LEAST RELATIVE BENEFIT ALGORITHM FOR CACHING
CONTINUOUS MEDIA DATA
AT THE WEB PROXY

by

HAOMING WU

A thesis submitted to the
Department of Computing and Information Science
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
June 2001

Copyright© Haoming Wu, 2001
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.
Abstract

Streaming multimedia data over the Internet has received tremendous attention. Transmission of real-time media typically has bandwidth, delay and loss requirements. Compared with traditional Web workloads, streaming media data can require significantly more storage and transmission bandwidth. Proxy caching is emerging as an important way to reduce user-perceived latency and network resource requirements in the Internet. The effectiveness of a proxy cache depends on the caching algorithm that is used. Existing techniques for caching text and images are not appropriate for multimedia streams.

In this thesis, we study caching strategies for multimedia data. In light of current web caching techniques, we propose a caching algorithm, called Least Relative Benefit (LRB), which integrates resource based caching, prefix caching, and a cost/benefit model in making decisions on cache insertion and replacement. Through trace-driven simulations, we show how LRB outperforms other algorithms in terms of the standard performance metric - byte hit ratio.
Acknowledgments

I would like to thank my supervisor, Dr. Pat Martin, for his support, advice, feedback, and above all, his patience. Without his guidance, this thesis could not have been finished. I would also like to thank my colleagues at the Database Research Lab for their help and friendship; CITO, Kingston Software Factory and Ontario Graduate Scholarship in Science and Technology (OGSST) for their financial support; Brian D. Davison, for making the Web traces and logs available to researchers. Finally, I would like to thank the Department of Computing and Information Science at Queen’s University for offering me such an invaluable opportunity to pursue my Master's degree.
# Contents

1 Introduction  
1.1 Goals of Research ................................................................. 5  
1.2 Outline of Thesis ................................................................. 6  

2 Background  
2.1 Web Caching Overview .......................................................... 7  
2.1.1 Web Caching Models ............................................................ 8  
2.1.2 Pros and Cons of Web Caching ............................................. 10  
2.2 Web Caching Algorithms ......................................................... 11  
2.2.1 Conventional Caching Algorithms ....................................... 11  
2.2.2 Continuous Media Caching Algorithms ............................... 13  
2.3 Summary .................................................................................. 30  

3 Least Relative Benefit Algorithm  
3.1 Algorithm Components ............................................................. 31  
3.2 Entity Characteristics ............................................................... 33  
3.2.1 Granularity ........................................................................... 33  
3.2.2 Resource Requirements ....................................................... 36  
3.2.3 Relative Value ...................................................................... 45
# CONTENTS

## 3.3 Entity Selection Algorithm ................................................................. 48

## 3.4 Replacement Algorithm ........................................................................ 50

## 3.5 Summary ............................................................................................... 55

## 4 Performance Evaluation of LRB Algorithm .................................................. 56

### 4.1 Simulation Assumptions ....................................................................... 56

### 4.2 Simulation Model ................................................................................. 57

#### 4.2.1 Structure ......................................................................................... 58

#### 4.2.2 Simulator ......................................................................................... 59

### 4.3 Experimental Design ............................................................................. 61

### 4.4 Parameter Settings ................................................................................ 63

### 4.5 Workload Parameters .......................................................................... 64

### 4.6 Varying Workload Access Skew .............................................................. 65

### 4.7 Performance Comparison ..................................................................... 70

#### 4.7.1 Experiment Set-1: Varying Cache Size ........................................... 70

#### 4.7.2 Experiment Set-2: Varying Cache Bandwidth .................................. 75

#### 4.7.3 Experiment Set-3: Varying Access Skew ......................................... 79

#### 4.7.4 Summary .......................................................................................... 81

## 5 Conclusions ............................................................................................... 85

### 5.1 Contributions ......................................................................................... 86

### 5.2 Future Work .......................................................................................... 88
List of Tables

3.1 Resource Allocation of Different CM Entities ........................................ 36
4.1 Data Structure of a Cached Entity ......................................................... 60
4.2 Access Queue Data Structure ............................................................... 60
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Interval Caching for Continuous Media</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>RBC Interval Caching</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Definition of Run Caching</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>RBC Run Caching</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>Components of LRB Algorithm</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>RBC Interval Caching</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>LRB-P Interval Caching</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>LRB-P Run Caching</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>Bandwidth Allocation of a Dynamic Entity</td>
<td>39</td>
</tr>
<tr>
<td>3.6</td>
<td>Partition of Disk Bandwidth</td>
<td>43</td>
</tr>
<tr>
<td>3.7</td>
<td>Partition of Used Bandwidth Pool in LRB Algorithm</td>
<td>45</td>
</tr>
<tr>
<td>3.8</td>
<td>Cache State</td>
<td>49</td>
</tr>
<tr>
<td>3.9</td>
<td>Cache Replacement Policy for Space Constrained Situation</td>
<td>51</td>
</tr>
<tr>
<td>3.10</td>
<td>Cache Replacement Policy for Space and Bandwidth Constrained Situation</td>
<td>52</td>
</tr>
<tr>
<td>4.1</td>
<td>Simulation Model</td>
<td>58</td>
</tr>
<tr>
<td>4.2</td>
<td>Access Pattern of Filtered NLANR Logs</td>
<td>66</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

4.3   Zipf-like Distribution ................................................................. 67
4.4   Generated Access Patterns ............................................................. 69
4.5   Comparisons of Caching Algorithms with Varying Cache Size ............ 73
4.6   Comparison of Average File Fraction of Cached Entities with .......... 74
      Varying Cache Size
4.7   Comparison of Hit Ratio with Varying Cache Size ............................ 74
4.8   Comparisons of Caching Algorithms with Varying Cache Bandwidth .... 77
4.9   Comparison of Average File Fraction of Cached Entities with .......... 78
      Varying Cache Bandwidth
4.10  Comparison of Hit Ratio with Varying Cache Bandwidth .................... 78
4.11  Performance of LRB-P with Varying Access Skew ............................ 80
Chapter 1

Introduction

The World Wide Web (WWW) employs a hierarchical data dissemination architecture in which hyper-media objects stored at a remote server are served to clients across the Internet. The dramatic growth of the World Wide Web has spurred the deployment of proxy caches, which are servers that store frequently-requested objects close to the clients in the hope of satisfying future client requests without contacting the original Web server. Highly localized request patterns that exhibit hot-spots, that is, frequent requests for a small number of popular objects, have made caching highly successful in reducing server load, network congestion, and client-perceived latency [1].

Digitization of video and audio yields a sequence of frames, or of samples, respectively. We refer to a sequence of continuously recorded video frames or audio samples as a stream. Media such as audio and video are referred to as "continuous" media (CM) because they consist of streams that convey meaning only when presented continuously in time. While most of the caching research to date has focused on caching textual and
image objects, Web-based streaming of CM data is becoming increasingly popular. A significant increase in the number of commercial products for playback of stored video and audio over the Internet, as well as a proliferation of server sites that support audio/video contents, has occurred over the past several years. In fact, it is expected that by 2003, CM data will account for more than 50% of the data available on Web servers [2].

There are two modes for transmission of CM data over the Internet, namely, the download mode and the streaming mode [3]. In the download mode, data is transferred completely to the client site before display. The large sizes of media files results in large space usage and long wait times at the client. In contrast, the media content is not downloaded in full in the streaming mode, but is played out while parts of the content are being received and decoded. This enables the client to initiate display of data with only small start-up latency and without waiting for the entire file to be downloaded. Due to the real-time nature of the media data, streaming typically has bandwidth, delay and loss requirements. However, current best-effort delivery on the Internet does not offer any quality of service guarantees to streaming media.

The design of effective caching policies for streaming media files is a challenging open research topic for several reasons. First, real-time multimedia, as the name implies, has timing constraints, that is, audio and video data must be played out continuously at a specified rate. To support the real time requirements for streaming, web caches need to
reserve cache bandwidth for each CM file. Thus, CM files stored in the cache occupy both space and bandwidth. Since disks have limited bandwidth, only a limited number of CM files can be guaranteed to have real-time delivery. Second, compared to traditional Internet applications such as accesses to text and image data, multimedia streams require high data rates and consume significant bandwidth over long continuous periods of time. This implies that effective caching algorithms should be deployed to avoid over-consumption of disk bandwidth to store new content in the cache during periods of high client interest in the current cache content. Third, the traffic generated by multimedia streams tends to be bursty and is highly sensitive to delay. Finally, streaming media files require significantly more storage than traditional Web objects, such as text and image data. A single file may require tens of megabytes to tens of gigabytes of storage, depending on the quality and duration of the video. Thus, cached content must be stored on disks. Disk caches differ from memory caches in that they are much larger in size but have much lower bandwidth.

Existing techniques for caching text and image resources are extensions of traditional memory caching algorithms. They are not appropriate for the rapidly growing number of CM streams in the Internet. The real-time requirements of multimedia streaming impose new requirements on caching algorithms and justify the development of new techniques.

Several continuous media caching algorithms have been proposed and evaluated in the research literature. Early papers proposed and analyzed algorithms for caching intervals
of video data in main memory [4,5,6,7] so as to satisfy multiple client requests that arrive close in time. Tewari et al. [8] define a disk-based caching policy called the resource-based caching (RBC) algorithm that considers disk bandwidth as well as disk storage capacity and that caches a mixture of intervals and full files. Simulation results show that RBC outperforms the recency and frequency-based algorithms. However, RBC fails to address the following important issues:

1. There are significant increases in user-perceived latency and network congestion for Internet applications. High latency and loss rates in the Internet make it difficult to stream audio and video without introducing a large playback delay at the client-end. Media quality as perceived by the end user is still very poor because support is lacking in the Internet to meet delay and jitter requirements for real time traffic. In particular, start-up latency, as well as frequency and length of interrupt of media streams increase significantly during periods of network congestion and media server overloads.

2. Improvement on network congestion and server overload can be achieved not only by caching hot objects that are most likely to be re-accessed, but also by storing in cache objects that are more costly (take longer) to download from the remote server. In other words, if we only assess the value of an object to the cache based on its probability of re-access, and evict those relatively “cold” objects, chances are we may still experience severe network congestion when the removed “pricey” objects accessed again.
The focus of our research is caching algorithms for streaming media. We claim that we can develop an algorithm that can overcome the two limitations of RBC. Given that the purpose of a cache is to reduce the file retrieval time as perceived by a user and to reduce the traffic generated by user requests, we propose a Least Relative Benefit (LRB) caching algorithm that integrates RBC, prefix caching and the cost of fetching an object in the approach. In addition to improving RBC, LRB also strives to hide network latency between the proxy and the server and maximize cache performance.

1.1 Goals of Research

The main goals of our research are:

1. To study different caching algorithms that handle CM data for disk-based caching at the Web proxy.

2. To propose an effective algorithm for caching CM data at a proxy cache that is limited by both size and disk bandwidth. The algorithm should focus on maximizing byte hit ratio by taking into account important performance factors, such as network latency and savings on disk bandwidth, and should improve upon the performance of previous algorithms.

3. Develop a trace-driven simulation to compare the performance of caching algorithms for CM data.
1.2 Outline of Thesis

The remainder of this thesis is organized as follows. In Chapter 2, we discuss the background material required for our work. In Chapter 3, we describe the rationale and implementation details of our proposed algorithm for caching CM data at the Web proxy. In Chapter 4, we present a performance comparison of our algorithm and other known algorithms. In Chapter 5, we conclude the thesis with a review of the contributions of our work and a discussion of future research directions.
Chapter 2

Background

In this chapter, we first introduce the concept of web caching. We then give an overview of previous work on caching algorithms that provide the basis for our work.

2.1 Web Caching Overview

The World Wide Web can be considered to be a large distributed information system that provides access to shared data objects. As the WWW continues its exponential growth, the Web today is characterized by a high volume of accesses to popular Web pages. Thus, identical copies of many documents pass through the same network links. This incurs several costs: network administrators see growing utilization that requires bandwidth upgrades, Web site administrators see growing server utilization that requires upgrading or replacing servers, and end users see longer latency for document requests to be satisfied [9].

Web caching has been recognized as an effective scheme to alleviate the service bottleneck and reduce the network traffic, thereby minimizing the user access latency. In
this section, we describe different types of web caching models, followed by a discussion about the advantages and disadvantages of the Web caching technique.

2.1.1 Web Caching Models

- **Client Caching**

  Caching can be performed by client applications and is integrated into virtually every Web browser. The cache aims to improve the response time for the client’s Web requests. The browser keeps a local copy of all recently displayed pages, and when the user returns to one of these pages, the local copy is reused. While caching at the client efficiently handles document re-accesses, it has no impact on the response time for newly accessed documents since browser caches are not shared among users.

- **Proxy Caching**

  To improve web server availability, caching can also be utilized between the client and the server as part of a proxy server where caches are managed on behalf of content providers. A proxy server is a shared network device, often located near network gateways, that can undertake Web transactions on behalf of clients, and that can store Web content. A client may benefit from faster access to a document previously placed in the cache for another client. A proxy server intercepts HTTP requests from clients and then delivers a locally-stored copy of the content if the requested object is found in its cache. This avoids a repeat of the download from the original content source and avoids the bandwidth required over expensive dedicated Internet connections. If the object is not
cached, the proxy server goes to the object’s originating server on behalf of the user, retrieves the object, possibly deposits it in its cache and finally returns the object to the user. In these cases, proxy servers not only improve the availability and fault-tolerance of the Web system, but also act as load balancers. For even better performance, proxy servers can be organized into cache hierarchies, in which a server can request documents from neighbouring caches instead of fetching them directly from the server [10].

In general, proxy caching results in wide-area bandwidth savings, improved response time, and increased availability of static Web-based data and objects. It is increasingly used to reduce bandwidth requirements and alleviate delays associated with the World Wide Web.

**Server Caching**

Finally, caches can be placed directly in front of a particular server, to reduce the number of requests the server must handle [11]. The purpose of server caching differs from that of browser or proxy caching. It aims to reduce the processing capacity required at a server node through (partial) replication of the server content over a number of nodes that are physically distinct in terms of both network and machine. Thus, the primary goal of server caching is not to improve the Web latency but to reduce the server node’s workload. Most proxy caches can be used in this fashion, but this form is often called *reverse caching* to reflect the fact that it caches objects for many clients but usually from only one server [10,12].
2.1.2 Advantages and Disadvantages of Web Caching

It has been shown that caching can significantly improve Web performance, such as user-perceived latency and response time [1]. In particular, Web caching offers the following advantages:

1. It reduces bandwidth consumption, decreases network traffic and lessens network congestion because fewer requests and responses need to go over the network.

2. It reduces access latency because:
   
   (a) frequently-accessed objects are fetched from nearby proxy caches instead of remote servers;

   (b) uncached objects can also be retrieved more quickly than without caching due to less congestion along the path and less workload at the server.

