FAST AND SCALABLE INTERNET SERVICE SCHEME FOR STATIC AND DYNAMIC WEB DATA CONTENTS

By

JAЕYONG LIM

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2003
To my dearest father and mother, my beloved wife, Youngjoo, and my daughter Nayoun
ACKNOWLEDGMENTS

I am deeply indebted to Dr. Sanguthevar Rajasekaran, chairman of my supervisory committee, for his continuous guidance, advice, and support throughout the course of my doctoral study. I would like to thank my supervisory committee members – Dr. Paul W. Chun and Dr. Douglas D. Dankel II for their constant help and suggestions; and Dr. Ralph Selfridge, and Dr. Abdelsalam Helal for their kind support and precious time.

I would like to especially thank my father and mother for their unconditional love and support which helped me overcome many obstacles throughout my studies. I also thank my father-in-law and mother-in-law. I am also grateful to my lovely wife, Youngjoo and my lovely daughter Nayoun from the bottom of my heart. I would also like to thank my sister, Sunhee; my uncle, Professor Sangen Kwark; my aunt, Dr. Oakran Lee; and my brother-in-law, Dr. Jaerak Ann for their love shown to me during the study.

Finally, I thank all of the colleagues and friends who helped me with inspiring discussions and made my stay pleasant and enjoyable. I wish them all success in their studies and a bright future.

This work was supported in part by the National Science Foundation under grant EIA-9872507 and CCR-9912395.
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xii</td>
</tr>
<tr>
<td><strong>CHAPTER</strong></td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 RELATED RESEARCH</td>
<td>11</td>
</tr>
<tr>
<td>2.1 Web Caching (Server Side and Network Side)</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Fast IP Routing</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Existing Approaches to IP Lookup</td>
<td>17</td>
</tr>
<tr>
<td>2.3.1 Level Compression-Based Lookup</td>
<td>17</td>
</tr>
<tr>
<td>2.3.2 Caching-Based Lookup</td>
<td>19</td>
</tr>
<tr>
<td>2.3.3 Hash-Based Lookup</td>
<td>21</td>
</tr>
<tr>
<td>2.3.4 Protocol-Based Solutions</td>
<td>21</td>
</tr>
<tr>
<td>2.3.5 Hardware Solutions</td>
<td>22</td>
</tr>
<tr>
<td>3 FAST AND SCALABLE INTERNET SERVICE MODEL</td>
<td>25</td>
</tr>
<tr>
<td>4 BYTE ACCESS FREQUENCY FACTOR BASED WEB CACHING ALGORITHMS FOR STATIC AND DYNAMIC WEB CONTENTS</td>
<td>29</td>
</tr>
<tr>
<td>4.1 Design of Scalable Web Server Architecture for BAFF Concept</td>
<td>29</td>
</tr>
<tr>
<td>4.2 Central Server Mechanism and Operation in Cooperative Web Servers</td>
<td>32</td>
</tr>
<tr>
<td>4.3 Web Server Mechanism and Operation in Cooperative Web Servers</td>
<td>33</td>
</tr>
<tr>
<td>4.4 Byte Access Frequency Factor Based Caching Mechanism</td>
<td>34</td>
</tr>
<tr>
<td>4.5 Randomly Select a Web Server Global Memory Mechanism Algorithm</td>
<td>36</td>
</tr>
<tr>
<td>4.6 Round Robin Method Global Memory Mechanism Algorithm</td>
<td>38</td>
</tr>
<tr>
<td>4.7 Dedicated Web Server Global Memory Mechanism Algorithm</td>
<td>38</td>
</tr>
<tr>
<td>4.8 Simulation Environment</td>
<td>40</td>
</tr>
<tr>
<td>4.9 Simulation Results</td>
<td>42</td>
</tr>
<tr>
<td>4.9.1 Performance Evaluation for Small Number of Documents Set</td>
<td>43</td>
</tr>
<tr>
<td>4.9.2 Performance Evaluation for Large Number of Documents Set</td>
<td>47</td>
</tr>
</tbody>
</table>
4.9.3 Performance Comparison in the Increasing Number of Documents
4.9.4 Analysis of Simulation Results

5 DIRECTLY SELECTED AND LIMITED LOOKUP CACHE ALGORITHM AND DIRECT REQUEST DISTRIBUTION AND LOAD BALANCING CACHE PROTOCOL FOR STATIC AND DYNAMIC WEB CONTENTS

5.1 Design of Scalable Web Server Architecture for Direct Control
5.2 Directly Selected and Limited Look up Web Cache Algorithm
5.3 Direct Request Distribution and Load Balancing Cache Protocol
   5.3.1 Central Server Mechanism and Operation in DLC Protocol
   5.3.2 Web Server Mechanism and Operation in DLC Protocol
   5.3.3 RLM Mechanism and Operation in DLC Protocol
5.4 Simulation Environment for DSLL and DLC
5.5 Simulation Results
   5.5.1 Performance Evaluation for Variation of Dynamic Data
   5.5.2 Performance Comparison in case of Using the RLM
   5.5.3 Analysis of Simulation Results

6 PARALLEL WEB SERVER CACHE CONTENT MANAGEMENT PROTOCOL AND ALGORITHM FOR STATIC AND DYNAMIC WEB CONTENTS

6.1 Design of Scalable Web Server Architecture for Parallel Control
6.2 Randomly Selected and Limited Look at Neighbor Web Cache Algorithm
6.3 Random Request Distribution and Load Balancing Cache Protocol
   6.3.1 Central Server Mechanism and Operation in RLC Protocol
   6.3.2 Web Server Mechanism and Operation in RLC Protocol
6.4 Simulation Environment for RSLL and RLC
6.5 Simulation Results
   6.5.1 Performance Results of Variation of Dynamic Data
   6.5.2 Performance Comparison between RSLL and DSLL
   6.5.3 Analysis of Simulation Results

7 FAST AND SCALABLE IP LOOKUP AND UPDATE SCHEME FOR IP ROUTING

7.1 IP Lookup and Design Concepts
7.2 Gigabit IP Switching Router Architecture
7.3 Reducing the Longest Matching Prefix Search to Binary Search
7.4 Generic Algorithm
7.5 Case of IP Routing
7.6 Experimental Results
   7.6.1 Original IP Router Data
   7.6.2 Randomized IP Router Data
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-1</td>
<td>Distinct end points search table with $&gt;$, $=$ values</td>
<td>97</td>
</tr>
<tr>
<td>7-2</td>
<td>Distinct end points search table with $\leq$, $&gt;$ values</td>
<td>100</td>
</tr>
<tr>
<td>7-3</td>
<td>Tested NAP IP router information</td>
<td>103</td>
</tr>
<tr>
<td>7-4</td>
<td>Search time for original NAP’s IP router data</td>
<td>104</td>
</tr>
<tr>
<td>7-5</td>
<td>Search time for randomly generated IP address data</td>
<td>105</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>FASTS model architecture</td>
<td>27</td>
</tr>
<tr>
<td>4-1</td>
<td>Scalable web server architecture</td>
<td>30</td>
</tr>
<tr>
<td>4-2</td>
<td>Functional components of web server</td>
<td>31</td>
</tr>
<tr>
<td>4-3</td>
<td>Suboptimal cache replacement operation</td>
<td>35</td>
</tr>
<tr>
<td>4-4</td>
<td>RWG web caching algorithm</td>
<td>37</td>
</tr>
<tr>
<td>4-5</td>
<td>WRG web caching algorithm</td>
<td>38</td>
</tr>
<tr>
<td>4-6</td>
<td>RDG web caching algorithm</td>
<td>39</td>
</tr>
<tr>
<td>4-7</td>
<td>Total response times for 4000 documents in the nonuniform case</td>
<td>43</td>
</tr>
<tr>
<td>4-8</td>
<td>Total response times for 4000 documents in the uniform case</td>
<td>44</td>
</tr>
<tr>
<td>4-9</td>
<td>Total memory hits for 4000 documents in the nonuniform case</td>
<td>44</td>
</tr>
<tr>
<td>4-10</td>
<td>Total memory hits for 4000 documents in the uniform case</td>
<td>45</td>
</tr>
<tr>
<td>4-11</td>
<td>Total response times for 5000 documents in the nonuniform case</td>
<td>46</td>
</tr>
<tr>
<td>4-12</td>
<td>Total response times for 5000 documents in the uniform case</td>
<td>46</td>
</tr>
<tr>
<td>4-13</td>
<td>Total memory hit ratios for 5000 documents in the nonuniform case</td>
<td>47</td>
</tr>
<tr>
<td>4-14</td>
<td>Total response times for 8000 documents in the nonuniform case</td>
<td>48</td>
</tr>
<tr>
<td>4-15</td>
<td>Total response times for 8000 documents in the uniform Case</td>
<td>48</td>
</tr>
<tr>
<td>4-16</td>
<td>Total memory hit ratios for 8000 documents in the nonuniform case</td>
<td>49</td>
</tr>
<tr>
<td>4-17</td>
<td>Total memory hit ratios for 8000 documents in the uniform case</td>
<td>49</td>
</tr>
<tr>
<td>4-18</td>
<td>Total response times for 10,000 documents in the uniform case</td>
<td>50</td>
</tr>
<tr>
<td>4-19</td>
<td>Total response times for 10,000 documents in the nonuniform case</td>
<td>50</td>
</tr>
</tbody>
</table>
7-5  Search time performance gain in original data ..........................................................104
7-6  Search time performance gain in randomized data .....................................................105
Abstract of Dissertation Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

FAST AND SCALABLE INTERNET SERVICE SCHEME FOR STATIC AND DYNAMIC WEB DATA CONTENTS

By

Jaeyong Lim

May 2003

Chairman: Sanguthevar Rajasekaran
Major Department: Computer and Information Science and Engineering

The Internet has gained tremendous popularity by providing a means to share information, to connect people and organizations, to perform business transactions, and to entertain ourselves. Together with the phenomenal growth of the Internet, several problems such as the increasing bandwidth of network links, excellent load balancing of web servers, faster response time for web client’s request, and faster Internet Protocol (IP) forwarding have been recognized. They must be solved for supporting faster service over the Internet. This dissertation proposes a so-called FAST and Scalable inTernet Service scheme (FASTS) that includes web server cache algorithms, cache protocols, and a fast and scalable IP lookup and update scheme for use over the Internet.

We have developed five Byte Access Frequency Factor (BAFF) based web cache algorithms and two web server cache protocols for Internet services with multimedia and dynamic web content. Our cache algorithms increase memory hit rate and memory utilization; and also reduce data traffic on the network by looking at the neighboring web
server’s cached contents and by direct or random distribution of web client requests. Our cache protocols manage and update cached static and dynamic contents completely and consistently in distributed cooperative web servers. These web cache algorithms and cache protocols have several advantages (reduced memory consumption, easy maintenance, and reduced communication overhead among web servers). They also have excellent load balancing among cooperative distributed web servers. Recently, fast response time for web client request has become more and more important in web servers. Therefore, by using these algorithms and protocols for web caching, we can build scalable web servers supporting very fast response times, high memory hit rates, and good load balancing of cooperative distributed web servers.

We also have developed a fast and scalable IP lookup algorithm. Our main contributions are construction of forwarding table in small, simple, and more accurate structure with grouping concepts and a faster searching algorithm to find the longest matching prefix for given any IP address in small forwarding group elements. We developed a new grouping algorithm that converts a large forwarding table to several small forwarding tables with even intervals. These small forwarding tables search faster and more accurately for any given IP address in an IP router. We reduced the general binary search concepts to the longest matching prefix problem in the IP router. Our algorithm has good performance in lookup, has excellent scalability, and does not require protocol change. Our algorithm also allows cheap, fast software implementations. Our algorithm is faster in a large IP routing table than in a small one. This is good for the current large IP routing table data for hastening lookup and update time.
CHAPTER 1
INTRODUCTION

In recent years, the use of Internet services has grown significantly. The World Wide Web (WWW) has become an important part of the common man’s life. Popular web servers have several million hits per day and it has become common for websites to serve 100,000 requests daily. The great impact of the Internet can be immediately seen in how the Internet has changed the traditional lifestyles of people all over the world. The way people obtain information has now changed to performing just a few mouse clicks on a computer connected to the Internet. People can communicate with almost anybody anywhere in the world using e-mail. All sorts of businesses are aggressively putting up their home pages on the Internet, and people are buying and selling products through the Internet. Entertainment is also being provided through the Internet in the form of games, movies, etc.

The Internet is now a vast sea of information where people to surf its waves for information. The amount of information being provided and the number of users and businesses being connected on the Internet are continuously increasing. It has been reported that the number of WWW users has grown dramatically \([1,2,3,4,5,6]\). Currently much of the bandwidth available on the Internet is used to transfer documents \([3,6,7,8,9]\). Servers for the WWW must manage a multi-gigabyte database of multimedia information while serving multiple request streams.

Thus we are experiencing explosive growth in Internet use. However, several problems related to the current Internet technology are preventing further growth and the
development of new applications on the Internet. The fundamental problem is that
requested service completion time over the Internet is too slow. This problem is related to
several factors (network links, type of services, load balancing of servers, IP forwarding,
etc.).

To hasten service of the Internet, we focus on web server cache and protocol; and
the fast IP forwarding mechanism. To define the problem and to propose solutions, we
describe limitations of the existing Internet technology for web server cache and IP
forwarding. Then we explain how our proposed FAST and Scalable inTernet Service
(FASTS) model alleviates these limitations.

First, we look at the current web server cache approach. To support the growing
number of web client’s requests, scalable WWW architecture that consists of a group of
loosely coupled servers is needed. Web servers with very fast response times are clearly
desirable. Currently, these servers typically use large amounts of memory as cache space
for supporting the high request rates on such systems. Available memory must be used as
efficiently as possible to reduce network traffic and to decrease response times for a web
client’s request.

To improve the performance of web servers, caching is one of the most commonly
used techniques. Caching can be performed at client (built-in browser cache), at web
server (primary web server cache), and at the network (proxy caches). We studied the
second and third type of caching – caching of web documents in the main memory of a
web server and a network server. Caching at web server and at network server enables
quick responses to frequently accessed documents. Caching of web contents has several
significant advantages (reduced server load; reduced network bandwidth; reduced latency
since cached responses are available immediately; and more reliability, as some objects may be retrievable via cache even when the original servers are not reachable). These features can make the World Wide Web less expensive and more effective. One drawback of caching is that it tends to retrieve out-of-date objects stored in a cache instead of fetching current objects from the origin server.

Web servers provide two types of data [10,11,12]: static data (from files stored on a server) and dynamic data (constructed by programs that execute at the time a request is made). Many websites need to provide dynamic content. Dynamic pages are essential at websites that provide frequently changing data [10,11]. Examples include sport sites, stock market sites, and virtual stores or auction sites where information on product availability or pricing is constantly changing. There are several problems with providing dynamic data to clients efficiently and consistently. A key problem with dynamic data is that dynamic pages can seriously reduce web server performance. The performance bottleneck is often the CPU overhead associated with generating dynamic pages. The overhead for dynamic data is a major problem for websites that receive substantial request volumes. Another problem with caching dynamic pages is determining what pages should be cached and when a cached page has become obsolete. Thus, to improve the performance of web server, a key requirement for many websites providing static and dynamic data is to completely and consistently update pages, that have changed, and to provide an excellent load-balancing mechanism among web servers.

Numerous papers [4,5,13,14,15,16,17,18,19] discuss caching of web content; however, they focus on caching static data and intentionally avoid caching dynamic data. Furthermore, they concentrate on proxy caching rather than server-side caching. Proxy
caching of dynamic content may not be feasible because it does not permit caching of authenticated content. Proxy caching has other problems (obsolete cached data; long response time for web client request while proxy server searches requested data in multilevel hierarchical structure; and increase of network bandwidth for message communication among distributed proxy servers).

Next, we look at the current IP forwarding approach. The introduction of the Gigabit Ethernet at the LANs and the random traffic patterns generated by Internet users have increased the demand for higher bandwidth in the Internet backbone. Speeding up packet forwarding in the Internet backbone requires high-speed communication links and faster routers. The bandwidths of transmission links keep improving and gigabit fiber links are readily available. This has shortened the time of processing and forwarding packets. Thus, the key to improving Internet speed is fast routers [20]. A multigigabit router must have enough internal bandwidth to switch packets between its interfaces at multigigabit rates and enough packet-processing power to forward several Millions of Packets Per Second (MPPS) [21]. Switching in the router has been studied extensively in the ATM community [22] and solutions for fast packet processing are readily available [23]. As a result, the major remaining bottleneck in the Internet router design is the slow, software-based IP lookup procedures.

An IP lookup scheme is the most fundamental operation in any IP routing product. A packet is received with a specific IP Destination Address (DA), a unique 32-bit field in current IP version 4 implementations. IP routes have been identified by a <routing prefix, prefix length> pair. The router must search forwarding tables using the DA as its key and determine which entry in the table represents the best route for the packet to its
destination. The search is complicated by the fact that entries in the table have variable
lengths, and also that many entries may represent valid routes to the same destinations.
The search may be time consuming, especially in a backbone router with a large number
of table entries. The search was done to conserve the memory space used by the
forwarding tables. Unlike a simple search or hash that seeks to find an exact match within
a table, the IP lookup algorithm must select the most specific route from a number of
entries for the given DA. For example, a forwarding database may have the prefixes P1 =
0101, P2 = 0101101 and P3 = 010110101011. An address whose first 12 bits are
010101101011 has the longest matching prefix P1. On the other hand, an address whose
first 12 bits are 010110101101 has the longest matching prefix P3. To meet the demand
for high-speed routers, IP lookup schemes that are efficient, flexible, and cost-effective
are required.

