BENCHMARK-BASED REPLACEMENT PAGE (BBPR) STRATEGY: A NEW WEB CACHE PAGE REPLACEMENT STRATEGY

Wei He

Problem in Lieu of Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

May 2003

He, Wei. Benchmark-based Page Replacement (BBPR) Strategy: A New Web Cache Page Replacement Strategy. Master of Science (Computer Science), May 2003, 50 pp., 5 tables, 25 figures, references, 15 titles.

World Wide Web caching is widely used through today's Internet. When correctly deployed, Web caching systems can lead to significant bandwidth savings, network load reduction, server load balancing, and higher content availability. A document replacement algorithm that can lower retrieval latency and yield high hit ratio is the key to the effectiveness of proxy caches. More than twenty cache algorithms have been employed in academic studies and in corporate communities as well. But there are some drawbacks in the existing replacement algorithms. To overcome these shortcomings, we developed a new page replacement strategy named as Benchmark-Based Page Replacement (BBPR) strategy, in which a HTTP benchmark is used as a tool to evaluate the current network load and the server load. By our simulation model, the BBPR strategy shows better performance than the LRU (Least Recently Used) method, which is the most commonly used algorithm. The tradeoff is a reduced hit ratio. Slow pages benefit from BBPR.

ACKNOWLEDGEMENTS

A couple of people I would like to acknowledge here. I wish to thank my advisor-Dr. Armin R. Mikler, who introduced the idea of this paper and directed me from the very beginning to the completion. Dr. Mikler gave me constant guidance and encouragement throughout the paper. His patience and kindness deeply impressed me. I would like to thank my wife, Guili Sun, for spending a lot of time looking for literature, checking the grammar and even helping me to execute the program. For everyone in the NRL (Network Research Lab), I simply say "Thanks"! They are very willing to give me valuable advice in our group meeting. I appreciate the Computer Science Department, who gave me financial support for two years. Without that I could not imagine I would finish my degree. Last, I would like to say "thank you" to my family and all my friends. I wish all the people have a happy life.

CONTENTS

ACKNOWLEDGEMENTS iii

1 Introduction 1

1.1 World Wide Web(WWW) and HyperText Transport Protocol(HTTP) 1

1.2 Proxy Server and Cache 3

1.3 Need for a Page Replacement Algorithm 6

2 Background 9

2.1 Total Latency 9

2.2 Review of Existing Cache Management Algorithms 11

2.2.1 Traditional Replacement Policies and their Direct Extensions . 12

2.2.2 Key-based Replacement Policies 13

2.2.3 Function-based Replacement Policies 14

3 Methodology of BBPR 20

3.1 Partitioning the Cache 21

3.2 Finding the Latency Threshold 22

3.3 Simulation 27

4 Result Analysis 32

4.1 Total Requests vs. Hit Ratio 32

4.2 Average Total Latency 36

4.3 Detailed Investigation for BBPR 38

4.3.1 Distribution of Total Latency 38

4.3.2 Incrementing the Latency Threshold 39

4.3.3 Slow Pages Benefit from BBPR 42

4.3.4 Overhead Analysis 43

5 Summary and Conclusions 45

5.1 Summary of BBPR 45

5.2 Limitations and Future Work 47

BIBLIOGRAPHY 49

LIST OF TABLES

4.1 URL Files 32

4.2 5M–Distribution of Total Latency 37

4.3 6M–Distribution of Total Latency 38

4.4 Page Replacement in Case 1(α=1.25) and Case 2(α=1.0) 41

4.5 Comparison of Number of Requests Completed in 1 Hour and the Hit Ratio in Case 1(α=1.25) and Case 2 ($\alpha = 1.0$) 41

LIST OF FIGURES

1.1 Client and Server with HTTP 2

1.2 Proxy Server *(Proxy server works as an intermediary between the browsers and the remote web servers by sitting behind the organization's network)* 4

1.3 Cache Works *(When the requested files are in the cache, the cached files will return to the clients directly without fetching through the external network)* 5

2.1 HTTP File Transfer 9

3.1 Cache Partition *(Cache is divided into two parts by setting replaceable or nonreplaceable flag. Then when the replacement occurs, it only needs to select file(s) randomly from the pages with replaceable flag so that iteration time is saved.)* 21

3.2 System Organization 22

3.3 Flow Chart of the BBPR Strategy 23

3.4 Simulation Systems 28

4.1 1M–Pages Visited in 1 Hour 33

4.2 1M–Hit Ratio 33

4.3 2M–Pages Visited in 1 Hour 34

4.4 2M–Hit Ratio 34

4.5 3M–Pages Visited in 1 Hour 35

4.6 3M–Hit Ratio 35

4.7 4M–Pages Visited in 1 Hour 35

4.8 4M–Hit Ratio 35

4.9 5M–Pages Visited in 1 Hour 36

4.10 5M–Hit Ratio 36

4.11 5M–Comparison of Average Total Latency 37

4.12 6M–Number of Pages Replaced in 1 Hour by Case 1 and Case 2 . . . 40

4.13 6M_slow–Pages Visited in 1 Hour 42

4.14 6M_fast–Pages Visited in 1 Hour 42

4.15 6M_slow–Hit Ratio 43

4.16 6M_fast–Hit Ratio 43

4.17 Cumulative Overhead of Executing the Benchmark("6M.in", Cache Size = 3MB) 44

CHAPTER 1

Introduction

1.1 World Wide Web(WWW) and HyperText Transport Protocol(HTTP)

The World Wide Web(WWW) is a large distributed information system that provides access to shared data objects. The WWW has been experiencing exponential growth since its origin in 1989. The number of static Web pages increases approximately 15% per month. About 7.3 million new pages are added on the WWW every day. There are about 32 million domains registered worldwide with about 22 million .com among them[1]. It is expected that the number of Internet users will continue to grow strongly in the next five years even during the economic depression. The number of worldwide Internet users will be 673 million at the end of 2002. By year-end 2005 the number of worldwide Internet users will surpass 1 billion.[2]

HyperText Transport Protocol (HTTP) requests dominates the Internet traffic up to 90%[3]. HTTP is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol that can be used for many tasks, such as name servers, and distributed object management systems, through extension of its request methods (commands). A feature of HTTP is to represent the typies of data, which allows systems to be built independently of the data being transferred. HTTP has been in use by the World Wide Web global information initiative since 1990.

HTTP is implemented both at client's side and server's side (See Figure 1.1).

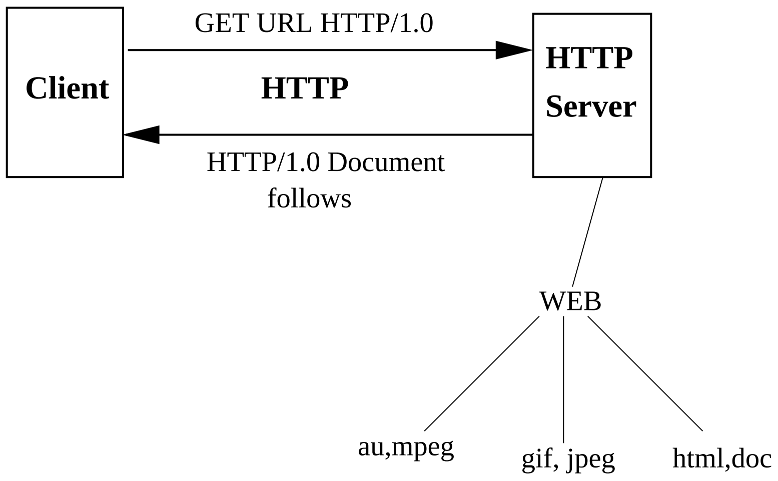

Figure 1.1: Client and Server with HTTP

A *client* is an application program that establishes connection for the purpose of sending requests. A *server* is an application program that accepts connections in order to service requests by sending back responses. Client and server can *talk* to each other by exchanging HTTP messages. HTTP defines the structure of these messages and how the client and server exchange the messages. There are two types of HTTP messages, request messages and response messages. Request messages sent from the client to the server tell the server what kind of the documents the client wants and the response messages from the server to the client return the information, which is either documents the client wants, or error messages. A *Web page* (also called a *document*) consists of objects. An *object* is simply a file, such as an HTML file, or a GIF image. There are six basic HTTP document types: HTML, images (e.g. gif), sound (e.g. au and wav), video (e.g. mpeg), dynamic (e.g. cgi), and formatted (e.g. ps, dvi). HTML is the initial for HyperText Markup Language. It is the language for publishing hypertext on the WWW. In HTTP 1.0, there are three different request messages: *GET*, *POST* and *HEAD*. These three commands are used to fetch the corresponding web content. The HTTP GET command is used to transfer pages, images, and other content viewed through a Web browser. Similar to GET, an HTTP HEAD request asks to retrieve only the HTTP response header for a document but not the document itself. POST passes form data to the server for use as input to some CGI program. The great majority of HTTP request messages use the GET method.

1.2 Proxy Server and Cache

Although the Internet backbone capacity increases 60% per year, today's Internet users still have to endure the consequences of two major problems: the network congestion and the server overload. Efforts to improve the performance of the Internet went back as far as its origin. New strategies must be developed to solve these problems, otherwise the WWW would become too congested and the webpages could not be retrieved as needed.

The time of accessing a document from the client to the server is referred as to *retrieval time* or *total latency.* One way to reduce the total latencies, the network congestion, and the servers' load is to store the copies of the Web documents in a proxy server. A proxy server serves as an intermediary between the web browsers and the servers in the World Wide Web. Figure 1.2 shows that the proxy server acts as a client for requesting documents from a remote server.