3. It alleviates the workload at the remote Web server by disseminating data among the proxy caches.

4. A client can obtain a copy of an object from a proxy server if the remote server is not available.

5. A group of caches cooperating to service clients’ requests is a powerful paradigm to improve cache effectiveness.

However, there are several disadvantages with Web caching:

1. The main disadvantage is that a client might be looking at stale data due to the lack of proper proxy updating.
2. The access latency may increase in the case of a cache miss due to the extra proxy processing. Hence, the cost of a cache miss should be minimized when designing a caching system.

3. A single proxy cache is always a bottleneck. A limit has to be set for the number of clients a proxy can serve.

4. Using a proxy cache reduces the hits on the original remote server, which means that information providers cannot maintain a true log of the hits to their Web pages.

2.2 Web Caching Algorithms

As in traditional memory caching, a crucial question in Web caching is what objects should be kept in the cache. In other words, when a new object must be brought into the cache and the cache is full, what object(s) should be removed to make room for the new one? The strategy used to make such decisions is referred to as the caching, or replacement, policy. Our work focuses on caching policies for proxy caches.

To review the work on caching algorithms, we first discuss some conventional algorithms used in the memory caching. We next describe previous work on caching algorithms for continuous media (CM) objects.

2.2.1 Conventional Caching Algorithms

Techniques for Web caching can be derived from traditional memory caching algorithms. To maximize the number of accesses served from the cache when the cache capacity is
exceeded, the caching algorithm seeks to replace pages to be accessed furthest in the future. To estimate the time of the next access to a page, these algorithms use heuristics based on the recency or frequency of access [8].

Recency-based algorithms exploit locality of reference. The most commonly used recency-based algorithm, Least Recently Used (LRU), discards the least recently used object among the objects present in the cache [13]. One of the main weaknesses of LRU is that the cache can be flooded by documents that are referenced only once, and flush out documents with higher probability of being reused. This situation is very likely to occur in Web caches, where references to objects accessed only once account for a large fraction of the total access [13].

Frequency-based algorithms are suited for skewed access patterns in which a large fraction of the accesses go to a disproportionately small set of hot objects [14]. Least Frequently Used (LFU), which maintains a reference count for each cached page, is the most commonly used frequency-based algorithm [15]. Nevertheless, LFU alone prevents "dead" documents with large reference counts from being purged. This causes the so-called "cache pollution" phenomenon.

Frequency and recency based algorithms form the two extremes of a spectrum of caching algorithms. Several algorithms, such as LRU-K and LRFU, which try to balance both recency and frequency, have been proposed [16,17]. In addition to these algorithms,
several other algorithms that utilize the knowledge of user access patterns have also been developed [15,18,19].

2.2.2 Continuous Media Caching Algorithms

Caching for CM data has been studied in the context of memory caching. Several CM caching algorithms [4,20,21,7,6,22] have been proposed and evaluated in prior work. These algorithms exploit the sequentiality of CM accesses and cache intervals of media streams formed by the portion of the stream between two consecutive user requests to the same object.

All of these algorithms assume that the system is limited by disk performance. However, in the context of the WWW, where a request involves retrieving objects from remote servers, the network becomes a bottleneck. Such an environment differs from memory caching in that: (1) cached objects have varying block sizes (instead of fixed size pages); (2) cached data can be shared among clients.

Current disk-based web caching algorithms are adaptations of traditional memory caching [23,24,9]. They are distinguished by how they choose what cached objects to evict. Examples of such algorithms include: (1) LRU [25]: evicts the object which was requested the least recently; (2) LFU [25]: evicts the object which is accessed least frequently; (3) Size [9]: evicts the largest object; (4) LRU-MIN [24]: a variant of LRU that uses the logarithm of object size as the primary key and the time-of-last-access as the
secondary key; (5) LRU-Threshold [24]: identical to LRU except that objects larger than a certain threshold size are never cached; (6) Log(Size) + LRU [24]: evicts the object that has the largest log(size) and is the least recently used object among all objects with the same log(size); (7) Hyper-G [9]: a refinement of LFU with last access time and size consideration; (8) PitKow/Recker [9]: removes the least-recently-used object, except if all objects are accessed today, in which case the largest one is removed; (9) Lowest-Latency-First [26]: tries to minimize average latency by removing the object with the lowest download latency first; and (10) Lowest Relative Value (LRV) [13]: associates a cost with each object and evicts objects with least cost/size ratio.

The above algorithms are adequate for text and static images. As streaming of CM data gains popularity, Web caches will need to reserve disk bandwidth (MB/sec) for each CM object. Since disks have limited bandwidth, memory caching algorithms for CM objects cannot be directly used for web caches. Given that disk caches are limited by both space and bandwidth, web cache management algorithms have to effectively manage both of these resources in order to optimize the cache performance.

Tewari et al. [8] define a disk-based caching policy for CM data called the resource-based caching (RBC) algorithm. It manages the heterogeneous requirements of multiple data types by considering disk bandwidth as well as disk storage capacity, and caches a mixture of intervals and full files that have the greatest caching gain. They showed, via simulation, that RBC outperforms other known caching algorithms.
Another approach to cache the CM data at the proxy servers caches the initial part of a CM file (the prefix) on local disk at the proxy to hide latency, network jitter and packet loss [27]. It also reduces the traffic between the server and the proxy, without having to store the entire stream.

In what follows, we discuss in detail the caching approaches most related to our work, namely interval caching, RBC, prefix caching and the LRV algorithm. The LRU-K algorithm is also discussed as it is used in the performance comparison.

- **Interval Caching Policy**

Due to the large size of CM data, caching the entire file is not always possible. Interval caching policies [4,5] have been proposed and analyzed for caching partial video streams in main memory. In these policies, if two client requests for the same file arrive close together in time, then the data delivered to the first client may be stored in the memory cache until it is delivered to the second client. Therefore, we only need to cache the part of the stream that is expected to be read by a reader sometime in the near future.

From the viewpoint of a cache, a client that requests data from it may be either a writer or a reader. If a client requests some data that is not already in the cache, the proxy will get it from the server, cache it and serve it to the client. Thus, the client acts as a writer for the cache because it is responsible for the data being brought into the cache. On the other hand, if the data requested by a client is already in the cache, the client acts as a reader of
the cached contents. In the interval caching policy, an interval is defined as a pair of consecutive writer and reader streams that access the same media file.

![Diagram of interval caching for continuous media](image)

**Figure 2.1 Interval Caching for Continuous Media**

The main idea behind the interval caching policy is illustrated in Figure 2.1. Consecutive pairs of playback streams \( (S_1, S_2) \), \( (S_2, S_3) \) and \( (S_3, S_4) \) form three intervals: \( b_1 \), \( b_2 \) and \( b_3 \). By caching the blocks brought in by the first stream of an interval, it is possible to serve the second stream from the cache. The cached part of the stream is expected to be read by a reader sometime in near future. The cache requirement of an interval is defined to be the minimum of the number of blocks needed to store the interval (equals to time interval * data delivery rate) and the number of blocks in the file. Therefore, the buffer requirement is different for different intervals. With a large number of intervals across all video streams, the interval caching policy caches the intervals that maximize the number of streams served from the cache. The algorithm orders all the intervals in terms of increasing cache requirements and caches as many of the shorter intervals as possible.

If a cached interval is only a segment of a CM file, it will be deleted from the cache when the client who comprises the reader stream finishes viewing the interval. The cache management process releases the space being held by the interval, returns the allocated
buffer space to the free pool and subsequently allocates the available buffer to the smallest unbuffered consecutive pair if the space satisfies the pair’s buffer requirement.

**RBC Caching Policy**

Most disk-based caching algorithms assume that the bottleneck of the system is disk space alone. They fail to consider that disk bandwidth is also an equally important bottleneck for serving CM streams. Based on the idea of interval caching, Tewari et al. [8] propose the RBC algorithm, a disk caching policy that characterizes each entity by its granularity, resource (space and bandwidth) requirements and caching gain, and tries to effectively utilize the limited cache resources, that is cache size and disk bandwidth. Although the algorithm was designed to handle heterogeneous files of both CM and non-CM types, we assume a workload of only CM files in the following discussion.

The RBC policy consists of two parts: (1) entity selection algorithm, and (2) cache replacement algorithm. When a client’s request for a file arrives and the file is not fully cached, RBC follows two steps. It first selects the entity to be cached using criteria that ensure both of the cache resources are effectively utilized. In step 2, it decides whether the selected entity should be cached, possibly by removing other cached entities.

(1) **Entity Selection Algorithm**

Each atomic unit of a CM object is called an entity. RBC associates each CM entity with a set of characteristics that comprise the building blocks of the algorithm. It is essential to
first understand each of these characteristics before looking into any details of the entity selection algorithm. The three characteristics that identify each entity are granularity, resource requirements, and caching gain.

**Granularity of cached entity**

Due to the sequential and periodic nature of access, a CM file can be cached partially or in its entirety. A request for a CM file can be served from the cache by maintaining a set of consecutive blocks forming an interval between two successive requests (a writer and a reader) for the same file, an idea similar to the interval caching policy. As shown in Figure 2.2, requests $S_1$, $S_2$, and $S_3$ arrive at times $t_1$, $t_2$, and $t_3$, respectively. The time interval between $S_1$ and $S_2$ is $T_1$, and $S_2$ is $T_2$ time units ahead of $S_3$. Since $S_1$ is the first request on the file, nothing has been cached for the file at time $t_1$ when $S_1$ arrives. Therefore, $S_1$ has to get the file from the remote server. If the selection algorithm decides not to cache the full file at $t_1$, the stream serving $S_1$ will retrieve the entire file from the remote server while bypassing the cache.

After the file has been streamed to $S_1$ up to time $t_2$, $S_2$ requests the same file. Now an interval can be formed between $S_1$ and $S_2$, and $S_1$ immediately becomes a writer that is responsible for placing the interval $(T_1, 2T_1)^*$ in the cache so that the reader $S_2$ can get this part of the file from the local cache rather than from the remote server.

---

*Interval Notation: $[T_1, T_2]$ means interval starts at time $T_1$ and ends at time $T_2$; $(T_1, T_2]$ means interval starts after time $T_1$ and ends at time $T_2$. 
Because the file has already been played back by $S_i$ for $T_i$ time units when $S_2$ arrives, $S_i$ starts caching the interval at $t_2$. As soon as it finishes caching, $S_i$ keeps getting the remaining part of the file from the remote server. $S_2$ gets the interval $[0, T_i]$ from the remote server before it can read the interval $[T_i, 2T_i]$ from the cache and it also gets the remainder of the file from the remote server.

![RBC Interval Caching Diagram](image)

**Figure 2.2 RBC Interval Caching**

Suppose, as $S_2$ is retrieving the remaining part of the file from the remote server, the request from $S_3$ arrives at time $t_3$. Now another interval between $S_2$ and $S_3$ for a duration of $T_2$ time units is created. $S_2$ becomes a writer and starts copying the newly formed interval $(T_2, 2T_2]$ into the cache. $S_3$ starts reading from the cached interval brought in by $S_2$ after having viewed the first $[0,T_2]$ part of the file.

In addition to caching intervals, RBC employs the notion of run caching in its approach. If several clients issue requests close together for the same CM file, a set of adjacent
intervals can be grouped together to form a run that may be stored in the cache until it has been delivered to all the clients in the run. Figure 2.3 shows that the runs \( r_1 \) to \( r_3 \), that are formed for a set of users accessing the same data object. The run \( r_1 \) is a simple interval of writer \( S_I \) and reader \( S_2 \), while run \( r_3 \) consists of writer \( S_I \) and readers \( S_2, S_3, S_4 \). Unlike an interval with a single reader and writer, a run consists of a single writer and multiple readers so that it not only amortizes the write overhead over multiple readers but also uses less space than the entire object. Therefore, run caching is useful when both space and bandwidth of the cache are equally limited.

![Figure 2.3 Definition of Run Caching](image)

A typical scenario in which run caching comes into play is shown in Figure 2.4. The first interval is formed between \( S_I \) and \( S_2 \) at time \( t_2 \), which allows \( S_I \) to become a writer for \( S_2 \) by copying the content of the interval into the cache. While the cache manager is having \( S_I \) fill the cache with the interval \((T_1, 2T_1]\), \( S_3 \) comes in at time \( t_3 \). Another interval between \( S_2 \) and \( S_3 \) is then created. Rather than having \( S_2 \) retain the interval for \( S_3 \), the cache manager will update the cache requirements of the contents it needs to cache by adding the newly formed interval. Because both \( S_2 \) and \( S_3 \) are readers of \( S_I \) in this case, they will be able to access the run \((T_1, 2T_1+T_2]\) from the cache.
In general, the RBC algorithm considers a range of possible entity granularities that can be cached, including intervals, runs, and full files. It further categorizes these granularities into dynamic or static entities. For dynamic entities, which include intervals and runs, the contents of an entity can change with time, while for static ones, which refer to full files, the contents remain the same. Interval and run caching can be used to serve some requests from the proxy. As a file becomes more popular, the entire file may be cached if necessary.

**Resource requirements**

The resource requirements of a CM entity are the cache space and bandwidth it occupies. While the space requirement seems to be straightforward, the bandwidth occupancy of an entity is the cumulative concurrent access bandwidth required by its readers and writers at the time the entity is formed. An interval has exactly one reader and one writer while a run, on the other hand, has multiple readers and one writer.
Caching gain

The two most common metrics for cache performance are Hit Ratio (HR) and Byte Hit Ratio (BHR). The HR denotes the ratio of the number of accesses from cache to the total number of accesses [8]. The BHR is the ratio of the total number of bytes accessed from the cache to the total number of bytes accessed [8]. BHR is normally used as the major performance metric, as it provides a direct measure of the savings in remote network and server bandwidth, and strongly influences the response time in serving clients’ request.

To maximize BHR, RBC defines the caching gain $g_i$ of a CM entity as the total bandwidth saved, that is, the bytes transferred per second, by caching this entity. For an entity with access bandwidth $b_i$ and an estimated number of concurrent readers $r_i$, the gain is given by $g_i = r_i \times b_i$.

In the RBC policy, the criteria used to select an entity for caching are designed to keep bandwidth and space utilization approximately equal. The selection methodology addresses three possible cache states described as follows:

1. If the bandwidth utilization of the cache is greater than the space utilization, the entity with the lowest write bandwidth overhead is selected.

2. If the space utilization of the cache is greater than the bandwidth utilization, select the entity with the lowest space usage.

3. If the bandwidth and space utilization are approximately equal, the entity with the largest effective bandwidth to space overhead ratio is selected.
(2) **Cache Replacement Algorithm**

Having selected the entity to be cached, RBC decides whether to cache the selected entity based on the current unallocated cache resources and on the entity’s resource requirements. The selected entity will only be cached if the required resources can be allocated to it. The requisite resources can be allocated either directly from currently available cache resources or by removing other cached entities.

When a new entity wants to enter the cache, the following cases arise:

(1) The cache is unconstrained with respect to the entity’s requirements, that is, both the available cache space and the available bandwidth exceed the requirements. In this case, RBC caches the entity directly and updates the state of the cache.