There are four requirements for a good lookup scheme: search time, memory,
scalability, and update time. Search time is important for lookup to not be a bottleneck.
Schemes that are memory-efficient lead to faster search because compact data structures
fit in fast and expensive Static RAM. Scalability in both number of prefixes and address
length is important because prefix databases are growing, and a switch to IPv6 will
increase address prefix lengths [24,25]. Finally, schemes with fast update time are
desirable because

- Instabilities in backbone protocols [26] can cause rapid insertion and deletion of
  prefixes
- Multicast forwarding support requires route entries to be added as part of the
  forwarding process and not by a separate routing protocol
- Any lookup scheme that does not support fast incremental updates will require two
copies of the lookup structure in fast memory; one for updates and one for lookup.
Internet address lookup would be simple if we could lookup a 32 bit IP destination address in a table that lists the output link for each assigned Internet address. In this case, lookup could be done by hashing, but a router would have to keep millions of entries. To reduce database size and routing update traffic, a router database consists of a smaller set of prefixes. This reduces database size, but at the cost of requiring a more complex lookup called longest matching prefix. It also requires a more complex update procedure when prefixes are added and deleted.

Backbone routers, in these days, should be able to handle 2 million searches per second to have 1Gps performance. A possibility also exists for many updating packets to be generated, especially when the network is in an unstable mode. Backbone routers must be able to update their table at least as quickly as one thousand updates per second. In addition to fast search and updating, routers also need to keep the routing table as small as possible. Backbone routers normally have tens of thousands of routing entries, and some have more than 300,000 entries. To achieve good scalability, the algorithm must consume less memory in maintaining and handing the routing table.

Recognizing all of the above problems of the current web server cache and IP forwarding mechanism in the Internet services, we propose a novel FASTS model that focuses on web cache algorithms and cache protocols; and a fast and scalable IP lookup and update scheme. To quickly service a web client’s request, advances in server and IP router technology are needed to improve response time, reduce server load, and reduce network bandwidth under heavy load conditions. Thus, we have explored two types of solutions: increasing the bandwidth of the IP router and using dynamic web caches and cache protocols over the Internet to achieve faster response time for a web client’s
request, a good load balance among web servers, and completely and consistently updating cached static and dynamic data.

We propose supporting fast Internet services over the Internet by designing and implementing the FASTS model. The FASTS model comprises five new web server cache algorithms, two new web cache protocols, and a new fast and scalable IP lookup and update scheme. These are described later in detail. Here we briefly explain the design issue and contributions of the FASTS model.

For web server cache and cache protocol, a well-known approach for predicting the next requested document in a cache uses a past document request history. Though this may not always guarantee the best performance, it is simple and easy. To cache a document in a web server’s memory, we consider past request frequency for static or dynamic data; and also the size of the static or dynamic data. We use BF values (Byte access Frequency factor of a document) to make caching decisions. This approach decreases network traffic, response times, and CPU loads on a web server; and increases memory use.

Requests for documents from different users are typically independent of each other. Dependence is found only when a single user makes successive requests. The current Internet is built on stateless technologies. Hyper Text Transfer Protocol (HTTP) [27] is a stateless protocol that does not remember anything from previous connections even to the same site. Therefore, every client request is independent of any other. This requires redundant information to be passed to the web server each time a request is made. Thus HTTP makes for highly inefficient continued interactions with a web server. In general, it is difficult to accurately model user request patterns. Although many
researchers [2,3,4,6,10,11,12,18,28] have tried to discover a model to fit a web client’s request patterns by analyzing some real log traces in several web servers, there are no satisfactory results in this area. A general observation that many researchers have made is that a small fraction (around 10% to 20%) of documents account for most user requests. With this in mind we developed new web cache algorithms and protocols.

These algorithms are designed to adapt to current web client randomized request patterns. These protocols have intelligent cache algorithms, an in-memory cache maintenance table, and as a consistent and complete cache maintenance mechanism. Our approach does not need extra applications for managing static and dynamic web contents. Our approach also achieves less communication cost among web servers using broadcast mechanisms. These algorithms and protocols achieve excellent cache hit rate, response time, load balancing, and memory use compared to LRU and LFU algorithms. Many investigators [2,3,4,5,6,12] have used different log traces for measuring their algorithms’ performance gains and there is no general basis for comparing these techniques. Therefore, we compare our algorithms with LRU and LFU algorithms which are the most widely used web cache algorithms. By using these cache protocols and cache algorithms for web servers, we can build scalable web servers supporting very fast response times, high memory hit rates, and excellent load balancing of cooperative distributed web servers.

For fast and scalable IP lookup and updating algorithms in IP forwarding table, our main contributions are constructing a forwarding table in a small, simple, and more accurate structure with grouping concepts; and a faster searching algorithm to find the longest matching prefix for given any IP address in small forwarding group elements. We
developed a new grouping algorithm that converts a large forwarding table to several small forwarding tables with even intervals. These small forwarding tables search faster and more accurately for any given IP address in an IP router. We reduced the general binary search concepts to the longest matching prefix problem in an IP router. Our algorithm has good performance in lookup, has excellent scalability, and does not require protocol change. Our algorithm also allows cheap, fast software implementations. Our algorithm is faster in a large IP routing table than in a small one. This is good for the current large IP routing table data for speeding up lookup and update time.

Each router has a different length of prefixes. Distribution of prefix length in each router is quite different. One router may have more 16-bit prefixes than others. Another router have more 24-bit prefixes than others. This is the main problem in constructing the forwarding table. This uneven prefix distribution pattern increases the depth of the tree and requires more search time in more dense regions. To overcome this problem, many previous approaches used the pre-computation information in a forwarding table to find the longest matching prefix faster for given any IP address and to update the forwarding table faster for the change of the IP router entries. However, this pre-computation information requires a more complicated data structure for storing the prefix information. This complicated data structure may cause difficulty in updating and maintaining IP forwarding table entries. To avoid this problem, our IP lookup algorithm uses the grouping concepts to classify the large number of IP forwarding table entries into even intervals and a small number of grouping entries. To group the large entries of the forwarding table into small and even interval group entries, we look at the first q bits of the each IP forwarding table entries. The total number of each IP forwarding table entries
and the longest prefix length of the IP forwarding table entries determines the q bits. This approach is faster and more accurate. This is the main factor in hastening the IP lookup search procedure in our algorithm.

The organization of the remainder of this dissertation is as follows. Chapter 2 shows related research on web server side and network side cache, and IP forwarding approaches. Chapter 3 discusses a FAST and Scalable inTernet Service scheme (FASTS), which is the basis for providing fast Internet service. Chapter 4 describes Byte Access Frequency Factor based web-caching algorithms and overall web server architecture. Chapter 5 presents the web server cache algorithm and web server cache protocol for direct control. Chapter 6 explains the parallel controlled web server cache algorithm and web server protocol. Chapter 7 discusses design issues and the proposed fast and scalable IP lookup and update scheme. Chapter 8 summarizes the work and suggests future work.
CHAPTER 2
RELATED RESEARCH

Our work on the design and development of a FAst and Scalable inTernet Service (FASTS) model involves several emerging fields of research and technology. The key technologies that have motivated us to pursue this research are web caching and cache protocol (server side and network side); and fast IP routing technology.

2.1 Web Caching (Server Side and Network Side)

Conventional caching techniques such as LRU and LFU perform poorly with large documents. Nowadays the most popular documents people handle happen to be audio and video files, which are inherently large. When a user is accessing large documents such as video, a single document itself may occupy the whole cache if LRU or LFU is used.

LRU policy treats every cache block equally and does not make a distinction between data blocks based on their data type (text, images, audio, video, etc.); or based on the size of the request. LRU is not designed for Internet services. It only uses the time since last access and does not take into account the access frequency of the documents when making replacement decisions. It may be possible to improve request response time by keeping frequently accessed documents in the web server’s own cache.

Another conventional cache replacement algorithm, LFU, uses the history of accesses to predict the probability of a subsequent reference. If LFU is applied in the Internet environment, one or more documents in the cache will always be replaced by a new document without considering the access probabilities of the replaced and the new documents. In this case, large documents such as video files will fill up the whole cache.
when they are accessed. This totally destroys the caching mechanism. Besides, documents which were recently active but currently cold may have large frequency counts and remain in the cache, thus preventing newer additions to the cache from gathering sufficient reference counts to stay in the cache. This phenomenon is called cache pollution [29]. This problem is more serious if those currently cold documents are large documents. Due to these reasons, LFU is also not well suited for Internet caching.

One of the earliest papers deals with the problem of system performance was the work by Satyanarayanan and Kistler [30]. Their work focused on improving the overall performance of the system at the file system level using caching and document replication. Danzig, Hall and Schwartz proposed a hierarchical caching system that caches files at Core Nodal Switching Subsystems within the NSFNET to reduce the bandwidth used by the system [31]. In these researches, document transfers were assumed to be mainly small. This assumption is no longer valid since audio and video files are accessed quite often.

Tatarinov, Rousskov, and Soloviev have proposed a static caching protocol [5]. This policy was based on the observation that web documents access patterns changed very slowly. Static caching showed good performance for highly loaded web servers that stored large amounts of rarely modified documents with a limited cache size. But the problem with this protocol is that it is not efficient for websites that provide up-to-date news.

Yeung, Suen, and Ng proposed an optimal cache replacement algorithm [32,33]. Through analysis and simulation, this algorithm was shown to have a cache hit rate 50% better than the LFU algorithm when the cache size was small. But each server cached the
same documents for some period and this approach did not consider load balancing among web servers.

Menaud, Issarny, and Banatre proposed a distributed solution to cooperative caching [4], which produced a tolerable bandwidth and storage overhead. It was based on distributing the state of all cooperating caches. This approach induced high memory consumption for storing system states locally. It further caused a nonnegligible network load because messages need to be issued to siblings each time a cache was updated. Load balancing among the servers was also not considered. It incurred a diminished performance gain. The performance of this approach depended on the actual state of locally distributed servers. Each server did not contain all of the information of the distributed servers’ local states and this information was also not updated consistently and completely.

Takuya, Hiroyoshi, and Yoshiaki proposed a hash-based query caching method using both a hash function and a query caching method [13]. This method could find cached objects among several cache servers by using only one query message. However, every cache server kept the whole of other cache servers’ cached information in its memory causing bad memory use. Memory management was very difficult as well. They did not allow the duplication of data among cache servers resulting in a skewed load balance among cache servers and longer response times for client requests. They also did not consider the consistency of cached data among cache servers.

Holmedahl, Smith, and Yang proposed the Swala protocol that distributed web servers for managing dynamic web data [12]. This approach had been based on the observation that the performance bottleneck was more often at the CPU than elsewhere in
the network. They only cached the results of CGI requests in file and stored only the cache directory in main memory. They did not consider the re-accessing of cached data and there was no mechanism for managing cached data. This approach was only reasonable for websites with infrequent updates of data and CGI scripts. Hence, Swala was not adopted for use in current dynamic websites.

Dykes, Jeffery, and Das proposed a distributed web caching method based on Server-Directed proxy sharing Protocol (SDP) [2], in which proxy servers located cache copies by looking in local metadata directories and propagated the metadata by piggybacking it onto data transfers. SDP implemented Internet-wide cache sharing through cooperating proxy server caches. They did not consider data transfer time between proxy and proxy or proxy and server.

Asaka, Miwa, and Tanaka proposed a hash-based query caching method using both a hash function and a query caching method [13]. This method could find cached objects among several cache servers by using only one query message. However, every cache server kept the whole other cache servers’ cached information in its memory causing worse memory utilization and more difficulty with memory management in terms of increasing the number of cache servers. They did not allow the duplication of data among cache servers resulting in a skewed load balance among cache servers and longer response time for client requests when popular data was requested to a specific cache server with convergence. They also did not consider the consistency of cached data among cache servers.

Vahdat and Anderson proposed a framework for monitoring source files, automatically invalidating the result whenever the source changed [18]. An application of
this is the monitoring system for cached dynamic web contents. They modified a version of the Apache HTTP server to cache CGI results and used their monitoring system to invalidate cached entries. Their model provided neither any method for limiting the size of the cache nor intelligently replacing cache entries, nor did it permit other methods of invalidation. It required file system access on every cache lookup because they did not maintain an in-memory cache table. This approach caused increasing response time and increasing cache misses to occur. They also could not update cached dynamic contents consistently and completely.

Caching dynamic web data had been studied by IBM [3,10,11]. They used the DUP (Data Update Propagation) algorithm for updating cached dynamic data. They implemented their own server application to insert, delete, and update cached items. There were two main drawbacks in this approach. First, it required a server application that managed the cached items. This could be a nontrivial task and required significant overhead maintaining cached items. Second, for every dynamic request, the web server still needed to invoke the application, even if it only returned a cache hit. For call mechanisms such as CGI, operating system overhead for this call was significant. This approach increased a server’s load and led to decreased performance gain.

We present new cache algorithms and new cache protocols. These algorithms consider the size and access frequency of a document while caching it in a web server’s memory. These algorithms are also designed to adapt to current web client’s randomized request patterns. These protocols have intelligent cache algorithms, in-memory cache maintenance table, as well as a consistent and complete cache maintenance mechanism. Our approach does not need extra applications for managing static and dynamic web
contents. Our approach also achieves less communication cost among web servers using broadcast mechanisms. Thus, by using these cache protocols and cache algorithms for web servers, we can build scalable web servers supporting very fast response times, high memory hit rates, and excellent load balancing of cooperative distributed web servers.

2.2 Fast IP Routing

The Internet began with a simpler form of hierarchy in which 32 bit addresses were divided into a network address and a host number, so that routers could only store entries for networks. For flexible address allocation, the network addresses came in variable sizes: Class A (8 bits), Class B (16 bits), and Class C (24 bits). So when a router receives a packet, it reads the packet’s network id only and forwards it easily to the interface matching the prefix. However, as the network size has been growing up, IP addresses have started to run out. To reduce backbone router table size, CIDR (Classless Inter-Domain Routing) [34] and Subnetting have been proposed and widely adopted to solve the IP address space scarce problem [35].

To reduce routing table space, aggregation is done aggressively. Suppose all the subnets in a big network have identical routing information except for a single, small subnet that has different information. Instead of having multiple routing entries for each subnet in the large network, just two entries are needed: one for the big network and a more specific one for the small subnet (which has preference, if both should match). This results in better usage of the available IP address space and decreases the amount of routing table entries. On the other hand, the processing power needed for forwarding lookup is increased. With CIDR, routers must find out the best route for IP packets forwarding, which complicates the IP lookup. Currently, this process is mainly done in software and several schemes have been proposed for hardware implementation.
2.3 Existing Approaches to IP Lookup

We survey existing approaches to IP lookups and their problems. Most IP table lookup schemes can be classified as several categories. We discuss approaches based on level compression based lookup, caching based lookup, hash based lookup, proposals for protocol changes to simplify IP lookup, and hardware solutions based on parallelism. For the convenience, we use BMP as a shorthand for Best Matching Prefix.

2.3.1 Level Compression-Based Lookup

Algorithms in this category use n-bit B-tree or hash [36,37,38]. Because they compare n bits simultaneously rather than 1 bit when traversing the tree, the height of the tree tends to be kept very small, which results in very fast search. But their performance is still affected by memory usage very much. In case of 4-bit B-tree, a parent node can have up to $2^4$ children nodes. If the number of children nodes is less than 16, there will be some empty nodes, which result in duplicated pointers that make table update very hard. Due to these reasons, the algorithms are apt to have very large routing table image, which causes slow update. These approaches either use complicated data structures which result in high complexity for updating/building the forwarding table, or they are not scalable to fit in IPv6. To overcome this problem, pre-computation scheme was proposed such as binary search on prefix length [39,40], Lulea compressed tries [41], or binary search on intervals [42,43]; however, adding a prefix can cause the data structure to be rebuilt. Thus, pre-computation-based schemes have a worst-case update time of $\Omega(n)$.

The basic scheme which inspired proposing some new approaches is radix tree, [44] or the binary trie [45] Binary trie is a simple data structure which represents strings with paths from the root to the leaf or any node in the middle. Each address lookup is performed by scanning the address bit by bit and matching it along a path in the trie. The
The worst-case cost of an IP lookup is thus $O(W)$, where $W$ is the address length (32 in IPv4, 128 in IPv6). This scheme requires $O(N)$ space, where $N$ is the total number of prefixes in the forwarding table. The main problem with trie is that it keeps some nodes correspond to any data element and thus, wastes memory space. The structure is inflexible in terms of the number of branches in the internal nodes. All modified approaches try to overcome these shortcomings in one way or another [46,47,48].

Classical fast lookup techniques such as hashing and binary search do not directly apply to the Best Matching Prefix (BMP) problem since they only do exact matches. A modified binary search technique, originally due to Butler Lampson, is described in [49]. However, this method requires $\log_2 2N$ steps, with $N$ being the number of routing table entries. With current routing table sizes, the worst case would be 17 data lookups, each requiring at least one costly memory access. As with any binary search scheme, the average number of accesses is $\log_2 (2N) - 1$. A second classical solution would be to reapply any exact match scheme for each possible prefix length [50]. This is even more expensive, requiring $W$ iterations of the exact match scheme used (e.g., $W = 128$ for IPv6).

A Patricia trie is a widely deployed algorithm for searching entries with varying network masks. A Patricia Trie modifies binary trie by compressing the paths and eliminating unnecessary nodes [51]. It has been implemented in the BSD kernel [52], NET/3, and Linux etc. While a Patricia trie runs very fast when an entry satisfying searching conditions exists, it suffers slow searching time when there is no such entry. This is because it has backtracking. A Patricia trie is the base of few new methods which have been proposed recently [38,53]. All these approaches try to check several bits of IP
address at each step instead of checking only one bit. Since checking several bits instead of one bit may deteriorate memory usage and leave many memory space unused [54], all these approaches try to minimize the memory waste in one way or another.

Waldvogel and Varghese proposed a binary search over the possible prefix lengths, requiring O(logW) steps [39]. For each test in the binary search, a hash table is consulted, requiring to break the prefixes into several hash tables which all together require O(NlogW) space. Srinivasan and Varghese in [55,56] propose to expand the original prefixes into an equivalent set of prefixes with fewer lengths and then, apply a dynamic programming technique to overall index structure in order to optimize the memory usage. Nilsson and Karlsson [37] proposed a specific case of [55] by locally optimizing memory usage in each step. And finally, a new scheme from Lulea University of Technology, [57], endeavors to reduce the size of the routing table so that it fits in the cache [40].

The DP-Trie, Dynamic Prefix Trie, [58], from IBM research center proposes a binary tree data structure for matching the longest prefixes. Beside keep prefixes, DP-Trie tries to reduce the height of trie by keeping the index of bit position that differentiates the prefix in the node from prefixes in the subtrie rooted in that node.