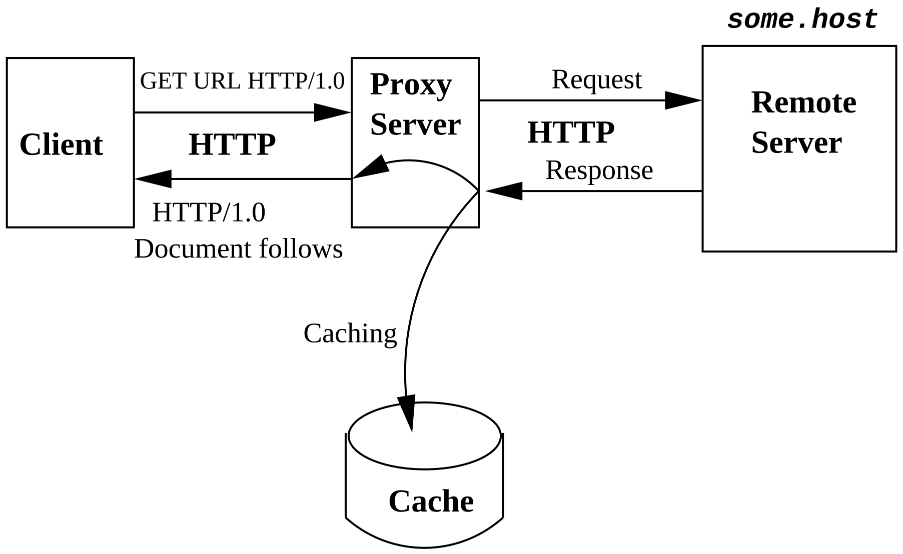

Figure 1.2: Proxy Server *(Proxy server works as an intermediary between the browsers and the remote web servers by sitting behind the organization's network)*

The primary use of a proxy server is to allow internal clients access to the Internet from behind a firewall. It intercepts all requests to the real server to see if it can fulfill the requests by itself. Otherwise, it forwards the request to the real server. Hence, it acts as both a server and a client on behalf of different requests. The proxy server employs specific caching strategies with the goal to optimize access to frequently used pages.

Figure 1.3 shows that it acts as a server for responding the request from the browser. Caching refers to the storing of copies of documents and other objects by a proxy server, so that this document is readily available for others who request to retrieve. The *cache* is a program's local store of response messages and the subsystem that controls its message storage, retrieval, and deletion. Regular HTML files are usually *cacheable*. Caching may require a valid Last-Modified header, and may not cache objects greater than a certain size or subject to other restrictions. A cache stores cacheable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. HTML documents generated by CGI scripts can be made cacheable or not by generating an Expires header, though some proxy server may not cache URLs with "cgi-bin" or a query string. Documents requiring authorization should not normally be cached.

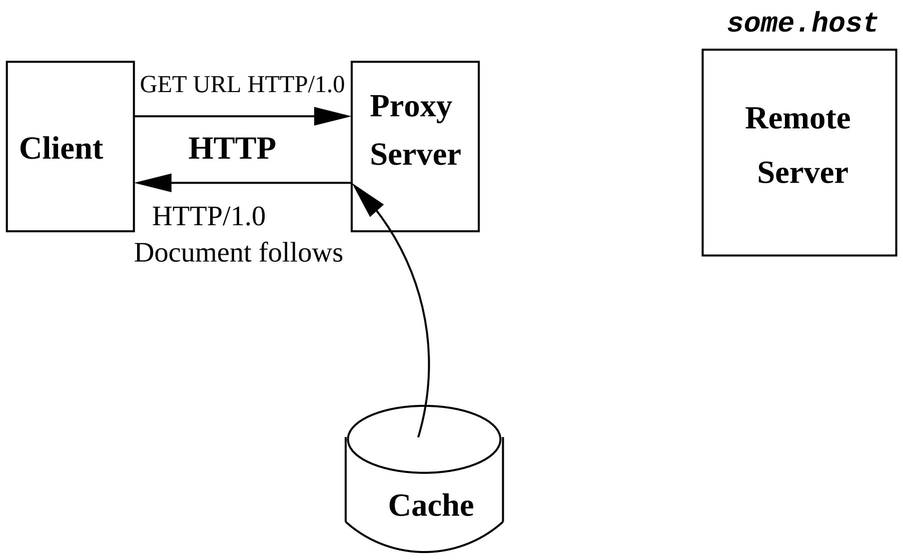

Figure 1.3: Cache Works *(When the requested files are in the cache, the cached files will return to the clients directly without fetching through the external network)*

Proxy caching could improve Internet performance in three ways. First, caching attempts to reduce the total latency associated with retrieving Web documents. Latency can be reduced because the proxy cache is located much closer to the client than the remote server. Second, caching can lower the network traffic. The network traffic can be reduced as documents are retrieved from the cache rather than from the network. Finally, proxy caching can reduce the servers' load since cache hits do not need to involve the content provider(server). It may also lower transit costs for access providers. Therefore, the network traffic, average latency of fetching Web pages and the load on busy web servers would be reduced.

On the other hand, there are some disadvantages of web caching. First, stale pages could be serviced due to the lack of proper updating. Second, latency may increase in case of a *cache miss* (the case when the page is not in the cache). Third, a single proxy cache may constitute a bottleneck. Fourth, some websites prefer to have a large volume of requests, but a proxy cache will reduce a hit rate to the original servers. Nevertheless, the advantages are deemed more significant than the drawbacks, and today, web caching is widely used through the Internet.

1.3 Need for a Page Replacement Algorithm

Because of the limited size of a cache, it may be necessary to remove/reduce some old documents in the cache when the cache is full or above a limit (generally, some free space is left in case that the cache is overwhelmed with pages). A document replacement strategy is needed at this time. A document replacement algorithm that can yield high hit ratio is the key to the effectiveness of proxy caches. Since the replacement algorithm decides which documents are cached and which documents are replaced, it affects the cache hits of future requests. Many cache algorithms have been proposed in recent studies, which attempt to minimize cost such as average latency and network load and at the meantime maximize the *hit ratio* and the *byte hit ratio*. The hit ratio is defined as the percentage of requests that can be served from previously cached document copies. The byte hit ratio is the percentage of the number of bytes sent by the cache divided by the total number of bytes sent to its clients.

In this paper we introduce a new algorithm to minimize the average latency by using a HTTP benchmark combined with a simple and most commonly used replacement algorithm, namely *Least Recently Used (LRU)*. This new strategy is a Benchmark-Based Page Replacement Strategy (BBPR). A benchmark is a publicly defined procedure designed to compare the performance of systems. A HTTP(Web) benchmark is basically a mechanism to generate a controlled stream of Web requests, with standard metrics to report the results. Generally, a HTTP benchmark consists of a set of client programs that emit a stream of HTTP requests, and measure the system response. Usually response time and throughput are measured. Simply speaking, for a certain system, a HTTP benchmark can be used as a tool to evaluate the workload of the network and the server by measuring the response time (latency) of HTTP request. Here we will use this result of benchmark combined with a known strategy (LRU) to reduce the average latency. The BBPR methodology is to be discussed in more detail in Chapter 3.

The remainder of the paper is organized as follows. First, we will review the existing replacement algorithms and compare and contrast them. Second, we will explain our new strategy and the implementation in detail . Third, we will discuss the results of our experiment and show the difference from the common used LRU method. Finally, we will conclude with a summary of our proposed replacement strategy.

CHAPTER 2

Background

2.1 Total Latency

When HTTP is used to request a file from a server, it works in the manner shown in Figure 2.1. TCP (Transmit Control Protocol) is the protocol suite upon which

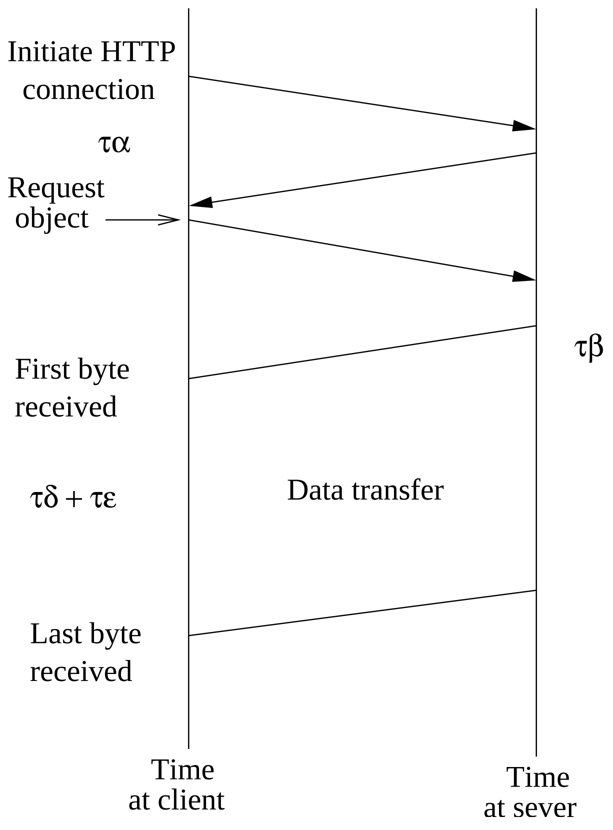

Figure 2.1: HTTP File Transfer

the Internet is based. The client sends a TCP connection request to the server. A

TCP connection is established with a 3-way handshake. The client sends a TCP SYN segment to the server. The server sends a TCP SYN segment and acknowledgement to the client. The client sends an acknowledgement back to the server. The client's acknowledgement is combined with the HTTP get file request. When the request is received at the server, the server begins sending the file to the client. When the entire file has been received, the transaction is complete. In another way, we can say that the access time to remote documents (in this paper we refer to as retrieval time or total latency) is the sum of following parts:

* t_α: the time for the client request to reach the server

* t_β: the time at the server to process the request

* t_δ: the time for the response to reach the client from the server

* t_ϵ: the time at the client to process the response

Times t_α, t_β and t_ϵ depend on the hardware platforms, operating systems both at the client and at the server. Time t_β also depends on the server's software and the load at the server. Times t_α and t_δ depend on locality (propagation delay) and network situation (bandwidth, network workload and routers' condition etc). Times t_β, t_δ and t_ϵ also depend on the size of the document asked by the client.

The proxy server is generally located on the organization's (like company and campus) network, so the access time to the pages which are in the cache could be much reduced because of the near locality. The primary goal of proxy-based cache is to reduce the amount of retrieval time by reducing the latency caused by network and web server load that are external to the organization. The choice of what kind of web pages should be cached is very important to the performance of proxy server. Obviously, the pages with high probability of being visited by the clients are preferred to be kept. Also, the pages should be kept such that the average latency of retrieving web pages could be minimized. Hence, we need to not only consider the frequency of being visited, but also take into account the cost of each cached page.

2.2 Review of Existing Cache Management Algorithms

As we mentioned above, the key factor of the effectiveness of proxy caches is a document replacement algorithm that attempts to minimize various costs, such as hit ratio, byte hit ratio, and average latency of retrieving files. Some replacement algorithms are developed from virtual memory paging replacement as we will discuss in the first part of this section. There are three primary differences between Web caching and conventional paging problem. First, the size of Web cache and the size of Web pages are both variable. Second, because of their different localities it takes a different amount of time to download different web pages even if they are of the same size. Third, the users who access the proxy cache vary from a big range, but there are a few programmed resources that will use the virtual memory paging.