(2) The cache is constrained on space but not on bandwidth, that is, the available bandwidth is more than required but space is not. In this case, each entity is assigned a space goodness value, which represents the gain per unit space used, or the gain density. If an entity has a caching gain of $g_i$ and space usage of $s_i$, then the space goodness value is given by $G_{si} = g_i / s_i$. RBC sorts the cached entities in descending order based on their space goodness value, and selects candidates for eviction from the top of the space goodness list until enough space is released to accommodate the new entity.

(3) The cache is constrained on bandwidth but not on space, that is, the available space is more than required but bandwidth is not. In this case, each entity is associated with a bandwidth goodness value which is defined as the gain per unit cache
bandwidth used, that is $G_{bi} = g_i / b_i$. RBC orders the cached entities in descending order based on their bandwidth goodness value, and selects candidates for eviction from the top of the bandwidth goodness list till the available bandwidth is sufficient to serve the new entity.

(4) The cache is constrained on both space and bandwidth, that is, the available space as well as the bandwidth is not sufficient to accommodate the entity. In this case, RBC generates two lists that order the cached entities by both space goodness value and bandwidth goodness value. The selection for removal proceeds alternatively from either the space goodness list or the bandwidth goodness list till the cache becomes unconstrained.

The pseudo-code descriptions of the RBC replacement policies are provided in Appendix A.

- Prefix Caching Policy

Motivated by the observation that audio and video applications typically experience poor performance, due to the unpredictable delay, throughput, and loss properties of the Internet, prefix caching has been deployed at proxies along the path from the server to the client. Similar to traditional caching of text and image data, storing a fixed set of frames from the beginning of each popular continuous media stream enables the proxy to deliver good quality-of-service to the client, while hiding the weaker service model between the server and the proxy. Upon receiving the first client request for a particular stream, the proxy retrieves the entire stream from the server, and caches the prefix. For future
requests, the proxy can initiate transmission directly from the cache, while receiving the remainder of the stream from the server.

To ensure that the client request goes through the proxy, the IP address of the streaming service can map to the proxy, which in turn contacts the server directly for the rest of the stream. Alternatively, the service provider can ensure that a proxy resides on the path from the client to the server, allowing the proxy to intercept the client request. Meanwhile, the proxy must have an effective way to identify and store the sequence of initial frames, and request the remaining frames from the server. For example, Real-Time Protocol (RTP) [28] encapsulation of packets includes sequence number and timestamp information, enabling the proxy to identify the frames in the prefix and schedule their transmission to the client. When a client requests the video, the proxy must ask the server to initiate transmission of the remaining frames, instead of sending the entire stream from the beginning. If the continuous media stream is a Web resource, the proxy can invoke the byte range operation in HTTP 1.1 [29] to request the appropriate portion of the stream. Similarly, the Real-Time Streaming Protocol (RTSP) [30] supports absolute positioning to initiate transmission with an offset from the beginning of the stream. Either mechanism allows the proxy to retrieve the necessary frames without requiring changes to the underlying HTTP or RTSP protocols.

Prefix caching is especially beneficial to popular streams that are accessed by multiple clients. By storing all or at least a sizeable portion of multimedia streams, the proxy
prefix cache can significantly reduce the load on the server, the latency at the client, and the network load.

- **LRV Policy**

Explicitly intended for the World Wide Web, the Lowest Relative Value (LRV) replacement policy aims at maximizing an objective function that expresses quantitatively the value of the whole cache contents according to some metric [13]. The objective function is computed using a cost/benefit model that determines the relative value of each document in the cache, and then the value of the whole cache can be obtained by adding the individual document values. The LRV algorithm simply selects the document with the Lowest Relative Value as the most suitable candidate for the replacement and so maximizes the whole cache content value.

The objective function is based on a cost/benefit model. Purging a document from the cache has a benefit and a cost. The benefit \( B \) essentially comes from the amount of space freed, which is roughly proportional to the size of the document plus its metadata, possibly rounded to a multiple of the file system’s block size. The cost can be expressed as the cost, \( C \), of fetching the document \( d \) from the original site, multiplied by \( P(d) \), the probability that document is accessed again in the future.

Given the cost and benefit in purging a document, the relative value \( V \) of a document that determines how valuable the document is for the proxy is defined as
The relative value of each document is obtained based on information readily available to the proxy server. It has been demonstrated that LRV outperforms LRU and other policies, and can significantly improve the performance of the cache [13].

- **LRU-K Policy**

LRU-K is a variant of the Least Recently Used (LRU) algorithm that was proposed for managing buffer areas in database management systems. It incorporates both recency and frequency of accesses when making replacement decisions [17]. Since the LRU buffering algorithm drops the page from the buffer that has not been accessed for the longest time when a new buffer is needed, it limits itself to only the time of last reference. Specifically, LRU does not discriminate well between frequently and infrequently referenced pages until the system has wasted a lot of resources keeping infrequently referenced pages in buffer for an extended period. It was proven that LRU-K is essentially optimal among all replacement algorithms that are solely based on stochastic information about past references.

LRU-K is a self-reliant page-replacement algorithm that takes into account more of the access history for each page to better discriminate pages that should be kept in the cache. The LRU-2 version of the algorithm takes into account knowledge of the last two references to a page. The classical LRU algorithm is referred as LRU-1. For \( K > 2 \), the
LRU-K algorithm provides somewhat improved performance over LRU-2 for stable patterns of access, but is less responsive to changes in access patterns, an important consideration for some applications [17].

Given the limitations of LRU-1 information on each page, the best estimate for inter-arrival time is the time interval to the prior reference, and pages with the shortest such intervals are the ones kept in the buffer. The basic idea of LRU-K is to keep track of the times of the last K references to popular database pages, using this information to statistically estimate the inter-arrival time of such references on a page-by-page basis.

Assume we are given a set of disk pages, denoted by the set of positive integers $N = \{1, 2, \ldots, n\}$, and that the database system under study makes a succession of references to these pages specified by the reference string: $r_1, r_2, \ldots, r_n, \ldots$, where $r_i = p (p \in N)$ means that reference $r_i$ to disk page $p$. Clearly, each disk page $p$ has an expected reference inter-arrival time, that is, the time between successive occurrences of $p$ in the reference string. The system then attempts to keep in memory buffers only those pages that seem to have an inter-arrival time to justify their residence, that is, the pages with shortest access inter-arrival times, or equivalently greatest probability of reference.

The LRU-1 (classical LRU) algorithm can be thought of as taking such a statistical approach, keeping in memory only those pages that seem to have the shortest inter-arrival time. Given the limitations of LRU-1 information on each page, the best estimate for
inter-arrival time is the time interval to prior reference, and pages with the shortest such intervals are the ones kept in buffer.

The LRU-K algorithm works as follows. When a buffer slot is needed for a new page from disk, the algorithm specifies that the page p to be dropped is the one whose Backward K-distance, \( b_d(p, K) \), is the maximum of all pages in buffer. Given a reference string known up to time \( t, r_1, r_2, \ldots, r_r \), the Backward K-distance \( b_d(p, K) \) is defined as the distance backward to the \( K^{th} \) most recent reference to the page p:

\[
b_d(p, K) = \begin{cases} 
  x, & \text{if } r_t \text{ has the value } p \text{ and there have been exactly } K-1 \text{ other values } i \\
  \text{with } t - x < i \leq t, \text{ where } r_i = p. \\
  \infty, & \text{if } p \text{ does not appear at least } K \text{ times in } r_1, r_2, \ldots, r_r.
\end{cases}
\]

The only time the choice is ambiguous is when more than one page has \( b_d(p, K) = \infty \). In this case, a subsidiary policy, such as classic LRU, may be used to select a replacement victim among the pages with infinite Backward K-distance.

In the memory caching context, an implementation of LRU-K must address two issues peculiar to the cases where \( K \geq 2 \). The first, known as Early Page Replacement, arises in situations where a page recently read into memory buffer does not merit retention in the buffer by standard LRU-K criteria, for example because the page has a \( b_d(p, K) \) value of infinity. We clearly want to drop this page from the buffer relatively quickly, to save memory resources for more deserving disk pages. However we need to allow for the fact
that a page that is not generally popular may still experience a burst of correlated references shortly after being referenced for the first time.

A second feature that we need to deal with in cases where $K \geq 2$, is the fact that there is a need to retain a history of references for pages that are not currently present in the buffer. This is a departure from current page replacement algorithms, and is referred to as the Page Reference Retained Information Problem. A pseudo-code outline of the simulated LRU-2 algorithm used for caching CM data at the Web proxy is given in Appendix B.

2.3 Summary

Highly localized request patterns have made caching highly successful in reducing network congestion, user-perceived latency and server load. Web-based streaming of continuous media (CM) data is becoming increasingly popular. The design of effective caching policies for CM data is a challenging research topic. Existing techniques for caching text and image data are extensions of conventional memory caching algorithms, which are not appropriate for caching CM data on the disk. The real-time requirements of multimedia streaming impose new requirements on caching algorithms and justify the development of new techniques.
Chapter 3

Least Relative Benefit Algorithm

Our proposed Least Relative Benefit (LRB) algorithm, which is based on RBC, prefix caching, and the cost/benefit model deployed by the LRV algorithm, explicitly takes into account the network latency, the savings on network traffic, and the resource requirements of each entity. We modify the RBC policy to cache the prefix in order to hide latency, and to use a different metric to measure the relative significance of each entity. In this chapter, we present details of the LRB algorithm. We start with an overview of the composition of the algorithm, followed by the anatomy of each component.

3.1 Algorithm Components

The composition of the LRB algorithm is depicted in Figure 3.1. Like any caching policy, LRB can be viewed as having two phases. First, it decides which entity to place in the cache. Second, it determines which of the entities present in the cache should be replaced, if necessary, in order to accommodate a new entity. In the first phase, each candidate entity is associated with a set of unique characteristics, namely granularity, resource requirements, and relative value, in order to facilitate the selection. The decision is made
after going through a selection process that evaluates the cache state and the characteristics of each entity. Once a candidate is chosen, LRB further examines the possibility of placing it into the cache. The candidate can be accommodated either directly or by replacing some of the cached entities, depending on the available cache resources relative to the resource requirements of the candidate and the candidate’s relative value.

Figure 3.1 Components of LRB Algorithm
Chapter 3: Least Relative Benefit Algorithm

3.2 Entity Characteristics

Due to the large size of a CM file, it usually cannot be cached in its entirety. The streaming mode of delivery makes it possible to only cache part of a CM file. This introduces a range of possible granularities of a CM entity, from partial segments to the full file. Moreover, should a CM entity be granted cache residency, it occupies cache resources that depend on its granularity. If either of the available cache resources, that is, cache size and disk bandwidth, is insufficient to accommodate a new entity, a criteria that can impartially discriminate each entity’s relative value to the cache is needed to determine which cached entities should be evicted. Therefore, the set of characteristics associated with each CM entity includes its granularity, resource requirements, and relative value.

3.2.1 Granularity

LRB classifies CM entities into the same granularities as RBC, that is, interval and run as dynamic entities, and full file as static entity. However, we also consider the prefix as another type of static entity. Unlike RBC, which only considers caching the full file upon receiving the first request to a file, LRB considers caching either the prefix or the full file.

Adding prefix into the granularity spectrum is intended to address a problem inherited from the RBC policy. Recall that in RBC, an interval is formed based on interval caching as shown in Figure 3.2. As discussed in Section 2.2.2, RBC only considers caching the entire file if nothing has been cached for the file by the time $S_1$ arrives. Suppose the selection algorithm determines not to cache the whole file at time $t_1$ but later decides to cache the interval $(T, 2T]$ at time $t_2$. $S_2$ has to stream the initial part of the file, $[0, T]$, from the remote
server. By the same token, the cached intervals never have the beginning part of the file unless the full file is cached. This introduces high and/or variable communication delays because each subsequent request has to go to the remote server to get the initial part of the file before it can start hitting in the proxy cache. Therefore, in the absence of prefix caching, the application must either increase playback delay or experience degraded quality. By storing the prefix of each CM stream in the proxy cache, the network jitter, the packet loss and the latency between the proxy and the server became invisible to the clients, which allows the proxy to deliver good quality-of-service.

![Figure 3.2 RBC Interval Caching](image)

We implemented two versions of the LRB algorithm. In what follows, we refer to the version that considers prefix caching as LRB-P and the non-prefix version as LRB. Both versions contain the same improvements on the RBC policy. The LRB version is implemented to study the effectiveness of those improvements, whereas the LRB-P version emphasizes the influence of prefix caching on the cache performance.
In LRB-P, we modified the interval and run caching of RBC by incorporating prefix caching as shown in Figure 3.3 and 3.4. The prefix is $\delta$ time units long. Intervals and runs are cached in the same way as in RBC. However, once the prefix is cached, each following request will first get this initial part of the file from the cache. For illustration purposes, we assume that the prefix is cached on the first request to the file, and intervals and/or runs are chosen by the selection algorithm on each subsequent request.

![Figure 3.3 LRB-P Interval Caching](image1)

![Figure 3.4 LRB-P Run Caching](image2)
3.2.2 Resource Requirements

Each CM entity requires space and bandwidth reservations. The resource requirements of each entity are given by the cache space and disk bandwidth it occupies, and are computed based on the granularity of the entity. Let \( s_i \) and \( b_i \) denote the cache space and disk bandwidth occupied by entity \( i \), respectively. The computation of the resource requirements of entity \( i \) is given in Table 3.1:

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Static Entity</th>
<th>Dynamic Entity (interval / run)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Requirement</td>
<td>Prefix</td>
<td>Full File</td>
</tr>
<tr>
<td>Space occupancy ( s_i )</td>
<td>( \min { s_{prefix}, s_{file} } )</td>
<td>( s_{file} )</td>
</tr>
<tr>
<td>Bandwidth occupancy ( b_i )</td>
<td></td>
<td>( (r_i + w_i) \cdot b )</td>
</tr>
</tbody>
</table>

Parameters:
- \( s_{prefix} \): size of the prefix
- \( s_{file} \): size of the entire file
- \( r_i \): number of concurrent readers
- \( w_i \): number of concurrent writers
- \( b \): data delivery rate
- \( t_i \): duration of a dynamic entity

Table 3.1 Resource Allocation of Different CM Entities

The size of a dynamic entity can be easily calculated when an interval or a run is formed. However, sizing the prefix needs some explanation.

Through careful selection of the prefix size, the proxy can hide the latency and loss along the path from the server and absorb jitter and retransmissions without affecting the
operation at the client site. Recent Internet measurements of delay, loss, and resource sizes lend insight into how to size the prefix.

Round trip delays vary widely, depending on the end-points and the degree of congestion in the network, but delays of several seconds are not uncommon [31-33]. Even when large delays do not occur, the audio or video player must introduce delay to build up a large playout buffer, or risk playout disruptions during periods of high delay. By storing several seconds (say, 5 seconds) of a CM stream, the proxy can hide these latencies from the client.