Lampson [59], Suri [60], and Yazdani [61] applys the binary tree search and extends it to a multiway tree by treating each prefix as a range and identifying each range with L, low, and H, high. Considering a fairly uniform distribution, the average search time on both of these methods should take O(log_2N) in the binary tree and O(log_mN) in the m-way tree.

2.3.2 Caching-Based Lookup

For years, designers of fast routers have resorted to caching to claim high speed IP lookups. This is problematic for several reasons. First, information is typically cached on the entire address, potentially diluting the cache with hundreds of addresses that map to
the same prefix. Second, a typical backbone router of the future may have hundreds of thousands of prefixes and be expected to forward packets at Gigabit rates. Although studies have shown that caching in the backbone can result in hit ratios up to and exceeding 90 percent [62,63], the simulations of cache behavior were done on large, fully associative caches which commonly are implemented using CAMs (Content Addressable Memories). CAMs are usually expensive. It is not clear how set associative caches will perform and whether caching will be able keep up with the growth of the Internet. So caching does help, but does not avoid especially in view of current network speedups.

As algorithms in this category use caching, search time tends to be short [57,64]. But there are some problems when they are used in backbone routers. The first problem is that the caching is not useful in large networks. The temporal correlation of destination address of traffic is very important for the efficient operation of cache based algorithms, and this property called locality affects cache performance very much. However, as networks of today become very large in size, there is not much of locality. Therefore, using caching cannot be a good approach to speed up searching in backbone routers [62]. The other problem is caused by the memory size for caching in large networks. So routers sometimes compresses routing table image, which results in the reduced memory requirement. Compressed image means that the router can load more entries in the same sized cache memory. But the compressed routing table makes updating very hard and slow. Each entry is correlated with the entire table. To update an entry, the router should update the table that is not compressed, and then compress it again. Therefore it takes much time for the router to process each update request.
2.3.3 Hash-Based Lookup

An alternative approach for IP routing table search is hashing [65]. Hash based lookup algorithms use hash functions to increase their searching velocity [66]. An entry can be found nearly in O(1) time by using hash. However, when hash is used for IP table lookup, it demands a quite large memory because it fits the exact prefix matching, not the longest prefix matching. So the algorithm should extend the network id length of an entry to be the same as the length of the hash key. Unlike the tree structures, hashing stores whole addresses and the assumption is that the number of addresses is limited and can be kept in cache. This method does not take advantage of the hierarchical address structure and is not well suited for Internet applications. Furthermore, Newman [62] reports poor cache hit ratios for backbone routers.

2.3.4 Protocol-Based Solutions

One way to get around the problems of IP lookup is to have extra information sent along with the packet to simplify or even totally get rid of IP lookups at routers. Two major proposals along these lines are IP Switching [62] and Tag Switching [67,68,69,70]. Both schemes require large, contiguous parts of the network to adopt their protocol changes before they will show a major improvement. The speedup is achieved by adding information on the destination to every IP packet.

In IP Switching, this is done by associating a flow of packets with an ATM Virtual Circuit; in Tag Switching, this is done by adding a “tag” to each packet, where a “tag” is a small integer that allows direct lookup in the router’s forwarding table. Tag switching is based on a concept originally described by Chandranmenon and Varghese [67,68] using the name “threaded indices.” The current tag switching proposal [70] goes further than threaded indices by adding a stack of indices to deal with hierarchies.
Neither scheme can completely avoid ordinary IP lookups. Both schemes require the ingress router (to the portion of the network implementing their protocol) to perform a full routing decision. In their basic form, both systems potentially require the boundary routers between autonomous systems (e.g., between a company and its ISP or between ISPs) to perform the full forwarding decision again, because of trust issues, scarce resources, or different views of the network. Scarce resources can be ATM VCs or tags, of which only a small amount exists. Thus towards the backbone, they need to be aggregated; away from the backbone, they need to be separated again.

Different views of the network can arise because systems often know more details about their own and adjacent networks, than about networks further away. Although Tag Switching addresses that problem by allowing hierarchical stacking of tags, this affects routing scalability. Tag Switching assigns and distributes tags based on routing information; thus every originating network now has to know tags in the destination networks. Thus while both tag switching and IP switching can provide good performance within a level of hierarchy, neither solution currently does well at hierarchy boundaries without scaling problems.

2.3.5 Hardware Solutions

Hardware solutions can potentially use parallelism to gain lookup speed. For exact matches, this is done using Content Addressable Memories (CAMs) in which every memory location, in parallel, compares the input key value to the content of that memory location. Some CAMs allow a mask of bits that must be matched. Although there are expensive so-called ternary CAMs available allowing a mask to be specified per word, the mask must typically be specified in advance. It has been shown that these CAMs can be used to do BMP lookups [71,72], but the solutions are usually expensive.
Large CAMs are usually slower and much more expensive than ordinary memory. Typical CAMs are small, both in the number of bits per entry and the number of entries. Thus the CAM memory for large address/mask pairs (256 bits needed for IPv6) and a huge amount of prefixes appears (currently) to be very expensive. Another possibility is to use a number of CAMs doing parallel lookups for each prefix length. Again, this seems expensive. Probably the most fundamental problem with CAMs is that CAM designs have not historically kept pace with improvements in RAM memory. Thus a CAM based solution (or indeed any hardware solution) runs the risk of being made obsolete, in a few years, by software technology running on faster processors and memory. Gupta, Lin, and McKeown and Huang, Zhao, and Pan propose a usage of pipelining to perform several lookups at the same time [66,73,74]. Newman and Minshall, and Partridge propose employing a cache to hold the results of recent lookups. It is possible to achieve a 90% hit rate [62,63] but by employing a large and very expensive cache based on the CAM technology. All the hardware solutions suffer from very high costs, especially when applied to large backbone routers, and they do not scale easily.

In summary, all existing schemes have problems of either performance, scalability, generality, or cost. Lookup schemes based on tries and binary search are (currently) too slow and do not scale well; CAM solutions are expensive and carry the risk of being quickly outdated; tag and IP switching solutions require widespread agreement on protocol changes and still require BMP lookups in portions of the network; Hash solutions need a quite large memory however, it take poor cache hit ratios for backbone
routers; finally, locality patterns at back bone routers make it infeasible to depend entirely on caching.

We now describe a scheme that has good performance in lookup and update, excellent scalability, and does not require protocol change. Our scheme also allows a cheap, fast software implementation.
CHAPTER 3
FAST AND SCALABLE INTERNET SERVICE MODEL

To serve fast and scalable Internet service, we propose FASTS model, which are composed of web cache algorithms, cache protocols, and fast and scalable IP forwarding approach. To achieve fast and scalable Internet service, we must solve current web server cache and IP forwarding problems. In web server cache, the cached data can be easily become out-of-date and caching dynamic data can seriously reduce web server’s performance. Another problem with caching dynamic pages is determining what pages should be cached and when a cached page has become obsolete. Thus, to improve the performance of web server, key requirements for many websites providing static and dynamic data are to completely and consistently update pages, which have changed, and to provide excellent load balance mechanism among web servers. In IP forwarding, the major remaining bottleneck in the Internet router design is the slow, software-based IP lookup procedures. There are four requirements for a good lookup scheme such as fast search time, less memory consumption, good scalability, and fast update time.

In FASTS model, for web server cache algorithm, we propose Randomly select Web server using Global memory mechanism (RWG), select Web server by Round-robin method and use Global memory mechanism (WRG), Randomly select Dedicated web server using Global memory mechanism (RDG), Directly Selected and Limited Look up (DSLL) mechanism, and Randomly Selected and Limited Look at neighbor (RSLL) mechanism, for web server cache protocol, we propose Direct request distribution and Load balancing Cache Protocol (DLC) and Random request distribution and Load
balancing Cache Protocol (RLC), for fast IP forwarding, we propose Fast and Scalable IP Lookup and uPdate scheme (FSLP).

We can combine any algorithm and protocol, which depend on the purpose of using web servers. We offer two kind of different web server control mechanism such as direct control and distribute control. The characteristic of direct control for web server is centralized-web server cache content management and central control of web client request distribution and load balancing among web servers. To support this mechanism, Central Server, web server, and Request Distributing and Load balancing Manager (RLM) are needed. In distributed control of web servers, the system is composed of Central Server, web server, and Distributor. The characteristic of distributed control for web server is distributed-web server cache content management, and randomly control of web client request distribution and load balancing among web servers. The Figure 3-1 shows the detail information about FASTS model architecture.

All web cache algorithms are based on BAFF (Byte Access Frequency Factor) concept. It is designed for supporting Internet services with multimedia, and static and dynamic web contents. These algorithms have a special feature treating hot documents in a web server. These cache algorithms are not only suitable for current web client’s request pattern such as nonuniform discrete random variation but also work well in ideal web client’s request pattern such as uniform discrete random variation. This means that our cache algorithms are applicable to any kind of Internet environment.

We propose new cache protocol DLC and RLC. DLC and RLC caching protocols update cached static and dynamic contents completely and consistently on a web server. They also improve response time for web client requests and load balancing among web
servers as well as cache static and dynamic data cooperatively among web servers. DLC and RLC allow duplication of a dynamic web content among web servers. Thus by using our cache algorithms and cache protocols for static or dynamic web contents, we can
build scalable web servers supporting very fast response time, high memory hit rate, excellent load balancing, and completely and consistently updating of cached web data in distributed cooperative web servers.

For fast and scalable IP lookup and update algorithm in IP forwarding table, we developed a new grouping algorithm, which converts a large forwarding table to several small forwarding tables with even intervals. Our new algorithm has an expected run time that is asymptotically better than those of existing algorithms.

We reduced the general binary search concepts to the longest matching prefix problem in IP router. To group the large entries of the forwarding table into small and even interval group entries, we look at the first q bits of the each IP forwarding table entries. The total number of each IP forwarding table entries and the longest prefix length of the IP forwarding table entries determine the q bits. For any given x IP address for searching the longest matching prefix in forwarding table only takes an expected time of $O(1+\log (n/2^q))$. For instance, if q is chosen to be $\log n$, then the expected search time is only $O(1)$. The memory requirement is still $O(n)$.

These small forwarding tables are given faster and more accurate search for any given IP address in a IP router. Thus, this approach achieves faster and more accurate founding the longest matching prefix for any given IP address. This is the main factor of the speed up in IP lookup search procedure in our algorithm. Our algorithm has good performance in lookup, excellent scalability, and does not require protocol change. Our algorithm is faster in a large IP routing table than in a small one. This is good for the current large IP routing table data for speeding the lookup and update time.
CHAPTER 4
BYTE ACCESS FREQUENCY FACTOR BASED WEB CACHING ALGORITHMS
FOR STATIC AND DYNAMIC WEB CONTENTS

4.1 Design of Scalable Web Server Architecture for BAFF Concept

In this section we provide details on the architecture that we have in mind. Ours is a multi-node web server architecture that consists of a Central Server and multiple web servers as shown in Figure 4-1. The Central Server communicates with the other servers through a high-speed network, like FDDI, or a switched ATM network. A cluster of commodity web servers is also connected through a high-speed network. Each web server has some disk and memory space associated with it. An identical copy of the Central Server runs on each web server. The web servers are designed to work as a collection of cooperating web servers with each web server containing persistent data files.

The web server implementation is based on a novel peer-to-peer process structure that enables one web server to access memory or persistent data files at other web servers. If a requested document is missing at the web server’s memory where the request originates, then the web server looks at a limited number of other web servers’ memory to serve the request to a web client. If the document cannot be found in these servers, a request to the Central Server will be made to send the requested document to the web client.

Each web server contains a Cache Maintenance Process (CMP) and a Web Server Process (WSP) that communicates with other web servers and serves a web client’s request by passing it through a communication network such as the Internet or Intranet.
The Cache Maintenance Process of each web server updates cached static and dynamic data consistently and completely when cached data has changed. Each web server’s memory is normally managed locally. High BF (Byte access Frequency factor of document) value documents are kept in the local cache and the Cache Maintenance Process maintains them. The detailed operations of the cache are discussed in later sections. Figure 4-2 shows the detailed functionality of a web server.

Incoming web client requests go through the Distributor that distributes the requests either in a round robin or a random fashion among the web servers. The intent of using round robin or random-distribution for web client requests is to balance the load among web servers. Our experimental results indicate that random-distribution results in
better response times than round robin when the documents are small and the number of requests is large (1,000,000 for example). Round robin yields better response times than random-distribution when the documents as well as the number of web client requests are large.

The Central Server stores a complete set of static and dynamic web documents and performs all document maintenance jobs. Documents that have higher BF values will be copied to and cached in the web servers. The Central Server will not be the performance bottleneck for web contents since the probability of any request reaching it is low. Web Server Processes keep access logs. At the end of each predefined period, all access logs are sent to the Central Server for updating the document information. The Central Server uses these access logs to re-calculate the BF value of each document. Based on the new information, each web server will change its cache contents.
4.2 Central Server Mechanism and Operation in Cooperative Web Servers

Communication between the Central Server and web servers is done using a broadcast mechanism. There are many reasons why a broadcast mechanism is used. First, the web server architecture can be easily scaled up by installing a web server to or removing it from a high-speed network. It gives more flexibility and scalability than other methods. Second, establishment of a network connection and network overhead for communicating between the Central Server and web servers are minimal. Communication is much more efficient between them. Third, the Central Server can easily optimize its I/O for transmitting web documents. Finally, static and dynamic data consistency between the Central Server and web servers can be maintained easily and completely. When a document is modified, the Central Server can broadcast the document maintenance information to web servers for avoiding stale information being accessed by web clients. This approach guarantees the freshness and consistency of cached static and dynamic web contents.

On a single server website, a simple list is sufficient to maintain cached data. However a more complicated cache mechanism must be employed when data are processed from many web servers to maintain data efficiently and consistently. A document table kept in the Central Server will be used for this caching mechanism.

After a predefined period, a new document table must be generated to reflect any changes in the documents. For each web server, the document request count can be obtained from the access log generated by the Web Server Process. By merging all the access logs from the web servers, the Central Server calculates the total count and the BF value of each document. When this calculation is finished, the Central Server will sort all the documents according to the BF values in descending order and generate the BF index.
Having done all of this, starting with the highest BF value document, the Central Server will assign new documents to web servers using a round-robin pattern until each web server fills out its memory. This approach achieves good load balance among web servers and faster response times to a web client, because hot document requests are evenly distributed among web servers. To maintain all documents efficiently and easily, the Central Server has hash table for all documents. This approach achieves fast retrieval of documents, consumes less memory, and maintains all documents easily. When a web server receives these new documents, it compares the new documents with the previously stored ones and updates its cache information.

4.3 Web Server Mechanism and Operation in Cooperative Web Servers

Each web server has two functional processes, viz., the Cache Maintenance Process (CMP) and the Web Server Process (WSP). When the Web Server Process receives a web client’s request, it will search for the document in its cache first. When a cache hit occurs, the document is sent to the web client. When the document is not found in its cache, then it communicates with a limited number of other web servers to locate the document. When a cache hit happens, the document is sent to the user. The limited number of look-ups on other web servers is at most 30% of the total number of web servers. This approach guarantees a limited response time to a web client for serving the requested document. When a cache miss happens, the Web Server Process starts a CGI process with the help of the Cache Maintenance Process. A CGI process then requests the document from the Central Server. In both cases, the Web Server Process records the result of the web client request in its access log file. The Cache Maintenance Process performs two main functions:

- It maintains the cache contents of the local cache.
• It performs the corresponding actions when it receives a broadcast message from the Central Server.

When a web server receives a broadcast message from the Central Server, the Cache Maintenance Process buffers the message and examines the identification of web server field and the type of broadcast field from the header information. If the message is sent to it, it takes the message and performs the corresponding procedure.

4.4 Byte Access Frequency Factor Based Caching Mechanism

We propose new cache algorithms based on the Byte Access Frequency Factor (BAFF) [75]. We show the performance of this algorithm compared with conventional LRU and LFU algorithms in a later Section. We first explain the basic concept of our web cache replacement algorithm.

Consider a generalized Internet server that containing $N$ unique dynamic and static web documents. The documents have different sizes and denote the size of document $i$ in bytes as $S_i$. Let $M$ be the total memory size available in the web server such that $M > S_i$ for all $i$. Let $F_i$ be the access frequency that a user requests document $i$. $F_i$ is known from past history of document access from web clients. We define $BF_i$ be the Byte Access Frequency Factor of document $i$ and is given by Equation 4-1.

$$BF_i = \frac{\text{Number of Access for Document } i}{\text{Size of Document } i} = \frac{F_i}{S_i} \quad (4-1)$$

The BAFF based cache algorithm is mainly based on the idea that documents with higher $BF_i$’s are always cached first when the cache replacement situation is occurred. This characteristic will guarantee the best performance in the Internet environment.

Consider a particular situation that the documents in the cache are sorted according to their $BF_i$ value in ascending order as shown in Figure 4-3. A set of documents with a
total size $S$ in the cache is going to be replaced by a new set of documents not in the cache but with the same total size $S$. We assume that the summation of the $BF_i$ value of the original set in the cache is larger than that of the new set. Thus, the overall hit rate of the cache after the replacement will be lowered increasing network traffic, disk I/O cost, and most importantly the response time for dynamic data. Therefore, the hit rate of the cache will be increased if no replacement such as the one mentioned above is made. That is to say that the best cache performance is obtained when the cache always stores the documents with the highest $BF$ first. Note that LFU algorithm will always replace documents in the cache when new document access happens. This will make the cache operate in nonoptimal manner. This situation is improved when the BAFF based algorithm is used.

