are incorporated into the SDS scheme without any changes. We have suitably modified the K-means clustering algorithm to overcome the drawback discussed in the previous section. We call this algorithm *server distance sensitive clustering (SDS clustering, for short)*, and it forms the third step of the SDS scheme.

The design of the SDS clustering scheme is based on the following observation. Since the cost of processing of a cloud-wide miss would be very high for the caches that are

located far from the origin server, achieving very high cumulative hit rates would be crucial for their performance. Therefore, in the tradeoff between the hit-rates and the intra-cloud interaction costs, achieving higher hit rates should be given precedence for the faraway clouds. In contrast, for the caches that are located near the origin server the costs of reaching the origin server are low. Hence, for these caches cooperation is beneficial only if the costs of interacting with other cooperating caches are minimal. So, for these caches minimizing the intra-cloud interaction costs should be given higher priority in the tradeoff between the hit-rates and the intra-cloud interaction costs.

Therefore, the problem now is to construct cache clouds such that maximizing the hit rates is given priority for faraway clouds, and at the same time minimizing the intra-cloud interaction costs is accorded higher precedence for clouds that are closer to the origin server. In the previous section we observed that the hit rates of the clouds increase as the clouds grow larger, and the intra-cloud interaction costs decrease as the clouds become smaller. Therefore, in order to satisfy the above criteria we have to create small clouds (containing fewer caches) near the origin server, and progressively increase the size of the clouds as we move farther away from the origin server.

The next question that has to be answered is what strategies can be designed to implement this key idea? In other words can we appropriately modify the existing clustering schemes to incorporate this idea? In this thesis we limit our discussion to adopting the K-means clustering algorithm for incorporating the above idea. However, any clustering algorithm can be similarly modified.

Recall that in the K-means clustering algorithm, if $CloudCount$ number of clouds were needed, the algorithm *randomly* selects $CloudCount$ caches as initial cluster centers. In the K-means algorithm, any cache may be selected to an initial cluster center with equal probability (which is equal to $\frac{CloudCount}{CacheCount}$). The strategy we adopt to incorporate the above idea (of creating smaller clouds nearer to the origin server and vice-versa) in the K-means clustering would be to choose larger fraction of the originators closer to the origin server, and select fewer originators as we move farther from it. In other words, in the SDS scheme, the probability that an edge cache is chosen as an initial cluster center is made inversely

proportional to its distance from the origin server. Specifically if $Pr(Ec_l)$ represents the probability of selecting the edge cache Ec_l as an initial cluster center, and if $Dist(Ec_l, OS)$ represents the network distance between the origin server and Ec_l, then:

$$Pr(Ec_l) \propto \frac{1}{Dist(Ec_l, OS)^\theta} \tag{2}$$

where θ is a configurable system parameter that controls the sensitivity of the SDS scheme towards the distances of the clouds from the origin server. When θ is set to higher values, the scheme is more sensitive to server distance, and vice-versa.

After selecting originators which satisfy Equation 2, the SDS scheme proceeds in a similar fashion as the SL scheme. Specifically, at each step the caches are assigned to their nearest cluster centers, and the cluster-centers are re-computed at the end of step. The algorithm terminates when the number of caches that change clusters between any two consecutive steps becomes minimal.

It may be noted that if θ is set to 0.0, then the SDS clustering scheme performs exactly like the SL scheme. Thus, the SL scheme can be regarded as a special case of the SDS scheme.

In Figure 8, we highlight the contrasts between the SDS scheme and the SL scheme by providing a two dimensional illustration of the clouds yielded by each of them on an example edge cache network. Recall that the only difference between the SL and SDS schemes is the clustering algorithm they use to group the caches into clouds. Therefore, the contrasts shown in Figure 8 are actually due to the different clustering algorithms used. In the figure we see that while K-means clustering yields nearly equal-sized clouds, in the SDS clustering scheme the clouds located near the origin server are small whereas the clouds located farther away from the server are larger in sizes.

We now analyze the complexity of the clustering phase of the SDS scheme. At each iteration of the clustering phase, the algorithm calculates the L_2 distances between the cluster centers and all the caches in the system. Therefore, the complexity of each iteration is $O(CacheCount \times CloudCount)$. The total number of iterations required for the algorithm to converge is dependent upon the actual dataset at hand. However, previous studies [55]
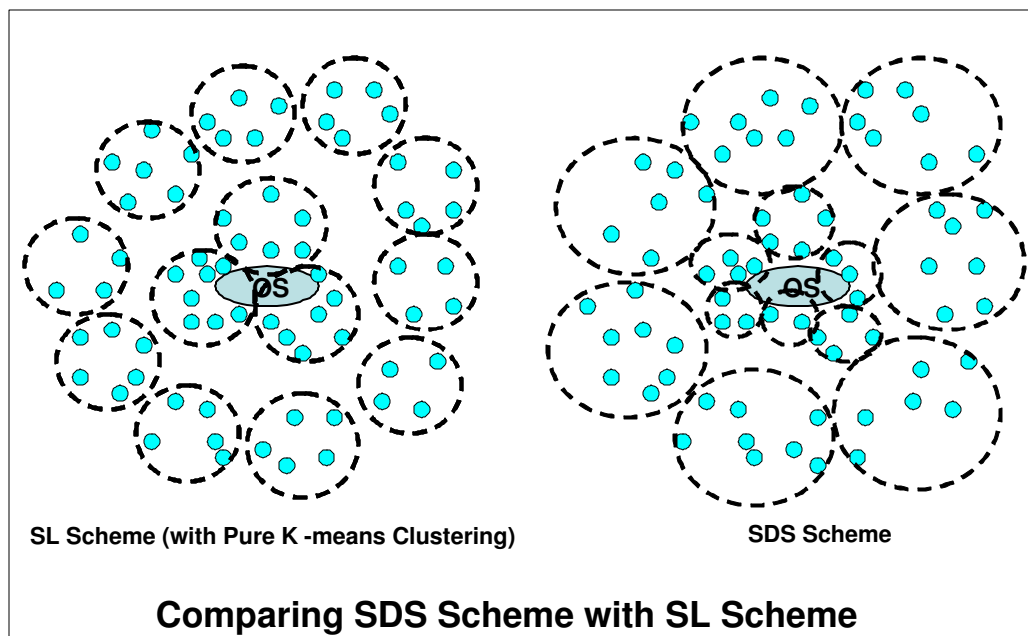
Figure 8: Comparing SL and SDS Schemes

have shown that the K-means clustering algorithm, of which the SDS scheme is a variant, converges reasonably fast.

3.4 Experiments and Results

We have performed a range of experiments to evaluate various aspects of the SL and the SDS mechanisms for constructing cache clouds. The goals of these experiments are three fold:

1. Evaluating the impact of selectively choosing the landmarks on the clustering accuracy.

2. Studying the effect of the feature vector representation used in the SL and the SDS clustering schemes on the clustering accuracy.

3. Evaluating the effects of server distance sensitive clustering when constructing cache clouds.

This section describes the experiments we performed and the results we obtained. We begin by giving an overview of our experimental setup and the datasets we have used in our experiments.

3.4.1 Experimental Setup

We have implemented a discrete event simulator that simulates a cooperative EC grid. The caches receive requests repeatedly from their respective request-log files. If the requested document is available within the cache it is recorded as a local hit. Otherwise, if the document is available within the cache cloud, the request is a group hit. If the document is not available in any of the caches in the cache cloud, then the request is a miss, in which case the document has to be retrieved from the origin server.

The request-logs used in our experiments were derived from the publicly available proxy server and client traces from Virginia Tech. These are a set of four traces each containing between 13,127 and 227,210 requests. The total disk-space requirements for storing the complete set of documents in the traces ranged between 0.159 GB and 2.30 GB. We have used only the three largest traces to derive the logs for our experiments, since the smallest trace had too few requests. The requests were segregated based on their client-ids, and the requests from a few clients were combined to generate the request-logs, which were used to drive the edge caches. Clients were randomly assigned to the edge caches. In case the number of caches in the simulation exceeded the number of traces, extra traces were generated such that the statistical properties of the new traces are identical to one of the original traces.

The simulations were executed on different network topologies that were generated through the GT-ITM network topology generator according to the hierarchical transit-stub model [109]. The configuration that we used to generate the topologies had 10 transit domains at the top level, each containing 5 routers on average. The transit routers have 10 stub domains attached to them on average. The stub domains in turn have an average of 10 routers each. The configuration-settings that we have used for generating the topologies has been adopted by several previous research project [31, 111]. The origin server and the

caches are attached to randomly selected routers in the generated network topologies. The disk-space at each cache in the simulation is set to 5% of the sum of sizes of all documents in the respective trace.

3.4.2   Evaluating the Accuracy of Selective Landmarks

In this set of experiments we study the improvements in clustering accuracy gained by choosing the landmarks selectively. For this purpose we consider edge cache grids of different sizes (containing between 100 and 500 edge caches). We now create cache clouds using three different techniques, and compare the accuracy of clouds obtained through them. The first technique, which we call the *random clustering*, creates clouds by randomly clustering the edge caches. In this technique each cache is randomly assigned to one of the clouds. In the second technique, we randomly select landmarks, and construct the feature vectors of all the edge caches using these randomly selected landmarks as frame of reference. We then use the K-means clustering algorithm to group the caches into specified number of clouds. The third technique is the SL scheme, wherein we use the selective landmarks-based greedy strategy (discussed in Section 3.2.1) for choosing the landmarks. These landmarks form the frame of reference for constructing feature vectors of the edge caches. The K-means clustering algorithm is used to cluster the caches into clouds. The goal of these experiments is to exclusively study the performance benefits obtained by adopting the selective landmarks mechanism. Hence, the same feature vector representation is used in both the second and the third techniques. Further, we use the pure K-means clustering algorithm in both of them. The accuracy of the clusters obtained from the three schemes is quantified using the average intra-cloud interaction cost (recall that average intra-cloud interaction cost is defined as the mean latency of transferring an average-sized document between any two caches belonging to the same cloud).

The graph in Figure 9 indicates the average intra-cloud interaction costs of the three schemes (in milliseconds) as the number of caches in the edge cache network varies from 100 to 500. In this experiment the number of clouds is always set to be 10% of the total number of caches in the grid (for example the number of clouds for edge cache grid with 100 caches
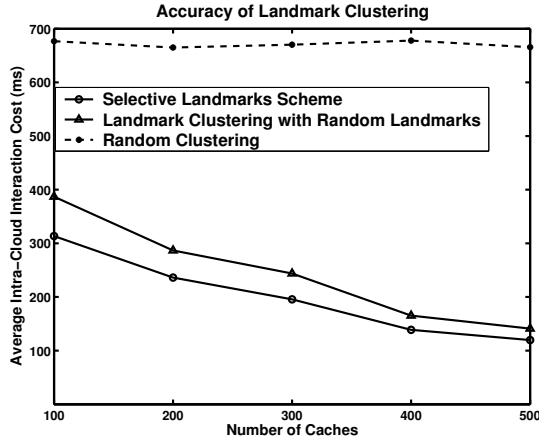
Figure 9: Accuracy of Cloud Creation Techniques

is set to 10, and so forth). In the graph we see that the random clustering always performs poorly, yielding very high intra-cloud interaction costs. This is because random clustering assigns caches to clusters without taking into account their relative positions within the network. Hence, the costs of interaction between caches belonging to the same cluster are likely to be very high. The intra-cloud interaction costs of both random landmarks scheme and the selective landmarks scheme (SL scheme) decrease as the number of caches in the network grows. This phenomenon can be explained as follows: As the edge cache network grows in terms of the number of caches, the average network distance between any two caches in the network decreases, thus reducing the costs of interaction between caches in the same cluster.

The graph in Figure 9 also indicates that average intra-cloud interaction costs of the selective landmarks scheme is always lesser than that of the random landmarks scheme. The improvements provided by the selective landmarks scheme over the random landmarks scheme range between 16% to 20% indicating that employing the landmarks scheme can significantly improve the accuracy of clustering.

In the second experiment (Figure 10) , we study the effect of number of clouds on the performance of random landmarks and selective landmarks clustering mechanisms. For this experiment we consider a cache clouds consisting of 500 caches and vary the number of
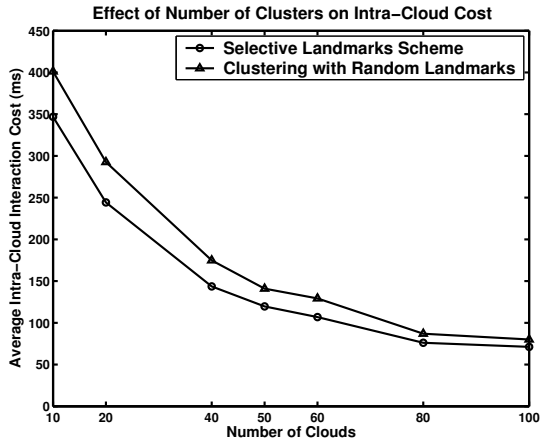
Figure 10: Effect of Number Clouds on Accuracy

clouds from 10 to 250. The intra-cloud interaction costs for both random and selective landmarks scheme decrease as the number of clouds increases. As the number of clouds in the edge cache grid increases, the individual clouds become smaller and hence, the intra-cloud interaction costs decrease (please see the discussion in Section 3.3). Similar to the previous experiment, the selective landmarks-based clustering always yields better intra-cloud interaction costs than the random-landmarks clustering. Thus, from Figure 9 and Figure 10 we conclude that employing our scheme for choosing landmarks improves the clustering accuracies irrespective of the number of caches in the edge cache network or the numbers of clouds being created, thereby providing considerable performance benefits.

3.4.3 Evaluating Feature Vector Representation

In both SL and the SDS schemes, the relative positions of the caches and the server within the network are specified through their feature vectors. We have chosen a simple representation scheme for these feature vectors, wherein the network distances from a cache to various landmarks forms the components of the cache's feature vector. The goal of our next experiment is to evaluate this method of representing the feature vectors.

Recently, researchers have proposed various schemes to map the nodes of Internet into an N-dimensional Euclidean space based on their relative distances to various landmarks. In these schemes the coordinates of a cache in the N-dimensional space forms its feature

vector. The primary motivation for these works is to minimize the error between the actual network distances between nodes and the corresponding L_2 distances between their feature vectors (or coordinates in the N-dimensional space). However, the process of mapping the nodes into a Euclidean space is very computation intensive. For example, mapping an edge cache grid containing 500 caches and 25 landmarks through the GNP takes about 2 hours and 27 minutes on a machine with Pentium-P3 900MHz processor and 512 MB RAM, running Linux-2.4.20. In this experiment we compare the performance of our feature vector representation method to one such Euclidean-space mapping scheme called the Global Network Positioning, or GNP for short [76]. GNP has been a popular Euclidean-space mapping technique. Its properties have been well studied, and it is known to generate accurate Euclidean-space mappings.

We consider an edge cache grids with the numbers of edge caches ranging from 100 to 500. The number of clouds is set to 10% of the total number of caches in the network. The caches in the edge cache networks are grouped to form cache clouds using two schemes. The first is the SL scheme, wherein we selectively choose 25 landmarks, and construct the feature vectors of caches and origin server. Then K-means clustering algorithm is employed to cluster the caches based on their feature vectors. In the second technique, we use the selective landmarks scheme to choose 25 landmarks, and determine the network distance of the origin server and the edge caches from these landmarks. Now the edge caches and the origin server are mapped into a 8 dimensional space through the GNP software [76], using standard settings. The K-means clustering algorithm is used to cluster the caches based on their coordinates in the 8-dimensional Euclidean space.

The graph in Figure 11 indicates the average intra-cloud interaction costs of the two techniques as the number of caches in the grid increases from 100 to 500. The performances of both the SL and the Euclidean-space clustering schemes are very similar to each other for all edge cache grids. However, the SL scheme performs slightly better than the Euclidean-space clustering scheme for most edge cache networks. This result was unexpected, since the primary motivation of mapping into Euclidean-space mapping is to minimize the error between the actual network distances between the nodes and the L_2 distances between their
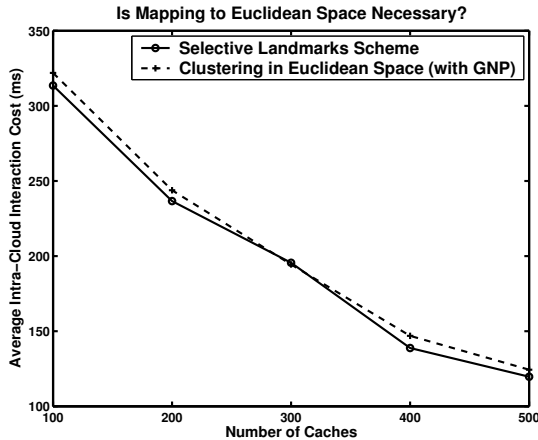
Figure 11: Effect of Euclidean Space Mapping on Accuracy

co-ordinates (feature vectors).

In order to find out the reasons for this unexpected result, we carefully studied the GNP software that was used for mapping the nodes into Euclidean space. We analyzed the mean and the variance of the error between the actual network distances between each pair of nodes, and the L_2 distances of the feature vectors (co-ordinates) before and after mapping into Euclidean-space. Our study supports the claim in [76], that mapping into Euclidean space reduces the mean error between the actual network distance and the L_2 distance between their feature vectors. However, when we analyzed the variance of error, we found out that variance of the error after Euclidean-space mapping was an order of magnitude higher than the variance of the error before mapping into Euclidean-space. This observation implies that although GNP scheme minimizes the mean error on a global scale, the error distribution among the edge caches is skewed. Some edge caches experience very high error values during the mapping, whereas others experience low error values. As the clustering accuracy is influenced by the relative positions of the edge caches rather than their absolute positions within the space, we hypothesize that some nodes having very high error values adversely impacts the clustering accuracy, thereby resulting in higher average intra-cloud interaction costs when the origin server and the edge caches are mapped to Euclidean space. However, we think that this issue merits detailed experimentation and study.

From the results of this experiment, we conclude that for clustering application, the simple feature vector representation we have used not only suffices, but in some case may yield better clustering results than Euclidean-space mappings. Therefore, the costly process of Euclidean-space mapping of caches can be avoided without any performance penalty.

3.4.4 Effect of Number of Landmarks on Clustering Accuracy

In this experiment we evaluate the effect of the number of landmarks on accuracy of clustering achieved by various scheme. The bar graph in Figure 12 indicates the average intra-cloud interaction cost for the three schemes (random landmarks with K-means clustering, selective landmarks with K-means clustering, and K-means clustering on Euclidean-space mappings) for an edge cache network of 500 caches when the number of landmarks is set at 5, 10 and 25.
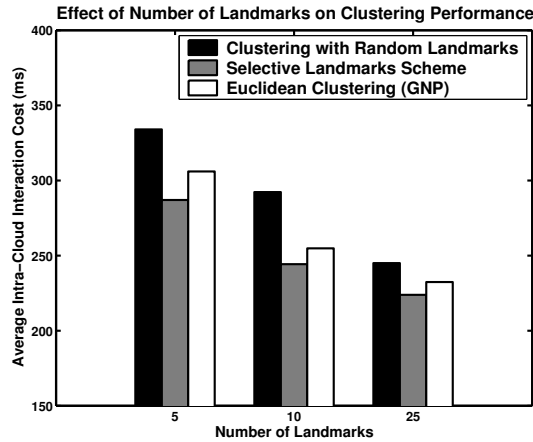
Figure 12: Impact of Number of Landmarks on Accuracy

The graph shows that the clustering accuracies of all the three schemes improve as the number of landmarks increases. When the landmark set is small, increasing the number of landmarks yields significant improvements in clustering accuracy of all the three schemes. For example when the number of landmarks is increased from 5 to 10 the average intra-cloud interactions costs of the random landmarks, selective landmarks and Euclidean clustering improves by about 13%, 15%, and 17% respectively. However, when the number of landmarks are increased beyond 20, the improvements in terms average intra-cloud interaction

costs of all three schemes are very minor, thus indicating that the around 20 landmarks are sufficient of creating good-quality cache clouds. Also, the selective landmarks-based clustering scheme performs better than both random landmarks-based clustering and Euclidean-space clustering irrespective of the number of landmarks. For example, the performance achieved by random landmarks-based clustering with 25 landmarks is almost equal to selective landmarks-based clustering scheme with just 10 landmarks.

3.4.5   Comparing SDS Scheme and SL Scheme

In the final set of experiments we experimentally evaluate the benefits of incorporating the distances between the server and various caches as a parameter in constructing the edge cache clouds. For this purpose we consider an edge cache grid with 500 caches. We construct the cache clouds through both the SL and the SDS clustering schemes and compare their performance at various *CloudCount* values. In both schemes the number of landmarks is set to 25. We quantify the performance of the schemes by measuring the average client latency of the cache clouds set generated by the two algorithms.
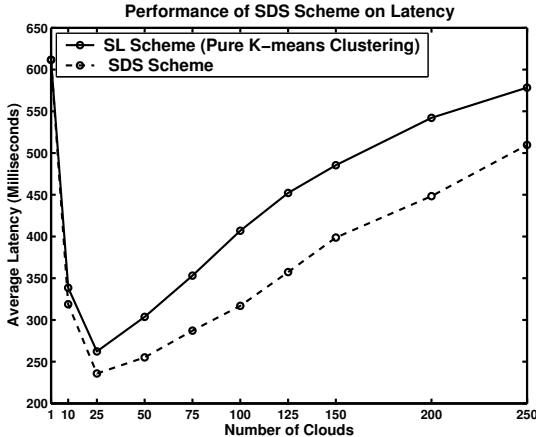
Figure 13: Performance of SDS Scheme on Latency

Figure 13 indicates the average client latency (in milliseconds) of the SL and SDS schemes at various numbers of cache clouds. We see that at all cloud count values, the clouds constructed through the SDS scheme yields lower latency values than those constructed through the SL scheme. For example, when *CloudCount* is set to 100, the SDS

scheme improves the average latency by around 22%, when compared with the SL scheme.
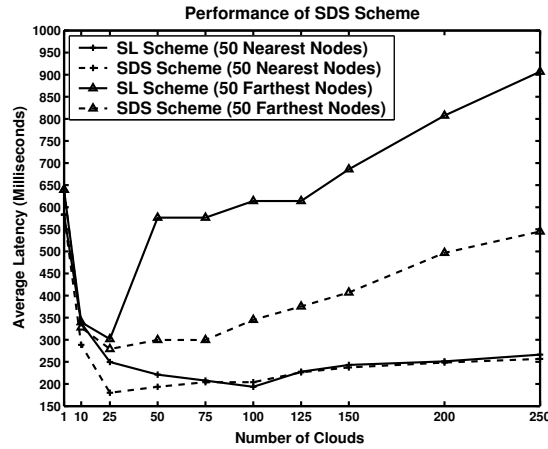
Figure 14: Effect of Distance SDS Scheme on the Nearest and the Farthest Caches

In Section 3.2.4 we had argued that as a consequence of the shortcomings of the K-means clustering algorithm, irrespective of the number of clouds created through the SL scheme, a considerable fraction of the caches in the network would be yielding sub-optimal performance. In order to see whether the server distance sensitive clustering scheme can successfully overcome this drawback, we plot the average latency values of the 50 caches that are nearest to the origin server, and the 50 caches that are farthest to the origin server at various *CloudCount* values. Figure 14 shows these latency values for both SL scheme (which uses the K-mean clustering algorithm) and the SDS scheme (which uses the server distance sensitive clustering algorithm).

The results lead to some very interesting observations. First, the server distance sensitive algorithm significantly improves the average latency of the caches that are located farthest from the server. For example, when *CloudCount* is set to 75, the server distance sensitive clustering reduces the average latency of the 50 farthest caches by around 48%. Second, for the caches that are situated closest to the origin server, the SDS scheme either improves the average latency, or provides similar performance as that of SL scheme for most *CloudCount* values. Only in a small region of the X-axis, when the *CloudCount* values are around 100, the SL scheme provides slightly better performance (around 4%) than the SDS algorithm.

58

However, in the same region server distance sensitive clustering algorithm improves the latency of the 50 farthest caches by around 44%. Third, as we discussed before, with the K-means clustering algorithm, the *CloudCount* value that yields optimal performance for a cache varies depending on its relative distance to the origin server. However, as the results in Figure 14 show, with the server distance sensitive clustering scheme, the 50 nearest and the 50 farthest caches both provide optimal performance when *CloudCount* is set around 25. Thus, we note that server distance sensitive clustering scheme can successfully overcome the main drawback of pure k-means algorithm for constructing cache clouds.

3.5 Conclusions

An important challenge in designing the cooperative EC grid is to organize its caches into cache clouds such that the cooperation is both effective and efficient. In this chapter we have proposed *selective landmarks-based distance sensitive clustering (SDS clustering) scheme* - a landmarks-based mechanism for organizing the edge caches into cache clouds. Our scheme uses the concept of Internet landmarks for accurately quantifying the relative positions of the server and the caches of the EC grid. It incorporates an efficient mechanism for selecting high quality landmarks, and a simple feature vector representation scheme for specifying the relative positions of the origin server and the edge caches. The SDS scheme also includes a clustering technique that considers both the mutual network proximities of the caches and the network distances between the origin server and the individual caches. The experiments presented in this chapter indicate that organizing the caches into clouds through the SDS scheme can significantly improve the performance of the cooperative EC grid.

CHAPTER IV

DESIGN OF CACHE CLOUDS

In this chapter we will describe the design of cache clouds in detail, and discuss the experiments we have performed to evaluate our design.

As we have briefly outlined in Chapter 2, cache clouds form the fundamental framework for cooperation in the cooperative EC grid. The caches belonging to a cache cloud cooperate with one another in three distinct ways, namely, collaboratively serving misses, cooperatively handling document updates, and optimally utilizing the resources within the cache cloud. Therefore, the design of the cache clouds should incorporate efficient schemes to support collaborative handling of misses and updates, and it should also be sensitive towards the resource availability within the cloud.

In order to collaboratively handle misses, a cache that needs to retrieve a document should be able to locate the copies of the document (if any) existing within the cache cloud. We refer to the mechanism of locating the copies of a document within a cache cloud for the purpose of retrieving it as the *document lookup protocol*. Similarly, the cache cloud design should also incorporate a *document update protocol* through which updates to documents are communicated to the caches in the cloud that are currently containing them.

Fundamentally, there are two approaches to cache cloud design, namely, centralized and distributed. In centralized design, a single designated cache handles the lookups and updates of all the documents. Contrastingly, in a distributed design, the document lookup and update responsibilities are distributed among some or all of the caches in the cloud. We have adopted a distributed approach to designing the cache cloud architecture, since it provides better scalability, load-balancing, and failure resilience properties.

In our design all the caches in the cloud share the functionalities of lookups and updates. Each cache is responsible for handling the lookup and update operations for a set of documents assigned to it. In a cache cloud, if the cache Ec_l^j is responsible for the lookup

and update operations of a cached document Dc, then we call the cache Ec_l^j as the *beacon point* of Dc. Each document cached in a cache cloud has an edge cache as its beacon point. The beacon point of a document maintains the up-to-date lookup information, which includes a list of caches in the cloud that currently hold the document. A cache that needs to serve a document Dc, contacts the Dc's beacon point, and obtains its lookup information. Then it retrieves the document by contacting one of the caches currently holding the document. Similarly, if the server wants to update or invalidate document Dc, it sends an update/invalidation message to its beacon point. The beacon point then distributes this message to all the cache holders of the document. Thus, in our design each edge cache in the cloud plays dual roles. As a cache it stores documents, and responds to client requests. As a beacon point it provides lookup and update support for a subset of documents served by its cache cloud.

An immediate question that needs to be addressed is how to decide which of the caches in the cloud should act as the beacon point of a given document. We refer to this as the beacon point assignment problem. In designing the Cache Cloud architecture, our goal is to assign beacon points to documents in such a manner that the following properties are satisfied:

- Both the caches within the cache cloud and the origin server can discover the beacon point of a document efficiently.

- The document lookups and updates are resilient to failures of individual beacon points.

- The load due to document lookups and updates is well distributed among all beacon points in the cache cloud.

- The beacon point assignment and its load balancing scheme should be dynamic and adaptive to the lookup and the update patterns change over time.

A straightforward solution for the beacon point assignment problem would be to use a random hash function. These hash functions uniquely hash the document's URL to one of the edge caches (beacon points) in the cache cloud, which acts as the document's beacon

point. We refer to this scheme as the *static hashing* scheme. The static hashing scheme has two significant drawbacks: First, the static hashing scheme provides no resilience to failures of individual beacon points. In this scheme when an individual beacon point fails the entire beacon information maintained by it is lost, and hence the only way to handle the lookups and updates of the documents that were assigned to the failed beacon point would be to flood the cache cloud with the respective messages. Second, although a good hashing function can distribute the documents equally among the beacon points, it still cannot ensure good load balancing among them. Lookup and update loads often follow the highly skewed Zipf distribution, and random hashing cannot provide good load balancing under such circumstances.

Consistent hashing [57] has been popular as a technique for providing good load balancing among a network of nodes. In this technique, the set of documents and the set of edge caches are both mapped on to a unit circle in terms of document identifiers and cache node identifiers. Each document is assigned to the cache node (its beacon point) that is nearest to its identifier on this circle. While consistent hashing can provide good load balancing properties, it significantly increases the lookup and the update costs, especially for those documents that are hot or that are updated frequently. If a cloud contains N edge caches (beacon points), with consistent hashing the beacon point discovery process might take up to $log(N)$ beacon point hops. This makes the consistent hashing scheme less attractive to the scenarios where the performance of the lookups and updates is very crucial. Further, in both simple hashing and consistent hashing mechanisms the assignment of documents to beacon points is static. Therefore these schemes cannot preserve load balancing when the update and lookup load patterns change over time.

