HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering

Jussi Kangasharju

SOFT CACHING – VARIABLE RATE CACHING OF MULTIMEDIA OBJECTS

Master's Thesis which has been submitted for revising
for the degree of Master of Science in Engineering.
Espoo, September 16, 1998

Supervisor	Prof. Olli Simula
Instructor	Prof. Antonio Ortega

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author:	Jussi Kangasharju
Name of the thesis:	Soft caching – Variable rate caching of multimedia objects
Date:	02.09.1998 **Number of pages:** 107
Department:	Department of computer science and engineering
Chair:	Tik-115 Computer and information science
Supervisor:	Prof. Olli Simula
Instructor:	Prof. Antonio Ortega

Caching of frequently accessed documents is widely used in the World Wide Web to reduce network traffic and alleviate congestion at popular servers. Traditional caching treats all objects as being immutable, but certain types of objects, namely multimedia objects, can sustain information loss without becoming unusable.

This thesis investigates the soft caching framework in which cached multimedia objects can be recoded into a lower resolution version in order to reduce the transmission time and space used by objects in the cache. Cache replacement algorithms specific to soft caching are derived and their impact on the overall performance is analyzed.

Experiments using actual proxy traces are conducted on the different cache replacement policies and recoding strategies using standard caching policies as a reference point for the performance comparisons.

Keywords: World Wide Web, proxy caching, image recoding

TEKNILLINEN KORKEAKOULU **DIPLOMITYÖN TIIVISTELMÄ**

Tekijä:	Jussi Kangasharju	
Työn nimi:	Joustavat välimuistit – Multimediaobjektien resoluution vaihtelu välimuistissa	
Päivämäärä:	02.09.1998	**Sivumäärä:** 107
Osasto:	Tietotekniikan osasto	
Professuuri:	Tik-115 Informaatiotekniikka	
Työn valvoja:	Prof. Olli Simula	
Työn ohjaaja:	Prof. Antonio Ortega	

Usein haettujen dokumenttien tallentaminen välimuistehin on laajassa käytössä World Wide Web -järjestelmässä verkkoliikenteen vähentämiseksi ja suosittujen palvelimien kuorman vähentämiseksi. Perinteisissä välimuistijärjestelmissä objekteja käsitellään muuttumattomina, mutta eräistä objektityypeistä, tarkemmin sanottuna multimediaobjekteista, voidaan poistaa informaatiota ilman, että objekti muuttuu käyttökelvottomaksi.

Tämä diplomityö tutkii joustavia välimuistijärjestelmiä, joissa tallennettuja multimediaobjekteja voidaan typistää alemmalle resoluutiotasolle, mikä lyhentää objektien lähettämiseen tarvittavaa aikaa ja pienentää niiden vaatimaa tilaa välimuistissa. Työssä kehitetään joustaviin välimuistehin soveltuvia korvaamisalgoritmeja, ja niiden vaikutuksia suorituskykyyn analysoidaan.

Eri korvaamisalgoritmeja sekä typistämismenetelmiä verrataan kokeellisesti perinteisiin korvaamisalgoritmeihin käyttäen aitoja proxy-palvelinten lokitiedostoja syötteinä.

Avainsanat: World Wide Web, proxy-välimuistit, kuvien typistäminen

Acknowledgments

I would like to thank professor Antonio Ortega of University of Southern California for the opportunity of having been able to perform my thesis research at USC as well as for his advice and suggestions. I would also like to thank Young Gap Kwon, my office mate at USC, for all the fruitful discussions and for taking care of our computer, dromio.

The first proxy trace used in chapter 1 as well as one of the traces used in the experiments in chapter 7 are provided by the National Science Foundation (grants NCR-9616602 and NCR-9521745), and the National Laboratory for Applied Network Research.

Finally, I would like to thank my family for their support throughout my studies.

In Los Angeles on September 2, 1998,

Jussi Kangasharju

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The explosive growth of World Wide Web (the Web) has surprised everyone. People today have an unprecedented amount of information right at their fingertips. They can access distant sources of information from the comfort of their own homes, share information with other people from all around the globe, and make purchases with a simple click of the mouse button. But the access to all this wealth of information comes at a price. If you want to see something, you must wait. Sometimes you must wait so long, that the World Wide Web has even been termed as the World Wide Wait.

Congested networks are by no means a newcomer to the computer world. They have been with us since the dawn of the computerized era and they will haunt us well into the future. In trying to solve the future problems we should look into the past and learn from the works of our predecessors. The past solutions have all revolved around multiple factors, as no single, magical remedy has proven to be enough to rid the networks of their congestion. New, more efficient protocols have been introduced, network links have been upgraded and the users have been educated in the virtues of bandwidth conservation.

This is also the way that the research around the problems of the Web has taken. Protocols are being updated, new techniques for reducing the traffic have been implemented, but will it be enough? As more and more people flock to connect themselves to the Web, and new and exciting services are opened, will these measures be enough? This thesis investigates a small facet of the global problem by studying the advantages of caching techniques specially designed for popular multimedia objects.

Multimedia data have been present in the Web from almost the beginning and visually pleasing Web-pages have been a large factor in Web's success among people. But the need to make Web pages more and more esthetically pleasing and attractive for commercial purposes is growing. No longer will a single image be sufficient, but to capture the attention of promiscuous users, a Web page must contain video and audio to spice it up. This multimedia revolution will increase the traffic in the network considerably, and ways to alleviate these problems must be found.

This chapter presents an overview of the Web architecture, its problems and motivates the advantages of using specific techniques geared for multimedia object-types to alleviate the current problems.

1.1 World Wide Web

The World Wide Web is based on a client-server architecture. The users use a client software (browser) to make requests for documents at remote servers. The server replies by sending the requested document and possibly some additional meta-data about the document itself, such as the content type, expiration date, etc. Most of the Web documents are written in Hypertext Markup Language (HTML) [1] and may contain images or links to other documents.

From a means of exchanging information between scientists, the Web has grown into a vast, chaotic mass of information accessible by anyone with a computer and a modem. The early popularity of the Web is widely attributed to graphical web browsers, such as NCSA Mosaic and Netscape Navigator, which quickly inserted themselves into the mainstream in 1994–95. Nowadays, information about almost anything is available on the Web and electronic commerce is making its mark.

To bring order into this chaos, some companies are running directory services, e.g. Yahoo![1], or search engines, e.g. AltaVista[2] which help users by either organizing the information in a hierarchical structure or indexing Web-documents and allowing users to make searches among the indexed documents.

The search engines have also brought a new type of client into the Web, the robot, or indexer. These are computer programs which fetch documents from the Web, parse and index them and should the document contain links to other documents, recursively go after those documents as well. This makes it possible to build a database containing information about an enormous number of Web documents. Users can then easily search using suitable keywords, and the search engine searches the database and gives as result links to documents which probably match the user's needs.

But even with the help of indexes and search engines, information still proves to be elusive in some cases. Sometimes the pages have not been updated which results in incorrect information being distributed and sometimes documents get removed which invalidates all links to the removed information. Of course, good administration and frequent updates will keep

[1] <URL:http://www.yahoo.com>

[2] <URL:http://www.altavista.digital.com>

the documents accurate but this solves only a part of the problems on the Web.

As some servers and documents are more popular than others, this can create congestion on some links which increases the time the users have to wait for complete delivery of the requested data. This congestion can be relieved in many ways, the three most popular being (in no particular order):

- Install better machines as servers and deploy faster links.
- Compress the transmitted data. Although this would only reduce the amount of data in the network, but would not relieve the heavily loaded servers.
- Use caching to reduce the traffic.

The first solution, to upgrade the infrastructure, would very likely lead into a vicious circle of continuous and costly upgrading. This is because it would seem to the users that the congestion problems have gone away, and they would still keep on happily wasting the available resources. Even though the cost of upgrading the links and servers would in the end be paid, at least in part, by the normal users, the price increases would probably not be sufficient to make the users find out about the causes of the problems.

This means that a more efficient way of using the current resources must be found. This is the way of the other two solution, compression and caching. Both of them have their merits, but most of the current research work has concentrated on caching.

1.2 Hypertext Transfer Protocol

The protocol used to transmit information in the Web is called Hypertext Transfer Protocol, or HTTP for short. The most widely deployed version currently is HTTP/1.0 [2]. It is a quite simple application level protocol, but unfortunately in its simplicity is a major contributor to Web congestion. This stems from the way the connections to servers are handled. For each object, be it an HTML-page, inlined image or a video sequence, a new HTTP-connection is made. Since many popular web pages contain quite a few inlined images [3] this causes a great number of connections being opened to the server and increases the load at the server. Also, due to the slow-start mechanism [36] of TCP, the underlying transport protocol, the connections start at a slow rate to avoid clogging the link even though the bandwidth would be available.

[3] At the time of this writing (May '98), the popular Dilbert-comic page located at <URL:http://www.unitedmedia.com/comics/dilbert/> has 22 inlined images!

The new HTTP/1.1 [3] remedies this problem by introducing persistent connections. A client opens a connection to a server and makes all requests using this one and same connection. This saves time since there is no need to set up new connections for the inlined images and the effects of the slow-start are mitigated by the longer connection time. HTTP/1.1 also introduces methods to control the caching of the documents and a possibility for clients to indicate which content types are preferred.

The other big improvement of HTTP/1.1 for the purposes of this thesis, is the possibility for progressive transmission using the range-request method. The advantages of progressive transmission under simple assumptions are studied in detail in [26, 28]. Progressive transmission means sending the object *in layers*, where the first layer contains a coarse version of the object and subsequent layers contain the finer and finer details. For progressive transmission to be effective, the object being transmitted must be encoded using a layered, or progressive, method.

1.3 Web Traffic Distribution

Knowing the nature of Web traffic is essential for developing methods to improve performance. For compression methods to be effective, most of the traffic should be uncompressed, since trying to compress already compressed documents is an exercise in futility. A study of FTP-traffic on the NSFNet backbone [10] found out that 31 % of the FTP-traffic was uncompressed. Most of this uncompressed traffic consisted of directory listings and similar frequently referenced objects.

Currently as HTTP has replaced FTP as the protocol causing most traffic on the Internet, the situation has changed since very little directory listings are being passed, but the users instead go directly for the HTML-pages. HTML-pages are typically not compressed, but virtually all other traffic in the Web is compressed in some way or another.

Similarly, for a caching scheme to be effective, the traffic must have locality of reference, at least to some degree. Locality of reference means that the same objects get requested again by the same or different users. Fortunately, this has been found to be the case [39].

A recent statistic about HTTP-traffic can be found in table 1.1. This table has been constructed from two days worth of proxy logfiles from <URL:ftp://ircache.nlanr.net/Traces/> and contains the relative parts of different types of Web-objects, in both the number of requests as well as the number of bytes transmitted. The total number of requests was 2,226,603.

Table 1.1 shows also the MIME-types [13] for the data. Because the available logfiles have been sanitized to protect the privacy of the users, information about the MIME-type, which is normally included in the logfiles, is not present. For table 1.1 this MIME-type has been determined from the

Type	Requests	Bytes	MIME-type
GIF-image	51.4 %	21.7 %	image/gif
JPEG-image	25.5 %	25.5 %	image/jpeg
HTML	16.0 %	9.9 %	text/html
CGI-Script	4.0 %	1.1 %	N/A
Other	0.9 %	4.5 %	N/A
Java	0.5 %	0.2 %	N/A
Binary	0.5 %	16.2 %	application/octet-stream
Audio	0.5 %	5.3 %	audio/*
Compressed	0.3 %	9.5 %	N/A
Text	0.3 %	0.3 %	text/plain
Video	0.1 %	5.8 %	video/*

Table 1.1: Traffic distribution from NLANR

filename extension in the URL and might therefore not be totally accurate.

In the table *CGI-Script* refers to URLs containing `cgi-bin` as part of the path or ending in `.cgi`, `.asp` or `.map`. *Java* is Java-related objects (`.java` or `.class`), *Binary* refers to executables (`.exe`, `.dll` or `.bin` and the like) and *Compressed* is URLs ending in `.gz`, `.zip`, `.Z`, etc. The category *Other* is for objects that could not be classified with the heuristic employed.

From table 1.1 we can clearly see, that requests to HTML-pages and a few image types constitute about 93 % of all traffic and that the two most popular formats are GIF and JPEG. These two formats account for almost 50 % of all the bytes transferred with the remaining bytes divided among binaries, HTML, compressed data, audio and video. Among these media types only HTML would lend itself well to compression, hence the use of compression to reduce the traffic on the Web might not yield the expected results.

In contrast with the above, more recent trace, table 1.2 shows the traffic distribution of the traces used in [9]. There were 1,143,842 requests in the trace.

This trace is from year 1995 and it is obvious that the content of the Web has changed dramatically over the last three years. Previously most of the traffic consisted of HTML-pages and GIFs with an occasional audio or video file thrown in. Nowadays, JPEG has become popular and the Web is also used as a means of distributing software, taking the role of FTP in this area.

Part of the JPEGs poor success in the Boston University trace could be explained by the software. The traces have been obtained from a modified version of the NCSA Mosaic which did not support inline JPEGs before version 2.6, released in July 1995, two months after the data collecting for the

Type	**Requests**	**Bytes**	**MIME-type**
GIF-image	70.4 %	38.8 %	image/gif
Other	17.7 %	16.6 %	N/A
HTML	10.7 %	8.3 %	text/html
JPEG-image	0.6 %	7.0 %	image/jpeg
Text	0.3 %	5.3 %	text/plain
CGI-Script	0.1 %	0.06 %	N/A
Audio	0.07 %	9.7 %	audio/*
Video	0.06 %	14.2 %	video/*

Table 1.2: Traffic distribution from Boston University (1995)

Boston University trace was finished. Therefore all the JPEGs in table 1.2 refer to images that have been loaded separately, and not as inlined images.

The Web has also become more interactive. This can be seen from the number of requests to CGI-scripts. These account for 4 % of the modern traffic but three years ago they were almost non-existent.

1.4 Organization

This thesis is organized as follows. Chapter 2 presents an overview of caching. Multimedia objects and their properties are discussed in chapter 3. The theory behind soft caching technology is detailed in chapter 4. In chapter 5 the JPEG-image format and its applications to soft caching are presented. Chapter 6 describes the different cache replacement algorithms studied in this thesis and chapter 7 presents the experiments performed. The results are presented in chapter 8 and chapter 9 discusses the results. Finally, chapter 10 provides some concluding remarks.

Chapter 2

Caching

Caching has been extensively used in microprocessors and operating systems to speed up memory look-ups or improve the performance of the virtual memory system. Also, distributed systems and, more recently, network applications can benefit greatly from caching. More details of caching in operating systems can be found in [37].

FOLDOC[1], Free On-Line Dictionary Of Computing, defines a cache as "a small fast memory holding recently accessed data, designed to speed up subsequent access to the same data." For processors the cache holds a few recently used memory pages in a special, fast memory in the processor. Virtual memory systems use the main memory of the computer to cache memory pages which otherwise would be paged out to disk.

A typical computer program exhibits large amounts of both temporal and spatial locality which can be exploited by a caching system to drastically improve performance. Temporal locality means that if a memory address is accessed, it will likely be accessed again in near future. Spatial locality means that when an address is accessed, the nearby addresses are likely to be accessed also.

If the address requested is already in the cache, a "hit" is scored and the data can be delivered quickly to the requesting program. If the requested address is not in the cache, a "miss" occurs and the data must be fetched from the main storage (main memory for processors, disk swap for virtual memory systems) at the cost of a usually significant slow-down. The ratio of hits to the total number of requests is called the "cache hit-rate".

If the requested data is not in the cache, it is fetched from the secondary storage, delivered to the requester and then it is written into the cache. This might cause the cache to overflow and some items will then need to be removed from the cache. Exactly which items are removed, depends on the replacement policy of the cache. In a virtual memory system, the Least Recently Used (LRU) (see [37]) policy, which removes the least recently used

[1] <URL:http://wombat.doc.ic.ac.uk/>

items has been found to yield the best performance.

These caching systems have some properties not shared by network or distributed applications. Since the objects being cached are memory pages, they all are of equal size and since a miss must be fetched from the secondary storage, which is the same for each memory page, the cost of a miss is also equal for each object.

2.1 Caching in the Web

The explosive growth of the traffic on the Web has forced people to find solutions to improve the speed of the Web. As the objects requested by the users are of varying sizes and spread out all over the world, over very differing links, the cost of a miss in a web cache depends greatly on the actual object requested. While it is relatively easy to find good algorithms for virtual memory caches, a web caching system must decide which of the following three criteria to optimize:

1. Minimize the number of requests passed to the origin servers, or maximize the cache hit-rate.

2. Minimize the number of bytes that need to travel in the network, or maximize the so-called byte hit-rate. The byte hit-rate is defined as the number of bytes found in the cache to the total number of bytes transmitted.

3. Minimize the time that the end-user must wait.

In a virtual memory system, where the objects are all of the same size and each miss costs an equal amount of time, maximizing the hit-rate will also optimize the other two criteria. But on the Web, where the sizes and link bandwidths differ, this is no longer true.

Most of the early work on web caching, such as [40], concentrated on maximizing either the hit-rate or the byte hit-rate and paid little attention to the time the users need to wait. Yet, the GVU's 8th WWW User Survey [14] found out, that the largest problem on the Internet experienced by the users is the download speed of the pages. Recently the work on web caching has taken also into account the time that the users must wait [4, 5, 32, 41, 42].

Other aspects of web caching which make it different from operating systems, is that not all objects can be cached and that the caching can be done in several places. It can be done at the client, the server or in the network between these two. In the following, issues regarding the cacheability of an object are explored and the three caching approaches are detailed and their properties are discussed.

2.2 Cacheable Objects

Objects on the Web are not identical. This means that the properties of the object must be taken into account when making decisions regarding that object. For example, a page, that shows the results from a database query should not be cached because the page might contain some sensitive information about the user which should not be shown to others. Likewise, any page which contains changing information, e.g. stock quotes, weather information, might give false and outdated information if taken from a cache.

Fortunately there are methods to counter these problems. The caching entity can discard certain types of objects (e.g. URLs containing `cgi-bin`) and the HTTP-server can inform the client about the lifetime of the object via the `Expires`-header. In the absence of explicit lifetime information, the caching entity must either verify the validity of cached objects (either periodically or when request is made) or use some heuristic to approximate the lifetime of the object. This latter method results in a faster performance but may also result in serving documents that contain outdated, or *stale* information.

Wessels [39] defines the *staleness* ratio as:

$$staleness = \frac{now - last\_update}{last\_update - last\_modification} \tag{2.1}$$

and estimates that the users would find a staleness ratio of 10 % acceptable. This way of calculating staleness has the advantage that older objects have low staleness values which intuitively means that if an object has not been changed in a long time, it will not likely be modified in the near future.

The new HTTP/1.1-protocol allows the server to give out much more information about the cacheability of the object by setting directives which the caching entity can use to help in determining the freshness of the object. An example of these methods is the possibility to state the conditions when a cached object should be revalidated and when it can be served directly from the cache.

Web caching does have one advantage compared to virtual memory systems. Because the Web is read-only, there are never any problems with viewers modifying pages which would invalidate all cached copies. The only place where the object can be modified is at the origin server and a cache can always make a request to see if the object has been modified.

Expiration issues will be discussed in more detail in chapter 6.

2.3 Client Caching

Caching at the client end has been present in the web for a long time. Even the earliest versions of the popular Netscape browser [24] implemented a cache which would store recently accessed pages in the client's memory and disk. Currently, all modern browsers implement some sort of caching.

Because the caching is done on the user's machine, this type of caching will especially benefit dial-up users, since a hit will avoid the need to transfer the object over the slow dial-up link. Another advantage of client caching is the automatic adaptation of the cache contents to the user's access patterns.

Suppose a user frequently accesses a given web page. If the page is cached at a dedicated per-user cache, as in a web browser cache, then the next access to the same page will very likely find the page in the cache. Were the cache shared between several users, the accesses of the other users might have caused this page to be removed from the cache.

2.4 Server Caching

While client caching serves to reduce the amount of data in the network, server caching reduces the load exerted on the server. A simple server cache would hold the most requested documents in main memory and thus avoid the need to read the object from the disk when it is requested.

Another way to reduce the load on the server is server replication. A classical example of server replication is the NCSA server (see [6, 20] for more details). To handle the increasing load, NCSA's solution was to make several workstations work in a round-robin fashion to balance the load on the server. This round-robin scheduling was achieved through a rotating DNS list which mapped the requests to the server[2] to the next available server. This scheme is also fault-tolerant with regard to server crashes.

2.5 Proxy Caching

The third possibility of caching in the Web is proxy-based caching. Proxies have been traditionally used to enable users behind a firewall to access the external network. But already in the early phases of Web development, in 1994, the first proxy caches appeared [21].

In proxy caching the cache is moved into the network, into a proxy server. This server receives requests from the clients and forwards them to the server. When the proxy receives the reply, it transmits it back to the client and caches a local copy of the object. This way, if another client using the same proxy requests the same object, it can be found in the cache, thus avoiding the need to fetch it from the remote server. A typical proxy setup is depicted in figure 2.1.