Similarly, Internet packet loss rates are in the range 2-10% [31-33]. Full motion video transmission requires the transmission of 30-frames a second, that is, 1 frame (about 15-30 packets) every 33ms. Because the packets are sent out so close to each other, even a short duration of congestion on the transmission path in the network can potentially result in the loss of a sequence of consecutive packets in the video stream. By caching an extra round-trip time of data, the proxy can tolerate bursts of lost packets during periods of heavy congestion and hide the delay of a single retransmission from the server. Normally one or two seconds of prefix caching can handle the common case of the loss of a single packet or a short sequence of packets. In our implementation, we assign a prefix size for each CM file for a fixed duration of 8 seconds long to hide delays and to handle packet loss and retransmission.

The bandwidth occupancy of an entity is the cumulative access bandwidth required by its concurrent readers and writers. An interval has exactly one reader and one writer; a run, on
the other hand, has multiple readers and one writer. If the entity is a prefix or an entire file, it could have multiple concurrent readers and at most one writer if the entity is being concurrently being written into the cache.

For a dynamic entity, the number of readers $r_i$ is known at the time a run or an interval is formed and equals the current number of clients actively reading but not writing the entity. As shown in Figure 3.5, $S_i$ occupies one unit of disk bandwidth (equal to the delivery rate of the file) when it starts writing the interval $(T_i, 2T_i]$ into the cache at time $t_2$. Because of the request from $S_3$, the bandwidth used by $S_i$ cannot be released until $S_i$ finishes writing the run $(T_i, 2T_i+T_2]$. Meanwhile, $S_2$ starts hitting in the cache after $T_i$ time units and occupies another unit of disk bandwidth. The same thing happens to $S_3$ when it starts reading the cached contents at time $t_2+T_i$. However, the bandwidth occupied by $S_i$ can be released after time $t_2+T_i+T_3$, and be allocated to one of the streams that serve $S_2$ or $S_3$. Similarly, $S_2$ no longer utilizes disk bandwidth once it finishes receiving the contents from the cache, so that its bandwidth can also be reassigned to another stream.

For a static entity, $i$, on the other hand, $r_i$ is the estimated number of clients that will concurrently access the entity during its delivery time. The estimated number of concurrent readers can be computed from the interarrival time between requests (also referred to as the time-to-reaccess or TTR). Thus, if a static entity has access probability $p_i$, and $\lambda$ is the request arrival rate in the system, then the interarrival time between requests is $\frac{1}{\lambda p_i}$. Since the total time needed to deliver a static entity equals $s_i/b$, the estimated number of
concurrent readers is given by the ratio of the delivery time for $i$ to the interarrival time between requests for $i$. Hence, $r_i = \frac{s_i/b}{TTR} = \frac{s_i/b}{\lambda p_i s_i / b}$.

![Diagram of concurrent readers and delivery time](image)

**Figure 3.5 Bandwidth Allocation of a Dynamic Entity**

Observe that the number of concurrent readers depends on the estimation of the time-to-reaccess or the access probability of a static entity. In the simplest cases, we can predict access probability using reference counts (e.g. LFU) or future access frequencies based on previous access frequencies. The reference count measure, however, does not capture the recency of access. Moreover, in the more common case, access probability changes over time in unpredictable ways. In this case, precisely how to estimate current access probability is an open question.
Chapter 3: Least Relative Benefit Algorithm

One proposed approach is to use reference count with an aging mechanism [25]. Another proposed approach is to use the history of access times to estimate the interarrival time. The LRU technique uses only the time of the last access, while LRU-K uses the time of the $k^{th}$-to-the-last access. This technique also has drawbacks. It requires the history of the last $k$ accesses to be maintained on each file, which imposes significant overhead for large values of $k$. Also, even though the LRU-K technique has the merit of addressing both recency and frequency in its approach, it can be hard to select a tradeoff between the two concerns. Selecting a large value of $k$ to favour frequency of access ignores the recency of the previous $k$-1 accesses, while a small $k$ value favors recency of access ignoring the long-term frequency.

To overcome the drawbacks of the above techniques, RBC uses a third approach, called the WLRU-$n$ technique [8], which estimates the long-term access probability by the mean time-to-reaccess (MTTR) each time a new request arrives. The MTTR is computed using a weighted sum of the most recent inter-arrival time and the mean inter-arrival time computed when the previous request arrived. Our LRB algorithm also employs the WLRU-$n$ technique to compute the weighted sum of the inter-arrival times between the $i^{th}$ and the $i-1^{st}$ access.

Suppose we are given a sequence of access times for the last $n$ accesses to a particular CM object $(t_n, t_{n-1}, \ldots, t_{i+1}, t_i, \ldots, t_0)$ where $t_i$ is the time of the $i^{th}$-from-last access, $t_{i+1}$ is the time of the access prior to the $i^{th}$-from-last access, and $t_0$ is the current time. By definition, the weighted sum of the inter-arrival time between the $i^{th}$ and the $i+1^{st}$ access is computed as:
Chapter 3: Least Relative Benefit Algorithm

\[ MTTR = \sum_{i=0}^{t} (t_i - t_{i+1}) \cdot W(i) \]

where \( W(i) \) is a decreasing weight function such that \( W(i) < W(i+1) \). Let \( \alpha (0 \leq \alpha \leq 1) \) be the averaging factor that can be tuned to bias for or against recency, \( W(0) = 1 - \alpha \) and \( W(i) = \alpha W(i+1) \). We have \( W(i) \leq 1 \). The MTTR at time \( t_i \) therefore equals:

\[ MTTR(t_i) = (1 - \alpha)(t_i - t_{i+1}) + \alpha MTTR(t_{i+1}) \]

However, the above heuristics only apply to entities that get more than one access. We still need to know the access probability of a file upon its very first access in order to estimate its bandwidth requirement.

A Zipf-like distribution is used to accurately model Web access patterns and can be applied as an effective way to solve the problem. There have been many studies on page request distribution, that is, the relative frequency with which Web pages are requested [34]. As measurements and observations indicate, Zipf's law is a good approximation for Web accesses [35]. Zipf's law predicts that the relative probability of access for an object is a function of its popularity, that is, the \( i^{th} \) most popular object will be accessed with a probability proportional to \( \frac{1}{i} \) [34].

Breslau et al. [36] claim that the distribution of Web requests does not precisely follow Zipf's law, but instead follows a more general Zipf-like distribution. The relative probability of a request for the \( i^{th} \) most popular page is proportional to \( \frac{1}{i^\alpha} \), where the exponent varies from trace to trace and the concentration of Web accesses to "hot" documents depends on \( \alpha \). They examined Web cache replacement algorithms and showed
that algorithms using this Zipf-like distribution perform best on real trace data. Furthermore, their results indicate that, while page requests do indeed reveal short-term correlations, a simple model for an independent request stream following a Zipf-like distribution is sufficient to capture certain asymptotic properties observed at Web proxies. Their result was demonstrated by extensive simulation with a variety of real traces from proxies at academic institutions, corporations and ISPs, including the NLANR traces, which are used as the workload for our simulation.

Specifically, Breslau et al. define the Zipf-like distribution of Web accesses as follows. Consider a cache that receives a stream of requests for Web pages. Let $N$ be the total number of Web pages, and $P_N(i)$ be the conditional probability that, given the arrival of a page request, the arriving request is made for page $i$. Let all the pages be ranked in order of their popularity where page $i$ is the $i$th most popular page (the ordering is determined by measuring the request frequency for each page). Then $P_N(i)$, defined for $i = 1, 2, ..., N$ has a Zipf-like distribution given by:

$$P_N(i) = \frac{\Omega}{i^\alpha}$$

Where $\Omega = \left(\sum_{i=1}^{N} \frac{1}{i^\alpha}\right)^{-1}, 0 < \alpha \leq 1$

The total disk bandwidth is the total bandwidth capacity of a cache and is given as one of the cache configuration parameters that are explained in Chapter 4. We view the total disk bandwidth as partitioned into a free bandwidth pool and a used bandwidth pool, which is illustrated in Figure 3.6. The used bandwidth pool denotes the disk bandwidth allocated to
the cached entities as explained in Table 3.1, where the free bandwidth pool represents the unused disk bandwidth. Once an entity is placed into the cache, its bandwidth allocation is subtracted from the free bandwidth pool and added to the used bandwidth pool. The bandwidth allocated to a stream is returned to the free bandwidth pool once it finishes writing or reading the content.

![Diagram of disk bandwidth partition](image)

**Figure 3.6 Partition of Disk Bandwidth**

Because the bandwidth allocated to deliver a static entity depends on the estimated number of concurrent readers accessing the file, a portion of the allocated bandwidth may not be utilized to deliver the data due to discrepancies between the estimated number, and the actual number, of active requests to a file.

In the RBC policy, the bandwidth allocated to each full file is reserved exclusively for the file itself so that it can only be used to serve the requests to the particular file and cannot be assigned to other streams even if it is not fully utilized. When a new request to a cached file arrives, it first checks if a stream can be allocated from the file's pre-allocated bandwidth. If all of the file's pre-allocated bandwidth is already in use, RBC cannot take advantage of the under-utilized bandwidth allocated to other files in the cache. It allocates a new stream.
to serve the request only if the required bandwidth can be allocated from the free bandwidth pool, or if enough used disk bandwidth can be freed by removing some of the cached entities.

To minimize waste of the pre-allocated bandwidth of static entities, our LRB algorithm allows the cached static entities, including prefixes and full files, to share their unused bandwidth allocation. The bandwidth sharing among static entities is facilitated by subdividing the used bandwidth pool into a static bandwidth pool and a dynamic bandwidth pool, as shown in Figure 3.7. The bandwidth allocated to each static entity in the cache is combined together to form the static bandwidth pool, while that allocated to a dynamic entity is managed by the dynamic bandwidth pool and returned to the free bandwidth pool once a stream finishes receiving the cached contents.

LRB keeps track of the actual utilization of the static bandwidth pool so that any of the cached static entities can take advantage of the unused portion of the combined bandwidth. When a request on a cached static entity arrives, LRB checks for the available bandwidth in the static bandwidth pool. It serves the request as long as there is enough unused bandwidth in the static bandwidth pool, even if the pre-allocated bandwidth of the requested entity has been fully utilized. The bandwidth allocated to a stream from the static bandwidth pool is returned to the pool as soon as the stream finishes delivering the contents from the cache.
Streams serving different requests finish at different times. Once a stream finishes, its allocated bandwidth is reclaimed and can be used to serve other requests. We therefore need to keep track of the delivery status of each stream and update the bandwidth usage each time a new request arrives.

3.2.3 Relative Value

Improvement of network performance largely depends on the effectiveness of the caching algorithm that is used to determine the contents of the cache. The effectiveness of a caching algorithm, in turn, depends on the precision of the criteria that are used to assess how valuable an entity is to the cache.

Having surveyed many parameters that influence the cache performance, we believe that the caching gain used by the RBC policy is insufficient to capture the value of an entity in terms of alleviating network load from the server to the proxy. We propose a metric that takes into account, for every entity, important parameters such as its resource requirement,
its popularity or probability of re-access, and the cost of fetching it from the remote server, so that the caching policy is effective for maximizing BHR.

We assign a relative value to each entity using the idea of the cost/benefit model from the LRV algorithm where the cost and the benefit are perceived as the loss and the gain incurred by removing an entity from the cache, respectively. Since Web users rate access time to remote data as an essential component of quality, we modified the cost function in the LRV algorithm to incorporate the download latency and the caching gain of the entity. Specifically, we define the cost \((C)\) to be the time needed to get the entity from the original site multiplied by the caching gain of the entity.

As in the RBC policy, the caching gain denotes the total bandwidth saved by caching the entity. By taking into consideration the cost of retrieving the entity from the original server, our cost function explicitly measures the savings in terms of the total bytes transferred in the network. Since each entity occupies both space and bandwidth in the cache, the benefit \((B)\) is the space or bandwidth that can be reclaimed after eviction. Therefore, the relative value associated with each entity is the ratio of the total saving on bytes transferred in the network to the amount of resource occupied by the entity:

Relative Value: \(V = \frac{C}{B} = \frac{\text{lat}_i \cdot g_i}{B} \)

Relative Space Value (RSV): \(V_s = \frac{C}{s_i} = \frac{\text{lat}_i \cdot g_i}{s_i} \)

Relative Bandwidth Value (RBV): \(V_b = \frac{C}{b_i} = \frac{\text{lat}_i \cdot g_i}{b_i} \)
where \( \text{lat}_i \) equals to the time needed to fetch the entity from the original server.

Our notion of caching gain \( g_i \) is the same as in the RBC policy. Recall that for the objective of maximizing BHR, \( g_i = r_i \times b \) in RBC where \( r_i \) is the number of concurrent readers and \( b \) is the data delivery rate. For a dynamic entity, the number of concurrent readers is readily available. For a static entity, we estimate \( r_i \) so that:

\[
g_i = r_i \times b = \frac{s_i}{TTR} = \frac{s_i}{1/\lambda p_i}
\]

We omit the probability used in the original LRV function because, in the case of a dynamic entity (i.e. interval or run), clients who will be accessing the entity are known by the time the entity is created; Hence, estimating the probability of future access is out of the question. For a static entity, the caching gain is proportional to the inverse of the time-to-reaccess for the entity (\( 1/TTR \)). This is identical to the metric used by traditional space-constrained processor caches, which replace the data block that has the largest time-to-reaccess, thus it is redundant to predict the probability of reaccess separately.

In general, our algorithm assesses the worthiness of an entity based on the saving of network traffic per unit of utilized resource. Compared to RBC, it seems \textit{a priori} better to keep in the cache entities that are more costly to retrieve (\( \text{lat}_i \) is large) and/or can save more bandwidth (\( g_i \) is large) while occupying less cache resources.
3.3 Entity Selection Algorithm

Choosing the appropriate entities to cache is a key factor in a caching algorithm. To ensure that both the cache space and disk bandwidth are effectively utilized, the granularity of an entity needs to be selected upon receiving each request to a CM file until the entire file is placed in the cache. This selection depends on the state of the cache and the entity’s characteristics.

We first discuss our representation for the state of the cache. Let $E = \{e_1, e_2, ..., e_n\}$ denote the set of entities in the cache. At any given instant, the state of the cache is represented by its space utilization, $U_s$, and bandwidth utilization, $U_b$. Specifically,

$$U_s = \frac{1}{S} \sum_{i \in E} s_i, \quad U_b = \frac{1}{B} \sum_{i \in E} b_i, \quad 0 \leq U_s, U_b \leq 1$$

where $s_i$ and $b_i$ denote the space and bandwidth occupancies of cached entity $i$, respectively. $S$ and $B$, are the total space and bandwidth available to the cache, respectively.