Considering the shortcomings of the above schemes, we propose a dynamic hashing-based mechanism for assigning the beacon point of a document. This mechanism supports very efficient lookup and update protocols, provides good load balancing properties, and can adapt to changing load patterns effectively.
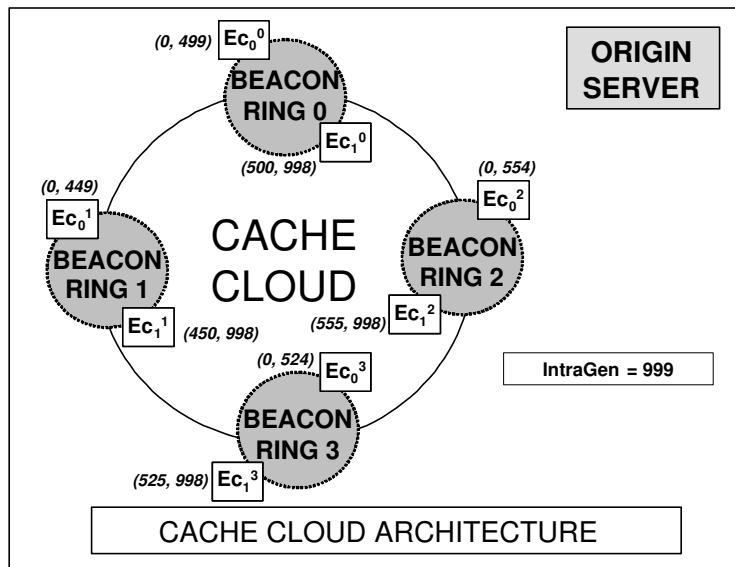
Figure 15: Architecture of Edge Cache Cloud

4.1 Design of Dynamic Hashing Scheme

Consider a cache cloud with N edge caches. Each of these caches maintains lookup information about a set of documents. In the dynamic hashing scheme, the assignment of documents to beacon points can vary over time so that the load balance is maintained even when the load patterns change.

In our scheme the edge caches of a cache cloud are organized into substructures called *beacon rings*. A cache cloud contains one or more beacon rings, and each beacon ring has two or more beacon points. Figure 15 shows a cache cloud with 4 beacon rings, where each beacon ring has 2 beacon points. All the beacon points in a particular beacon ring are collectively responsible for maintaining the lookup information of a set of documents. In our scheme each document is uniquely mapped to a beacon ring. This mapping of documents to beacon rings is done through a random hash function. Suppose the cache cloud has M beacon rings numbered from 0 to $M-1$. A document Dc is mapped to beacon ring j where $j = MD5(URL(Dc)) \bmod M$. Here $MD5$ represents the MD-5 hash function, and

URL(Dc) represents the unique identifier of the document Dc.

Suppose the document Dc is mapped to the beacon ring j, which has K caches in it. One of the caches within the beacon ring j would be assigned to serve as the primary beacon point for the document Dc. The primary-beacon point of a document maintains the up-to-date lookup information of the document, and is responsible for handling the lookup and updates of the document. All the other caches in the beacon ring serve as backup nodes for maintaining the document's lookup information. When the primary beacon point of a document fails, the lookup and update requests would be handled by one of the backup nodes. Hence, the lookups and update operations on documents can proceed without interruption even when some individual beacon points experience failures. There are two basic approaches for replicating the lookup information on the backup nodes, namely, eager replication and periodic replication. We will explain these two replication schemes in Section 4.2.1.

The question that has to be addressed is: how is the primary beacon point assignment done? Or in other words for any document, say Dc which is mapped to beacon ring j, how do we decide which of the K caches would acts as its primary beacon-point. A simple and straightforward solution to this problem would be to designate one of the caches in the beacon ring j to be the primary beacon point of all documents that are mapped to that particular beacon ring. In this scheme all the other caches in the beacon ring act as backup nodes. Although, this scheme is attractive due to its ease of implementation, it has two major disadvantages. First, the cache that is assigned to serve as the primary beacon point encounters the entire lookup and update loads of all the documents that are mapped to the bacon ring, and hence can become a hotspot. Second, the scheme is not adaptive to changing load patterns. In order to address these concerns we propose *dynamic hashing scheme*, which not only balances the lookup and update loads among all the caches belonging to a beacon ring, but is also designed to be adaptive to changing load patterns.

Let us suppose that the K caches belonging to the beacon ring j are represented as $\{Ec_0^j, Ec_1^j, \ldots, Ec_{k-1}^j\}$. The dynamic hashing technique uses *intra-ring hash* function for distributing the documents to the K beacon points. The intra-ring hash is designed as

follows. An integer, which is relatively large compared to the number of beacon points in the beacon ring is chosen and designated as the intra-ring hash generator (denoted as $IntraGen$). The range of intra-ring hash values $(0, IntraGen-1)$ is divided into K consecutive non-overlapping sub-ranges represented as $\{(0, MaxIrH_0^j), (MinIrh_1^j, MaxIrh_1^j), \ldots,$ $(MinIrh_{N-1}^j, IntraGen - 1)\}$. Each edge cache within the beacon ring is assigned to be responsible for one of such sub-ranges in the sense that this cache will be chosen as the primary beacon point for all the documents hashed into that sub-range. For example, the beacon point Ec_l^j is assigned to be responsible for the range $(MinIrH_l^j, MaxIrH_l^j)$. The process of splitting the intra-ring hash value range into a collection of sub-ranges is dynamic. The sub-ranges are updated periodically to adapt to load pattern changes. We explain the sub-range determination process in Section 4.2.

The scheme also hashes each document's URL to an integer value between 0 and $(IntraGen - 1)$. This value is called the document's intra-ring hash value or IrH value for short. For example, for a document Dc, the IrH value would be $IrH(Dc) = MD5(URL(Dc))$ Mod $IntraGen$, where $MD5$ represents a MD5 hash function, $URL(Dc)$ represents the URL of the document Dc and Mod represents the modulo function. Ec_l^j will serve as the primary beacon point of a document Dc, if $IrH(Dc)$ lies within the sub-range currently assigned to it.

4.2 Determining the Beacon Point Sub-Ranges

In this section we explain the mechanism of dividing the intra-ring hash range into sub-ranges such that the load due to document lookups and updates is balanced among the beacon points of a beacon ring. This process is executed periodically (in cycles) within each beacon ring, and it takes into account factors such as the beacon point capabilities, and the current loads upon them. Any beacon point within the beacon ring may execute this process. This beacon point collects the following information from all other beacon points in the beacon ring.

- **Capability:** Denoted by Cp_l^j, it represents the power of machine hosting the cache Ec_l^j. Various parameters such as CPU capacity or network bandwidth may be used

as measures of capability. We assume a more generic approach wherein each beacon point is assigned a positive real value to indicate its capability.

- **Current Sub-Range Assigment:** Denoted by $(CMinIrH_l^j, CMaxIrH_l^j)$, it represents the sub-range assigned to the beacon point Ec_l^j in the current cycle.

- **Current Load Information:** Represented by $CAvgLoad_l^j$, it indicates the cumulative load due to document lookup and update propagation averaged over the duration of the current period. The scheme can be made more accurate, if the beacon points also collect load information at the granularity of individual IrH values, denoted by $CIrHLd_l^j(p)$, which indicates the load due to all documents whose IrH value is p. However, the $CIrHLd$ information is not mandatory for the scheme to work effectively.

After obtaining this information the process of determining the sub-ranges for the next cycle begins. The aim is to update the sub-ranges such that the load a beacon point is likely to encounter in the next cycle is proportional to its capability. For each beacon point we verify whether the fraction of the total load on the beacon ring that it is currently supporting is commensurate with its capability. If the fraction of load currently being handled by a beacon point exceeds its share then its sub-range shrinks for the next cycle, thus shedding some of its load. On the other hand if a beacon point is handling a smaller fraction of load, its sub-range expands increasing its load for the next cycle.

Specifically, the scheme proceeds as follows. First, we calculate the total load being experienced by the entire beacon ring (represented as $BRingLd^j$), and the sum of the capabilities of all the beacon points belonging to the ring (represented as $TotCp^j$). Then for each beacon point we calculate its appropriate share of the total load on the beacon ring as $AptLd_l^j = \frac{Cp_l^j}{TotCp^j} \times BRingLd^j$. Now, we examine all the beacon points in the beacon ring starting from Ec_0^j, and compare their $CAvgLd$ with their $AptLd$. If $CAvgLd_l^j > AptLd_l^j$, then Ec_l^j is currently supporting more load than its appropriate share. Hence, the scheme decides to shrink the sub-range of the beacon point for the next cycle. This is done by decreasing its $CMaxIrH_l^j$ value. The amount by which the $CMaxIrH_l^j$ is decreased is
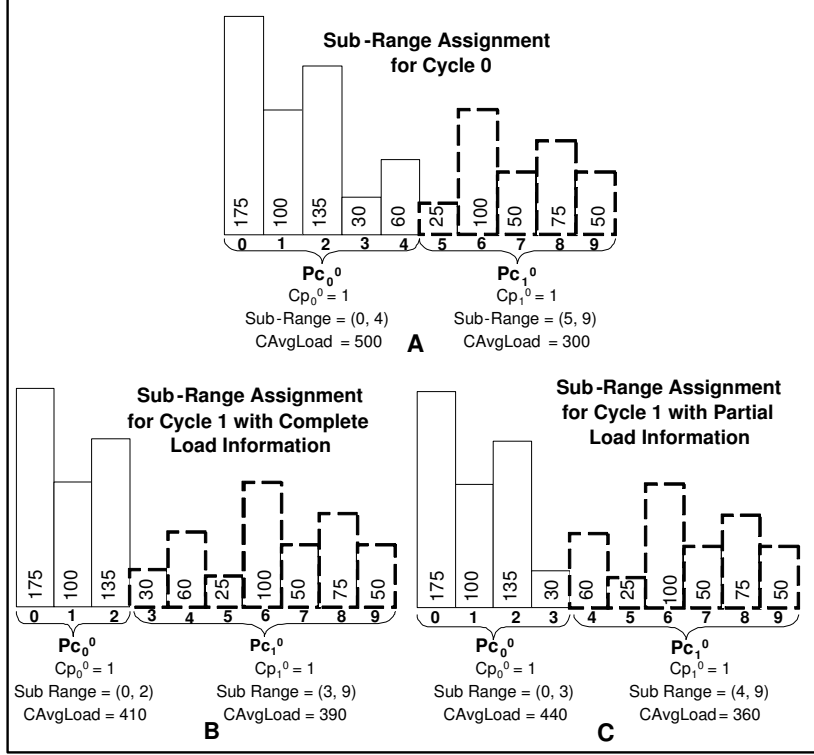
Figure 16: Illustration of Sub-Range Determination

calculated using the $CIrHLd$ information. $CMaxIrH_l^j$ is decreased by a value t such that $\sum_{p=CMaxIrH_l^j-t}^{CMaxIrH_l^j} CIrHLd_l^j(p) \approx (CAvgLd_l^j - AptLd_l^j)$. Therefore, the new $MaxIrH$ value represented as $NMaxIrH_l^j$ would be equal to $(CMaxIrH_l^j - t)$. When the sub range of a beacon point Ec_l^j shrinks, some of its load would be pushed to the beacon point Ec_{l+1}^j. The scheme takes into account this additional load on the beacon point Ec_{l+1}^j when deciding about its new sub-range.

We now illustrate the sub-range determination scheme with an example. Consider the beacon ring with two beacon points Ec_0^0 and Ec_1^0. Let both the beacon points have equal capabilities, and let $IntraGen$ be 10. Initially the range $(0, 9)$ is divided equally between the two beacon points. Figure 16-A illustrates this scenario. The vertical bars represent the update and object lookup loads corresponding to each hash value. As we see equal division of the intra-ring hash range does not ensure load balancing between the two beacon points due to the skewness in the load. The total load experienced by the two beacon points in

67

cycle 0 are 500 and 300 respectively. At the end of cycle 0, the sub-ranges are updated taking into account the current load patterns. Now we consider 2 scenarios. Figure 16-B represents the first scenario, wherein the beacon points maintain $CIrHLd$ for each hash value. In this case, two hash values are moved from Ec_0^0 to Ec_1^0. The loads on the two beacon points would now be 410 and 390 respectively. In the second scenario, which is represented in Figure 16-C, the beacon points do not maintain the $CIrHLd$ information, and hence they have to use $CAvgLd_l^0$ to approximate the $CIrHLd$ value, which would be 100 for all hash values belonging to Ec_0^0. We shift only one hash value between beacon points. The loads on the two beacon points would be 440 and 360 thus showing that the scheme is more accurate when the load information is available at the granularity of IrH values.

On the other hand if $CAvgLd_l^0 < AptLd_l^0$ then the scheme expands the sub-range of the beacon point Ec_l^0 by increasing its $CMaxIrH$ value. The amount by which $CMaxIrH$ is increased is determined in a very similar manner as the shrinking case discussed above. Thus in this case, Ec_l^0 acquires additional load from the beacon point Ec_{l+1}^0. In this manner the sub-range assignments of all the beacon points in the beacon ring are updated. Some beacon points might find it costly to maintain the $CIrHLd$ information for each of the hash value within its sub-range, in which case the $CIrHLd_l^0(p)$ for all hash values in the sub-range of the beacon point Ec_l^0 are approximated by averaging $CAvgLd_l^0$ over its sub-range of IrH values.

After determining the sub-range assignments for the next cycle, all the caches in the cache ring and the origin server are informed about the new sub-range assignments. Beacon points that have been assigned new IrH values obtain lookup records of the documents belonging to the new IrH values from their current beacon points.

The origin server and the caches within a cloud can determine the beacon point of any document by a simple and efficient two-step process. Suppose a cache cloud has K beacon rings, and each beacon rings has M beacon points. In the first step, the beacon ring of the document is determined by the random hash function. For example, if $j = MD5(URL(Dc))$ Mod K, then the ring-j would be the beacon ring of Dc. In the second step, out of the M

beacon points within the jth beacon ring, the beacon point of Dc is determined through the intra-ring hash function as we discussed before. The beacon point whose current sub-range contains $IrH(Dc)$ would be the beacon point of Dc.

In our scheme, the document lookup and update protocols work as follows. When a cache needs to locate a document Dc, it determines the document's primary beacon point using the two-step process described above. Then it contacts the document's beacon point and obtains the list of caches that hold the cached copies of the document within the given cache cloud. When the server needs to communicate an update to the document Dc, it uses the two-step process and determines the document's beacon point for each cache cloud. It sends a document update message to these beacon points (one for each cloud), which in turn communicate it to the caches in their cache clouds, which are currently holding the document.

4.2.1 Providing Resilience to Beacon Point Failures

As we explained previously, the lookup and the update information available at a document's primary beacon point is replicated at the other beacon points in the document's beacon ring for providing resilience to beacon-point failures. In this section we explain two different strategies for replication of the beacon information, namely, eager replication scheme and periodic replication scheme. In this discussion we refer to the lookup information that is maintained at a document's primary beacon point as the master-copy of document's beacon information, and the copies present at the other beacon points as backup-replicas.

In the eager replication scheme the backup-replicas of the beacon information are always consistent with the information at the primary beacon point. This means that any changes to the lookup and update information of a document are immediately reflected at all the backup replicas. For example, when a cache within the cloud stores an additional copy of a document, or evicts an existing document, all the replicas of the document's lookup information (available at the beacon points in the document's beacon ring) are updated immediately to reflect the change. When a beacon point fails, one of the other beacon points within beacon ring assumes the lookup and update responsibilities of all the documents that

were being handled by the failed beacon point. As all the replicas of the lookup information are consistent, the backup node has complete and up-to-date information to accurately handle the lookups and updates of all the documents that were being handled by the failed beacon point. When a failed beacon point is reactivated, it obtains the up-to-date lookup information from any of the other beacon points that are currently alive, and then resumes its normal operation.

In contrast to the eager replication scheme, the periodic replication does not enforce strict consistency between the lookup information at the primary beacon point and its backup copies. In other words, in the periodic replication scheme, there are brief durations of time in which the backup copies of the lookup information may be slightly out of sync when compared with the master lookup information available on the primary beacon point. Periodic replication derives its name from the fact that in this scheme the backup replicas of the lookup information are made consistent with the master copy available at the primary beacon point periodically. For example, the backup replicas may be synchronized with the master-copy at the end of each load balancing cycle.

The main observation that motivates the periodic replication is that the lookup information of a document is a *soft state*, which can be recovered even when the primary beacon point fails and no other available replica is completely up-to-date. We now briefly outline the periodic replication mechanism. As we mentioned above, in the periodic replication the backup replicas of the lookup information are synchronized with the master copy that is available at the primary beacon point at the end of each pre-defined time period called the *beacon synchronization cycle*. Within a beacon synchronization cycle any updates to the lookup information of a document (such as caches storing additional copies or evicting existing copies of the document) occurs only at the document's primary beacon point if it is alive, or at the backup node that is currently handling it lookup and update operations.

Let us now consider what happens when a document's primary beacon point fails. Suppose the cache Ec_l^j is the primary beacon point of an arbitrary document Dc. As in the eager replication case, if Ec_l^j fails, one of the backup replicas take over the lookups and update operations of all the documents (including Dc) that were being handled by the

failed cache. Without loss of generality, let us assume that the cache Ec_{l+1}^j takes over the lookup and update operations from Ec_l^j. However, it should be noted that due to periodic replication, Ec_{l+1}^j may not have the most up-to-date lookup information of the documents which it takes over from the failed cache Ec_l^j. Hence the cache Ec_{l+1}^j marks the lookup information of these documents as *out-of-sync*.

We will now explain how the update and lookup operations of Dc are handled in the periodic replication when Dc's lookup information is marked as out-of-sync. First, let us consider the update operation. On detecting the failure of Dc's primary beacon point (Ec_l^j), the origin server sends the update message to the backup node (Ec_{l+1}^j). The cache Ec_{l+1}^j in turn sends the update message to all the caches in the cloud. However, this is a special update message, wherein the cache Ec_{l+1}^j notifies that the lookup information regarding the document is out-of-sync, and asks all the caches that hold the document to re-register with it. The caches within the cloud that contains the document re-register with Ec_{l+1}^j. Thus, Ec_{l+1}^j recovers the up-to-date lookup information of the document Dc, and hence removes the out-of-sync tag from Dc's lookup information.

Similarly, a cache that needs to retrieve Dc, on noticing that its primary beacon point has failed contacts Ec_{l+1}^j. Now there are two cases to consider. In the first case the lookup information of Dc has been recovered through a previous update operation on Dc. In this scenario, Ec_{l+1}^j just forwards the list of caches that currently hold a copy of the document. In the second case the lookup information of Dc is marked as out-of-sync. In this scenario, Ec_{l+1}^j sends the lookup information it currently holds to the requesting cache, but it explicitly states that the lookup information may not be up-to-date. In this case, there is a small chance that the lookup information is slightly stale (the lookup information might indicate that Dc is available at a cache, when it is actually not present, and vice-versa). Our experiments show that the probability of a cache receiving stale lookup information is very small. The requesting cache then decides whether to contact one of the caches that is supposed to contain the document as per the lookup information it received, or to contact the origin server directly.

When the primary beacon point of Dc (Ec_l^j) is reactivated it obtains the current lookup

information of all the documents that are assigned to it from (Ec_{l+1}^j) and resumes the lookup and update functionalities and notifies the origin server and the other caches that it has recovered from the failure. In the rare event of all the caches in a beacon ring failing simultaneously, the lookups and updates of the documents are handled by sending the appropriate messages to all the caches in the cloud.

We now briefly discuss the relative pros and cons of the eager and the periodic replication schemes. The eager replication scheme maintains all the replicas of the lookup information consistent and up-to-date. Hence, a cache trying to retrieve a document always receives up-to-date lookup information. Further, with eager replication there is no need to flood the network with lookup and update messages unless all the caches belonging to a beacon ring have failed simultaneously. However, the disadvantage of the eager replication scheme is that it entails significant message costs. Whenever a cache in the cloud stores an additional copy of a document or evicts an existing copy, all the beacon points in the corresponding beacon ring have to be informed. On the other hand, periodic replication requires a single message each time a cache in the cloud stores or removes a document copy. Therefore, the message costs of the periodic replication are considerably lower when compared with that of the eager replication scheme. But, with periodic replication there is a small chance that a cache trying to retrieve a document might receive lookup information that is slightly stale. Our experiments show that the probability of a cache receiving stale beacon information is very low. Therefore, we advocate the periodic replication, unless the rate of beacon point failure is very high.

4.3 Utility-based Document Placement in Cooperative EC Grid

In this section we consider the third aspect of cooperation in the EC grid, which is designing effective cache management policies for caches in the EC grid. We are interested in designing cache management policies that take into account the benefits and costs of caching documents, and are sensitive to the on-going cooperation among the caches belonging to a cache cloud. One such cache management scheme is the document placement in cache

clouds.

First, we will briefly describe the document placement problem. Let us consider a cache cloud with N caches. Suppose the cache Ec_l^j receives a client request for document Dc, which is not available locally (i.e., the request is a local miss). The document may be available in one or more caches within the cloud, or it might have to be retrieved from the origin server. Suppose, the document is available in the cloud, then should Ec_l^j make a local copy of the document Dc. On the contrary, if the document is not available at any of the caches in the cloud, should it be stored in one of the caches in the cloud, and if so, where (in which cache) should it be stored? Another cache management policy that bears resemblance to document placement is the document replacement policy. The problem here is to decide which documents that are currently in the cache should be evicted in order to create disk-space for an incoming document. We discuss more about the relationship between these two mechanisms at the end of this section.

In this thesis, we have addressed the document placement problem by proposing a document placement technique, called the *utility-based* document placement scheme. This scheme takes into account the cooperation among the caches belonging to a cloud. Further, as the name suggests, the caching decisions in our scheme rely upon the utility of a document-copy to the cache storing it, and to the entire cache cloud. This value is termed as the *utility value* of the document copy, and is represented as $Utility(Dc)$ for document copy Dc. When a cache retrieves a document it calculates the document's utility value. Based on this utility value the cache decides whether or not to store the document.

The utility of document copy Dc estimates the benefit-to-cost ratio of storing and maintaining the new copy. A higher value of utility indicates that benefits outweigh the costs, and vice-versa. The costs of caching dynamic documents are two fold, namely, the consistency maintenance costs and the storage costs. Consistency maintenance costs correspond to the overheads involved in maintaining the freshness of the cached copy of a document. Storage costs correspond to physical storage space needed for storing the document copy. On the other hand caching a document copy benefits the edge network by reducing the server load, network load and the client latency. Our formulation of the utility function has

four components. Each of these components quantifies one aspect of the interplay between benefits and the costs. We now briefly discuss each of these components. Throughout this discussion we assume that a cache Ec_l^j has retrieved the document copy Dc, and is calculating its utility value to decide whether to store it locally.

Document Availability Component

Represented $DAIC(Dc, Ec_l^j)$, this component quantifies the improvement in the availability of the document in the cache cloud achieved by storing the document copy at Ec_l^j. Improving the availability of a document increases the probability that a future request for the document would be served within the cache cloud. Let $Rblty(Ec_p^q)$ denote the reliability of the cache Ec_p^q. The current document availability of document Dc is computed as $CAvblty(Dc) = \sum_{Ec_p^q \text{contains} Dc} Rblty(Ec_p^q)$. We note that $CAvblty(Dc)$ becomes 0 when Dc is not currently stored at any cache in the cache cloud. If an additional copy of the document is stored at the cache Ec_l^j, it improves the document availability by $Rblty(Ec_l^j)$. Document availability improvement component is now defined as:

$$DAIC(Dc, Ec_l^j) = \begin{cases} MaxValue & \text{If CAvblty(Dc)} = 0 \\ \frac{Rblty(Ec_l^j)}{CAvblty(Dc)} & \text{Otherwise} \end{cases}$$

Here $MaxValue$ denotes a large positive real number.

Disk-Space Contention Component

This component captures the storage costs of caching the document copy at Ec_l^j in terms of the disk-space contention at Ec_l^j. The disk-space contention at the cache Ec_l^j determines the time duration for which the document can be expected to reside in the cache Ec_l^j before it is replaced. Suppose the cache cloud already contains a copy of the document at cache Ec_p^q. If the disk space contention at that cache is lower than that of cache Ec_l^j, then even if we were to make a new copy at the cache Ec_l^j, the new copy would removed earlier than the existing copy. Hence, the benefits of storing the new copy are limited, whereas it adds to the disk-space contention at Ec_l^j. On the other hand if the disk-space contention at Ec_l^j is significantly lower than that of Ec_p^q, the new copy is expected to remain in the cache much

longer than the copy at Ec_p^q benefiting the cache cloud for a longer duration. We use the concept of *cache expiration age* [88] to accurately quantify the disk space contention at edge caches. Higher the cache expiration age of a cache, the lower is its disk space contention, and vice versa.

If the cache Ec_l^j retrieves the document Dc from another cache in the cache cloud, say Ec_p^q, then the disk-space component for document Dc is defined as the ratio of the expiration age of Ec_l^j to the expiration age of Ec_p^q. However, if the document is not available in the cache cloud, the disk-space component is assigned a large positive value, since it is always advantageous to store a copy of Dc.

$$DsCC(Dc, Ec_l^j) = \begin{cases} \frac{CacheExpAge(Ec_l^j)}{CacheExpAge(Ec_p^q)} & \text{If } Dc \text{ is retrieved from} Ec_p^q \\ \\ MaxValue & \text{If } Dc \text{ is retrieved from server} \end{cases}$$

A higher value of $DsCC(Dc, Ec_l^j)$ implies that the new document copy is likely to remain in the cache cloud for a duration than the old one, and it is beneficial to store this copy.

Consistency Maintenance Component

Denoted by $CMC(Dc, Ec_l^j)$ this component accounts for the costs incurred for maintaining the consistency of the new document copy at Ec_l^j, and the advantages that are obtained as a result of storing Dc at Ec_l^j by avoiding the cost of retrieving the document from other caches on each access. Suppose we observe the access and the update patterns of the document Dc at cache Ec_l^j for t_w time units. Let there be $AccCount$ accesses during this time period. An access A_v is termed as an *updated-access* if the document Dc is updated between the accesses A_{v-1} and A_v, and as *nonupdated-access* otherwise. Let $NonUpCount$ represent the number of nonupdated-accesses within the time duration t_w. The consistency maintenance component is obtained as:

$$CMC(Dc, Ec_l^j) = \frac{NonUpCount}{AccCount}$$

A high value of $CMC(Dc)$ indicates that the document Dc is accessed more frequently than it is updated, and vice-versa

Access Frequency Component

The final component of our utility function quantifies how frequently the document Dc is accessed in comparison to other documents in the cache. If access frequency of Dc at the cache Ec_l^j is high when compared to other documents in the cache, it is advantageous to store it. Let $ReqCount(Dc, Ec_l^j, T_z)$ indicate the number of requests to the document Dc at cache Ec_l^j in the past T_z time units, $TotReqs(Ec_l^j, T_z)$ denote the total number of client requests received at cache Ec_l^j in the past T_z time units, and let $NumDocs(Ec_l^j)$ represent the total number of documents currently cached at Ec_l^j. Therefore, the mean of the number of requests received by the documents in the cache is given by $MnReqs(Ec_l^j, T_z) = \frac{TotReqs(Ec_l^j, T_z)}{NumDocs(Ec_l^j)}$. The access frequency component of the utility function is computed as below.

$$AFC(Dc, Ec_l^j) = \frac{ReqCount(Dc, Ec_l^j, T_z)}{MnReqs(Ec_l^j, T_z)}$$

The Utility Function

The above four components form the building blocks of the utility function. We observe that for each component, a higher value implies that benefits of storing Dc are higher than the overheads, and vice-versa. We define the Utility of storing the document Dc at cache Ec_l^j, denoted as $Utility(Dc, Ec_l^j)$, to be a weighted linear sum of the above four components.

$$Utility(Dc, Ec_l^j) = W_{DAIC} \times DAIC(Dc, Ec_l^j) + W_{DsCC} \times DsCC(Dc, Ec_l^j)$$
$$+ W_{CMC} \times CMC(Dc, Ec_l^j) + W_{AFC} \times AFC(Dc, Ec_l^j)$$

In the above equation W_{DAIC}, W_{DsCC}, W_{CMC}, and W_{AFC} are positive real constants such that $W_{DAIC} + W_{DsCC} + W_{CMC} + W_{AFC} = 1$. These constants are assigned values reflecting the relative importance of the corresponding component of the utility function to the performance of the system. For example, if the documents in the system have high update rates, then W_{CMC} is assigned a high value. Similarly if disk-space availabilities at the caches are limited W_{CMC} would be set to a high value.

If the value of the utility function $Utility(Dc, Ec_l^j)$ exceeds a threshold, represented as $UtlThreshold(Ec_l^j)$, then Dc is stored at cache Ec_l^j. Otherwise the edge cache Ec_l^j does not store a local copy of Dc. Concretely, suppose a client-request for document Dc encounters a local miss at cache Ec_l^j. Then Ec_l^j retrieves the document either from another cache in the cache cloud, or from the origin server. If the document Dc does not currently exist in the cache cloud, Ec_l^j stores the document Dc and registers it with the beacon point of Dc. This is because, in this scenario the $DAIC$ and the $DsCC$ components of its utility function assume very high values. If at least one cache in the cache cloud contains Dc, then Ec_l^j does not decide immediately whether to store the document. Instead, it monitors the pattern of requests and updates to the document for fixed time duration in order to evaluate its utility value. At the end of this time duration, the cache evaluates the utility function $Utility(Dc, Ec_l^j)$. The cache stores the document Dc, only if $Utility(Dc, Ec_l^j) \geq UtlThreshold$, in which case it registers the new copy with its beacon point. The pseudo-code of the utility-based document placement is outlined in Algorithm 1.