![Figure 4-3 Suboptimal cache replacement operation](image)

Figure 4-3 Suboptimal cache replacement operation
We explain the cache replacement mechanism of BAFF based algorithm. Consider an example that N documents in the cache are sorted according to their BF in ascending order. As a result, the document with the highest BF is stored at the bottom of the cache such that \( BF_i < BF_j \) if \( i < j \). When a new document, not in cache, with \( BF_k \) and size \( S_k \) is accessed, a replacement decision has to be made. The new document will only replace the first \( j \) documents such as \( D_1, D_2, D_3, \ldots, D_j \) at the top of the cache if and only if

\[
BF_{new} \geq \sum_{i=1}^{j} BF_i \text{ and } \sum_{i=1}^{j} S_i \leq S_{new} \leq \sum_{i=1}^{j+1} S_i \quad (4-2)
\]

\( j \) is the smallest number of the documents in the cache and it has a total size greater than equal \( S_k \). \( j \) is then chosen from the above criteria. Using this mechanism, documents with a relatively higher BF will be cached. Therefore, the maximum cache hit rate can be obtained if the BAFF based algorithm is used. This BAFF based algorithm also considers size of a document in this mechanism resulting in reducing the I/O cost and response time for servicing the request to a web client. Thus, we can earn better performance such as response time and reducing overload of the Central Server.

From the basis of the BAFF concept, we developed new web caching algorithms. In the following, we present new cache replacement algorithms in applicable to scalable web servers for achieving excellent performance in response time, load balancing among web servers, and cache hit rate for web client’s requests compared with conventional LRU and LFU algorithms.

4.5 Randomly Select a Web Server Global Memory Mechanism Algorithm

When a web client request enters through the network the Distributor distributes the request to a web server chosen randomly. The selected web server first checks if the
requested document is in its memory. If the document is in the cache, the server sends the requested document to the web client and updates the BF value of that document. If the document is not in the cache, it looks for the document in a limited number of other web servers’ memories. If the requested document is located in another server’s memory, the server sends the requested document to the web client and updates BF value. After making these attempts if the document cannot be found the document is requested from the Central Server. In this case, after receiving the document from the Central Server, the server sends it to the web client. In any case the BF value of the new document is computed to decide if it should replace any document(s) in the cache. If needed a replacement operation is performed.

Algorithm 1. Randomly select a Web server using Global memory mechanism (RWG)

Step 1. The Distributor randomly selects a web server.

Step 2. If the requested document exists in the selected web server’s memory,
   Service the web client request,
   Increase the value of the number of requests for that document,
   Update the BF value of that document, quit and go to Step 1;
Else repeat
   Look at the colleague’s web server memory
   until reaching the limited number
   If the requested document is found in another web server’s memory,
   Service the web client request,
   Increase the value of the number of requests for that document,
   Update the BF value of that document, and quit go to Step 1;
Else send a request to the Central Server,
   If the document arrives from the Central Server,
   Service the web client request,
   Increase the value of number of requests for that document,
   Update the BF value of that document, and Compare the BF value of the new
document with those of existing documents in selected web server’s memory.
Check if a_j can be found as in equation (2).
If so:
   Replace the existing document(s) with new document,
   Send this information to the Central Server, quit and go to Step 1.
Else store the new document in the selected web server’s disk, quit and go to Step 1.
Else send a message: “Document not found” to the web client, quit and go to Step 1.

Figure 4-4 RWG web caching algorithm
4.6 Round Robin method Global Memory Mechanism Algorithm

The WRG algorithm is quite similar to the RWG. The only difference is that the request distribution method is Round Robin.

Algorithm 2. Select a Web server by Round Robin method and use Global memory mechanism (WRG)

Step 1. Select a web server by Round Robin method.

Step 2. If the requested document exists in the selected web server’s memory,
   Service the web client request,
   Increase the value of the number of requests for that document,
   Update the BF value of that document, quit and go to Step 1;
Else repeat
   Look at the colleague’s web server memory
   until reaching the limited number
   If the requested document is found in another web server’s memory,
   Service the web client request,
   Increase the value of the number of requests for that document,
   Update the BF value of that document, and quit go to Step 1;
Else send a request to the Central Server,
   If the document arrives from the Central Server,
   Service the web client request,
   Increase the value of number of requests for that document,
   Update the BF value of the new document with those of existing documents in selected web server’s memory.
   Check if a \( j \) can be found as in equation (2).
   If so:
       Replace the existing document(s) with new document,
       Send this information to the Central Server, quit and go to Step 1.
   Else store the new document in the selected web server’s disk, quit and go to Step 1.
Else send a message: “Document not found” to the web client, quit and go to Step 1.

Figure 4-5 WRG web caching algorithm

4.7 Dedicated Web Server Global Memory Mechanism Algorithm

This algorithm (Randomly select a Dedicated web server using Global memory mechanism (RDG) algorithm) partitions the web servers into two groups: High BF servers and Low BF servers. High (Low) BF servers are dedicated to serving documents with “high” (“low”) BF values. “High” and “low” BF values can be defined
appropriately. For example, they can be defined empirically as those that yield the best performance. When a web client’s request enters the network the Distributor sends it to a dedicated web server randomly. If the request belongs to High BF, it is sent to a High BF server randomly. Otherwise, it is sent to a Low BF server randomly.

Algorithm 3. Randomly select a Dedicated web server using Global memory mechanism (RDG)

Step 1. The Random Distributor checks the BF value of the document
   If the request is a High BF document,
     Send the request to a random High BF web server
   Else
     Send the request to a random Low BF web server

Step 2. Case 1: High BF web server
   If the requested document exists in the randomly selected web server’s memory,
     Service the web client request,
     Increase the value of the number of requests for that document,
     Update the BF value of that document, quit and go to Step 1;
   Else repeat
     Look at a colleague’s web server memory
     until the limited number is reached
   If the requested document is found in other web server’s memory,
     Service the web client request,
     Increase the the number of requests for that document,
     Update the BF value of that document, quit and go to Step 1;
   Else send the request to the Central Server,
     If the document arrives from the Central Server,
       Service the web client request,
       Increase the number of requests for that document,
       Update the BF value of that document,
     Else sends the message: “Document not found” to the web client, quit and go to Step 1.

Case 2: Low BF web server
   The same as in RWG

Figure 4-6 RDG web caching algorithm

High BF servers always keep high BF documents in their memory. If we have multiple high BF servers, the memory content of each high BF sever is different. Low BF servers keep low BF documents in their memory. Note that the BF value of a document
can change dynamically and hence its labeling (as either a high BF document or a low BF
document) might also change.

Our simulation results indicate that the performance of High BF servers is very good
with respect to response time and cache hit rate but the performance of Low BF servers is
low. Thus the overall performance gain is no better than those of RWG and WRG
algorithms for the same number of web servers. But this algorithm has better
performance gain in response time compared with the LRU and LFU algorithms. If one
wants to employ the RDG algorithm in web servers, we suggest that the number of Low
BF servers be increased to achieve the same performance gain as those of the RWG and
WRG algorithms.

4.8 Simulation Environment

We have conducted experiments to compare the performances of our three
algorithms with those of the LRU and LFU algorithms. We have developed a web client
access pattern generator instead of using real data access logs in specific web servers.
The reason for using our own web client access pattern generator is that specific web
servers’ logs cannot stand as a general guide for web clients’ request patterns. In other
words, a single web server’s logs may not exhibit general patterns that one could expect
to see over a variety of such servers. This access pattern generator is designed to follow
current web clients’ request patterns based on previous research [2,3,4,6,12]. We used
Algorithm Generate 1 [76] to generate web clients’ request patterns.

We simulated our three algorithms for two kinds of web client request patterns. The
first pattern is to generate the requests uniformly randomly. The second pattern is to
generate the requests nonuniformly randomly. The nonuniform random distribution could
be anything and is specified by the Probability Distribution Function (PDF). The
algorithm Generate 1 of [76] is employed to generate a nonuniform random distribution. It is conceivable that certain documents are more popular than the others. In general the request patterns for documents at any given time can be specified with a PDF. In Algorithm Generate 1, we inserted the concept of High BF value factor to generate web client requests. In next the Section, we provide comparison results of our three algorithms with the LRU and LFU algorithms.

We chose nine web servers. To begin with, the number of times each document has been accessed is assigned randomly. This value was changed during the simulation. The size of each document was also assigned randomly from 100 bytes to 3.2 Mbytes.

There are numerous variables that could be changed in the experiments. We have carefully chosen the variables to be changed. In each measurement, we changed the values of one or more variables such as the total number of documents, the number of web client requests, the memory size of each server, and so on. Each experiment was repeated over 100 times and the averages taken. The total number of documents stored in the Central Server varied from 4000 to 10,000. The memory size of each web server varied from 64 Mbytes to 512 Mbytes. The number of web client requests also varied from 10,000 to 1,000,000. The total memory size of the web servers is 10% - 20% of the total document size.

To calculate the response times accurately, we associated different weighting factors to account for delays involved in various scenarios. If the request is serviced from the first selected web server’s memory, we gave one weighting factor for each document. If the request is serviced from a neighbor web server’s memory, we gave 1.5 weighting factor for each document. If the request is serviced from the Central Server’s memory, we
gave two weighting factor. If the request is serviced from the Central Server’s disk, we
gave three weighting factor. To analyze the performance gain of our algorithms, we used
two values, namely, the response time for the web client’s request and the cache hit rate.
We assumed that the web servers ran concurrently to service a web client’s request. We
also assumed that each web server’s memory contents were updated using one of the
cache protocols proposed by us [75,77]. The redundancy of web contents among the web
servers was also controlled using one of our cache protocols [75,77].

In the RDG algorithm, we partitioned the web servers into High BF servers and
Low BF servers. We decided if a document had low BF or high as follows. We
accumulated the BF values of all documents and determined a threshold percentage (30%
for example). Those documents that have the highest BF values and which account for
30% of the total BF value will be designated as high BF documents. The others will be
low BF documents. This percentage can be chosen empirically. There could be a
situation where some of the clients (and their documents) are preferred over the others.
For example, a merchant may have preferred customers. In this case the merchant might
classify the documents of preferred customers as high BF documents. Other possibilities
exist as well. We changed the number of High BF servers from one to eight.

4.9 Simulation Results

In our simulations, as indicated above, we have changed the values of one or more
variables at a time and studied the effects. We present our results in several cases. Each
case has two results, one for nonuniform and one for uniform distribution of web client
requests. In the graphs to follow, RDG1 means that there is one low BF server among
nine web servers.
4.9.1 Performance Evaluation for Small Number of Documents Set

In this case the set of requests has 4000 documents. The total response time of the RWG and WRG algorithms is up to 57% better than that of LRU in the non-uniform case. Here the memory size is 512 Mbytes. When the memory size is 64 Mbytes, the response time is 10% better. In the case of 128 Mbytes and 256 Mbytes, RWG and WRG were, respectively, 19% and 34% better than the LRU.

Between the RWG and WRG algorithms, WRG has better response time when the number of requests is small such as 10,000 or 100,000. In contrast, RWG has better response time for large number (say 1,000,000) of requests. The performance of RWG and WRG improves with an increase in the memory size. For instance, the response time is 57% better with 512 Mbytes than with 64 Mbytes. In comparison, the same percentage for the LRU and LFU are 19% and 15%, respectively. RDG also has a better percentage than the LRU and LFU. RDG1 is better than RDG2. Figures 4-7 and 4-8 show the results of each algorithm’s performance gain for total response time in the case of 4000 documents.

![Figure 4-7 Total response times for 4000 documents in the nonuniform case](image-url)
Figures 4-8 and 4-10 show the memory hit rates in the case of 4000 documents. In the nonuniform case, RWG and WRG had 8.5 times more memory hits than the LRU. Here the memory size was 64 Mbytes and the number of requests was 1,000,000. When the memory size is increased from 64 Mbytes to 512 Mbytes, both RWG and WRG improve their memory hits by a factor of 5.7. The same factor for LRU and LFU are 7.7 and 7.4, respectively. RWG has a better memory hit rate than WRG in the case of 1,000,000 requests. This was also true when the client requests were uniform. RDG has better memory hit rates for all request patterns than the LRU and LFU algorithms.

Figure 4-9 Total memory hits for 4000 documents in the nonuniform case
In the uniform case, RWG and WRG have more memory hits than in the nonuniform case. Both of them earned up to 8.8 times more memory hits with 64 Mbytes memory than the LRU and LFU algorithms for all request patterns. In the rest of the cases such as 128 Mbytes, 256 Mbytes, and 512 Mbytes the same performance gain was seen. RWG and WRG had 5.8 times more memory hits in the 512 Mbytes case than in the 64 Mbytes case. LRU has eight times more memory hits with 512 Mbytes than with 64 Mbytes. LFU has seven times more memory hits.

In the case of 5000 documents, the results were very similar. RWG and WRG had 46% better response times than the LRU for all request patterns (the memory size being 512 Mbytes). Note that this improvement is less than in the case of 4000 documents. RWG and WRG had up to 46% better response times with 512 Mbytes than with 64 Mbytes. However, the same percentage was only 7% and 6.3% for LRU and LFU, respectively. WRG has better response times than RWG when the number of requests is small (such as 10,000 and 100,000). When the number of requests is large, RWG has better response times. RDG has better response times than LRU and LFU for all request patterns. Figures 4-11 and 4-12 show response times in the case of 5000 documents.
Figure 4-11 Total response times for 5000 documents in the nonuniform case

Figure 4-12 Total response times for 5000 documents in the uniform case

Figure 4-13 shows memory hit rates of various algorithms in the case of 5000 documents. In both the nonuniform and uniform cases, RWG has better memory hits than the WRG. In the nonuniform case, RWG had 8.6 times more memory hits than the LRU with 64 Mbytes. In the uniform case, RWG had 9.2 times more memory hits than the LRU with 64 Mbytes and 10,000 requests. In the rest of the cases such as 128 Mbytes, 256 Mbytes, and 512 Mbytes the results were the same when the request patterns were nonuniform. With an increase in memory size from 64 Mbytes to 512 Mbytes both RWG
and WRG improved their memory hits by a factor of 6.1. The same factor for LRU and LFU were 8.3 and 8, respectively.

It is clear from the results for 4000 and 5000 documents that RWG and WRG are better suited as web caching algorithms than the LRU and LFU.

![Memory Hit Ratio (Doc.5000;Req.10000;NonUniform)](image)

Figure 4-13 Total memory hit ratios for 5000 documents in the nonuniform case

### 4.9.2 Performance Evaluation for Large Number of Documents Set

In the case of 8000 documents, WRG was marginally better than RWG. Both RWG and WRG had up to 39% better response times at 512 Mbytes than the LRU.

An increase in memory size from 64 Mbytes to 512 Mbytes meant an improvement in response times by a factor of 39% for both RWG and WRG. The same factor for LRU and LFU were 6% and 5.8%, respectively. RDG performed better than LRU and LFU. Figures 4-14 and 4-15 summarize the response times for 8000 documents.

In the nonuniform case, for 10,000 requests, WRG has better memory hits than RWG. For 100,000 and 1,000,000 requests, RWG has better memory hits than WRG. In the uniform case, for 1,000,000 requests, WRG has better memory hits than RWG. For less than 100,000 requests, RWG has better memory hits than WRG.
Figure 4-14 Total response times for 8000 documents in the nonuniform case

Figure 4-15 Total response times for 8000 documents in the uniform Case

Figures 4-16 and 4-17 show the memory hits of different algorithms in the case of 8000 documents. RWG and WRG achieved 8.8 times more memory hits compared with LRU when the memory size was 64 Mbytes. When the memory size increased from 64 Mbytes to 512 Mbytes RWG and WRG improved their memory hits by a factor of 6.4. The same factor for LRU and LFU were 8.1 and 7.9, respectively.
In the case of 10,000 documents WRG has up to 1% better response times than that of RWG for all request patterns. With 512 Mbytes, RWG and WRG had 34% better response times than LRU. This gain was 23% less than for the 5000 documents case. RWG and WRG were 34% better (in terms of response time) with 512 Mbytes than with 64 Mbytes. The same percentage for LRU and LFU were 5% and 8%, respectively. RDG has better response times than LRU and LFU. Figures 4-18 and 4-19 show the response times for the case of 10,000 documents.
Figure 4-18 Total response times for 10,000 documents in the uniform case

Figure 4-19 Total response times for 10,000 documents in the nonuniform case

For both nonuniform and uniform request patterns, RWG has better memory hits than WRG except in the case of 100,000 and uniform pattern. Figure 4-20 shows the memory hits for the case of 10,000 documents. RWG and WRG achieved 8.9 times more memory hits compared with LRU when the memory size was 64 Mbytes for all request patterns. An increase in memory size from 64 Mbytes to 512 Mbytes proved to be 6.5 times advantageous for RWG and WRG in terms of memory hits. The same factor for
LRU and LFU were 7.8 and 7.9, respectively. Results for 8000 and 10,000 documents also reveal the superiority of RWG and WRG algorithms.

![Memory Hit Ratio (Doc.10,000;Req.1000000;NonUniform)](image)

Figure 4-20 Total memory hit ratios for 10,000 documents in the nonuniform case

### 4.9.3 Performance Comparison in the Increasing Number of Documents

Now we show the effect of increasing the number of documents in our algorithms. We compared RWG, WRG, and RDG algorithms with LRU algorithm in the case of 5000 and 10,000 documents. In the nonuniform case, when the number of documents is doubled (for example, when the number changes from 5000 to 10,000), the total response times for RWG and WRG algorithms suffer by up to 37%, the memory size being 512 Mbytes for all request patterns. In the case of 64 Mbytes, 128 Mbytes, and 256 Mbytes, the performance deterioration was 5%, 10%, and 20%, respectively. The performance deterioration for the LRU algorithm was up to 4% with 512 Mbytes memory. In the case of 64 Mbytes and 128 Mbytes, there was no performance change in the LRU algorithm. In the case of 256 Mbytes, LRU lost 2% performance compared with the 5000 documents case.
In the uniform case, the performance deterioration for RWG and WRG was up to 38%, the memory size being 512 Mbytes. In the case of 64 Mbytes, 128 Mbytes, and 256 Mbytes, the performance deterioration was 6%, 11%, and 21%, respectively. This indicates that RWG and WRG algorithms perform slightly better in the nonuniform case than in the uniform case. The performance deterioration for LRU algorithm was up to 6% in the 512 Mbytes case for all request patterns. In the case of 64 Mbytes, 128 Mbytes, and 256 Mbytes, LRU lost 2%, 3%, and 4% performance compared with the 5000 documents case. Figure 4-21 summarizes these results.