The *size* and *latency/cost* of the pages in the cache make web caching more complicated than traditional virtual memory paging replacement. The variable document sizes in web caching make it much more difficult to determine an optimal replacement algorithm. If a sequence of requests is given to uniform size blocks of memory, the optimal performance is achieved by removing the next farthest request in the future. But for variable-size case, determining the optimal replacement has been proven NP-hard[4].

The latency/cost consideration is even more complicated than the size because the cost for each page is more dynamic. There are many factors which can effect the cost, such as size of the page, locality, network load and the servers' load. Therefore, the algorithms designed to reduce the cost would be more complex.

The algorithms designed for page replacement can be classified into the following three categories as suggested in [5] and [6]:

1. Traditional replacement policies and their direct extensions

2. Key-based replacement policies

3. Function-based replacement policies

In this paper we will review totally nineteen different replacement algorithms designed to reduce the retrieval time in recent studies.

2.2.1 Traditional Replacement Policies and their Direct Extensions

The policies in this category are well-known cache replacement strategies and are developed from virtual memory page replacement.

* *First In First Out* (FIFO): removes the page first stored in the cache. The pages in the cache are associated with time. The earliest cached page will be removed when the cache is full or the size is greater than a threshold.

* *Least Recently Used* (LRU): removes the least recently accessed page when the replacement occurs. It is the most common strategy used in today's proxy cache and is very straightforward.

* *Least Frequently Used* (LFU): removes the least frequently accessed page. It is associated with the frequency of each page in the cache accessed by the client. The often accessed pages will remain available to be accessed later.

* *Pitkow/Recker*[7] removes pages in LRU order, but if all pages are accessed within the same day, the page with the largest size is removed.

The advantage of the policies in this category is its simplicity. The disadvantages of these policies are that the above policies fail to take into account object sizes except the last one and they do not consider the cost of each page when caching it.

2.2.2 Key-based Replacement Policies

The replacement strategies in this category remove the pages in the cache based on a primary key, breaking ties on secondary key, tertiary key, and so on.

* *Size*: the pages in the cache are removed in order of size, with the largest one removed first. The cached Web pages with the same size are somehow rare, but when it happens, ties are broken by using LRU policy or by random selection.

* *LRU-MIN*: a variant of LRU that is designed in favor of smaller sized objects so as to minimize the number of pages replaced. Given the size of the incoming page is S. When the replacement occurs, the pages in the cache with at least

size S which are least recently used will be removed. If there are no pages with size at least S, the pages with size S/2 will be considered, then pages of size at least S/4, and so on until there is enough free cache space. This algorithm is also named as *Log(Size)+LRU*, because largest log(size) is considered.

* *LRU-Threshold*: it is the same as LRU, but pages larger than a certain threshold size are never cached. It avoids storing large pages in the cache so that more pages could be accessed from the cache.
* *Hyper-G*: a refinement of LFU, break ties using the recency of last use and size.
* *Lowest Latency First*: minimizes the average latency of retrieving files by removing the document with the lowest download latency first.

The factors affect the overall performance of the replacement algorithms include locality, size and latency/cost associated with the documents in the cache. The policies in this category attempt to combine these factors to achieve the best result. But in the real Web these policies may not always be ideal, because the prioritization proposed above could not always reflect the reality.

2.2.3 Function-based Replacement Policies

The replacement policies in this category employ a potential cost function derived from different factors such as time since last access, entry time of the object in the cache, transfer time cost, object expiration time and so on.

* *GreedyDual*[8]: The name GreedyDual comes from the technique used to prove that this entire range of algorithms is optimal according to its *competitive ratio*. The competitive ratio is essentially the maximum ratio of the algorithm cost to the optimal offline algorithm's cost over all possible request sequences[9]. This algorithm is a range of algorithms which include a generalization of LRU and a generalization of FIFO. The algorithm associates a value, H, with each cached page p. At the time the page is cached, H is set as the cost of bringing the page into the cache. When a replacement occurs, the page with the lowest H value, min_H, is replaced, and then all pages reduce their H values by min_H. If a page is accessed, its H value is restored as its cost. Thus, the H values of recently used pages retain a larger portion of the original cost than those of pages that have not been accessed for a long time. In this way, the algorithm combine the locality and cost of cached pages.

* *GreedyDual-Size(GD-Size)*[4]: a variation on GreedyDual algorithm. It attempts to combine locality, size and latency/cost concerns effectively to achieve the best overall performance. GD-Size algorithm extends the GD algorithm by setting H to *cost/size* upon an access to a document, where *cost* is the cost of bringing the document, and *size* is the size of the document in bytes.

* *Hierarchical GreedyDual(Hierarchical GD)*: does page placement and replacement cooperatively in a hierarchy.

* *Hybrid*[10]: it sorts cached pages by a utility function and removes the one has the least utility value to reduce the total latency. The utility value depends on the following parameters: c_θ, the time to connect with server s, b_θ the bandwidth to server s, n_p, the number of times p has been requested since it was stored into the cache, and z_p the size (in bytes) of document p. The function is defined as:

  $$f(c_\theta, b_\theta) = \frac{(c_\theta + \frac{W_b}{b_\theta})(n_p)^{W_\theta}}{z_p}$$

  where W_b and W_n are constants that set the relative importance of the variables c_θ and b_θ respectively. Estimates for c_θ and b_θ are based on the times to retrieve documents from server s in the recent past.

* *Lowest Relative Value (LRV)*[11]: it also sorts cached pages by a utility function and removes the page which has the lowest value. The value is calculated by the cost and size of each document in the cache. The calculation is based on extensive empirical analysis of trace data. LRV's utility functions also consider locality, cost and size of a document.

* *Size-Adjusted LRU (SLRU)*: orders the pages by the ratio of cost to size and chooses pages with the *best* cost-to-size ratio to be replaced first. The *best* here means the *smallest*. The pages with high cost-to-size ratio will benefit from this algorithm.

* *Server-assisted Scheme*[12]: removes the pages which are the least likely to be accessed in the near future. The proxy can also perform pre-fetching based

on accurate estimates of future request to avoid burdening the server and the network with extra transmissions. The *hints* (in groups called *volumes* included in the server response messages are used as tunable parameters to calculate the future access probabilities of the cached pages. When replacement occurs, the pages with the lowest future access probabilities will be removed first. A greedy algorithm is used to construct the volumes, but it incurs a significant computational overhead. However, another new algorithm named as pairwise volumes is a faster alternative.

* *Least Normalized Cost Replacement (LCN-R)* [13]: maximizes the *delay-savings ratio* which is a performance metric used to generalized the hit ratio metric by explicitly considering cache miss cost. The delay-savings ratio(DSR) is defined as:

$$DSR = \frac{Sum_i(d_i * h_i)}{Sum_i(d_i * r_i)}$$

where d_i is the average delay to fetch document D_i to cache, r_i is the total number of references to D_i and h_i is the number of references to D_i which were satisfied from the cache. LNC-R maintains the following statistics with each cache document D_i: l_i - average rate of reference to document D_i, s_i - size of document D_i, d_i - delay to fetch document D_i to cache. In order to maximize the DSR, the above statistics are combined into one performance metric, called *profit*, defined as $profit(D_i) = (l_i * d_i)/s_i$. LNC-R selects for replacement the least profitable documents.

* *Least Normalized Cost W3 Replacement (LCN-R-W3)*: an extension of LCN-R. This algorithm bases the estimate of average reference rate not only on the past reference pattern as LNC-R, but also on the size of the document.

* *Bolot/Hoschka Replacement*[14]: it uses a weighted rational function based on the retrieval time associated with each document in the cache to determine which document should be replaced first. One weight function is suggested as:

$$W(ti, Si, rtti, ttli) = (w1 * rtti + w2 * Si)/ttli + (w3 + w4)/ti$$

where $w1$, $w2$, $w3$, and $w4$ are constants, ti is the time since the document was last referenced, Si is the size of the document, $rtti$ is the time it took to retrieve the document, $ttli$ is the time to live of the document(i.e., the expected time until the document will be updated at the remote site, which is also the time interval until the cached document becomes stale).

The improvement of the policies in this category is that they are aware of the cost of retrieving pages and consider the factors which determine the cost. Some of them update the factors dynamically, some of them not. The disadvantage of this category is their extensive parameterization, which introduces uncertainty about their performance across the Internet in the real world.

Although there are many different cache replacement algorithms existing, feasible choices for actual Web proxy cache narrowed down to LRU, SIZE, Hybrid, and LRV. Some studies [15] [7] say that SIZE outperforms LFU, LRU-threshold, SIZE+LRU, Hyper-G and Pitkow/Recker. Some [11], however, says LRU performs better than SIZE. Studies in [4] show LFU performs worse than LRU in most cases.

In summary, even with the many algorithms existing, there is no conclusive agreement of which replacement strategy is the best, because the performance of different policies depends highly on the characteristics of WWW's traffic. No known policy can outperform others for all Internet access patterns. Today, most proxy systems still use some form of the Least-Recently-Used(LRU) replacement algorithm.

We introduce a new cache replacement algorithm to minimize the average latency by using a HTTP benchmark. The benchmark is used to measure and evaluate the network load and how busy the server is and then the information will be given to arrange the cached pages. This new algorithm considers the transfer latency/cost and the network load and server load to achieve the best overall performance. In next chapter we will discuss this new algorithm in more detail.

CHAPTER 3

Methodology of BBPR

There are three main shortcomings for the existing replacement algorithms. First, the most commonly used traditional methods consider the frequency of request, sometimes the size of the documents, but they do not take retrieval time into account. Because the purpose of proxy cache is to minimize the average retrieval time of fetching the Web pages, ignoring retrieval time may not result in the expected improvement, especially in cases where retrieval time changes continuously. Second, most strategies that depend on the cost either have too many parameters or use static formulas. Strong parameter dependency may not reflect accurately the real situation and the static formulas can not capture the change of the cost dynamically. The cost/latency of fetching a Web page is not a constant. It may change over time due to the traffic on network and the load of the server. For instance, the total latency of fetching pages from business websites is generally longer at daytime than at night. In contrast, the latency for some oversea webpages may be less at daytime than at night due to the time difference. Therefore, the strategies that order the documents in the cache according to the static patterns may not be optimal. Third, almost every proposed algorithm sorts the documents in the cache in a sequence according to a certain value such as the last access time in LRU and the frequency in LFU. When replacements occur they will remove the page with least or largest value. This is time-consuming when there are many documents in the cache because the selection procedure has to iterate through the whole document list to find the lowest or highest valued page. We developed a new page replacement strategy to overcome these drawbacks. The name of the new strategy is Benchmark-Based Page Replacement Strategy (BBPR).