A proxy can cache either in-bound or out-bound traffic or both. An in-bound proxy receives requests from remote clients and requests the documents from the local server. At first glance, this type of arrangement might

[2] <URL:http://www.ncsa.uiuc.edu/>

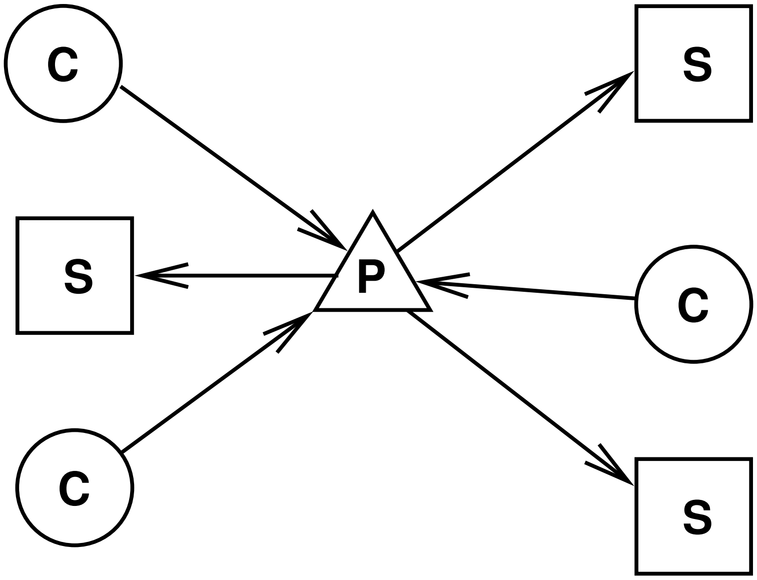


Figure 2.1: Clients (C), Proxy (P) and Servers (S)

seem a waste of money. Why should an organization run a server to the benefit of remote users? The answer comes from the caching done at the proxy. When a remote user requests the same document as a previous user, the document can likely be found in the cache and it can be served from there. This diminishes both the load at the server as well as in the local network. This type of proxy should be placed at the border of the local network and the Internet.

A much more typical application of proxy caching is the out-bound proxy. In this scheme, the proxy receives the requests from local clients and caches the documents from the remote servers. Typically this type of caching is used in campus and corporate networks to reduce the amount of out-going traffic. The advantage of out-bound proxy caching with regard to client caching is that if the users access mostly the same documents, it will be more efficient to cache only one copy of the document at the proxy instead of caching one copy at each client. The slow-down in speed, local disk vs. proxy server on local network, is usually negligible making this a very efficient form of caching.

2.5.1 Hierarchical Proxies

A new caching scheme is the hierarchical caching. The key idea is to have several proxy caches working together in a hierarchy. Hierarchical caching in the Web is discussed in more detail in [8]. The hierarchy can be anything from a small campus proxy tree to a large scale national hierarchical caching system [23].

In this setup, each proxy has a defined relationship to other proxies in the hierarchy (parent, child, sibling) and this relationship governs the proxy's actions. For example, when a requests results in a miss in the proxy, it might query its parent for the object. If the parent has the object in its cache, the document can be served quickly, since the proxies in the hierarchy are usually connected to each other by high-bandwidth links. If the parent does not possess the object requested, it can either query its parent or siblings or request the object from the origin server, depending on the configuration of the hierarchy.

2.6 Summary

Web caching is here to stay and current research efforts are aimed at making the most out of the existing infrastructure by means of caching.

Caches in the Web can be placed in different places – client, server or proxy. Each of these solutions has merits and shortcomings. Different types of caches can work together in a hierarchy which can span anything from a small corporate network to a large-scale nation-wide caching mesh.

Chapter 3

Multimedia Objects

As the statistics about Web traffic in chapter 1 show, most of the Web traffic is generated by multimedia data, i.e. images, sound and video. These multimedia objects account for 75 % of all requests and create almost 60 % of the bytes transmitted in the network. Since these types of objects have properties not shared by other object type (e.g. they can sustain information loss), it could be advantageous to treat them differently from other objects.

Currently most of the images on the Web are inline-images, meaning that the HTML-page contains a reference to the image which is shown by the browser as a part of the page. The other way of sending images over the Web is as separate objects. In this case the image is displayed either by the browser or by an external program. The images of the first type tend to be smaller in size.

Modern Web-browsers make it possible for users to enable and disable automatic loading of inline-images, and by default this is enabled. The GVU's 8th WWW User Survey [14] confirms that 90 % of the people turn off automatic image loading under only 25 % of the time. In other words, even though speed is perceived as the main problem, the image content is deemed important enough to warrant longer download delays.

Video and audio are slowly making their way into the Web as more and more user computers are powerful enough to handle the display of video sequences. A significant part of the video content on the Web is produced by large news companies, such as CNN[1], who use video sequences to deliver their news reports on the Web. The handling of video sequences at the client is done by an external program which sometimes is able to show the sequence in the browser window (working as a plug-in).

The use of multimedia objects to spice up the web pages is spreading and if the infrastructure can keep up with the development, there is little doubt that the future web pages will contain more and more multimedia objects to attract the users.

[1] <URL:http://cnn.com>

This chapter presents an overview of the different multimedia object types, their properties and suggests possible ways of processing them to improve the performance.

3.1 Images

Images have been a part of the Web since its beginning. Nowadays many popular web pages use graphical images in an attempt to make them more appealing. Even though the old saying about a picture being worth a thousand words might still hold, in the Web transmitting an image might be far costlier than transmitting those "thousand words." It is therefore imperative, that the image formats compress the information but still retain a good quality image. As seen in chapter 1, the most popular image formats used in the Web are GIF and JPEG and in the following an overview of these two formats and their uses is given.

Other image formats are also sometimes used, but major browsers do not support displaying them inlined and therefore they would have to be used as separate images.

3.1.1 GIF

The Graphics Interchange Format (GIF) is a lossless image format developed by CompuServe. Because GIF can only handle 256 different colors in one image, it is best suited for computer graphics, line art, drawings and such. Natural images, e.g. photographs, usually contain much more colors and thus degrade in quality if compressed with GIF. In the Web, GIF images are mostly used for navigation buttons, graphs and icons which can be efficiently compressed with GIF.

The newer GIF89a-specification [15] defines an interlaced mode which allows images to be transmitted progressively. Using the interlaced mode allows the browser to display a coarse version of the image already when only a small part of the image has been downloaded. However, the specification only allows for 4 layers in the interlaced mode and the last layer contains half of the lines in the image. This results in the image being usually almost unusable until the last layer has been received, which defeats the purpose of progressive transmission.

3.1.2 JPEG

The Joint Photographic Expert Group (JPEG) [29] is an ISO standard for compressing natural, i.e. real-world images. JPEG is a lossy compression method and is best suited for natural images. Line art and drawings usually do not take well to JPEG-compression. JPEG-standard defines both sequential and progressive modes of coding and leaves great latitude to the implementor regarding the choice of the details of progressive coding.

Since JPEG-format is the format chosen for soft caching in this thesis, it will be discussed in more detail in chapter 5.

3.1.3 Future Image Formats

Given the patent problems surrounding the LZW-compression used in GIF, the image coding community has set out to develop alternative, problem-free formats. The first to surface was the Portable Network Graphics (PNG) [30] which is intended as a replacement for GIF. For lossy compression, the JPEG-2000 (see [17] for the call of contributions) is being developed as the successor of JPEG. Other alternatives, such as the wavelet-based EZW [33] are also being studied.

It is not certain whether any of the proposed formats will emerge as a clear winner or if the current duality between lossy and lossless compression will persist. However, what is already known is that that the future image format *will be progressive.* All three formats mentioned above will support a progressive mode of coding and could therefore benefit from the advantages of progressive transmission.

Another important question surrounding future image formats is whether any of them will replace current GIFs and JPEGs. In order to be successful, a format must be supported by both the utilities used in creating the images and the browsers used by the viewers.

3.2 Video and Audio

The video and audio traffic on the Web can be divided into two categories:

1. Video and audio sequences transmitted as single objects.
2. Streaming transmission.

The video and audio sequences transmitted as single objects are better suited for caching. These are for example files stored at an FTP-server which can be downloaded and played later. The other category, streaming media, is a newer addition to the Web and is not easily cacheable. Streaming video and audio transmissions have usually been transmitted over the MBone [22] where all intervening entities are aware of the streaming nature of the transmission.

To cache streaming transmission, the caching entity, e.g. a proxy server, must intercept the traffic and store it locally. In addition, due to the lossy nature of streaming transmission (e.g. over UDP), simply storing the data is not enough. The caching entity must decode part of the transmission in order to discover which parts are missing and act accordingly. This would create extra load on the cache but would also allow the cache to *improve the quality* of the transmission by trying to recreate or approximate the missing bits.

While for images there are only two popular formats, this is not the case for video and audio. The streaming transmission are usually done using a special format, such as RealAudio [31]. For single objects the multitude of different formats is overwhelming. Audio files are mostly `.wav`, `.au`, AIFF, MP3 and video files mostly MPEG, QuickTime or AVI. The different formats require each different procedures to manipulate and since some formats contain proprietary extensions, it would be also costly to implement the functions needed to process these types of objects.

3.3 Recoding

Recoding is the name for the operation where an object is transformed through a lossy transform to a lower quality representation. An example of a recoding operation might be the removal of a scan layer from a progressive JPEG-image. Recoding throws away information contained in the object but this information loss should be done in a way that the observable effects of the operation are minimized.

Recoding works best in an environment where information loss can be tolerated and the objects being handled can be transformed into a progressive representation. It is clear that a text file can suffer little or no loss at all before becoming unusable, but an image, especially already lossily compressed JPEGs, do not suffer much from the removal of some information.

The recoded object has a lower quality than the original. This is obviously true, since some information was thrown away, but in most instances this information loss cannot be seem because it is very small. What can be easily seen, on the other hand, is the often substantial reduction in object size compared to the information loss. This means that objects can be recoded a few times without any observable loss of quality and a significant decrease in object size.

All this comes at a price. The recoding requires some CPU-time to perform the operation. It is therefore essential, that the attainable gains (shorter transmission time, smaller storage size) be carefully weighed against the CPU-time required to perform the operation. For this reason, object types for which the recoding operation is costly, i.e. video and audio, might not be useful for recoding purposes. All the more so since they are rarely requested. In this thesis, only JPEG-images are studied because they are widely supported and software to manipulate is freely available.

Recoding also raises a non-technical issue which must be dealt with. Even though the information loss is hardly visible, the image is not identical to the original image. The question is how the information about the recoding should be conveyed to the user. This matter is still under study, but a good way to tell the user could be a small icon or button outside the main browser window[2]. Equally important is that the user *must be able to request the full object* when the quality of the recoded object is not satisfactory.

3.4 Object Size

As mentioned above, before recoding an object it must be decided whether the potential gains of recoding are worth the CPU-time required to perform the operation.

Since memory is "cheaper" than CPU-time, recoding small object is not useful if only smaller storage size is the objective. For larger objects the potential savings are quite significant and the recoding is not that much more "expensive" to perform.

If one also considers the savings in transmission time, then also smaller object should be recoded, but whether even the smallest objects present enough gains depends on the environment. If the link from the cache containing the recoded objects is a high-speed LAN, the gains of recoding very small objects are measured in fractions of a second due to the small object size. For slower links, such as dial-up, even a small difference in size will make it worthwhile the CPU-power because the potential gains are large enough.

Another size consideration is how much space the object takes up on the screen. The amount of data taken up by an object could be diminished by shrinking the object both horizontally and vertically, i.e. downsampling the image. Using this method, large images could be made smaller and the amount of bytes in the image would be reduced drastically. It should be noted that this method could wreck havoc if the composition of the page containing the image has been carefully determined for this given image size. Even though HTML is *not* a page layout language, some content providers try to control the layout with certain sized images and other tricks. This is a completely futile pursuit because of the heterogeneous nature of the clients accessing the Web, e.g. different screens, graphical vs. non-graphical, etc. Nevertheless, it is widely used.

In this thesis only JPEG-images are considered. Because of this reason, the image size issues need more attention than with audio and video files. For these other two object types, because of the large sizes of the objects, recoding provides large gains. For example, if a 2 megabyte MPEG-video file is recoded by 5 %, it will be transmitted almost 30 seconds faster over a

[2] For example the bottom right corner of browser window as it is done in current browsers.

28.8 kbps link![3]

3.5 Summary

Multimedia objects are already popular on the Web and in the future their number will increase. Since multimedia objects have properties not shared by other types of objects, e.g. they can sustain information loss, it is possible to treat them differently. If the object can be divided into layers, then it can also be recoded into a lower quality version.

Recoded objects can be transmitted in less time than originals and they take up less space but the loss in quality is far less than the reduction in size making recoding an attractive option. The CPU-cost of recoding must be taken into account, especially in the case of very small objects.

Objects can be recoded either by removing higher layers of resolution or downsampling them. Both methods have their uses and present some dangers.

[3] GVU's survey found this to be the most common connection speed for users

Chapter 4

Soft Caching

Current caching strategies treat all objects the same way and are based on the assumption that object integrity *must* be preserved. This holds true for all caching schemes, regardless of where it is done. For many object types this assumption is perfectly valid. A file containing an executable binary program is useless unless all the bits are present. Likewise, is is not possible to remove much data from a textual file without rendering it unintelligible.

However, there is one large class of objects which can suffer information loss without becoming unusable. These are the *multimedia objects* as described in chapter 3. A multimedia object can be *recoded* to a lower quality form which takes up significantly less space without losing much of the quality. The main idea of soft caching is to take advantage of this recoding possibility to treat cached multimedia objects differently from generic objects. The goal is to improve the performance of the cache on all three measures from chapter 2, hit-rate, byte hit-rate and download time.

The term "soft caching" will be used in this thesis to describe a caching system, where the objects can be recoded and the cache can thus contain partial objects. The term "hard caching" or normal caching, refers to a classical caching system, where the object either is completely in the cache or is not cached at all.

Earlier work on soft caching [7, 27, 38] has studied the cache replacement criteria in a soft caching system and provides some initial simulations on the performance gain of a soft caching system against a normal system. The work has indicated that soft caching can have a performance clearly superior to that of a normal system. For example, the results in [38] show that the average download time in a soft caching system is in the best of cases only one third of that of the normal system.

This chapter will present the soft caching framework and discuss the differences of soft caching and normal caching. The impact of these differences on the three performance metrics from chapter 2 – hit-rate, byte hit-rate and download time – is explored. The implementation issues will also be discussed.

4.1 Framework

Soft caching has many possible application arenas. Figure 4.1 shows a simplified view of a proxy caching system.

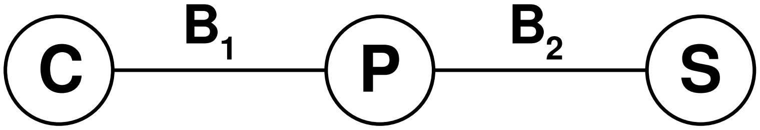


Figure 4.1: Simplified Proxy Caching System

In figure 4.1, C is the client, P the proxy and S is the server. The link bandwidths between these three are B_1 between client and proxy and B_2 between proxy and server.

4.1.1 Scenarios

Interesting scenarios where soft caching can be applied are:

1. $B_1 << B_2$. This is the typical dial-up scenario where the link between the client and the proxy is the bottleneck. To speed things up, the only possibility is to recode objects in the proxy before they are transmitted over the slow dial-up link.

2. $B_1 > B_2$. This scenario represents a LAN-setup where the client is connected to the proxy by a high-speed LAN. In this case, the objective of soft caching is, by using recoding and progressive transmission, to reduce utilization of the outgoing bandwidth.

3. $Cost(B_2) >> Cost(B_1)$. This is an interesting variation of scenario 2. In this case, the outgoing bandwidth is much more expensive financially than the bandwidth between the proxy and the client. The same philosophy applies, but soft caching should be more aggressive to reduce the link usage even more than in scenario 2.

4. $B_1 = \infty$. This is not a proxy caching scheme, but client caching. The client uses recoding to overcome memory limitations imposed on it. Since recoding makes objects smaller, the client can hold more objects. Because it is done at the client, the user can set preferences on how much objects should be recoded.

4.1.2 User Interaction

The interaction between the different parts – client, proxy and server – in a soft caching system is almost identical to the normal system. The phases of making a request are depicted in figure 4.2.

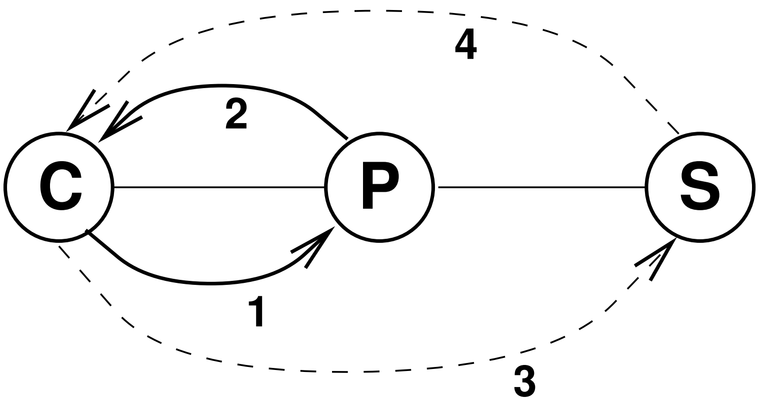


Figure 4.2: User Interaction in a Soft Caching System

The different phases in figure 4.2 are as follows:

1. Client makes a request.
2. Proxy responds by sending the object at the resolution currently stored in the cache.
3. If that resolution is not enough, the client makes a request to the origin server for the missing bits
4. Server responds. This reply is also cached at the proxy.

Ideally, the proxy would contain the exact resolution required by the client. If the resolution is too high (object is too large) then the resources at the proxy and the bandwidth on the client-proxy link are wasted. If the resolution is too low (quality is too low) then the resources on the external links are wasted.

The above strategy is known as the "hard access" strategy [38] in which the user always makes requests for full objects. This is the way the Web works today.

In the future, the "soft access" scenario, where the user can request partial objects, e.g. using the methods provided by HTTP/1.1, will be more likely. In this case the steps 1–4 from above will read:

1. User makes a request which may refer only to a part of an object.

2. Proxy responds immediately with the resolution available in the cache. If this is enough to satisfy the range specified in the request, the request is fulfilled.

3. In the case where the user requested more than was available in the cache, the proxy makes *immediately* the request to the origin server to get the missing bits. This happens in parallel with sending the already available data to the user.

4. The server responds with the missing bits which are forwarded to the user and stored in the cache.

4.2 Soft Caching Performance

The performance of a caching system can be measured along three axis. These are hit-rate, byte hit-rate and download time as presented in chapter 2. In the processor and virtual memory caches, all these three share the optimum due to the homogeneous nature of the objects being cached.

In the heterogeneous Web, the optimal point according to one of them, might not yield optimal performance according to the other two. Even though the metrics are not completely dependent on each other, they are not completely independent either.

For example, if one wishes to achieve a high byte hit-rate, one should cache large objects in favor of small ones. This is because a single hit on a large object will boost the byte hit-rate. Because high byte hit-rate means that a larger part of the bytes needed were found in the cache, one would be tempted to conclude that this also yields a shorter download time. Alas, this is not necessarily true, since the cached bytes might have come from a nearby, well-connected server while the uncached bytes needed to be fetched from a far-away server. In this example, caching the objects from the far-away server would have resulted in a shorter average download time.

The existence of partial objects in a soft caching system adds a twist into this already interesting scenario. This additional complexity follows from the lower quality of recoded objects and the subsequent possibility that a user might find this quality unacceptably low and request a reload. Such hits on cached objects should not be counted as full hits, since after all, the cache had made an erroneous decision concerning the object, i.e. recoded it when more quality should have been kept.

In the following, these three metrics will be analyzed in a soft caching context to present how they differ from their normal counterparts.

4.2.1 Hit-rate

Hit-rate is defined as the ratio of the objects found in the cache to the total number of objects referenced. Traditionally, hit-rate has been the most important metric because in the traditional caches all the three metrics were equal. In the early studies on Web caching hit-rate was also used as the primary metric.

However, on the Web, hit-rate is not a very useful metric. This is because gains in hit-rate do not translate into anything meaningful to the person operating the cache. A high hit-rate means that a large portion of the documents referenced were found in the cache. In fact, this means that the cache was doing more work and the operator should make sure that the cache is powerful enough to sustain this load.

High hit-rate does have one benefit, though. The more hits on the documents in the cache are scored, the less requests are sent to the origin server. This means a lighter load on the origin server which could be crucial for a very popular server where any reduction in load could result in visible performance boost.

In a soft cache all of the above remains true. The added complexity stems from the recoded objects. If a hit is scored on a heavily recoded object and the user requests a reload, should this count as a hit? The answer is no, since in this case the origin server has to serve the missing bits and therefore the load on the origin server is increased. This is contrary to the effects of high hit-rate in a normal system and therefore discarding the hit is justified.

4.2.2 Byte Hit-rate

Byte hit-rate is defined as the ratio of the number of bytes found in the cache to the total number of bytes transmitted. It can also be defined as the hit-rate, where each hit is weighted by the size of the object, and in some previous research it was called the weighted hit-rate [40].

Because the hits are weighted by the size of the object, having a high hit-rate does not guarantee a high byte hit-rate, nor is the converse of a low hit-rate resulting in a low byte hit-rate true.

Unlike hit-rate, byte hit-rate translates into something meaningful to the operator of the cache. The higher the byte hit-rate, the fewer bytes were sent on the out-going network link. A high byte hit-rate will therefore save bandwidth on the out-going link giving more room for other traffic or reducing the cost of existing traffic.