Figure 3.8 shows the possible states of the cache. The points in region 2 capture the state where $U_s = U_b$, whereas points in region 1 and region 3 represent the state where $U_s < U_b$ and $U_s > U_b$, respectively. When a client request for a file arrives and the file is not fully cached, our algorithm selects the granularity of the entity so as to minimally use the resource that is currently over-utilized. The selection process attempts to ensure that utilization remains within region 2 of Figure 3.8. The selection addresses three possible cases and is described as follows. Recall that $r_i$ and $w_i$ are the number of concurrent readers and writers of object $i$, respectively.
Chapter 3: Least Relative Benefit Algorithm

(1) If the bandwidth utilization of the cache is greater than the space utilization (region 1), then choose the entity with the lowest write bandwidth overhead (i.e. $\frac{w_i}{r_i + w_i}$) to minimize the wasted bandwidth.

(2) If the space utilization of the cache is greater than bandwidth utilization (region 3), then select the entity with minimum space usage $s_i$ to minimize the space usage.

(3) If the bandwidth and space utilization of the cache are approximately equal (region 2), then both the space and wasted bandwidth overheads should be minimized so to choose the entity with the largest effective bandwidth to space overhead ratio. Since the effective bandwidth is the bandwidth that is available for reads, the ratio is given by $\frac{r_i/(r_i + w_i)}{s_i}$

The procedure invoked when a new request $S$ on a CM file is received by the proxy is as follows:

(1) If $S$ is the first request to the file, then estimate resource requirements for the prefix and the full file, update the cache state, and then select between the prefix and the full file.

(2) If $S$ is not the first request to the file but the entire file is in the cache, then do not make a selection.
(3) If $S$ is not the first request to the file and the prefix is cached, then if any of the previous requests are still writing data into the cache, for example, some request, a writer, is filling up the cache for an interval or a run, then we can either let $S$ grow the previous interval or run into a longer run, or we can cache the full file.

(4) If $S$ is not the first request to the file and neither the prefix nor the full file is cached, we choose among interval, run, prefix and full file.

To sum up, the selection process ensures that both the cache space and disk bandwidth are effectively utilized because it prevents either space or bandwidth to be saturated while the other remains under-utilized.

### 3.4 Replacement Algorithm

Once a new entity is selected in the first phase, LRB proceeds to the second phase in order to determine whether or not to cache the selected entity. The required resources of the entity can be acquired from the unallocated cache space and disk bandwidth. However, if either of the resource requirements exceeds the current available cache resource, then we need to decide whether to free up resources by removing some of the cached entities or simply to forward the request to the remote server. If replacement is necessary, we need to ensure that only the entities with the lowest relative values are removed each time so that the total value of the cached entities can be maximized.

To facilitate replacement, LRB first identifies all the cached entities whose relative values are smaller than that of the new entity, and then orders these entities in ascending order of
their relative values. The selection of victims for removal proceeds from the one with the lowest relative value until sufficient resources have been released to accommodate the new entity. Specifically, our replacement algorithm identifies victims for removal as follows:

(1) Unconstrained: add the entity to the cached queue and update the cache state.

(2) Space or Bandwidth Constrained: Identify all of the cached entities whose RSV or RBV is smaller than that of the new entity and sort these less valuable entities in terms of their RSVs or RBVs. Start from the least valuable entity in the sorted list, remove entities from the list one at a time until the combined released resource is no less than the required resource allocation of the new entity. The cache replacement algorithm in the space constrained case is outlined in detail in Figure 3.9. The bandwidth constrained case is similar.

**Space constrained (new entity $E_i$):**

\[
\forall \text{entities } E_k \text{ in cache} \\
\text{if } (RSV_k < RSV_i) \text{ slist } = \text{slist } + E_k \\
\text{if (slist is empty)} \quad \text{return failure} \\
\text{else} \\
\text{sort the slist according to each entity's RSV such that the entity with the smallest RSV is placed at the beginning of the list} \\
\text{j = index of the entity at the beginning of the slist} \\
\text{while } (j \geq 0) \\
\text{free_cache_space } = \text{free_cache_space } + s_j \\
\text{free_cache_bandwidth } = \text{free_cache_bandwidth } + b_j \\
\text{mark } E_j \text{ for deletion} \\
\text{if } (\text{free_cache_space } < s_i) \text{ decrement } j \\
\text{else} \\
\text{remove all the corresponding marked entities from the cache} \\
\text{empty slist} \\
\text{return success} \\
\text{end of else} \\
\text{end of while}
\]
empty slist
return failure

dend of else

Figure 3.9 Cache Replacement Policy for Space Constrained Situation

(3) Space and Bandwidth Constrained: In this case, entities whose RSV or RBV is smaller than that of the new entity are identified and are sorted into an s-list and a b-list according to their RSV and RBV values respectively. Starting from the least valuable entity, the algorithm removes entities from either of the two lists until enough resources are released for the new entity. In this procedure, the entities selected for removal are those that have the smallest relative value in either of the bandwidth or the space constrained case. The replacement algorithm for the dual constrained case is shown in Figure 3.10.

Space_bandwidth_constrained (new entity \( E_i \))

\( \forall \) entities \( E_k \) in cache
   \( \text{if} \ (RSV_k < RSV_i) \ slist = slist + E_k \)
   \( \text{if} \ (RBV_k < RBV_i) \ blist = blist + E_k \)

sort the slist and blist according to each entity's RSV and RBV respectively such that the entity with the smallest RSV or RBV is placed at the top of either list

\( m = \) the index of the top entity in the blist
\( n = \) the index of the top entity in the slist

while (space_and_bandwidth_constrained)
   \( \text{if} \ (m \geq 0 \ \&\& \ n \geq 0) \ // \) victims are identified in both the slist and the blist
      \( \text{if} \ (\text{free_cache_bandwidth} / b_i < \text{free_cache_space} / s_i) \ // \) remove from the top of the blist
        while (\( E_m \) has already been marked for deletion)
            decrement \( m \)
        end of while
        free_cache_bandwidth = free_cache_bandwidth + \( b_m \)
        free_cache_space = free_cache_space + \( s_m \)
mark $E_n$ for deletion on both list if $E_n$ also in the slist
decrement $m$
end of if
else //remove from the top of the slist
while ($E_n$ has already been marked for deletion)
decrement $n$
end of while
free_cache_space = free_cache_space + $s_n$
free_cache_bandwidth = free_cache_bandwidth + $b_n$
mark $E_n$ for deletion on both lists if $E_n$ also in the blist
decrement $n$
end of else
end of if
else if ($m > 0$ && $n \leq 0$) //victims are only identified in the blist, not in the slist
while ($E_n$ has already been marked for deletion)
decrement $m$
end of while
free_cache_bandwidth = free_cache_bandwidth + $b_m$
free_cache_space = free_cache_space + $s_m$
mark $E_m$ for deletion
decrement $m$
end of else if
else if ($n > 0$ && $m \leq 0$) //victims are only identified in the slist, not in the blist
while ($E_n$ has already been marked for deletion)
decrement $n$
end of while
free_cache_space = free_cache_space + $s_n$
free_cache_bandwidth = free_cache_bandwidth + $b_n$
mark $E_n$ for deletion
decrement $n$
end of else if
else
empty both slist and blist
return failure
end of while

if (space_constrained)
   if ($n > 0$) execute space_constrained()
   else
      empty both slist and blist
      return failure
end of if
else if (bandwidth_constrained)
    if (m > p) execute bandwidth_constrained()
    else
        empty both slist and blist
        return failure
end of else if

else
    remove all the corresponding marked entities in the cache
    empty both slist and blist
    return success
end of else

Figure 3.10 Cache Replacement Policy for Space and Bandwidth Constrained Situation

Our replacement policy makes use of the following property to maximize the total cache value by replacing entities with smaller relative values.

**Property of maximizing total cache value:** The total cache value is maximized if multiple cached entities that are ordered by relative value are replaced by a new entity.

**Proof:** Take RSV for example. Let \( E_m \) and \( E_n \) be the cached entities that are replaced by a new entity \( E_i \), and \( RSV_m < RSV_n \). First, replacing \( E_m \) and \( E_n \) with \( E_i \) means that

\[
RSV_m < RSV_n < RSV_i \tag{1}
\]

We calculate the combined relative space value (\( RSV_{mn} \)) of \( E_m \) and \( E_n \) as the ratio of the combined cost to total space occupied. Because \( RSV = \text{lat} \cdot g/s \), we have

\[
\frac{\text{lat}_m \cdot g_m}{s_m} < \frac{\text{lat}_n \cdot g_n}{s_n} < \frac{\text{lat}_i \cdot g_i}{s_i} \tag{2}
\]

\[
RSV_{mn} = \frac{\text{lat}_m \cdot g_m + \text{lat}_n \cdot g_n}{s_m + s_n} \tag{3}
\]

From (2), we can deduce
Chapter 3: Least Relative Benefit Algorithm

\[(lat_m \cdot g_m) \cdot s_i < (lat_i \cdot g_i) \cdot s_m\]  \hspace{1cm} (4)

\[(lat_n \cdot g_n) \cdot s_i < (lat_i \cdot g_i) \cdot s_n\]  \hspace{1cm} (5)

(4) + (5):

\[(lat_m \cdot g_m + lat_n \cdot g_n) \cdot s_i < (lat_i \cdot g_i) \cdot (s_m + s_n)\]  \hspace{1cm} (6)

Divide both sides of (6) by \(s_i \cdot (s_m + s_n)\):

\[\frac{lat_m \cdot g_m + lat_n \cdot g_n}{s_m + s_n} < \frac{lat_i \cdot g_i}{s_i}\]  \hspace{1cm} (7)

Therefore, we have \(RSV_{mn} < RSV_i\). Second, according to the replacement algorithm, the \(RSV\) of each entity that is remained in the cache should be greater than \(RSV_i\). By the same token, the combined relative space value of the remained entities is also greater than \(RSV_{mn}\). Combining the above two factors, we conclude that adding \(E_i\) to the cache maximizes the total cache value.

3.5 Summary

The LRB algorithm associates three characteristics, namely granularity, resource requirements and relative values, with each entity. Through a selection process, an entity of a specific granularity is identified. The algorithm, depending on the available cache resources relative to the resource requirements of the candidate and the candidate’s relative value, then either caches the selected entity directly or replaces some of the cached entities.
Chapter 4

Performance Evaluation of LRB Algorithm

In this chapter, we study the effectiveness of the LRB algorithm and compare its performance with other caching policies in the literature. We have developed a simulation model in Java. The model simulates each of the algorithms with multiple cache configurations and various workloads. Our evaluation uses BHR as the performance metric. Recall that BHR is defined as the ratio of the total number of bytes accessed from cache to the total number of bytes accessed. It provides a direct measure of the savings in remote network and server bandwidth, and directly influences the latency and response time in serving clients’ request. The higher the BHR, the lower the latency and the delivery time. In the context of streaming media caching, this ratio may be smaller than the fraction of delivered bytes that are found in the cache, due to the finite disk bandwidth.

4.1 Simulation Assumptions

The design of our simulation experiments is based on the following assumptions:
1. The space and bandwidth capacities assumed for the cache are not representative of a single disk but of a logical disk cache that could be mapped to a disk array.

2. The workload is restricted to CM files with varying size (from short audio samples to large video streams) and bandwidth requirements, not including discrete media data. In the case of handling heterogeneous data types, the disk can be partitioned into two parts, one of which handles CM data and the other part deals with discrete media data.

3. Each experiment starts with an empty cache and finishes after having processed the last request in the log file.

4. As in previous studies [37,38,39,6,8], we assume a fixed number of files; each access is independent, with fixed access probabilities that are skewed with a Zipf-like distribution with parameter $\theta(\theta = 1 - \alpha)$.

5. Both LRB and RBC use the WLRU-$n$ technique for estimating the time-to-reaccess for different files.

6. The LRU-2 algorithm only caches full files whose bandwidth allocation is computed using the WLRU-$n$ technique.

7. Video and audio files have uniform delivery rates.

8. Each client requests the entire file.

4.2 Simulation Model

To evaluate the performance of the LRB algorithm, we developed a simulator that models the behaviour of a proxy cache server and allows us to observe and measure the
performance of both versions of LRB and other algorithms under a variety of cache configurations. We compare LRB with RBC and LRU-2. We chose to compare with RBC because it is the only disk-based caching algorithm that was developed for caching CM data at the proxy and it provides the theoretical basis for LRB. The reason for comparing LRB with LRU-2 is that LRU-2 is considered one of the most effective caching algorithms in memory and disk caching environments. Experimental comparisons of LRB with other algorithms are left for future work.

4.2.1 Structure

The structure of our simulation model is depicted in Figure 4.1. The trace pattern generator generates logs of specific access patterns according to a pre-defined Zipf parameter \( \theta \) and each file's popularity rank. The simulator takes log files and simulates each algorithm under various cache configurations. The output of the simulator is the respective BHR of each algorithm.

Figure 4.1 Simulation Model
4.2.2 Simulator

The Java simulator contains all the logic to implement different algorithms. The main components of the simulator are as follows. We describe the details about the simulator in Appendix C.

(1) Cache queue: maintains a list of all the cached entities. Each entity in the queue is identified with its unique URL. The data structure of a cached entity is shown in Table 4.1. *Remote access duration* denotes the length of a stream that each of its subsequent requests must obtain from the remote server; *Cached duration* is the duration of the cached content and *Removal flag* is used to identify entities that are marked for deletion when executing the replacement algorithm. In the LRB and RBC policies, the bandwidth allocated to a CM entity changes with time, so we need to update the cache state each time before we make a selection. The *active streams vector* keeps track of the status of each stream writing or delivering the entity’s content so that freed bandwidth can be reclaimed on a timely basis. It maintains the starting times, finishing times and the access nature (write/read) for all streams accessing the entity. Once allowed to receive data from the cached contents, each stream calculates its own finishing time by adding up its starting time, the remote access duration and the cached duration. On receiving a new request to a cached entity, the cache state is updated by scanning through the *active streams vector* of each cached entity in the cache queue and releasing any bandwidth occupied by the finished streams. For dynamic entities, the freed bandwidth is returned to the free bandwidth pool, while for static ones, it is returned to the static bandwidth pool. Note
that an entry in the active streams vector will be deleted if the corresponding stream is found to have finished writing or receiving the cached contents.

*Granularity code: 0=prefix; 1=interval; 2=run; 3=full file

<table>
<thead>
<tr>
<th>URL</th>
<th>granularity code*</th>
<th>size requirement</th>
<th>bandwidth requirement</th>
<th>RSV</th>
<th>RBV</th>
<th>data delivery rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>most recent</td>
<td>current</td>
<td>MTTR</td>
<td>remote access duration</td>
<td>cached duration</td>
<td>active streams (vector)</td>
<td>removal flag</td>
</tr>
<tr>
<td>access TS</td>
<td>access TS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>access nature (R/W)</th>
<th>stream starting time</th>
<th>finishing time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 4.1 Data Structure of a Cached Entity

(2) Access queue: records the access history for each distinguished CM file. If an uncached file is accessed again, we run the selection algorithm to choose among different entities based on its previous access record. The data structure of the access queue is shown in Table 4.2.