3.1 Partitioning the Cache

To reduce the iteration time of finding a page to remove, we separate the documents in the cache into two parts by assigning a flag called replaceable/non-replaceable to each individual page (see Figure 3.1). When the cache is full, we will *randomly* choose

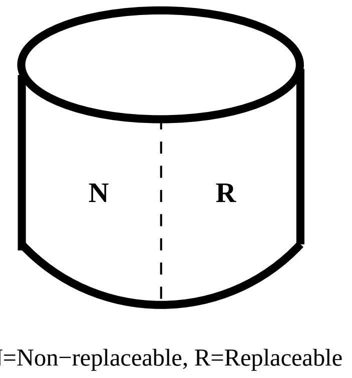

Figure 3.1: Cache Partition *(Cache is divided into two parts by setting replaceable or nonreplaceable flag. Then when the replacement occurs, it only needs to select file(s) randomly from the pages with replaceable flag so that iteration time is saved.)*

one or more pages to be removed only from the pages that have the *replaceable* flag set. The pages in the non-replaceable portion will be kept in the cache. Because of random choice we do not need to go through the entire list to compare the key value. The flag will be set as replaceable if the total latency of this page is smaller than a cost *threshold* which we will see next. Otherwise, the flag will be set to non-replaceable.

3.2 Finding the Latency Threshold

The threshold for total latency, which is decided by an HTTP benchmark, is the key point to our algorithm. The threshold is the limit value that if the total latency of fetching a Web page is lower than it, a flag will be set as replaceable and this page will be put into the replaceable portion in the cache. The *optimal* threshold we are looking for is the value which could reflect the workload of the network and the servers precisely and make the cache work most effectively. We use an HTTP benchmark to calculate an appropriate value. An HTTP benchmark is a simple tool to evaluate the workload of the network and the server. We establish the relationship between the benchmark and the proxy server to make the entire system work efficiently (See Figure 3.2). Figure 3.3 is the flow chart of our new strategy.

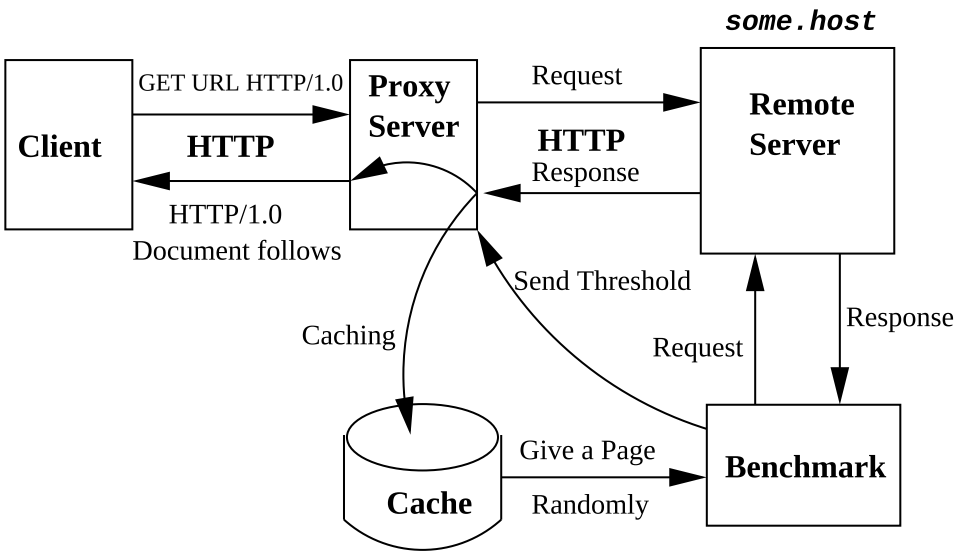

Figure 3.2: System Organization

Clients keep sending HTTP requests to the proxy server. At a certain interval

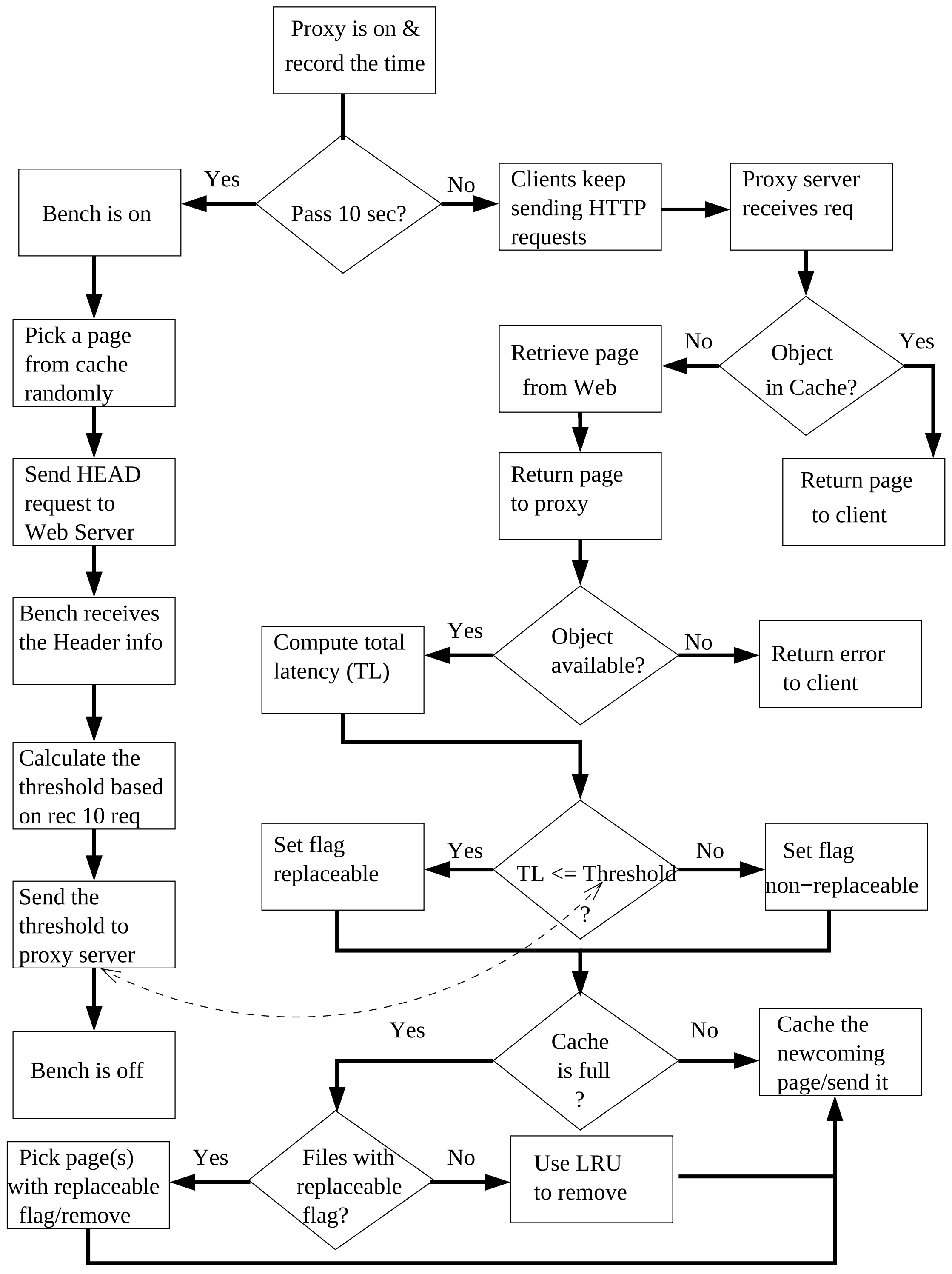
Proxy is on & record the time
Pass 10 sec?
Yes
No
Bench is on
Clients keep sending HTTP requests
Proxy server receives req
Pick a page from cache randomly
Object in Cache?
No
Yes
Retrieve page from Web
Send HEAD request to Web Server
Return page to proxy
Return page to client
Bench receives the Header info
Compute total latency (TL)
Yes
Object available?
No
Return error to client
Calculate the threshold based on rec 10 req
Set flag replaceable
Yes
TL <= Threshold ?
No
Set flag non−replaceable
Send the threshold to proxy server
Bench is off
Yes
Cache is full ?
No
Cache the newcoming page/send it
Pick page(s) with replaceable flag/remove
Yes
Files with replaceable flag?
No
Use LRU to remove

Figure 3.3: Flow Chart of the BBPR Strategy

time, the benchmark will start to execute. It will randomly pick a page in the cache, send the HTTP request to the Web server and record the latency of fetching this page. We will choose the recent total latency of some different Web pages to develop a formula to derive the threshold and send it to the proxy server. The cache can set replaceable/non-replaceable flag to different cached pages based on the threshold. There are three types of HTTP/1.0 request methods:

* *GET*: is used when the client requests an object. It will respond the client with the whole content of the requested Web page.

* *HEAD*: is identical to GET except that the server must not return any Entity-Body in the response. The meta-information contained in the HTTP headers in response to a HEAD request should be identical to the information sent in response to a GET request. This method can be used for obtaining meta-information about the resource identified by the Request-URI without transferring the Entity-Body itself. It is often used to test hypertext links for validity, accessibility, and recent modification. This command will provide an estimated retrieval time of the page that has to be fetched at some particular point.

* *POST*: the HTTP client uses the POST method when the user fills out a form. With a POST method, the client is still requesting a Web page from the server, but the specific contents of the Web page depend on what the client wrote in the form fields.

The only purpose of the total latency is to evaluate how busy the network and the server is, and the requests sent by the benchmark will not be used by any clients, so we do not need to download the entire Web page (large size page transfer will introduce unnecessary overhead). The method for this purpose we chose is to use the *HEAD* command to avoid fetching the whole page.