Figure 4-21 Total response times comparison for 5000, 10,000 documents

In the nonuniform case, the performance deterioration in terms of memory hits for RWG and WRG algorithms was up to 43% when the memory size was 512 Mbytes. In the case of 64 Mbytes, 128 Mbytes, and 256 Mbytes, the performance deterioration was 49%, 48%, and 47%, respectively. The performance deterioration for LRU was up to 52% with 512 Mbytes memory. In the case of 64 Mbytes, 128 Mbytes, and 256 Mbytes, LRU lost 53%, 51%, and 51% performance compared with the 5000 documents case.
In the uniform case, the performance deterioration for RWG and WRG algorithm was up to 42% with 512 Mbytes memory. In the case of 64 Mbytes, 128 Mbytes, and 256 Mbytes, the performance loss was 49%, 48%, and 46%, respectively. The performance loss for LRU was up to 51% with 512 Mbytes memory. In the case of 64 Mbytes, 128 Mbytes, and 256 Mbytes, LRU lost 50%, 49%, and 51% performance compared with the 5000 documents case. Figure 4-22 shows these results.

Figure 4-22 Total memory hits comparison for 5000, 10,000 documents

The above results indicate that RWG and WRG algorithms have good total response times even when the number of documents is increased and the memory size is small. In the case of a large memory size also RWG and WRG algorithms perform better than the LRU and LFU algorithms.

4.9.4 Analysis of Simulation Results

Our experimental results clearly indicate that our algorithms have better response times and memory hits than the LRU and LFU algorithms. In the case of 4000
documents, RWG and WRG algorithms had up to 57% better response times than the LRU algorithm. A similar outcome has also been found in other cases.

In general, RWG and WRG had better performance gains for small memory sizes such as 64 Mbytes and 128 Mbytes than for large memory sizes such as 256 Mbytes and 512 Mbytes when compared with the LRU and LFU algorithms. In the case of small documents set and large number of web client requests, the RWG has marginally better response times than WRG. The WRG has marginally better response times than RWG in the case of large documents set and large number of web client’s requests. The RWG and WRG have good performance gains when increasing the memory size and the number of documents in the set. The performances of RWG and WRG algorithms with respect to response time as well as memory hits were not influenced by the web client’s request pattern (such as nonuniform and uniform) demonstrating that the RWG and WRG algorithms are suitable for any kind of Internet environment. RDG also performs better than the LRU and LFU algorithms.

With respect to memory hits, RWG has a slightly better performance gain in the nonuniform case than WRG. In the uniform case, WRG has a better performance.

We have compared the RWG and WRG algorithms with LRU and LFU algorithms under the conditions of increasing number of documents and memory size. If the number of document is doubled (in the nonuniform case), the response times of RWG and WRG suffer by up to 37%. In the uniform case, the performance loss was up to 38%. The memory hits of RWG and WRG algorithms suffer by up to 43% in the case of large memory size and large number of web client’s requests. When the memory size decreases, the performance loss could be up to 49% in the RWG and WRG algorithms.
5.1 Design of Scalable Web Server Architecture for Direct Control

To support the growing number of web client’s static and dynamic data requests, a scalable WWW architecture that consists of a group of loosely coupled WWW servers is needed. Currently, these servers typically employ large amounts of memory as cache space for supporting high request rates for static and dynamic data on such systems. It is essential that the available memory be used as efficiently as possible to reduce network traffic and to decrease response time for a web client’s request.

We propose a multi-node web server architecture that consists of a Central Server, multiple web servers, and the Request-distributing and Load-balancing Manager as shown in Figure 5-1 [75]. The web server architecture can be easily scaled up by installing a new web server into the high-speed network. The Central Server communicates with web servers through a high-speed network. A cluster of commodity web servers and the Request-distributing and Load-balancing Manager are also connected through a high-speed network.

The Central Server stores a complete set of static and dynamic web documents and performs all documents maintenance jobs. The Central Server also stores all information about static and dynamic documents using the hash table in the document table, which including their BF value, document identification number, version of the document,
duplication of the documents, size of document, etc. The details on how to construct, maintain, and use this information are discussed later.

Figure 5-1 Scalable web server architecture for direct control

Incoming web client requests go through the Request-distributing and Load-balancing Manager which distributes the request directly through a specific web server that caches the requested web content. The Request-distributing and Load-balancing Manager maintains information for cached web contents of all web servers. The Request-distributing and Load-balancing Manager also keeps information for duplicated web contents among web servers. In case the web client’s request is duplicated on several web servers the Request-distributing and Load-balancing Manager selects a web server randomly in duplicated web servers and sends the web client’s request to a selected web
server. The intent of using the Request-distributing and Load-balancing Manager is to balance a load among web servers and to decrease response time for a web client’s request. Better performance is achieved compared with not using the Request-distributing and Load-balancing Manager. The Request-distributing and Load-balancing Manager knows which web server caches the requested data and reduces retrieval time among web servers to find requested data in the memory of the cooperative system. Later we show the performance gain for response time and load balancing among web servers using the Request-distributing and Load-balancing Manager.

Based on this information, static and dynamic web contents that have a high BF value will be copied to and cached in the web servers. When web servers have cached a high BF value for both static and dynamic web contents they can serve most of web client’s requests immediately without accessing the Central Server. Therefore, the Central Server will not be the performance bottleneck for serving static and dynamic web contents in the system.

5.2 Directly Selected and Limited Look up Web Cache Algorithm

When a web client’s request enters through the network, the Request Manage Process (RMP) of the Request-distributing and Load-balancing Manager looks at the document manage table for distributing a web client’s request to a web server. Either the requested document is cached in only one web server the Request-distributing and Load-balancing Manager forwards the web client’s request to the web server, or the requested document is cached to several web servers the Request-distributing and Load-balancing Manager chooses any one of web servers randomly and forwards the web client’s request to the selected web server. When the requested document is not found in the document manage table a request for the requested document is sent to the Central Server.
Algorithm 4. **Directly Selected and Limited Look up (DSLL) Web Cache Algorithm**

**Step 1.** Directly select a web server by Request-distributing and Load-balancing Manager

- If the requested document is cached by only one web server,
  - Forwards the web client’s request to the web server,
- Else if the requested document is cached to several web servers,
  - Choose any one of web servers randomly and forward the web client’s request to the selected web server.
- If the requested document is not found in the document manage table,
  - Forward the web client’s request to the Central Server.

**Step 2.** If the requested document exists in selected web server’s memory,

- Service the web client request,
- Increase the value of number of request for that document,
- Update the BF value of that document, and quit go to Step 1;

Else repeat

- Look at the colleague’s web server memory to find the requested document until reaching the limited number
- If the requested document is found in other web server’s memory,
  - Service the web client’s request,
  - Increase the value of number of request for that document,
  - Update the BF value of that document, and quit go to Step 1;
- Else if request the document to the Central Server,
  - If receives the document from the Central Server,
    - Service the web client request,
    - Increase the value of number of request for that document,
    - Update the BF value of that document, and Compare the BF’s value of new request document with that of existing document in selected web server’s memory based on the size of new document,
    - If the BF value of new document is greater than existing document,
      - Replace the existing document with new document,
      - Send this information to the Central Server and the Request-distributing and Load-balancing Manager, and quit. go to Step 1.
    - Else stores the new document to the selected web server’s disk, and quit. go to Step 1.
  - Else sends a message such as document not found to the web client, and quit. go to Step 1.

---

Figure 5-2 DSLL web cache algorithm

The selected web server first checks that it keeps the requested document in memory or not. If the document is kept in the cache, the server sends the requested document to the web client and updates the BF value of that document. If the server does not keep the document in its cache, it looks for the document in a limited number of other
web servers’ memory to find the requested document. If the requested document is found in another servers’ memory, the server sends the document to the web client using the Request Manage Process and updates the BF value. After making these attempts and the document cannot be found the document is requested from the Central Server. After receiving the document from the Central Server, the server sends it to the web client. After that, the document is compared to the BF value of the requested document with that of the existing document in its cache content based on the size of the document. If the BF value of the document is greater than that of existing document, then it replaces the existing document with the requested document. Otherwise, it stores the document on the disk for preparing future access.

5.3 Direct Request Distribution and Load Balancing Cache Protocol

5.3.1 Central Server Mechanism and Operation in DLC Protocol

Communication among the Central Server, web servers, and the Request-distributing and Load-balancing Manager is done using a broadcast mechanism. To maintain cached static and dynamic data efficiently and consistently, a document table kept in the Central Server keeps the following information for maintaining all documents:

- **Document identification number.** A unique number assigned for distinguishing each document. Dynamic data sometimes has been synchronized with the database and this value is helpful to maintain documents. For example, a document identification number used to retrieve a specific data among web servers and is also used to update a cached data. This value is also kept in the Request-distributing and Load-balancing Manager for distributing web client’s requests.

- **Version of document.** Version is the number of updates for each document. This value helps maintain cache consistency between the Central Server and the web servers.

- **Number of request.** During the predefined period, the total number of requests for each document are recorded in this field. This value is used to calculate the BF value for each document.
• **Size of document.** Size of each document is kept in this field. This value is used to calculate the BF value for each document.

• **BF value of document.** Each document has this value. This value is calculated with number of requests and size of document value. This value uses an in-cache algorithm.

• **Duplicate of document.** This value uses to maintain the duplicated document among web servers. We allow duplication of the document for controlling load balance among web servers. This value helps to maintain the consistency of duplicated cached data among web servers.

• **Server ID of duplicated document.** This value also uses to maintain the duplicated document consistently among web servers.

Our caching protocol allows duplication of documents among web servers. There are several reasons why we choose this policy. Especially for dynamic data, the document generation time is much higher than that of static data. If several dynamic web contents are hot documents and cached on only one web server, then the cached web server suffers from a bottleneck of processing the dynamic data and service time to web clients is longer than that for cached several web servers. If hot documents are cached to several web servers, then we achieve a fast response time and good load balancing among web servers. If there are too many duplicated documents among web servers, then the overhead of redundant documents management is significant and efficiency of cache utilization of among web servers is decreased. Thus, we only allow the duplication of documents among web servers at most 30% of total number of web servers. For duplicated documents, a web client’s request is distributed randomly for achieving even distribution among web servers, where the duplicated documents are cached. We can guarantee the freshness of duplicated cached documents using a broadcast mechanism controlled by the Central Server.
After a predefined period, a new document table must be generated to reflect any change in documents. Starting with the highest BF value document, the Central Server will assign new documents to web servers using a round-robin pattern until each web server fills out its memory. Having done all these, the Central Server sends the cached documents information for each web server to the Request-distributing and Load-balancing Manager. The Request-distributing and Load-balancing Manager keeps cached documents information for each web server in the hash table. Until this step, there is no duplication of documents among web servers.

The Central Server broadcasts static and dynamic web documents to web servers and to the Request-distributing and Load-balancing Manager. There are several different kinds of broadcast messages. We allow duplication of documents among web servers and these duplicated documents must be up-to-date to avoid stale information being accessed by web clients. It is difficult for the web servers to distinguish these different types of broadcast messages without additional information, but since each broadcast message is encapsulated with in the header information, for example, type of broadcast and identification of web server. The most up-to-date cached documents can be accessed by the web client.

- **Type of broadcast.** This field is used to distinguish each type of broadcast message, which is sent to each web server. There are six different types of broadcast messages.

- **Identification of web server.** This value is used to determine which web server must receive each broadcast message. This value helps to maintain duplication of cached documents. This value is also used to distribute a web client’s requests to web servers by the Request-distributing and Load-balancing Manager.

There are six different types of broadcast messages among the Central Server, web servers, and the Request-distributing and Load-balancing Manager.
• **Static or dynamic document broadcast.** When a cache miss happens among web servers, the web server will forward the request to the Central Server. The Central Server will retrieve and send the document to all web servers. If any web server has already cached this new document it just ignores it.

• **Abnormal document broadcast.** When a web server is started from either a new installation or crash recovery, it will request static and dynamic document information from the Central Server for cache maintenance. For a newly installed web server, the Central Server sends some high BF value documents to new installed web server, which fills out its memory with these documents that are not cached among web servers during the initial documents assignment procedure. This document information is also sent to the Request-distributing and Load-balancing Manager. Following crash recovery, the Cache Maintenance Process of crashed web server requests the Central Server to send documents which were kept in its memory before the crash. When it receives documents from the Central Server, the Cache Maintenance Process will perform comparison between the received documents information and its previously stored one to maintain its local cache contents. During the down period, all currently outdated documents whose version field does not match or are already deleted from the Central Server are sent to the crashed web server by the Central Server.

• **New document broadcast.** In static or dynamic data, newly created data is almost always hot data. Thus, we perform a cache replacement with each new document. Before broadcasting, the Central Server chooses a number of web servers randomly, which represents 30% of the total number of web servers. For example, if there are nine web servers then we can choose at most three web servers for broadcasting a new document. After the Central Server chooses the web servers, the Central Server sets the identification of web server field with the selected web servers’ identification number, and then the Central Server broadcasts the new document to the selected web servers and the Request-distributing and Load-balancing Manager.

• **Document deletion broadcast.** When a document is deleted, the corresponding entry in the Central Server’s document table is removed. At the same time, the Central Server discovers in which web servers cache the deleted document in its memory using the duplicate of document field. When found, the Central Server immediately broadcasts a corresponding message to the web servers to delete the document in its cache. The Central Server also sends this information to the Request-distributing and Load-balancing Manager to delete the corresponding entry in its document manage table.

• **Document modification broadcast.** When a document is modified, the version of that document in the Central Server’s document table is updated. At the same time, the Central Server discovers which web servers cache the modified document in its memory using the duplicate of document field. After that, the Central Server immediately broadcasts a modified document with header information to the web
servers. The Central Server also sends this information to the Request-distributing and Load-balancing Manager to be updated the corresponding entry in the Request-distributing and Load-balancing Manager’s document manage table.

- **Document not found broadcast.** When a web server cannot find a requested document in memory or other web servers’ memory, then it requests it from the Central Server but the requested document may not exist in the Central Server. In that case, the Central Server broadcasts this message to web servers.

### 5.3.2 Web Server Mechanism and Operation in DLC Protocol

We design web servers which operate cooperatively with each other to serve a web client’s requests. This approach achieves greater memory utilization, higher cache hit rate, faster response time, and excellent load balancing among web servers. In the following paragraphs, we discuss the actions taken by the web server when different types of broadcast message are received.

- **Static or dynamic document broadcast.** When this type of broadcast message is received each web server’s Cache Maintenance Process checks the document based on BF value and size of the document and decides to cache it or not. If any web server decides to update its memory information, it replaces the existing document with the new one, and sends the update information to the Central Server and the Request-distributing and Load-balancing Manager as well as the deleted document identification number, inserted new document identification number, version of document, and the web server identification number using Cache Maintenance Process. Any web server already caching this document just ignores it.

- **Abnormal document broadcast.** When a web server is under normal operation, it ignores this type of broadcast message. Only the web server, which is undergoing the start up process takes this type of broadcast message in order to maintain the cache. A newly installed web server fills out its memory with some high BF value documents, which are sent by the Central Server. For crash recovery, the Cache Maintenance Process of crashed web server receives broadcast message from the Central Server. The Cache Maintenance Process will perform a comparison between the received document’s information and its previously stored one to maintain its local cache contents. The crashed web server receives information such as all currently outdated documents and already deleted documents in the Central Server.

- **New document broadcast.** For static or dynamic data newly created data is almost always hot data. Thus, the Cache Maintenance Process copies the new document to
its local cache if the local cache is not full. If it is full, it performs a cache replacement with new document.

- **Document deletion broadcast.** A web server checks its cached document to find the deleted document. At that time, document identification number field is used to find the deleted document. After it finds a deleted document in its cache, it deletes it.

- **Document modification broadcast.** When a web server receives this type of message the Cache Maintenance Process checks its cached document to find the modified document. At that time, document identification number and version of document fields are used to find the modified document. After finding a modified document in its cache, it deletes it and copies the modified document into its own cache.

- **Document not found broadcast.** When a web server receives this type of broadcast message, the Cache Maintenance Process returns a this document not found message to the Web Server Process through the CGI process. After the Web Server Process receives this message from the Cache Maintenance Process, it sends it to the web client.

### 5.3.3 RLM Mechanism and Operation in DLC Protocol

The Request-distributing and Load-balancing Manager (RLM) has two functional processes such as the Cache Control Process and the Request Manage Process. Figure 5-3 shows the functional components of the RLM. When the Request Manage Process receives a web client’s request, it will search the document manage table in its cache. It keeps all cached documents information for the web servers using a hash table in its cache. The document manage table has the following fields, for example, Document Identification Number, Identification of web server, Version of document, BF value of document, Duplicate of document.

When a requested document is found in the document manage table it checks the Identification of web server and the Duplicate of document fields. Either the requested document is cached on only one web server it forwards the web client’s request to the web server, or the requested document is cached several web servers it chooses any one
of web servers randomly and forwards the web client’s request to the selected web server. When the requested document is not found in the document manage table, then it sends the web client’s request to the Central Server. This approach achieves a fast response time for a web client’s request because of direct request forwarding. This approach also achieves excellent load balancing among web servers because of distributing the document requests evenly among web servers.

The Cache Control Process performs two main functions:

- It maintains the cached contents of the document manage table when he receives a message from a web server.
- It performs the corresponding actions when it receives a broadcast message from the Central Server or the web servers.


When the Request-distributing and Load-balancing Manager receives a broadcast message from the Central Server or a web server, the Cache Control Process buffers the message and examines the Identification of web server field and the Type of broadcast field from the header information. If the message is sent to it, it takes the message and performs the corresponding procedure, such as updating the document manage table. In the following paragraphs, we discuss the possible actions taken by the Request-
distributing and Load-balancing Manager when different types of broadcast message are received from the Central Server or a web server.

Figure 5-3 Functional components of the RLM

- **Static or dynamic document broadcast.** When this type of broadcast message is received the Cache Control Process updates its cache content with these information.

- **Abnormal document broadcast.** For a newly installed web server, new document information is received from the Central Server. For crash recovery, during the down period, information such as all currently outdated documents and already deleted documents in the Central Server is received from the Central Server. The Request-distributing and Load-balancing Manager updates its cache content with these information.