Algorithm 1 Utility-based Document Scheme: Algorithm performed by a cache on receiving a request for document Dc

---

if Dc is available locally **then**
  Serve request and update document statistics
else
  Contact Dc's beacon point and obtain lookup information
  if Dc is not available within cache cloud **then**
    Obtain Dc from the origin server
    Store Dc locally
    Register the new copy with beacon point
  else
    Obtain Dc from one of the caches currently holding Dc
    Along with Dc obtain the responding cache's Expiration Age
    if A $DocumentRecord$ for Dc exists in $DocumentMonitorList$ **then**
      Update the $DocumentRecord$
      if $DocumentRecord$ of Dc has resided in the $DocumentMonitorList$ for more than $MonitorDurationThreshold$ **then**
        Remove $DocumentRecord$ of Dc
        Compute the utility function $Utility(Dc, Ec_l^j)$
        if $Utility(Dc, Ec_l^j) \geq UtlThreshold(Ec_l^j)$ **then**
          Store Dc locally
          Register the new copy with beacon point
        else
          Discard the $DocumentRecord$ of Dc
        end if
      end if
    else
      Create a $DocumentRecord$ for Dc and store it in $DocumentMonitorList$
    end if
  end if
end if

We note that the utility function accurately quantifies the costs and benefits of caching dynamic documents, and it can also be adopted as the cost function in cost-based document replacement policies such as the Greedy-dual size [29], Greedy-dual* [21] algorithms.

4.3.1 Comparing Document Placement and Document Replacement

In this section we compare and contrast two categories of cache management schemes, namely, the document placement policies and document replacement policies. Before discussing the similarities and the differences between the two, we will briefly describe the document placement and document replacement problems.

As we explained in the beginning of Section 4.3, the problem of document placement is to decide whether a document Dc that has been retrieved by a cache Ec_l, should be stored at the cache Ec_l, or whether it should be stored at a different cache in the cloud, or should just be discarded after serving the user request. In contrast the document replacement problem can be summarized as follows: Suppose the cache Ec_l wants to store a document Dc locally, but its storage-space is already full. Now the cache has to evict some of the documents that are currently stored by it, so that the incoming document can be stored. In this scenario, deciding which documents to evict is known as the document replacement problem. While researchers have proposed many techniques to address the document replacement problem [21, 29, 38, 80], the studies on the document placement problem are relatively few in number [61, 88, 102].

Both document placement and document replacement problems are concerned with managing the available resources such as disk-space and network bandwidth effectively. The goal of both problems is to maximize the benefits of caching. The document placement approach may be considered as a proactive approach to resource management, whereas document replacement can be regarded as a reactive approach, since it is triggered when the available resources such as disk-space become insufficient. We believe that caches need to implement a good document placement as well as a good document replacement scheme in order to effectively manage their resources.

In the context of cooperative EC grid, incorporating proactive approach through good

document placement schemes provides some unique benefits. First, utilizing resources judiciously even when the resource consumption at individual caches have not reached their respective limits is likely to benefit the performance of the cooperative cache group. For example, suppose a cache Ec_l retrieves a document Dc which is being modified very frequently. If the cache stores this document, Dc's beacon point would have to communicate Dc's updates to Ec_l until it is replaced, which would place significant load on the beacon point and on the network. This cost could have been avoided had the cache Ec_l made the placement decision judiciously. Similarly, suppose the cache Ec_l retrieves a document Dc_1, which is available at many other caches in the cloud. Then making an additional copy at Ec_l blindly, could result in the replacement of one or more documents, which may not available in any other caches within the cache cloud. This affects the cumulative hit rate of the cache cloud.

The second advantage of document placement scheme is the ease and efficiency of its implementation in a cooperative cache group setting. For cooperative cache groups, good placement and replacement schemes have to take into account the availability of documents in various caches of the cache group. In cooperative EC grid, this information is available at the beacon point. When a cache contacts the beacon point in order to retrieve a document, the document's beacon point can also send this information along with lookup data. Therefore, the cache can obtain all the information needed to make the placement decisions at no extra-cost. In contrast, if a cache has to implement a cooperative replacement scheme, it has to obtain the availability information for a subset of documents that would be possible candidates for eviction. These documents may have been assigned to different beacon points. Hence, for each document replacement decision, the cache might have to contact various beacon points, which makes the cooperative replacement strategy expensive to implement. Considering these factors, we have investigated the document placement problem, and we have proposed the utility-based document placement scheme.

4.4 Experiments and Results

In this section we discuss the experiments we performed to evaluate our cache cloud design architecture. We have evaluated the proposed schemes through trace-based simulations of an edge cache network. The simulator can be configured to simulate different caching architectures such as edge network without cooperation, cooperative caching with static hashing, and cooperative cache clouds with dynamic hashing. Further, it can also simulate different failure resilience mechanisms, namely, no failure resilience, failure resilience with eager replication, and failure resilience with periodic replication.

Each cache in the cache cloud receives requests continuously according to a request-trace file. If the requested document is available within the cache it is recorded as a local hit. Otherwise, if the document is available within the cache cloud, the request is a group hit. If the document is not available in any of the caches in the cache cloud, then the request is a miss. In our simulation the server continuously reads from an update trace file. Upon reading an update entry for a document Dc, the server sends the updated version of Dc to its beacon points within each cache cloud, which is then distributed by the beacon points to all the caches in their cache clouds, which are currently holding the document. The document request rates, the document update rate, the number of caches and the number of beacon rings in the cache cloud are system parameters, and can be varied.

We have used two datasets for our experiments. The first dataset, which we refer to as the Sydney dataset is a real trace from a major IBM sporting and event web site [1]. Detailed description of this trace is given in Chapter 2. The second dataset is a synthetic trace, wherein both access and follow the Zipf distribution with the Zipf parameter value set to 0.9. This workload, called as the Zipf-0.9 dataset, contains request and update entries corresponding to 25,000 unique documents. Each edge cache is configured to have 10% of the total disk-space needed to store all the documents in the dataset. The standard LRU algorithm is used for document replacements at all caches. All the measurements reported in this thesis were taken when the system reached its steady state.

---

[1] The 2000 Sydney Olympic Games web site

4.4.1 Evaluating the Effectiveness of Beacon Rings

In the first set of experiments we study the load balancing properties of the dynamic hashing mechanism. All the beacon points within the cache cloud are assumed to be of equal capabilities, which implies that perfect load balancing is achieved when all the beacon points encounters same amount of load. In all the experiments in this set, the intra-ring hash generators ($IntraGen$s) are set to 999 for all beacon rings in the cache cloud and the cycle length of sub-range determination is set to 1 hour. We use the coefficient of variation of the loads on the beacon points to quantify load balancing. Coefficient of variation is defined as the ratio of the standard deviation of the load distribution to the mean load. The lower the coefficient of variation, better is the load balancing.
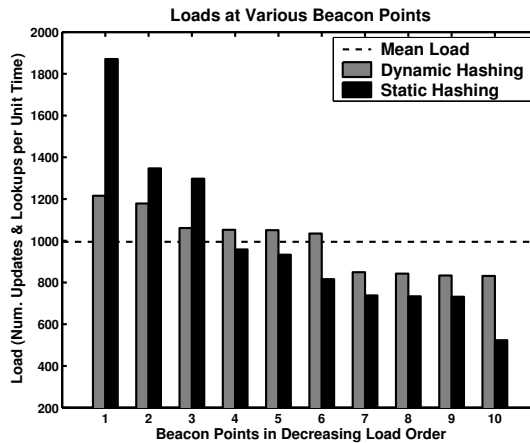
Figure 17: Load Distribution among Beacon Points for Zipf-0.9 Data Set

First, we compare the load balancing accomplished by the static and the dynamic hashing schemes in a cache cloud with 10 caches. For the dynamic hashing scheme the cache cloud is configured to contain 5 beacon rings, each with 2 beacon points. The bar-graphs in Figure 17 and Figure 18 show the load distribution among the beacon points for the static and the dynamic hashing schemes on the Zipf-0.9 synthetic data set and the Sydney datasets respectively. On the X-axes of these graphs are the beacon points in the decreasing order of their loads, and on the Y-axes are the loads in terms of the number of updates and lookups being handled by the beacon points per unit time. The dashed-lines in the

two graphs indicate the mean value of the loads on the beacon points. The Zipf-0.9 dataset induces a high degree of load imbalance in the cache cloud with static hashing. In this case, the load on the most heavily loaded beacon point is 1.9 times the mean load of the cache cloud. In the dynamic hashing scheme this ratio decreases to 1.2, thus providing 37% improvement over static hashing scheme. The dynamic hashing scheme also provides a 63% improvement on the coefficient of variation when compared with static hashing. On the Sydney data set, the dynamic hashing scheme improves the ratio of the heaviest load to the mean load by around 20%, and the coefficient of variation by 63%. For this dataset, the ratio of heaviest load to mean load for the dynamic hashing scheme is just 1.06, thus showing that this scheme achieves very good load balancing.
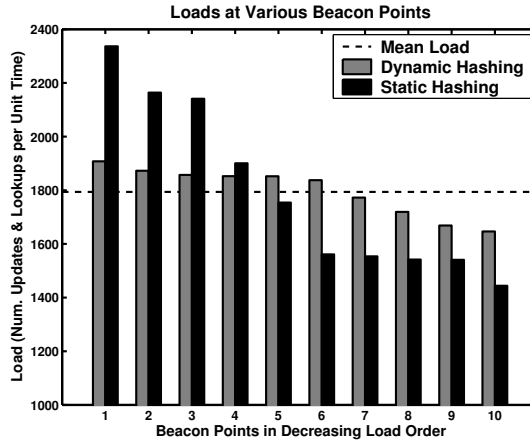
Figure 18: Load Distribution among Beacon Points for Sydney Data Set

The next experiment studies the effect of the size of the beacon rings on the load balancing. We evaluate the dynamic hashing scheme on cache clouds consisting of 10, 20 and 50 caches. For each cache cloud we consider three configurations in which each beacon ring contains 2, 5 and 10 beacon points. Figure 19 indicates the results of the experiment on the Olympics dataset. The dynamic hashing scheme with 2 beacon points per ring provides significantly better load balancing in comparison to static hashing. When the size of the beacon rings is further increased we observe an incremental improvement in the load balancing achieved by the dynamic hashing scheme. The observation that bigger beacon

rings yield better load balancing can be explained as follows: The beacon point sub-range determination process tries to balance the load only among the beacon points within each beacon ring. Larger beacon rings result in the load being balanced among larger number of beacon points, and hence provide better load balancing.
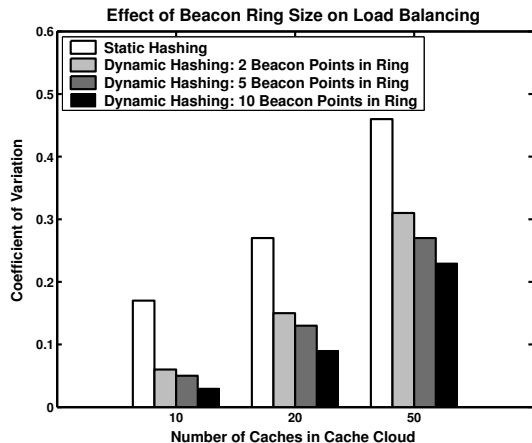
Figure 19: Impact of Beacon Ring Size on Load Balancing

In the third experiment, we study the impact of the dataset characteristics on the static and the dynamic hashing schemes. For this experiment we consider several datasets all of which follow the Zipf distribution with Zipf parameters ranging from 0.0 to 0.99. The skewness of the load increases with increasing value of the Zipf parameter. Figure 20 shows the coefficient of variation values at different Zipf parameters. At low Zipf values both schemes yield low coefficient of variation values. As the skewness in the load increases the coefficient of variation values also increase for both schemes. However, the increase is more rapid for the static hashing scheme. At Zipf parameter value of 0.9, the coefficient of variation for the static hashing scheme is 0.65, whereas it is 0.44 for the dynamic hashing scheme.

4.4.2 Evaluating Failure Resilience Properties

One of the motivating reasons for organizing the caches in a cloud into beacon rings was to provide resilience to failure of individual beacon points. In this set of experiments we evaluate the performance of the two approaches for providing resilience to beacon point

Figure 20: Impact of Zipf parameter on load balancing

failures, namely, the eager and the periodic replication mechanisms.

Similar to the previous set of experiments, we consider a cache cloud in which the caches are organized into beacon rings. In this set of experiments the individual beacon points fail at arbitrar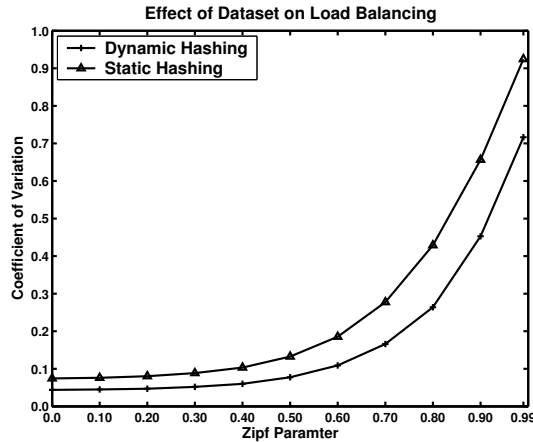y points of time. We consider a simple, yet commonly occurring, failure mode where in the beacon points stop responding to all document lookup requests from other peer caches, and to the update messages from the server. There are other failure modes such as beacon points, maliciously or otherwise, not responding to particular caches or server, or not responding to particular type of requests, or sending out spurious update messages. Providing resilience to these complex types of failures is beyond the scope of current work.

The failure pattern is modeled as a *Weibull* process. Weibull process is commonly used by researchers in reliability engineering to model failure patterns of systems and system components. The Weibull process is characterized by the probability distribution function $f(t) = \kappa \lambda^\kappa t^{\kappa-1} e^{-(\lambda t)^\kappa}$, where λ is called the scale parameter and the κ is called the shape parameter. The exponential process is a special case of the Weibull process, wherein $\kappa = 1$. When κ assumes values greater than 1.00, we obtain a class of distributions having *increasing failure rates (IFR)*. In IFR distributions, the probability of a component failing during an arbitrary time period $(T_0 + T_d)$ given that the component has not failed until the time instant T_0, grows with increasing values of T_0. Analogously, the failure of a

component is said to follow *decreasing failure rate (DFR)* process if the probability of its failure during the arbitrary time period $(T_0 + T_d)$ given that it has not failed until time T_0 decreases with increasing values of T_0. When κ is set to values that are lesser than 1, the Weibull process yields a class of distributions that demonstrate this behavior. It is widely believed that the lifetime distributions of most practical systems/components demonstrate IFR characteristics. Accordingly, in our experiments κ is set to various values all of which are greater than 1.00. In our context, IFR lifetime distributions imply that a beacon point that has not failed in the recent past has a higher probability of failure than a beacon point that has failed and has been restored recently. With the Weibull failure model the mean lifetime of a beacon point is given by the formula $\mu = \frac{1}{\lambda}\mathbf{\Gamma}(1 + \kappa^{-1})$, and its standard deviation is provided by $\sigma^2 = \frac{1}{\lambda^2}[\mathbf{\Gamma}(1 + 2\kappa^{-1}) - \mathbf{\Gamma}^2(1 + \kappa^{-1})]$. Here $\mathbf{\Gamma}(z)$ represents the Gamma function.

In our experiments the beacon are active for relatively long stretches of time. Comparatively, the time duration for which a failed beacon point remains non-functional is very short, at the end of which it is reactivated, and begins the recovery process.

In our experiments the simulation prototype can be configured to support failure resilience through eager replication or periodic replication. It can also be configured to simulate the simple hashing scheme wherein cache clouds do not provide any resilience to failures of individual beacon points. In the case that the cache clouds do not provide failure resilience, the lookups and updates of those documents that were assigned to the failed beacon points are handled by flooding the cloud with the respective messages. We measure the performance of the three schemes through the average number of messages circulated in the cloud per unit time.

Figure 21 compares the dynamic hashing scheme (with failure resilience support through eager or periodic replication) and the simple hashing scheme (with no failure resilience support) with respect to their beacon information availability. In this experiment each beacon ring is configured to have 2 beacon points. The X-axis of the graph indicates the failure rate of the individual beacon points and the Y-axis shows the percentage of lookup and update requests for which the beacon information was not available. It can be seen
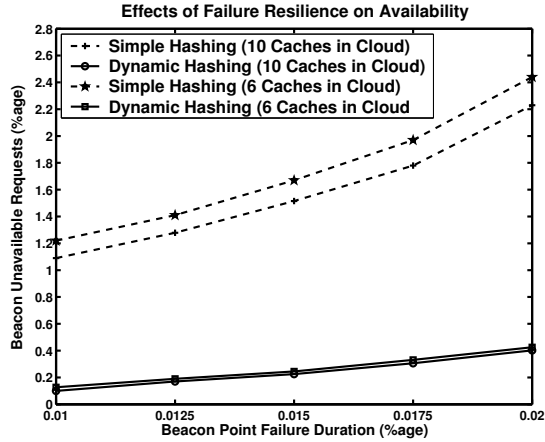
Figure 21: Improvement in Beacon Information Availability

from the graph that the dynamic hashing scheme with failure resilience support provides an order of magnitude improvement in beacon information availability over the static hashing scheme with no failure resilience. For example, when the failure rate is set to 0.0125 the percentage of requests encountering unavailable beacon information is around 1.28% for the simple hashing scheme, whereas it is 0.17% for the dynamic hashing scheme with failure resilience support. The observed results can be explained as follows: In simple hashing scheme, when a beacon point fails, the lookups and updates of all the documents mapped to it encounter unavailable beacon information, whereas in the dynamic hashing scheme a lookup or an update request for a document encounters the unavailable beacon information condition only when all the beacon points belonging to the document's beacon ring have failed.

Next, we study the performance of the various failure resilience schemes with respect to the message loads induced by them within the cache cloud. In Figure 22 we plot the number of lookup/update messages circulated per unit time with each of the three schemes in a cache cloud consisting of 5 beacon rings, and each beacon ring containing 2 beacon points (the cloud contains 10 caches in total). The X-axis of the graph indicates the failure rate of individual beacon points, and the Y-axis shows the number of messages circulated in unit time.
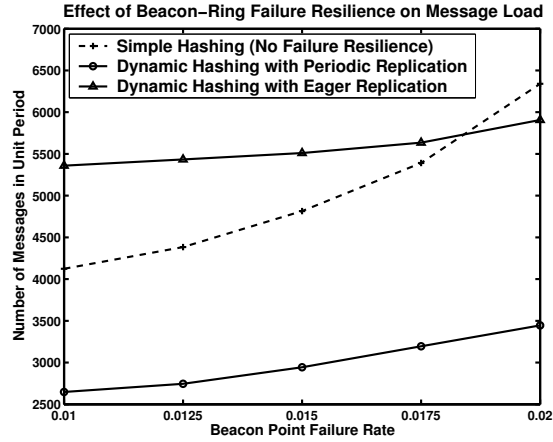
Figure 22: Network Load with Various Failure Resilience Schemes (10 Caches Per Cloud)

A few interesting observations emerge from the graph in Figure 22. First, as we mentioned in Section 4.2.1, eager replication scheme is costly in terms of the message cost. The higher message cost of the eager replication is caused by the necessity to reflect any changes in the lookup information of a document on all its backup replicas immediately. In fact, when the failure rates of the beacon points are low, the no failure resilience scheme (in which the lookups and updates of the documents that were assigned to a failed beacon point are handled through flooding) better than the eager replication scheme. However, as the failure rate increases, the number of message in the no resilience scheme grows at a fast rate, and it overtakes the message cost of the eager replication scheme when the failure rate is around 0.018. This phenomenon is due to the fact that as the failure rate increases, the frequency of cloud-wide flooding in the no resilience scheme raises sharply.

The periodic replication scheme performs better than both the no failure resilience approach and the eager replication scheme at all values of failure rates. For example, when the failure rate is 0.010, the message cost of periodic replication scheme is around 36% less than that of the no resilience approach, and it is 51% less than the eager replication scheme.

Figures 23 shows the performance of the three schemes on cache clouds having 6 caches. From these results, we observe that the relative performance of the three schemes is similar to that of the cache cloud with 10 caches, albeit on different scales.
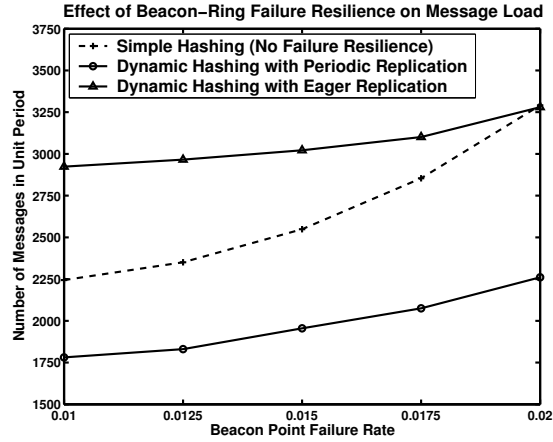
Figure 23: Network Load with Various Failure Resilience Schemes (10 Caches Per Cloud)

Although the periodic replication scheme provides significant performance benefits, it suffers from the drawback that a cache trying to retrieve a document might receive stale lookup information. In Section 4.2.1, we had mentioned that this would not be a severe drawback as the probability of its occurrence is very low. In the next set of experiments we show the validity of this argument by measuring the percentage of lookup requests that receive stale information.

In Figure 24 and Figure 25, we plot the percentage of lookup request receiving stale beacon information at various values of the synchronization cycle duration for a cache cloud with 10 caches and 6 caches respectively. The duration of the synchronization cycle is likely to have considerable impact on the percentage of lookups receiving stale beacon information, because it determines how frequently the backup copies of the lookup information are made consistent with the master copy.

We observe that the percentages of lookups receiving stale information are very small at both cache cloud sizes. For a cache cloud with 10 caches, when the failure rate is 0.010, the percentage of stale lookups ranges between 0.03% and 0.08% as the duration of the synchronization duration varies from 10 time units to 500 time units. For the same cache cloud the percentage of lookups receiving stale beacon information range from 0.033% to 0.095%, when the failure rate is 0.02%. Hence, we see that even the duration of the
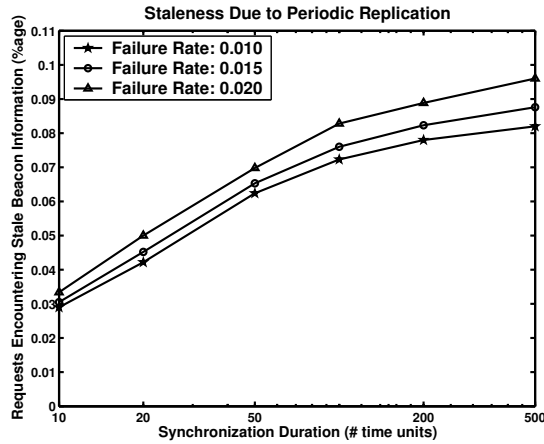
Figure 24: Staleness of Beacon Information in Periodic Replication (10 Caches in Cloud)

synchronization cycle is significantly long, the percentage of stale lookups is very low. It should also be noted that a stale lookup does not affect the correctness of the document retrieval protocol. In the worst case, a cache receiving stale lookup information might have to contact the origin server, although the document might already be available in the cache cloud. Considering the low message overhead and the very small probability of stale lookups, we conclude that periodic replication scheme is a very good choice for providing resilience to beacon point failures.
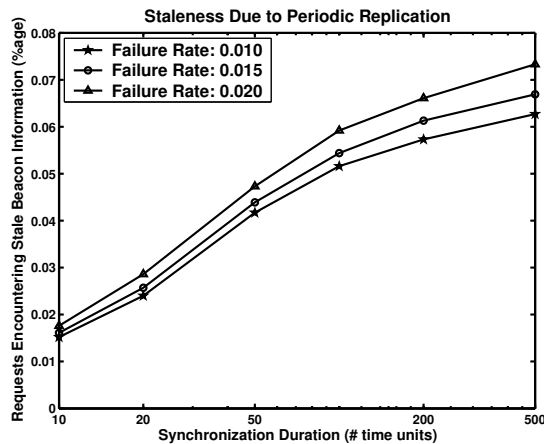
Figure 25: Staleness of Beacon Information in Periodic Replication (6 Caches in Cloud)

4.4.3 Evaluating Utility-based Placement Scheme

In this section we present the experimental evaluation of the utility-based document placement scheme. We study the performance of the proposed scheme by comparing it 2 other schemes. The first scheme is the ad-hoc document placement mechanism, wherein a cache stores every document for which it receives a client-request. The second scheme is the beacon-point placement. In this scheme, each document is only cached at its beacon point. Any other cache receiving a request from a client retrieves the document from its beacon point and serves the request.

In the first experiment in this set we consider a cache cloud comprising of 10 caches. On this cache cloud we simulate the various document placement policies. The caches in this experiment are assumed to have unlimited amount of disk-space. Therefore the disk-space component of the utility function is turned off by setting W_{DsCC} to 0. The weights of availability, consistency maintenance, and access frequency components are all set to 0.33. In this experiment the access rates at caches are fixed, whereas we vary the document update rate to study the effect of the five document placement policies.
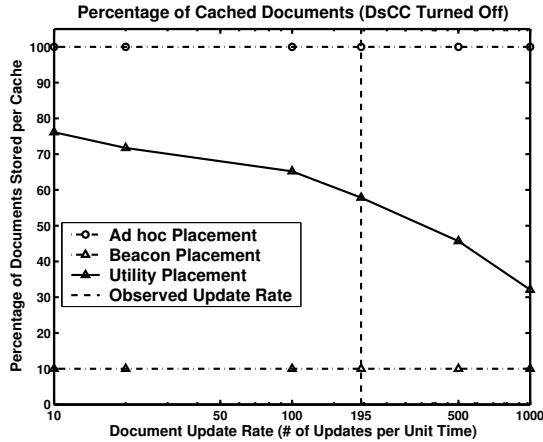
Figure 26: Percentage of Documents Stored (DsCC Turned Off)

The graph in Figure 26 shows the percentage of the total documents in the log that are stored at each cache in the cache cloud at various document update rates. The X-axis represents the document update rate in number of updates per minute on the log scale, and

the Y-axis represents the percentage of documents cached. The vertical broken line indicates the observed document update rate. As the ad hoc policy places each document at every cache which receives a request, almost all documents are stored at all caches. Similarly, the greedy-dual size algorithm also stores large numbers of documents in individual caches, since the disk-spaces in this experiment are unlimited. In contrast the beacon point placement stores each document only at its beacon point. Hence, each cache stores around 10% of the total documents. The percentage of documents stored per cache in the utility-based scheme varies with the update rate, indicating its sensitivity towards document update costs. In the utility-placement scheme, when the update rates are low, large percentage of documents are stored at each cache, owing to the small consistency maintenance cost. As the update rate increases the CMC values of all the documents decrease, leading to a decrease in the percentage of documents stored at each cache.
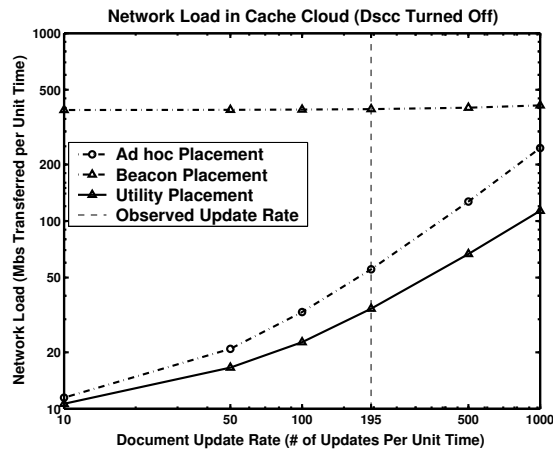
Figure 27: Network Load Under Various Placement Schemes (DsCC Turned Off)

The immediate question that arises is: what is the advantage of being sensitive to update costs in document placement. To answer this question we plot the total network traffic in the clouds generated by various document placement policies in the Figure 27. The results indicate that the utility-based document placement creates the least network traffic at all update-rates. The improvements provided by the utility-based placement scheme over the ad hoc placement scheme increase with increasing update rate. This is because,

while the number of replicas present in the cache cloud essentially remains a constant in the ad hoc placement scheme, utility-based scheme creates fewer replicas at higher update rates, thereby reducing the consistency maintenance costs significantly. The network traffic produced by the beacon point caching is very high at all update-rates, as in this scheme only one copy of each document is stored per cache cloud. Hence, most of the network traffic is due to caches repeatedly accessing the single copies of the documents.
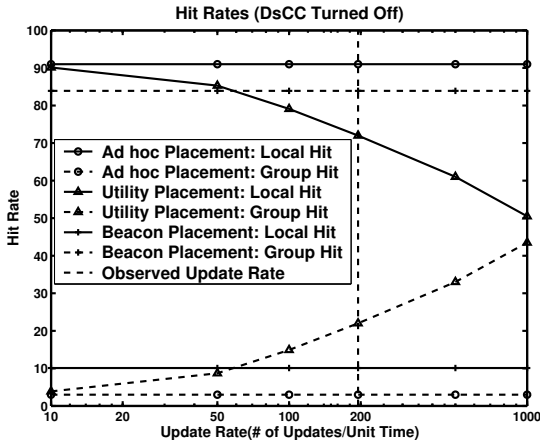
Figure 28: Hit Rates of Ad-hoc, Beacon Point and Utility Placement Schemes (DsCC Turned Off)

Figure 28 indicates the local and the group hit-rates of ad-hoc, beacon-point and utility-based placement schemes. The local hit rates and the group hit rates of ad hoc and beacon point placement schemes remain constant at all document update rates, since these schemes are not sensitive to document update costs. In contrast to these schemes the hit rate of the utility scheme varies with the document update rates. As the document update rate gets higher, the local hit rate of the utility scheme drops and its group hit increases.