An array $L_r[index]$ is used to keep track of the total latency of recently chosen pages by the benchmark. The *index* is the parameter that determines how many recently selected pages are used to calculate the threshold. It could be any integer number. In our design, it is set to 10, which means the total latencies of 10 recently selected pages are used to calculate the threshold. Before the benchmark is executed, each item of $L_r[]$ array is initialized to 100ms. L_s is the sum of the latency array, i.e.,

$$L_s = \sum_{i=0}^{9} L_r[i]$$

L_θ is the total latency of the new page randomly selected by the benchmark. When the new L_θ arrives, the threshold is updated by the following formula:

$$L_s = \sum_{i=0}^{9} L_r[i] - L_r[k] + L_\theta$$

where $L_r[k]$ is the total latency of the least recently selected page in the previous 10 pages. The latency of the oldest page is subtracted and the latency of the newest page is added, hence a new L_s is calculated. We can get the threshold by calculating the average latency of these recently selected pages.

$$threshold = L_s/index$$

where the *index* is equal to 10. We can execute our benchmark by picking 20 newest pages. The interval time can also set to 10 seconds, 20 seconds, 30 seconds, or longer. To reduce the overhead, we can execute the benchmark between a longer interval. However, if the interval time is too long, the data may not reflect precisely the current real network load. Besides, a timeout limit of 500 milliseconds in the benchmark is set to reduce the overhead caused by sudden change of network's congestion. If the retrieval time for a particular page is longer than the timeout limit, this request is simply ignored.

The problem with the *HEAD* method is to determine whether the average latency can reflect the actual load of network and server and the average locality. The latency introduced by using HEAD method includes the time of establishing the socket between the client and the server and the time of retrieving the header information of the requested Web page. The size of the headers of all pages is about 300 bytes. We have found the total time to retrieve the header is almost always less than the latency of fetching the entire page. Because we choose pages randomly from the cache, our method is a valid way to set the threshold for the cache. The results of our experiment show that our method results in a reduced latency as compared to LRU.

LRU is used as a backup replacement strategy. If all pages in the cache have been set non-replaceable flag, when a replacement has to occur, the pages in the cache can not be removed by the replaceable flag because there are no pages in the cache with replaceable flag set. At this time, LRU is used as the replacement algorithm. Since we always replace the pages with the replacement flag set first, so the pages with larger total latency would stay longer in the cache. There exists a relationship between the pages replaced by using replaceable flag and the pages replaced by using LRU. We will discuss this relationship between the cache size and the threshold in Chapter 4.

3.3 Simulation

A client-server model has been designed for the simulation. The performance by the BBPR algorithm and the LRU algorithm will be compared. The cached pages are stored in the local *tmp* directory in the *csp* system at our campus. The processor of the terminal where the simulation executes is a 2GHz *Pentium 4* processor. Figure 3.4 shows how the components cooperate in our simulation system.

All the modules have been implemented in JAVA. They include the following classes: Cache, Configuration, Proxy, Benchmark, Daemon, Data, Client, RandomeInt-Generator, and url_list.

Cache creates a directory named “cache” in the local “tmp” directory to store the cached files. Manages all caching activities. A *hashtable* is used to store the information of all cached files. The information includes the attributes declared in the *Data* Class.

Configuration sets configurable parameters of the proxy. This class is used both

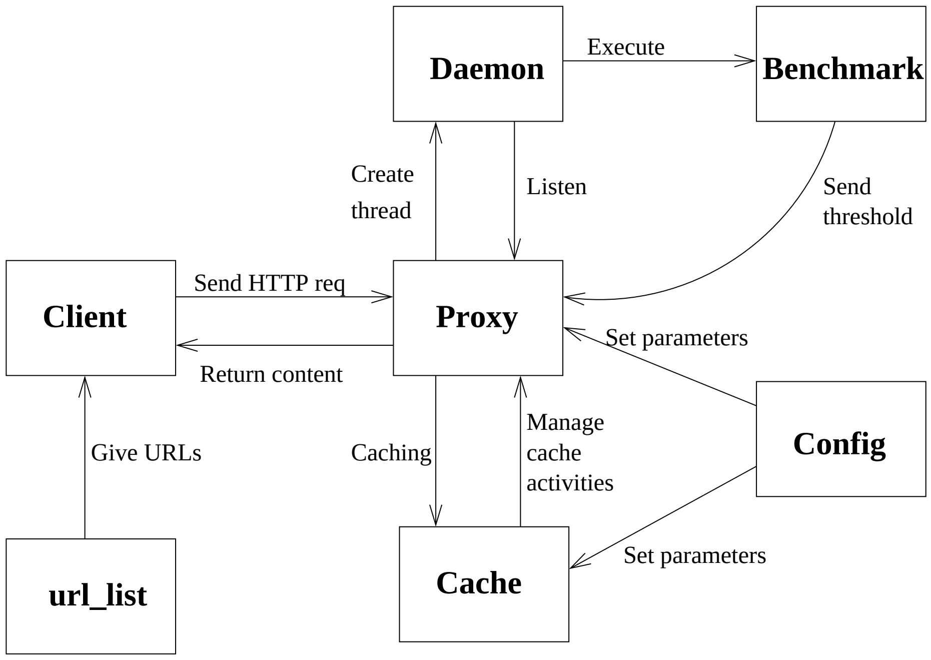

Figure 3.4: Simulation Systems

by cache and proxy.

Proxy creates thread to handle one client request. It gets the requested object from the web server or from the cache, and delivers the bits to the client.

Benchmark chooses a file randomly from the cache and sends a HEAD request to the web server to measure the latency. Calculates the threshold of total latency for caching and sends the value of the threshold to Proxy.

Daemon is the web daemon thread. This class creates main socket on port 8080 and listens on it until the user stops it. For each client request, it creates a proxy thread to handle the request. It records the time and starts to execute the benchmark at a certain interval time. In our programs, the interval time is set as 10 seconds.

Data includes the attributes of each file, such as size, last access time, total latency, replaceable flag set, and so on.

Client asks the user how long this client program will execute. Then the client keeps reading the urls from the file *urllist.in*, generates the corresponding HTTP requests and fetches the file from the proxy server.

RandomIntGenerator generates random integers.

url_list is a log of URLs with different size from 1 Megabyte to 6 Megabyte. It only includes three types of HTTP documents: html, gif and jpeg. The following is part of the file:

http://www.unt.edu

http://www.unt.edu/imageassets/americaunited.gif

http://www.unt.edu/imageassets/capitalcampaignsm.gif

http://www.unt.edu/imageassets/eventbitheader.gif

http://www.unt.edu/imageassets/finalindex_r10_c1.gif

http://www.unt.edu/imageassets/finalindex_r10_c1_f2.gif

http://www.unt.edu/imageassets/finalindex_r11_c1.gif

http://www.unt.edu/imageassets/finalindex_r11_c1_f2.gif

http://www.unt.edu/imageassets/finalindex_r12_c5.gif

http://www.unt.edu/imageassets/finalindex_r12_c6.gif

The *Client* will read from the url_list file and send the corresponding http requests to the *Proxy*. The detailed information of the url_list files will be listed in Chapter 4.

Because we will compare the BBPR algorithm with the commonly used LRU, two different cache classes have been implemented, one for BBPR and the other for LRU. Since only the proxy cache's performance based on these two page replacement algorithms is concerned, all other factors that may influence on the performance are assumed to be equivalent. Therefore, the two sets of simulation implemented with BBPR and the corresponding LRU algorithm are always executed *simultaneously.* Two aspects of these different algorithms are compared: the number of requests made in one hour and the hit ratio. The number of requests made in one hour shows how many requests can be made in one hour. A *counter* is used to track how many pages have been retrieved in the *Client* class. The average total latency of the urls in the file log can be derived from the counter.

The *Client* will prompt the user to input the time that the program is to execute. The program will begin to execute and print out the number of pages that has been retrieved. Upon completion, the program will print out the total requests. The following is how the client side looks like:

```
whe@csp07: java Client
Please input the time this program will run:
Please input the days: 0
Please input the hours: 1
Please input the minutes: 0
Please input the seconds: 0
This program will run 0 days 1 hours 0 minutes 0 seconds.
Run Time = 3600 seconds
```

```
Finish page 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800
3000 3200 3400 3600 3800 4000 4200 4400 4600 4800 5000 5200 5400 5600 5800
6000 6200 6400 6600 6800 7000 7200 7400 7600 7800 8000 8200 8400 8600 8800
9000 9200 9400 9600 9800 10000 10200 10400 10600 10800 11000 11200 11400 11600
11800 12000 12200 12400 12600 12800 13000 13200 13400 13600 13800 14000 14200
14400
Totally, 14492 pages have been visited.
```

From the output, it is evident that in one hour totally 14492 requests have been completed.

The urls in "*M.in" are all different and the *Client* class selects the url from the file randomly, so the probability of selecting each page is theoretically same. But in the real world, the popular websites have more visit rate and so do the near located websites. To account for this phenomenon, the urls with the same host have been copied in order to let them have higher visit rate in the simulation. For example, in "1M.in" if only the urls beginning with "www.unt.edu" have been copied 3 times, these copied *unt*'s websites will have 3 times visit rate than other websites in "1M.in". The total size of this url_list file is still same, i.e. 1M, but theoretically the UNT's url would be visited more than the rest urls.

CHAPTER 4

Result Analysis

This section summarizes the analysis in three parts. In each part we compare the performance of our method with that of LRU method under different conditions. First, for different url files with the size ranging from 1 Megabytes to 6 Megabytes, we compare the number of requests made in one hour and the corresponding hit ratio. Second, we derive the average total latency and make the comparison with two algorithms. Third, we give a detailed investigation of BBPR, which includes the threshold setting, what kind of pages will benefit more from BBPR, and the overhead introduced in BBPR.

4.1 Total Requests vs. Hit Ratio

Table 4.1 summarizes the information about the total size of all urls and the number of urls in different url files. From the table we identify the total size of all pages as

Name of URL File	Size of File(KB)	Number of URLs
1M.in	962	215
2M.in	1942	412
3M.in	2990	702
4M.in	3998	970
5M.in	4978	1167
6M.in	5976	1403

Table 4.1: URL Files

well as the number of pages in each url file. For example, in the url file “1M.in”, there are totally 215 different urls listed. If all the pages are to be cached, the size of the cache must be at least 962KB.

Six different url files were tested. The size of these six url files changes from 1 Megabytes to 6 Megabytes. Each simulation experiment that implemented the BBPR method and its corresponding counterpart LRU always execute simultaneously. Different cache sizes were used, ranging from 10% percent to 50% percent of the total size of all urls in the url files. The requests made over one hour period are compared by using the BBPR method and the LRU method.