- **New document broadcast.** When a web server inserts a new document into its cache, it also sends the new document information to the Request-distributing and Load-balancing Manager which then updates its own cache content.

- **Document deletion broadcast.** When a document is deleted in the Central Server, the Request-distributing and Load-balancing Manager receives a message from the Central Server for deleting a corresponding entry in its document manage table.
• **Document modification broadcast.** When a document is modified in the Central Server the Central Server sends this information to the Request-distributing and Load-balancing Manager to be updated with the corresponding entry in the Request-distributing and Load-balancing Manager’s document management table.

### 5.4 Simulation Environment for DSLL and DLC

We simulated our new caching algorithm with DLC cache protocol to measure the performance gain compared with LRU and LFU algorithms. We simulated our cache algorithm for two kinds of different web client request patterns. First, we simulated our algorithm in uniform discrete random variation. This is an ideal web client request pattern. Every document is accessed evenly. Second, we simulated in non-uniform discrete random variation. This is strongly related to current web client’s request pattern. To support non-uniform discrete random variation of web client requests, we used the Algorithm Generate 1 [76] and modified it to fit our situation.

We chose nine web servers and the Request-distributing and Load-balancing Manager. To test for our caching protocol and cache algorithm, we made initial document information. Initial documents are composed of static and dynamic data with no more than 40% of the total documents containing dynamic data.

To verify the performance of our caching protocol and cache algorithm, we simulated several cases. Each case changed the value of several factors such as the portion of dynamic data in total documents and the memory size of each server. Each case was simulated over 100 times. The total number of documents stored in the Central Server was 10,000. The memory size of each web server varied from 64 Mbytes to 512 Mbytes. The number of web clients’ request’s is fixed to 1,000,000. The total memory size of the web servers is 10% - 20% of the total document size.
To analyze the performance gain of our algorithm, we used two values such as the response time for total web client’s requests and cache hit rate. We assumed each web server ran concurrently to service a web client’s request. Each web server’s memory contents were updated by the DLC cache protocol. The redundancy of web contents among web servers was also controlled by DLC cache protocol.

We classified each document to High BF document and Low BF document based on the BF values of each document. To divide the documents to High BF and Low BF categories, we accumulated the BF value of all of the documents and determine the portion which are hot documents based on an arbitrary percentage of hits. The portion is selected to any point, in our case, we assumed that 30% of the total documents belong to High BF documents. This portion is based on previous research [3,4,6,8,11,12,18]. The performance of web servers depended on 10 - 20% of the total documents as being hot. This threshold value is determined by dividing the BF value for each document in the web servers’ cache by the arbitrary percentage to determine which documents are considered hot. In graphs, DSLL10 means that there are 10% dynamic data in total documents.

5.5 Simulation Results

5.5.1 Performance Evaluation for Variation of Dynamic Data

In the case of different memory sizes, we compare the conventional LRU and LFU algorithms with the DSLL algorithm. The performance gain for total response time for DSLL is increased by up to 67% in 512 Mbytes memory size compared with LRU in nonuniform case. There is a 21% performance gain for 64 Mbytes memory. In 128 Mbytes and 256 Mbytes, DSLL earned 32% and 44% performance gain in total response time, respectively compared with LRU.
By increasing the memory size, performance with DSLL increased by up to 67% in total response time at 512 Mbytes compared with 64 Mbytes. However, conventional LRU increased by up to 11% for performance and LFU increased up to 14%. For both nonuniform and uniform cases the results were the same. This means that DSLL has a good performance regardless of the web client’s request pattern such as nonuniform or uniform. Figures 5-4 and 5-5 show the results of DSLL algorithm’s performance gain for total response time in case of different memory size.

![Figure 5-4 Total response time for nonuniform case](image)

Figures 5-6 and 5-7 show the results of DSLL algorithm’s performance gain for memory hit rate in case of different memory size. In nonuniform, DSLL had 9.3 times more memory-hit rate in 64 Mbytes compared with LRU algorithm. When increasing the memory size, the gain was decreased in 7.4 times more in 512 Mbytes compared with LRU. It is a reasonable result because the memory size is increased. DSLL has 6.6 times more memory hit according to increasing the memory size up to 512 Mbytes compared with 64 Mbytes in DSLL10 case. LRU earned 7.8 times more memory hit in 512 Mbytes.
compared with 64 Mbytes. This means that LRU is more sensitive in memory hits than DSLL.

In uniform case, DSLL has more memory hits compared with nonuniform. DSLL earned up to 9.8 times more memory hits in 64 Mbytes memory than the LRU algorithm. In the rest of cases such as 128 Mbytes, 256 Mbytes, and 512 Mbytes the same performance gain was seen as for nonuniform. With an increase in memory size, DSLL had 6.7 times more memory hits in 512 Mbytes compared with 64 Mbytes in the DSLL10 case. LRU has eight times more memory hits in 512 Mbytes than compared with 64 Mbytes.

From these experimental results, we know that DSLL has good performance gain in response time and memory hit rate regardless of web client’s request pattern such as nonuniform or uniform. DSLL achieved good performance gain in total response time of up to 67% for nonuniform and 512 Mbytes compared with the LRU algorithm. DSLL also experienced 9.8 times more memory hit in 64 Mbytes and uniform case compared
with LRU. It means that DSLL algorithm is suitable for the Internet environment as web caching algorithm.

![Graph showing Memory Hit Ratio for nonuniform case](image)

Figure 5-6 Total memory hit rate for nonuniform case

![Graph showing Memory Hit Ratio for uniform case](image)

Figure 5-7 Total memory hit rate for uniform case

### 5.5.2 Performance Comparison in case of Using the RLM

Now we show the effect of using the Request-distributing and Load-balancing Manager in our algorithm. We compared DSLL algorithm using the Request-distributing and Load-balancing Manager with a non-DSLL algorithm, which is not using the Request-
distributing and Load-balancing Manager in the case of differencing memory sizes. In the nonuniform case, the performance of total response time for non-DSLL algorithm was decreased by up to 19% with 512 Mbytes memory compared with the DSLL10 case. In the case of 64 Mbytes, 128 Mbytes, and 256 Mbytes, the performance lost of total response time is 3%, 6%, and 11%, respectively, compared with DSLL10 case.

In the uniform case, the performance of total response time for non-DSLL algorithm was decreased by up to 20% in 512 Mbytes memory compared with DSLL10 case. In the case of 64 Mbytes, 128 Mbytes, and 256 Mbytes, the performance lost a total response time is 4%, 7%, and 12%, respectively, compared with the DSLL10 case. It means that the DSLL algorithm has a little better performance gain in response time when the web client’s requests are distributed in nonuniform pattern. Figure 5-8 shows the results of non-DSLL and DSLL algorithms’ performance difference in total response time at nonuniform case.

Now we compare the memory-hit rate between non-DSLL and DSLL algorithms. In the nonuniform case, the performance of memory-hit rate for non-DSLL algorithm was decreased by up to 22% in 512 Mbytes memory at DSLL10 case. In the case of 64 Mbytes, 128 Mbytes, and 256 Mbytes, the performance lost was 36%, 31%, and 26%, respectively, compared with DSLL10 case. For the uniform case, the performance of memory-hit rate for non-DSLL algorithm was decreased by up to 21% for 512 Mbytes memory at DSLL10 case. In case for 64 Mbytes, 128 Mbytes, and 256 Mbytes, the performance lost was 36%, 32%, and 27%, respectively, compared with DSLL10 case. Figure 5-9 shows the results for non-DSLL and DSLL algorithms’ performance difference in memory-hit rate for the nonuniform case.
From the comparison of non-DSLL and DSLL algorithms, we found that the performance lost for total response times for non-DSLL is up to 19% in 512 Mbytes memory of nonuniform case compared with DSLL10 case. In the uniform case, the performance lost increased by 1% more, but for small memory sizes, such as 64 Mbytes and 128 Mbytes, the performance lost of total response time for non-DSLL is reduced 4% and 7% meaning that the Request-distributing and Load-balancing Manager has a positive role in DSLL algorithm for reducing the total response time regardless of memory size. It is an important result to note that the Request-distributing and Load-balancing Manager reduces the response time for web client’s requests.

### 5.5.3 Analysis of Simulation Results

We simulated our DLC cache protocol and DSLL cache algorithm under several cases and we know that our new DLC cache protocol and DSLL cache algorithm each have good performance gain with regard to response time and memory hit rate compared with LRU and LFU algorithms. In the case of 512 Mbytes memory, the DSLL algorithm
showed up to a 67% performance gain in total response time compared with the LRU algorithm under the nonuniform case. The DSLL algorithm achieved up to 9.3 times more memory-hits than LRU algorithm at 64 Mbytes in the nonuniform case.

We compared the non-DSLL algorithm with the DSLL algorithm under conditions of increasing memory size. The performance of total response time in non-DSLL algorithm was decreased by up to 19% at 512 Mbytes and DSLL10 case in nonuniform case. For the uniform case performance lost was increased by 1% more. In the memory-hit case, the performance of non-DSLL algorithm is reduced up to 22% in case of 512 Mbytes memory at DSLL10 case and nonuniform. According to decreasing the memory size to 64 Mbytes, the performance lost in memory-hit was increased by up to 36% in non-DSLL algorithm in case of uniform. From these results, we know that the Request-distributing and Load-balancing Manager has an important role in the DSLL algorithm for reducing response time for web client’s requests and increasing memory hit.
6.1 Design of Scalable Web Server Architecture for Parallel Control

We propose a multi-node web server architecture that consists of a Central Server, multiple web servers, and the Distributor as similar in Figure 4-1 [77,78]. Each web server contains a Cache Maintenance Process and a Web Server Process and the functionality is the same as in Section 4.1.1. Each web server contains limited number of neighbor web servers’ cache content information in its memory. This approach helps to reduce the response time, load of the Central Server, and the network bandwidth for serving the web client’s request. It also increases the cache-hit rate for cached data among web servers. The detailed operations of web server using the RLC caching protocol and RSLL cache algorithm are discussed in later sections. Figure 6-1 shows the detail functionality of web server.

Incoming web client requests go through the Distributor which distributes the request randomly through among web servers. The intent of using the Distributor is to balance a load among web servers and to decrease response time for a web client’s request. On our previous work [75] shows the functionality of the Distributor more detail. It has good performance gain for response time and memory hits in large number of web client requests such as over than 1,000,000 per day.

6.2 Randomly Selected and Limited Look at Neighbor Web Cache Algorithm

When a web client’s request enters through the network, the Distributor selects a
Figure 6-1 Functional components of web server in parallel cache content management

web server randomly for distributing a web client’s request. The selected web server first checks that it keeps the requested document in memory or not using hash table. If the document is kept in the cache, the server sends the requested document to the web client and updates the BF value of that document. If the server does not keep the document in its cache, it looks for the document in the nearest neighbor web servers’ cached information which is stored in its memory.

If the requested document is found in the nearest neighbor server’s memory, it is forwarded to the requesting web server. The applicable web server serves the request to the web client and updates the information for the requested document. If the server does not find the requested document in its cached information, which includes its cache contents and its nearest web server’s cache contents, it looks at the limited number of its next nearest neighbor web servers’ cached information to find the requested document. If the server finds the requested document in its next nearest neighbor web server’s
memory, the server forwards the request to the applicable web server and the selected applicable web server serves the web client’s request and updates some values. If the server finds the request in its next nearest neighbor web server’s neighbor cache, the server forwards the request to applicable web server and the selected applicable web server serves the web client’s request and updates the BF value.

If after making these attempts the document cannot be found, the requested document is requested from the Central Server. After receiving the document from the Central Server, the server sends it to the web client. Then the document is compared to the BF value of the requested document with that of the existing document in its cache content based on the size of the requested document. If the BF value of the requested document is greater than that of existing document, then it replaces the existing document with the requested document and sends this information to the Central Server and applicable neighbor web servers. Otherwise, the document on the disk is prepared for future access.

6.3 Random Request Distribution and Load Balancing Cache Protocol

6.3.1 Central Server Mechanism and Operation in RLC Protocol

There are six different types of broadcast messages between the Central Server and web servers.

- **Static or dynamic document broadcast.** When a cache miss happens among web servers, the web server will forward the request to the Central Server. The Central Server will retrieve and send the document to selected web servers, which is chosen by the Central Server randomly. It is not over 30% of total number of web servers.

- **Abnormal document broadcast.** When a web server is started from either a new installation or crash recovery, it will request static or dynamic document information from the Central Server for cache maintenance. For a newly installed web server, the Central Server sends some high BF value documents to new installed web server. The web server fills out its memory with these documents that
Algorithm 5. Randomly Selected and Limited Look at neighbor (RSLL) Web Cache Algorithm

Step 1. Randomly select a web server by Distributor

Step 2. If the requested document is found in the selected web server’s cached information,
    If the requested document exists in selected web server’s memory
        Service the web client request,
        Increase the value of number of request for that document,
        Update the BF value of that document, and quit go to Step 1;
    Else if the requested document exists in selected web server’s the nearest neighbor’s web server’s memory
        Forward the request to applicable web server
        The applicable web server serves the web client request,
        Increase the value of number of request for that document,
        Update the BF value of that document, and quit go to Step 1;
    Else repeat
        Look at the neighbor web servers’ cached information to find the request document
        until reaching the limited number of nearest neighbors
        If found in the next nearest neighbor web servers’ cached information,
            If the requested document exists in the next nearest neighbor web servers’ memory
                Forward the request to applicable web server
                The applicable web server serves the web client request,
                Increase the value of number of request for that document,
                Update the BF value of that document, and quit go to Step 1;
            Else if the requested document exists in the next nearest neighbor web server’s neighbor’s cache
                Forward the request to applicable web server
                The applicable web server serves the web client request,
                Increase the value of number of request for that document,
                Update the BF value of that document, and quit go to Step 1;
            Else if requests the document to the Central Server,
                If receives the document from the Central Server,
                    Service the web client request,
                    Increase the value of number of request for that document,
                    Update the BF value of that document,
                    Compare the BF’s value of the requested document with that of existing document in selected web server’s memory based on the size of new document,
                    If the BF value of the requested document is greater than the existing document,
                        Replace the existing document with the requested document,
                        Send this information to the Central Server and applicable neighbor web servers, and quit. go to Step 1.
                    Else stores the requested document to the selected web server’s disk,
                        and quit. go to Step 1.
                Else sends a message “document not found” to the web client,
                        and quit. go to Step 1.

Figure 6-2 RSLL web cache algorithm
are not cached among web servers during the initial documents assignment procedure. Following crash recovery, the Cache Maintenance Process of crashed web server requests the Central Server to send documents, which were kept in its memory before the crash. During the down period, all currently outdated documents whose version field does not match or are already deleted from the Central Server are sent to the crashed web server by the Central Server.

- **New document broadcast.** In static or dynamic data, newly created data is almost always hot data. Thus, we perform a cache replacement with each new document. Before broadcasting, the Central Server chooses a number of web servers randomly, which represents 30% of the total number of web servers. After the Central Server chooses the web servers, the Central Server sets the identification of web server field with the selected web servers’ identification number, and then the Central Server broadcasts the new document to the selected web servers.

- **Document deletion broadcast.** When a document is deleted, the corresponding entry in the Central Server’s document table is removed. At the same time, the Central Server discovers in which web servers cache the deleted document in memory using the duplicate of document field and the Server ID of duplicated document. When found, the Central Server immediately broadcasts a corresponding message to web servers for indicating deleted document.

- **Document modification broadcast.** When a document is modified, the version of that document in the Central Server’s document table is updated. At the same time, the Central Server discovers which web servers cache the modified document in its memory using the duplicate of document field and the Server ID of duplicated document. After that, the Central Server immediately broadcasts a modified document with header information to the web servers.

- **Document not found broadcast.** When a web server cannot find a requested document in memory or other web servers’ memory, then it requests it from the Central Server but the requested document may not exist in the Central Server. In that case, the Central Server broadcasts this message to web servers.

### 6.3.2 Web Server Mechanism and Operation in RLC Protocol

We design web servers which operate cooperatively with each other to serve a web client requests. Each web server has two functional processes called the Cache Maintenance Process and the Web Server Process. Each web server keeps its cached contents information such as document identification number, version of document, BF value of document, duplicate of document, etc. in its memory. It also keeps the Selected
Neighbor Web Servers’ cached contents information in its memory. The Selected Neighbor Web Servers are the nearest left and right web servers. Thus, each web server keeps the content of three web servers’ cached contents information in its memory using a hash table. This approach [79] achieves faster retrieval of the requested data, consumes less memory, and maintains the cached information easily. When each web server keeps so many other web servers’ cached information in its memory, it consumes more memory to keep this information and it becomes difficult to cache real data because of fewer memory spaces. It is also difficult to maintain the cached information because of increased cached neighbor web servers’ information. This approach also increases the communication overhead among web servers because an increased number of broadcast messages is requested for making cache requests.

When the Web Server Process receives a web client’s request, it will search for the document in its own cached information first. If a cache hit occurs, the document is sent to the web client. When the document is not found in its cached information, it then looks at the limited number of Neighbor Web Servers’ memory to find the requested document. The Neighbor Web Servers are the next nearest left and right web servers. When a cache hit occurs, the document is sent to the web client. The limited number of Neighbor Web Servers is at most two. This approach guarantees a limited response time to a web client requesting a certain document. If we allow searching on all web servers to find the document, then not only the cache-hit rate but also the response time will be increased for serving the web client’s request. This incurs a worse performance because it only considers the cache-hit rate. When a cache miss happens, the Web Server Process of first selected web server starts a CGI process with the help of the Cache Maintenance Process.
A CGI process then requests the document from the Central Server and serves the request to the web client. In both cases, the Web Server Process records the result of the web client request in its access log file. Figure 6-3 explains the mechanism of the Web Server Process in web server.

The total number of web servers (N) depends on each web servers’ configuration. For example, we choose N=9. When a web client request arrives at web server 0 and a
cache miss happens, then it looks at its Neighbor Web Servers web server 2 and web
server 7 to find the requested document. With this procedure, web server 0 looks at only
7 of the 9 web servers’ cached information such as web server 0, 1, 2, 3, 6, 7, 8. It covers
78% of the total web servers’ cached information. This approach achieves an excellent
cache hit rate compared to fully storing all of the other web servers’ cached information
in each web server. This approach also reduces the Central Server’s load, which occurs
only in the case of a cache miss among web servers. This approach has several
advantages compared with fully storing all other web servers’ cached information in each
web server such as less memory consumption, easy maintenance, and reduction of
communication overhead among web servers.