Because a high byte hit-rate means less traffic, one might be tempted to maximize the byte hit-rate in order to minimize the costs. This would mean caching the largest objects, since a hit on one of them would result in a much higher byte hit-rate. For a small ISP this might seem like an attractive scenario.

However, this would mean that a lot of smaller objects would not have been cached because of the space taken up by the large object. Since many objects are small (HTML-pages, small GIFs, etc.) this would drastically increase the average download time which makes the users unhappy. This might lead to the users deserting the small ISP in favor of a bigger one which provides a more sensible cache setup.

In soft caching the above is, again, true. Unlike hit-rate in soft caching, byte hit-rate does not suffer from recoded objects in soft caching. Even if a user requests a reload of a heavily recoded object, the bits that were cached need not be retransmitted from the origin server, under the assumption that the latter is capable of progressive transmission.

And when a user accepts a recoded object, the benefits should be calculated based on the original size of the object. The rationale behind this is that if the user accepts a recoded object, then the user thinks that this representation is equal to the original and therefore the cache has saved that many bytes on the out-going link.

4.2.3 Average Download Time

The average download time is the time that the user must wait on average for an object to be downloaded from the Web to the user's computer. It should be noted, that this is *per object* and not per Web-page which might contain several objects.

Of the three performance metrics, download time is the only one visible to the user. The shorter the average wait for a page, the happier the user. For smaller ISPs keeping their users happy might mean the difference of staying in business and going bankrupt.

Unfortunately, the download time is somewhat orthogonal to the other two metrics making it harder to optimize it. Bolot et al. [5] have derived the replacement criterion which minimizes the average download time of objects as seen by the user. The criterion is a function of the last reference time, object size and the time it took to retrieve the object.

As with byte hit-rate, soft caching has advantages with download time as well. Because the connection between the cache and the client is usually faster than the connection from the cache to the origin server, all the bytes that can be sent from the cache will reduce the average download time.

In the unlikely event that the connection from the client to the cache is slower than most of the links to the origin servers, then caching will not help much on reducing the download time since the object can be fetched faster from the server to the cache than the cache can transmit it to the client. This might be the case for some dial-up users.

4.3 Implementation

The important issues to be considered when implementing the soft caching framework are:

1. What types of clients?
   Are the clients connected through dial-up or LAN? In the former case recoding should be rather aggressive in order to make the objects smaller so that they can be transmitted faster. In the latter case efforts should be concentrated on maximizing the gains in byte hit-rate and download time.

2. When to recode?
   The fundamental issue here is whether recoding should be used instead of removing objects or should recoding be used as an additional tool. Recoding objects which would have been removed increases their lifetime in the cache and reduces the traffic when these objects are referenced again, since they usually are quite large.

   Using recoding on objects which are not being removed helps create more space in the cache while decreasing the download time.

3. Pure soft caching or mixed?
   Should the cache handle only recodable objects or both recodable and non-recodable objects. If the recodable and non-recodable objects are kept in separate proxy caches, it is easier to use different methods for treating them, such as different replacement policies, lifetimes, etc.

   The drawback however is, that the cache sizes must be predetermined. In a mixed proxy the recodable and non-recodable objects co-habit in the same space and can share it according to their true proportions. In a mixed proxy it is slightly harder to use different kinds of method for different kinds of objects, but a mixed proxy also means only one proxy to administer.

4. Where to recode? Recoding can be done at either the client or the proxy. At the client, recoding would be most useful in conserving space and at the proxy, recoding would be more useful in decreasing the download time and number of bytes transmitted.

   Even though all modern Web-clients allow the user to define how much memory should be used for the cache, it is far less certain whether the average user is aware of this possibility or would be able to adjust the cache size. Because the processing power at the personal computers is increasing faster than required by the applications, CPU-power of the clients is largely unused and could be used for recoding. The availability of CPU-cycles at a busy proxy is not guaranteed.

A working implementation of the soft caching framework is presented in [18]. It presents a proxy caching system which recodes JPEG-images instead of removing them in a mixed proxy setting. This is also the main setup considered in this thesis.

4.4 Related Research

An approach similar in philosophy to soft caching is the real-time distillation [11, 12]. In this framework, the clients can specify what types of documents and at which resolutions or representations they can handle. The proxy then extracts (distills) the required information from the original object.

The distillation is aimed at producing a suitable representation for a client with severe limitations in its capabilities. For example, a user using a hand-held PDA with a small, grey-scale screen could specify that all images are to be re-encoded to a resolution matching that of the screen and to use only grey-scale colors. Similarly, the proxy might extract the text from a PostScript-document to a client using a text terminal.

This approach is more aimed at overcoming client limitations and reducing the bandwidth usage on the client-proxy link than using the proxy to reduce the usage of the backbone. This technique also requires an added interface to set the target rates and formats for the distillation process.

The main difference between distillation and soft caching is that in the former, the amount of data transmitted over the backbone is the same as with any caching scheme, only the link from proxy to client is affected. Soft caching reduces the global amount of data transmitted while retaining the proxy-client link more like the normal system.

Recently a commercial implementation of a distillation-type framework has come to the market. It is manufactured by Spyglass and sold under the name Prism [34]. This system aims itself at non-PC-devices such as PDAs which have severely limited displays.

4.5 Summary

This chapter has presented the soft caching framework. Soft caching differs from traditional caching in that in a soft caching system objects can be recoded in order to reduce the transmission time and space used in the cache.

Issues related to the implementation of a soft caching system were discussed and an overview of an approach similar to soft caching, the real-time distillation, was presented.

Chapter 5

JPEG Image Format

Joint Photographic Expert Group, or JPEG, is an ISO standard for lossy compression of real-world photographic images. For a complete reference on JPEG, see [29].

Soft caching requires objects which support a layered coding where some of the layers can be removed without making the object unusable. The JPEG-format has these properties. The standard defines a progressive coding and since JPEG is lossy, a JPEG-image can sustain a little extra information loss. In addition, JPEG is a popular format which makes it easy to study soft caching using normal Web pages instead of artificially creating content.

Even though the GIF-format is far more popular than JPEG, it is at heart a lossless compression. In addition, the progressive GIF-coding places far too much information, half of the lines, into the last layer, making recoding immediately visible.

This chapter presents a short introduction to the baseline sequential JPEG-format and the progressive format and shows how they can be used in a soft caching system through recoding.

5.1 Sequential JPEG

The baseline format as defined by the standard is called sequential JPEG. The standard requires that a decoder *must* implement the sequential mode, all other modes – progressive, hierarchical and lossless – are optional.

The encoding process goes through 4 steps:

1. The image is divided into square blocks of 64 pixels. Each block is therefore 8-by-8.

2. Each block is transformed using the Discrete Cosine Transform.

3. The DCT-coefficients are quantized.

4. Quantized DCT-coefficients are compressed block by block using either Huffman or arithmetic coding.

In this process, all of the information loss occurs at quantization step (step 3) and the compression is performed at step 4.

To form the compressed JPEG-stream, the blocks are traversed in sequential order starting from top left corner and going left to right and top down.

The DC-coefficients (coefficient 0) are encoded with DPCM using the DC-coefficient from the previous block as the predictor. The AC-coefficients (coefficients 1–63) are collected in zig-zag-order (see figure 5.1) and are compressed using the compression method from step 4. The standard defines both Huffman and arithmetic coding but most implementations implement only the Huffman coding because of the patents covering the arithmetic coding.

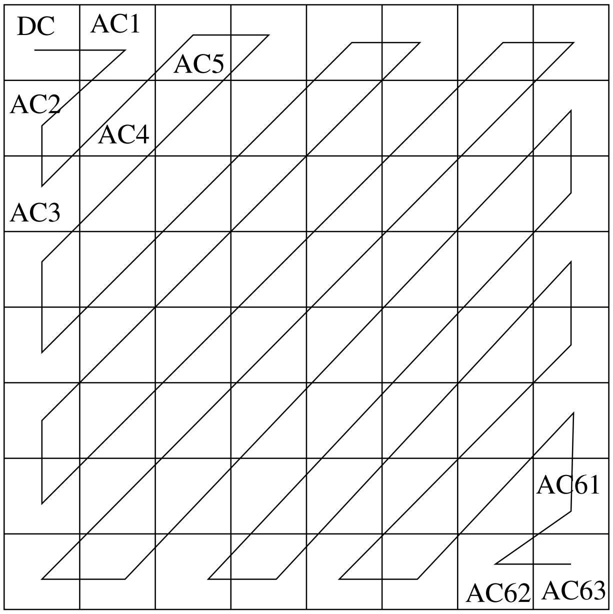


Figure 5.1: Zig-zag pattern for AC-coefficients

5.2 Progressive JPEG

Even though the standard includes the definition of a progressive mode for encoding JPEG-images, the first implementations did not implement the functionality needed to use the progressive mode. Later this functionality has been added into programs and currently all browsers and many other viewer programs are able to decode and show progressive JPEG-images.

When creating a progressive JPEG-image, the steps 1 to 3, i.e. transform and quantization, remain the same. The only difference is the order in which the coefficients are placed in the encoded JPEG-stream.

The standard does not define how to place the quantized coefficients into the JPEG-stream but leaves this choice to the implementation. The widely used Independent JPEG Group's library [16] divides the information into 10 layers by placing the coefficients as described in table 5.1.

Layer	**Component**	**Bits**	**Coefficients**	**Color**
1	DC	11111110	0	Y, Cb, Cr
2	AC	11111100	1–5	Y
3*	AC	11111110	1–63	Cr
4*	AC	11111110	1–63	Cb
5	AC	11111100	6–63	Y
6	AC	00000010	1–63	Y
7	DC	00000001	0	Y, Cb, Cr
8*	AC	00000001	1–63	Cr
9*	AC	00000001	1–63	Cb
10	AC	00000001	1–63	Y

Table 5.1: Scan format of IJG-library

As can be seen from table 5.1, the layers containing only DC-coefficients (layers 1 and 7) contain both luminance (Y) and chrominance (Cb, Cr) information. The AC-layers on the other hand, contain either luminance or one of the color components but never both luminance and chrominance. This means that a grey-scale JPEG-image coded in progressive mode has only 6 layers since the color components on layers 3, 4, 8 and 9 are not present.

Since the coefficients in the progressive mode are fed into the compression step in a different order than in sequential mode, the resulting JPEG-streams are not identical. In some cases, the progressive image is significantly smaller even though it contains exactly the same information. Table 5.2 presents the effects of progressive JPEG-encoding on three different types of JPEG-images.

In table 5.2 column "photographic" is for normal photographic images, column "graphic" for multicolored graphical images (e.g. navigation buttons) and column "line art" line art drawings containing only two colors, black and white (e.g. a comic strip). Each of the images was found on the Web, but their distribution in this study cannot be taken to represent the actual distribution of these types of images on the Web.

It can be seen that for photographic images, progressive JPEG-image is slightly more likely to be smaller than sequential, but that overall, there is

	Photographic	Graphic	Line art
Images	402	77	30
Progressive smaller	223	70	30
Progressive larger	179	7	0
Average size of progressive (% of original)	100 %	73 %	79 %
If smaller	98 %	70 %	79
If larger	101 %	104 %	N/A
Smallest progressive	96 %	29 %	46 %

Table 5.2: Savings from progressive JPEG-coding

no gain in size when going from sequential to progressive.

For the other two types, progressive mode offers substantial savings in image size. This is because these images contain large uniform areas which the progressive mode can compress more efficiently. The sequential mode works by 8-by-8 blocks and can not exploit similarities between blocks. The progressive mode does not possess this limitation and could, in the best of cases, compress a whole layer in one codeword!

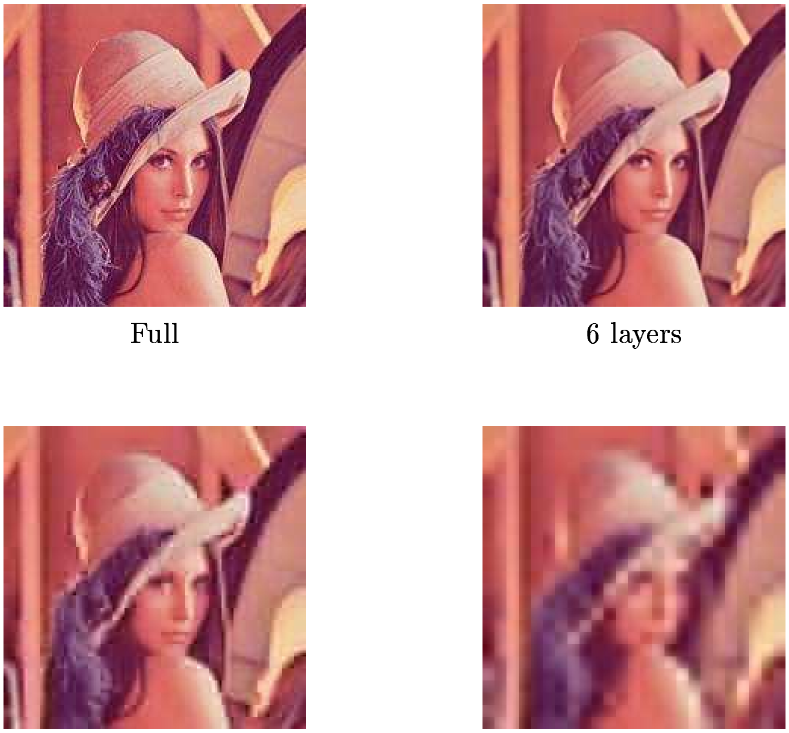
The graphical and line art JPEG-images are not very common, since the GIF-format is better suited for these types of images and usually yields better compression. However, there are cases when the GIF-format cannot be used and one must resort to JPEG. Such cases occur when the image has over 256 colors or the patents surrounding GIF make it impossible to use GIF.

5.3 Recoding and JPEG

The progressive JPEG-encoding offers an easy method of recoding. Since the exact format of the encoded stream and its contents is unspecified in the standard, any progressive JPEG-image can be recoded by simply removing the last layer of information. The result is still a valid JPEG-stream that can be decoded by any entity that is able to decode progressive JPEGs.

Of course, as more and more layers are removed from the image, the quality becomes lower and lower until at the last layer the only thing that is left is the seven most significant bits of the DC-coefficients. Figure 5.2 shows an example of the recoded Lena-image. In the figure are represented the original, unrecoded image as well as images with only 6, 3 and 1 layer of information left.

As pointed out in table 5.1 on page 29, the IJG-library does not distribute the coefficients uniformly in the scan layers, but instead uses a different method. This means, that different layers contain different amounts of bits



Full

6 layers

3 layers

1 layer

Figure 5.2: Progressively coded Lena

from the image. This opens up great possibilities for recoding. Table 5.3 shows the image size as a function of the original and previous layer during recoding.

The values in table 5.3 were obtained from a sample of 800 JPEG-images found on the Web which were recoded through all the 10 levels used by the IJG-library. The images were mostly small photographic images, resolution about 300x200 pixels and average size around 12 KB. Some of the images were much larger (1500x1000, 400 KB or even larger) and a significant part of them were non-photographic graphical images such as buttons and tool bars.

For reference, the values for the Lena-image in figure 5.2 are 57 % of original at level 6, 28 % at level 3 and 13 % at level 1.

As we see from table 5.3, the information is far from uniformly packed. Especially noteworthy are layers 9, 8, 4 and 3 which contain only color information and therefore do not amount to much of the total information. Another interesting observation is that the last layer represents almost 40 % of the total number of bytes in the image. This is because the last layer contains only the least significant bits of the AC-coefficients, which are generally

Level	**Percentage from previous level**	**Percentage from original image**
10	100 %	100 %
9	64.7 %	64.7 %
8	92.6 %	59.9 %
7	92.7 %	55.5 %
6	95.5 %	53.0 %
5	71.0 %	37.6 %
4	71.3 %	26.8 %
3	86.0 %	23.1 %
2	85.4 %	19.7 %
1	55.2 %	10.9 %

Table 5.3: Image size in the recoding process

close to a random sequence and therefore do not compress well.

5.4 Summary

This chapter has presented a short overview of the JPEG-image format and how to use the progressive JPEG in soft caching. Progressive JPEG-images were found to be at most the size of the sequential versions of the same images and in some cases impressive gains in image size are possible. Progressive JPEG also is well suited for recoding since several layers contain a lot of bits which are not visually important and can therefore be removed.

Chapter 6

Replacement Algorithms

The property having the greatest effect on the performance of a caching system is the cache replacement algorithm. When more space is needed in the cache, the replacement algorithm chooses the objects to be removed. This choice can be made on several different criteria, each having their own merits and shortcomings.

The results concerning caching from operating system theory are well-known, but the Web presents many new difficulties, to which the standard algorithms do not give a good response. As discussed in chapter 2, on the Web, the objects are of different sizes and reside on different servers, making the cost of a cache miss almost unpredictable.

An important decision concerns the amount of space to be freed. There are two possibilities:

1. Free only enough space so that the new object fits into the cache.
2. Whenever the cache size goes above a configured high-water mark, free enough objects to make the cache size go below a specified low-water mark.

Both of these approaches have their merits. The first one yields a better utilization of the cache space, since no unneeded empty space is created. This would make the latter strategy seem like a waste of space, since the cache utilization is never total. However, the second approach has the advantage that the replacement algorithm is not run as often as using the first method. If the criterion used to make the removal decision is complicated to calculate, or keeping a sorted list of the objects is infeasible, then this approach would yield superior performance.

In the following, classical hard cache replacement algorithms in Web caching systems are presented and their extensions to soft caching are discussed. The issues regarding stale documents and objects expiration are also explored further.

6.1 Hard Algorithms

This section presents an overview of some of the classical cache replacement algorithms. Only a few basic strategies are treated and their extensions to soft caching will be discussed in the next section. For a detailed analysis of the performances of hard algorithms, see [25, 41].

6.1.1 Least Recently Used (LRU)

The LRU replacement strategy removes the object that has been referenced the least recently. For this purpose, it is necessary to keep in memory the time when the document was referenced. This reference or access time needs also to be updated when a cached object is referenced. The philosophy behind this strategy is that if an object has not been recently referenced, it is unlikely that it would be referenced again in the near future.

LRU has been a very popular policy in virtual memory systems and has also been used widely in Web caches, such as the Squid proxy [35]. However, recent research has proven that this policy does not optimize the performance of a Web cache.

6.1.2 Size

The size replacement policy uses the object size as the key to determine which objects are to be removed. The usual way is to remove large objects in favor of smaller ones. This increases the cache hit-rate but may not help to improve the byte hit-rate. Since a Web cache must keep in any case the object size in memory, this policy incurs no memory overhead.

Since this policy keeps the smaller objects, it can keep more objects in the cache which also means more object meta-data. This could turn out to require enormous amount of memory. For example removing a 1 MB object and replacing it with 1000 1 KB objects would multiply the amount of meta-data for these objects by a factor of 1000!

6.1.3 Second Chance

The second chance algorithm works in the following way. When an object is to be removed, it is not removed but instead a flag is set to mark this event. If the object is subsequently referenced, this flag is cleared. If the object is not referenced again before it gets selected for removal the second time, this flag is still set and then the object is removed. It should be noted, that the second chance algorithm can use any algorithm to determine the objects to be evicted. Its only purpose is to give the object a “second chance”. It is therefore not a real replacement algorithm.

6.1.4 Others

In the last years the research community has come up with many improved Web cache replacement strategies to remedy the problems of LRU. Trace based simulations have shown that these strategies have superior performance over the basic LRU but they all require either complex calculations or holding a significant amount of extra data required by the algorithm.

Even though these newer strategies have slightly superior performance, it is still unclear whether this difference is in any way significant in a practical system.

6.2 Soft Algorithms

Any of the above-mentioned algorithms is naturally directly applicable to a soft caching system. However, because of the recoded objects in a soft cache, the normal version of the algorithms usually fall short. Since the purpose of this thesis is not to find yet another cache replacement strategy, only extensions to the existing algorithms are considered.

6.2.1 Soft LRU

The LRU algorithm uses the time that the object was last referenced as the key to decide which objects are to be removed. If the system decides not to remove a recodable object but instead recodes it, then the reference time must also be modified. If this is not done, then when the recoding is finished, the very same object will still be the least recently used and would be recoded again and again until it would be completely gone. This is clearly undesirable.

This negative effect can be handled by modifying the object reference time when the object is recoded. Even though the object has not been referenced, the recoding in a way "breathes new life" to the object, justifying this method. The only question is to which value should the reference time be set.

Possible values can be taken from the interval between current time and the old reference time. Ideally, the adjustment should depend on the recoding level of the object to reflect the usefulness of keeping the object in the cache.

6.2.2 Soft Size

Since recoding diminishes object size and the Size-policy favors small documents then it would prefer the recoded version of an object to the original one. This means that an object that only has a few layers of information left would be kept in the cache. Again, this is not desirable given the low quality of heavily recoded objects.

The way to counter this problem is to use the "useful size" of the object as the replacement key instead of the real size on the disk. The useful size is calculated from the original size and the recoding level. Intuitively, it should be obtained from the original size and current size in such a way as to take into account the gains in object size in regard to the degradation in quality.