Figure 4.1 and Figure 4.2 show the results with a set of five cache sizes at 100 KB, 200KB, 300KB, 400KB, 500KB, which is 10%, 20%, 30%, 40% and 50% of the total size of url_list, respectively. Figure 4.1 depicts the comparison of pages visited

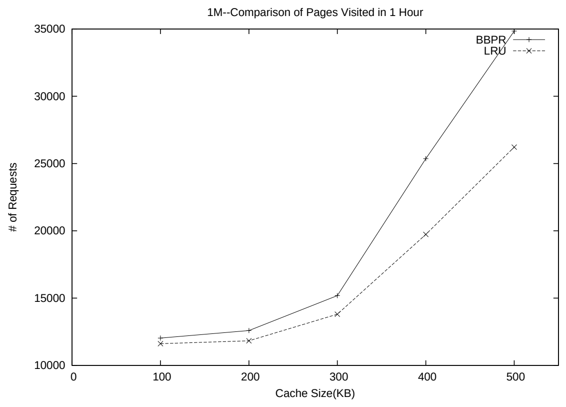

Figure 4.1: 1M–Pages Visited in 1 Hour

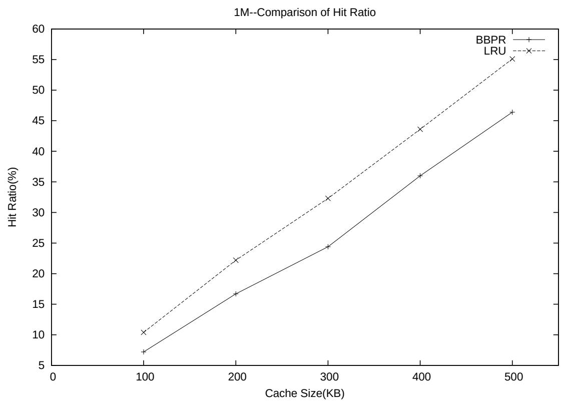

Figure 4.2: 1M–Hit Ratio

in 1 hour for the url file “1M.in”. This figure shows that more requests were made by BBPR as compared to LRU for each cache size. The improvement of BBPR is most significant when the cache size is 500KB, which is half of the total size of url_list file. The reason for large improvement with 500KB cache size is that there are more pages with replaceable flag set in the cache and hence the time for iterating through the entire file list reduces more than the smaller cache size. Because more requests were made in the same time by BBPR as compared to LRU, we conclude that the average total latency has improved by using BBPR. Figure 4.2 is the comparison of the hit ratio for the same url file and it shows that the hit ratio is always smaller by BBPR than by LRU. The reason for this is that with BBPR the pages with replaceable flag set are removed randomly, which causes those pages with small total latency that have been visited recently are deleted from the cache.

Figure 4.3 and Figure 4.4 show the comparison of the number of requests made in one hour and hit ratio for url files "2M.in", respectively. Identical to the experiment

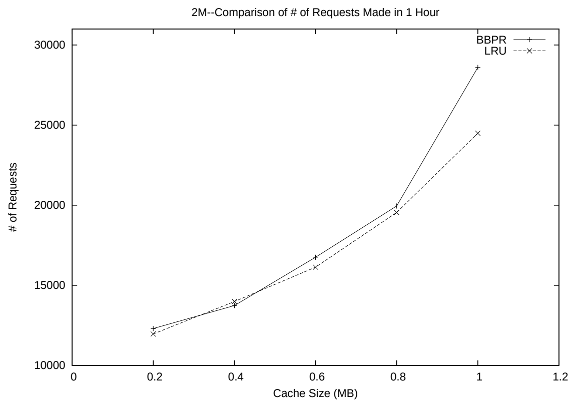

Figure 4.3: 2M–Pages Visited in 1 Hour

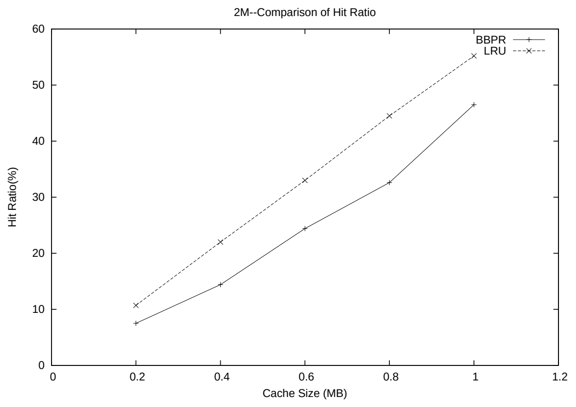

Figure 4.4: 2M–Hit Ratio

with "1M.in", the number of requests completed in one hour and the hit ratio were tested with both BBPR and LRU when the cache size is from 10% to 50% of the total size of all files. From the figure, there are more pages fetched by LRU than by BBPR only for the cache size 400KB. For other cache sizes, BBPR completed more requests than LRU. When the cache size reaches 1MB, i.e., 50% of the total size of all files, the difference becomes most apparent for the same reason mentioned for "1M.in". On the other hand, as discussed above, the LRU method also achieves higher hit ratio.

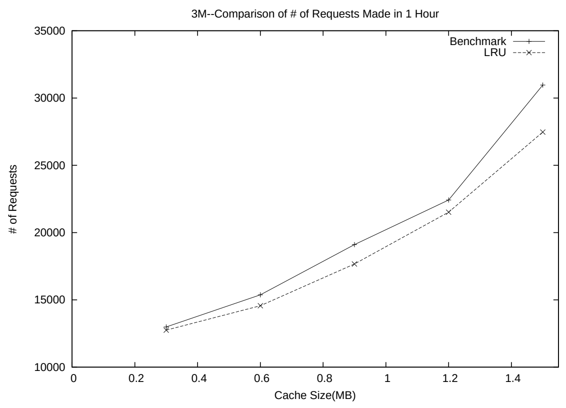

Figure 4.5: 3M–Pages Visited in 1 Hour

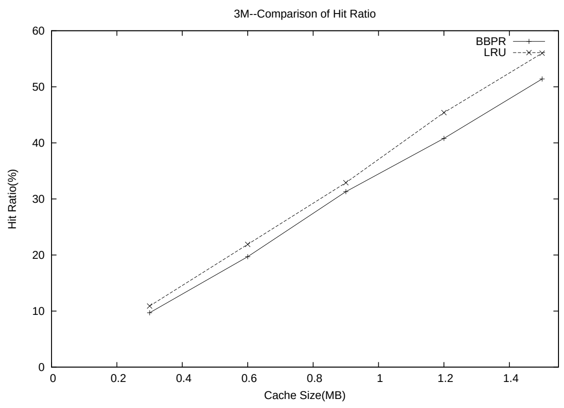

Figure 4.6: 3M–Hit Ratio

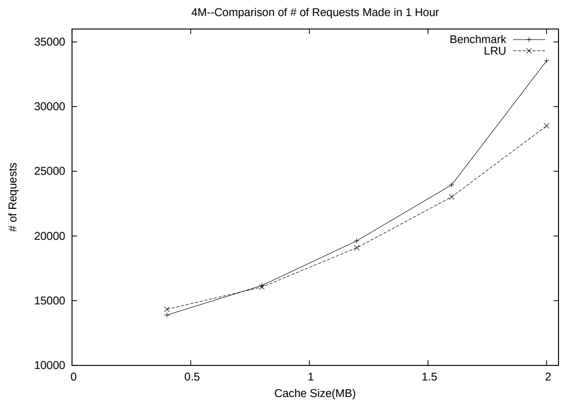

Figure 4.7: 4M–Pages Visited in 1 Hour

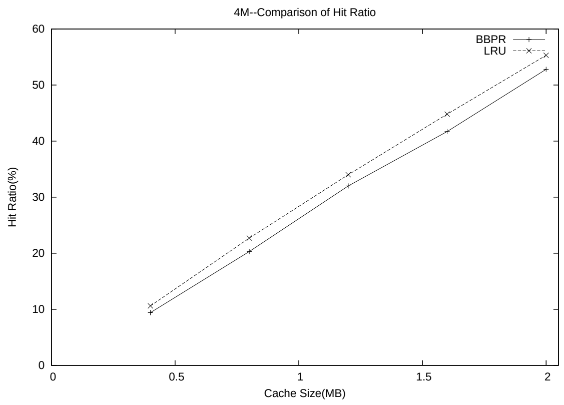

Figure 4.8: 4M–Hit Ratio

Figure 4.5 to Figure 4.10 are the results for the url_list "3M.in", "4M.in" and

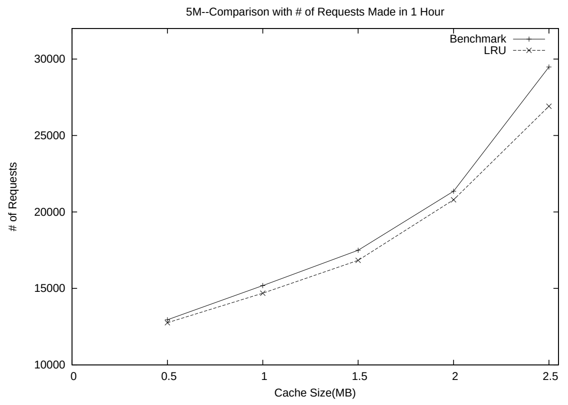

Figure 4.9: 5M–Pages Visited in 1 Hour

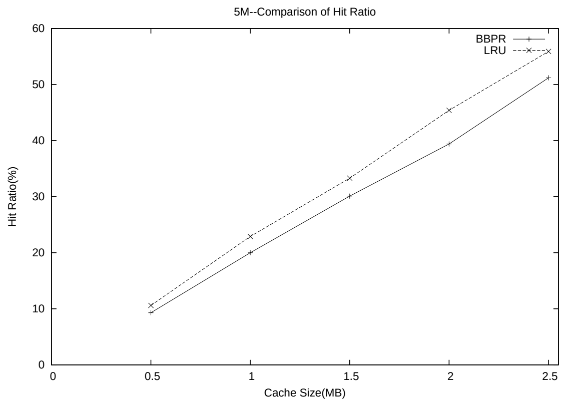

Figure 4.10: 5M–Hit Ratio

“5M.in”. From these results, the BBPR method almost always completed more requests than LRU. Only at 0.8MB cache size for “4M.in”, LRU outperformed BBPR. As far as the hit ratio is concerned, LRU always results higher hit ratio as previously discussed.

In summary, from the above results, more requests were completed by BBPR than by LRU. It shows that using BBPR will achieve lower average total latency than LRU. However, the LRU method results in a higher hit ratio. Because of the lower hit ratio the proxy server using BBPR will send more http requests to the remote servers as compared to LRU. That indicates there will be more network load by using BBPR. However, the clients will benefit from lower retrieval time.

4.2 Average Total Latency

For “5M.in”, we also derived the distribution of the number of urls and their corresponding total latencies without using the cache just before executing the other simulations. The data is shown in Table 4.2.