In the second experiment, we study the performance of the three document placement policies when the disk spaces available at the edge caches are limited. In these experiments the disk space at each cache is set to 25% of the sum of sizes of all documents in the trace. As the disk space is a limiting factor in this series of experiments, we turn on the disk-space component of the utility function. The weights of all the utility function components are set to 0.25.
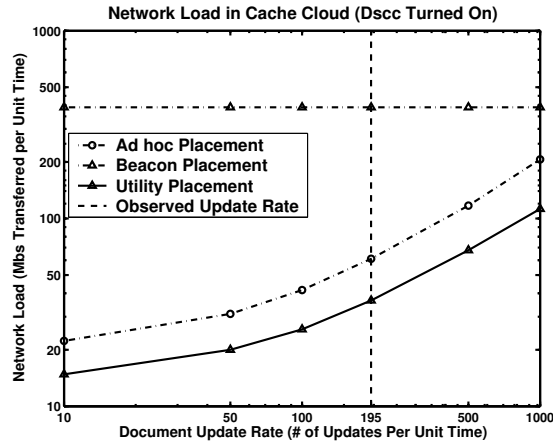
Figure 29: Network Load Under Various Placement Schemes (DsCC Turned On)

Figure 29 indicates the total network traffic generated by the three document placement policies at various update rates. As in the previous experiment, the utility-based document placement places the least load on the network. However, the results in this experiment differ from the previous experiment considerably. The percentage improvement in the network load provided by the utility scheme over the ad hoc scheme is higher in the limited disk-space case at low document update rates. However, the percentage improvement in the unlimited disk space case grows much faster in the limited disk space scenario. These observations are the manifestations of the different roles the utility placement scheme is playing at different update rates in the limited disk space scenario. At low document update rates the utility scheme assumes the predominant role of reducing disk space contention at individual caches. Whereas at higher update rates its predominant effect is to reduce consistency maintenance costs.

Figure 30 shows the hit rates of ad-hoc, beacon-point and utility-based placement schemes for the limited disk space experiment at different update rates. As in the unlimited disk space scenario, the local hit rate of the utility scheme drops as the update rate increases, whereas its group hit rate starts to increase. A crucial observation here is that the cumulative hit rate (the sum of local and group hit rates) of the utility scheme is around 5.5% higher than that of the ad hoc scheme. This shows that when disk pace becomes a
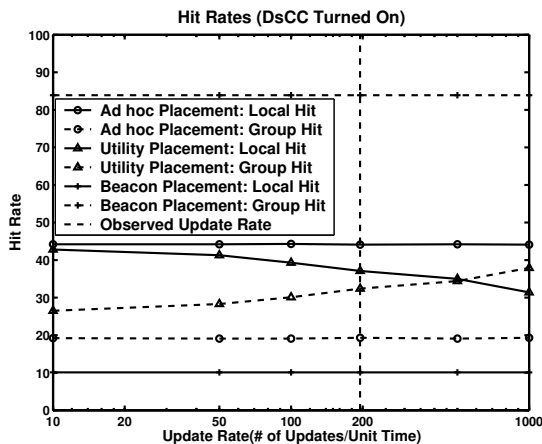
Figure 30: Hit Rates of Ad-hoc, Beacon Point and Utility Placement Schemes (DsCC Turned On)

limiting factor, utility scheme uses the aggregate disk space available in the cache cloud more efficiently.

The utility-based document scheme bears similarity to the cost-based document replacement schemes such as Greedy-dual size [29] and Greedy-dual* [21]algorithms. As an example let us consider the greedy-dual size algorithm, which is one of the most popular cost-based document replacement schemes. In this scheme each document is associated with a value, called its $HValue$. The $HValue$ of a document Dc is calculated as $HValue(Dc) = \frac{Cp(Dc)}{Size(Dc)}$, where $Cp(Dc)$ represents the cost of retrieving the document Dc and $Size(Dc)$ represents the size Dc. For an arbitrary cache Ec_l, let $Min_{HValue}(Dc, Ec_l)$ represent the minimum of the $HValues$ of all the documents that are currently stored in Ec_l. When a document needs to be evicted from the cache Ec_l in order to make space for an incoming document, the greedy-dual size algorithm selects the document with the minimum $HValue$. That is the document whose $HValue$ is equal to $Min_{HValue}(Dc, Ec_l)$ is selected for eviction. After the document eviction, the $HValues$ is reduced by $Min_{HValue}(Dc, Ec_l)$. If a document stored in the cache is hit, its $HValue$ is restored to the original $HValue$ it had, when it entered the cache.

Although both the greedy-dual size algorithm and our utility-based document placement

94

scheme adopt a cost-benefit approach for cache management, there are also important differences between them. While, the greedy-dual size algorithm is a document replacement scheme, the utility-based scheme is a document placement scheme, which takes a proactive approach to cache management. Further, most current formulations of the greedy-dual size algorithm's cost function do not consider the document consistency costs, and hence, they are not directly applicable to caching dynamic documents. In this scenario, the question that comes up is: whether the utility function of our document placement scheme can be utilized in conjunction with schemes such as greedy-dual size algorithms to design document replacement policies for dynamic web content caches? The four components of our utility function capture various costs and benefits involved in caching dynamic web content. Therefore, we think that our utility function can be adapted in a framework such as the greedy-dual size, to yield highly effective document replacement strategies for caches storing dynamic web content.

4.5 Conclusion

In this chapter we have presented the architectural design of the cache clouds, which form the basic framework of cooperation in the cooperative EC grid. We have adopted a distributed approach in designing the cache clouds, wherein the caches belonging to the cloud share the load of document lookups and updates. The caches are organized into sub-structures called beacon rings, and the caches belonging to a beacon ring are collectively responsible for handling the updates and lookups of a set of documents. We have proposed a dynamic hashing protocol for document lookups and updates, which not only balances the load due to these operations among the caches, but also maintains the balance of loads even when the load pattern changes. We have also presented two mechanisms to provide failure resilience, namely, the eager and the periodic replication schemes. Further, we have proposed a utility-based scheme for placing document within the cache cloud such that the system resources are optimally utilized, and the client latency is minimized. We have reported a series of experiments to evaluate the proposed schemes. Our experiments indicate that these schemes can provide significant performance benefits.

CHAPTER V

AUTOMATIC FRAGMENT DETECTION IN DYNAMIC WEB PAGES

In this chapter, we present our scheme for automatically detecting fragments in dynamic web pages, which is the second major contribution of this thesis.

Among the several research efforts that have been made to address the challenges posed by the enormous increase in dynamic web content, *fragment*-based publishing and caching of web pages [4, 32, 33, 43] stands out; it has been successfully commercialized in recent years. Conceptually, a fragment is a portion of a web page which has a distinct theme or functionality and is distinguishable from the other parts of the page. A web page has references to these fragments, which are stored independently on the server and in caches. In a fragment-based publishing scheme, the cacheability and the lifetime are specified at a fragment granularity rather than at the web page level. While cacheability properties specify whether a fragment can be cached, its lifetime indicates how long the fragment would remain fresh (in-sync with the server-copy).

The advantages of the fragment-based schemes have been conclusively demonstrated [33, 43]. By separating the non-personalized content from the personalized content and marking them as such, it increases the cacheable content of the web sites. Furthermore, with the fragment-based solution, a whole web page need not be invalidated when only a part of that page expires. Hence the amount of data that gets invalidated at the caches is reduced. In addition, the information that is shared across web pages needs to be stored only once, which improves disk space utilization at the caches.

Although researchers have made considerable efforts to improve the performance and benefits of fragment-based caching, there has been little research on detecting cache-effective fragments in web sites. Fragment-based caching solutions typically rely upon web pages that have been manually fragmented at their respective web sites by the web administrator or

Figure 31: Fragments in a Web Page

the web page designer. Manual markup of fragments in dynamic web pages is both labor-intensive and error-prone. More importantly, identification of fragments by hand does not scale as it requires manual revision of the fragment markups in order to incorporate any new or enhanced features of dynamic content into an operational fragment-based solution framework. Furthermore, the manual approach to fragment detection becomes unmanageable and unrealistic for edge caches that deal with multiple content providers. Thus there is a need for schemes that can automatically detect "interesting" fragments in dynamic web pages, and that are scalable and robust for efficiently delivering dynamic web content. By "interesting" we mean that the fragments detected are cost-effective for fragment-based caching.

Automatic detection of fragments presents two unique challenges. First, compared with static web pages, dynamically generated web pages have three distinct characteristics. On the one hand, dynamic web pages seldom have a single theme or functionality and they typically contain several pieces of information with varying freshness or sharability requirements. On the other hand, most of the dynamic and personalized web pages are not completely dynamic or personalized. Often the dynamic and personalized content are embedded in

97

relatively static web page templates [20]. Furthermore, dynamic web pages from the same web site tend to share information among themselves.

Figure 31 shows a dynamic web page generated through a fragment-based publishing system. This Football Sport Today Page was one of the web pages hosted by IBM for a sporting event. It contains five interesting fragments that are cost-effective candidates for fragment-based caching: (1) the latest football results on the women's final, (2) the latest medal tally, (3) a daily schedule for women's football, (4) the navigation menu with the IBM logo for the sport site on the top of the page and (5) the sport links menu on the left side of the page. These fragments differ from each other in terms of their themes, functionalities, and invalidation patterns. For example, the latest results fragment changes at a different rate than the latest medal tally fragment, which in turn changes more frequently than the fragment containing the daily schedule. In contrast, the navigational menu on the top of the page and the sport links menu on the left side of the page are relatively static and are likely to be shared by many dynamic pages generated in response to queries on sport events hosted from the web site.

Second, it is apparent from the above example that humans can easily identify fragments with different themes or functionality based on their prior knowledge in the domain of the content (such as sports in this example). However, in order for machines and programs to automate the fragment detection process, we need mechanisms that on the one hand can correctly identify fragments with different themes or functionality without human involvement, and on the other hand are efficient and effective for detecting and flagging such fragments through a cross-comparison of multiple pages from a web site.

In this chapter, we present a novel scheme to automatically detect and flag fragments which are cost-effective for fragment-based caching. The proposed scheme examines the web pages from a given web site and analyzes their properties such as the information shared among them, the personalization characteristics they exhibit, and their change frequencies. Based on this analysis, our system detects and flags the "interesting" fragments in a web site. We consider a fragment interesting if it has good sharability with other pages served from the same web site or it has distinct lifetime characteristics. The fragments detected

through our system may be used in any system that supports fragment-based dynamic content generation and delivery, including the cooperative EC grid.

This chapter makes three specific contributions:

- First, we propose a framework for automatic fragment detection. This framework includes augmented fragment tree with shingles [24, 25, 69] encoding, which is a fragment-aware data structure for modeling dynamic web pages. Further, we also provide a fast algorithm for incremental shingle computation.

- Second, we present an efficient algorithm for detecting fragments that are shared among M documents, which we call the *Shared Fragment Detection Algorithm*. This algorithm has two distinctive features:

  1. It uses node buckets to speed up the comparison and the detection of exactly or approximately shared fragments across multiple pages.
  2. It introduces sharing factor, minimum fragment size, and minimum matching factor as the three performance parameters to measure and tune the performance and the quality of the algorithm in terms of the fragments detected.

- Third, we present an effective algorithm for detecting fragments that have different lifetime characteristics, which we call the *Lifetime-Personalization based (L-P) Fragment Detection Algorithm*. A unique characteristic of the L-P algorithm is that it detects fragments which are most beneficial to caching based on the nature and the pattern of the changes occurring in dynamic web pages.

We discuss several performance enhancements to these basic algorithms, and evaluate the proposed fragment detection scheme through a series of experiments, showing the effectiveness and costs of our approach. Further, we also report our experimental study on the effect of adopting the fragments detected by our system on the web caches and the origin servers.

5.1 Candidate Fragments

In general, a fragment can be considered as a part of a web page. Our goal for automatic fragment detection is to find interesting fragments in dynamic web pages, which exhibit

potential benefits and thus are cost-effective as cache units. We refer to these interesting fragments as *candidate fragments* in the rest of the paper.

The web documents considered here are *well-formed* HTML documents [26] although the approach can be applied to XML documents as well. Documents that are not well formed can be converted to well-formed documents through document normalization, for example using HTML Tidy [5].

Concretely, we introduce the notion of candidate fragments as follows:

- Each Web page of a web site is a candidate fragment.

- A part of a candidate fragment is itself a candidate fragment if any one of the two conditions is satisfied:

  - The part is shared among "M" already existing candidate fragments, where M > 1.

  - The part has different personalization or lifetime characteristics than those of its encompassing (parent or ancestor) candidate fragment.

A formal definition of candidate fragments for web pages of a web site is given below:

Definition 1 *(Candidate Fragment)*

Let W denote the set of web pages available on a web site S and $CF(v)$ denote the set of all the fragments contained in fragment v. A fragment y is referred to as an ancestor fragment of another fragment x iff y directly or transitively contains fragment x. Let $AF(v)$ denote all the ancestor fragments of the fragment v and FS denotes the set of fragments corresponding to the set of documents in W such that $FS = \cup_{i=1}^{\|W\|} CF(D_i)$. For any document D from web site S, a fragment x in $CF(D)$ is called a candidate fragment if one of the following conditions is satisfied:

1. *x is an entire web page available at web site S, i.e $x \in W$.*

2. *x is a maximal Shared fragment, namely:*

- *x is shared among M distinct fragments $F_1, \ldots, F_M$, where $M > 1$, $F_i \in FS$, and if $i \neq j$ then $F_i \neq F_j$; and*

- *there exists no fragment y such that $y \in AF(x)$, and y is also shared among the M distinct fragments $F_1, \ldots, F_M$.*

3. *x is a fragment that has distinct personalization and lifetime characteristics. Namely, $\forall z \in AF(x)$, x has different personalization and lifetime characteristics than z.*

We observe that this is a recursive definition with the base condition being that each web page is a fragment. It is also evident from the definition that the two conditions are independent. These conditions define fragments that benefit caching from two different perspectives. We call the fragments satisfying Condition 1 **Shared fragments**, and the fragments satisfying Condition 2 **L-P fragments** (denoting Lifetime-Personalization based fragments). Lifetime characteristics of a fragment govern the time duration for which the fragment, if cached, would stay fresh (in tune with the value at the server). The personalization characteristics of a fragment correspond to the variations of the fragment in relation to cookies or parameters of the URL.

It can be observed that the two independent conditions in the candidate fragment definition correspond well to the two aims of fragment caching. By identifying and creating fragments out of the parts that are shared across more than one fragment, we aim to avoid unnecessary duplication of information at the caches. By creating fragments that have different lifetime and personalization properties we not only improve the cacheable content but also minimize the amount and frequency of the information that needs to be invalidated.

Our definition of candidate fragments permits candidate fragments to be embedded in one or more existing candidate fragments. All the candidate fragments that contain a candidate fragment cf_j are considered to be parent fragments of cf_j. Thus Cf_j can have multiple parent fragments. The *Depth* of a candidate fragment indicates how deep the fragment is embedded in the web pages, and it is defined as follows: Suppose $W = \{w_1, w_2, \ldots, w_n\}$ denote the set of all web pages available at a web site S, and $CF = \{cf_1, cf_2, \ldots, cf_m\}$ denote the set of all candidate fragments from the web site S. Let $Pf(cf_j)$

denote the set of all parent fragments of cf_j. The depth of a fragment cf_j (denoted as $Depth(cf_j)$) is defined as follows: $Depth(cf_j)$ is 1 if cf_j is a web page and it is not embedded in any other fragment. Otherwise, $Depth(cf_j)$ is one more than the maximum of the depths of its parent fragments. Formally,

$$Depth(cf_j) = \begin{cases} 1 & \text{If } cf_j \in W \text{ and } PF(cf_j) = \varnothing \\ (1 + Maximum(Depth(cf_k)), \forall cf_k \in PF(cf_i)) & \text{Otherwise} \end{cases}$$

The depth of fragmentation of a fragment detection scheme is defined as the maximum of the depths of all the candidate fragments, i.e., $DepthFragmentation = Maximum(Depth(cf_k))$, $\forall cf_k \in CF$.

As we are interested only in the fragments that satisfy the conditions in Definition 1, in the rest of the thesis we use the term fragments to refer to candidate fragments when there is clearly no confusion.

5.2 Framework for Automatic Fragment Detection

In this section we discuss the basic design of our automated fragment detection framework, including the system architecture, the efficient fragment-aware data structure for automating fragment detection, and the important configurable parameters in our system.

5.2.1 System Overview

The primary goal of our system is to detect and flag candidate fragments from dynamic pages of a given web site. The fragment detection process is divided into three steps. First, the system is conceived to construct an *Augmented Fragment Tree* (AF tree) for the dynamic pages fed into the fragment detection system. Second, the system applies the fragment detection algorithms on the augmented fragment trees to detect the candidate fragments in the given web pages. In the third step, the system collects statistics about the fragments such as size, how many pages share the fragment, and access rates. These statistics aid the administrator in deciding whether to enable fragmentation. Figure 32 gives a sketch of the system architecture.
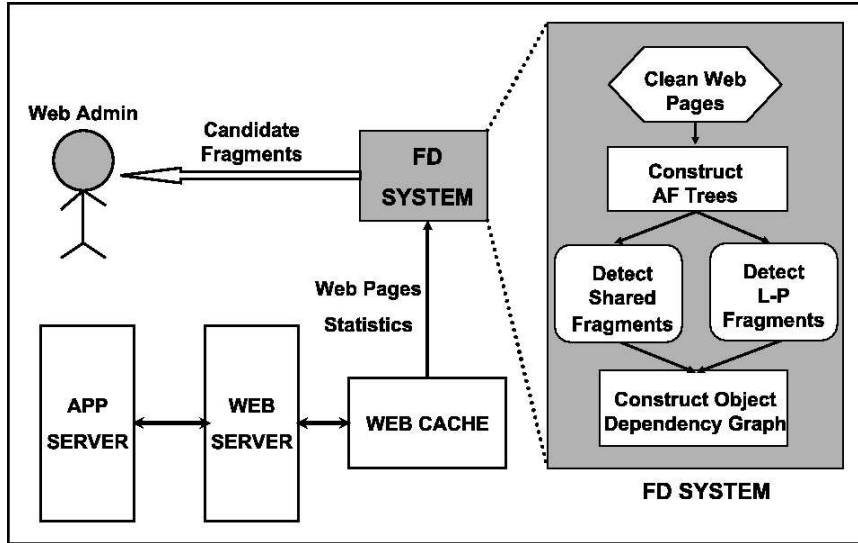
Figure 32: Fragment Detection System Architecture

We provide two independent fragment detection algorithms: one for detecting Shared fragments and the other for detecting Lifetime Personalization based (L-P) fragments. Both algorithms can be collocated with a server-side cache or an edge cache, and they work on the dynamic web page dumps from the web site.

The algorithm for detecting Shared fragments works on a collection of different dynamic pages generated from the same web site, whereas the L-P fragment detection algorithm works on different versions of each web page, which can be obtained from a single query being repeatedly submitted to the given web site. For example, in order to detect L-P fragments, we need to locate parts of a fragment that have different lifetime and personalization characteristics. This can be done by comparing different versions of the dynamic web page and detecting the parts that have changed over time and the parts that have remained constant. While the input to the L-P fragment detection algorithm differs from the shared fragment detection algorithm, both algorithms work directly on the augmented fragment tree representation of its input web pages. The output of our fragment detection algorithms is a set of fragments, which will be served as recommendations to the fragment caching policy manager or the web administrator.

5.2.2 Augmented Fragment Trees with Shingles Encoding

Detecting interesting fragments in web pages requires efficient traversal of web pages. Thus a compact data structure for representing the dynamic web pages is critical to efficient fragment detection. Of the several document models that have been proposed, the most popular model is the Document Object Model (DOM) [3], which models web pages using a hierarchical graph.

However, the DOM tree structure is not very efficient for fragment detection for a number of reasons. First, our fragment detection algorithms compare pages to detect those fragments whose contents are shared among multiple pages or whose contents have distinctive expiration times. The DOM tree of a reasonably sized HTML page has a few thousand nodes. Many of the nodes in such a tree correspond to text formatting tags that do not contribute to the content-based fragment detection algorithms. Second and more importantly, the nodes of the DOM do not contain sufficient information needed for fast and efficient comparison of documents and their parts. These motivate us to introduce the concept of an augmented fragment tree (AF tree), which removes the text formatting tag nodes in the fragment tree and adds annotation information necessary for fragment detection.

An AF tree with shingles encoding is a hierarchical representation of a web (HTML or XML) document with the following three characteristics: First, it is a compact DOM tree with all the text-formatting tags (e.g., <Big>, <Bold>, <I>) removed. Our experiments indicate that the number of nodes in the AF tree of a typical web page is around 20% lesser than the number of nodes in its corresponding DOM tree. Second, the content of each node is fingerprinted with shingles encoding [24, 25, 69]. Shingles are fingerprints with the property that if a document changes by a small amount, its shingles encoding also changes by a small amount. Third, each node is augmented with additional information for efficient comparison of different documents and different fragments of documents. Concretely each node in the *AF* tree is annotated with the following fields:

- Node Path (NodePath): A vector indicating the location of the node in the tree.

- SubtreeValue: A string that is defined recursively. For a leaf node, the SubtreeValue

Figure 33: Example of Shingles versus MD5

is equal to the text contained in the node. For all internal nodes, the SubtreeValue is a concatenation of the SubtreeValues of all its children nodes. The SubtreeValue of a node can be perceived as the fragment (content region) of a web document anchored at this subtree node.

- SubtreeSize: An integer whose value is the length of SubtreeValue in bytes. This represents the size of the structure in the document being represented by this node.

- SubtreeShingles: An encoding of the SubtreeValue for fast comparison. SubtreeShingles is a vector of integers representing the shingles of the SubtreeValue.

We use shingles because they have the property that if a document changes by a small amount, its shingles also change by a small amount. Other fingerprinting techniques such as MD5 do not behave similarly, if applied to the entire document.

Figure 33 illustrates the high sensitivity of shingles by comparing it with the MD5 hash through an example of two strings. The first and the second strings in Figure 33 are essentially the same strings with small perturbations (the portions that differ in the two strings have been highlighted). The MD5 hashes of the two strings are totally different, whereas the shingles of the two strings vary just by a single value out of the 8 values in the

shingles set (shingle values that are present in one set but are absent in the other have been underlined in the diagram). This property of shingles has made it popular in estimating the resemblance and containment of documents [24].

5.2.2.1   AF Tree Construction

The first step of our fragment detection process is to convert web pages to their corresponding AF trees. The AF tree can be constructed in two steps. The first step is to transform a web document to its DOM tree and prune the fragment tree by eliminating the text formatting nodes. The result of the first step is a specialized DOM tree that contains only the content structure tags (e.g., like <TABLE>, <TR>, <P>). The second step is to annotate the fragment tree obtained in the first step with NodePath, SubtreeValue, SubtreeSize and SubtreeShingles.

Once the SubtreeValue is known, we can use a shingles encoding algorithm to compute its SubtreeShingles. We discuss the basic algorithm [24] to compute the shingles for a given string.

The Basic Shingling Algorithm

Any string can be considered as a sequence of tokens. The tokens might be words or characters. Let $Str = T_1 T_2 T_3 ... T_N$, where T_i is a token and N is the total number of tokens in Str. Then a shingles set of window length W and sample size S is constructed as follows. The set of all subsequences of length W of the string Str is computed. $SubSq = \{T_1 T_2 ... T_W, T_2 T_3 ... T_{W+1}, ..., T_{N-W+1} T_{N-W+2} ... T_N\}$. Each of these subsequences is hashed to a number between $(0, 2^K)$ to obtain a token-ID. A hash function similar to Rabin's function [82] is employed for this purpose. The parameter K governs the size of the hash value set to which the subsequences are mapped. If the parameter K is set to a small value many subsequences might be mapped to the same token-ID, leading to collisions. Larger values of K are likely to avoid these collisions of subsequence, but increase the size of the hash value set. We now have $(N - W + 1)$ token-IDs, each corresponding to one subsequence. Of these $(N - W + 1)$ token-IDs, the minimum S are selected as the (W, S) shingles of string Str. The parameters W, S, and K can be used to tune the performance

106

and quality of the shingles encoding [24]. For example, larger values of S provides better estimates of the resemblance between documents, but at higher storage and computation costs. In our experiments we have set W to be 20 bytes, S to be 25 samples, and K to be 32.

The basic shingles computation algorithm is suitable for computing shingles for two independent documents. However, computing the shingles on the SubtreeValues independently at each node would entail unnecessary computations and is inefficient. This is simply because the content of every node in an AF tree is also a part of the content of its parent node. Therefore computing the SubtreeShingles of each node independently leads to a much higher cost due to duplicated shingles computation than computing the SubtreeShingles of a parent node incrementally. We propose an incremental shingles computation method and call it the **Hi**erarchical **Sh**ingles Computing scheme (the **HiSh** scheme for short).

5.2.3   Efficient Shingles Encoding - The HiSh Algorithm

In this section we describe a novel method to compute shingles incrementally for strings with hierarchical structures such as trees. By incremental we mean the HiSh algorithm reuses the previously computed shingles in the subsequent computation of shingles.

Consider a string $A = A_1A_2A_3...A_nA_{n+1}...A_m$ with m tokens, $m \geq 1$. Let B and C be two non-overlapping substrings of A such that A is a concatenation of B and C. Let $B = A_1A_2...A_n$ and $C = A_{n+1}A_{n+2}...A_m$. Now we describe how to incrementally compute the (W, S) shingles of A, if (W, S) shingles of B and C are available. Let $Shng(A, W, S)$, $Shng(B, W, S)$ and $Shng(C, W, S)$ denote the (W, S) shingles of the strings A, B and C respectively. We define the *Overlapping Sequences* to be those subsequences which begin in B and end in C. These are the subsequences that are not completely present in either shingles of B or shingles of C. Let the hashes of these subsequences be represented by the set $OvlpHsh = \{Hsh(A_{(n-W+2,n+1)}),$
$Hsh(A_{(n-W+3,W+2)}), ..., Hsh(A_{(n,n+W-1)})\}$. Then we can obtain the (W, S) shingles of A as follows:

$$Shng(A, W, S) = Min_S\{Shng(B, W, S) \bigcup Shng(C, W, S) \bigcup OvlpHsh\}$$

Figure 34: HiSh Algorithm

Here $Min_S(Z)$ denotes the operation of selecting the S minimum values from values in set Z.

As the shingles of B and C are available, the only extra computations needed are to compute the hashes of overlapping sequences. This is the central idea of the HiSh algorithm. Figure 34 illustrates the working of the HiSh scheme on an example string. In this example, $(8,4)$ shingles of the string B and string C are pre-computed and available, and we want to compute the $(8,4)$ shingles of the concatenation of the two strings. The HiSh algorithm computes the overlapping subsequences between the two strings (which is shown as "Overlap" in the figure) and computes the shingles on this overlapping string. Finally, the algorithm selects the minimum 4 values from all three strings to yield the shingles of the entire string.

Our experiments (see Section 5.5.4) indicate that the HiSh optimization can reduce the number of hashes computed in constructing the AF tree by as much as 9 times and improve the shingles computation time by 6 times for 20-Kbyte documents, when compared to the basic algorithm. The performance gain will be greater for larger documents.

5.3   Detecting Shared Fragments

This section discusses our algorithm to detect shared fragments. Given a collection of N dynamic web pages generated in response to distinct queries over a web site, let AF_i $(1 \leq i \leq N)$ denote the AF tree of the ith page. We call a fragment $F \in AF_i$ a *maximal shared fragment* if it is shared among $M(M < N)$ distinct fragments (pages) and there is no ancestor fragment of F which is shared by the same M fragments (pages). Here M is a system-defined parameter. With this definition in mind, the immediate question is how to efficiently detect such shared fragments, ensuring that the fragments detected are cost-effective cache units and beneficial for fragment-based caching.

5.3.1   Algorithm for Shared Fragment Detection

Our experiences with fragment-based solutions show that any shared fragment detection algorithm should address the following two fundamental challenges. First, one needs to define the measurement metrics of sharability. In a dynamic web site it is common to find web pages sharing portions of content that are similar but not exactly the same. In many instances the differences among these portions of content are superficial (e.g., they have only formatting differences). Thus a good automatic fragment detection system should be able to detect these approximately shared candidate fragments. Different quantifications of what is meant by "shared" can lead to different quality and performance of the fragment detection algorithms. The second challenge is the need for an efficient and yet scalable implementation strategy to compare the fragments (and the pages) and identify the maximal shared fragments.

Approximate Sharability Measures

The Shared fragment detection algorithm operates on various web pages from the same web site and detects candidate fragments that are "approximately" shared. We introduce three measurement parameters to define the appropriateness of such approximately shared fragments. These parameters can be configured based on the needs of a specific application.

- **Minimum Fragment Size***(MinFragSize):* This parameter specifies the minimum

size of the detected fragment.

- **Sharing Factor***(ShareFactor):* This indicates the minimum number of pages that should share a segment in order for it to be declared a fragment.

- **Minimum Matching Factor***(MinMatchFactor):* This parameter specifies the minimum overlap between the SubtreeShingles to be considered as a shared fragment.