A candidate for such a function would be:

$$useful\_size = current\_size + R(original\_size - current\_size) \tag{6.1}$$

where R is defined as

$$\frac{current\_recoding\_level}{maximum\_recoding\_level}$$

If the object can be recoded with exact precision, that is, if each recoding level is of the exact same size, then this formula will result in the $useful\_size$ being always equal to $original\_size$. In these situations another criteria is called for.

However, with JPEGs the information is spread non-uniformly over the layers and then the $useful\_size$ will be smaller at first and grow up to $original\_size$ as the object is recoded to the last few levels.

6.2.3 Soft Second Chance

The second chance algorithm is also easily modified to accommodate soft caching. In this simple modification, in addition to counting reference times, also the recoding instants are memorized. When an object is selected for removal, it is recoded and the time instant is memorized. If the object has not been referenced after this instant and is again selected for removal, it is removed instead of being recoded again. If it has been referenced, then it is recoded again.

As in the normal case, the second chance is not the real removal algorithm, but rather a modification of whatever algorithm is used to select the replaced objects.

6.2.4 Optimal Soft Replacement Algorithm

In [43] Yang and Ramchandran derive a criterion which can be used in a soft caching system to decide which parts of an object should be discarded in such a way as to minimize the average download time.

The authors propose to treat each layer as a separate objects and suggest that the objects be sorted according to the following criterion:

$$\beta_{ij} = P_{ij}(B_{S_i}^{-1} - B_C^{-1}) \tag{6.2}$$

In this criterion, P_{ij} is the probability, that resolution j of object i is accessed, B_{S_i} is the bandwidth to the server holding object i and B_C is the bandwidth from the proxy to the client. The objects with smallest β_{ij} are removed first.

Albeit optimal, this algorithm is far from simple to implement efficiently. The practical issues of implementing are discussed in more detail in [19]. The main problem is, that although it is possible to get accurate estimates for both the access probabilities and the bandwidths, this would consume a lot of memory which could otherwise be used to store objects.

Because modern browsers are able to request only whole objects, the access probabilities for the intermediate resolutions are the same as for the last resolution and the optimal replacement policy degrades into a hard replacement policy. In order to counter this deficiency, soft accesses must be simulated in order to make recoding work in this context.

In the simulations performed in this thesis, soft access was simulated by adding 0.1 to the object reference count every time an object was recoded. This choice is completely arbitrary, but necessary for recoding to work as expected.

The motivation behind this particular choice of adjustment is to make it appear as if the lower layers had been accessed a bit more often than the higher layers while trying to keep the effect on the global state of things minimal. With 10 recoding levels, an object will get at maximum 0.9 extra accesses through recoding.

The client bandwidth, B_C, was assumed constant and can therefore be removed. If the cache handles a multitude of differently connected clients, then the value of B_C would be different for each client and therefore the optimal choice of recoded objects would depend on the client as well. This presents a dilemma, and in a practical implementation it would be wiser to treat all clients in a uniform manner.

6.2.5 Summary

In all the strategies modified above, the modifications were simple since the effect of the recoding on the removal criterion could easily be determined. However, if the replacement strategy is a complex function of several different parameters, e.g. reference time, size and download time, it is far more difficult to assess the effects of the recoding.

6.3 Expiration Issues

An important requirement for a caching system is that it should not serve stale, or out-of-date information. In the interest of performance, most modern systems do not guarantee that an object served is *always* fresh (up-to-date) but instead use some heuristic to guess the lifetime of the object which sometimes results in serving stale objects. One such heuristic is the *staleness factor* from chapter 2.

To guarantee that the document is fresh, the cache must make a query to the origin server to make sure that the cached object has not been modified. This adds one round-trip time from cache to server to *each* request and is clearly undesirable. An easy way to avoid this would be for the servers to add explicit expiration information (HTTP `Expires`-header) to their replies. This way all intervening entities could explicitly know if a cached version is still current. Based on our experiences from our proxy, only 5 % of replies come back with this information.

If explicit expiration information is not available, then some heuristics must be employed. The goal of these heuristics is to make sure that only a certain, pre-defined fraction of the documents would be served stale. HTTP/1.1 will give the server and content provider more possibilities to share information about the object's lifetime to other entities. This information could help a cache to make decisions concerning the cached object.

In a soft caching system, these conditions can be relaxed because of the properties of multimedia objects. While a text file might be updated from time to time to add new information or correct errors, a multimedia object is likely to remain constant throughout its life. An image for example, is extremely rarely edited, it is rather replaced with a completely new version and these replacements are rather infrequent. It is therefore safe to assume that an image does not need to be verified nearly as often as normal objects.

The only exception to this "rule" are some special types of images, such as weather maps, that change content on a regular basis. In these cases it should be up to the content provider to produce the appropriate expiration information, since it is usually known.

Even though it is possible for content providers to provide expiration information, some use it in an incorrect manner. Because many web sites contain advertisements and the advertising revenue is based on the number of hits on the advertisement on the page, it is in the content providers financial interest to claim that the object is not cacheable.

This way all honest caches do not cache the object but fetch it always from the origin server causing another hit which increases revenue for the content provider. Otherwise the object could have been cached at any intervening cache which would have save network resources but any hits on the object would not have been registered at the origin server.

6.4 Summary

This chapter has presented an overview of some classical cache replacement algorithms as well as their possible extensions to accommodate soft caching. Also, an algorithm tailored to optimized the download time in a soft caching system was presented.

Since multimedia objects tend to stay constant for the duration of their lives, the criteria used to determine the freshness and staleness of objects in a cache can be relaxed when dealing with multimedia objects. Objects not sharing this property should be flagged as such by the content provider.

Chapter 7

Experiments

To compare the performance of the soft caching system to that of a normal system, simulations were run using actual traces from different proxies as input and the behavior of the simulator mimicked that of real proxies. The simulator would read the log file generated by a real proxy and would then act based on the parameters given. When the amount of data in the simulated cache would hit a high-water mark, the simulator would empty the cache enough to make the amount of data go below a low-water mark. This behavior is similar to the way the Squid-proxy behaves. More details of the simulator are given in appendix B.

Simulations were run using different cache sizes, different replacement algorithms and trying out different parameters for the soft caches. The high-water and low-water marks were fixed at 95 % and 90 % of the cache size, respectively.

This chapter first presents the traces used in the simulations and then details the different sets of cache parameters that were used in the simulations.

7.1 Traces

In the simulations three traces from three different places were used. The traces were obtained from the following sources:

1. USC
   This is the trace from our own laboratory and contains the accesses for a period of about one month. We have about 20 people using the proxy and some of them do not use browser caches making this trace a very accurate representation of user interests in an academic setting.

2. NLANR
   This is a trace from one of the NLANR top-level proxies[1] and contains two days worth of accesses. It should be noted that since this trace comes from a proxy high in a hierarchy, many of the requests may have been satisfied by proxies closer to the user. These requests do not, naturally, appear in the logfile.

3. Other
   This trace is a trace from the main proxy of a major university. Unlike the other two which have all the user requests present, it contains only the cacheable requests from the real trace, e.g. no references to CGI-scripts. The trace spans a period of one week and it had been sanitized before it was given to us to preserve the privacy of the users.

The trace from Boston University [9] was not used because of the low number of JPEGs in that trace.

Table 7.1 shows the number of requests, total number of megabytes contained in the traces as well as the average object size.

Trace	**Requests**	**Bytes (MB)**	**Average object size (KB)**
USC	147 099	1 410	9.5
NLANR	351 839	2 003	12.8
Other	1 940 904	16 354	8.4

Table 7.1: Properties of the three traces

In order to be able to evaluate the performance attained by the simulated cache, it is important to know, how much of the traffic could be cached and what would the maximum attainable hit-rate (HR) and byte hit-rate (BHR) be. To evaluate the gains in download time, the download time in the absence of any caching was also measured.

These numbers are presented in table 7.2.

Trace	**HR**	**BHR**	**Time (sec)**
USC	48 %	21 %	2.425
NLANR	18 %	8 %	4.518
Other	49 %	30 %	2.135

Table 7.2: Maximum hit-rates, byte hit-rates and download times

[1] This trace is from the SJ-proxy. It is not the same as the NLANR-trace in chapter 1 and table 1.1

From the numbers in table 7.2 it is evident that the high-level proxy (NLANR) does not see as many references to cached objects since these requests have been satisfied by proxies closer to the client. Also because the download time values were estimated from the actual service times of the proxies from which the traces came, the high-level NLANR-proxy trace also includes the time spent sending the information to the potentially distant client. This makes the value higher than that of the other two traces, but does not effect the results since the only the relative gain in download time was measured.

Of course, the possible benefits of soft caching depend on how much recodable objects are present in the trace. Table 7.3 presents the amount of JPEGs and GIFs in the traces.

Trace	**% JPEGs of requests**	**% JPEGs of bytes**	**% GIFs of requests**	**% GIFs of bytes**
USC	10 %	16 %	56 %	32 %
NLANR	21 %	22 %	36 %	16 %
Other	16 %	23 %	56 %	29 %

Table 7.3: JPEGs in the traces

Even though the current soft caching implementation [18] uses only JPEGs, the GIFs have been included in many simulations for a better understanding of the possible gains in situations where progressive formats are in common use.

7.2 Simulations

The goals of the simulations were to both compare the performances of a normal and a soft caching systems as well as to find good parameters for a soft caching system, e.g. number of recoding levels. Different sized caches were used to study how much the effects of the other parameters are affected by the size of the cache.

To observe the performance of soft caching in both proxy and client caches, also very small caches were used. The smallest were on the order of 5 MB which is the default disk cache of the Netscape Navigator.

Unfortunately, because of the small amount of memory[2] in the machines available for the simulations, not all combinations of cache sizes and parameters could be run. These memory limitations mostly affected the simulations of large size caches.

In order to be consistent with the behavior of the Squid-proxy, very large objects were not cached. In Squid this threshold is configurable by

[2] Only 64 MB of RAM

the operator, and is 4 MB by default. This same value was used for the simulations.

By caching extremely large objects it is possible to boost the byte hit rate if said large objects are referenced several times. Because caching an extremely large, rarely referenced object would evict a large number of smaller, potentially much more useful objects, not caching large objects has become standard practice.

7.2.1 Replacement Policy

Four different replacement policies were studied. These were:

1. Least Recently Used.
2. Size.
3. Second Chance.
4. Optimal Soft Replacement Policy.

The first two are as described in chapter 6. The normal system was only used with these two. LRU and Size were also used in the soft cache using the modifications from chapter 6. Second chance was used in the simulations of the soft cache to complement the main replacement policy as described in chapter 6.

The optimal soft replacement policy was used without soft access estimation as a hard replacement policy and with soft access estimation as a recoding cache. The access probability of an object was estimated by its reference count. Because the reference count is reset when an object is removed, this estimator does not give the true access probabilities of the objects. The bandwidth estimations were done on a per object basis instead of the more efficient per server basis.

Even though better estimators could be used, their implementation requires complex data structures and a significant amount of memory. This choice of estimators is the simplest and least costly to implement and is therefore a good candidate for a production-scale implementation. This is why it was chosen as the implementation of this replacement policy.

The bandwidth was estimated from the data available in the log file. The log files were all produced by the Squid-proxy and contained the object size in bytes and the time elapsed in fulfilling the request in milliseconds. The bandwidth was taken directly from these values and the resulting estimates were therefore in bytes per millisecond. For the traces used, these values ranged between 2.5 and 4.7 bytes per millisecond.

7.2.2 Recode Model

Two recoding models were used depending on which objects were being considered as recodable. These models were:

1. JPEG-model.

2. Linear.

The first model was used for JPEG-images. It follows the results from table 5.3 on page 32 which represent the real gains obtained by recoding JPEGs. Because GIFs were also considered as recodable in many simulations, a linear model was used to estimate the gains.

The linear model is intended to simulate the recoding effects of a popular future object format for which exact recoding would be possible.

7.2.3 Recoding Levels

Progressive JPEG as implemented by the IJG-library has 10 recoding levels. Interlaced GIF has only 4 layers and a progressive PNG has 8 layers. Naturally, the cache could follow the structure of each object but as seen from the recoding gains of JPEGs (in table 5.3), some layers contain only a small amount of data.

It could be advantageous for the recoding entity to treat every object internally as having a certain number of layers, a sort of a logical structure built on the true physical structure of the object. When the object would be recoded, the decision made on the logical structure would be translated to the real number of layers to strip.

In the simulations, three values of recoding levels were used. These were 3, 6, and 10 levels. These were mapped to the actual JPEG-recoding gains.

7.2.4 User Model

Users need to be simulated on two different factors. The first determines how much an object can be recoded before the user deems it unacceptable, the second relates to the capabilities of the software employed by the user.

In the simulations, the user was assumed to accept all objects, regardless of how many times they had been recoded. This would be a typical behavior of a dial-up user who would not want to waste time on fetching visually unimportant data.

The second factor in the user model was the ability to make requests to partial objects. To be consistent with modern browsers, the users were able to request only whole objects. This has important consequences for the optimal soft replacement policy, because in this setup, access probabilities for individual layers are unavailable; only accesses to whole objects can be counted. Therefore the soft accesses were simulated in the manner described in section 6.2.4.

The choice of accepting all recoded objects may yield a slightly higher performance for the soft caching system. This benefit is however countered by the inability to make partial requests, forcing the retrieval of the whole object from the server when an object is referenced the first time, even when in reality the user would only require a small part of it.

7.2.5 Refreshment Policy

The refreshment policy defines how recoding affects the object's access time in a system where replacements are based on object access times, such as LRU. Two models were studied:

1. Set access time to current time.
2. Use a linear model.

The first model always sets the access time of a recoded object to current time. This makes the recoded object the most valued object in the cache, but is also least costly to implement because the proxy must in any case keep track of time, and therefore the current time is available without any extra calculations.

The second, linear, model uses the following formula to set the new access time:

$$new\_access\_time = R(current\_time - old\_access\_time) \tag{7.1}$$

where R is defined as

$$(1 - \frac{new\_recoding\_level}{maximum\_recoding\_level})$$

This assigns less importance to heavily recoded objects and therefore such objects are removed faster.

7.3 Modifications to Squid

The freely available Squid-proxy [35] has been modified to do soft caching. This modified proxy has been in use for a little over a year at the Signal and Image Processing Institute (SIPI) at the University of Southern California (USC), Los Angeles.

More details on the implementation and its performance can be found in [7, 18]. Overall, the performance is good and recoded images have not been found to be disturbing.

In [18] the authors conduct an experiment where they compare the performance of the normal Squid with their modified version. Under a reasonable load, the performance is found to be as good as normal Squid's and no ill effects are caused by recoding. Under an excessively high load, the processing time taken by recoders is found to cause a significant performance hit.

7.4 Summary

This chapter has presented the proxy traces used in the simulations in this thesis. The choices for the parameters used in the simulations were detailed and discussed. Some of the choices made are sub-optimal but were chosen deliberately so in order to account for the difficulties inherent in the implementation of the optimal choices.

Chapter 8

Results

The simulations were conducted using the traces and parameters described in chapter 7. Because the available computers had only a small RAM-memory, 64 megabytes, not all simulations could be run. Because the memory requirements of the simulator grow as either the cache size or the number of object grows, this made simulating large caches impractical.

Similarly affected were simulations in which the number of objects in the cache is higher. These were mostly simulations with the Size-replacement policy which keeps small objects in favor of large ones and therefore the average number of objects in the cache at any given time is higher. Simulations which were not run because of these limitations, are indicated.

This chapter presents the results obtained from the different simulation runs. For more information about the simulator itself, see appendix B. Results are also analyzed in this chapter and the next chapter will discuss them in more detail. In the following the refreshment policy and the number of recoding levels will be investigated first, followed by the replacement policies.

Because both client and proxy caches were considered, the results have been divided into two groups. The first one presents small caches and the second one large caches. The threshold between large and small was set depending on the trace, but was never larger than 50 MB. Table 8.1 summarizes the cache sizes and how they were classified.

Trace	**Small Caches**	**Large Caches**
USC	5, 10, 20, 25	50, 100, 200, 400
NLANR	5, 10, 25, 50	100, 200, 400, 600, 800
Other	5, 10, 25, 50	600, 800

Table 8.1: Large and small cache sizes

8.1 Refreshment Policy

The refreshment policy governs how a recoded object is treated after it has been recoded. For normal replacement policies, such as LRU, which do not understand the layered structure of the recoded object, it is necessary to tweak some of the variables associated with the replacement decision.

Only the case of LRU was studied and the two policies were as defined in section 7.2.5. Different possible estimators for object size (for Size-policy) or soft-access estimates were not simulated.

The effects on all three performance metrics were studied separately and are presented in the following. In the tables, column *1* refers to the first choice and column *2* refers to the second choice from section 7.2.5, i.e. set access time to current time and use the linear model, respectively.

The results in this section represent the performance over the whole range of recoding levels studied.

8.1.1 Hit-rate

The results for small caches are presented in table 8.2 and large caches are presented in table 8.3.

Trace	**1**	**2**
USC	1	35
NLANR	16	0
Total	17	35

Table 8.2: Refreshment policy, Small caches (HR)

Trace	**1**	**2**
USC	0	8
NLANR	4	4
Other	0	8
Total	4	20

Table 8.3: Refreshment policy, Large caches (HR)

The results in tables 8.2 and 8.3 show that if the cache can expect a high locality of reference in the traffic, then it is better to use the linear refreshment model. If there is less locality of reference, then the simpler strategy outperforms the linear model.

The logic behind this lies in the ways these two models treat the objects. Setting the access time to current time makes the object the most recently "used" in the cache and thus prolongs the object's lifetime. Because of the low maximum hit-rate in the NLANR-trace, most objects are referenced only once. Therefore, by hanging onto objects, the cache sometimes stumbles on the right one and by keeping it, increases the hit-rate.

In larger caches, some effects of this are still visible, but the larger cache size makes it easier to keep the correct set of documents, thus the more sophisticated method gains in performance.

8.1.2 Byte Hit-rate

The effects of the refreshment policy on byte hit-rate in small caches are presented in table 8.4 and large caches are presented in table 8.5.

Trace	**1**	**2**
USC	10	26
NLANR	12	4
Total	22	30

Table 8.4: Refreshment policy, Small caches (BHR)

Trace	**1**	**2**
USC	0	8
NLANR	2	6
Other	0	8
Total	2	22

Table 8.5: Refreshment policy, Large caches (BHR)

Again, the same effect as was seen with hit-rate-results appears. Even though the NLANR-trace benefits more from the linear model than with hit-rate, the same situation prevails. As before, larger caches benefit more from the linear model than from the simpler "set to current time"-model.

8.1.3 Download Time

The effects of the refreshment model on the download time are shown in table 8.6 for small caches and in table 8.7 for large caches.

As with hit-rate and byte hit-rate, the high-level proxy does not benefit from the linear model. The results are almost identical to the results obtained for hit-rate.

Trace	**1**	**2**
USC	3	33
NLANR	16	0
Total	19	33

Table 8.6: Refreshment policy, Small caches (Time)

Trace	**1**	**2**
USC	0	8
NLANR	4	4
Other	0	8
Total	4	20

Table 8.7: Refreshment policy, Large caches (Time)

8.2 Recoding Levels

Three different choices for the number of recoding levels for recodable objects were simulated. These were 3, 6 and 10 levels. Due to early results showing, that recoding with only 3 levels is less effective than using more levels, the later simulations were run with only 6 and 10 levels.

In the following, columns in the tables indicate the number of recoding levels used in the experiment. The numbers reflect the number of times that choice was better than one of the other two possibilities. Therefore, for each simulation run, the best performing choice of levels got 2 points and the second best only 1 point.

8.2.1 Hit-rate

Results for small caches with 3 and 2 recoding level possibilities are presented in tables 8.8 and 8.9, respectively. Large caches are shown in tables 8.10 and 8.11.

Trace	**3**	**6**	**10**
USC	14	41	76

Table 8.8: Recoding levels, Small caches, 3 options (HR)

The results clearly show that having more recoding levels is better than having fewer recoding levels. Except for the NLANR-trace, higher number of recoding levels yields better performance more often than lower number

Trace	6	10
NLANR	12	20

Table 8.9: Recoding levels, Small caches, 2 options (HR)

Trace	3	6	10
USC	2	18	28
NLANR	22	22	19
Total	24	40	47

Table 8.10: Recoding levels, Large caches, 3 options (HR)

Trace	6	10
USC	2	14
NLANR	9	5
Other	4	4
Total	15	23

Table 8.11: Recoding levels, Large caches, 2 options (HR)

of levels. It is also clearly visible that having 10 levels is better than having only 6.

Even though simulations on the NLANR-trace do not show a clear performance advantage to a higher number of recoding levels, the results do *not* give an advantage to a system with a small number of recoding levels.

A high number of recoding levels means that the cache will hold onto the object longer, but because this results in a performance increase, the recoded objects are useful and therefore recoding and storing recoded objects is not a waste of resources.

8.2.2 Byte Hit-rate

The results on the effects of recoding levels on different cache sizes and different number of recoding level options are in tables 8.12–8.15.

Trace	3	6	10
USC	11	48	73

Table 8.12: Recoding levels, Small caches, 3 options (BHR)

Again, the results are similar to the ones obtained with hit-rate. Even

Trace	**6**	**10**
NLANR	9	23

Table 8.13: Recoding levels, Small caches, 2 options (BHR)

Trace	**3**	**6**	**10**
USC	0	18	27
NLANR	16	20	30
Total	16	38	57

Table 8.14: Recoding levels, Large caches, 3 options (BHR)

Trace	**6**	**10**
USC	5	11
NLANR	2	12
Other	3	5
Total	10	28

Table 8.15: Recoding levels, Large caches, 2 options (BHR)

the high-level NLANR-trace shows a clear advantage for simulations with a larger number of recoding levels.