Total Latency(ms)	Number of URLs(totally 1167)
0 to 100	163
100 to 200	540
200 to 300	197
300 to 400	86
400 to 500	71
above 500	110

Table 4.2: 5M–Distribution of Total Latency

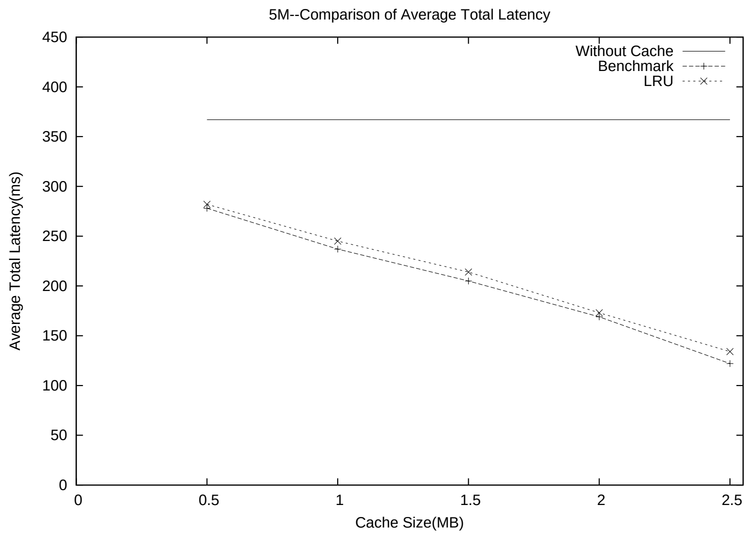

Figure 4.11: 5M–Comparison of Average Total Latency

Without cache the average total latency for the total 1167 different pages is 367ms. The average total latency with our BBPR strategy and with LRU is calculated as this: $atl = 3600/np$, where atl is the average total latency, 3600 seconds is the time for simulation, np is the number of pages visited in one hour. Figure 4.11 shows the average total latency in each case respectively. This figure shows the average total latency in BBPR is lower than that in the LRU method. The conclusion made here is the same as that previous discussion.

4.3 Detailed Investigation for BBPR

In this section, a more detailed investigation of the BBPR strategy will be described. The url_list file "6M.in" is taken as an example. The investigation includes whether longer latency threshold setting will give a better result, the benefit os slow retrieval pages using BBPR, and the overhead introduced by BBPR.

4.3.1 Distribution of Total Latency

There are totally 1403 different urls in "6M.in". The total size of these urls is 5976KB. Table 4.3 shows the distribution of the retrieval latency in this url file log. The laten-

Total Latency(ms)	Number of URLs(totally 1403)
0 to 100	142
100 to 200	725
200 to 300	272
300 to 400	118
400 to 500	42
above 500	104

Table 4.3: 6M–Distribution of Total Latency

cies for all urls were obtained without using cache just before the actual experiments have been conducted. The average retrieval latency is 333ms.

4.3.2 Incrementing the Latency Threshold

As described above, LRU is used as a backup replacement algorithm in situations when the replacement has to occur and there is no pages in the cache with replaceable flag set. At this point, it has to iterate through the entire file list to find the least recently used page(s) to remove. Since iterating through the entire file list is time-consuming, one way to reduce the iteration is to increase the latency threshold value, which is used to set the replaceable flag. When the threshold is large, there will be more pages in the cache with replaceable flag set. To see the effect of using larger threshold, the url_list file "6M.in" is investigated. The latency threshold value is increased by using a coefficient , say α, where $\alpha \geq 1$. In the previous experiments, the 10 most recently were selected pages to calculate the threshold.

$$L_s = \sum_{i=0}^{9} L_r[i]$$

$$\mathit{threshold} = L_s/\mathit{10}$$

Now we let $L_r[i]$ times α, and we can get a new larger threshold. In the experiment, α is set to 1.25.

$$L_s = \sum_{i=0}^{9} \alpha * L_r[i]$$

$$\mathit{threshold} = L_s/\mathit{10}$$

We compared the number of pages visited in one hour, hit ratio, the number of pages replaced by BBPR and the number of pages replaced by the LRU method. We refer to the test with $\alpha > 1$, i.e., $\alpha = 1.25$, as *Case 1* and the test with the threshold calculated with the coefficient set 1, i.e., $\alpha = 1$, as *Case 2*.

Figure 4.12 shows the statistics of the number of pages replaced in 1 hour both in Case 1 and in Case 2 when the cache size is 0.6MB, which is 10 percent of the size of total files. When the cache size changes, the situation is similar to what happened

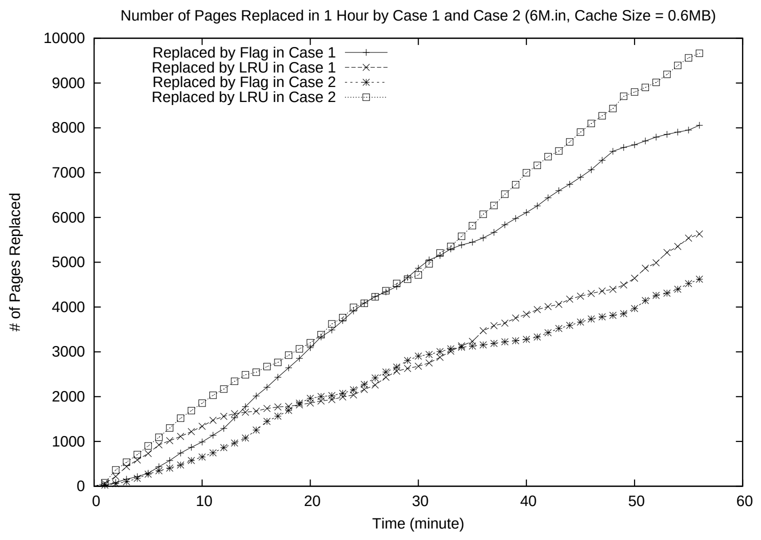

Figure 4.12: 6M–Number of Pages Replaced in 1 Hour by Case 1 and Case 2

in Figure 4.12. Table 4.4 shows the number of pages replaced by the replaceable flag and by LRU iteration when the cache size changes.

From Table 4.4, it is evident that in Case 1 the number of pages replaced due to the replaceable flag set is larger than the number of pages replaced by LRU, but in Case 2, the number of pages replaced by using the replaceable flag is smaller as compared to LRU. Because Case 1 has more pages removed due to the replaceable flag set, Case 1 can remove more pages by randomly finding replaceable documents

Cache Size (MB)	# of Pages Replaced by Flag in Case 1	# of Pages Replaced by LRU in Case 1	# of Pages Replaced by Flag in Case 2	# of Pages Replaced by LRU in Case 2
0.6	8057	5632	4621	9664
1.2	8640	5370	3908	9357
1.8	7381	6295	4464	8989
2.4	8280	5919	8808	5574
3.0	9229	5797	4860	8925

Table 4.4: Page Replacement in Case 1(α=1.25) and Case 2(α=1.0)

in the cache and hence avoid more iteration required by LRU than the iteration in Case 2.

Table 4.5 lists the number of requests made in 1 hour and the hit ratio both in Case 1 and in Case 2. Surprisingly, there is no significant difference of the number of

Cache Size (MB)	# of Requests Made in Case 1	# of Requests Made in Case 2	Hit Ratio in Case 1(%)	Hit Ratio in Case 2 (%)
0.6	11982	12606	8.1	10.1
1.2	13231	13299	16.3	20.6
1.8	15604	16222	27.8	31.4
2.4	18347	19593	38.8	38.8
3.0	26643	26474	51.3	54.0

Table 4.5: Comparison of Number of Requests Completed in 1 Hour and the Hit Ratio in Case 1(α=1.25) and Case 2 ($\alpha = 1.0$)

requests made in Case 1 with Case 2 from the data. The reason is that the hit ratio in Case 1 is lower than that in Case 2. A lower hit ratio indicates more pages have to be retrieved from the real server instead of the proxy cache. This lower hit ratio in Case 1 is a tradeoff of the time saving by less file list iteration.

4.3.3 Slow Pages Benefit from BBPR

The BBPR method is designed in favor of pages with slow retrieval latency. We conducted the following experiment to show that slow pages will benefit more than pages with small latencies from the BBPR strategy. We construct the url_list "6M_slow.in" as the following: 100 different urls whose total retrieval time is greater than 500ms have been selected from "6M.in" and copied in order to let the occurrence of those urls increase. In the url_list "6M_slow.in", these urls have been duplicated four times, which results in the frequency of those 100 pages being requested is *five* times larger than that of other pages. We also construct a url_list "6M_fast.in" for comparison. In "6M_fast.in", 100 different pages with the retrieval time lower than 100ms have been selected and duplicated four times.

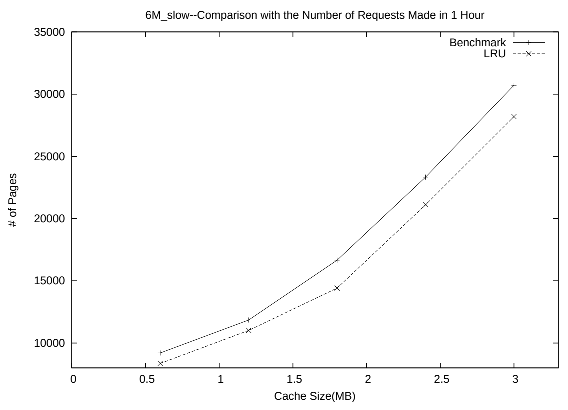

Figure 4.13: 6M_slow–Pages Visited in 1 Hour

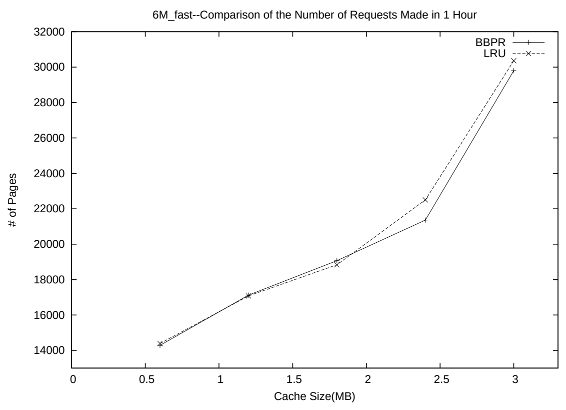

Figure 4.14: 6M_fast–Pages Visited in 1 Hour

Figure 4.13 and Figure 4.14 are the comparisons of requests made in one hour for each url_list. Figure 4.15 and Figure 4.16 depict the number of hit ratio for each