The parameter $MinFragSize$ is used to exclude very small segments of web pages from being detected as candidate fragments. This threshold on the size of the documents is necessary because the overhead of storing the fragments and composing the page would be high if the fragments are too small. The parameter $ShareFactor$ defines the threshold on the number of documents that have shared each candidate fragment. Finally, we use the parameter $MinMatchFactor$ to model the significance of the difference between two fragments being compared. Two fragments being compared are considered as sharing significant content if the overlap between their SubtreeShingles is greater than or equal to $MinMatchFactor$.

Detecting Shared Fragments with Node Buckets

The shared fragment detection algorithm detects the shared fragments in two steps as shown in Figure 35. First, the algorithm creates a sorted pool of the nodes in the AF trees of all the web pages examined using node buckets. Then, the algorithm groups those nodes that are similar to each other together and runs the condition test for maximal shared fragments. If the number of nodes in the group exceeds the minimum number of pages specified by the $ShareFactor$ parameter, and the corresponding fragment is indeed a maximal shared fragment, the algorithm declares the node group as a shared fragment and assigns it a fragment identifier.

Step 1: Putting Nodes into a sorted pool of node buckets

More concretely, our algorithm uses the *bucket* structures to create a sorted pool of nodes. Buckets are used to efficiently sort the nodes of the AF trees based on their sizes. The

Figure 35: Shared Fragment Detection Algorithm

algorithm creates N_B buckets. Each bucket Bkt_i is initialized with bucket size Bs_i, and is associated with a pre-assigned range of the SubtreeSizes, denoted as $(MinSize(Bkt_i)$, $MaxSize(Bkt_i))$. The AF trees are processed starting from the root of each tree, and a node is placed into an appropriate bucket based on its SubtreeSize, such that the SubtreeSizes of all nodes in bucket Bkt_i are between $MinSize(Bkt_i)$ and $MaxSize(Bkt_i)$. If in the process of putting nodes into buckets, a bucket grows out of its current size Bs_i, it will be split into two or more buckets. Similarly, if the first step results in a pool of buckets with uneven distribution of nodes per bucket, a merge operation will be used to merge two or more buckets into one.

After all the AF trees have been processed and the nodes entered into their corresponding buckets, each buckets is sorted based on the SubtreeSize of the nodes in the bucket. At the end of the process we have a set of buckets containing nodes, each of which is sorted based on the SubtreeSize of the node. The STEP 1 in Figure 35 shows how this step works on two AF trees A and B. The nodes of the two trees are put into 5 buckets based on their SubtreeSizes. The buckets are sorted, and the buckets BT_3, BT_4 and BT_5 are merged to obtain a set of sorted buckets.

There are three system-supplied parameters: (1) the number of buckets (N_B) employed for this purpose, (2) the size B_i of each bucket, and (3) the range of each bucket $(MinSize(Bkt_i), MaxSize(Bkt_i))$. Various factors may affect the decision on how to set these parameters, including the number of AF trees examined, the average number of nodes in each AF tree and the range of the SubtreeSizes of all the nodes.

The performance of this step would be better if the nodes are evenly distributed in all the available buckets. One way to achieve such balanced distribution of nodes across all buckets is to set the ranges of the buckets at the lower end of the size spectrum to be smaller, and let the range of the buckets progressively increase for the buckets at the higher end of the size spectrum. This strategy is motivated by the following observations. First, it is expected that the number of nodes at a lower level of the AF trees would be larger than the number of nodes at a higher level. Second, the SubtreeSizes of the nodes at the lower level is expected to be smaller than the SubtreeSizes of the nodes in the higher levels of the AF tree.

Step 2: Identifying maximal shared fragments through grouping of similar nodes

The task of the second step is to compare nodes and group nodes that are similar to each other together and then identify those groups of nodes that satisfy the definition of maximally shared fragments. This step processes the nodes in the buckets in decreasing order of their sizes. It starts with the node having the largest SubtreeSize, which is contained in the bucket with the highest $MaxSize$ value. For each node being processed, the algorithm compares the node against a subset of the other nodes. This subset is constructed as follows. If we are processing node A_i, then the subset of nodes that A_i is compared against should include all nodes whose sizes are larger than $P\%$ of the SubtreeSize of A_i, where P can range from 0% to 100%. Let $CSet(A_i)$ denote the subset of nodes with respect to node A_i. We can use the following formula to compute $CSet(A_i)$.

$$CSet(A_i) = \{A_j | SubtreeSize(A_j) \geq \frac{P \times SubtreeSize(A_i)}{100}\}$$

It is important to note that the value setting of the parameter P has implications on both the performance and the accuracy of the algorithm. If P is too low, it increases the number of comparisons performed by the algorithm. If P is very close to 100, then the

number of comparisons decrease; however, it might lead the comparison process to miss some nodes that are similar. In practice we have found a value of 90% to be appropriate for most web sites.

When comparing the node being processed with the nodes in its $CSet$, the algorithm compares the SubtreeShingles of the nodes. Let Resemblance(A_i, B_j) denote the resemblance between the nodes A_i and B_j based on similarity of their SubtreeShingles. We can compute Resemblance(A_i, B_j) using the following formula [24]:

$$Resemblance(A_i, B_j) = \frac{SubtreeShingles(A_i) \cap SubtreeShingles(B_j)}{SubtreeShingles(A_i) \cup SubtreeShingles(B_j)}$$

All such nodes whose resemblance with the node being processed exceed minimal matching factor ($MinMatchFactor$) are grouped together. Specifically, a node $B_j \in CSet(A_i)$ is put into the group if $Resemblance(A_i, B_j) \geq MinMatchFactor$.

STEP 2 of Figure 35 demonstrates the comparison and grouping of the nodes in the sorted buckets. The cost of computing the overlap between two nodes is equal to the sum of the costs of computing the intersection and the union of two sets with S elements, where S is the sample size of the SubtreeShingles.

If this group has at least $ShareFactor$ nodes then we have the possibility of detecting it as a fragment. However before we declare the group as a candidate fragment, we need to ensure that the fragment corresponding to this group of nodes is indeed a maximally shared fragment.

To ease the decision on whether a group of nodes with similar shingles is a maximally shared fragment, we mark the descendent of each declared fragment with the fragment-ID assigned to the fragment. When similar nodes are detected, we check whether the ancestors of all of the nodes belong to the same fragment. If so, we reject the node group as a trivial fragment. Otherwise we declare the node group as a candidate fragment, assign it a fragment-ID and mark all of the descendant nodes with the fragment-ID. Once we declare a node-group as a candidate fragment, we remove all the nodes belonging to that group from the buckets. The algorithm proceeds by processing the next largest node in the node group in the same manner. We provide the pseudo-code in Algorithm 2.

Figure 36: Illustration of Shared Fragment Detection on BBC Website

5.3.2 Illustration on Real Web Data

In this section we illustrate the working of shared fragment detection algorithm on real web pages and demonstrate the effect of the configurable parameters on the detected fragments.

Figure 36 shows parts of two web pages from BBC. The first web page part is taken from the BBC's World news page and the second part appeared in the BBC's Mid-East page. AF Tree - 1 and AF Tree - 2 depict the corresponding Augmented Fragment trees. The arrows in the figure show the mapping between nodes of the AF trees and their contents. The nodes A_1 and B_1 correspond to the entire web-page segments, whereas the nodes A_2, B_2 and A_3, B_3, represent the heading and the paragraph text respectively. The nodes A_4, B_4 represent the bulleted list in the two segments and the nodes $A_5 \ldots A_7$ and $B_5 \ldots B_7$, the individual bullet points.

114

Algorithm 2 The Shared Fragment Detection Algorithm
___
INPUT:
AF trees of web pages: $\{Af_1, Af_2..Af_P\}$
$MinFragSize$, $SharFactor$ and $MinMatchFactor$
OUPUT:
A set of Shared fragments: $\{SFd_1, SFd_2..SFd_q\}$
PROCEDURE:
Create a set of node buckets: $\{Bkt_1, Bkt_2..Bkt_l\}$
Initialize $AncestorFragment$ to $False$
Initialize $AncestorFragmentArray$ to NULL of all nodes
for $i = 0$ to P **do**
  Put all the nodes in tree Af_i whose $SubtreeSize \geq MinFragSize$ into appropriate buckets
end for
Sort and Merge the buckets $\{Bkt_1, Bkt_2..Bkt_l\}$
while Buckets are Non empty **do**
  $LrNd \Leftarrow$ Largest Node in Buckets
  $NewNodeGroup \Leftarrow LrNd$
  Compute the $CSet(LrNd)$
  for Each $Nd_g \in CSet(LrNd)$ **do**
    if Overlap between the shingles of Nd_g and $LrNd \geq MinMatchFactor$ **then**
      Add Nd_g to $NewNodeGroup$
    end if
  end for
  if Number of Nodes in $NewNodeGroup \geq$ SharFactor **then**
    if At least one Node has $AncestorFragment = False$ OR
    $AncestorFragmentArray$ of at least one Node differs from others **then**
      {/* New Maximal Fragment Detected */}
      Assign a $FragmentID$ to the fragment and add it to Fragment Set
      for All descendants of the nodes in $NewNodeGroup$ **do**
        Set $AncestorFragment$ to $True$
        Add $FragmentID$ to $AncestorFragmentArray$
      end for
    end if
  end if
  Remove all nodes in $NewNodeGroup$ from buckets
end while
Output the fragments in the $FragmentSet$
___

A high degree of similarity between these two web page parts makes them prime candidates for shared fragments. The contents of nodes A_2, A_3 of AF Tree-1 are identical to their corresponding nodes in AF Tree-2, whereas the contents of nodes A_5 and A_7 differ from their counterparts.

The shared fragment detection algorithm puts these nodes into buckets, sorts these buckets, and processes the nodes in decreasing order of their sizes. Hence the node A_1 is compared with other nodes to group similar nodes based on the value of $MinMatchFactor$. If $MinMatchFactor$ is set to 0.70, then the nodes A_1 and B_1 are grouped together as similar nodes and are detected as a candidate fragment. In this case the entire web page segment is detected as a single, large fragment. The nodes $A_2 \ldots A_7$ and $B_2 \ldots B_7$ are not declared

as fragments because they share the same set of ancestor fragments, and hence are not maximal fragments.

However, if $MinMatchFactor$ is set to a higher value, say 0.90, then the nodes A_1 and B_1 are not considered to be similar and these nodes are just removed from the buckets. In this case (A_2, B_2), (A_3, B_3), (A_6, B_6) are detected as fragments. Therefore, we see that the number of detected fragments increase as $MinMatchFactor$ increases, whereas the size of the detected fragments decrease. The experimental results in Section 5.5.1 reflect this effect of $MinMatchFactor$ on the number and the size of the detected fragments.

5.4 Detecting L-P Fragments

In this section we discuss the algorithm for detecting L-P fragments. One way to detect the L-P fragments is to compare various versions of the same web page and track the changes occurring over different versions of the web page. The nature and the pattern of the changes may provide useful lifetime and personalization information that is helpful for detecting the L-P fragments.

5.4.1 Algorithm for L-P Fragment Detection

The first challenge in developing an efficient L-P fragment detection algorithm is to identify the logical units in a given web page that may change over different versions, and to discover the nature of the change.

The second challenge is to detect candidate fragments that are most beneficial to caching. Suppose we have a structure such as a table in the web page being examined. Suppose the properties of the structure remain constant over different versions of the web page, but the contents of the structure have changed over different versions. Now there are two possible ways to detect fragments: Either the whole table (structure) can be made a fragment or the substructures in the table (structure) can be made fragments. Which of these would be most beneficial to caching depends upon what percentages of the substructures are changing and how they are changing (frequency and amount of changes).

In the design of our L-P fragment detection algorithm, we take a number of steps to address these two challenges. First, we augment the nodes of each AF tree with an additional

field *NodeStatus*, which takes one value from the set of three choices {*UnChanged*, *ValueChanged*, *PositionChanged*}. Second, we provide a shingles-based similarity function to compare different versions of a web page, and determine the portions of a web page that have distinct lifetime and personalization characteristics. Third, we construct the Object Dependency Graph (ODG) [33] for each web document examined on top of all candidate fragments detected.

An Object Dependency Graph is a graphical representation of the containment relationship between the fragments of a web site, which can be used to efficiently compose web pages at the servers and the caches [33]. The nodes of the ODG correspond to the fragments of the web site and the edges denote the containment relationship among them. Finally, we use the following configurable parameters to measure the quality of the L-P fragments in terms of cache benefit and to tune the performance of the algorithm:

- **Minimum Fragment Size***(MinFragSize):* This parameter indicates the minimum size of the detected fragment.

- **Child Change Threshold***(ChildChangeThreshold):* This parameter indicates the minimum fraction of children of a node that should change in value before the parent node itself can be declared as *ValueChanged*. It can take a value between 0.0 and 1.0.

The L-P fragment detection algorithm works on the AF trees of different versions of web pages. It installs the first version (in chronological order) available as the *base version*. The algorithm compares each subsequent version to the base version and identifies candidate fragments. A new base version is installed whenever the web page undergoes a drastic change when compared with the current base version. In each step, the algorithm executes in two phases. In the first phase it marks the nodes that have changed in value or in position between the two versions of the AF tree. In the second phase the algorithm outputs the L-P fragments which are then merged to obtain the object dependency graph.

Phase 1: Comparing the AF trees and detecting the changes

Concretely, if we have two AF trees A and B corresponding to two versions of a web page,

Figure 37: L-P Fragment Detection Algorithm

our algorithm compares each node of the tree B, to a node from A which is most similar to it. We employ the *Resemblance* measure (defined in Section 5.3) for similarity comparison between nodes.

If we are processing node B_j from AF tree B, we obtain a node A_i from tree A such that $Resemblance(A_i, B_j) \geq OvlpThrshld$, and there exists no A_h such that $Resemblance(A_h, B_j) > Resemblance(A_i, B_j)$ where $OvlpThrshld$ denotes a user-specified threshold for the quantity *Resemblance*, which can take a value between 0 and 1.0. If no such node is found in tree A, then it means that there is no node in A that is similar to the node B_j. Hence, the node B_j is marked as *ValueChanged*.

If a node A_i is found similar to node B_j, the algorithm begins comparing node B_j with node A_i. The algorithm compares the SubtreeValues and the NodePaths of the two nodes. If both SubtreeValue and NodePath of the two nodes exactly match then the node is marked *UnChanged*. If the NodePaths of the two nodes differ, then it means that the node has changed its position in the tree and hence it is marked as *PositionChanged*.

If the SubtreeValues of the nodes A_i and B_j do not exactly match then the algorithm

118

checks whether they are leaf nodes. If so, they are marked as *ValueChanged*. Otherwise, the algorithm recursively processes each child node of B_j in the same manner described above marking them as *ValueChanged*, *PositionChanged* or *UnChanged*.

The algorithm addresses the second issue of discovering the fragments based on the extent of changes it is undergoing by calculating the fraction of B_j's children that are marked as *ValueChanged*. If this fraction exceeds a preset threshold, which we call the *ChildChangeThreshold*, then B_j itself is marked as *ValueChanged*. The algorithm recursively marks all the nodes in the tree in the first phase.

In our algorithm, the decision regarding the status of a node is based upon the fraction of its children which have been marked as *ValueChanged*. Therefore, the statuses of all children nodes have equal impact on the status of their parent node. Alternatively, the weight given to the status of a child node in deciding its parent's status may be scaled according to size of the child node. Although we have not experimentally evaluated this variant of our basic algorithm, we believe that it would yield results comparable to those obtained by the basic algorithm (equal weights to all children nodes).

Phase 2: Detecting and labeling the L-P fragments

In the second phase, the algorithm scans the tree again from the root and outputs the nodes that are marked as *ValueChanged* or *PositionChanged*. The algorithm descends into a node's children if the node is marked as *PositionChanged* or *UnChanged*. If the node is marked as *ValueChanged*, the algorithm outputs it as a L-P fragment, but does not descend into its children. This ensures that we detect maximum-sized fragments that change between versions.

Figure 37 demonstrates the execution of one step in the L-P fragment detection algorithm. In the figure we compare the nodes of the AF tree of version 2 with the appropriate nodes of the AF tree of version 1. For example the node B_7 is compared with A_7 although these nodes appear at different positions in the two AF trees. In this example we set the *ChildChangeThreshold* to be 0.5. The node A_6 is marked as *ValueChanged* as both of its children have changed in value. The figure also indicates the fragments discovered in the

---

Algorithm 3 The L-P Fragment Detection Algorithm

INPUT:
AF trees of web pages: $\{Af_1, Af_2..Af_P\}$
Child Change Threshold: $ChildChangeThreshold$
Minimum Fragment Size: $MinFragSize$
OUTPUT:
Object Dependency Graph of the Detected Fragments: ODG
PROCEDURE:
for $i = 2$ to P **do**
   Compare the trees Af_i and Af_0 and mark the changed nodes via the recursive procedure $MarkChangedNodes(RootNode_1, RootNode_i)$
   Detect fragments and merge the fragments into ODG via the recursive procedure $DetectLPFragments(Root_i)$
end for
PHASE 1: MarkChangedNodes($Node_A$**, **$Node_B$**)**
Compare the $NodePaths$ of $Node_A$ and $Node_B$
if The $NodePaths$ of the two Nodes differ **then**
   Mark the NodeStatus as $PositionChanged$: $NodeStatus \Leftarrow PositionChanged$
end if
$NumChangedChildren = 0$
for Each Child Node ($ChldNode_B$) of $Node_B$ **do**
   Obtain the Nearest Node ($ChldNode_A$) from Children of $Node_A$ to compare $ChldNode$ with
   if There is no nearest node from children of $Node_A$ **then**
     Mark $NodeStatus$ of $ChldNode_B$ as $ValueChanged$
     $NumChangedChildren \Leftarrow NumChangedChildren + 1$
   else
     $ChldStatus \Leftarrow MarkChangedNodes(ChldNode_A, ChldNode_B)$
     if $ChldStatus = ValueChanged$ **then**
       $NumChangedChildren \Leftarrow NumChangedChildren + 1$
     end if
   end if
end for
if $\frac{NumChangedChildren}{TotalChildren} \geq ChildChangeThreshold$ **then**
   Mark $NodeStatus$ as $ValueChanged$: $NodeStatus \Leftarrow ValueChanged$
end if
Return($NodeStatus$)
PHASE 2: DetectLPFragment($Node_B$**)**
if $NodeStatus = ValueChanged$ **then**
   Declare $Node_B$ a Fragment and Merge the $Node_B$ into ODG
   Return;
end if
if $NodeStatus = PositionChanged$ **then**
   Declare $Node_B$ a Fragment and Merge the $Node_B$ into ODG
end if
for Each Child node of $Node_B$ say $ChldNode_B$ **do**
   DetectLPFragment($ChldNode_B$)
end for
Return

---

second pass of the algorithm. We provide the pseudo-code in Algorithm 3.

As the L-P fragment detection algorithm works on different version of a web page, the fragments detected by the algorithm are affected by the versions of the web page provided to it as input. The algorithm yields most accurate fragments when it receives each new

version of the web page within a given time period. In other words the performance of the algorithm is best when the web pages versions are captured each time the web page changes. If the web pages are not sampled at each observed change, the algorithm might miss some fragments.

In summary, our L-P fragment detection algorithm detects the parts of a web page that change in value and parts of web pages changing their position between versions. The fragments detected by the algorithm might either have different lifetimes or differ in their personalization characteristics. The lifetime-based fragments are detected by comparing various versions of the web page that are time-spaced, whereas fragments representing personalized information are detected by comparing versions of web pages that are generated for different users and tracking the changes the web page is undergoing. Fragments representing personalized content represent information that is specific to particular users, and hence might not be cacheable. Although lifetime fragments and personalization fragments are detected in a similar manner, they serve different purposes. While lifetime-based fragments decrease the amount of data invalidations at the caches, personalization-based fragments increase the cacheable content of the web site by clearly demarcating the potentially non-cacheable content.

5.4.2 Illustration of L-P Fragment Detection on Real Web Data

Next we illustrate the working of the L-P fragment detection algorithm and the effect of its configuration parameters on a web page from Slashdot. Figure 38 indicates the same segment from two versions of a web page from Slashdot and their corresponding AF trees. The arrows indicate the correspondence between the nodes of the AF tree and the web page content.

As we previously described, the first phase of the algorithm compares each node of the AF Tree-2 with the most similar node from AF Tree-1, and marks them as *UnChanged*, *PositionChanged* or *ValueChanged*. In the figure, the nodes B_7 through B_{10} are marked as *ValueChanged*, since these are new information appearing in this version and have no corresponding nodes in AF Tree-1. The nodes B_{11} through B_{14} are marked as *PositionChanged*

121

as they have changed the relative position in which they appear in the AF tree. For example, while the content of node B_{11} is identical to the content of node A_7 in AF Tree-1, they vary in the relative positions in which they appear in their respective AF trees.

Figure 38: Illustration of L-P Fragment Detection

In the second phase, the algorithm traverses the tree from the root and detects the L-P fragments. The value of the $ChildChangeThreshold$ parameter influences the fragments that are identified in this phase. In this example, we see that 4 out of 8 children of the node B_4 are marked as $ValueChanged$. Hence, if the $ChildChangeThreshold$ is set to 0.5 or lower values, the node B_4 would be detected as a single fragment. However, if the $ChildChangeThreshold$ is set to higher values, the individual nodes B_7 through B_{14} are

detected as fragments. Thus, when *ChildChangeThreshold* is set to higher values, it is more likely that nodes that are located deeper in the tree are flagged as fragments. As there are more nodes deeper in the tree, the number of fragments detected is higher. Equivalently, the average size of the fragment decreases as *ChildChangeThreshold* increases.

The appropriate value of *ChildChangeThreshold* depends upon factors such as the request rate at the web pages, the capacity of the server and the caches, and the bandwidth of the network connection between the caches and the server. While the amount of data invalidation and the bandwidth consumption of the network connection between the caches and the server decrease with increasing numbers of fragments, having a very large number of fragments increases the page composition costs, placing additional load on the individual caches.

5.5 Experimental Evaluation

We have performed a range of experiments to evaluate our automatic fragment detection scheme. In this section we report four sets of experiments. The first and second sets test the two fragment detection algorithms, showing the benefits and effectiveness of the algorithms. The third set studies the impact of the fragments detected by our system on improving caching efficiency, and the fourth set evaluates the Hierarchical Shingles computation scheme.

The input to the schemes is a collection of web pages including different versions of each page. We periodically fetched web pages from the web sites of BBC (http://news.bbc.co.uk), IBM's portal for marketing (http://www.ibm.com/us), Internetnews (http://www.internet-news.com) and Slashdot (http://www.slashdot.org) and created a web 'dump' for each web site. While most of these sites share information across their web pages and hence are good candidates for Shared fragment detection, the Slashdot web page forms a good candidate for L-P fragment detection for reasons explained in Section 5.5.2.

5.5.1 Detecting Shared Fragments

In the first set of experiments, we study the behavior of our Shared fragment detection algorithm. The data sets used in this experimental study were web page dumps from BBC,

Internet news and IBM. We primarily report the results obtained from our experiments on the BBC web site.

Figure 39: Number of Fragments Detected in BBC Data Set

BBC is a well-known news portal. Primarily, the web pages on the BBC web site can be classified into two categories: web pages reporting complete news and editorial articles (henceforth referred to as the 'article' pages) and the 'lead' pages listing the top news of the hour under different categories such as 'World', 'Americas' 'UK' etc. We observed that there is considerable information sharing among the lead pages. Therefore, the BBC web site is a good case study for detecting shared fragments. Our data set for the BBC web site was a web dump of 75 distinct web pages from the web site collected on 14th July 2002. The web dump included 31 'lead' pages and 44 'article' pages.

Figure 39 illustrates the number of Shared fragments detected at two different values of $MinFragSize$ and $MinMatchFactor$ (recall that $MinFragSize$ is the minimum size of the detected fragment and $MinMatchFactor$ is the minimum percentage of shingles overlap). When $MinFragSize$ was set to 30 bytes and $MinMatchFactor$ was set to 70%, the number of fragments detected was 350. The number of fragments increased to 358 when the $MinMatchFactor$ was set to 90% and to 359 when the $MinMatchFactor$ was set to 100%. In all of our experiments we observed an increase in the number of detected fragments with increasing $MinMatchFactor$. As we explained in Section 5.3.1, when $MinMatchFactor$

Figure 40: Maximum Size of the Detected Fragments

is set to a high value, the algorithm looks for (almost) perfect matches, which leads to an increase in the number of detected fragments and a drop in the size of the detected fragments.

Figure 41: Distribution of Fragment Sharing in BBC Data Set

Figure 40 indicates the maximum size of the detected fragments for various data sets when $MinMatchFactor$ was set to 70% and 90%. For the BBC web site, the change in the size of the largest detected fragment is rather drastic. The size falls from 5633 bytes to 797 bytes when $MinMatchFactor$ increases from 70% to 90%.

Table 2 shows the sum of sizes of the shared fragments detected by our algorithm at

Table 2: Sum of Sizes of Shared Fragments Detected in the BBC Data Set

$MinMatchFactor$	$MinFragSize$ = 30 bytes	$MinFragSize$ = 50 Bytes
0.70	136 Kbytes	130 Kbytes
0.90	122 Kbytes	115 Kbytes
1.00	121 Kbytes	115 Kbytes

different values of $MinMatchFactor$ and $MinFragSize$ for the BBC data set. The total number of bytes in the shared fragment set is higher at lower values of $MinMatchFactor$ and vice-versa.

The pie chart in Figure 41 indicates the percentage of fragments according to the number of pages sharing the fragments for the BBC data set. We see a large number of fragments (a little over 50%) are being shared by exactly two pages. 13% of the fragments were shared among exactly 3 pages, and 11% of the pages were shared by 10 pages or more. All 75 pages shared one fragment, and 3 fragments were shared by 69 pages. The mean of the number of pages sharing each of the detected fragments was 13.8. The fragments which were shared across a small number of pages were fragments such as synopses of news items (with links to news articles), tables indicating statistics that were relevant to news articles, etc. In contrast the fragments shared across a large percentage of web pages were typically fragments such as headers, footers, and navigational bars.

A similar type of behavior was observed in all three data sets. A large percentage of the detected fragments were shared by a small number of pages, but a few fragments were shared by almost all the web pages of the site.

5.5.2   Detecting L-P Fragments

We now present the experimental evaluation of the L-P fragment detection algorithm. We have primarily used the web pages hosted on the web site from Slashdot (http://www.slashdot.org) for our experiments.

Slashdot is a well known web site providing IT, electronics and business news. The front page of the Slashdot web site carries headlines and synopses of the articles on the site. The

Table 3: Statistics for L-P Fragment Detection

$ChildChangeThreshold$	0.50	0.70
Total Fragments	79	285
Average Fragment Size (in bytes)	822	219
Depth of Fragmentation	3	3
Sum of Sizes of Detected Fragments (in bytes)	64938	62415

page indicates the number of comments posted by other users under each article. Thus, as new comments are added to existing articles and new articles are added to the web site, the page changes in small ways relative to the entire content of the page. It therefore forms a good case for L-P fragment detection, as well as other techniques that identify similarity across pages. The same Slashdot data set has been used in another study of similarity across pages at the level of unstructured bytes, finding that different versions of the Slashdot home page within a short time frame are extremely compressible relative to each other [62].

This web page provides a good case study to detect L-P fragments for a number of reasons. First, this web page is highly dynamic. Not only are there parts of the page that change every few minutes, the web page experiences major changes every couple of hours. Second, various portions of the web page have different lifetime characteristics. Third, the web page experiences many different kinds of changes like additions, deletions, and value updates. Furthermore, there are parts of the web page that are personalized to each user.

Table 3 provides a synopsis of the results of the L-P fragment detection experiments. A total of 79 fragments were detected when the $ChildChangeThreshold$ was set to 0.50, and 285 fragments were detected when $ChildChangeThreshold$ was set to 0.70. As explained in Section 5.4.2, when $ChildChangeThreshold$ is set at higher values, larger numbers of small fragments are detected and vice versa.

In both cases, the depth of fragmentation was 3. The depth of fragmentation is defined at the end of Section 5.1. When $ChildChangeThreshold$ was set to 0.50, the number of fragments detected at depths 1, 2 and 3 were respectively 10, 7 and 62.

Figure 42 and Figure 43 indicate the lifetime distribution of the L-P fragments from

Figure 42: Fragment Lifetime Characteristics

Figure 43: Cumulative Distribution of Fragment Lifetimes

the Slashdot web site. Figure 42 shows the lifetime of the fragments detected when *ChildChangeThreshold* is set to 0.7 and 0.5. Figure 43 indicates the cumulative distribution of the detected fragments with respect to their lifetimes. As the cumulative distributions of the fragments detected by setting *ChildChangeThreshold* to 0.7 and 0.5 were similar to each other, the figure shows the cumulative distribution at *ChildChangeThreshold* = 0.7. We observe that when *ChildChangeThreshold* was set to 0.70, around 8% of the detected fragments had a lifetime of less than 5 minutes, around 17% of the fragments had lifetimes between 5 minutes and 1 hour, and around 6% had lifetimes of more than 24 hours.

5.5.3 Impact on Caching

Having discussed the experimental evaluation of our fragment detection system with regard to its accuracy and efficiency, we now study the impact of fragment caching on the performance of the cache, the server and the network when web sites incorporate fragments detected by our system into their respective web pages.

Figure 44: Total Storage Requirements

We begin by studying the savings in the disk space requirements of a fragment cache when the web pages incorporate the fragments discovered by our fragment detection system in comparison to a page cache that stores entire pages. Earlier we had explained the experimental evaluation of our shared fragment detection system on the BBC data set. We now compare the disk space needed to store the web pages in the data set when they are stored at the page granularity with disk space requirements for storing these web pages when they are fragmented as determined by our system.