Holding onto recodable objects (JPEGs and GIFs) can yield high gains in byte hit-rate since a hit on an image (or a similar object) gives a boost on byte hit-rate because these objects are generally larger than HTML-pages.

This advantage stems also from the small number of unique bytes present in the traces. Because the maximum byte hit-rates are quite small (from 10 % to 30 %), hanging onto objects already in the cache likely results in higher byte hit-rate.

8.2.3 Download Time

Tables 8.16–8.18 show the results from the experiments on the effects of the number of recoding levels on the average download time.

Trace	**3**	**6**	**10**
USC	22	42	68

Table 8.16: Recoding levels, Small caches, 3 options (Time)

Once more, the results from these experiments confirm the findings from

Trace	**6**	**10**
NLANR	22	10

Table 8.17: Recoding levels, Small caches, 2 options (Time)

Trace	**3**	**6**	**10**
USC	4	16	28
NLANR	21	12	24
Total	25	28	52

Table 8.18: Recoding levels, Large caches, 3 options (Time)

Trace	**6**	**10**
USC	6	10
NLANR	10	4
Other	3	5
Total	19	19

Table 8.19: Recoding levels, Large caches, 2 options (Time)

the experiments on hit-rate and byte hit-rate. In other words, more levels is better than fewer levels.

For the NLANR-trace the results are less clear, but this is in part due to the low locality of reference observed in this trace, and in part due to the trace including also the time it took to send the information to the client. This is naturally included in the other two traces, but in these the clients are network-wise very close to the proxy and therefore the delays in getting the data from the proxy to the client are negligible compared to the delays on the global Web.

In the NLANR-trace, the clients are all the clients in the hierarchy served by this proxy, spread over a wide area which results in large variation of transmission speeds towards the clients.

8.3 Replacement Policy

The replacement policies were simulated using different combinations of refreshment policies and numbers of recoding levels. The results presented in this section show the soft replacement strategies only with the set of parameters that was found to yield the best performance in the experiments on refreshment policy and recoding levels. The choice that is shown in the following corresponds to the linear refreshment model and 10 recoding levels.

In some cases, this choice was not the best of choices for the soft replacement strategy, but was always among the top performers. This was especially true for the NLANR-trace for which the choice of refreshment policy and recoding levels is not as straightforward as for the other two. Regardless, the same choice was used to plot all of the following graphs.

In the graphs, the different replacement policies are indicated as shown in table 8.20.

Label	**Policy**
LRU	Normal LRU, no recoding
LRU-SC	LRU, second chance recoding, see sec. 6.1.3
LRU-SOFT	LRU, always recoding
SIZE	Size, no recoding
SIZE-SC	Size, second chance recoding
SIZE-SOFT	Size, always recoding
OPT-HARD	Optimal Soft Policy, no estimation of soft access
OPT-SOFT	Optimal Soft Policy, soft access estimated, see sec. 6.2.4

Table 8.20: Replacement policies

Only comparisons between replacements with the same base algorithms were done, i.e. LRU-based were never compared with Size-based algorithms. This was done in order to be able to evaluate the gains due to soft caching and eliminate the effects of the underlying replacement algorithm.

The exception to this rule was the optimal soft replacement policy, in both its non-recoding and recoding versions. The "hard" version establishes a baseline which shows how much of the gain in using the recoding version is due to using a different replacement strategy. The "soft" version which simulated soft accesses then shows how much of the gain is due to soft caching.

The optimal policy was compared against both the LRU-based and Size-based algorithms to see how well it stacks up against normal replacement policies.

Plots on hit-rate and byte hit-rate show the performance relative to their respective maximum values for each trace as shown in table 7.2 on page 41. The download time plots show the average download time in the simulation relative to the download time for the no-caching scenario as per table 7.2.

8.3.1 USC-trace, Small caches

Figures 8.1– 8.3 show the performance plots for the USC trace on small caches and the LRU-based replacement policies.

In each of the plots, the soft variants of the replacement policies, i.e. second chance and full recoding for normal LRU and simulated soft access for the optimal policy, outperform the normal policies, often with by a large margin. This is especially true for the optimal replacement policy and its simulated soft variant.

This large gain might be in part due to the manner in which the access probabilities are estimated. Because the reference count is zeroed every time an object is removed, the estimates are not fully accurate. Furthermore, the small cache size means that unless an object is referenced very frequently it will have a low access probability estimate even if it is among the most popular objects. Therefore the normal policy suffers a performance hit which is seen on the plots.

The simulated soft access does affect the reference count (by adding 0.1 for each recoding) and therefore changes the access probabilities somewhat. However, the gains on hit-rate are between 25 % and 65 % and on byte hit-rate between 30 % and 65 % and this gain is due to recoding performed on the objects. This becomes obvious when looking at the relative performances of the normal LRU and the non-recoding version of the optimal replacement policy. In all cases, the performance of the hard optimal policy is inferior to that of the LRU, yet the optimal policy with simulated soft accesses clearly beats LRU in all of the situations.

For the soft LRU, the gains are smaller, around 15 % for hit-rate and around 20 % for byte hit-rate. These gains are, however, almost constant regardless of the cache size and therefore any implementation of a similarly sized cache would benefit from them.

The gains in download time are smaller, but in every case, the recoding version performs better than the non-recoding version. Significant reductions in download time are much harder to bring about because when an object is referenced the first time, it must be fetched from the origin server, no matter what replacement policy is implemented in the cache.

The only exception to this rule would be true soft access, because in this case the user can explicitly requests only a part of the object thus decreasing the time needed to transmit the required bytes from the server. Due to the nature of the log files, true soft access statistics were not available and all accesses were assumed to be hard.

The soft variants of LRU do not offer significant savings on download time, but the optimal policy with recoding gives around 6–7 % savings on the average download time.

The plots of the Size-based replacement policies are shown in figures 8.4–8.6.

The results are in general similar to the ones obtained with LRU-based replacement policies. Soft variants of the Size-policy have a performance far superior to that of the baseline algorithm.

The gains for the soft variants of the normal Size-policy in hit-rate range between 13 % and 34 % and in byte hit-range the gains are situated between 66 % and 147 %! Clearly, soft caching has a definite advantage over normal caching.

The soft optimal strategy is on par with the recoding Size-policy and on byte hit-rate has a slight advantage. The hard version of the optimal strategy is worse than the baseline Size-algorithm except on byte hit-rate where it slightly outperforms the second chance recoding.

In download time, the gains are smaller than in hit-rate or byte hit-rate, but now the soft variants of the Size-policy outperform the baseline version by a clear margin, unlike in the LRU-case (figure 8.3). The gains are in the order of 6–7 %, the same as for the optimal soft replacement policy.

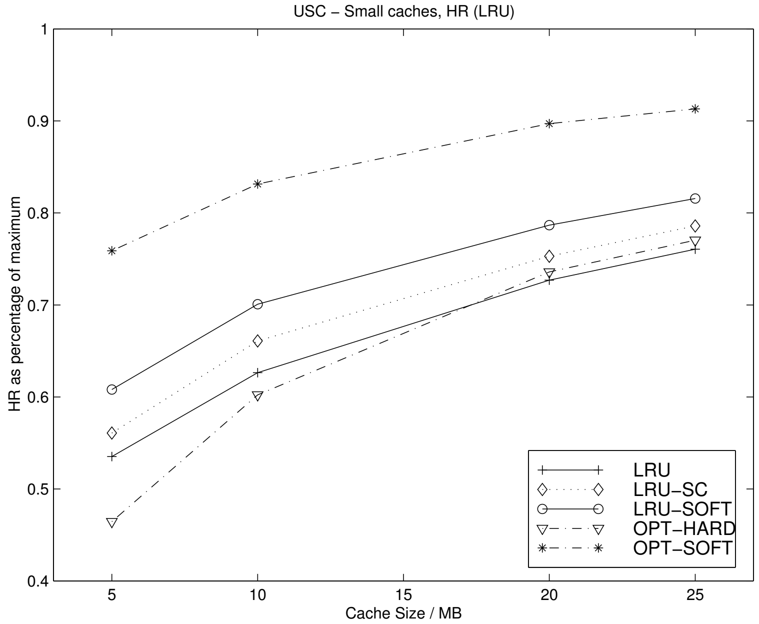


Figure 8.1: USC-trace, Small caches, HR (LRU)

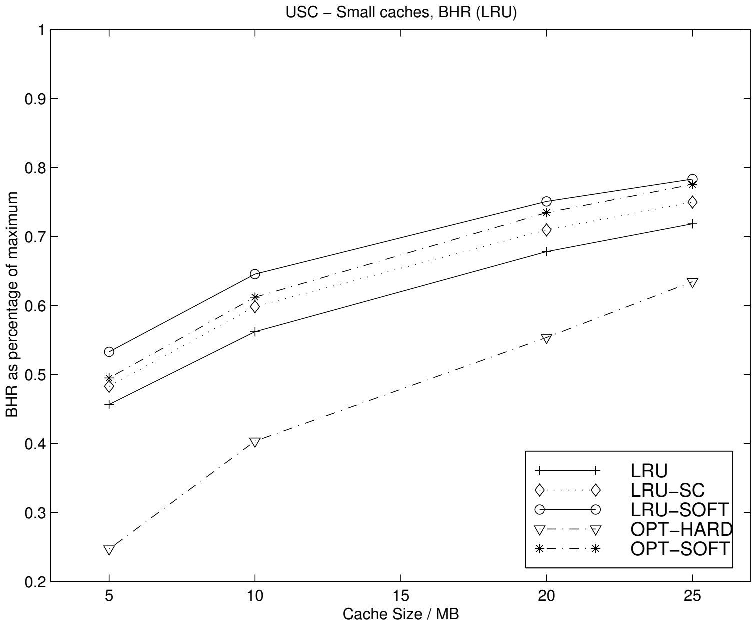


Figure 8.2: USC-trace, Small caches, BHR (LRU)

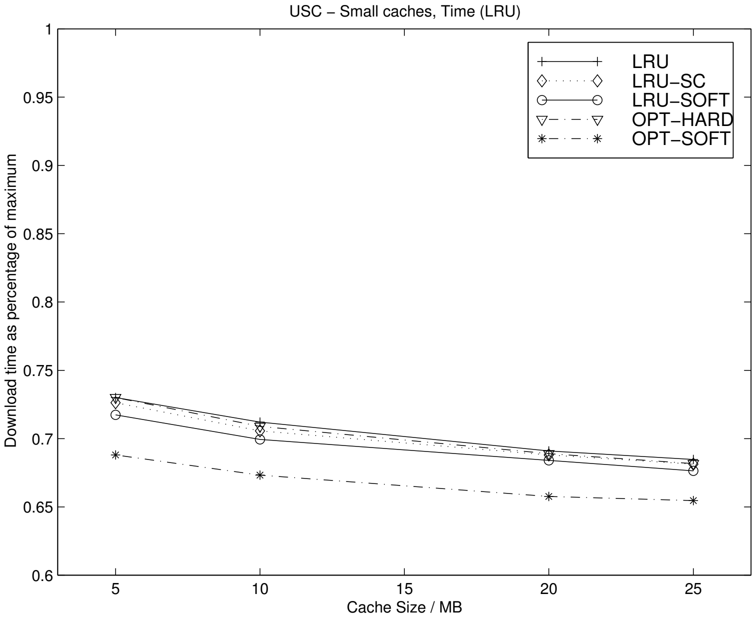


Figure 8.3: USC-trace, Small caches, Time (LRU)

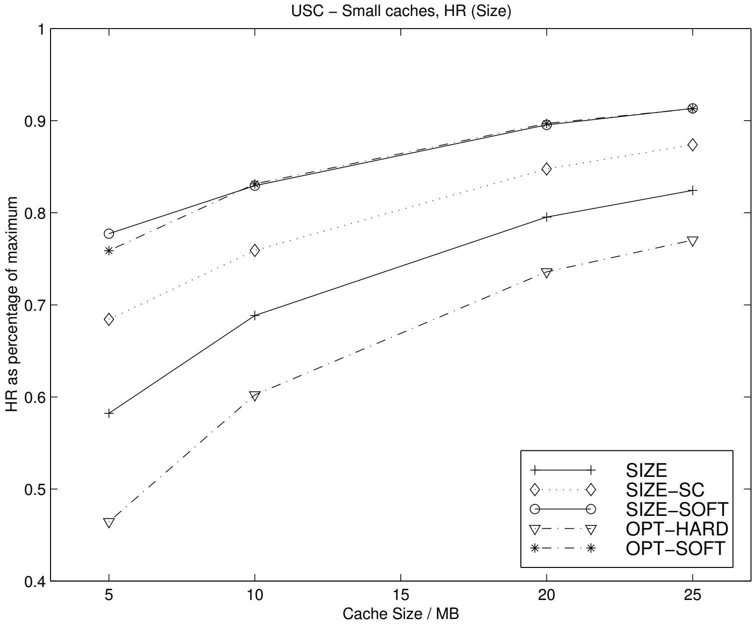


Figure 8.4: USC-trace, Small caches, HR (Size)

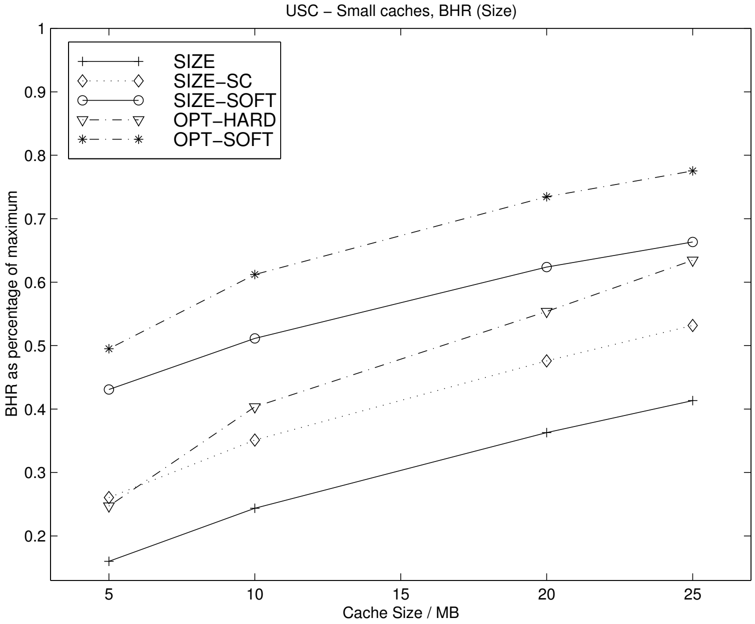


Figure 8.5: USC-trace, Small caches, BHR (Size)

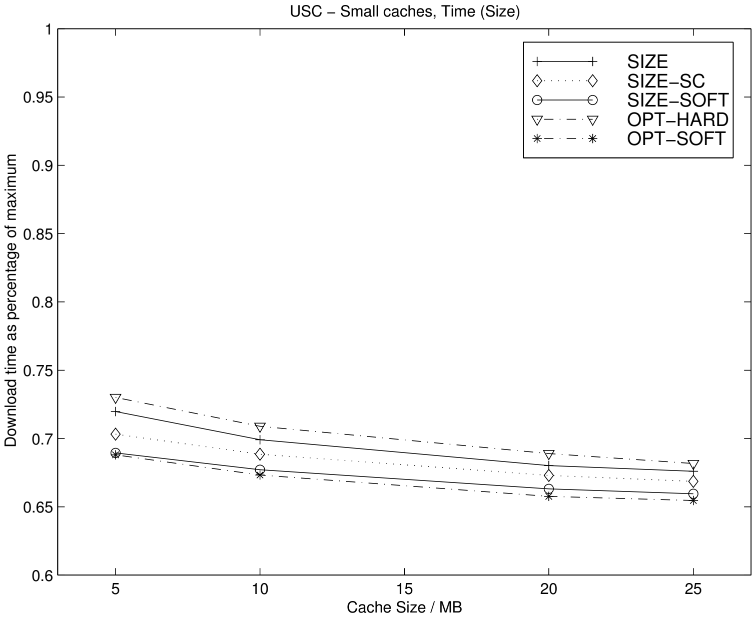


Figure 8.6: USC-trace, Small caches, Time (Size)

8.3.2 USC-trace, Large caches

The plots of the large caches using LRU-policies are shown in figures 8.7–8.9.

Overall, the results are similar to the results obtained on the smaller cache sizes. Due to the larger cache size and the subsequent ability of the cache to hold most or even all of the cacheable traffic, the differences among the replacement policies have become almost negligible.

The largest gain on hit-rate is the 13 % advantage of the soft optimal policy over stock-LRU for the smallest cache size. About one third of this is attributable to the improved replacement policy, but the rest is brought about by recoding. As the cache size grows, all replacement policies tend to the maximum attainable hit-rate.

The same is true for byte hit-rate as well, but the largest gain is only around 7 %, again for the smallest, 50 MB cache. Large caches make it also next to impossible to reduce the download time which is clearly demonstrated by figure 8.9. The largest difference in download times is 3 % for the smallest cache, all other situations resulting in almost no gain at all.

For the Size-based replacement policies, depicted in figures 8.10–8.12, the results follow those of the LRU-based policies.

Hit-rates for all policies are close to the maximum, and the differences are small. On byte hit-rate, the differences among replacement policies come into a clearer view. The soft optimal replacement policy has a 55 % better performance than the normal Size-policy for the 50 MB cache, and this difference, albeit becoming smaller as the cache size grows, remains rather large for other cache sizes. The only exception is the 400 MB cache, which is clearly sufficient to hold all of the cacheable traffic for this trace.

Overall, the results for both small and large caches on the USC-trace indicate that the more there is recoding, the better the performance. The optimal replacement policy with simulated soft access is always better than the same policy in its hard version. Likewise, policy with full recoding is always better than second chance which in turn is always better than the normal, non-recoding version. This holds true for both the LRU-based and Size-based experiments.

Looking at the big picture, the best policy is the soft optimal policy but in some cases the full recoding version of the base policy has a slight advantage.