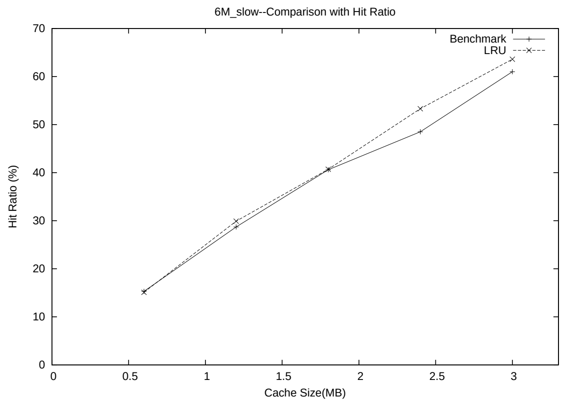

Figure 4.15: 6M_slow–Hit Ratio

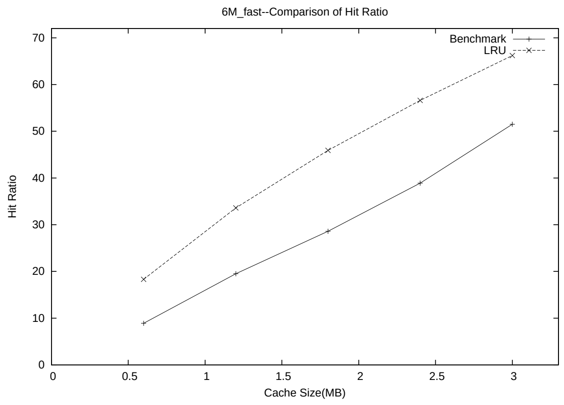

Figure 4.16: 6M_fast–Hit Ratio

case respectively. From the data shown in these graphs, we can see clearly that for "6M_slow.in" there is a significant improvement of the pages visited in 1 hour by BBPR. The hit ratio detected by BBPR is very close to that by LRU. Therefore, we conclude that slow pages will benefit more from BBPR. But there is no significant difference in the number of requests made for "6M_fast.in" by the two replacement methods, and the hit ratio calculated by BBPR is much lower than LRU. However, we may still conclude that the slow pages (pages with longer retrieval time) will benefit from the LRU method.

4.3.4 Overhead Analysis

The BBPR strategy will introduce some overhead when the benchmark executes. It includes the time of initializing the benchmark, randomly choosing a URL from the cache, fetching this url from the web server, and calculating the threshold. Figure 4.17 shows the cumulative time that the benchmark program spent by the end of each minutes with the cache size 3MB for “6M.in”.

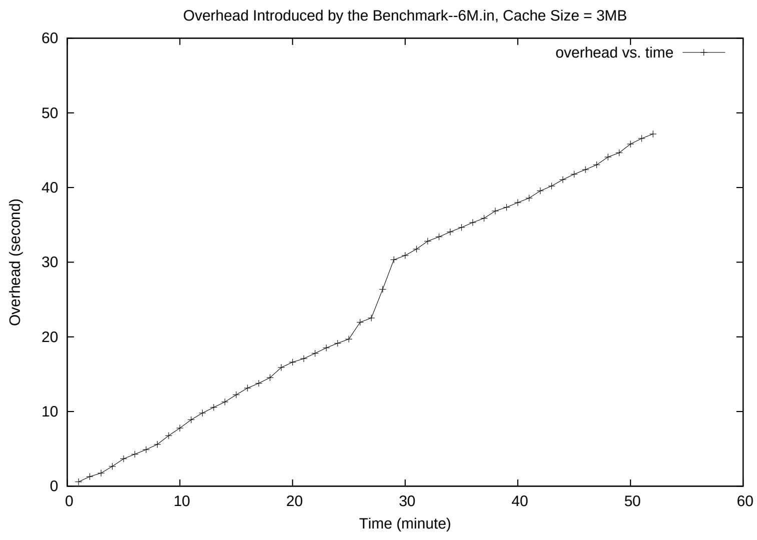

Figure 4.17: Cumulative Overhead of Executing the Benchmark(“6M.in”, Cache Size = 3MB)

The figure shows the cumulative overhead caused by the benchmark over 60 minutes is about 50 seconds. To reduce the overhead, we can consider to increase the time period that the benchmark executes. For example, the execution time of the benchmark will be reduced by increasing the interval period to execute the benchmark from 10 seconds to 20 seconds or 1 minute. However, by doing so, the threshold calculated by the benchmark will probably less precisely reflect the real situation of the network than choosing short benchmark executing period.

CHAPTER 5

Summary and Conclusions

5.1 Summary of BBPR

World Wide Web caching is widely used through today's Internet to reduce the network load, servers load and save time to fetch remote documents. The page replacement algorithm plays an important role in the proxy cache. Many page replacement strategies have been employed both in academic studies and in the application area. These algorithms can be classified into three categories based on how to execute the replacement as follows. First, traditional replacement policies and their extensions, such as FIFO, LRU and LFU, which were derived from the replacement polices used in operating system. Second, key-based replacement policies, which arrange the replacement on some key(s) such as size, lowest latency, etc. Third, function-based replacement policies, which are based on some function value(s). There is no agreement on which page replacement algorithm is the optimal one. The policies in the first two categories do not take the document retrieval time into account. Although some algorithms in the third category consider the document transfer time, they can not reflect the dynamic load change of the networks and the servers. Another drawback of the existing algorithm is that replacement is based on a particular value, such as size and latency, which is generally a maximum or minimum one among the entire set. To find this particular value is time consuming.

In this paper, to overcome the drawbacks of the existing page replacement algorithms, we introduced a new cache page replacement strategy, i.e., BBPR strategy. BBPR stands for Benchmark-Based Page Replacement strategy. In BBPR strategy, an HTTP benchmark is used as a small tool to evaluate the dynamic load of the Internet. The benchmark helps cache to decide which documents may be removed when the cache is full by deriving a document retrieval latency threshold. This latency threshold was designed to reflect the dynamic load of the Internet. We have partitioned the files in the cache into two parts by setting a replaceable flag thereby reducing the number of iterations through the entire cache file list. When there are no files with the replaceable flag set in the cache, the LRU method is used as a backup strategy. Because the dynamic detection of the network load and the server load and reduction of the file list iteration, BBPR will lower the average documents retrieval time.

We designed a client-server model to simulate the performance of BBPR. The simulation mainly focused on two aspects of the proxy cache performance compared with LRU which is the most commonly used cache document replacement strategy: the hit ratio and the documents' average retrieval time. To avoid the other factors such as time and different platforms that could have effect on the cache performance, the simulation implemented with BBPR and the corresponding LRU are always executed simultaneously and on the same terminal. We conducted the experiments with different URL files as well as different cache size. From the experimental results, BBPR effectively improved the average document retrieval latency as compared to

LRU. The improvement by BBPR becomes more significant with larger cache size. Slow request pages benefit more from BBPR than fast pages. However, the tradeoff of BBPR is a reduced hit ratio. We also investigate how the latency threshold setting influences the performance of the proxy cache and the overhead introduced by the benchmark execution.

From our simulation, the performance of proxy cache has been effectively improved by BBPR as compared with LRU. Therefore, we conclude that the BBPR strategy is a feasible way to be used in the proxy cache.

5.2 Limitations and Future Work

There are several limitations in the BBPR strategy.

Executing the benchmark program will introduce overhead. Our simulation programs are implemented in JAVA. One drawback of JAVA is that it does not execute as efficiently as C/C++. A C/C++ implementation is needed for the future work.

Our simulation only tests HTTP/1.0 requests. Since it does not check the documents' TTL (time-to-live) when the documents are cache available, some documents in the cache may be stale. The simulation is still simple. The client reads the urls from a url file log and sends them out. We only compared the BBPR strategy with one other existing strategy – LRU. We only simulated one client, one proxy model. Therefore, the experimental analysis needs to be expanded to include more clients.

The best simulation is that the implementation of BBPR could be associated with a real proxy cache as compared with other existing strategies, and then the comparison will be more precise and more persuasive.

The formula we used to set the latency threshold is empirical. More experiments should be conducted to optimize the latency threshold in order to give more precise feedback about the network and the server's load.

BIBLIOGRAPHY

[1] http://www.domainstats.com.

[2] http://www.etforecases.com/pr/pr201.htm.

[3] http://www.firstmonday.dk/issues/issue2_7/almeida/.

[4] Pei Cao, Sandy Irani, Cost-Aware WWW Proxy Caching Algorithms.

[5] C. Aggarwal, J.L. Wolf, and P. S. Yu, Caching on the World Wide Web, IEEE Transactions on Knowledge and data Engineering, Vol. 11, No.1, January/February 1999.

[6] Jia Wang, A Survey of Web Caching Schemes for the Internet .

[7] S.Williams, M.Abrams, C.R.Standbridge, G.Abdulla and E.A.Fox. Removal Polocies in Network Caches for World-Wide Web Documents. In *Proceedings of the ACM Sigcomm96*, August, 1996, Stanford University.

[8] N.Young. The k-server dual and loose competitiveness for paging. *Algorithmica*,June 1994, vol. 11,(no.6):525-41. Rewritten version of "Online caching as cache size varies", in The 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, 241-250, 1991.

[9] D. Sleator and R.E.Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202-208, 1985.

[10] R. Wooster and M.Abrams. Proxy Caching the Estimates Page Load Delays. In the *6th International World Wide Web Conference*, April 7-11, 1997, Santa Clara, CA. http://www6.nttlabs.com/HyperNews/get/PAPER250.html.

[11] P.Lorenzetti, L.Rizzo and L.Vicisano. Replacement Policies for a Proxy Cache. http://www.iet.unipi.it/luigi/research.html.

[12] E.Cohen, .Krishnamurthy, and J.Rexford, Evaluating server-assisted cache replacement in the Web, Proceedings of the European Symposium on Algorithm-98, 1998.

[13] Peter Scheuermann, Junho Shim, Radek Vingralek, A Case for Delay-Conscious Caching of Web Documents, http://citeseer.nj.nec.com/scheuermann97case.html.

[14] Jean-Chrysostome Bolot, Philipp Hoschka, Performance Engineering of the World Wide Web, http://www.w3journal.com/3/s3.bolot.html

[15] M.Abrams, C.R.Standbridge, G.Abdulla, S.Williams and E.A.Fox. Caching Proxies: Limitations and Potentials. WWW-4, Boston Conference, December, 1995.