<table>
<thead>
<tr>
<th>URL</th>
<th>Number of accesses</th>
<th>Most recent TS (sec)</th>
<th>Current TS (sec)</th>
<th>Total File delivery time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.nba.com/redlace.avi">http://www.nba.com/redlace.avi</a></td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>6000</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

Table 4.2 Access Queue Data Structure

(3) Process queue: if parts of the requested file (intervals or runs) are already cached upon receiving a new request, all the cached entities relevant to the file are copied into the process queue in order to select the granularity for the new entity. The process queue is emptied after processing a request.
(4) Space & bandwidth queue: used in the replacement process to temporarily store copies of entities whose RSV or RBV is smaller than those of the new entity. The entities in either queue are ordered based on their relative values and are removed after the replacement process terminates.

4.3 Experimental Design

In accordance with the goals of our research, we design three sets of experiments to assess the following:

1. How do the cache size and access skew affect the BHR of each algorithm, given a fixed disk bandwidth?

2. How do the disk bandwidth and access skew affect the BHR of each algorithm, given a fixed cache size?

3. How does the access skew affect the BHR of LRB-P? Although the effect of access skew on the BHR of each algorithm is addressed in the previous experiments, we would like to emphasize that effect on the performance of LRB-P in particular, because it is the complete version of our LRB algorithm.

Experiment set-1: For each access pattern with a pre-defined access skew given by the Zipf-like distribution, compare the BHR yielded by each algorithm, including both versions of LRB, with varying cache size (0 ~ 32GB) and fixed bandwidth capacity (8MB/sec). The purpose of these experiments is to study the impact of cache size and access skew on the algorithms' performance, assuming a fixed disk bandwidth. Note that
the results shown for this experiment set is for the bandwidth capacity of 8MB/sec. However, we also tested with other bandwidth configurations such as 16MB/sec, 32MB/sec and 64MB/sec. It turned out that results are quite similar.

*Experiment set-2:* For each access pattern with a pre-defined access skew given by the Zipf-like distribution, compare the BHR yielded by each algorithm, including both versions of LRB, with varying cache bandwidth capacity (0 ~ 64MB/sec) and fixed cache size (4GB). In this set of experiments, we investigate the effect of cache bandwidth and access skew on the algorithms' performance, assuming a fixed cache size. Note that the results shown for this experiment set is for the cache size of 4GB. However, we also tested with other size configurations such as 8GB, 16GB and 32GB. It turned out that results are quite similar.

*Experiment set-3:* Compare the BHR yielded by LRB-P by varying the access skew (Zipf parameter $\theta = 0; 0.15; 0.3; 0.5$) of the workload. In this set of experiments, we investigate the effect of access skew on the LRB-P performance in particular.

Our simulator allows us to vary the cache configuration so that proxy caches with different settings are simulated. Recall that one of the goals of our research is to develop an algorithm for caching CM data at the Web proxy that is limited both on size and bandwidth capacity (see Section 1.1). Moreover, there is an infinite number of different combinations of cache configuration; it is impractical to test on all the possibilities.
Therefore, we use the configuration suggested in the RBC paper [8]. However, as explained in each experiment set, we also experimented with some other settings for cache size and disk bandwidth. The results shown in this chapter are representative of those settings.

4.4 Parameter Settings

We explain the settings of some of the key parameters as follows:

- **Data delivery rate \( b \)**

One parameter that is used in the computation of the bandwidth requirement of an entity is the data delivery rate \( b \). CM streams convey meaning only when presented continuously in time, so service to the CM data must ensure that the playback of each media stream proceeds at its real-time rate. Specifically, during playback, the server must retrieve data from the disk at a rate that ensures that an output device (e.g., speaker, video display) consuming the data does not starve. MPEG-2 is an accepted standard for continuous media data storage/transmission [7]. The real-time data rate (the minimum disk transfer rate that guarantees real-time delivery) of video and audio data based on this standard are as follows (in compressed form): video stream \( = 0.42 \text{MBytes/sec} \), high quality audio sample \( = 16 \text{KB/sec} \) [7]. We use these rates as the delivery rates in the calculation of bandwidth requirements for video and audio entities in our simulation.
Remote access duration \textit{lat}

In the case of web caching, \textit{lat} includes the time needed to set up a connection, to process the request, and to transfer the entity from the original server. Since CM files are relatively large in size, we assume that the connection time is negligible. The time needed for the server to process the request can be obtained from the "elapsed_time" field of the NLANR trace (see Section 4.5). The time to download the entity from the original server equals \( \frac{s_i}{\text{con_bw}} \), where \text{con_bw} is the connection bandwidth between the server and the client. As typical cable modem is one of the most dominant means of Internet access, we use its nominal downstream connection bandwidth of 3Mbps for calculating the download latency. Varying bandwidths could be incorporated in the simulator for future experiments.

4.5 Workload Parameters

The workload for our simulation is derived from access logs for the proxy caches at the National Laboratory for Applied Network Research (NLANR). NLANR traces [40] are widely used in Web caching research because they are publicly available on the Web and up-to-date. A log entry in the NLANR traces contains information on a single request for a particular file, and is made up of the following fields:

1. timestamp: The time the client socket is closed. The format is "Unix time" (seconds since Jan.1, 1970) with millisecond resolution.
2. elapsed_time: The elapsed time of the request, in milliseconds. This is the time between the accept() and close() of the client socket. For persistent HTTP connections, this is the time between reading the first byte of the request, and writing the last byte of the reply, that is, the time needed by the server to process the request.

3. client_IP_address: IP address identifying the client.

4. log_tag/HTTP_code: The Log Tag describes how the request was treated locally (hit, miss, etc).

5. size: The number of bytes written to the client.

6. request_method: The HTTP request method.

7. URL: The requested URL.

8. hierarchy_data_and_hostname: A description of how and where the requested object was fetched.

9. content_type: The content-type field from the HTTP reply.

The NLANR access traces contain requests for text, image, video and audio data. We extracted all the video and audio entries from the traces according to the content_type field and excluded text and still images.

### 4.6 Varying Workload Access Skew

Clearly, not all Web documents are created equal. Some documents are extremely “hot” or popular and are accessed frequently and at short intervals by many clients at many sites. Other documents are accessed rarely, if at all. This non-uniform reference behaviour, which is referred to as access skew, can be extracted from a workload by
sorting the list of distinct files into decreasing order based on how many times they were accessed, and then plotting the cumulative frequency of requests versus the fraction of the total files referenced. The plot for our input trace file is shown in Figure 4.2.

![Cumulative Access Probability vs. Fraction of Files](image)

**Figure 4.2 Access Pattern of Filtered NLANR Logs**

Considering the observation that Internet accesses (object requests) follow a Zipf-like distribution, prior work has shown that a simple model of independent requests with a Zipf-like distribution of file popularity can be used to accurately model the access skews observed at Web proxies [36]. Figure 4.3 depicts the cumulative access probability of objects as a function of the fraction of objects accessed for a Zipf-like distribution with different values of $\theta$, where $\theta = 1 - \alpha$. It can be seen that, for the access skew with $\theta = 0$, about 70% of the accesses are restricted to 20% of the files, which is a 70-20 skew. As $\theta$ decreases, popular documents receive a greater fraction of requests, so a Zipf-like distribution yields high access skews at low values of $\theta$, and tends to become more uniform at larger values of $\theta$. 
Based on the original pattern of the filtered NLANR logs shown in Figure 4.2, we generated references to each distinct media file with a Zipf-like distribution of reference probabilities. Different patterns are generated by holding the number of accessed files constant and varying the Zipf parameter $\theta$. We counted the number of accesses for each distinct media file in the filtered logs, and ordered all of the files in terms of their popularity. We assumed that this popularity rank remained constant for each pattern we generated.

As we discussed previously in Section 3.2.2, given the set of $N$ objects used by a set of clients and the probability rank of each distinct object in the set. A Zipf-like distribution relates the access probability to an object to its popularity rank $i$. For each distinct media file in the list, we can compute its access probability given its popularity rank, the total number of accessed files, and the Zipf parameter $\theta$. We assume that the least popular file
gets one access. Since the number of accesses to the \(i^{th}\) most popular file \(n(i) = P(i) \times n\), where \(P(i)\) is the access probability of the file and \(n\) is the total number of accesses to all files, we can further deduce \(n\) from the access probability of the least popular file, and the number of accesses to each of the other files can be easily calculated.

Given the list of all of the distinct media files ranked in terms of their popularity order and the number of accesses to each file, we can proceed to generate the input workload to our simulation model. The input is implemented as a request queue, where each entry in the queue stores all the attributes that identify a particular request. Considering parameters used in each of the algorithms and all the data fields available in the original NLANR logs, we identify five attributes for each entry, namely timestamp, elapsed_time, size, URL and content_type. For each request entry, we randomly assign a position in the queue to represent when the request is issued relative to the rest of the requests. If the specified position is already occupied by another request, then another index within the same range will again be generated until a free place is found. The same process iterates for the number of accesses to the particular object. The size of the queue equals the total number of accesses.

Figure 4.4 depicts four pre-generated access patterns (for \(\theta = 0, 0.15, 0.3\) and 0.5 respectively) used as the input to our simulator. However, to ensure the quality of the randomization we generated for each pattern, a 90% confidence level, with 10% intervals, was used in the simulations (see Appendix D). According to the Student-t
Distribution Table [41], we generate five different access sequences for each pattern. The confidence intervals for all results presented in this chapter are within \(\pm 2\%\) of the reported value. For clarity of presentation, confidence intervals are not explicitly depicted on each of the performance plots for a specific cache configuration under a certain access pattern.

![Figure 4.4 Generated Access Patterns](image)

In Section 3.2.2, we discussed estimating request inter-arrival times for the calculation of caching gain for a full file. In a realistic Web caching system, we do not have \textit{a priori} knowledge of a file's popularity so we must instead deduce the relative popularity of each file by observing Web traffic. In the case of handling the very first request to each file for a specific access pattern, we compute the TTR value given the Zipf-like distribution rule, and estimate MTTR on each additional access by applying the WLRU-\(n\) technique.
4.7 Performance Comparison

To evaluate the performance of the LRB algorithm, we compare its BHR values with RBC and LRU-2 on different access patterns.

4.7.1 Experiment Set-1: Varying Cache Size

Figure 4.5 (a – d) shows the variation in the BHR for different caching algorithms with fixed disk bandwidth (8MB/sec) and varying cache size. We also experimented with other fixed disk bandwidth settings of 16MB/sec, 32MB/sec and 64MB/sec. The results are similar to what are shown for 8MB/sec. It can be seen that the LRB algorithm outperforms the RBC and the LRU-2 algorithms by an average of 3% and 20%, respectively. Figure 4.5 also demonstrates that, when the cache size keeps increasing, the cache bandwidth eventually saturates and becomes a constraint. The BHR reaches this threshold when the cache size is increased to 4GB and remains insensitive to further increases in cache size.

The higher BHRs for the LRB and LRB-P, compared with RBC, are attributable to the prefix caching and bandwidth sharing among static entities. As depicted in Figure 4.6, when cache size is small (less than 2GB), LRB, LRB-P and RBC fill the cache with more intervals or runs because of their relatively small average file fractions of the cached entities. However, LRB-P caches prefixes as well as intervals/runs, while LRB and RBC only cache intervals/runs. RBC and both versions of LRB try to amortize write overhead by caching more full files as cache size increases and cache bandwidth becomes a
constraint. Both versions of LRB achieve slightly better BHR performance than RBC because they allow static entities to share bandwidth allocation. LRB-P yields better BHR performance compare to LRB because of the extra caching of the prefix. As cache size increases, the difference becomes less obvious since both of them cache more full files.

We assume that the higher BHR yielded by LRB is due to the bandwidth sharing and that yielded by LRB-P is attributable to both bandwidth sharing and prefix caching. As explained in Section 3.2.2, if a request to a cached file arrives and the file’s pre-allocated bandwidth is all in use, RBC serves the request only if it can allocate a new stream from the free bandwidth pool, or, if enough entities with lower relative bandwidth value can be evicted. This can make many requests that find the requested file in the RBC cache but cannot actually read it from the cache due to limited disk bandwidth. LRB and LRB-P eliminate this phenomenon through bandwidth sharing, which maximizes the number of requests serviced from the cache. In order to justify our assumption, we measure the hit ratios (HRs) achieved by LRB, LRB-P and RBC for various cache sizes as shown in Figure 4.7. When cache size is smaller than 2GB, the higher HR yielded by LRB-P can be attributed to the cached prefixes that are not cached by LRB and RBC. When the cache is filled with more full files, LRB-P and LRB yield higher HR than RBC because of their bandwidth sharing capacity. Once the cache size is greater than 4GB, the majority of cached entities are full files, so the difference in HR performance between the LRBs and RBC becomes less significant since both versions of LRB can only take advantage of bandwidth sharing.
Compared to the LRU-2 policy, the LRB and the RBC algorithms owe their higher BHRs to interval/run caching. For a fixed cache bandwidth, when the cache size is small, both algorithms cache a large number of intervals and runs (the LRB algorithm may also cache prefixes) in addition to some of the popular files. The LRU-2 algorithm, on the other hand, caches a file whenever it is accessed with the possibility of evicting some of the cached popular files. The BHR performance of the LRU-2 algorithm is limited by the relatively small number of files that can be cached. As the cache size increases and the cache bandwidth becomes the bottleneck, both the LRB and the RBC algorithms are more likely to fill the cache with full files, so that the cache bandwidth can be used more efficiently by amortizing write overhead over multiple reads. Therefore, their advantage over LRU-2 decreases as cache size keeps increasing. LRU-2 may eventually perform as well as LRBs and RBC for a cache with a very large size. However, evaluation of caching algorithm under such large cache sizes and disk bandwidths are out of the scope of our research. We leave the performance evaluation under large cache configurations to future work.
Figure 4.5 Comparisons of Caching Algorithms with Varying Cache Size
(Disk Bandwidth = 8MB/sec)
Chapter 4: Performance Evaluation of LRB Algorithm

Figure 4.6 Comparison of Average File Fraction of Cached Entities with Varying Cache Size (Disk Bandwidth = 8MB/sec, $\theta = 0.15$)

Figure 4.7 Comparison of Hit Ratio with Varying Cache Size (Disk Bandwidth = 8MB/sec, $\theta = 0.15$)
4.7.2 Experiment Set-2: Varying Cache Bandwidth

The second set of the experiments compares the BHRs of the different caching algorithms when cache bandwidth is varied and cache size remains constant (4GB). We also experimented with other fixed cache size settings of 8GB, 16GB and 32GB. The results are similar to what are shown for 4GB. From the results in Figure 4.8 (a-d), we find that on average, the LRB-P algorithm yields 5% and 25% better BHRs than RBC and LRU-2, respectively. Similar to the case with fixed cache bandwidth and varying cache size, when the cache bandwidth keeps increasing as the cache space saturates and becomes the constraint, no further improvements in relative BHR performance are achieved.