There are six different types of broadcast messages between the Central Server and
web servers.

• **Static or dynamic document broadcast.** If any web server decides to update its
memory information, it replaces the existing document with the new one and sends
the update information through Cache Maintenance Process to the Central Server
and Selected Neighbor Web Servers, which is decided by RSLL cache algorithm.
The sending information includes deleted document identification number, inserted
new document identification number, version of document, and the web server
identification number. Any web server already caching this document just ignores
it.

• **Abnormal document broadcast.** When a web server is under normal operation, it
ignores this type of broadcast message. Only the web server, which is undergoing
the start up process takes this type of broadcast message in order to maintain the
cache. A newly installed web server fills out its memory with some high BF value
documents, which are sent by the Central Server. For crash recovery, the Cache
Maintenance Process of crashed web server receives broadcast message from the
Central Server. The Cache Maintenance Process will perform a comparison
between the received document’s information and its previously stored one to
maintain its local cache contents. The crashed web server receives information such
as all currently outdated documents, and already deleted documents in the Central
Server. Having done all these, the web server sends its cached information to
specific Neighbor Web Servers decided by RSLL cache algorithm.
- **New document broadcast.** For static or dynamic data newly created data is almost always hot data. Thus, the Cache Maintenance Process copies the new document to its local cache if the local cache is not full. If it is full, it performs a cache replacement with new document. After that, the web server sends its cached information to Selected Neighbor Web Servers.

- **Document deletion broadcast.** When a web server receives this type of message from the Central Server it checks its cached document to find the deleted document. At that time, document identification number field is used to find the deleted document. After it finds a deleted document in its cache, it deletes it. This information is sent to the Selected Neighbor Web Servers.

- **Document modification broadcast.** When a web server receives this type of message, the Cache Maintenance Process checks its cached document to find the modified document. At that time, document identification number and version of document fields are used to find the modified document. After finding a modified document in its cache, it deletes it and copies the modified document into its own cache. The web server sends this update information to the Selected Neighbor Web Servers to be updated the corresponding entry in the Selected Neighbor Web Servers’ cache information.

- **Document not found broadcast.** When a web server receives this type of broadcast message, the Cache Maintenance Process returns a this document not found message to the Web Server Process through the CGI process. After the Web Server Process receives this message from the Cache Maintenance Process, it sends it to the web client.

### 6.4 Simulation Environment for RSLL and RLC

The simulation environment is almost the same as in Section 5.4. To calculate the response time accurately from a web server or the Central Server to a web client for each request, we selected different weighting factor for each different case to calculate total processing time for applying latency. We apply different type of processing time required for serving web client request. If the request is serviced from the first selected web server’s cache, we gave 0.3 weighting factor for each document. If the request is serviced from the first selected web server’s the nearest web server’s cache, we gave 0.7 weighting factor for each document. If the request is serviced from the first selected web server’s cache, we gave 0.7 weighting factor for each document.
server’s next nearest web server’s cache, we gave 1.3 weighting factor. If the request is serviced from the first selected web server’s next nearest web server’s neighbor web server’s cache, we gave 1.8 weighting factor. If the request is serviced from the Central Server, we gave 2.3 weighting factor for each document. To analyze the performance gain of our caching protocol and algorithm, we used two values such as the response time for total web client requests and cache hit rate. We assumed each web server ran concurrently and cooperatively to service a web client’s request. Each web server’s memory contents were updated by the RLC cache protocol. The redundancy of web contents among web servers was also controlled by RLC cache protocol. In graphs, RSLL10 means that there are 10% dynamic data in total documents.

6.5 Simulation Results

6.5.1 Performance Results of Variation of Dynamic Data

In the case of different memory sizes, we compare the conventional LRU and LFU algorithms with the RSLL algorithm. The performance gain for total response time for RSLL is increased by up to 63% in 512 Mbytes memory size compared with LRU in nonuniform case. There is a 20% performance gain for 64 Mbytes memory. In 128 Mbytes and 256 Mbytes, RSLL earned 31% and 42% performance gain in total response time, respectively compared with LRU.

By increasing the memory size, performance with RSLL increased by up to 63% in total response time at 512 Mbytes compared with 64 Mbytes. However, conventional LRU increased by up to 9% for performance and LFU increased up to 12%. For both nonuniform and uniform cases, the results were the same. This means that RSLL has a good performance regardless of the web client’s request pattern such as nonuniform or
uniform. Figures 6-4 and 6-5 show the result of RSLL algorithm’s performance gain for total response time in case of different memory size.

![Response Time (NonUniform)](image1)

**Figure 6-4** Total response time for nonuniform case

![Response Time (Uniform)](image2)

**Figure 6-5** Total response time for uniform case

Figures 6-6 and 6-7 show the results of RSLL algorithm’s performance gain for memory hit rate in case of different memory size. In nonuniform, RSLL had 9.1 times more memory hit rate in 64 Mbytes compared with LRU algorithm. When increasing the
memory size, the gain was decreased in 6.9 times more in 512 Mbytes compared with LRU. It is a reasonable result because the memory size is increased. RSLL has 6.3 times more memory hit according to increasing the memory size up to 512 Mbytes compared with 64 Mbytes in RSLL10 case. LRU earned 7.8 times more memory hit in 512 Mbytes compared with 64 Mbytes. This means that LRU is more sensitive in memory hits than RSLL.

In uniform case, RSLL has more memory hits compared with nonuniform. RSLL earned up to 9.3 times more memory hits in 64 Mbytes memory than the LRU algorithm. This result is the same in every different portion of dynamic data cases. In the rest of cases such as 128 Mbytes, 256 Mbytes, and 512 Mbytes the same performance gain was seen as for nonuniform. With an increase in memory size, RSLL had 6.5 times more memory hits in 512 Mbytes compared with 64 Mbytes in the RSLL10 case. LRU has eight times more memory hits in 512 Mbytes than compared with 64 Mbytes. It means that RSLL has more memory hit in small memory size and uniform web client requests.
From these experimental results, we know that RSLL has good performance gain in response time and memory hit rate regardless of web client’s request pattern such as nonuniform or uniform. RSLL achieved good performance gain in total response time of up to 63% for nonuniform and 512 Mbytes compared with the LRU algorithm. RSLL also experienced 9.3 times more memory hit in 64 Mbytes and uniform case compared with LRU. It means that RSLL algorithm is suitable for the Internet environment as web caching algorithm regardless of web client’s request pattern.

6.5.2 Performance Comparison between RSLL and DSLL

Now we show the performance Comparison between RSLL and DSLL (Directly Selected and Limited Look up) cache algorithms. The function of DSLL is that DSLL has the Request Distributing and Load Balancing Manager and this manager distributes every web client’s requests to a web server directly, which caches the requested document in its memory. If directly selected web server has a cache miss in its memory, then the selected web server looks at the limited number of colleague web servers’ memory to find the
request and serve it. There are more detail information in Section 5.2. We compared RSLL algorithm which used neighbor web servers’ cached information mechanism with the DSLL algorithm, which used the direct request distribution manager and looked at the limited number of colleague web servers’ memory, in the case of different memory sizes.

In the nonuniform case, the performance of total response time for RSLL algorithm was decreased by up to 5% with 64 Mbytes memory compared with the DSLL algorithm. In the cases of 128 Mbytes, 256 Mbytes, and 512 Mbytes, the performance lost of total response time is 4.5%, 3%, and 2.2%, respectively, compared with DSLL10 case. These results are the same in different portions of dynamic data. In the uniform case, the performance of total response time for RSLL algorithm was decreased by up to 5.7% in 64 Mbytes memory compared with DSLL10 case. In the cases of 128 Mbytes, 256 Mbytes, and 512 Mbytes, the performance lost a total response time is 4.8%, 3.4%, and 2.3%, respectively, compared with the DSLL10 case. It means that the RSLL algorithm has a little better performance gain in response time when the web client requests are distributed in nonuniform pattern. Figure 6-8 shows the results of RSLL and DSLL algorithms’ performance difference in total response time at nonuniform case.

We now compare the memory-hit rate between RSLL and DSLL algorithms. In the nonuniform case, the performance of memory-hit rate for RSLL algorithm was decreased by up to 5.6% in 64 Mbytes memory in the DSLL10 case. In the cases of 128 Mbytes, 256 Mbytes, and 512 Mbytes, performance loss was 4.8%, 3.2%, and 2.3%, respectively, compared with DSLL10 case. These results are the same in different portions of dynamic data. For the uniform case, the performance of memory-hit rate for RSLL algorithm was decreased by up to 5.7% for 64 Mbytes memory in the DSLL10 case. In the cases of 128
Mbytes, 256 Mbytes, and 512 Mbytes, the performance loss was 4.9%, 3.3%, and 2.5%, respectively, compared with DSLL10 case. Figure 6-9 shows the results for RSLL and DSLL algorithms’ performance difference in memory-hit rate for the nonuniform case.

![Response Time Comparison RSLL vs. DSLL](image)

Figure 6-8 Total response time comparison for nonuniform case

From the comparison of RSLL and DSLL algorithms, we found that the performance lost for total response times and total memory hit rate for RSLL is up to 5.7% in 64 Mbytes memory of uniform case compared with DSLL10 case. This means that the DSLL algorithm is a little better than RSLL in both of nonuniform and uniform web client’s requests distribution regardless of memory size.

Although the DSLL has a greater performance gain for response time and increased memory hit rate by up to 5.7% compared with RSLL, the RSLL has several advantages compared with the DSLL and these benefits cover the performance gap. First, the RSLL approach is more reliable than the DSLL approach. When the distribution manager in DSLL approach crashes the DSLL method cannot serve the web client request anymore.
However in RSLL, if any one of web servers are crashed, the RSLL method still serves the web client request. Second, the RSLL approach can expand its system power easily by adding a new web server into the network compared with the DSLL. Third, we can use the distribution manager in the DSLL approach as a web server when we choose to use the RSLL approach. It reduces the configuration cost and increases the processing power. Finally, the RSLL approach reduces the communication overhead among web servers since it does not need to communicate between web servers and the distribution manager which are needed in the DSLL approach. The RSLL approach causes a reduction in network bandwidth usage in the network.

![Memory Hit Ratio Comparison (Req.1000000;NonUniform)](image)

Figure 6-9 Total memory hit comparison for nonuniform case

### 6.5.3 Analysis of Simulation Results

We simulated our RLC cache protocol and RSLL cache algorithm under several cases and we know that our new RLC cache protocol and RSLL cache algorithm each have better performance gain with regard to response time and memory hit rate compared
with LRU and LFU algorithms. In the case of 512 Mbytes memory, the RSLL algorithm showed up to a 63% performance gain in total response time compared with the LRU algorithm under the nonuniform case. The RSLL algorithm achieved up to 9.3 times more memory-hits than LRU algorithm at 64 Mbytes in the uniform case.

We compared the RSLL algorithm with the DSLL algorithm under conditions of increasing memory size. The performance of total response time in RSLL algorithm was decreased by up to 5.7% at 64 Mbytes and DSLL10 case in uniform case. For the nonuniform case, performance loss was slightly reduced. In the memory-hit case, the performance of the RSLL algorithm was reduced up to 5.7% in case of 64 Mbytes memory in the DSLL10 and uniform cases. Increasing the memory size to 512 Mbytes caused the performance gap in memory-hit to be reduced by up to 2.3% in the DSLL10 and nonuniform cases. From these results, we know that the DSLL approach has a little better performance gain in total response time and total memory hit rate than those of the RSLL in nonuniform and uniform cases. However, the RSLL has several benefits compared with the DSLL and it has enough advantages to cover the performance gap as described in the previous step.
CHAPTER 7
FAST AND SCALABLE IP LOOKUP AND UPDATE SCHEME FOR IP ROUTING

7.1 IP Lookup and Design Concepts

The introduction of the Gigabit Ethernet at the LANs and the random traffic patterns generated by the users of the Internet have increased the demand for higher bandwidth in the Internet backbone. Speeding up the packet forwarding in the Internet backbone requires high-speed communication links and faster routers. The bandwidths of transmission links keep improving and provision of gigabit fiber links is readily available. This has shortened the time of processing and forwarding packets. Consequently, the key to improve the speed of the Internet lies in fast routers [20]. A multigigabit router must have enough internal bandwidth to switch packets between its interfaces at multigigabit rates and enough packet processing power to forward several Millions of Packets Per Second (MPPS) [21]. As a result, the major remaining bottleneck in the Internet router design is the slow, software-based IP lookup procedures.

An IP lookup scheme is the most fundamental operation in any IP routing product. A packet is received with a specific IP Destination Address (DA), a unique 32-bit field in current IP version 4 implementations. IP routes have been identified by a <routing prefix, prefix length> pair. The router must search forwarding tables using the DA as its key and determine which entry in the table represents the best route for the packet to its destination. The search is complicated by the fact that entries in the table have variable lengths, and also that many entries may represent valid routes to the same destinations.
The search may be time consuming, especially in a backbone router with a large number of table entries. This was done to conserve the memory space used by the forwarding tables. Unlike a simple search or hash that seeks to find an exact match within a table, the IP lookup algorithm must select the most specific route from a number of entries for the given DA. For example, a forwarding database may have the prefixes P1 = 0101, P2 = 0101101 and P3 = 010110101011. An address whose first 12 bits are 010101101011 has longest matching prefix P1. On the other hand, an address whose first 12 bits are 010110101101 has longest matching prefix P3. To meet the demands for high-speed routers, IP lookup schemes that are efficient, flexible, and cost-effective are required.

Numerous algorithms have been proposed for this problem. Both dynamic and static versions have been considered. In the dynamic version, the entries in the table $T$ could change with time. In other words, inserts and deletes in the table are permitted. In this chapter we focus our attention on the static version [80].

All existing schemes have problems of either performance, scalability, generality, or cost. We now describe a scheme that has good performance in lookup, excellent scalability, and does not require protocol change. Our scheme also allows a cheap, fast software implementation.

Our contribution is to group large forwarding table entries into small and even interval forwarding table entries and to provide fast and scalable search times for given any IP address. In standard binary search [42], each prefix maps to an address range. A set of $n$ prefixes partition the address line into at most $2n$ intervals. We show how to modify binary search to adapt into longest matching prefix in IP lookup using grouping
scheme. We also show how to find the longest matching prefix faster and more accurately in forwarding table for given any IP address.

7.2 Gigabit IP Switching Router Architecture

The architecture of the Gigabit IP switching router is schematically shown in Figure 7-1, where a number of link interfaces, a CPU module, and a forwarding engine are interconnected with a switching fabric. The forwarding engine employs a forwarding database, a local version of the routing table, downloaded from the CPU module to make the routing decision. The CPU module executes the routing protocols, such as Routing Information Protocol (RIP) and Open Shortest Path First (OSPF), and needs a dynamic routing table for fast updates and fast generation of forwarding databases. On the other hand, it is better to optimize the forwarding database to furnish fast lookups. This also implies that the forwarding databases need not be dynamic.

![Figure 7-1 Architecture of gigabit IP switching router](image-url)
The architecture of the Forwarding Engine (FE) is shown in Figure 7-2. For an incoming IP packet, the route lookup process (based on the destination IP address of the packet), header verification, and header update are initiated simultaneously. If the IP header is not correct, the packet is dropped and the lookup is terminated. Otherwise, the header is updated into the packet header (TTL decrement and checksum update) and the route lookup module will provide the next hop (port number) where the packet should be forwarded. The MAC address substitution module then substitutes the source MAC
address and the destination MAC address of the packet before it is forwarded into the interface port. The source MAC address is replaced by that of the output interface, and the destination MAC address is replaced by that of the immediate next hop (a router or the destination host). The bottleneck of the forwarding engine is the route lookup and we will focus on the design of fast and scalable lookup scheme.

7.3 Reducing the Longest Matching Prefix Search to Binary Search

Our algorithm is based on the algorithm of Lampson, Srinivasan, and Varghese [42]. They reduce the problem of longest prefix matching to binary search. Every prefix is thought of as a range \([b, e]\) where \(b\) is the beginning and \(e\) is the end of the range. Let \(W\) denote the maximum length (in bits) of any IP address. Consider an example where \(W=6\) and \(011^*\) is a prefix. This prefix will be considered as the range: \([24, 31]\). It is assumed that \(P1=^*\) is one of the prefixes in the router table. This assumption does not alter the functioning or the performance of the algorithm in any significant way (see e.g., [81]).

The algorithm of [42] constructs a table of distinct range end-points for the prefixes in the router table. For instance let the prefixes in the router table be: \(^*, 01^*, 011^*, 1010^*, 110^*, 110011,\) and \(00110^*\). The ranges represented by these prefixes are: \([0,63], [16, 31], [24, 31], [40, 43], [48, 55], [51, 51]\) and \([12,13]\), respectively. The distinct end points are sorted to create a search table. Table 7-1 shows the results.

Given any IP address \(A\) we can perform a binary search in the sorted column of end points to figure out its longest matching prefix. For instance if \(A\) is \(001001\) we search for 9 in the end point column and realize that 9 falls between 0 and 12. Also, 9 is greater than 0 and hence the \(>\) column corresponding to 0 gives us \(P1\) as the longest matching prefix.

If \(A\) is \(011000\) we perform a search for 24 and realize that there is a match. Thus the =
column corresponding to 24 tells us that P3 is the longest matching prefix. As a final example, if A is 100011, a search for 35 results in the realization that 35 falls in between 31 and 40 and 35 is greater then 31. Thus the > column corresponding to 31 yields P1 as the longest matching prefix.