Figure 44 indicates the total storage requirements as a function of the number of pages both for page caches and fragment caches. The graph shows that caching at the fragment level requires 22% to 31% less disk space than the conventional page level caching. The graph also shows that the improvements are higher when $MinMatchFactor$ is set to low values. This is because when $MinMatchFactor$ is set to low values, larger size fragments are discovered. When they are stored only once rather than being replicated, the savings

obtained in terms of the disk space are higher.

In the next experiment we study the effects of L-P fragments detected by our system on the load on the network connecting the cache and the server. As we discussed in Section 5.1, incorporating L-P fragments into web pages reduces the amount of data invalidated at the caches, which in turn reduces the load on the origin servers and the backbone network. In order to study the impact of the L-P fragments on the server and network load, we use the L-P fragments detected by our algorithm on the Slashdot web site.

To study the load on the network we also need the access patterns of the web pages and the lifetime characteristics of the fragments. We model the lifetime characteristics based on the fragment lifetime data collected from the Slashdot's web site. As we do not have the access pattern data for the web pages from Slashdot, we make certain assumptions, which aid us to model the access pattern. We assume that the requests for web pages arrive according to a *Poisson process*, as supported by past analysis [73]. We vary the request arrival rate from 1000 requests per minute to 10000 requests per minute.

Figure 45 indicates the total bytes transferred as a function of the number of requests arriving at the cache, at page access rates of 5000 and 7500 accesses per minute. The X-axis indicates the number of accesses and the Y-axis indicates the total number of bytes transferred, on log scale. The number of bytes transferred for page-level caching is always higher than for fragment-level caching. The effect is more pronounced when the access rates are low. This is because, at low access rates the probability of fragments getting invalidated between two consecutive accesses are higher. In case of fragment caching, only the invalidated fragments have to be fetched from the server, whereas the entire page has to be fetched if the caching is done at page granularity.

In Figure 46 we indicate the compulsory byte miss rates for the page caching and the fragment level caching schemes. Compulsory byte miss rate of a cache is defined as the miss rate (ratio of the bytes obtained from the server to the total bytes accessed at the cache) incurred exclusively due to freshness constraints on the web pages being served by the cache. In other words compulsory byte miss rate does not include byte-misses due to storage limitation of the cache. The X-axis indicates the ratio of mean fragment invalidation

Bytes Transferred between Server and Cache

Figure 45: Bytes Transferred between Server and Cache

rate to access rate on a log scale. In this graph, it is assumed that the cache has enough storage capacity to contain all the web pages and hence, there is no data replacement in the caches, which in turn implies that there are no misses due to storage limitations. All the misses are occurring because of the data invalidations occurring in the cache. The compulsory byte miss rate is defined as the ratio of the bytes fetched from the origin server to the total bytes accessed from the cache, when the cache only experiences compulsory misses.

Compulsory Miss Rate

Figure 46: Compulsory Byte Miss Rate

The graph in Figure 46 indicates that when the invalidation to access rate ratio is very low, the miss rates for both page level caching and fragment caching are very low. However,

when this ratio reaches 0.0001, the byte miss rate of the page level caching is 8.86%, whereas it is just 0.10% for fragment caching. When the invalidation to access rate ratio reaches 0.001, the byte miss rate of the page cache jumps to 51.4%, compared to 7.8% for fragment-based caching. Therefore fragment-based caching is clearly very useful when the web pages contain parts that are highly dynamic.

Figure 47: Server Load with Constant Fragment Generation Cost

Next we compare page caching and fragment caching with respect to the load on the server using two cost models. In both models the web pages were updated multiple times within the duration of the experiment. The first model, called the constant-cost model, assumes that the cost of generating each fragment is constant. In other words, generating each fragment involves one unit cost at the server. Figure 47 indicates the total cost incurred at the server per web page as a function of the total number of page accesses for the page cache and the fragment cache at page access rates of 5000 and 7500 accesses per minute. The X-axis indicates the total number of page requests, and the Y-axis shows the total cost at the server per page on a log scale.

In the second model, which we call the weighted-cost model, we assume that the cost of generating a fragment is proportional to the size of the fragment, with an average sized fragment costing one unit cost at the server. Figure 48 indicates the total cost incurred at the server per web page under this model.

Figure 48: Server Load with Weighted Fragment Generation Cost

In both models, the load on the server in the fragment-based caching mechanism is always less than the caching mechanism that stores the web pages at page granularity. For example in the constant-cost model, at an access rate of 7500 accesses per minute, the total cost at the end of 30K accesses is around 9.6 times the total server cost for fragment-based caching. Similarly, the total cost of page caching at the end of 30K accesses for the weighted-cost model at 7500 accesses per minute is around 9.48 times the corresponding cost for the fragment-based caching scheme.

In a page cache, when a single fragment expires, the whole page has to be fetched from the origin server. Fetching a particular page from the origin server involves all the fragments corresponding to the page to be regenerated. In the fragment-based caching, when a fragment gets invalidated only that particular fragment has to be fetched from the origin server. Therefore, the load on the server in the page caching scheme is much higher than the server load in the fragment-based caching scheme.

In conclusion, these experiments demonstrate that caches which store the fragments detected by our system effectively reduce server load and network bandwidth consumption, which are the key goals of web caching.

Figure 49: Number of Hashes Computed

5.5.4   Improving Fragment Detection Efficiency

We have proposed a number of techniques to improve the performance of the fragment detection process including an incremental scheme to compute the *SubtreeShingles* of the nodes in the AF trees (HiSh algorithm) and pruning the nodes of the DOM tree to obtain the more compact AF tree representation. In this section we evaluate these techniques. We first present the experimental evaluation of the HiSh algorithm.

Figure 50: Total Shingle Computation Time

Figure 49 shows the total number of hash computations involved in constructing the AF tree for various documents, and Figure 50 illustrates their total shingle computation

134

times. For a document with 1.7K characters in its content string, whose AF tree contained 409 nodes and had a depth of 6, the number of hash computations needed for the direct computation is 2.6 times the number of hashes computed in the HiSh scheme and takes 1.9 times more computation time than that of the HiSh scheme. For a document whose content string had 19K characters, and whose AF tree had a depth of 11 and contained 1390 nodes, the number of hashes computed in the direct computation is almost 8.5 times and takes 5.8 times more computation time. The improvements provided by the HiSh algorithm in terms of total shingle computation times may be less than the corresponding improvements in terms of the number of hash computations. This is due to overheads involved in operating on large strings and bookkeeping operations, which cannot be completely avoided.

Figure 51: Number of Nodes in DOM and AF Trees

Next we discuss the effectiveness of pruning the unnecessary nodes from the DOM tree. Figure 51 shows the number of nodes in the original DOM tree and the number of nodes after pruning. The web page from IBM had 1.7K bytes of content and 478 nodes. After node pruning the number of nodes comes down to 409, which is a reduction of 14%. For a page from the Internet News, which had a total size of 1416 nodes, the pruning reduces the number of nodes to 1152, which is a reduction of around 18%. Similarly for a web page from Slashdot, which had 1750 nodes in its DOM tree, the reduction was by 360 nodes or 20.5%. It should be noted that the percentage of pruned nodes is consistently increasing with the number of nodes in the original DOM tree.

5.6 Conclusion

Fragment-based generation and caching of dynamic web pages is widely recognized as an effective technique to counter the scalability and performance challenges caused by the enormous growth of dynamic web content. Most Fragment-based caching solutions rely upon web pages that have been manually fragmented at their respective web sites by the web administrator or the web page designer. However, manual fragmentation of web pages is expensive, error prone, and unscalable.

In this chapter we have presented a novel scheme to automatically detect and flag "interesting" fragments in dynamically generated web pages that are cost-effective cache units. This scheme is based on analysis of the web pages dynamically generated at given web sites with respect to their information sharing behavior, personalization properties and change patterns. Our approach has three unique features. First, we propose a hierarchical and fragment-aware model of the dynamic web pages and a data structure that is compact and effective for fragment detection. Second, we present an efficient algorithm to detect maximal fragments that are shared among multiple documents. Third, we develop an algorithm that effectively detects fragments based on their lifetime and personalization characteristics. We evaluate the proposed scheme through a series of experiments, showing the benefits and costs of the algorithms. We also report our study on the impact of adopting the fragments detected by our system on disk space utilization and network bandwidth consumption.

Our fragment detection scheme is a general technique, and it may be used in conjunction with any fragment-based dynamic content delivery system, including the cooperative EC grid which is appropriately configured to cache dynamic content at fragment granularities.

CHAPTER VI

RELATED WORK

The area of generation and delivery of dynamic web content has received considerable research attention in recent years. Caching has been a popular technique for efficient delivery of dynamic web content. Researchers have designed and implemented various systems for dynamic web content caching. Further, researchers have also attempted to address some of the general challenges that arise in designing caching solutions for dynamic web content delivery. In this chapter, we first survey the area of dynamic content delivery, discussing important research directions that have been pursued in the past. Later in the chapter we discuss prior works that are most relevant to the research presented in this thesis.

6.1 Architectures and Systems for Dynamic Web Content Caching

Designing new architectures, developing new systems, and studying the artifacts of existing architectures and systems constitute a significant portion of the existing research in the area of dynamic web content caching. The design issues and the architectural features of a caching system are, to a large extent, influenced by two factors, namely, the location of the cache(s) within the Internet and the granularity at which content is stored in the caches. Accordingly, the proposed architectures and systems can be classified based on either on the location of the cache(s) or on the granularity of caching.

If the dynamic content caching systems were to be classified based on the location of the caches, the major categories of classification would be (a) Client-side caching, (b) Edge caching, and (c) Server-side caching. Similarly, if we were to adopt granularity of caching as the basis of classification, we obtain three classes, namely (1) Whole-page caching, (2) Fragment caching, and (3) Table or query caching. The works that are most related to the research presented in the thesis belong to the categories of edge caching and fragment

caching, and are discussed in separate sections (see Section 6.4 and Section 6.6

Architectures wherein the caches are located within client's infrastructure are known as client-side caching schemes. Caching dynamic content on the client-side has not received much research attention. One of the few research efforts in this direction was the active cache project [30], which explored the possible ways of developing closer interaction between the client-side caches and the origin server. The central idea of the Active Cache scheme is to shift some of the operations associated with dynamic pages from the origin server to the proxy caches through server-supplied code called *cache applets*. Although the idea of active caching was very novel, it did not become popular because of the security risks involved in executing server-supplied code at client-side caches. Caching dynamic content at the edge of the Internet has become popular in recent years.

In contrast to client-side caching, the field of server-side caching of dynamic content have been well researched [54, 27, 44, 67, 94, 34]. Server-side caching of dynamic content has been a popular technique reducing the load on various servers on the origin site, thereby increasing its throughput and reducing the latencies of requests. In this discussion we consider any cache that is located within the origin-site's infrastructure to be a server-side cache. Server-side caching schemes may be classified into various categories based on location of the caches within the site's infrastructure.

Caching at the interface of the web servers and the external network is referred to as *front-tier caching* or *reverse-proxy caching* [27, 64, 94]. It was one of the earliest server-side caching schemes and it remains very popular even today. Song et al. [94] discuss the design and implementation of a *web server accelerator*, which is a high-performance reverse proxy cache with a collocated load balancer. This work concentrates on the operating system-level issues involved in designing high performance front-tier cache.

Candan et al. [27] present the architectural design of *CachePortal*, a front-tier cache for a database-driven, e-commerce web site. The system is designed to be deployed in a web-farm environment with multiple web servers and collocated application servers, but having a centralized non-replicated database server. The requests arriving at the web site are intercepted by the front-end cache, and any requests that can be satisfied by the data

available in the cache receive responses from the cache, and they do not reach the web farm. If a request reaches the web farm, it is directed to one of the web servers by a load-balancer, which handles the request and generates the response web page. The generated web page flows through the front-end cache, which may store it for responding to future requests.

Middle-tier caching is another popular category of server-side caching mechanisms. The goal of middle-tier caching schemes is to alleviate the load on the application server and the backend databases, thereby improving their throughputs and average response times. In these schemes caching is done at the interface of application server and database server. The cache might be either collocated with the application server or might be located on a separated, dedicated machine. In middle-tier caches, content is generally stored at the granularity of html fragments [44], database-tables [67] or query results [17]. Middle tier caching solutions have also been incorporated into commercial products such as Oracle Application Server [12].

The *Dynamic Content Acceleration (DCA)* solution by Datta et al. [44] is a middle-tier cache which stores HTML fragments. The experiments reported in the paper show that caching at the granularity of fragments provides distinct advantages in terms dynamic page composition costs incurred at the application server.

Luo et al. [67] present the architectural design of *DBCache*, which is a middle-tier cache storing complete database tables. The DBCache is a modified DB2-version 7 database system, which acts as a database cache. The SQL queries issued by the application server are processed using the information stored at the DBCache, and if necessary by contacting the backend databases. DBCache uses the federated features of DB2 for generating a distributed query plan, in case a query references tables that are not cached locally.

Another flavor of server-side caching schemes is the multi-tiered cache architectures, wherein multiple caches are located at various cache-points within the server infrastructure [34, 66]. These architectures are employed in web sites experiencing very heavy loads in which single tiered caching solutions cannot provide acceptable levels of throughput and latency. Challenger et al. [34] discuss the design and implementation of a multi-tiered caching solution, which has been used in web sites for events such as 2000 Summer Olympic

Games, Wimbledon, US Open, and French Open Tennis tournaments.

The architecture consists of multiple front-tier caches (reverse proxy caches), referred to as the origin caches. Other than these front-tier caches, the architecture also has *Point of Distribution* caches that are spread out in the network and are analogous to edge caches. The scheme maintains the freshness and consistency of the cached data through the *data update propagation* algorithms. These algorithms employ the Object Dependency Graph structure to represent the containment relationship between the underlying data and the cached objects and decide when a cached data item is stale and has to be refreshed.

Li et al. [66] extend the CachePortal architecture [27] to a multi-tiered caching scheme. In addition to the front-end cache located at the interface of the web farm and the external Internet, the proposed architecture includes a cache in-between the application servers and the DBMS. This cache, which is termed as a XJDBC cache, stores the results of the database queries issued by the application server. The database queries are intercepted by the XJDBC cache, and if a query cannot be answered by the cached results, the XJDBC cache provides the connectivity functionality between the application server and the databases.

Server-side caching of dynamic web content continues to be a popular area of research. Researchers are studying several variants of these basic approaches for optimizing specific performance parameters, or to address the needs of specific applications.

In terms of the caching granularity, the early systems only supported caching at the granularity of entire pages. These are referred to as *whole-page caches*. However, it was soon realized that whole-page caching is not very well suited for delivering dynamic web content, due to its frequent changing nature and its stringent freshness requirements. This prompted researchers to explore caching at finer granularities, which eventually led to the development of fragment and query caching schemes. Of these two categories, fragment caching has received a higher degree of research attention, and it has also been commercialized. Fragment caching schemes are discussed in detail in Section 6.6.

Query caching schemes, as the name suggests, store results of queries issued to backend databases. Although query caching has been studied in the database community for quite sometime [41, 47], its application to dynamic web content is relatively recent [46, 68, 17]. A

key problem in query caching is to decide whether the data stored in the cache can satisfy an incoming query. This is known as the query-containment problem. While researchers in databases have explored a few solutions to the general query containment problem, most solutions are not very efficient and scalable. To address this concern, query caching schemes typically restrict the class of queries stored in the caches. While Luo and Naughton [68], propose a scheme to cache results of HTML forms-based queries, the template-based query caching scheme [17] is slightly more general in the sense that it permits queries to be originating from HTML forms or from programs using pre-specified statements.

6.2   *Data Freshness Issues in Dynamic Content Caching*

Irrespective of the location of the cache and the granularity of caching, a fundamental and important issue that has to be addressed by all dynamic content caching mechanisms is to maintain the freshness and consistency of cached content. The time-to-live mechanism [52], which was one of the very popular schemes for maintaining consistency of static web content, is not very useful for dynamic content caching. Dynamic web content changes frequently and unpredictably. Therefore, stronger consistency mechanisms are needed to maintain consistency of these documents.

Mechanisms for ensuring data freshness can be broadly classified as push-based schemes and pull-based schemes. In Push-based schemes, the server informs the caches when a data-item changes through update/invalidate messages. Hence these schemes are also called server-driven consistency mechanisms. In Pull-based schemes the caches query the server to determine whether a particular cached data item is fresh. Push-based mechanisms can support stronger consistency requirements in general, but they are also costlier in terms of the load on the server. Most dynamic content caching schemes have adopted some form of push-based mechanism for maintaining consistency of cached documents. Researchers have also investigated schemes that combine push and pull based mechanisms for consistency maintenance. An example of such schemes is the adaptive push-pull strategy proposed by Bhide et al. [22].

Several server-driven consistency mechanisms have been proposed in the literature. *Volume leases* [105] provides a generic framework for push-based consistency mechanisms. In this framework, when a cache stores a document, it requests a *lease* on the document for a finite duration of time. The server registers the lease for the document. The lease is revoked by the server when the document is modified. Researchers have studied various implementations of the basic volume leases mechanism [65, 104, 107].

Yin et al. [103] present a comparative study involving the various variants of the volume leases mechanism, the time-to-live mechanism, and the client-polling mechanism. These mechanisms are compared with respect to a host of performance parameters including read latency, hit rate, message load, and average staleness. The results of the simulation-based experiments lead to some important observations. First, server-driven consistency mechanisms can significantly improve the effectiveness of dynamic content caching. Second, for lease-based mechanisms, it is possible to limit the callback state information maintained at the server without suffering high performance penalties. Fourth, the synchronization duration, which is defined as the time for which the caches and the server are kept synchronized, can have considerable performance impact.

An important challenge in maintaining consistency of dynamic documents is to determine which cached documents need to updated/invalidated when an underlying data item changes. Most dynamic web pages are generated by using information stored in backend databases. Therefore, when a data-item stored in these databases is modified, one or more of the cached dynamic web pages may have to be updated. Establishing the correspondence between the cached dynamic documents and the underlying data-items can have profound performance impact. Challenger et al. [32] use the object dependency graph for maintaining this correspondence information. Cacheportal [27] adopts a bi-layered mechanism for this purpose. A *sniffer* component constantly monitors the client requests and the database queries issued by each request. Through this process, it creates a Query-to-URL mapping, which indicates the URLs that contain the results of a particular query instance. An *invalidator* module on the other hand monitors the changes occurring at the database. This module determines the queries that are affected by the changes at the database. The exact

technique used for determining the affected queries is discussed in section [28]. Once the queries that are affected by the update are determined, the invalidator uses the Query-to-URL mapping to discover the web pages affected by the data update.

The process of generating appropriate update/invalidate messages when an underlying data-item changes is a computationally intensive process. Hence, there is a non-negligible time lag between the instant at which data-item is modified, and the instant when the corresponding web pages are invalidated. During this time period the caches would be serving stale web pages. Labrinidis et al. [63] propose a scheme for intelligently scheduling updates such that the average freshness of all the cached documents is maximized.

A fundamental assumption in all of the above schemes is that the caches are *read-only*. That is the caches store copies of data and serve read requests from clients, but the caches do not modify (or write to) the cached data. All the writes to all data-items happen only at the origin site. This model is termed as *multiple-reader-single-writer* model. Relaxing this assumption, and permitting the caches to also modify the cached copies of data-items leads to the *multiple-readers and writers* model. This mode of caching/replication introduces new data consistency issues.

Traditionally, replicated transactional systems, which follow the multiple readers and writers replication model, ensure strong data consistency through locking. However, supporting strong consistency entails very high performance overheads [45]. This has prompted researchers to propose various optimistic consistency models for applications that can tolerate relaxation in consistency of replicated data to some extent [50, 51, 97]. These systems, under certain circumstances, permit replicas to modify data without obtaining locks, thereby introducing the possibility of inconsistencies and associated rollbacks. Yu and Vahdat [106] note that there is a continuum between the strong and optimistic (relaxed) consistency models. They argue that relaxed consistency models that can provide, and enforce bounds on inconsistencies of replicated data-items are meaningful for a considerable class of applications. Based on this premise, they design the TACT toolkit - a middleware layer which supports relaxed consistency models, and can also enforce bounds on the inconsistencies of replicated data. The work presented in this thesis assumes the single-writer

143

multiple reader model of replication. As a part of our future work, we intend to support multiple readers and writers consistency model in the cooperative EC grid.

6.3   Policies for Cache Management

Another important challenge in the area of web caching is to design and implement policies for effectively managing the resources available at the caches. Previously researchers have studied two types of cache management policies, namely, document placement and document replacement policies. We have explained the similarities and differences between the two categories of cache management polices in Chapter 4.

6.3.1   Document Replacement Policies

Of the two categories of cache management policies, the field of document replacement has received considerable attention from the web caching community. Researchers have proposed several schemes for optimizing various performance parameters. Podliping and Bozszormenyi [79] classify existing cache replacement schemes into five broad categories, namely, recency-based strategies, frequency-based strategies, recency/frequency-based strategies, function-based strategies, and randomized strategies.

Recency-based strategies base their document replacement decisions on how recently various documents in the cache have been last accessed. Most recency-based strategies are variants of the well-known least recently used (LRU) policy, wherein the document for which the time since the last access is the longest is evicted from the cache. All the recency-based strategies are based on the premise of temporal locality, which states that documents which have been accessed in the recent past, have higher chances of being accessed again in the near future. Some examples of recency-based strategies are LRU-threshold [16], Pitkow/Reckers scheme [78], and value-aging [110].

Frequency-based strategies on the other hand make replacement decisions based on frequently the documents in the cache are being accessed. The most basic frequency-based strategy is the least frequently used (LFU) document replacement strategy, which evicts the document whose access frequency is the least. LFU-aging [18], LFU-DA [18], and server-weighted LFU [58] are a few examples for frequency-based cache replacement schemes.

Policies such as LRU* [36] and HYPER-G [16] use both access frequency and access recency information while making document replacement decisions. Thus they may be classified as recency/frequency document replacement strategies.

Function-based document replacement strategies use different functions for quantifying the values of the individual documents that are stored in the cache. In most function-based replacement strategies, the document with the least value is evicted from the cache. Examples of function-based replacement schemes include Greedy-dual size [29], Greedy-dual* [21], and GDSF [18, 38].

In contrast to all the above strategies, the replacement decisions in the randomized strategies are not entirely deterministic. As the name suggests, these schemes incorporate some amount of randomness in their document replacement decisions. The pure random strategy chooses a document from the set of cached documents completely at random and evicts it. While pure random strategy chooses the document to be evicted with equal probability, policies such as harmonic [60], LRU-C, LRU-S [95], and randomized replacement with general value functions [80] use weighted probabilities for selecting the document for removal. The weight may depend on factors such as recency of access, frequency of access, document size, and cost of retrieving the document from the server.

6.3.2 Document Placement Policies

In contrast to the document replacement schemes, the field of document placement has not received much research attention. Of the few research efforts in this area the works by Korupolu and Dahlin [61], and by Wu and Yu [102] are among the prominent ones. We discuss these schemes in Section 6.5.

6.4 Caching on the Edge of the Internet

Computing on the edge of the Internet has gained considerable popularity as technique for efficiently delivering dynamic web content. Companies like Akamai [1], Speedera [13]have successfully commercialized edge computing technology. Most application servers available on the market today such as IBM's Websphere [7] and Oracle's Oracle-10g application server [12] can support edge computing and caching.

The white paper from Akamai Inc. [14] describes the benefits of adopting the edge computing technology for delivering highly dynamic web content, from a commercial perspective. The listed benefits include better scalability, higher reliability, improved performance, and on-demand extensibility.

Prior research efforts have focused on addressing issues like scalability, performance, and security of edge cache networks. One of the critical issues that can have profound implications on the scalability and performance of edge caching is deciding the extent to which the origin site's functionality has to be offloaded to edge cache nodes. At the most basic level, the edge caches might store entire dynamic web pages and serve them upon receiving client requests. Advanced edge caching solutions offload parts of the application logic to the edge caches. The problem is to decide how much of the application logic has to be offloaded to the edge servers?

A typical architecture for systems generating dynamic content is composed of three tiers, namely, database tier, application tier (or business logic tier), and web server tier (or the presentation tier). Of these three layers, the database layer is generally maintained at the origin server for the purposes of simplicity. Chun et al. [108] present a comparative study of 4 different offloading strategies. The four strategies are evaluated with respect to the security concerns, implementation complexities, and performance. The results presented in the paper show that replicating all application components except the database provides best average response time, but only when the database interaction is highly optimized. Further, this strategy is also the easiest to implement. The authors also find that a simple strategy of offloading the functionality of composing web pages from fragments can be very effective in terms of latency and server load reduction.

Rabinovich et al. [83] describes the design, implementation, and performance study of an edge computing system, which they call ACDN. A key feature of this system is the automatic redeployment of application as necessitated by changing request patterns. For this purpose, the ACDN system has a central replicator engine that keeps track of the application replicas, and the loads on the edge caches. When the central replicator detects a load imbalance, it triggers application redeployment.

Gao et al. [49] consider the scenario wherein the database is also replicated at the edge-sites, and the edge servers are capable of updating the database replicas in addition to generating dynamic content. In this scenario, ensuring strong consistency for replicated data items is very costly. Therefore, developing low-cost replication strategies which can satisfy consistency requirements of applications is very attractive. In this work, the authors utilize the semantics of the applications to design an edge caching strategy based on application-specific distributed object. The application considered in the paper is that of an online bookstore. TPC-W, which is an industry-standard benchmark for transactional web workloads, is also based on the same application. The TPC-W benchmark has five distinct kinds of objects. The paper identifies the application semantics associated with each type of object and determines its consistency requirement. In the proposed scheme, the consistency guarantees accorded to different types of objects vary depending upon their application semantics. As an example, for an inventory object the exact count of the inventory is not important, as long as it is ensured the count does not go below zero (or appropriate messages are generated to clients who order an item whose inventory count is below zero). Such relaxations in consistency guarantees reduce the costs of replicating data on edge servers, while ensuring that application semantics are not violated.

All of the above schemes regard individual edge caches as completely independent entities and they do not provide adequate support for cooperation among the edge caches. In contrast, the cooperative edge caching architecture proposed by us supports low-cost cooperation among the edge caches. Although we have assumed the basic level of application offloading, we believe that cooperation among the edge caches can improve the performance of the edge cache network, irrespective of the offloading strategy employed.

6.5 Cooperative Web Caching

Another research area that is related to the work presented in this thesis is that of cooperative proxy caching. The basic research in this field has been focused on designing cache-sharing protocols and cooperative architectures aimed at improving hit rates and document access latencies.

The idea of cooperative proxy caching was first explored in the Harvest system by Chankhunthod et al. [37]. This system promotes a hierarchical cooperation architecture, wherein the caches are organized into a tree-like hierarchy. The document lookups in this system were implemented through remote procedure calls (RPCs). A cache suffering a local miss performs a RPC to all of its parents and siblings. If none of the parents or siblings have the document, then the responsibility of handling the miss is entrusted to the parents, which continue the process of document lookup by initiating RPCs to their parents and siblings.

One of the drawbacks of hierarchical cooperation architecture is that the document lookup messages might need to traverse multiple hops before locating the document. Noting that this might severely affect the latencies experienced by clients, Tewari et al. [98] propose a completely distributed cooperative architecture. In this architecture, the only relationship among the cooperating caches is the peer relationship. When a cache suffers a miss, it contacts its peers to find out whether they have the document. If none of the cooperating peers have the requested document, the cache contacts the origin server to retrieve the document.

Another shortcoming of the harvest scheme was the high cost of RPC mechanism for document lookups. In order to overcome this drawback Internet Cache Protocol (ICP) [53] was designed specifically for communication among web caches. ICP is a lightweight protocol and is implemented on top of UDP. The protocol consists of two types of messages, namely, ICP queries and ICP replies, that are exchanged between neighboring caches for the purpose of document lookups.

Although the costs of individual ICP messages are low when compared to the RPC mechanism, the total number of ICP messages circulated in a group of cooperating proxy caches might be potentially very high, thus placing heavy loads on the network and the cooperating proxies. Several research projects were aimed at addressing this drawback, of which summary cache [48] and adaptive web caching [71] are prominent. These mechanisms optimize the number of ICP messages circulated among cooperating caches. For example in summary cache, each proxy stores a summary of URLs of documents cached at all other

proxies. When the proxy cache suffers a miss, it checks the summaries to see whether the document might be available at any of the other proxies. If the summary indicates that the document might be available in one or more of the other proxies, the cache sends out ICP messages to those proxies. Otherwise, the document is retrieved from the server. Other techniques such as hints, directories, name-based mapping, and hash routing [57, 91, 92, 101, 99] have been proposed to reduce the overall cost of document location process.

Most of the cooperative caching schemes discussed above were designed in the context of proxy caches for storing static web pages. They assumed the Time-to-Live-based mechanism for maintaining their consistency. These schemes cannot be applied directly for caching dynamic web pages, since they neither provide support for stronger consistency mechanisms, nor consider the costs of document updates in their design.

In contrast to the research efforts on cooperative proxy caching schemes Ninan et al. [77] describe *cooperative leases* - a lease-based mechanism for maintaining document consistency in cache groups. They statically hash each document to a cache, which bears the responsibility of maintaining its consistency. However, this work exclusively considers the problem of consistency management in cache groups, and does not study the problems related to document lookups, serving misses and document placement in cache groups. In contrast we believe that all these problems are interrelated, and hence our system adopts a holistic approach in designing the edge cache network based on cache clouds. The research by Shah et al. [93] on dynamic data dissemination among cooperating repositories is also related to the work presented in this thesis. In this system, a dissemination tree is constructed for each data item based on the coherency requirements of the cooperating repositories. The server circulates the updates to the data item through this tree. One of the key differences between this scheme and our work is that, in this scheme the dissemination tree construction is based purely on the coherency requirements of repositories, and it does not take into account the relative positions of the repositories. In our work the cache clouds are constructed based on the network proximities of the caches.