The reason why the performance of the LRB algorithm is superior to RBC can be explained as follows. The average fraction of file cached by LRB, LRB-P and RBC is shown in Figure 4.9. Although all of them fill the cache with more full files when the cache bandwidth is low (less than 8MB/sec), LRB and LRB-P yield higher BHRs due to their bandwidth sharing ability. When the cache bandwidth increases, and until cache size saturates, all algorithms cache more intervals/runs in order to take advantage of the excessive bandwidth. LRB-P may, however, also cache prefixes in this case. The performance of LRB is almost equivalent to that of the RBC because the bandwidth sharing has less significant influence when the cache is filled with very few full files.
Figure 4.10 depicts the benefits of prefix caching and bandwidth sharing in the scenario of fixed cache size. When cache bandwidth is less than 8MB/sec, both versions of LRB yield higher HR mainly because of bandwidth sharing. As the number of cached full files decreases when disk bandwidth increases, LRB-P gradually takes more advantage of the prefix caching. The only benefit of bandwidth sharing that LRB enjoys becomes limited as less and less files are cached in full.

Compared to LRU-2, both LRB and RBC algorithms outperform LRU-2 with respect to BHR. When cache bandwidth is deficient, both LRB and RBC algorithms cache more popular files in full. However, the LRB algorithm can benefit from bandwidth sharing and the small amount of intervals/runs cached by LRB and RBC algorithms also contributes to the higher BHR performance. As cache bandwidth grows, both the LRB and the RBC policy cache a significant number of intervals/runs (the LRB-P may also cache prefixes) instead of some of the least popular files, which results in a greater use of bandwidth for writing into the cache. LRU-2, on the other hand, cannot take advantage of excessive bandwidth when there are many concurrent accesses on each file. Therefore, the advantage of LRB-P, LRB, and RBC over LRU-2 increases as the disk bandwidth keeps increasing.
Figure 4.8 Comparisons of Caching Algorithms with Varying Cache Bandwidth (Cache Size = 4GB)
Chapter 4: Performance Evaluation of LRB Algorithm

Figure 4.9 Comparison of Average File Fraction of Cached Entities with Varying Cache Bandwidth (Cache Size = 4GB, \( \theta = 0.15 \))

Figure 4.10 Comparison of Hit Ratio with Varying Cache Bandwidth

Disk Bandwidth (Bytes/sec)
Despite the marginal improvement in BHR of LRB-P and LRB over RBC, the LRBs are able to provide better quality-of-service by hiding user-perceived latency, network jitter and packet loss through prefix caching. Moreover, bandwidth sharing improves the HR performance of RBC by an average of 10% (see Figure 4.7 and 4.10), which is a significant improvement on the number of requests served from the cache.

4.7.3 Experiment Set-3: Varying Access Skew

BHR is also affected by the access skew. Figure 4.11 shows the effect of varying the access skew on the BHR performance of LRB-P. As the access skew becomes more uniform (with increasing $\theta$), the BHR falls as there are fewer concurrent accesses. Similar observations also hold for the results shown in Figure 4.5 and 4.8. The respective BHR of each algorithm, as well as the gaps between the LRU-2 and the other algorithms decrease, as the access pattern becomes more uniform. This is because both versions of LRB, as well as RBC tends to cache more popular files in full, and interval/run caching becomes less effective as short intervals occur more rarely for the less popular files.
Chapter 4: Performance Evaluation of LRB Algorithm

Figure 4.11 Performance of LRB-P with Varying Access Skew
4.7.4 Summary

Our simulation experiments demonstrate the performance benefits of the LRB caching algorithm. The evaluation independently varies each of the important parameters, namely cache size, cache bandwidth and access skew, and measures the impact on the BHR of each algorithm. The results can help guide the selection of these parameters in a real system, to allow network service providers to maximize the benefits of the proxy caches.

The simulation results demonstrate the benefits of resource-based caching over traditional caching approaches and further show that considering both space and bandwidth allocation of an entity offers significant improvements. The higher BHR performance achieved by the LRB algorithm compared to RBC is attributable to the following improvements:

1. Prefix caching: Prefix caching improves the BHR performance because prefixes efficiently use bandwidth while occupying less space compared to full files or intervals, thus prefix caching is especially advantageous when the cache is limited with respect to both size and bandwidth [27]. The main impact of prefix caching however is in its benefit for the quality-of-service in terms of hiding delays and packet loss [27]. Prefix caching also reduces the traffic between the server and proxy. This may not be a significant advantage for long streams, since the initial part of the stream would not represent a significant portion of the transfer. But, many Internet audio and video clips are short. Although the size and duration of CM streams are
likely to grow dramatically over time, particularly as high-bandwidth access networks become more common, the Internet is still likely to have a large number of short clips. The proxy prefix cache can store all, or at least a sizeable portion, of these short streams. For popular streams that are accessed by multiple clients, prefix caching can significantly reduce the load on the server, and on the network.

2. Bandwidth sharing: RBC only caches a full file if there is enough unallocated disk bandwidth. Moreover, it does not allow unused bandwidth that is currently allocated to a full file to be used to serve a request to another file. Statistical fluctuations in the actual number of concurrent requests for each file imply that the allocated bandwidth is often not fully utilized. In the case when a new request to a cached file arrives and all of the file's pre-allocated bandwidth is already in use, RBC allocates a new stream from the cache to serve the request only if there is enough unallocated disk bandwidth, or, if enough entities with lower relative bandwidth value can be evicted from the cache in order to allocate this new stream. As a consequence, these drawbacks result in a significant number of client requests that find their requested files in the RBC proxy cache but cannot actually read them from the cache due to limited disk bandwidth. Our algorithm improves on such limitations through bandwidth sharing. All the bandwidths allocated to the cached static entities (full files and prefixes) are grouped together to form a static bandwidth pool so that the unused portions of all the pre-allocated bandwidth could be shared among all the static entities in the cache. A request to any of the cached static entities can be served as long as a stream can be allocated from the static bandwidth pool, even though the pre-
allocated bandwidth of the corresponding entity has all been used. This increases the number of requests being serviced from the cache (see Figures 4.7 and 4.10). Once the stream finishes serving the request, its allocated bandwidth is returned to the static bandwidth pool to be used for other requests. Sharing of allocated bandwidth among the cached files results in improvement in performance especially for caches that are bandwidth deficient.

3. Replacement method: for the objective of maximizing BHR, the LRB algorithm defines the worthiness of an entity by its relative value, which measures the cost of fetching the entity from the remote server, the caching gain and the resource usage. By taking into account the cost of fetching the entity from the remote server along with the caching gain, the relative value quantifies the total savings (total bytes transferred) on the network more precisely than RBC, which only evaluate on the caching gain.

The reasons that LRB, LRB-P and RBC perform better than LRU-2 are (1) the two algorithms differ from LRU-2 by choosing the granularity of the cached CM entity from prefix, interval, run and full file, based on the cache state; (2) both the LRB algorithm and the RBC policy only replace a new entity with some of the cached ones if the relative value or the goodness value of the new entity is bigger. LRU-2, caches a file whenever it is accessed and selects for eviction the file with the largest backward-2-distance, that is, the largest difference between the current time and the time at which the 2nd most recent access was made to the file, until both the required space and bandwidth can be satisfied.
LRU-2 algorithm cannot benefit from granularity selection and must cache a full file whenever it is accessed. For a fixed bandwidth, when the cache size is small, both LRB and RBC algorithms select more intervals/runs than full files, except that LRB-P may also choose prefixes. Caching only full files has a negative impact on the performance of LRU-2 relative to the other algorithms in this case. As the cache size increases and the cache bandwidth becomes the bottleneck, the LRB and the RBC algorithms cache more full files, as does LRU-2, so the advantage of interval/run caching becomes less significant. In another scenario with a fixed cache size and varying cache bandwidth, when the cache bandwidth is low, both the LRB and the RBC algorithms fill the cache with more full files so as to amortize the writing overhead over multiple concurrent readers. As the cache bandwidth increases and the cache size becomes the constraint, the number of intervals selected by both algorithms increases. The relative performance of LRU-2 suffers from the limited number of full files it caches. Note this advantage is likely to remain even as caches get larger because the average size of CM objects is also likely to increase.

Overall, our simulation results demonstrate that the BHR performances of the LRB algorithms are statistically significantly better than LRU-2 and are marginally better than RBC. The relative execution times of LRB vs. RBC are approximately equal.
Chapter 5

Conclusions

Effective caching algorithms are very important to the development of Web-based multimedia applications. Multimedia objects are sufficiently large that they can consume huge disk space and bandwidth when cached indiscriminately. Motivated by the Resource-Based Caching (RBC) policy, prefix caching, and the cost/benefit model, we have proposed and evaluated a new policy, the LRB algorithm, for caching streaming media data at a Web proxy. In addition to effectively managing the local disk resources, the LRB algorithm also strives to maximize the BHR, an important metric that measures savings in the network traffic. Maximizing BHR reduces network traffic and server loads and improves end users' perceived quality of multimedia streams. We developed a model to simulate two versions of the LRB algorithm, as well as RBC and LRU-2 algorithms. We compared their BHR performance by independently varying three important parameters, namely cache size, cache bandwidth and access skew. Trace-driven simulations have shown that LRB overcomes limitations of RBC and exhibits uniformly better BHR performance than the other policies.
In general, our work has accomplished each of the goals set forth in Section 1.1 as follows.

1. We studied the RBC algorithm that handles CM data for disk based caching at the Web proxy, and used it as the basis for developing our algorithm. We also analyzed LRU-2, one of the most effective replacement algorithms in the memory and disk caching environments, and applied it to the streaming media context.

2. We proposed an effective algorithm, LRB, for caching CM data at a Web proxy server that is limited both on size and disk bandwidth. Our algorithm improves upon the BHR performance of RBC and LRU-2 by incorporating an improvement to RBC and by taking into account important performance factors, such as remote access latency and caching gain.

3. Through trace-driven simulation, we compared the BHR performance of the two versions of LRB algorithm with RBC and LRU-2, and proved that LRB outperforms the other algorithms under a wide range of cache configurations and access patterns.

5.1 Contributions

The research makes the following contributions to caching streaming media at Web proxies:
1. Design and implementation of the LRB algorithm, which contains the following three important improvements to RBC:

   (1) Caching the prefix to improve BHR and facilitate further savings on network traffic.

   (2) Sharing of allocated bandwidth among the cached static entities, which results in an improvement in performance.

   (3) Using criteria based on a cost/benefit model to assess the value of each entity. The definition of this criterion takes into account the network latency, caching gain and resource requirements of each entity; therefore, it more accurately reflects the total saving on the network traffic.

2. Development of a simulation model and implementation of different algorithms through trace-driven simulation. The simulation model is developed in Java. Using log files as the input, the model contains all the logic to implement LRB, LRB-P, RBC and LRU-2 algorithms and to compute the respective BHR of each algorithm under various cache configurations and access patterns. Through extensive trace-driven simulation, we were able to gain more insight into the behaviour and performance of each policy. We conclude that the LRB algorithms outperform the other policies over the system and workload parameters examined in this research.

3. Development of techniques to analyze and then modify the access skews of Web log files. To investigate effects of access skew on the BHR performance of each algorithm, we generated trace files with varying access skew based on the Zipf-like
distribution rule and the original log file. We simulated all the algorithms on different access pattern and the results suggested the more skewed the accesses, the higher the BHR.

5.2 Future Work

A number of topics remain open for future research:

1. Memory caching. Our algorithm is developed in the context of disk caching. However, it is more beneficial to deploy the caching strategy for streaming media in a real system environment that also includes memory caching.

2. Hierarchical cache system. Our algorithm only investigates the caching of streaming media data in a local disk array. However, the caching problem has intrinsically large size, the number of documents available in all the Web servers and number of requests served by a proxy cache are both large. The impact of the hierarchical cache system and, more generally, the architecture of multiple caches, needs to be investigated.

3. Streaming cache design. Our observations on the different caching algorithms are only based on simulation results. It is necessary to evaluate the performance of each algorithm in a real proxy cache environment. Congested networks and overloaded servers resulting from the ever-growing number of Internet users contribute to the lack of good-quality video streaming over the Internet. An effective streaming cache
should utilize its local memory and disk resources to reduce network and server load, while also improving the video and audio quality perceived by end users.

4. Study the influence of the RTP streaming protocol on the caching policy, particularly with respect to the issue of composing outgoing streams. The proxy cache might not have the entire media object in its local disk and might have to compose an outgoing stream by fetching the prefix data from the local disk and the rest of the data from the server. Therefore, special care is needed to create RTP headers for data streamed by the proxy.
References


Appendix A

Pseudo-Code of RBC Replacement Algorithm

- **Space_constrained (new entity \( E_i \)):**

\[ \forall \text{entities } E_k \text{ in cache} \]
\[ \text{if } (G_{s_k} < G_{s_i}) \text{ slist = slist + } E_k \]

if (slist is empty) return failure

else

sort the slist according to each entity's \( G_s \) such that the entity with the smallest \( G_s \) is placed at the top of the list

\[ j = \text{index of the entity at the top of the slist} \]
\[ \text{while } (j \geq 0) \]
\[ \text{free_cache_space = free_cache_space + } s_j \]
\[ \text{free_cache_bandwidth = free_cache_bandwidth + } b_j \]
\[ \text{mark } E_j \text{ for deletion} \]
\[ \text{if } (\text{free_cache_space} < s_i) \]
\[ j = j - - \]
else

remove all the corresponding marked entities from the cache
empty slist
return success

end of else
end of while
empty slist
return failure

end of else
• **Space_bandwidth_constrained (new entity $E_i$)**

∀ entities $E_k$ in cache
- if ($G_{s_k} < G_{s_i}$) slist = slist + $E_k$
- if ($G_{b_k} < G_{b_i}$) blist = blist + $E_k$

sort the slist and blist according to each entity's $G_s$ and $G_b$ respectively such that the entity with the smallest $G_s$ or $G_b$ is placed at the top of either list

m = the index of the top entity in the blist
n = the index of the top entity in the slist

while (space_and_bandwidth_constrained)
  if (m>0 && n>0) // victims are identified in both the slist and the blist
    if (free_cache_bandwidth/b_i < free_cache_space/s_i) //remove from the top
      while ($E_m$ has already been marked for deletion)
        m--
      end of while
      free_cache_bandwidth = free_cache_bandwidth + b_m
      free_cache_space = free_cache_space + s_m
      mark $E_m$ for deletion on both list if $E_m$ also in the slist
      m--
  end of if
  else //remove from the top of the slist
    while ($E_n$ has already been marked for deletion)
      n--
    end of while
    free_cache_space = free_cache_space + s_n
    free_cache_bandwidth = free_cache_bandwidth + b_n
    mark $E_n$ for deletion on both lists if $E_n$ also in the blist
    n--
  end of else
end of if

else if (m>0 && n=0) //victims are only identified in the blist, not in the slist
  while ($E_m$ has already been marked for deletion)
    m--
  end of while
  free_cache_bandwidth = free_cache_bandwidth + b_m
  free_cache_space = free_cache_space + s_m
  mark $E_m$ for deletion
  m--
Appendix A: Pseudo-Code of RBC Algorithm

end of else if

else if (n>0 & m≤0) //victims are only identified in the slist, not in the blist
  while (E_n has already been marked for deletion)
    n --
  end of while
  free_cache_space = free_cache_space + s_n
  free_cache_bandwidth = free_cache_bandwidth + b_n
  mark E_n for deletion
  n --
end of else if
else
  empty both slist and blist
  return failure
end of while

if (space_constrained)
  if (n > 0) execute space_constrained()
  else
    empty both slist and blist
    return failure
end of if

else if (bandwidth_constrained)
  if (m > p) execute bandwidth_constrained()
  else
    empty both slist and blist
    return failure
end of else if
else
  remove all the corresponding marked entities in the cache
  empty both slist and blist
  return success
end of else
Appendix B

Pseudo-Code of LRU-2 Algorithm

The pseudo-code uses the following notations:

\( \text{HIST}(p) \) denotes the history control variable of file \( p \); it contains the times of the two most recent references to file \( p \). Specifically, \( \text{HIST}(p,1) \) denotes the last reference, \( \text{HIST}(p,2) \) the second to the last reference.