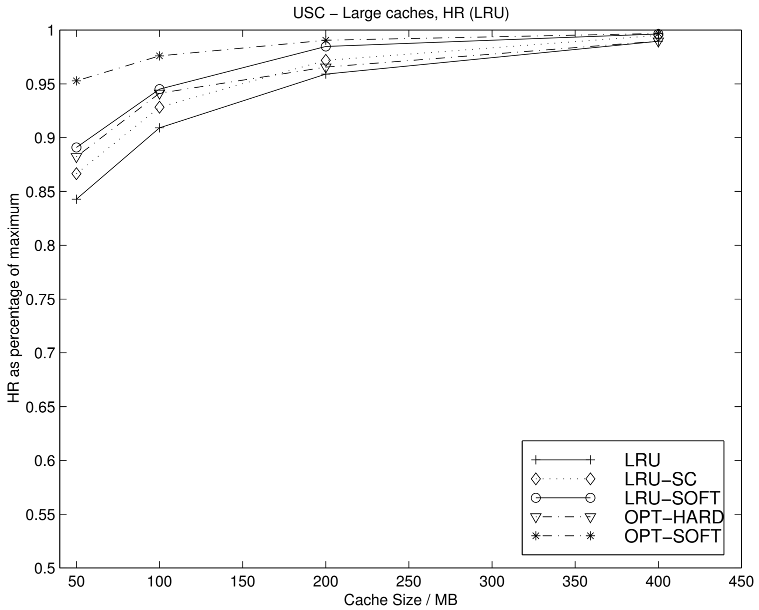


Figure 8.7: USC-trace, Large caches, HR (LRU)

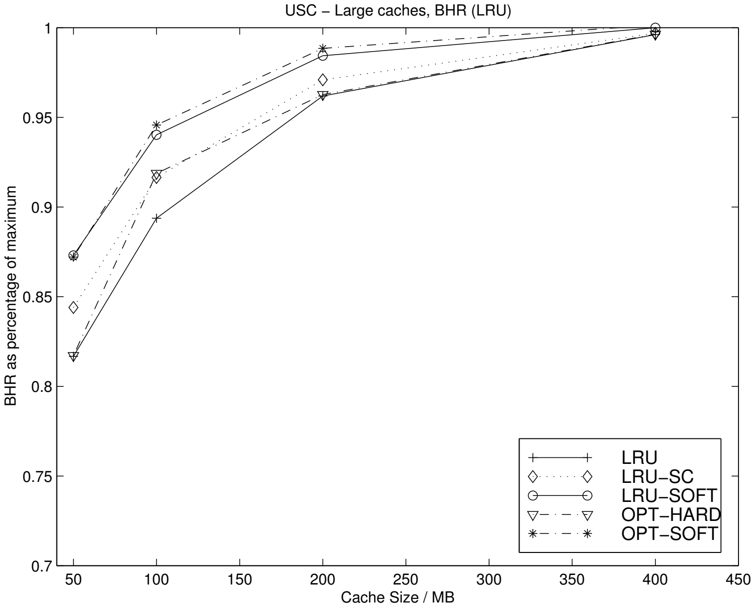


Figure 8.8: USC-trace, Large caches, BHR (LRU)

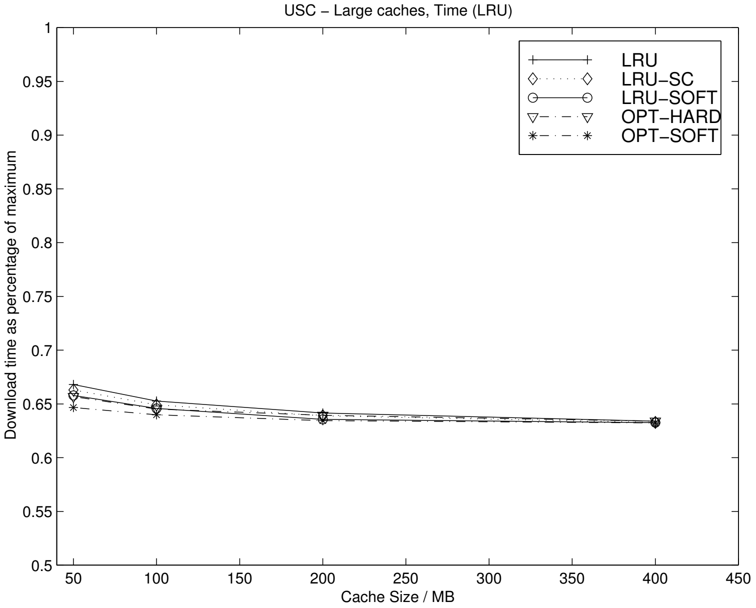


Figure 8.9: USC-trace, Large caches, Time (LRU)

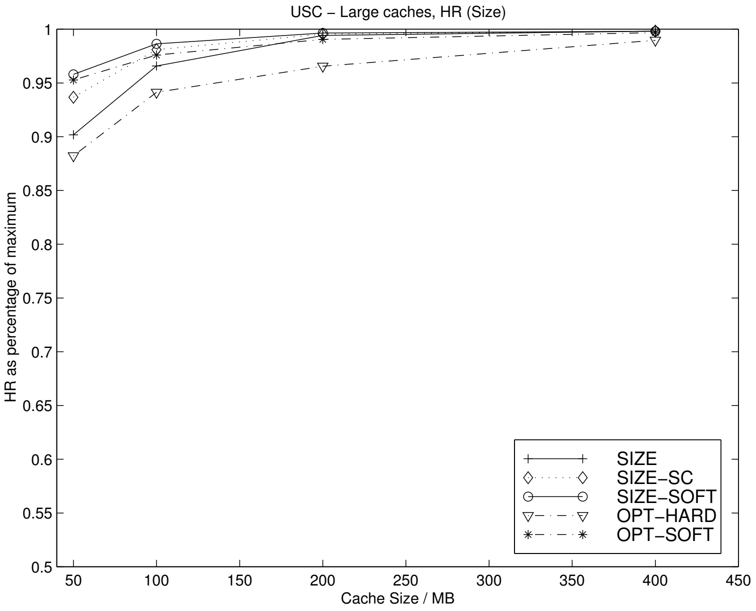


Figure 8.10: USC-trace, Large caches, HR (Size)

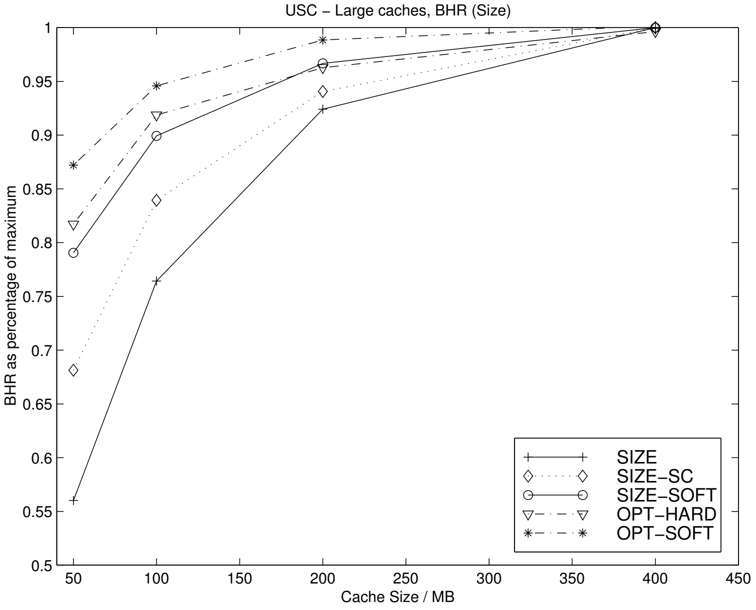


Figure 8.11: USC-trace, Large caches, BHR (Size)

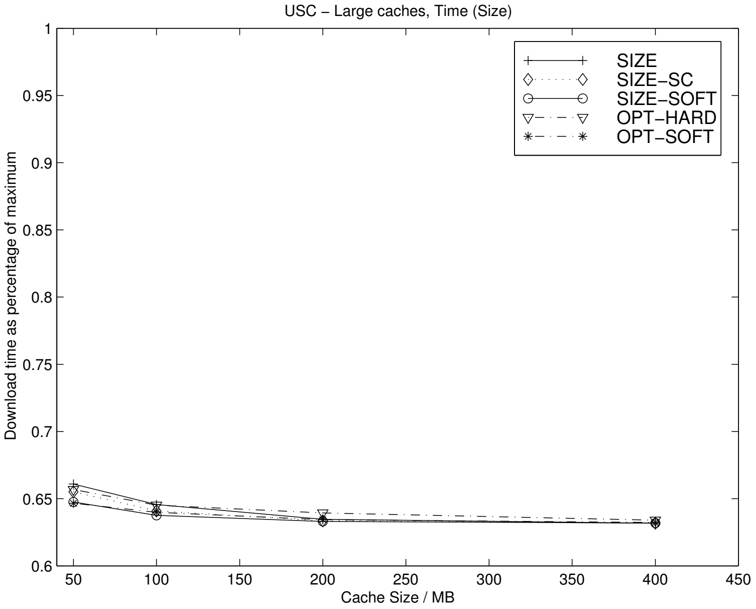


Figure 8.12: USC-trace, Large caches, Time (Size)

8.3.3 NLANR-trace, Small caches

Figures 8.13–8.15 present the plots from experiments on small caches using the NLANR-trace with LRU as the base policy.

The effects of the low locality of reference inherent in this high-level trace are clearly visible. Although overall performance is not bad, the differences among the replacement policies are practically speaking non-existent.

All three LRU-variants have identical performance in all three metrics, except for some random fluctuations.

The optimal replacement policy does not perform as well as the standard policies. This is also caused by the low locality of reference. Because of the small caches sizes and because most objects get referenced only once, the replacement criterion is essentially the inverse of the bandwidth to the server. This criterion would try to keep documents from servers behind a slow connection without considering any other factors, an intuitively illogical criterion.

It is also evident that the recoding version of the optimal policy is worse than the non-recoding version. This is caused in part again by the low locality of reference but also in part by the way soft accesses were estimated. Because this trace had the smallest average bandwidth (2.5 bytes per millisecond) the 0.1 which is added to recoded objects to simulate soft access has much more effect than in the USC-trace where the average bandwidth was 4.7 bytes per millisecond.

Therefore, since the recoded objects have their access probabilities increased, they become more important, even though the low hit-rate does not justify this behavior.

The results with the Size-policy and small caches are shown in figures 8.16–8.18.

Here recoding has more of an advantage. The gains in hit-rate are between 7 % and 28 % and between 23 % and 116 % in byte hit-rate. These gains are due to the recoding of large objects. Because Size-policy keeps small objects and throws away large objects, a recoding cache with Size-policy would recode these larger objects. The results suggest that, in this case, this is a good thing.

The performance of both of the versions of the optimal policy is no match for the Size-policy for the reasons discussed above.

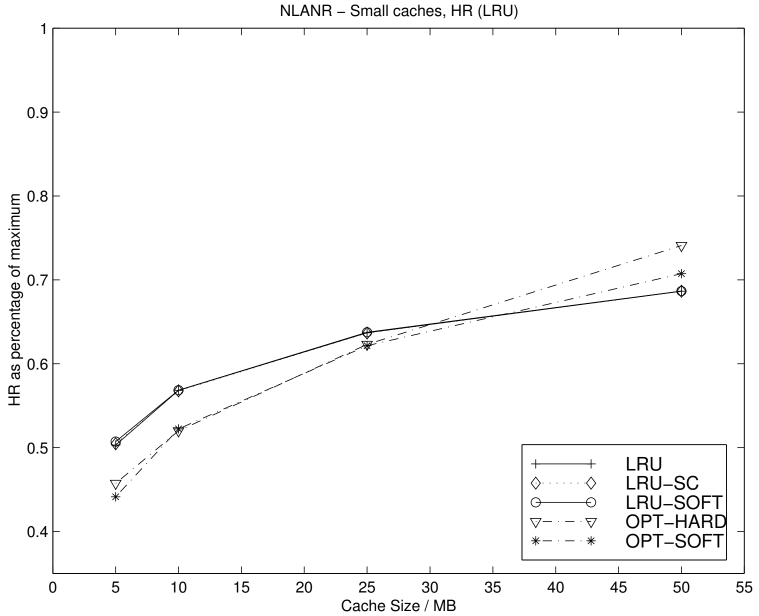


Figure 8.13: NLANR-trace, Small caches, HR (LRU)

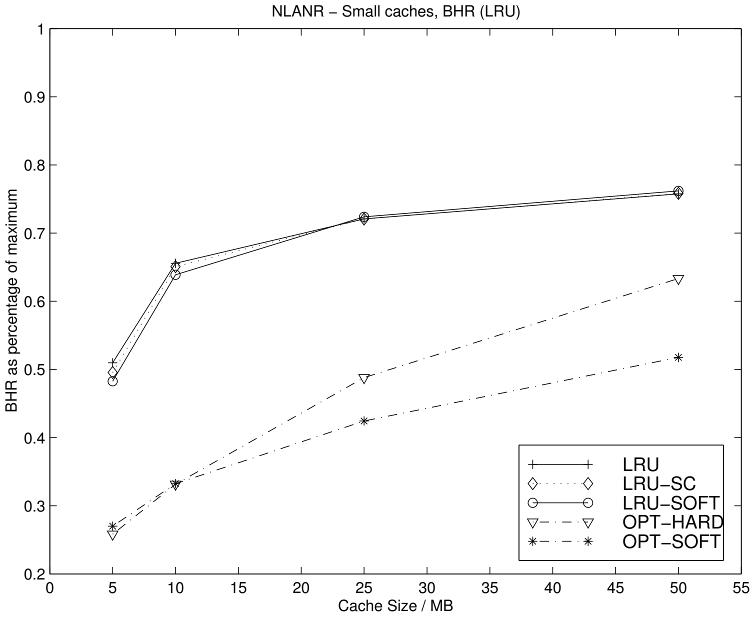


Figure 8.14: NLANR-trace, Small caches, BHR (LRU)

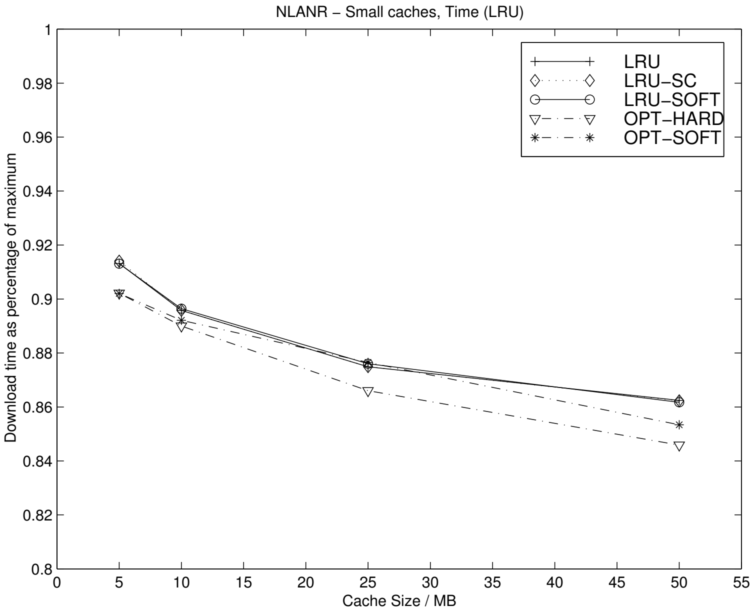


Figure 8.15: NLANR-trace, Small caches, Time (LRU)

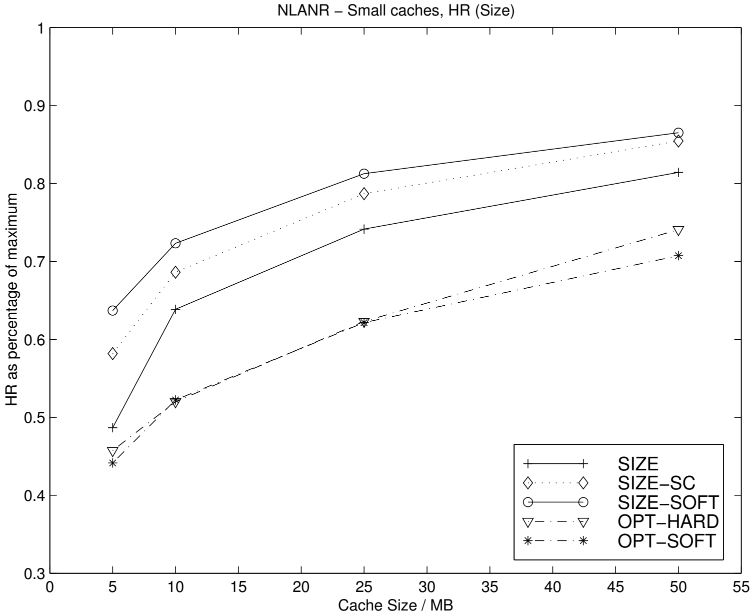


Figure 8.16: NLANR-trace, Small caches, HR (Size)

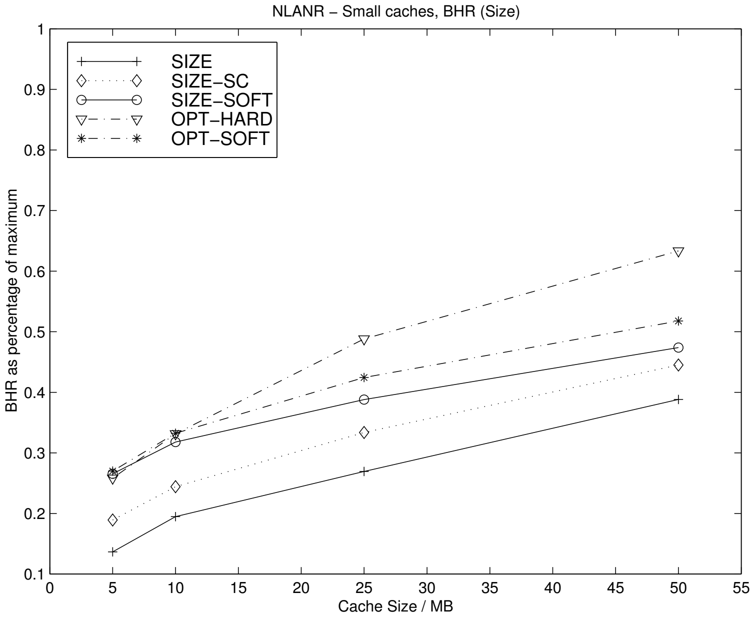


Figure 8.17: NLANR-trace, Small caches, BHR (Size)

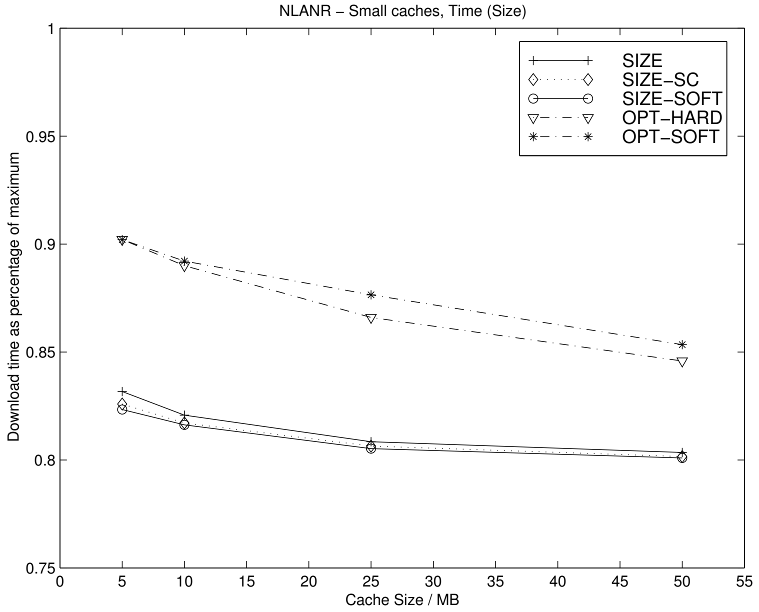


Figure 8.18: NLANR-trace, Small caches, Time (Size)

8.3.4 NLANR-trace, Large caches

Figures 8.19–8.21 show the results from the experiments with LRU-based policies on large caches.

As with the smaller cache sizes, the LRU-variants perform almost equally well. The only difference is the full recoding policy which has a slight gain on the largest cache sizes (600 and 800 MB). Even though the hit-rate is still 5 % shy of the maximum, the byte hit-rate is at maximum and therefore not much real-world gain could be expected for even larger caches.

The optimal policies suffer again from the effects of the low locality of reference and the estimated soft accesses. Their strongest suit is download time where it catches up to LRU.

As with the USC-trace, the differences in performance become extremely small as the cache size grows.

The Size-based policies for these cache sizes are shown in figures 8.22–8.24.

The results agree with those obtained from the experiments on small caches with this trace. The differences among the LRU-policies are greater and always to the advantage of more recoding in favor of less or no recoding. It is interesting to note, that while the recoding version of LRU achieved 100 % of the maximum byte hit-rate, the recoding variant of the Size-policy can get up to only 80 % of the maximum. This is directly related to the Size-policy's keeping of small objects and throwing away of large objects, therefore the large gains in byte hit-rate associated with these types of objects become unattainable.

It is also interesting to note, that although the optimal policy is designed for minimizing the average download time, all of the Size-policies have a slightly shorter download time in this experiment.

Overall, the results on the NLANR-trace are somewhat disappointing. Because of the low locality of reference, the differences between replacement policies almost disappear. This is because recoding does not change the fundamental approach of the replacement algorithm. If the base algorithm is LRU, the recoding algorithm will also be LRU. While recoding does increase the lifetime of some objects, in this world where objects are mostly referenced only once, this approach can do only little to improve performance.