The utility-based document placement scheme proposed in this thesis is related to prior works on placing documents in cooperative web caches [61, 102]. However, most of these

schemes are again targeted for caching static web content, and they do not consider the document consistency costs while making document placement decisions. Korupolu and Dahlin [61] present coordinated document placement and replacement schemes for cooperative caches. This work assumes that the caches are organized into *clusters* and these clusters themselves are arranged in a tree structure. They define a cost function that quantifies the cost incurred by a cache to obtain the document. The proposed document placement scheme aims at minimizing the sum of the cost functions of all documents over all caches in the group. However, this document placement differs from the utility-based placement scheme in various aspects. First, the cost function used in the coordinated placement scheme is exclusively based on document retrieval costs, and it does not take into account the document consistency costs. In contrast, our utility function has four components, and has been designed to estimate the benefit to cost ratio of storing a document at a given cache with respect to different criteria including document freshness maintenance. Second, the coordinated placement scheme assumes for cluster-based hierarchical organization of caches, while the utility scheme can be used for document placements in both hierarchical and distributed cooperation architectures. Third, the decision whether to cache a document at a particular node is made on the fly in the utility scheme, whereas the scheme proposed by Korupolu and Dahlin involves a bottom-up pass of the cluster tree structure to determine the cache where the document has to be stored. Hence, the utility scheme can be considered to be more general and practical than the coordinated placement scheme.

The adaptive replication (ACR) scheme proposed by Wu and Yu [102] augments the work on hash-based routing schemes by permitting documents to be placed in more than one cache. In the ACR scheme, the numbers of copies existing within the cache group is purely based on the relative popularities of the documents. This scheme does not take into account the consistency costs or the storage costs when making placement decisions, and hence it cannot be directly applied in caches storing dynamic web content.

6.6 Fragment-based Caching of Dynamic Web Content

Fragment-based publishing, delivery and caching have been popular techniques to address challenges caused by the enormous increase of the dynamic web content over the past few years [33, 43]. One of the first works in this field was the fragment-based dynamic web data publishing system developed by Challenger et al. at IBM [33]. This approach simplifies designing web sites by allowing recursive embeddings of fragments. This permits reuse of generated fragments, thereby avoiding unnecessary regeneration of the same information multiple times.

Another early work on fragment-based caching was by Datta et al. [43]. The work is motivated by the observation that the dynamic natures of web pages are exhibited along two orthogonal dimensions, namely, a web page's layout, and its content. Based on this premise they propose a fragment-based web caching system, in which the dynamic content is cached at proxy caches, whereas the layout of the web page is fetched from the server on each access to the web page. This approach minimizes the load on the network connecting the server and the proxy cache by transferring only the layout and the fragments that are not present in the cache. However, the proposed scheme has two significant drawbacks. First, it does not address the problem of ameliorating the load on the origin server. Second it entrusts the responsibility to the origin server, further increasing its load.

The concept of fragment-based web page composition has also been applied for providing efficient QoS support and security mechanisms for web documents. Mohapatra and Chen [74] propose a system called WebGraph, which is a graphical representation of the containment relationship among weblets, which are analogous to fragments. In the proposed system the QoS and security attributes can be specified at the granularity of a weblet. The system decides whether to include a particular weblet in a web page based on several factors including the QoS specifications of the weblet, the desired QoS guarantees, and the present system conditions such as the current loads on the server and the network.

Successful commercialization of fragment-based solutions requires industry-wide standards for fragment-caching. Efforts to standardize fragment-based caching resulted in Edge

Side Includes [4], which is a XML-based markup language to define web page components for page assembly at the edge caches. ESI provides mechanisms for specifying the cacheability properties at fragment level. ESI can be used by content providers to specify the fragments to be included in a web page. Caches constructs web pages by inserting the specified fragments either from the cache, or by fetching them from the origin server. The key functionalities provided by the ESI include *Inclusion*, *Conditional inclusion/exclusion*, *Exception handling* and *Fragment invalidation*.

Naaman et al. [75] present analytical and simulation-based studies to compare ESI and delta-encoding, finding that ESI has potential performance advantages due to its ability to deliver only changing fragments.

While assembling fragments at edge caches has been shown to be effective in reducing the loads on the backbone network and origin servers, it does not ameliorate the load on the network links connecting the end clients to the edge caches (the so called last-mile links). However, reducing the load on the last-mile links is very important, especially for clients that are connected through dial-up connections. Rabinovich et al. [84] address this problem by taking the concept of edge-side includes one step further. They propose the *Client Side Includes* scheme, wherein the composition of web pages from fragments is done at the client, rather than at the reverse proxy. The CSI mechanism enables the browsers to assemble the web pages from the individual fragments. The paper also describes a JavaScript-based implementation of the CSI mechanism.

In addition to the fragment-based solutions discussed above, researchers have explored related research ideas in different contexts. The work by Wills and Mikhailov [100] bears some similarity to the automatic fragment-detection scheme presented in Chapter 5. They propose to use change characteristics of objects embedded in web pages, and interrelationships among the web objects for reducing the consistency maintenance overhead at web caches. Researchers have also explored utilizing structural properties of web pages for efficient delivery of dynamic content. Chan and Woo [35] propose to use the structural similarity existing among various pages of a single site to efficiently delta-encode multiple web pages over time.

The work of Bar-Yossef and Rajagopalan [20] is related to our research on automated fragment detection, although the authors were addressing a different problem. They discuss the problem of template detection through discovery of *pagelets* in the web pages. However, our work differs from the work on template detection both in context and content. First, the work on template detection is aimed towards improving the precision of search algorithms. Our work is aimed at detecting fragments that are most beneficial to caching and content generation. Second, the syntactic definition of a pagelet in their paper is based on the number of hyperlinks in the HTML parse tree elements, which is very different from our working definition of a candidate fragment provided in Section 5.1. Further, their definition of pagelets forbids recursion. In contrast we permit embedded fragments. Finally, our system has two algorithms: one to detect Shared fragments and another to detect L-P fragments. Both of these detect embedded fragments.

Fragment detection also has similarities to comparing two similar structured documents. While some tools look at longest matching subsequences, it is possible to use signatures of subtrees to identify pieces of a document that have moved rather than been deleted. An example of this approach is the Xyleme XML diff application [39].

There has been significant work in identifying web objects that are identical, either at the granularity of entire pages or images [19, 59, 72] or pieces of pages [90], using MD5 or SHA-1 hashes to detect and eliminate redundant data storage and transfer. While the motivations of these researches are similar to that of the shared fragment detection algorithm, they are more restrictive in the sense that they work on full HTML pages and can only detect and eliminate pages (or byte-blocks) which are exact replicas. Although these techniques have the potential to reduce transfer sizes, decomposing web pages into separately cached fragments accomplishes similar reductions in size without the need for explicit version management.

6.7 Conclusions

In the past few years the field of caching and delivering dynamic content has been a popular area of research. Several projects have studied the research issues in this area, and

have proposed various architectures, mechanisms and techniques to address the challenges involved in delivering dynamic web content to the users.

In this chapter we have provided a brief overview of the important research milestones in the general area of generation, caching, and delivery of web content. We have focused on the prior research efforts that are most relevant to the work presented in this thesis. We have explained the similarities and differences between our research and the important earlier works in this area, thereby identifying the unique features of the work presented in this thesis.

CHAPTER VII

CONCLUSIONS

The tremendous growth of dynamic web content in recent years has introduced new challenges to the scalability and performance of the World Wide Web. There is a heavy demand for technologies for efficient generation and timely delivery of fresh dynamic content.

In this thesis we have studied the various challenges involved in efficiently serving dynamic content to the end-users. We have investigated several techniques to enhance the efficiency and scalability of the dynamic web content delivery process. This dissertation makes two major technical contributions.

We have designed cooperative edge cache grid (cooperative EC grid, for short) - a novel cooperative edge cache network for delivering dynamic web content. The cooperative EC grid is, to our knowledge, the first edge cache network that has been specifically designed to harness the power of low-cost cooperation to efficiently, scalably, and reliably deliver highly dynamic web content with varying server update rates to the clients. Through the design of the cooperative EC grid, we demonstrate that efficient and effective cooperation can considerably enhance the benefits provided by edge cache networks.

Cooperative EC grid incorporates several innovative techniques and mechanisms that are aimed at promoting efficient and effective edge cache cooperation. The proposed EC grid architecture is based on the concept of cache clouds, which forms the fundamental framework of cooperation among the edge caches. We have designed the selective landmarks-based server distance sensitive clustering scheme (SDS scheme) for creating cache clouds in the cooperative EC grid by accurately clustering its caches. The design architecture of individual cache clouds includes dynamic hashing-based mechanisms for lookups and updates, which not only distribute the lookup and update loads among all the caches of a cloud, but also are resilient to failures of individual caches. Further, we have developed a utility-based scheme for strategically placing documents among caches of a cache-cloud,

so that available system resources are optimally utilized. We have performed an extensive experimental study to evaluate the costs and benefits of the proposed architecture and techniques, the results of which indicate that these schemes can enhance the performance of edge cache networks on critical parameters like latency, network load, hit-rates, and client-latencies.

The second key novelty of this dissertation research is the design and development of a framework for automatically identifying cache-effective fragments in dynamic web pages. This is the first, and remains to be one of the very few schemes proposed for automatically fragmenting dynamic web pages. This thesis hereby demonstrates the feasibility automating the fragment detection process in dynamic web pages.

While fragment-based caching has been shown to provide considerable benefits for delivering dynamic web content, most existing fragment-based techniques rely upon manual web-page fragmentation, which is costly, error-prone, and unscalable. Our automatic fragment detection scheme exhibits three unique features. First, it includes a fragment-aware data-structure to model the dynamic web pages, called the augmented fragment tree (AF tree). Second, we present an efficient algorithm to detect maximal fragments that are shared among multiple documents. Third, we develop a practical algorithm that effectively detects fragments based on their lifetime and personalization characteristics. We have evaluated the proposed fragment detection scheme through a series of experiments, showing the effectiveness and costs of our approach. Further, we have also presented our experimental study on the effects of adopting the fragments detected by our system on the web caches and the origin servers.

Although the main focus of this dissertation is the efficient delivery of dynamic web content, many of the schemes that we have proposed in this thesis can be adopted in designing various types of distributed systems and applications. For example, the dynamic hashing technique can be used to design efficient document-lookup mechanisms in peer-to-peer systems and other overlay networks. Similarly, our utility-based document placement scheme can be used as a basis for designing data placement strategies for peer-to-peer networks, grids, and mobile ad-hoc networks. Further, the automatic fragment detection

scheme can be adopted for discovering data/document fragments in various applications.

7.1   Open Problems and Future Work

While this thesis presents techniques to solve some of the important problems in the area of dynamic content delivery, several more challenges remain to be addressed. In fact, our work in this field opens up many research issues. In this section we provide a brief overview of the open problems in this area and future research directions that we intend to pursue.

In the context of the cooperative EC grid, a key research challenge that remains to be addressed is that of deciding the appropriate number of edge caches in the EC grid, and the locations where these edge caches have to be placed in order to obtain optimal performance. Although researchers have proposed a few techniques for solving similar problems in the context of content distribution networks, they are mostly designed for delivering multimedia content. Hence, these may not be directly applicable for cooperative EC grid whose main focus is serving dynamic web content. We believe that it is important to design cache placement techniques that consider the issues that are specific to dynamic content delivery.

The SDS scheme for cache cloud formation presented in this thesis considered two important factors while creating cache clouds, namely, proximity among the caches and the distances between the caches and the server. Another factor that impacts the effectiveness of cooperation in the EC grid is the document access patterns at various caches. Clearly it is advantageous to have some amount of diversity in the access patterns of the belonging to a cloud. However, creating clouds from caches having completely disparate document access patterns may also be counter-productive. Cloud formation mechanisms that consider these access patterns trade-offs along with the other two further enhance the effectiveness of cooperation in edge cache networks.

In the current cooperative EC grid design, the cooperation is limited to caches within a cache cloud. We believe that incorporating mechanisms for inter-cloud cooperation can enhance the resilience of the cooperative EC grid against massive failures such as network partitions. Similarly, the cooperation infrastructure of the cooperative EC grid can also be utilized to design low-cost schemes for revalidating documents (to ensure consistency)

stored at a cache that has just started re-functioning after a failure.

The utility function for document placement presented in this thesis comprises of four components, of which we have studies the disk-space component in-detail. We think that it is important to study other components of the utility-based placement scheme as well as to design cooperative replacement policies for effectively managing the caches of the EC grid.

The current design of the cooperative EC grid makes two assumptions. First, the caching is done at either page granularity or at fragment granularity. Second the individual caches are read-only, and all the document modifications occur only at the origin server. Extending the cooperative EC grid to support caching of database query results presents interesting research challenges. Similarly relaxing the read-only restriction on the caches and permitting them to modify the data (the multiple readers and writers model) brings up important research issues.

The automatic fragment detection scheme presented in this thesis is based on the analysis of web pages with respect to their information sharing characteristics and their change patterns. An alternate approach would be to analyze the application programs and detect fragments based on their data access patterns. Studying this approach, comparing it with our scheme, and to evolve mechanisms that combine these to fragment detection approaches are interesting research problems.

Dynamic web content delivery systems like the cooperative EC grids carry large amounts of private information such as personal information and financial data. These systems can be targets of various kinds of malicious attacks. Ensuring privacy of security and privacy of data as well as protecting the content delivery infrastructure from different types of attacks are very important problems. We think that considerable research efforts are required in this direction to satisfactorily address the security and privacy concerns.

Finally, a growing population of users is accessing the web through resource-constrained devices such as PDA's and cell phones. We believe that caching in general and cooperative caching in particular can play very important roles in efficiently delivering dynamic content to these users. As a part of our future work we intend to devise specialized caching mechanisms for resource-constrained clients as well as completely decentralized systems such as

mobile ad-hoc networks.

To summarize, we believe that the work presented in this thesis has brought several important issues to the fore and addressing these issues can significantly improve the performance, scalability and security of dynamic web content.

APPENDIX A

ANALYTICAL STUDY OF THE DYNAMIC HASHING MECHANISM

In this chapter we analytically study the load balancing properties of the static and the dynamic hashing mechanisms presented in Chapter 4. We consider an edge cache cloud with N edge caches. Suppose the number of unique documents being served by these caches during a given time period is represented as $DocCount$. We assume that the cumulative load due to updates and lookups is distributed across the $DocCount$ documents according to the popular Zipf model. This assumption is supported by previous studies in the area of web caching [23].

Without loss of generality, we rank the documents in decreasing order of the load induced by them, and represent the document at rank i as D_i. As the load induced by the documents are distributed according to the Zipf-model, the load generated by D_i is proportional to $\frac{1}{i^\alpha}$, where α is the Zipf constant. Analysis of actual workloads have indicated that α is close to 1.00. Let the cumulative load due to all documents in the time period is represented as $TotalLoad$. Now the load generated by D_i is given by $Load(D_i) = \frac{TotalLoad}{\sum_{k=1}^{D} ocCount \frac{1}{k^\alpha}} \times \frac{1}{i^\alpha}$.

In order to obtain a closed form expression for the $Load(D_i)$, we employ an approximation technique. When $\alpha \approx 1$, the series $\frac{1}{1^\alpha} + \frac{1}{2^\alpha} + \cdots + \frac{1}{DocCount^\alpha}$ can be approximated by the *harmonic series* $1 + \frac{1}{2} + \cdots + \frac{1}{DocCount}$. The term $\sum_{k=1}^{DocCount} \frac{1}{k^\alpha}$ would be approximated by $H_{DocCount}$, where H_k represents the kth harmonic number. Therefore, we obtain $Load(D_i) = \frac{TotalLoad}{H_{DocCount}} \times \frac{1}{i}$. When $DocCount$ is large enough the harmonic number $H_{DocCount}$ can be approximated by $ln(DocCount) + \gamma$, where ln represents the natural logarithm function and γ is the Euler-Mascheroni constant whose value is approximately 0.57721. Substituting $H_{DocCount}$ with $ln(DocCount) + \gamma$, we get

$$Load(D_i) = \frac{TotalLoad}{(ln(DocCount) + \gamma) \times i} \tag{3}$$

We now analyze the worst-case scenario for both static and dynamic hashing schemes. First we consider the static hashing scheme. A reasonable hashing function distributes the documents approximately evenly across the beacon points in the cache cloud. However, this does not always ensure good load balancing, as some caches might receive a large portion of hot documents, while others might receive very few of them. The worst case for the static hashing would be when the documents inducing the heaviest load, namely the documents $\{D_1, D_2, \ldots, D_{\frac{DocCount}{N}}\}$ all get hashed to the same beacon point. This beacon point would encounter the maximum cumulative load due to lookups and updates. The load being encountered by this beacon point is given by:

$$MaxCacheLoad_{StaticHashing} \quad = \sum_{i=1}^{\frac{DocCount}{N}} Load(D_i) \tag{4}$$

$$= \frac{TotalLoad}{(ln(DocCount+\gamma)} \sum_{i=1}^{\frac{DocCount}{N}} \frac{1}{i} \tag{5}$$

The term $\sum_{i=1}^{\frac{DocCount}{N}} \frac{1}{i}$ is equal to $H_{\frac{DocCount}{N}}$ which can be approximated by $ln(\frac{DocCount}{N}) + \gamma$. Therefore

$$MaxCacheLoad_{StaticHashing} = TotalLoad \times \frac{ln(DocCount) - ln(N) + \gamma}{ln(DocCount) + \gamma} \tag{6}$$

Let $MaxLoadRatio$ represent the ratio of the load encountered by the most heavily loaded beacon point to the mean of the loads on all beacon points. For the static hashing scheme the MaxLoadRatio is given by:

$$MaxLoadRatio_{StaticHashing} \quad = \frac{TotalLoad \times \frac{ln(DocCount) - ln(N) + \gamma}{ln(DocCount) + \gamma}}{\frac{TotalLoad}{N}} \tag{7}$$

$$= N \times \frac{ln(DocCount) - ln(N) + \gamma}{ln(DocCount) + \gamma} \tag{8}$$

Next, we analyze the worst-case scenario of the dynamic hashing scheme. Consider a dynamic hashing scheme in which each beacon ring has K beacon points. Therefore the edge cache cloud has $M = \frac{N}{K}$ beacon rings. In the dynamic hashing scheme the worst case scenario occurs when the documents with ranks 1 through $\frac{DocCount}{M}$ (the $\frac{DocCount}{M}$ documents inducing the heaviest loads) maps to the same beacon ring. Therefore the maximum load

experienced by any beacon ring within the edge cache cloud is given by

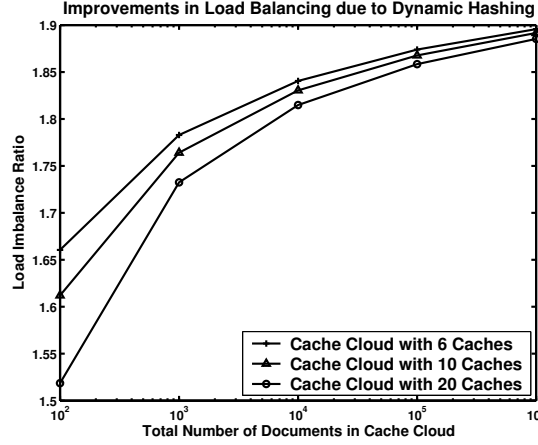$$MaxRingLoad_{DynamicHashing} = \sum_{i=1}^{\frac{DocCount}{M}} Load(D_i) \qquad (9)$$

Figure 52: Comparing Load Balancing Properties of Static and Dynamic Hashing Schemes

Applying similar approximations as in the static hashing scenario, we get:

$$MaxRingLoad_{DynamicHashing} \quad = TotalLoad \times \frac{ln(DocCount)-ln(M)+\gamma}{ln(DocCount)+\gamma} \qquad (10)$$

$$= TotalLoad \times \frac{ln(DocCount)-ln(N)+ln(K)}{ln(DocCount)+\gamma} \qquad (11)$$

The sub-range determination process occurring at the end of each cycle distributes this load approximately equally among all beacon points in the beacon ring. Therefore, the maximum load experienced by any beacon point in the worst case would be

$$MaxCacheLoad_{DynamicHashing} = TotalLoad \times \frac{ln(DocCount) - ln(N) + ln(K) + \gamma}{(ln(DocCount) + \gamma) \times K} \qquad (12)$$

The $MaxLoadRatio$ for the dynamic hashing scheme would therefore be:

$$MaxLoadRatio_{DynamicHashing} = N \times \frac{ln(DocCount) - ln(N) + ln(K) + \gamma}{(ln(DocCount) + \gamma) \times K} \qquad (13)$$

We compare the worst-case load-balancing properties of the static and the dynamic hashing scenarios by considering the ratio $\frac{MaxLoadRatio_{StaticHashing}}{MaxLoadRatio_{DynamicHashing}}$. The ratio

162

$\frac{MaxLoadRatio_{StaticHashing}}{MaxLoadRatio_{DynamicHashing}}$ is an indicator to the load imbalance occurring in static hashing when compared with that of dynamic hashing. Hence we call it the *Load Imbalance Ratio* (represented as *LoadImbalanceRatio*).

$$LoadImbalanceRatio = K \times \frac{ln(DocCount) - ln(N) + \gamma}{ln(DocCount) - ln(N) + ln(K) + \gamma} \tag{14}$$

If *LoadImbalanceRatio* is greater than 1.00, it indicates that the dynamic hashing scheme provides better load balancing, and vice-versa. In order to provide an insight into the behavior of the above function we plot it for different values of *DocCount* in Figure A. In this graph, M is kept constant at 2, which means each beacon ring contains 2 caches. The graph shows that for a cache cloud with 10 caches, the load encountered by the most heavily loaded beacon point in the static hashing scheme is 1.81 times the load encountered by the most heavily loaded beacon point in the dynamic hashing scheme, when *DocCount* is 10,000. We note that this ratio increases with increasing values of *DocCount*. This analysis shows that dynamic hashing scheme exhibits very good load-balancing properties, even when the update and lookup load distributions are highly skewed.

APPENDIX B

ANALYTICAL STUDY OF FRAGMENT-BASED CACHING

In this chapter we present an analytical study of the benefits of caching dynamic web content at the granularity of web page fragments. Although several researchers have studied various aspects of fragment-based dynamic content caching, none, to our knowledge, have tried to analytically quantify the benefits of caching web content at the granularity of individual fragments. In this chapter we specifically analyze the following two aspects of fragment-based caching:

1. Effect of fragment-based caching on the efficiency of disk-space utilization in web caches.

2. Improvements in server load obtained by caching at fragment granularity

B.1   Effect of Fragments on Disk-space Efficiency

In this section we mathematically analyze the improvements provided by fragment-based caching on the efficiency of disk-space utilization at web caches. We compare fragment-based caching to a scheme wherein the caching is done at the granularity of entire web pages, which we refer to as whole-page caching. Specifically, we show that the number of documents available in fragment cache is an order of magnitude higher than the number of documents present in a corresponding whole page cache of identical size. The number of documents available at a cache is an important metric because the hit rate of a cache is directly dependent on the number of documents available in the cache.

In order to simplify the analysis we make certain assumptions. We consider a cache with disk capacity of DC_C bytes. We assume that web pages contain Nf number of fragments on an average. Let the average size of each fragment be Sz bytes. We also assume that

164

a fragment is included in Fp web pages on an average (the methodology to compute the number of web pages including a particular fragment is discussed later part of this section).

First, we consider the case when the server and the cache support only whole-page caching. As each fragment is of size Sz bytes and each page has Nf fragments on an average, the average size of each webpage would be $Sz \times Nf$. Therefore the number of pages that are contained in a whole-page cache of capacity DC_C bytes is given by

$$NumPages_{WC} = \frac{DC_C}{Nf \times Sz} \tag{15}$$

Now we consider a fragment-based caching scheme. We note that in a fragment-based cache, each fragment is stored only once irrespective of how many pages it is contained in. Therefore the number of fragments that can be stored in the cache of size DC_C is given by

$$NumFrags_{FC} = \frac{DC_C}{Sz} \tag{16}$$

As we have assumed that each contains Nf fragments on average, the number of web pages corresponding to the $NumFrags_{FC}$ stored in fragment cache is $\frac{NumFrags_{FC} \times Fp}{Nf}$. Therefore, the number of pages present in a cache of size DC_C supporting fragment caching is:

$$NumPages_{FC} = \frac{DC_C \times Fp}{Sz \times Nf} \tag{17}$$

The percentage improvement in the number of pages that can be stored in fragment cache over a whole page cache of equal size is given by:

$$
\begin{aligned}
PercImprovment &= \frac{(Numpages_{FC} - NumPages_{WC})}{NumPages_{WC}} \times 100 \tag{18} \\
&= \frac{(Fp - 1) \times \frac{DC_C}{Sz \times Nf}}{\frac{DC_C}{Sz \times Nf}} \times 100 \tag{19} \\
&= (Fp - 1) \times 100 \tag{20}
\end{aligned}
$$

This equation indicates that percentage improvement in the number of pages in the cache is directly and linearly dependent on the average number of pages each fragment is contained in.

Many studies in the past have shown that hit rate of a cache is directly dependent on the average number of documents it contains. Though the exact relationship between the number of pages available in the cache and the hit rate is dependent on many factors studies have shown that it resembles a logarithmic function. As we know that $NumPages_{FC} = Fp \times NumPages_{WC}$, the relationship between the hit rate of fragment-based cache and a corresponding whole page cache can be approximated as $HitRate_{FC} = HitRate_{WC} + \alpha \times log(Fp)$, where $HitRate_{FC}$ and $HitRate_{WC}$ denote the hit rates of fragment cache and whole-page cache respectively. This indicates that as Fp increases, the improvements in hit rates provided by the fragment-based caching scheme also grows.

We now briefly describe the method to count the number of pages that contain a particular fragment. We will call this as the *Page-Count* of the fragment. It has to be remembered that the fragments can be recursively embedded. Hence the method to compute the Page-Count is also recursive.

Consider a fragment Fg_A. Let us define $Contain(Fg_A)$ as the set of all fragments that include Fg_A. The page count of Fg_A can be calculated recursively as:

$$\text{Page-Count } (Fg_A) = \begin{cases} 1 & \textit{if } Fg_A \textit{ is a complete web page} \\ \sum_{Fg_B \in Contain(Fg_A)} \text{Page-Count}(Fg_B) \\ \textit{otherwise} \end{cases} \tag{21}$$

The above equation provides a basis for computing the average number of web pages that fragments are contained in.

B.2 Impact of Fragment-based Caching on Server Load

We now study the impact of fragment based page composition on the load experienced by the server. As in the previous section we make some assumptions to simplify the analysis. Consider an arbitrary web page WP in a web server. Let the web page be composed of N fragments, $\{Frag_1, Frag_2, ...Frag_N\}$, each of which has distinct lifetime characteristics. That is these fragments remain fresh for different amounts of times, after which they need to be refreshed. We assume that constructing each fragment places one unit load on the origin server. We model the invalidation of each fragment to follow a *Poisson process.*

Let the fragment $Frag_i$ is invalidated at a rate of μ_i per unit time. This means that the time between two invalidations of a single fragment is exponentially distributed with mean $frac1\mu_i$. As the fragments have distinct lifetime characteristics, the invalidation processes of fragments are assumed to be independent of one another. Further, let us assume that generating any fragment places one unit load on the origin server. Therefore generating the entire web page (in whole-page caching) places a load of N units on the server.

Let a cache receive requests for the web page WP according to a Poisson process with the rate of λ_{WP} requests per unit time. Now let us consider the case wherein the server and cache only supports whole-page caching. In this scenario, the web page WP becomes stale when any one of the fragments gets stale. As the invalidations of each of the fragments is a poisson process with rates $\{\mu_i \mid 1 \leq i \leq N\}$, the invalidation of the page itself is poisson. Further the theory on Poisson process suggests that the rate of invalidation of the web page WP would be:

$$\mu_{WP} = \sum_{i=1}^{N} \mu_i \qquad (22)$$

As we have assumed that requests for web page WP arrives at the cache according to a Poisson process with rate λ_{WP}, from the theory on Poisson processes, we can show that the requests that are sent from the cache to the server (due to document expiry at the cache) follow a poisson process with rate of $\frac{\lambda_{WP} \times \mu_{WP}}{\lambda_{WP} + \mu_{WP}}$. Let us call this rate as $ReqRate_{WP}^{Server}$.