Table 7-1 Distinct end points search table with > and = values

<table>
<thead>
<tr>
<th>End Point</th>
<th>&gt;</th>
<th>=</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P1</td>
<td>P1</td>
</tr>
<tr>
<td>12</td>
<td>P7</td>
<td>P7</td>
</tr>
<tr>
<td>13</td>
<td>P1</td>
<td>P7</td>
</tr>
<tr>
<td>16</td>
<td>P2</td>
<td>P2</td>
</tr>
<tr>
<td>24</td>
<td>P3</td>
<td>P3</td>
</tr>
<tr>
<td>31</td>
<td>P1</td>
<td>P3</td>
</tr>
<tr>
<td>40</td>
<td>P4</td>
<td>P4</td>
</tr>
<tr>
<td>43</td>
<td>P1</td>
<td>P4</td>
</tr>
<tr>
<td>48</td>
<td>P5</td>
<td>P5</td>
</tr>
<tr>
<td>51</td>
<td>P5</td>
<td>P6</td>
</tr>
<tr>
<td>55</td>
<td>P1</td>
<td>P5</td>
</tr>
<tr>
<td>63</td>
<td>-</td>
<td>P1</td>
</tr>
</tbody>
</table>

In summary, the problem of longest matching prefix reduces to binary search. Thus this algorithm takes O(log n) time for each search. Our new algorithm is for performing binary search in an efficient manner. In particular, it applies to the longest matching prefix problem as well.

7.4 Generic Algorithm

Let K=K_1,K_2,\ldots,K_n be a given sequence of numbers on which we desire to perform binary searches. Without loss of generality let these be binary numbers of
maximum length Q. The idea is to use a very simple hashing scheme as follows. Let q (≤ Q) be an integer. We partition the range [0, 2^Q-1] into 2^q equal intervals. Keys that fall in each interval are kept as a separate group. Given a key x to search for, we look at the first q bits of x to determine which group it belongs to and search for x in that group. This scheme reduce the size of the search entries into small group entries. Thus we can get faster search time compared with the original size of the search entries.

There are many ways to keep the groups themselves. They can be kept as balanced tree schemes such as 2-3 trees, red-black tree, etc. [82]. Or the groups can be kept as sorted arrays.

If the list K is kept as a single sorted array, each search will take Ω(log n) time [82]. In the new scheme, each search only takes an expected time of O(1+log (n/2^q)). For instance, if q is chosen to be log n, then the expected search time is only O(1). The memory requirement is still O(n).

### 7.5 Case of IP Routing

In this section we specialize the generic algorithm for the case of the longest matching prefix problem. There are two cases to consider. Let n be the number of prefixes and W be the maximum length of any IP address.

In the first case W (which is the same as Q) is less than log n. In this case choose q=W. The expected run time for a search is O(1+log (n/2^q)). The other case is where W is ≥ log n. Here choose q=log n. The expected search time is O(1).

In either case, the range [0,2^W-1] is partitioned into 2^q intervals. End points of prefixes that have a value falling in the same interval form a group. In other words, all the end points whose first q bits are the same form a group. Call these groups G_0, G_1, G_2, . . .
, $G_{2^q}. G_0$ is the group of all prefixes whose first $q$ bits have the value 0. $G_1$ is the group of all prefixes whose first $q$ bits have the value 1, and so on. Figure 7-3 shows the Grouping algorithm for given any IP routing table entries.

Algorithm 6. *Grouping*

```c
{ // return groups decided by the q value
  // find the appropriate value of q in IP route database
  n = the number of prefixes;
  W = the maximum length of any IP address;
  if ( W < log n ) q = W;
  else if ( W ≥ log n ) q = log n;
  while ( end of IP route database ) {
    // find the q bits value of each IP route database for inserting proper group
    key = ipvalue >> (W − qbits);
    // find the group pointer array
    alist = pointer[key];
    if ( alist is empty ) alist = a prefix corresponding to this interval;
    else {
      // insert new ip value’s endpoints in a group with <,=,> values
      insertGroup (alist, ipvalue’s endpoints);
      // sort the group elements
      sortGroup (group(key));
    }
  }
}
```

Figure 7-3 Grouping algorithm

Each distinct end point is now associated with three values (<, =, >). These three values give more accurate next IP address for given any x IP address. For the above example, the values associated with the end points are shown below in Table 7-2.
The groups themselves can be organized either as sorted arrays or balanced trees (such as 2-3 trees, red-black trees, etc.\cite{82}). We use an array \text{Pointer}[0:2^{l-1}] to help us locate an appropriate group for any given IP address that we are interested in searching for. \text{Pointer}[i] points to the group \text{G}_i if it is nonempty; otherwise \text{Pointer}[i] points to the prefix corresponding to this interval.

Table 7-2 Distinct end points search table with <, =, > values

<table>
<thead>
<tr>
<th>End Point</th>
<th>&lt;</th>
<th>=</th>
<th>&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>P1</td>
<td>P1</td>
</tr>
<tr>
<td>12</td>
<td>P1</td>
<td>P7</td>
<td>P7</td>
</tr>
<tr>
<td>13</td>
<td>P7</td>
<td>P7</td>
<td>P1</td>
</tr>
<tr>
<td>16</td>
<td>P1</td>
<td>P2</td>
<td>P2</td>
</tr>
<tr>
<td>24</td>
<td>P2</td>
<td>P3</td>
<td>P3</td>
</tr>
<tr>
<td>31</td>
<td>P3</td>
<td>P3</td>
<td>P1</td>
</tr>
<tr>
<td>40</td>
<td>P1</td>
<td>P4</td>
<td>P4</td>
</tr>
<tr>
<td>43</td>
<td>P4</td>
<td>P4</td>
<td>P1</td>
</tr>
<tr>
<td>48</td>
<td>P1</td>
<td>P5</td>
<td>P5</td>
</tr>
<tr>
<td>51</td>
<td>P5</td>
<td>P6</td>
<td>P5</td>
</tr>
<tr>
<td>55</td>
<td>P5</td>
<td>P5</td>
<td>P1</td>
</tr>
<tr>
<td>63</td>
<td>P1</td>
<td>P1</td>
<td>-</td>
</tr>
</tbody>
</table>

Given any \(x\) to be searched for, we look at the first \(q\) bits of \(x\). Let the first \(q\) bits represent the integer \(i\). Look up \text{Pointer}[i]. If \text{Pointer}[i] points to \text{G}_i, then we search for \(x\) in \text{G}_i to output the appropriate prefix. If \text{Pointer}[i] has a prefix value, then we output this prefix. Figure 7-4 shows the Searching algorithms for given any \(x\) to be searched for the longest matching prefix in IP forwarding table.
Algorithm 7. Searching

{ // return the longest matching prefix for given any x to be searched for

  while {
      xIpValue = given any x to be searched for;
      // find the first q bits value of given any x IP address
      fqkey = xIpValue >> (W – qbits);
      // look up the group pointer array
      alist = pointer[fqkey];
      if ( alist points Group[fqkey] ) {
        // search for x in Group[fqkey] to find the appropriate prefix
        // using <,=, > values
        longPrefix = Search (Group[fqkey], endpoints of xIpValue);
        return (longPrefix);
      }
      else if ( alist points a prefix value ) {
        longPrefix = a prefix value of Group[fqkey] points
        return (longPrefix);
      }
  }
}

Figure 7-4 Searching algorithm

We illustrate our algorithm with the above numerical example. In the example, $W=6$. Say we choose a value of 3 for q. There will be eight groups $G_0$, $G_1$, . . . , $G_7$. The group $G_0$ will have all the prefixes whose first q bits have a value 0. Thus, in our example, $G_0$ has the end point 0. $G_1$ has all the end points whose first q bits have the value 1. In our example, $G_1$ have 12 and 13 end points. Similarly the other groups have the following end points: $G_2$={16}; $G_3$={24, 31}; $G_4$=Φ; $G_5$={40, 43}; $G_6$={48, 51, 55};
$G_7 = \{63\}$. Also, the array $\text{Pointer}[0:7]$ will point to $G_0, G_1, G_2, G_3, P1, G_5, G_6, G_7$, respectively.

Now consider the case of $x=001011$. Since the first three bits of $x$ has the value 1, we look up $\text{Pointer}[1]$ that points to $G_1$. We look for 11 (the decimal value of $x$) in $G_1$. There is no matching value. We realize that 12 is the closest end point and lookup the $<$ entry corresponding to the end point 12 to get the prefix P1.

Let $x=101101$ (45 in decimal). Since the first 3 bits of $x$ have a value 5, we lookup $\text{Pointer}[5]$ that points to $G_5$. We look for 45 in $G_5$ and realize that 43 is the closest end point and $x$ is greater than 43. Thus we output the $>$ entry corresponding to 43 which is P1.

As a third example, consider an $x=001100$. Since the first 3 bits of $x$ have a value 1, we look up $\text{Pointer}[1]$ that points to $G_1$. We look for 12 in $G_1$. There is a matching value. The $=$ entry corresponding to 12 yields P7.

As a final example let $x=110100$ (52 in decimal). We lookup $\text{Pointer}[6]$ that points to $G_6$. We look for 52 in $G_6$ and realize that $x$ is in between 51 and 55. The $>$ entry corresponding to 51 yields P5.

### 7.6 Experimental Results

We programmed our algorithm scheme in JAVA and measured its performance using real IPv4 prefix data obtained by the IPMA project web sites [83] (http://www.merit.edu/ipma/) and randomly generated data by IP address generator, which we developed. The simulation codes were run on a SUN ULTRA 10 Workstation. For the performance tests, we used the five IPv4 prefix data of Table 3.

To measure the search time performance in each NAP data, we calculated the execution time for each given IP address to find the longest matching prefix using $>$, $=$,
and < column in pre-computation IP lookup table. We compared our algorithm’s performance with Lampson, Srinivasan, and Varghese’s one [42].

Table 7-3 Tested NAP IP router information

<table>
<thead>
<tr>
<th>NAP name</th>
<th>Collected Date</th>
<th>Number of IP router’s Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAD’s</td>
<td>2/12/2001</td>
<td>27894</td>
</tr>
<tr>
<td>Mae-East</td>
<td>3/7/2001</td>
<td>29324</td>
</tr>
<tr>
<td>Mae-West</td>
<td>3/7/2001</td>
<td>37448</td>
</tr>
<tr>
<td>PacBell</td>
<td>2/5/2001</td>
<td>40123</td>
</tr>
<tr>
<td>Paix</td>
<td>2/18/2001</td>
<td>15589</td>
</tr>
</tbody>
</table>

7.6.1 Original IP Router Data

In these tests, we used the original NAP’s IP router information obtained by the IPMA project web site. We also used the each router’s access log information to find the search time for each IP routing requests. We simulated the same case at least twenty times and we wrote the average value of the test results. Table 7-4 and Figure 7-5 show the results of a search time for each IP router data. In these test, we obtained at most 33.1% faster search time compared with the Lampson’s algorithm [42].

7.6.2 Randomized IP Router Data

In these tests, we used the randomly generated IP address data, which are generated by IP address generator. The IP address generator generates each IP address in randomly and evenly such as even distribution of IP address from one byte to four bytes. These data showed more general case of IP forwarding requests compared with the original IP forwarding database. Thus, we believed that these test got more reliable performance results than the Section 7.6.1 test. Table 7-5 and Figure 7-6 show the results in tests.
Table 7-4 Search time for the original NAP’s IP router data

<table>
<thead>
<tr>
<th>NAP name</th>
<th>Search time for given any one of IP address (in μ sec)</th>
<th>Performance gain in our algorithm (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multicolumn [42]</td>
<td>Our</td>
</tr>
<tr>
<td>AAD’s</td>
<td>2.275</td>
<td>1.816</td>
</tr>
<tr>
<td>Mae-East</td>
<td>2.145</td>
<td>1.611</td>
</tr>
<tr>
<td>Mae-West</td>
<td>2.201</td>
<td>1.677</td>
</tr>
<tr>
<td>PacBell</td>
<td>2.284</td>
<td>1.716</td>
</tr>
<tr>
<td>Paix</td>
<td>2.145</td>
<td>1.735</td>
</tr>
</tbody>
</table>

From the simulation results, our approach achieves the great performance gain in search time compared with the Multicolumn approach [42]. In real data, our approach achieves at most 33.1% faster than the Multicolumn algorithm. In the randomized data, our algorithm has 43.7% faster than the Multicolumn one.
In the experimental results, we make sure that our algorithm has better performance gain in large IP routing table. Our algorithm divided the IP routing table data into even interval small group and this approach performed faster search given any IP address. In our grouping approach, each group had at most 4 - 5 elements and usually each group had

Table 7-5 Search time for the randomly generated IP address data

<table>
<thead>
<tr>
<th>NAP name</th>
<th>Search time for given any one of IP address (in µ sec)</th>
<th>Performance gain in our algorithm (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multicolumn [42]</td>
<td>Our</td>
</tr>
<tr>
<td>AAD’s</td>
<td>1.941</td>
<td>1.504</td>
</tr>
<tr>
<td>Mae-East</td>
<td>1.766</td>
<td>1.278</td>
</tr>
<tr>
<td>Mae-West</td>
<td>1.751</td>
<td>1.256</td>
</tr>
<tr>
<td>PacBell</td>
<td>1.885</td>
<td>1.312</td>
</tr>
<tr>
<td>Paix</td>
<td>1.859</td>
<td>1.456</td>
</tr>
</tbody>
</table>

Figure 7-6 Search time performance gain in randomized data
one or two elements in real data. This is the main factor of great performance gain in search time.

We have presented an algorithm for IP routing that has a small expected search time. The expected run time can be as small as $O(1)$. We have also presented experimental results. Our algorithm is faster in large IP routing table than in small one. Our next step for fast IP lookup is adapting our algorithm into dynamic version.
CHAPTER 8
CONCLUSION

In this dissertation, we have presented the concept, architecture, functionalities, and implementation strategy of the FAST and Scalable InTernet Service Scheme (FASTS) model. This work was motivated by the limitations we observed in the existing Internet, Web servers, and IP routers for supporting fast service for Web client’s requests. We propose the FASTS model which focus on web cache algorithms, web cache protocols, and fast and scalable IP lookup and update scheme can serve fast Internet services by supporting fast response time for a web client’s request, excellent load balance among web servers, completely and consistently updating cached static and dynamic data, and fast and scalable search and update time for IP forwarding procedure.

The FASTS model compiles of web cache algorithms, cache protocols, and fast and scalable IP forwarding approach. We can combine any algorithm and protocol, which depend on the purpose of using web servers. We offer two kind of different web server control mechanism such as direct control and distribute control. All web cache algorithms are based on BAFF (Byte Access Frequency Factor) concept. It is designed for supporting Internet services with multimedia, and static and dynamic web contents. These algorithms have a special feature treating hot documents in a web server. These cache algorithms are not only suitable for current web client’s request pattern such as non-uniform discrete random variation but also work well in ideal web client’s request pattern such as uniform discrete random variation. This means that our cache algorithms are applicable to any kind of Internet environment. DLC and RLC caching protocols
update cached static and dynamic contents completely and consistently on a web server. DLC and RLC allow duplication of dynamic web content among web servers. For fast and scalable IP lookup and update algorithm in IP forwarding table, we show how to modify binary search to adapt into longest matching prefix in IP lookup using grouping scheme. We also show how to find the longest matching prefix faster and more accurately in forwarding table for given any IP address. Our algorithm has good performance in lookup, excellent scalability, and does not require protocol change. Our algorithm also allows a cheap, fast software implementations. Our algorithm divides large IP routing table data into even interval small group and provides fast and scalable search time for given any IP address. This is the main factor of great performance gain in search time. Our algorithm is faster in large IP routing table than in small one. This is good for the current large IP routing table data for speeding up lookup and update time.

There are some issues that need to be addressed in the future research. First, there are security issues related to original web server and cooperative web servers. Since they communicate each other to send/receive a message for update the cached contents in each web server. The potential damage caused by not having the proper security control among web servers can be substantial. Second, in this work, we consider web server side cache algorithms and protocols. To achieve faster response time for web client’s requests, we expand these algorithms and protocols to network server side cache algorithms and protocols. This work must consider the network packet routing between network servers and web servers, and the authentication mechanism for cached data between network servers and web servers. Third, we can combine our web cache algorithms and protocols with real time database management systems for supporting real time web contents.
Many areas such as e-commerce, sports web sites, stock exchange, and etc. currently use these applications. Finally, we expand our fast and scalable IP lookup scheme to dynamic version. To support gigabit rate IP lookup and update, the dynamic version of IP lookup scheme is needed. To do this work, we consider update time and memory usage in IP routing table. This work must consider the construction of forwarding table in small and simple structure and the dynamic location of pre-computation information for speed up IP lookup and update procedure. We also consider the adaptability of our work to the IPv6 to support the scalability and flexibility.
LIST OF REFERENCES


[38] Pak W. and Bahk S., ‘Flexible and Fast IP Lookup Algorithm’, IEEE International Conference on Communications, June 11-15, 2001, St.Petersburg, Russia


[46] Chen W. and Tsai C., ‘A Fast and Scalable IP Lookup Scheme for High-Speed Networks’, IEEE International Conference on Networks, September 28 - October 1, 1999, Brisbane, Australia


[54] Turner J., ‘Design and Analysis of Switching systems’, Washington University, St. Louis, Missouri, January 1999


[75] Lim J. and Rajasekaran S., ‘Directly Selected and Limited Look up Cache Algorithms for Static and Dynamic Web Contents’, IASTED International Conference on Networks, Parallel and Distributed Processing, and Applications, October 2-4, 2002, Tsukuba, Japan


BIOGRAPHICAL SKETCH

Jaeyong Lim was born on October 17, 1968, in Pusan, Republic of Korea. He received his Bachelor of Science degree in computer science and engineering from HanYang University, Seoul, Republic of Korea, in February 1994. He then joined the Information Systems Lab of the Department of Computer Science and Engineering at the Korea Advanced Institute of Science and Technology, Daejoun, Republic of Korea, and received his Master of Science degree in February 1996. He worked as a senior research engineer for two and half years at the Code Division Multiple Access System Development Team of Network Group of Samsung Electronics, Seoul, Republic of Korea. He joined the Department of Computer and Information Science and Engineering at the University of Florida in August 1999 to pursue a Doctor of Philosophy degree. Since then he has worked as a research assistant and a teaching assistant in the Parallel and Randomized Algorithm and System Research Lab of the department. His research interests include web caching, web cache protocol, parallel web server architecture, fast IP routing, real time systems, real time database management systems, mobile communication database and systems, and storage area network.