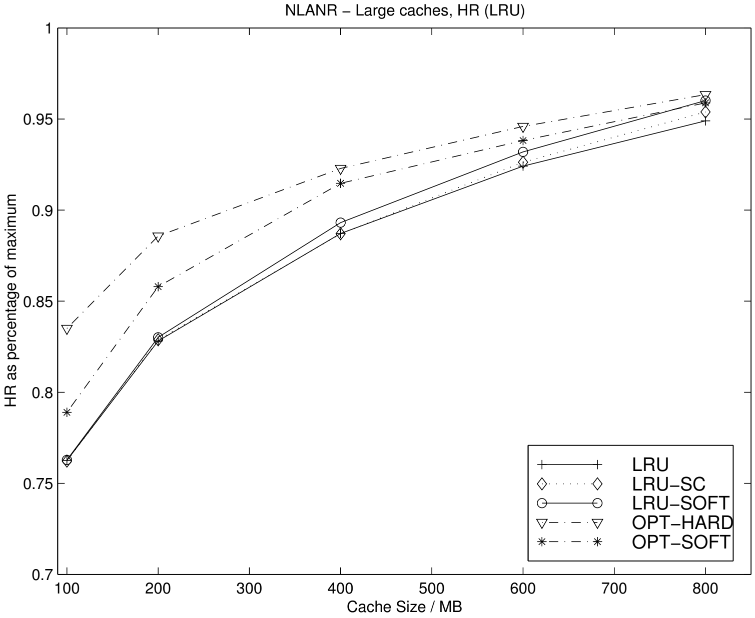


Figure 8.19: NLANR-trace, Large caches, HR (LRU)

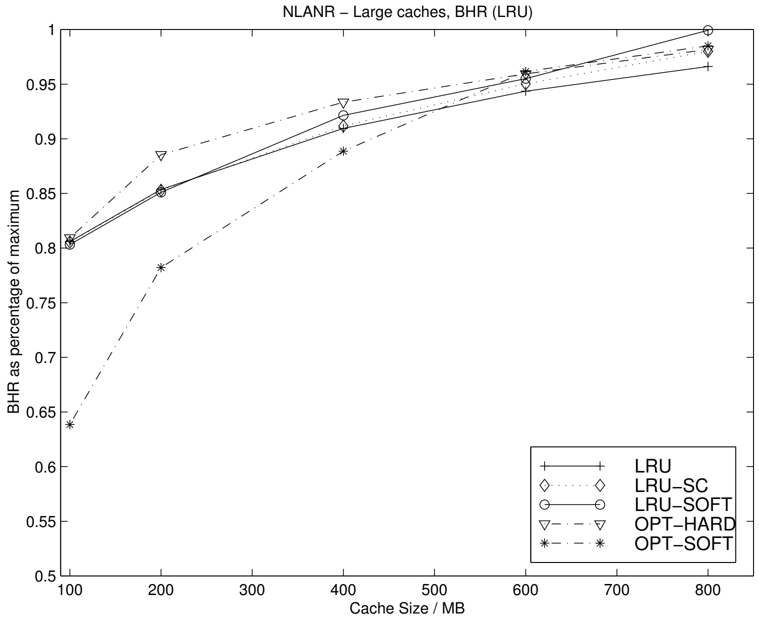


Figure 8.20: NLANR-trace, Large caches, BHR (LRU)

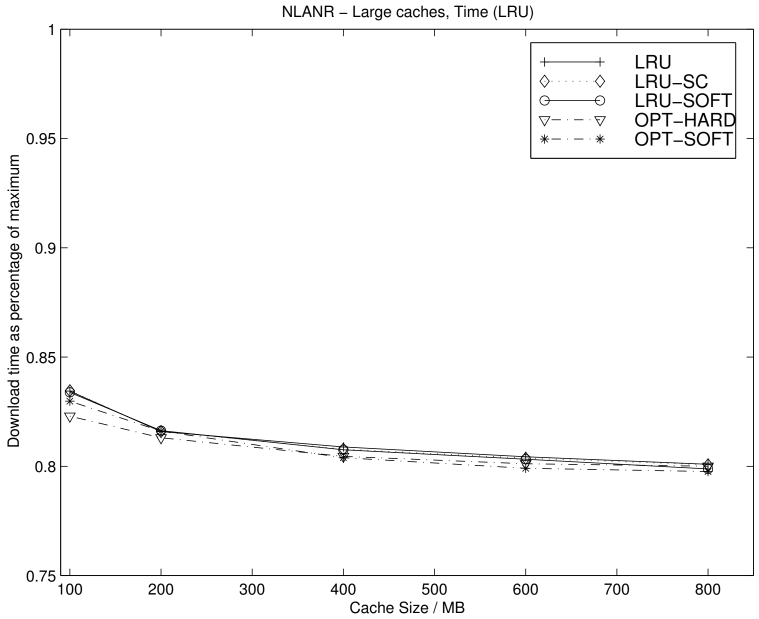


Figure 8.21: NLANR-trace, Large caches, Time (LRU)

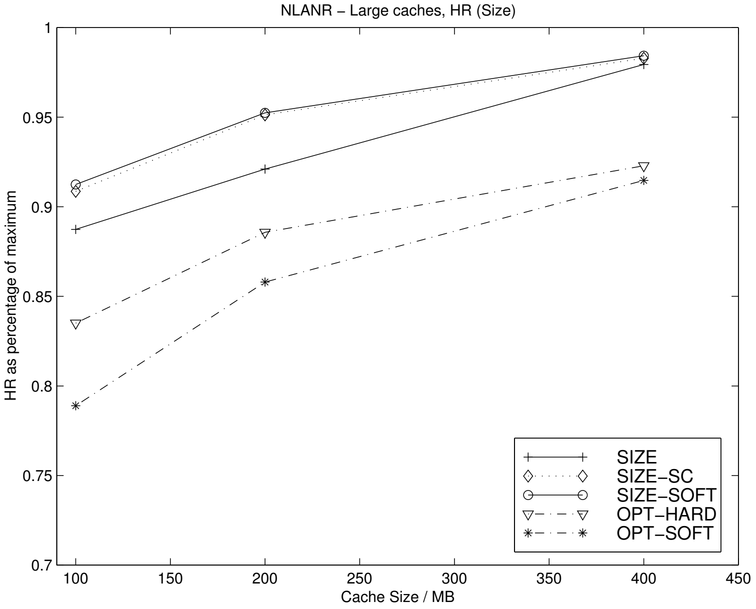


Figure 8.22: NLANR-trace, Large caches, HR (Size)

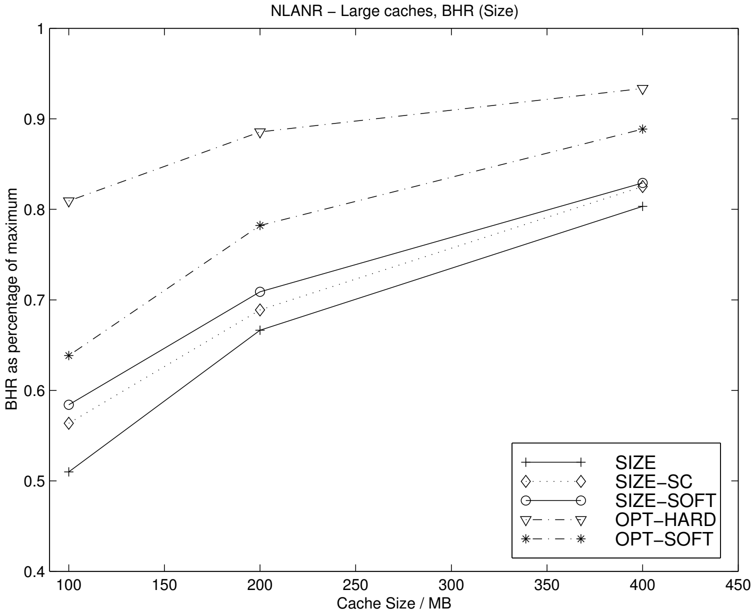


Figure 8.23: NLANR-trace, Large caches, BHR (Size)

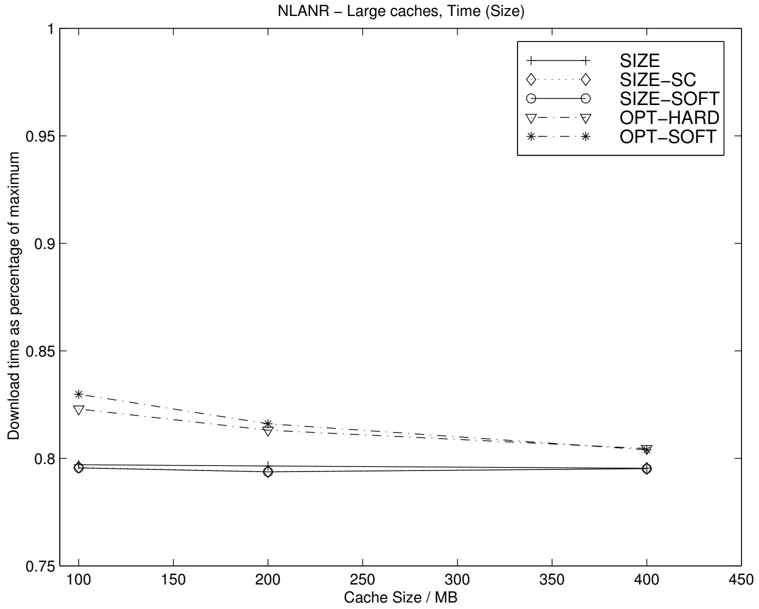


Figure 8.24: NLANR-trace, Large caches, Time (Size)

8.3.5 Other-trace, Small caches

The results from the experiments with the LRU-policies and small caches on the Other-trace are shown in figures 8.25–8.27.

The results bear a strong resemblance to those obtained on the USC-trace with small caches. The recoding versions of the policies are better than the non-recoding versions, although for the LRU-variants, the differences are small.

For the optimal policies, the soft version beats the hard one clearly on hit-rate and download time, the difference on byte hit-rate being somewhat smaller. On byte hit-rate, the problems with the access probability estimation can also be seen, as the optimal policies have a performance slightly lower than that of the LRU-variants. This was not the case with the USC-trace, even though it is similar to this trace on the maximum attainable performance.

The reason behind this difference is, that although the maximum performance is roughly the same, the much larger size of this trace puts the hits potentially much further apart from each other than in the USC-trace. In a small cache this means that the object is hit less times while in the cache and therefore the reference count will not reflect the true access probability.

On the download time, however, the soft optimal policy is a clear winner. Although the gain is only around 3.5 %, this gain is constant with respect to cache size.

The experiments on the Size-policies and small caches are plotted in figures 8.28–8.30.

Once more, recoding policies have a clear advantage over the standard policies. Full recoding LRU shows gains of between 27 % and 80 % on hit-rate and between 92 % and 266 % on byte hit-rate! On download time, the gains are smaller, around 4 % but constant over different cache sizes.

The soft optimal replacement policy is on almost all situations the best performing policy. It has an enormous 366 % percent performance gain over basic LRU on byte hit-rate in the smallest cache, although one half of this is due to the replacement algorithm and only one half due to recoding.

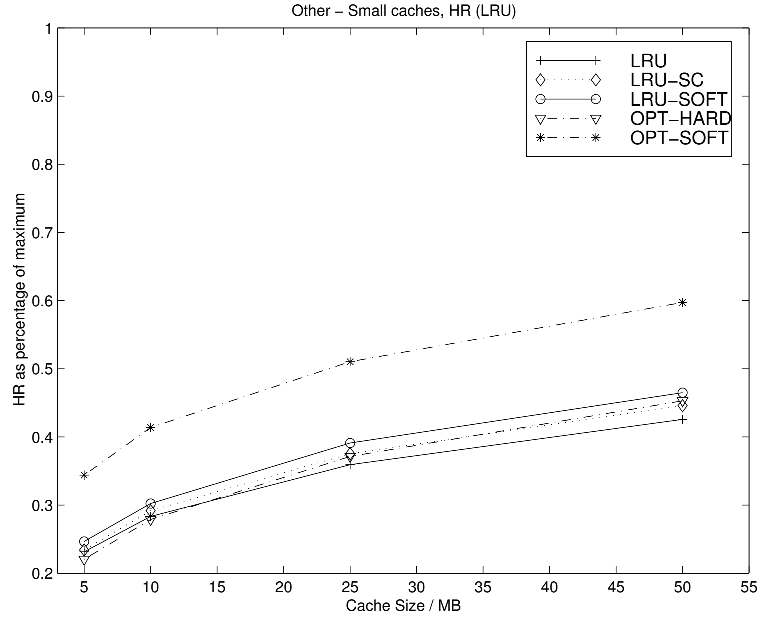


Figure 8.25: Other-trace, Small caches, HR (LRU)

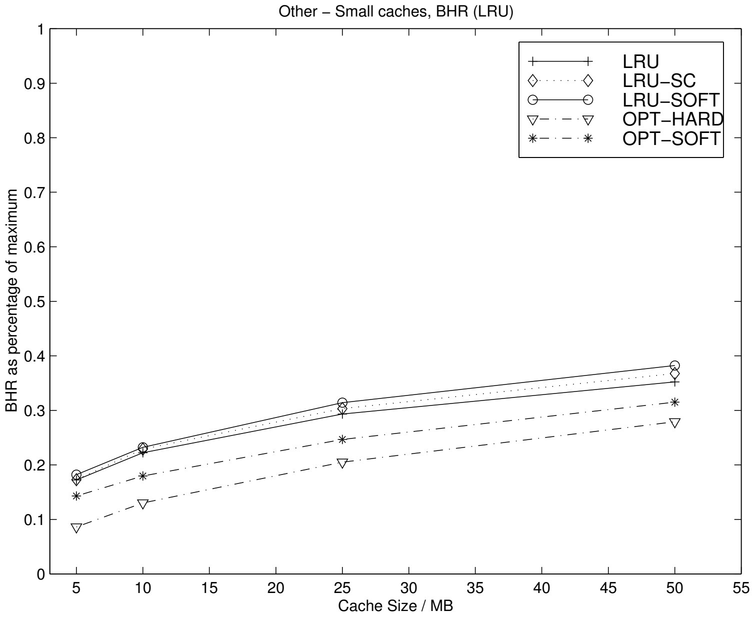


Figure 8.26: Other-trace, Small caches, BHR (LRU)

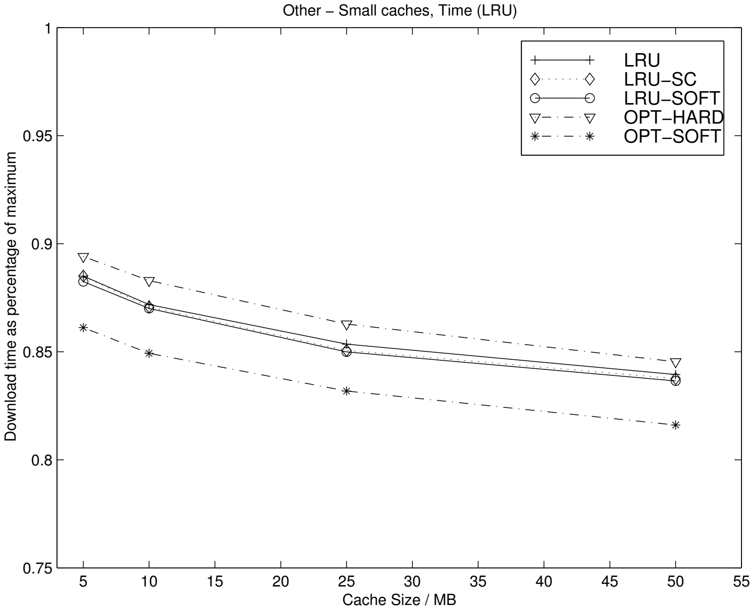


Figure 8.27: Other-trace, Small caches, Time (LRU)

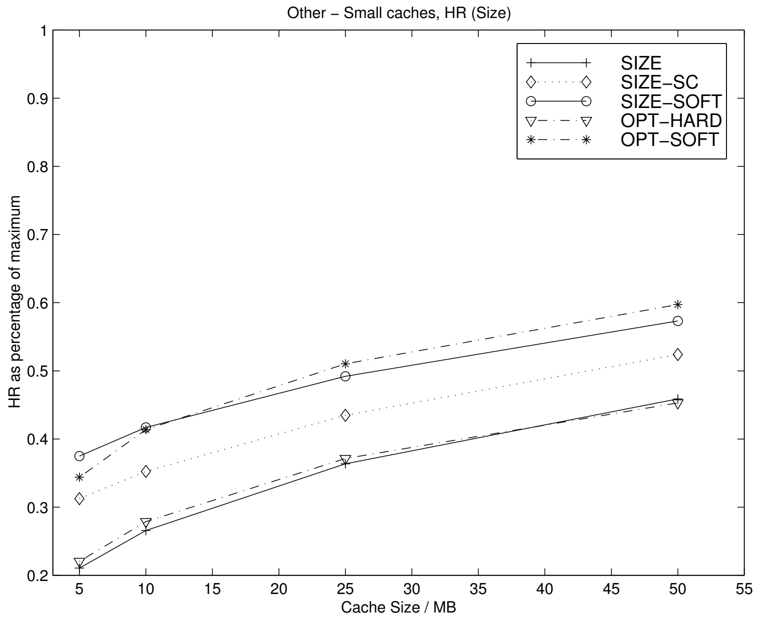


Figure 8.28: Other-trace, Small caches, HR (Size)

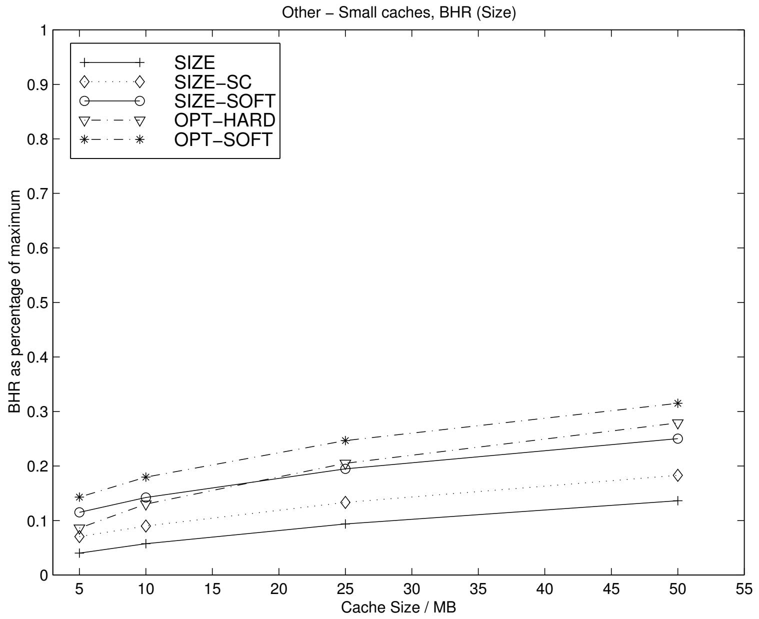


Figure 8.29: Other-trace, Small caches, BHR (Size)

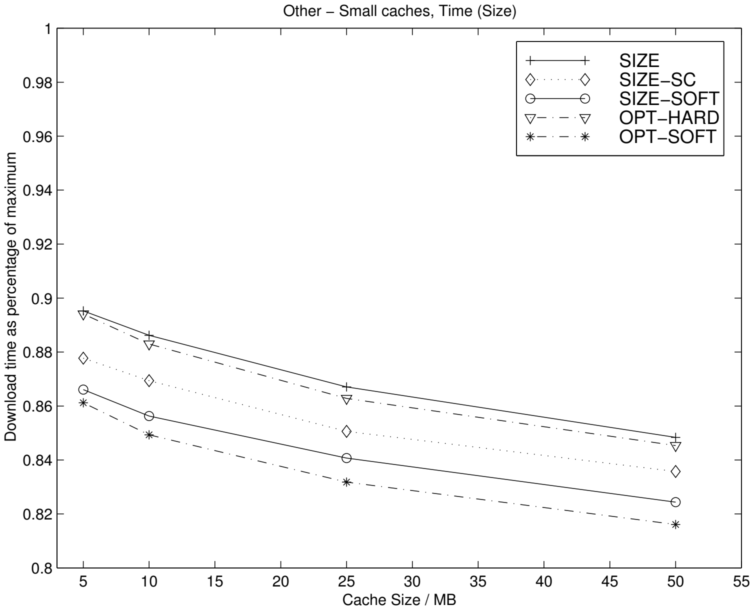


Figure 8.30: Other-trace, Small caches, Time (Size)

8.3.6 Other-trace, Large caches

Because of memory shortage, only some of the planned runs were possible to run on the Other-trace with large caches. Most notably, the soft optimal policy (OPT-SOFT) could not be run in any of the configurations because there was not enough available memory to hold information about all the objects the policy would have required. For the same reason, none of the Size-based policies could be run.

The results for the experiments that could be run, i.e. all the other four LRU-based replacement policies, are shown in figures 8.31–8.33.

The results are similar to those obtained on the USC-trace on large caches. Recoding has a definite advantage over the standard LRU with a around 6 % gain on hit-rate and around 5 % gain on byte hit-rate. These gains are independent of cache size.

The hard version of the optimal policy beats LRU by about 9 % on both hit-rate and byte hit-rate. The results from all the previous experiments suggest, that the soft optimal policy which estimates the soft accesses would perform even better than the hard version.