Now, each time the request for the document WP arrives at the server, the page has to be regenerated. Therefore, the load on the server per unit time due to the web page WP, in whole page caching is $N \times ReqRate_{WP}^{Server}$, which yields:

$$TotalCost_{WP}^{WholeCache} = \frac{N \times (\sum_{i=1}^{N} \mu_i) \times \lambda_{WP}}{(\sum_{i=1}^{N} \mu_i) + \lambda_{WP}} \qquad (23)$$

For further simplification, let us assume that all fragments are invalidated at the same rate, represented as μ. Now the above equation becomes:

$$TotalCost_{WP}^{WholeCache} = \frac{N^2 \times \mu \times \lambda_{WP}}{N \times \mu + \lambda_{WP}} \qquad (24)$$

Now let us consider the scenario wherein the caches and the server support fragment-based caching. Let us consider the web page WP with the same fragments as in the previous

167

scenario. However, in this case the cache stores each fragment separately and composes the page upon request arrival. Therefore, when a fragment is invalidated in the cache, the next request to WP causes the cache to contact the server and obtain the invalidated fragment. As in the previous case, let us assume that $Frag_k$ is invalidated in accordance with a poisson process with rate μ_k per unit time and request rate to page WP is poisson process at rate of λ_{WP}. The request rate to each of the component fragments of page WP is the same as the request rate of the parent fragment (λ_{WP}). We can show that the number of requests sent to the server for $Frag_k$ follows a poisson process with rate $ReqRate_k^{Server}$ as given by:

$$ReqRate_k^{Server} = \frac{\mu_k \times \lambda_{WP}}{\mu_k + \lambda_{WP}} \tag{25}$$

Therefore, the total number of requests reaching the server due to all the fragments in web page WP is still a poisson process with rate $\sum k = 1^N ReqRate_k^{Server}$. Let us call this $ReqRate_{WPFrag}^{Server}$.

$$ReqRate_{WPFrag}Server = \sum_{k=1}^{N} \frac{\mu_k \times \lambda_{WP}}{\mu_k + \lambda_{WP}} \tag{26}$$

As in the previous scenario, to simplify the analysis we assume the invalidation rates of all the fragments are the same and are equal to μ. Then the above equation reduces to

$$ReqRate_{WPFrag}Server = \frac{N \times \mu \times \lambda_{WP}}{\mu + \lambda_{WP}} \tag{27}$$
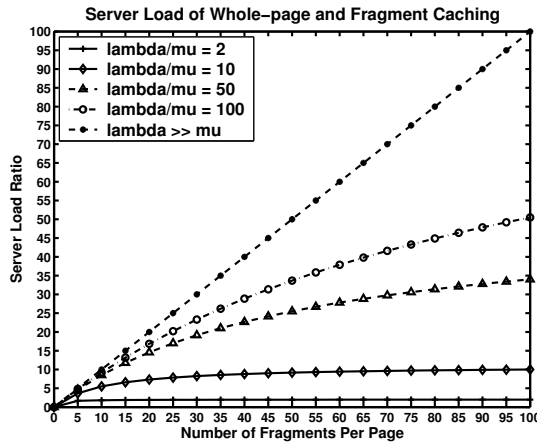

Figure 53: Server Load Patterns in Whole Page and Fragment Caching

However, in fragment-based caching the server needs to only regenerate the fragment for which it received the request. Therefore the total server load per unit time due to requests for all the fragments in web page WP is given by

$$TotalCost_{WP}^{FragCache} = \frac{N \times \mu \times \lambda_{WP}}{\mu + \lambda_{WP}} \tag{28}$$

To compare the server load in fragment-based caching with that of whole-page caching let us consider the ratio $\frac{TotalCost_{WP}^{WholeCache}}{TotalCost_{WP}^{FragCache}}$. We shall refer to it as $CostRatio$.

$$CostRatio = \frac{N \times (\mu + \lambda_{WP})}{(N \times \mu + \lambda_{WP})} \tag{29}$$

Let us now see how the $CostRatio$ evolves with respect to $\frac{\lambda_{WP}}{\mu}$. If $\lambda_{WP} = \mu$, then $CostRatio_{(\lambda_{WP}=\mu)} = \frac{2 \times N}{N+1}$. If $\lambda_{WP} = l \times \mu$ then $CostRatio_{(\lambda_{WP}=\mu)} = \frac{(l+1) \times N}{N+l}$. If $\lambda_{WP} \gg \mu$, then the equation reduces to $CostRatio_{(\lambda_{WP}=\infty)} = N$. This shows that when the access rate of web page is much larger than the invalidation rates, the load induced by whole page caching is N times the load induced by an equivalent fragment cache. The graph in Figure B.2 shows the pattern of $CostRatio$ at various values of $\frac{\lambda_{WP}}{\mu}$ and N. The graph in Figure B.2 shows the pattern of $TotalCost$ at various values of $\frac{\lambda_{WP}}{\mu}$ and N.

Finally, we note that our experimental results discussed in Figure 47 substantiates the above mathematical analysis.

REFERENCES

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

[2] "BEA Systems Incorporated: BEA Weblogic Server." http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic/server.

[3] "Document Object Model - W3C Recommendation." http://www.w3.org/DOM.

[4] "Edge Side Includes - Standard Specification." http://www.esi.org.

[5] "HTML TIDY." http://www.w3.org/People/Raggett/tidy/.

[6] "IBM WebSphere Application Server Product Overview." http://www-306.ibm.com/software/webservers/appserv/was/.

[7] "IBM WebSphere Edge Server." http://www-3.ibm.com/software/webservers/edgeserver/.

[8] "Internet Statistics via WhoIs Source." http://www.whois.sc/internet-statistics/.

[9] "Internet World Usage Statistics." http://www.internetworldstats.com/stats.htm.

[10] "Nua Internet Survey." http://www.nua.net/surveys/how_many_online/world.html.

[11] "OCLC - Web Growth Characterization." http://www.oclc.org/research/projects/archive/wcp/stats/size.htm.

[12] "Oracle Corporation: Application Server 10g Release 2 White Paper." http://www.oracle.com/technology/products/ias/pdf/1012_nf_paper.pdf.

[13] "Speedera Networks, Incorporated." http://www.speedera.com/.

[14] "Turbo-Charging Dynamic Web Sites with Akamai EdgeSuite." http://www.akamai.com/en/resources/pdf/whitepapers/Akamai_EdgeSuite_Turbo--charging_Websites.pdf.

[15] "Zooknic Internet Geography Project." http://www.zooknic.com/index.html.

[16] ABRAMS, M., STANDRIDGE, C. R., ABDULLA, G., FOX, E. A., and WILLIAMS, S., "Removal Policies in Network Caches for World-Wide Web Documents," in *Proceedings of the ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, August 1996.

[17] AMIRI, K., PARK, S., TEWARI, R., and PADMANABHAN, S., "A Self Managing Data Cache for Edge-Of-Network Web Applications," in *ACM CIKM International Conference on Information and Knowledge Management*, November 2002.

[18] ARLITT, M. F., CHERKASOVA, L., DILLEY, J., FRIEDRICH, R., and JIN, T., "Evaluating Content Management Techniques for Web Proxy Caches," *SIGMETRICS Performance Evaluation Review*, vol. 27, March 2000.

[19] BAHN, H., LEE, H., NOH, S. H., MIN, S. L., and KOH, K., "Replica-Aware Caching for Web Proxies," *Computer Communications*, vol. 25, no. 3, 2002.

[20] BAR-YOSSEF, Z. and RAJAGOPALAN, S., "Template Detection via Data Mining and its Applications," in *Proceedings of the 11th International World Wide Web Conference*, May 2002.

[21] BESTAVROS, A. and JIN, S., "Popularity-Aware Greedy Dual-Size Web Proxy Caching Algorithms," in *Proceedings of the 20th International Conference on Distributed Computing Systems(ICDCS-2000)*, April 2000.

[22] BHIDE, M., DEOLASSE, P., KATKER, A., PANCHBUDHE, A., RAMAMRITHAM, K., and SHENOY, P., "Adaptive Push-Pull: Disseminating Dynamic Web Data," *IEEE Transactions on Computers*, vol. 51, June 2002.

[23] BRESALU, L., CAO, P., FAN, L., FAN, L., PHILLIPS, G., and SHENKER, S., "Web Caching and Zipf-like Distributions: Evidence and Implications," in *Proceedings of IEEE INFOCOM 1999*, March 1999.

[24] BRODER, A., "On resemblance and Containment of Documents," in *Proceedings of SEQUENCES-97*, 1997.

[25] BRODER, A., GLASSMAN, S. C., MANASSE, M. S., and ZWEIG, G., "Syntactic Clustering of the Web," in *Proceedings of the 6th International World Wide Web Conference*, April 1997.

[26] BUTTLER, D. and LIU, L., "A Fully Automated Object Extraction System for the World Wide Web," in *Proceedings of the 21st International Conference on Distributed Computing Systems*, 2001.

[27] CANDAN, K. S., LI, W.-S., LUO, Q., HSIUNG, W.-P., and AGRAWAL, D., "Enabling Dynamic Content Caching for Database-Driven Web Sites," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 2001.

[28] CANDAN, K. S., AGRAWAL, D., LI, W.-S., PO, O., and HSIUNG, W.-P., "View Invalidation for Dynamic Content Caching in Multitiered Architectures," in *Proceedings of Proceedings of the 28th International Conference on Very Large Data Bases (VLDB-2002)*, August 2002.

[29] CAO, P. and IRANI, S., "Cost-Aware WWW Proxy Caching Algorithms," in *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 1997.

[30] CAO, P., ZHANG, J., and BEACH, K., "Active Cache: Caching Dynamic Contents on the Web," *Distributed Systems Engineering*, vol. 6, March 1999.

[31] CASTRO, M., DRUSCHEL, P., KERMAREC, A.-M., and ROWSTRON, A., "Scribe: A Large Scale and Decentralized Application-level Multicast Infrastructure," *IEEE Journal on Seleceted Areas in Communications*, vol. 20, no. 8, 2002.

[32] CHALLENGER, J., IYENGAR, A., and DANTZIG, P., "A Scalable System for Consistently Caching Dynamic Web Data," in *Proceedings of IEEE INFOCOM 1999*, March 1999.

[33] CHALLENGER, J., IYENGAR, A., WITTING, K., FERSTAT, C., and REED, P., "Publishing System for Efficiently Creating Dynamic Web Content," in *Proceedings of IEEE INFOCOM 2000*, May 2000.

[34] CHALLENGER, J., DANTZIG, P., IYENGAR, A., SQUILLANTE, M. S., and ZHANG, L., "Efficiently Serving Dynamic Data at Highly Accessed Web Sites," *IEEE/ACM Transaction on Networking*, vol. 12, April 2004.

[35] CHAN, M. C. and WOO, T. W. C., "Cache-Based Compaction: A New Technique for Optimizing Web Transfer," in *Proceedings of IEEE INFOCOM 1999*.

[36] CHANG, C.-Y., McGREGOR, T., and HOLMES, G., "The LRU* WWW Proxy Cache Document Replacement Algorithm," in *Proceedings of the Asia Pacific Web Conference*, 1999.

[37] CHANKHUNTHOD, A., DANZIG, P., NEERDAELS, C., SCHWARTZ, M., and WORELL, K., "A Hierarchical Internet Object Cache," in *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[38] CHERKASOVA, L. and CIARDO, G., "Role of Aging, Frequency, and Size in Web Cache Replacement Policies," in *Proceedings of the 9th International Conference on High-Performance Computing and Networking*, June 2001.

[39] COBENA, G., ABITEBOUL, S., and MARIAN, A., "Detecting Changes in XML Documents," in *Proceedings of the 18th International Conference on Data Engineering*, February 2002.

[40] DABEK, F., COX, R., KAASHOEK, F., and MORRIS, R., "Vivaldi: A Decentralized Network Coordinate System," in *Proceedings of ACM-SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, August 2004.

[41] DAR, S., FRANKLIN, M. J., JONSSON, B. T., SRIVASTAVA, D., and TAN, M., "Semantic Data Caching and Replacement," in *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, September 1996.

[42] DATTA, A., DUTTA, K., THOMAS, H., VANDERMEER, D., and RAMAMRITHAM, K., "Accelerating Dynamic Web Content Generation," *IEEE Internet Computing*, September/October 2002.

[43] DATTA, A., DUTTA, K., THOMAS, H., VANDERMEER, D., SURESHA, and RAMAMRITHAM, K., "Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 2002.

[44] DATTA, A., DUTTA, K., THOMAS, H., VANDERMEER, D., RAMAMRITHAM, K., and FISHMAN, D., "A Comparitive Study of Alternate Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration," in *Proceedings of 27th International Conference on Very Large Data Bases (VLDB-2001)*, September 2001.

[45] DAVIDSON, S. B., GARCIA-MOLINA, H., and SKEEN, D., "Consistency in Partitioned Networks," *ACM Computing Surveys*, vol. 17, no. 3, 1985.

[46] DEGENARO, L., IYENGAR, A., LIPKIND, I., and ROVELLOU, I., "A Middleware System Which Intelligently Caches Query Results," in *Proceedings of Middleware 2000, IFIP/ACM International Conference on Distributed Systems Platforms*, April 2000.

[47] DESHPANDE, P., RAMASAMY, K., SHUKLA, A., and NAUGHTON, J. F., "Caching Multidimensional Queries Using Chunks," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 1998.

[48] FAN, L., CAO, P., ALMEIDA, J., and BRODER, A., "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," in *Proceedings of ACM SIGCOMM 98*, September 1998.

[49] GAO, L., DAHLIN, M., NAYATE, A., ZHENG, J., and IYENGAR, A., "Improving Availability and Performance with Application-Specific Data Replication," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 17, January 2005.

[50] GUY, R. G., HEIDEMANN, J. S., MAK, W.-K., JR., T. W. P., POPEK, G. J., and ROTHMEIER, D., "Implementation of the Ficus Replicated File System," in *Proceedings of USENIX Summer Conference*, 1990.

[51] GUY, R. G., REIHER, P. L., RATNER, D., GUNTER, M., MA, W., and POPEK, G. J., "Rumor: Mobile data access through optimistic peer-to-peer replication," in *Proceedings of the 17th International Conference on Conceptual Modelling (ER'98)*, 1998.

[52] GWERTZMAN, J. and SELTZER, M., "World Wide Web Cache Consistency," in *Proceedings of the 1996 Usenix Annual Technical Conference*, January 1996.

[53] "Internet Cache Protocol: Protocol Specification, Version 2," September 1997. http://icp.ircache.net/rfc2186.txt.

[54] IYENGAR, A. and CHALLENGER, J., "Improving Web Server Performance by Caching Dynamic Data," in *Proceedings of the 1st USENIX symposium on Internet Technologies and Systems*, December 1997.

[55] JAIN, A. K., MURTY, M. N., and FLYNN, P. J., "Data Clustering: A Review," *ACM Computing Surveys*, vol. 31, no. 3, 1999.

[56] JAMIN, S., JIN, C., KURC, A. R., and SHAVITT, Y., "Constrained Mirror Placement on the Internet," in *Proceedings of IEEE-INFOCOM'01*, April 2001.

[57] KARGER, D., SHERMAN, A., BERKHEIMER, A., BOGSTAD, B., DHANIDINA, R., IWAMOTO, K., KIM, B., MATKINS, L., and YERUSHALMI, Y., "Web Caching with Consistent Hashing," in *Proceedings of the 8th International World Wide Web Conference*, 1999.

[58] KELLY, T., JAMIN, S., and MACKIE-MASON, J. K., "Variable QoS from Shared Web Caches: User Centered Design and Value-Senssitive Replacement," in *Proceedings of the MIT Workshop on Internet Service Quality Economics*, 1999.

[59] KELLY, T. and MOGUL, J., "Aliasing on the World Wide Web: Prevalence and Performance Implications," in *Proceedings of the 11th International World Wide Web Conference*, May 2002.

[60] KHAYAT-HOSSEINI, S., *Investigation of Genaralized Caching.* PhD thesis, Washington University, St. Louis, MO, 1997.

[61] KORUPOLU, M. R. and DAHLIN, M., "Coordinated Placement and Replacement for Large-Scale Distributed Caches," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 6, 2002.

[62] KULKARNI, P., DOUGLIS, F., LAVOIE, J., and TRACEY, J., "Redundancy Elimination Within Large Collections of Files," in *Proceedings of the USENIX Annual Technical Conference*, June 2004.

[63] LABRINIDIS, A. and ROUSSOPOULOS, N., "Update Propagation Strategies for Improving the Quality of Data on the Web," in *Proceedings of 27th International Conference on Very Large Data Bases (VLDB-2001)*, September 2001.

[64] LEVY-ABEGNOLI, E., IYENGAR, A., SONG, J., and DIAS, D. M., "Design and Performance of a Web Server Accelerator," in *Proceedings of IEEE-INFOCOM'99*, March 1999.

[65] LI, D. and CHERITON, D. R., "Scalable Web Caching of Frequently Updated Objects Using Reliable Multicast," in *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS'99)*, October 1999.

[66] LI, W.-S., HSIUNG, W.-P., KALSHNIKOV, D. V., SION, R., PO, O., AGRAWAL, D., and CANDAN, K. S., "Issues and Evaluations of Caching Solutions for Web Application Acceleration," in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB-2002)*, August 2002.

[67] LUO, Q., KRISHNAMURTHY, S., MOHAN, C., PIRAHESH, H., WOO, H., LINDSAY, B. G., and NAUGHTON, J. F., "Middle-tier Database Caching for E-Business," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 2002.

[68] LUO, Q. and NAUGHTON, J. F., "Form-Based Proxy Caching for Database-Backed Web Sites," in *Proceedings of the 27th International Conference on Very Large Databases (VLDB)*, September 2001.

[69] MANBER, U., "Finding Similar Files in a Large File System," in *Proceedings of the USENIX Winter 1994 Technical Conference*, January 1994.

[70] MENDES, M. A. and ALMEIDA, V. A., "Analyzing the Impact of Dynamic Pages on the Performance of Web Servers," in *Proceedings of International CMG Conference*, December 1998.

[71] MICHEL, S., NGUYEN, K., ROSENSTEIN, A., ZHANG, L., FLOYD, S., and JACOBSON, V., "Adaptive Web Caching: Towards a New Global Caching Architecture," *Computer Networks and ISDN Systems*, vol. 30, November 1998.

[72] MOGUL, J., CHAN, Y., and KELLY, T., "Design, Implementation, and Evaluation of Duplicate Transfer Detection in HTTP," in *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, March 2004.

[73] MOGUL, J., "Network Behavior of a Busy Web Server and its Clients," tech. rep., DEC Western Research Laboratories, 1995.

[74] MOHAPATRA, P. and CHEN, H., "A Framework for Managing QoS and Improving Performance of Dynamic Web Content," in *Proceedings of the IEEE Global Telecommunications Conference*, November 2001.

[75] NAAMAN, M., GARCIA-MOLINA, H., and PAEPCKE, A., "Evaluation of ESI and Class-Based Delta Encoding," in *Proceedings of the 8th International Workshop on Web Content Caching and Distribution*, 2003.

[76] NG, E. and ZHANG, H., "Predicting Internet Network Distance with Coordinates-Based Approaches," in *Proceedings of IEEE-INFOCOM*, June 2002.

[77] NINAN, A., KULKARNI, P., SHENOY, P., RAMAMRITHAM, K., and TEWARI, R., "Scalable Consistency Maintenance in Content Distribution Networks Using Cooperative Leases," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 15, July 2003.

[78] PITKOW, J. and RECKER, M., "A Simple Yet Robust Caching Algorithm Based on Dynamic Access Patterns," in *Proceedings of the 2nd International World Wide Web COnference (WWW-94)*, October 1994.

[79] PODLIPNIG, S. and BOSZORMENYI, L., "A Survey of Web Cache Replacement Strategies," *ACM Computing Surveys*, vol. 35, December 2003.

[80] PSOUNIS, K. and PRABHAKAR, B., "A Randomized Web-Cache Replacement Scheme," in *Proceedings of IEEE-INFOCOM-2001*, April 2001.

[81] QIU, L., PADMANABHAN, V. N., and VOELKER, G. M., "On the Placement of Web Server Replicas," in *Proceedings of IEEE-INFOCOM'01*, April 2001.

[82] RABIN, M. O., "Fingerprinting by Random Polynomials," tech. rep., Center for Research in Computing Technology, Harvard University, 1981.

[83] RABINOVICH, M., XIAO, Z., and AGGARWAL, A., "Computing on the Edge: A Platform for Replicating Internet Applications," in *Proceedings of the 8th International Workshop on Web Content Caching and Distribution*, September 2003.

[84] RABINOVICH, M., XIAO, Z., DOUGLIS, F., and KALMAN, C. R., "Moving Edge-Side Includes to the Real Edge - the Clients," in *Proceedings of Usenix Symposium on Internet Technologies and Systems*, March 2003.

[85] RAMASWAMY, L., IYENGAR, A., LIU, L., and DOUGLIS, F., "Techniques for Efficient Fragment Detection in Web Pages," in *Proceedings of ACM-CIKM International Conference on Information and Knowledge Management*, November 2003.

[86] RAMASWAMY, L., IYENGAR, A., LIU, L., and DOUGLIS, F., "Automatic Detection of Fragments in Dynamically Generated Web Pages," in *Proceedings of the 13th World Wide Web Conference*, May 2004.

[87] RAMASWAMY, L., IYENGAR, A., LIU, L., and DOUGLIS, F., "Automatic Detection of Fragments in Dynamic Web Pages and its Impact on Caching," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 17, June 2005.

[88] RAMASWAMY, L. and LIU, L., "An Expiration Age-Based Document Placement Scheme for Cooperative Web Caching," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 16, May 2004.

[89] RAMASWAMY, L., LIU, L., and IYENGAR, A., "Cache Clouds: Cooperative Caching of Dynamic Documents in Edge Networks," in *Proceedings of the 25th International Conference on Distributed Computing Systems(ICDCS-2005)*, June 2005.

[90] RHEA, S. C., LIANG, K., and BREWER, E., "Value-Based Web Caching," in *Proceedings of the 12th International World Wide Web Conference*, 2003.

[91] ROSS, K. W., "Hash-Routing for Collections of Shared Web Caches," *IEEE Network Magazine*, November-December 1997.

[92] SARKAR, P. and HARTMAN, J., "Efficient Cooperative Caching using Hints," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, October 1996.

[93] SHAH, S., RAMAMRITHAM, K., and SHENOY, P., "Resilient and Coherence Preserving Dissemination of Dynamic Data Using Cooperating Peers," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 15, July 2004.

[94] SONG, J., IYENGAR, A., LEVY-ABEGNOLI, E., and DIAS, D. M., "Architecture of a Web Server Accelerator," *Computer Networks*, vol. 38, January 2002.

[95] STAROBINSKI, D. and TSE, D. N. C., "Probabilistic Methods for Web Caching," *Performance Evaluation*, vol. 46, October 2001.

[96] TANG, L. and CROVELLA, M., "Virtual Landmarks for the Internet," in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, October 2003.

[97] TERRY, D. B., THEIMER, M., PETERSEN, K., DEMERS, A. J., SPREITZER, M., and HAUSER, C., "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.

[98] TEWARI, R., DAHLIN, M., VIN, H., and KAY, J., "Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet," in *Proceedings of International Conference on Distributed Computing Systems*, May 1999.

[99] THALER, D. and RAVIHSANKAR, C., "Using Name-Based Mappings to Increase Hit Rates," *IEEE/ACM Transactions on Networking*, vol. 6, February 1998.

[100] WILLS, C. and MIKHAILOV, M., "Studying the Impact of More Complete Server Information on Web Caching," in *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, 2000.

[101] WU, K.-L. and YU, P. S., "Latency sensitive hashing for collaborative web caching," *Computer Networks*, vol. 35, June 2000.

[102] Wu, K.-L. and Yu, P. S., "Replication for Load Balancing and Hot-Spot Relief on Proxy Web Cache with Hash Routing," *Distributed and Parallel Databases*, vol. 13, no. 2, 2003.

[103] Yin, J., Alvisi, L., Dahlin, M., and Iyengar, A., "Engineering Web Cache Consistency," *ACM Transactions on Internet Technology*, vol. 2, August 2002.

[104] Yin, J., Alvisi, L., Dahlin, M., and Lin, C., "Using Leases to Support Server-Driven Consistency in Large-Scale Systems," in *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS-1998)*, May 1998.

[105] Yin, J., Alvisi, L., Dahlin, M., and Lin, C., "Volume Leases for Consistency in Large-Scale Systems," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 11, June 1999.

[106] Yu, H. and Vahdat, A., "Design and Evaluation of a Conit-based Continuous Consistency Model for Replicated Services," *ACM Transactions on Computer Systems*, vol. 20, August 2002.

[107] Yu, H., Breslau, L., and Shenker, S., "A Scalable Web Cache Consistency Architecture," in *Proceedings of the ACM SIGCOMM '99 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, August 1999.

[108] Yuan, C., Chen, Y., and Zhang, Z., "Evaluation of Edge Caching/Offloading for Dynamic Content Delivery," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 16, November 2004.

[109] Zegura, E. W., Calvert, K., and Bhattacharjee, S., "How to Model an Internetwork," in *Proceedings of IEEE-INFOCOM*, 1996.

[110] Zhang, J., R.Izmailov, Reininger, D., and Ott, M., "Web Caching Framework: Analytical Models and Beyond," in *Proceedings of the IEEE Workshop on Internet Applications*, 1999.

[111] Zhang, J., Liu, L., Pu, C., and Ammar, M., "Reliable Peer-to-peer End System Multicasting through Replication," in *4th International Conference on Peer-to-Peer Computing (P2P 2004)*, August 2004.

[112] Zhou, S., Ganger, G. R., and Steenkiste, P., "Location-based Node IDs: Enabling Explicit Locality in DHTs," Tech. Rep. CMU-CS-03-171, Computer Science Department, Carnegie Mellon University, 2003.

VITA

Lakshmish Ramaswamy was born in Chamarajanagar, a small town near Bangalore in India. He obtained the bachelors degree (B.E.) in computer science and engineering from the University of Mysore in 1996. He was conferred the masters in science degree (M.S. by research) in computer science by the Indian Institute of Science (IISc.), Bangalore in 1999 for the thesis entitled "Wavelets for Volume Graphics". Subsequently, he worked as a software engineer in the Multimedia Codecs Group of Silicon automation systems (now SASKEN communications limited) until July 2000.

LOOKING BACK

Lakshmish Ramaswamy

Completing the PhD degree is indeed a great experience. It also provides an opportunity to look-back and contemplate upon the path that has led me to this momentous day in my life.

I was born in Chamarajanagar, a small town near Bangalore in India, on July 8th, 1974. My father, M.S. Ramaswamy used to work as an aeronautical engineer at Hindustan Aeronautics Limited (HAL) and my mother, C.R. Padmavathy used to be a professor of chemistry at Bangalore University. Both of them are now retired and are living in Bangalore. My primary, secondary, and pre-university education was all in Bangalore. Today Bangalore is known as the silicon valley and technological capital of India. In my school days most people in Bangalore had never seen a computer. However, my high school had a course in computer science as an extra curricular activity. In addition to the MS-DOS fundamentals, we were thought elementary programming concepts, most of which we implemented in the BASIC programming language. When I look back, I think probably this was what made me choose computer science as my career path.

I did my Bachelors degree in Sree Jayachamarajendra College of Engineering (SJCE), which was then affiliated to Mysore University, majoring in computer science and engineering. Other than studying and preparing for exams, most of my time during those 4 years was spent in passionately arguing with friends about any random topic. The subjects of our discussion included, but were not limited to, politics, international affairs, social issues, economy, cricket, and of course computer science. For our final year project we worked on image compression and coding, wherein we implemented a discrete cosine transform-based image compression software.

When I was in my pre-final year of the bachelor's degree, I had made up my mind that I would do a masters rather than immediately taking up a job. Accordingly I wrote

the Graduate Aptitude Test in Engineering (GATE) exam, and I was ranked 6th among more than 5000 candidates who attended the exam all over India. Through this exam, I got admitted to the masters by research (MS by research) program in computer science at Indian Institute of Science (IISc.), which is one of the prestigious research institutions in India. As a master's student I was interested in graphics and computational geometry. For my thesis, I worked with Prof. Vijaya Chandru and Prof. Swami Manohar on volume graphics and visualization. My master's thesis was titled "Wavelets for Volume Graphics". In fact, my interest in computer graphics and visualization was partly the reason behind applying to Georgia Tech.

When I finished my master's I was in a dilemma regarding my future career. I was interested in doing a PhD but was not sure whether I had the requisite talents and patience. So I decided to take-up a job temporarily and then go in for a PhD, if I really felt the motivation to do so. I started working in Silicon Automation Systems (now SASKEN communications limited) as a software engineer in their multimedia codecs group, where I was employed until I embarked on the journey towards my PhD. When I decided to take up a job as a software engineer, many of my friends and professors at IISc. had warned me that the transition from student to a working professional is usually a one-way door, and it would be very hard to come back to academics. Although I enjoyed my job as a software engineer at SAS, I was feeling the urge to go back to academics and obtain a PhD. With enough moral support from my parents, sister, relatives and other friends I started applying to different universities in the US, and finally chose Georgia Tech. Thus I landed in Atlanta on August 10, 2000.

As most PhD students, in my first semester I was undecided regarding the research area, as many areas in computer science seemed interesting to me. I had dabbled with software architecture, my interest in computer graphics was urging me to go back to that area, and I was also interested in databases and distributed systems. I decided to do a CS-7001 project with Dr. Ling Liu. At the end of the first meeting with her, I had decided to work with her for my PhD. Then, I started reading a couple of papers in the area of distributed data-intensive systems. As I read through those papers I was amazed by the challenges posed by the large amounts of data being pumped into the Internet by various kinds of sources. I got particularly interested in designing systems and techniques for efficient delivery of dynamic web content, and chose this as the topic of my PhD dissertation. My internships at IBM TJ Watson research center with Dr. Arun Iyengar further enhanced my interest in this area.

In addition to the work presented in this thesis, during my years at Georgia Tech, I have also worked on a few other topics including distributed node clustering schemes for peer-to-peer networks, and countering free-riding in peer-to-peer systems through utility-based schemes.

In conclusion, PhD has been a great learning experience for me. It provided me with the an opportunity to explore my talents. It has also been an enjoyable experience, although frustrating at times. In addition to the academic and research skills, I have also learnt many other skills, which I am sure, will help me in my future professional and personal life. Last, but not the least, it was during my years as a PhD student I got married to Seema, who has been a wonderful life partner.