Procedures to be invoked upon a reference to file \( p \) at time \( t \):

// update history information of \( p \) in the Access Queue:
scan through the Access Queue
if (\( p \) does not exist)
    create a new entry for \( p \) and allocate \( \text{HIST}(p) \);
    \( \text{HIST}(p,2) := 0 \);
else of if
    \( \text{HIST}(p,2) := \text{HIST}(p,1) \);
end of else
\( \text{HIST}(p,1) := t \);
if (\( p \) is already in the cache)
    // update history information of \( p \) in the cache queue
    \( \text{HIST}(p,2) := \text{HIST}(p,1) \);
    \( \text{HIST}(p,1) := t \);
    if (a stream can be allocated to serve the request without replacement)
        update cache state;
        return success;
Appendix B: Pseudo-Code of LRU-2 Algorithm

end of if
else

// identify victim(s) for removal
∀ file q in the cache queue
if (HIST(q,2) > HIST(p,2)) removelist + q;
else if (((HIST(q,2) == HIST(p,2)) && (HIST(q,1) > HIST(p,1)))
removelist + q;
if (removelist is empty) return failure;
else
sort all the files q in the removelist in an ascending order of
HIST(q,2) and HIST(q,1);
j = index of the file with maximum HIST(q,2) in the removelist;
while (j ≥ 0)
free_cache_space = free_cache_space + s_j;
free_cache_bandwidth = free_cache_bandwidth + b_j;
mark j for removal;
if (a stream can be allocated to serve the request)
remove all the marked files from the cache queue;
update cache state;
empty removelist;
return success;
end of if
else decrement j;
end of while
empty removelist;
return failure;
end of else
end of else
end of if
else // p is not in the cache
if (free_cache_space ≥ s_p && free_cache_bandwidth ≥ b_p)
place p in the cache queue;
free_cache_space = free_cache_space − s_p;
free_cache_bandwidth = free_cache_bandwidth − b_p;
end of if
else
// identify replacement victim(s)
sort all files q in the cache queue in ascending order of HIST(q,2) and
HIST(q,1);
j = index of the file with maximum HIST(q,2) in the cache queue;
while (j ≥ 0)
remove the victim with the maximum HIST(q,2) from the cache
queue one at a time;
free_cache_space = free_cache_space + s_q;

free_cache_bandwidth = free_cache_bandwidth + b_q;
if ((free_cache_space ≥ s_p) && (free_cache_bandwidth ≥ b_p))
  //cache the referenced file p
  place p in the cache queue;
  free_cache_space = free_cache_space - s_p;
  free_cache_bandwidth = free_cache_bandwidth - b_p;
  return success;
end of if
else decrement j;
end of while
return failure;
end of else
Appendix C

Pseudo-Code of Simulator

- Notations:

  - `bytesAccessed`: total number of bytes accesses
  - `bytesHit`: total number of bytes accessed from the cache
  - `diskBW`: total disk bandwidth (e.g. 8MB/sec, 16MB/sec, etc.)
  - `diskSP`: total cache size (e.g. 4GB, 16GB, etc.)
  - `bfree`: total available bandwidth in the free bandwidth pool
  - `sfree`: total available cache size
  - `bpool`: total bandwidths allocated to the static entity
  - `U_s`: space utilization
  - `U_b`: bandwidth utilization

- Pseudo-code:

  initialize cache configuration, parameters and variables;
  extract filtered logs from the original NLANR traces;
  for (i = 0 to 3)
    generate an access pattern based on the filtered log and the access skew parameter \( \theta \);
    for (j = 0 to 3)
      set a cache size configuration;
      for (k = 0 to 7)
        set a disk bandwidth configuration;
        proceed to LRB-P;
        proceed to LRB;
        proceed to RBC;
        proceed to LRU-2 (see Appendix B);
        increment k;
    end of for
    increment j;
  end of for
  for (m = 0 to 3)
Appendix C: Pseudo-Code of Simulator

set a disk bandwidth configuration;  
for (n = 0 to 7)  
  set a cache size configuration;  
  proceed to LRB-P;  
  proceed to LRB;  
  proceed to RBC;  
  proceed to LRU-2 (see Appendix B);  
  increment n;  
end of for;  
increment m;  
end of for
  increment i;  
end of for;

LRB-P (LRB is the same except for prefix caching)

i = index of the request in the request queue;  
for (i = 0 to index of the last request)  
  get a new request from the request queue;  
  update b_{free}, b_{pool}, s_{free}, U_s and U_b;  
  update access queue;  
  increment bytesAccessed;  
  if (cache queue is empty) proceed to A;  
  else  
    copy all the cached entities of the requested file into the process queue;  
    if (process queue is empty) proceed to A;  
    else if (the full file is cached)  
      if (a stream can be allocated to serve the request)  
        increment bytesHit;  
      else forward the request to the remote server;  
    else proceed to B;  
  end of else  
  increment i;  
end of for

output BHR;

A: Procedures to be invoked if nothing has been cached for the requested file

Check the access queue for the access record of the requested file;  
if (the file has been accessed before)  
  if (the previous stream has finished delivery)  
    select an entity between prefix and full file;  
  else select an entity among prefix, full file and interval;
end of if
else select an entity between prefix and full file;
run replacement algorithm;
if (the new entity is cacheable)
    if (the new entity is a static entity) update $b_{\text{free}}$, $b_{\text{pool}}$, $s_{\text{free}}$, $U_s$ and $U_b$;
    else
        update $b_{\text{free}}$, $s_{\text{free}}$, $U_s$ and $U_b$;
        increment bytesHit;
    end of else
    add the new entity into the cache queue;
end of if

**B: Procedures to be invoked if part of the requested file has been cached**

if (the prefix has been cached)
    if (a stream can be allocated to the request to read the prefix from the cache)
        increment bytesHit;
    else forward the request to the remote server;
end of if
if (there is a stream still writing part of the file into the cache)
    if (the prefix has been cached) select a new entity between run and full file;
    else select a new entity among prefix, run and full file;
    run the replacement algorithm;
    if (the new entity is cacheable)
        if (the new entity is a static entity) update $b_{\text{free}}$, $b_{\text{pool}}$, $s_{\text{free}}$, $U_s$ and $U_b$;
        else
            update $b_{\text{free}}$, $s_{\text{free}}$, $U_s$ and $U_b$;
            increment bytesHit;
            update attributes of the cached entity;
        end of else
        add the new entity to the cache queue;
    end of if
end of if
else
    check the status of the most current access on the file;
    if (it is still delivering)
        if (the prefix has been cached)
            select an entity between interval and full file;
        else select an entity among prefix, interval and full file;
    end of if
else
    if (the prefix has been cached) select the full file;
    else select an entity between prefix and full file;
end of else
Appendix C: Pseudo-Code of Simulator

run the replacement algorithm;
if (the new entity is cacheable)
  if (the new entity is a static entity) update b_free, b_pool, s_free, U_s and U_b;
  else
    update b_free, s_free, U_s and U_b;
    increment bytesHit;
  end of else
  add the new entity to the cache queue;
end of if
end of else

RBC

i = index of the request in the request queue;
for (i = 0 to index of the last request)
  get a new request from the request queue;
  update b_free, s_free, U_s and U_b;
  update access queue;
  increment bytesAccessed;
  if (cache queue is empty) proceed to C;
else
  copy all the cached entities of the requested file into the process queue;
  if (process queue is empty) proceed to C;
else if (the full file is cached)
    if (a stream can be allocated to serve the request)
      increment bytesHit;
    else
      run RBC replacement algorithm (see Appendix A);
      if (successful) increment bytesHit;
      else forward the request to the remote server;
    end of else
  else proceed to D;
end of else
increment i;
end of for
output BHR;

C: Procedures to be invoked if nothing has been cached for the requested file

Check the access queue for the access record of the requested file;
if (the file has been accessed before)
  if (the previous stream has finished delivery) select full file;
  else select an entity between full file and interval;
end of if
else select the full file;
run RBC replacement algorithm (see Appendix A);
if (successful)
    if (the new entity is an interval) increment bytesHit;
    update b_{free}, s_{free}, U_s and U_b;
    add the new entity into the cache queue;
end of if

D: Procedures to be invoked if part of the requested file has been cached

if (any stream is still writing)
    select a new entity between run and full file;
    run RBC replacement algorithm (see Appendix A);
    if (successful)
        if (the new entity is a run)
            increment bytesHit;
            update attributes of the cached entity;
        end of if
        update b_{free}, s_{free}, U_s and U_b;
        add the new entity to the cache queue;
    end of if
else
    check the status of the most current access on the file;
    if (it is still delivering) select an entity between interval and full file;
    else select the full file;
    run RBC replacement algorithm (see Appendix A);
    if (successful)
        if (the new entity is an interval) increment bytesHit;
        update b_{free}, s_{free}, U_s and U_b;
        add the new entity to the cache queue;
    end of if
end of else
• Program Structure

**Class Simulator**

Includes the main method. It initializes the cache configurations, parameters and variables used in the simulation; generates various access patterns based on the filtered log and access skew parameters of the Zipf-like distribution; and uses the generated patterns as the input to each of the simulated algorithm.

**Class AccessPatterns**

Generates a specific pattern based on the filtered NLANR logs and the access skew parameter of the Zipf-like distribution.

**Class LRBP**

Implements LRB-P algorithm and output corresponding BHR values.

**Class LRB**

Implements LRB algorithm and output corresponding BHR values.

**Class RBC**

Implements RBC algorithm and output corresponding BHR values.

**Class LRU2**

Implements LRU-2 algorithm and output corresponding BHR values.

**Class CacheQueue**

Maintains a list of all the cached entities (see Table 4.1 for the data structure of a cached entity).
Appendix C: Pseudo-Code of Simulator

**Class AccessQueue**
Records and update the access history for each distinguished CM file. See Table 4.2 for the data structure of the access queue.

**Class SpaceQueue**
Temporarily stores copies of entities whose RSV is smaller than that of the new entity to facilitate the replacement process. It includes methods that search for victim(s) for replacement and sort the entities according to their RSV(s). It is emptied after the replacement decision is made.

**Class BandwidthQueue**
Temporarily stores copies of entities whose RBV is smaller than that of the new entity to facilitate the replacement process. It includes methods that search for victim(s) for replacement and sort the entities according to their RBV(s). It is emptied after the replacement decision is made.

**Class CacheState**
Manages and updates the cache state. It includes methods that reclaim space and bandwidth allocation from any finished stream or deleted entity and allocate resources to a new entity, and execute the replacement algorithm.

**Class CachedEntity**
Creates a new entity object with specified attributes (see data structure in Table 4.1) and updates entity attributes upon a cache hit.
Class RequestQueue

Stores a list of request objects for each access pattern. Includes methods that sort the filtered logs according to the access numbers of each file, and modify the filtered logs according to the Zipf parameter.

Class Request

Generates a request object that is to be processed by each algorithm.

Class TraceParser

Extracts the logs on CM files from the original NLANR traces according to the "content_type".
Appendix D

Confidence Intervals

The accuracy of simulation results can be described in terms of confidence intervals placed on the mean values of the results. The confidence interval calculation procedure is described as follows:

Let \( Y_1, Y_2, \ldots, Y_N \) be the statistically independent results from \( N \) different runs of the same simulation. The sample mean, \( \bar{Y} \), of these results is:

\[
\bar{Y} = \frac{\sum_{i=1}^{N} Y_i}{N}
\]  

(1)

The variance of the distribution of the same values, \( S_Y^2 \), is:

\[
S_Y^2 = \frac{\sum_{i=1}^{N} (Y_i - \bar{Y})^2}{N - 1}
\]  

(2)

The standard deviation of the sample mean is

\[
\frac{S_Y}{\sqrt{N}}
\]  

(3)
Under the assumption of independence and normality, the sample mean is distributed in accordance to the Student-t distribution [41]. The upper and lower limits of the confidence interval regarding the simulation results are:

\[
\text{LowerLimit} = \bar{Y} - \frac{S_Y t_{\alpha/2, N-1}}{\sqrt{N}}
\]

\[
\text{UpperLimit} = \bar{Y} + \frac{S_Y t_{\alpha/2, N-1}}{\sqrt{N}}
\]

where \( t_{\alpha/2, N-1} \) is the upper \( \alpha/2 \) percentile of the t-distribution with \( N-1 \) degrees of freedom.

The simulation experiments in this thesis were run with a 90% confidence level with 10% confidence intervals for each data point. The number of simulation runs has been chosen big enough to ensure stability and tight confidence intervals.
## Glossary

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BHR</td>
<td>Byte Hit Ratio</td>
</tr>
<tr>
<td>CM</td>
<td>Continuous Media</td>
</tr>
<tr>
<td>HR</td>
<td>Hit Ratio</td>
</tr>
<tr>
<td>LFU</td>
<td>Least Frequently Used</td>
</tr>
<tr>
<td>LRB</td>
<td>Least Relative Benefit without Prefix Caching</td>
</tr>
<tr>
<td>LRB-P</td>
<td>Least Relative Benefit with Prefix Caching</td>
</tr>
<tr>
<td>LRU</td>
<td>Least Recently Used</td>
</tr>
<tr>
<td>LRV</td>
<td>Lowest Relative Value</td>
</tr>
<tr>
<td>MPEG</td>
<td>Motion Picture Experts Group</td>
</tr>
<tr>
<td>MTTR</td>
<td>Mean Time to Reaccess</td>
</tr>
<tr>
<td>NLANR</td>
<td>National Laboratory for Applied Network Research</td>
</tr>
<tr>
<td>RBC</td>
<td>Resource Based Caching</td>
</tr>
<tr>
<td>RSV</td>
<td>Relative Space Value</td>
</tr>
<tr>
<td>RBV</td>
<td>Relative Bandwidth Value</td>
</tr>
<tr>
<td>RTP</td>
<td>Real Time Protocol</td>
</tr>
<tr>
<td>RTSP</td>
<td>Real Time Streaming Protocol</td>
</tr>
<tr>
<td>TTR</td>
<td>Time to Reaccess</td>
</tr>
</tbody>
</table>