Overall, the results of the Other-trace are similar to the ones obtained on the USC-trace. Recoding replacement policies have clear advantage over the normal policies, and the optimal policy yields even higher gains. These results are most likely explained by the nature of the traces. Both traces are from proxies close to the users, in both cases the proxy was either a replacement for browser cache or the first proxy after browser cache. This means that there is a high locality of reference inherent in the trace and therefore caching pays off. Especially in the small caches where memory is at premium, the memory saving effects of soft caching can reap great rewards

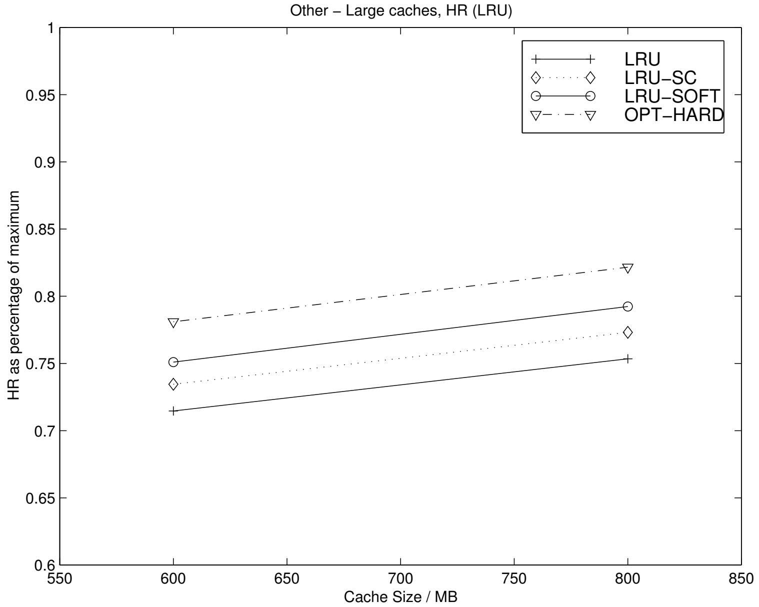


Figure 8.31: Other-trace, Large caches, HR (LRU)

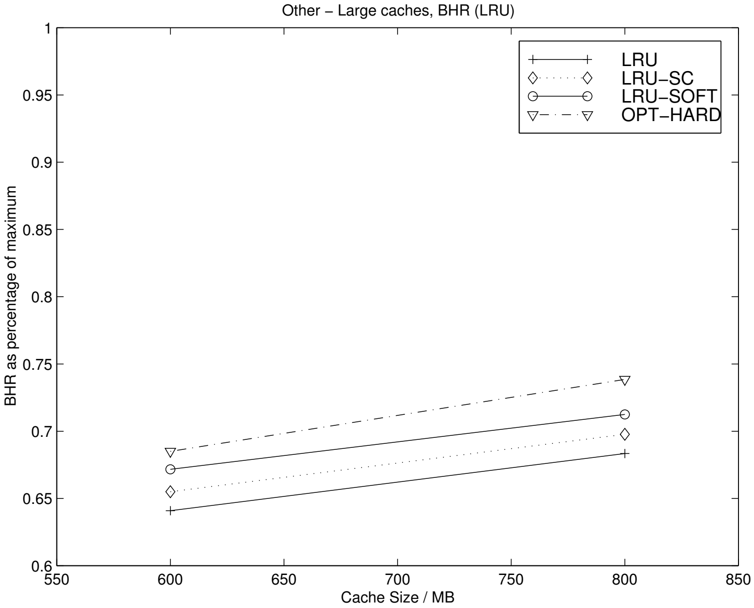


Figure 8.32: Other-trace, Large caches, BHR (LRU)

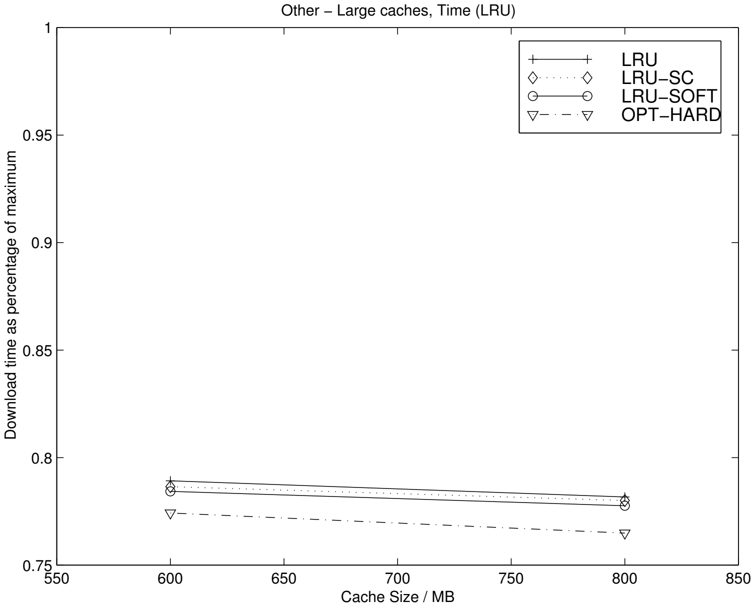


Figure 8.33: Other-trace, Large caches, Time (LRU)

8.4 Summary

The results from the experiments were presented in this chapter.

First, the results on the refreshment policy and recoding level showed the best strategies, linear refreshment and a large number of recoding levels. The experiments on the replacement policy showed that recoding policies have a better performance and that the optimal replacement policy yields in most cases the best performance.

Overall, soft caching and recoding was found to yield clearly higher performance and in some cases even remarkable gains are possible. The results will be discussed in more detail in the next chapter.

Chapter 9

Discussion

The results on the previous chapter show a clear advantage to systems using soft caching over standard systems. This was found to be the case for almost all situations explored. This chapter will analyze some of the results in more detail as well as provide directions for future work.

9.1 Locality of Reference

When comparing the results on the NLANR-trace with the other two, it is obvious that the more there is locality of reference in the traffic seen by the cache, the larger the possible gains. With the NLANR-trace, the differences among the LRU-based policies were almost non-existent. With the Size-based policies, recoding has an advantage. This advantage is due to the soft cache's recoding of large objects. This way, they stay in the cache and since there is evidently a later hit on some of them, the overall performance is increased.

This advantage is made greater by keeping the recoded objects and recoding them also later when the situation arises. This is demonstrated by the performance gap between the full recoding policy and the second chance policy, which recodes a recoded object only when it has been hit since the last recoding. With a low locality of reference, this might not often be the case, and the object gets its second hit much later.

The other two traces which possess a much higher locality of reference, show more benefits for soft caching. This is especially true for the small, memory constrained caches where recoding frees up valuable space, but also in the larger caches, the advantages are undeniable. Again, the Size-based policies show a much bigger advantage due the the reasons discussed above, but the LRU-based policies show also non-negligible gains.

An important discovery is also the consistency of the gains. This means that soft caching and recoding give an advantage, regardless of the cache size. When the cache is large enough to hold all of the cacheable traffic, this difference disappears from a practical standpoint, but in all other situations, soft caching has the upper hand.

9.2 Small Caches and Recoding

In the smaller caches, the advantages of recoding are at the largest. This is because a small cache can hold only a small fraction of the cacheable traffic and since recoding helps to reduce object size. This means that the cache can hold more objects.

As discussed in chapter 4, a typical scenario where a small, memory constrained cache is used, is at the client. Although it is possible to change the default cache size of the browser, it is not at all certain that the majority of users would be able to do this. And even if that were the case, most people would probably set the cache size to a few tens of megabytes at maximum.

The results from the previous chapter show, that for a small cache, it would be advantageous to use soft caching and a Size-based replacement policy.

The USC- and Other-trace come from proxies near the users, and it would seem, in contrast with the NLANR-trace, that caches nearer to users see a significantly higher locality of reference in the traffic. It is safe to assume, that a cache at an active user's browser would see similar traffic to the ones in the USC- and Other-traces.

9.3 Optimal Replacement Policy

The optimal soft replacement policy was found to yield mixed results. On the NLANR-trace, with a low locality of reference, the estimator used for the access probability (reference count) is not accurate enough. This is because is is reset every time an object is released from the cache. However, keeping reference counts for all objects ever seen by a cache would simply consume too much memory.

As the cache size grows, the estimator becomes better and the results improve. If the traffic seen by a very small cache had only a low locality of reference, it might be useful to include other parameters, such as the last reference time, to the access probability estimator.

In the other two traces the higher locality of reference results in a better estimate of the relative popularities of the objects. Therefore, the performance of the optimal replacement policy improves greatly.

To make the optimal policy to make recoding decisions, the incoming traffic must consist of soft accesses. If this is not the case, the policy degrades into a normal, hard replacement policy. Because no modern browser allows user to set soft access preferences, all accesses in logfiles are hard accesses.

In the simulations this problem was circumvented by simulating the soft accesses by adjusting the reference count of the object. The choices made to simulate the soft accesses are discussed below.

9.4 Choice of Parameters

In a soft caching system there is a certain number of choices that must be made regarding the internal behavior of the system. These choices concern the number of recoding levels and how to treat recoded objects.

Every layered object type has a certain number of layers. Naturally, the cache could follow the internal structure of each object, but this would complicate the treatment of objects since each object type would have to be treated differently. This would create a need for additional processing time at the cache, which in a busy cache could be too much to ask.

If the cache treats each object as having a logical structure, each object is treated in the exact same way, thus speeding up the decision-making process. The actual mapping from the logical structure to the real physical layout of the object can be made in a separate process handling the details of the recoding. This is the approach of the current soft caching implementation.

It was found to be more advantageous to have more layers in the logical structure than less layers. However, if an object has a certain number of layers, like 4 in interlaced GIF, it is very hard to treat such objects as logically having more layers than they really do. This needs to be taken into consideration when choosing the logical structure for objects.

Normal, hard replacement policies cannot handle recoded objects because they treat objects as immutable. Therefore they need modifications in order to avoid undesirable effects when handling recoded objects. In LRU-based policies, the objects access time has to be modified, because otherwise the same object would get chosen for removal again and again. A strategy that uses the number of layers still left in the object was found to have a better performance than a strategy that simply sets the access time to current time.

This is because the latter strategy prolongs the object's lifetime in the cache more, and if the object is not referenced again, it is taking up space uselessly in the cache. The linear strategy does not favor heavily recoded objects that much, and therefore if the object is not referenced, it gets removed faster.

As far as the Size-policies and the optimal replacement policy are concerned, only one type of modification was studied. The performance of both of the modifications was found to be good, yet, the effects of different modifications should be studied in more detail.

The choice for the simulated soft accesses needs to be weighed carefully. In the absence of real data on user preferences for different resolutions, an educated guess is called for. The choice used, to add 0.1 to the reference count, does increase the priority of the object, but does not completely throw off the normal probability estimates.

9.5 Savings on Download Time

The savings on average download time were not found to be very different from one replacement policy to another. The main reason behind this, is that the measured value includes the time needed to fetch an object from the origin server when the object is referenced the first time. This time is always spent, regardless of the replacement policy.

Another aspect of the download time is that the measured times reflect the time it would take to download the object to the proxy from which the trace originates. To this value should be added the time it would take to transmit the object to the client.

Using the values of the average object size from tables 7.1, the download times from the proxy to client on different link speeds for the USC-trace are shown in table 9.1.

Speed (kbps)	**Time (sec)**
14.4	5.3
28.8	2.6
33.6	2.2
56.7	1.3
T1	0.05

Table 9.1: Download time of average object on different links

From table 7.2 on page 41, the download time for a no-cache situation is 2.4 seconds. Therefore, for speeds less than 33.6 kbps, caching can do very little to help reduce the download time.

Even though in theory, no gains in download time seem possible, the reality is different. Because the network can be congested, some packets can be lost along the way. If the packets are going straight to the client behind a slow connection, it takes a longer time to discover that a loss has occurred.

A cache on the other end of the dial-up link acts as a buffer for changes in network since it can react faster to congestion-related losses. Then the cache can send the object as a whole over the dial-up link, where congestion does not cause a problem.

If the soft cache is situated at the client behind a dial-up link, a cache should therefore be used on the other end of the link for the above reason. As far as the soft cache in this setup is concerned, all advantages on hit-rate and byte hit-rate will translate into gains on download time. This is because if an object is found in the local cache, it does not need to be transmitted over the slow link, therefore resulting in large gains on download time.

9.6 JPEG

The advantages of the JPEG-format concerning recoding were presented in chapter 5. The biggest advantage is that the standard defines a progressive mode of encoding and that this mode is widely supported in modern programs.

The way the information is packed into the progressive mode makes it possible to recode images quite heavily while still retaining most of the original quality.

Because of the way the widely used IJG-library implementation of progressive JPEG spreads the information into 10 layers, the implementor of a soft caching system is left with a great latitude in choosing the right number of layers to use inside the cache.

9.7 Future Work

Possible directions for future work in soft caching include:

- Better estimates for soft accesses.
  Currently the value 0.1 used in simulating the soft accesses with the optimal replacement policy is completely arbitrary and not based on any hard facts. To get accurate data on possible parameter values, a more complete implementation of a soft caching system is needed. The current implementation treats only hard accesses.

- Implement soft access.
  In connection with the previous point, browsers should be upgraded to use soft access where applicable. The simplest way would be to simply make a request to a part of the object if the object looks like a progressive, recodable object.

- Study recoding of other objects.
  Currently, only JPEGs are being recoded in the real implementation. Most of the simulations were based on the assumption that also GIFs are recodable. The most promising candidates for recoding are video objects and other image formats, such as GIFs and PNGs.

- Modify a client to do soft caching.
  Because of the large gains in small caches observed in the simulations, recoding should be implemented at a client to take full advantage of these possibilities. In addition, given the processor capacities installed

in modern client machines, recoding would be a very light operation on such machines.

9.8 Summary

The results from the simulations were explored deeper in this chapter. The causes and effects of different parameters and network setups were considered.

The average download time was found to be much shorter than the time it would take to transmit the average object over typical modem links. The only advantage a cache can present in this situation is acting as a buffer between the client and the network.

Chapter 10

Conclusion

This thesis has investigated soft caching. Soft caching differs from traditional caching in that in a soft caching system objects can be recoded. Recoding is an operation where information is removed from an object. If the object is a layered object which can sustain information loss, e.g. a progressive JPEG-image, the act of removing a layer of information is usually not visible to a viewer.

Standard cache replacement algorithms must be modified in order to make them take advantage of recoding and layered objects. These modifications are simple and are aimed at adjusting the replacement criterion (access time, size, etc.) to factor in the effects of recoding.

The JPEG-format is well suited for soft caching since it includes a progressive mode of encoding. The progressive mode has a great potential of recoding savings, since a lot of the bits are dedicated to visually unimportant information. Also, progressive JPEG-images tend to be smaller in size than the corresponding sequential JPEG-images.

Different cache replacement algorithms were studied in simulations using actual traces from proxy caches. The algorithms studied were Least Recently Used and Size. These were modified to accommodate soft caching and two recoding strategies, second chance and full recoding, were used.

The results show that a cache implementing soft caching performs better in most situations under all the three metrics, hit-rate, byte hit-rate and average download time. Using the Size-policy, the potential for gains is much greater because recoding helps keep large objects in the cache, objects which would otherwise be removed.

Using second chance recoding, the performance was slightly better than with the normal algorithm, but inferior to that of the full recoding algorithm. A specialized soft caching algorithm designed to minimize the average download time was also studied and it was found to be among the top performers.

The biggest relative gains were obtained in very small caches which correspond to typical client cache sizes. It would therefore be advantageous to implement soft caching at the client in order to fully maximize the benefits of caching.

Given that both of the normal replacement policies studied, LRU and Size, present advantages in soft caching, even better results could be obtained by combining these two with the optimal replacement policy. This matter is, however, left for further study.

Bibliography

[1] T. Berners-Lee and D. Connolly. *RFC 1866: Hypertext Markup Language — 2.0*, November 1995.

[2] T. Berners-Lee, R. Fielding, and H. Frystyk. *RFC 1945: Hypertext Transfer Protocol - HTTP/1.0*, May 1996.

[3] T. Berners-Lee, R. T. Fielding, H. Frystyk, J. Gettys, and J. Mogul. *RFC 2068: Hypertext Transfer Protocol - HTTP/1.1*, January 1997.

[4] J.-C. Bolot and P. Hoschka. Performance engineering of the world wide web: Application to dimensioning and cache design. *The World Wide Web Journal*, 1(3):185–195, 1996.

[5] J.-C. Bolot, S. M. Lamblot, and A. Simonian. Design of efficient caching schemes for the world wide web. In *Proceedings of ITC*, Washington, D.C., June 23–27, 1997.

[6] H.W. Braun and K. Claffy. Web traffic characterization: An assessment of the impact of caching documents from the NCSA's web server. In *Proceedings of Second International World Wide Web Conference '94*, October 1994.

[7] F. Carignano. Soft caching: Web cache management techniques for images. Internship report, Institut Eurécom, Sophia Antipolis, France, July 1997.

[8] A. Chankhunthod, P. Danzig, C. Neerdals, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *Proceedings of USENIX 1996 Annual Technical Conference*, pages 153–163, San Diego, CA, January 22–26, 1996.

[9] C. A. Cunha, A. Bestavros, and M. E. Crovella. Characteristics of WWW client traces. Technical Report TR-95-010, Computer Science Dept., Boston University, April 1995.

[10] P. Danzig, R. Hall, and R. Schwartz. A case for caching file objects inside internetworks. Technical Report CU-CS-642-93, University of Southern California and University of Colorado at Boulder, March 1993.

[11] A. Fox and E. A. Brewer. Reducing www latency and bandwidth requirements by real-time distillation. In *Proceedings of Fifth International WWW Conference*, Paris, France, May 1996.

[12] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 160–170, Cambridge, MA, October 1–5, 1996.

[13] N. Freed and N. Borenstein. *RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*, November 1996.

[14] Graphic, Visualization, & Usability Center's 8th WWW user survey. <URL:http://www.gvu.gatech.edu/user_surveys/>.

[15] CompuServe Inc. Graphics Interchange Format, version 89a, 1990. <URL:http://www.dcs.ed.ac.uk/home/mxr/gfx/2d/GIF89a.txt>.

[16] Independent JPEG Groups's software. <URL:ftp://ftp.uu.net/graphics/jpeg/>.

[17] ISO. *Call for Contributions for JPEG 2000 (JTC 1.29.14, 15444): Image Coding System*, March 1997. (work in progress).

[18] J. Kangasharju, Y. Kwon, and A. Ortega. Design and implementation of a soft caching proxy. In *Proceedings of 3rd International WWW Caching Workshop*, Manchester, UK, June 15–17 1998.

[19] J. Kangasharju, Y. Kwon, A. Ortega, X. Yang, and K. Ramchandran. Implementation of optimized cache replenishment algorithms in a soft caching system. In *Proceedings of IEEE Signal Processing Society Workshop Multimedia on Signal Processing '98*, Los Angeles, CA, December 7–9 1998.

[20] T. Kwan, R. E. McGrath, and D. A. Reed. User access patterns to NCSA's world wide web server. Technical Report UIUCDCS-R-95-1934, Computer Science Dept., University of Illinois, Urbana-Champaign, IL, February 1995.

[21] A. Luotonen and K. Altis. World wide web proxies. *Computer Networks and ISDN Systems*, 27(2):147–154, November 1994.

[22] MBone Information Web. <URL:http://www.mbone.com/>.

[23] A distributed testbed for national information provisioning. <URL:http://ircache.nlanr.net/>.

[24] Netscape Communications home page.
<URL:http://home.netscape.com/>.

[25] N. Niclausse, Z. Liu, and P. Nain. A new efficient caching policy for the world wide web. In *Proceedings of Workshop on Internet Server Performance (WISP'98)*, Madison, WI, June 1998.

[26] A. Ortega. *Optimization Techniques For Adaptive Quantization Of Image And Video Under Delay Constraints.* PhD thesis, Dept. of Electrical Engineering, Columbia University, New York, NY, 1994.

[27] A. Ortega, F. Carignano, S. Ayer, and M. Vetterli. Soft caching: Web cache management techniques for images. In *Proceedings of IEEE Signal Processing Society Workshop on Multimedia Signal Processing*, Princeton, NJ, June 25–27, 1997.

[28] A. Ortega, Z. Zhang, and M. Vetterli. A framework for optimization of a multiresolution remote image retrieval system. In *Proceedings of INFOCOM '94*, volume 2, pages 672–679, Toronto, Canada, June 1994.

[29] W. Pennebaker and J. Mitchell. *JPEG Still Image Data Compression Standard.* Van Nostrand Reinhold, 1994.

[30] PNG (Portable Network Graphics) specification, version 1.0, October 1996. W3C Recommendation,
<URL:http://www.boutell.com/boutell/png/>.

[31] RealNetworks home page.
<URL:http://www.real.com/>.

[32] P. Scheuermann, J. Shim, and R. Vingralek. A case for delay-conscious caching of web documents. In *Proceedings of Sixth International WWW Conference*, Santa Clara, CA, April 1997.

[33] J. M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, December 1993.

[34] Spyglass Prism.
<URL:http://www.spyglass.com/solutions/technologies/prism/>.

[35] Squid Internet Object Cache.
<URL:http://squid.nlanr.net/>.

[36] W. R. Stevens. *TCP/IP Illustrated, Vol. 1.* Addison-Wesley, 1994.

[37] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[38] C. Weidmann, M. Vetterli, A. Ortega, and F. Carignano. Soft caching: Image caching in a rate-distortion framework. In *Proceedings of ICIP*, Santa Barbara, CA, October 26–30, 1997.

[39] D. Wessels. Intelligent caching for world-wide web objects. In *Proceedings of INET'95*, Honolulu, HI, June 1995. Internet Society.

[40] S. Williams, M. Abrams, G. Abdulla, S. Patel, R. Ribler, and E. A. Fox. Removal policies in network caches for world-wide web documents. In *Proceedings of ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, volume 26,4, pages 293–305, Stanford, CA, August 26–30, 1996.

[41] R. P. Wooster. Optimizing response time, rather than hit rates, of www proxy caches. Master's thesis, Virginia Polytechnic Institute, Blacksburg, VA, December 1996.

[42] R. P. Wooster and M. Abrams. Proxy caching that estimates page load delays. In *Proceedings of Sixth International WWW Conference*, Santa Clara, CA, April 1997.

[43] X. Yang and K. Ramchandran. An optimal and efficient soft caching algorithm for network image retrieval. In *Proceedings of International Conference on Image Processing*, 1998.

Appendix A

Glossary

AIFF: Audio Interchange File Format
BHR: Byte hit-rate
CGI: Common Gateway Interface
DCT: Discrete Cosine Transform
DNS: Domain Name Service
DPCM: Differential Pulse Code Modulation
EZW: Embedded Zero-tree of Wavelet coefficients
FOLDOC: Free On-Line Dictionary of Computing
FTP: File Transfer Protocol
GIF: Graphics Interchange Format
GVU: Graphics, Visualization & Usability Center
HR: Hit-rate
HTML: Hypertext Markup Language
HTTP: Hypertext Transfer Protocol
IP: Internet Protocol
ISP: Internet Service Provider
JPEG: Joint Photographic Expert Group
LRU: Least Recently Used
LZW: Lempel-Ziv Welch compression
MBone: IP Multicast Backbone
MIME: Multipurpose Internet Mail Extensions
MP3: MPEG Audio Layer 3
MPEG: Moving Picture Experts Group
NCSA: National Center for Supercomputing Applications
NLANR: National Laboratory for Applied Network Research
NSF: National Science Foundation
PC: Personal Computer
PDA: Personal Digital Assistant
PNG: Portable Network Graphics
SIPI: Signal and Image Processing Institute

TCP: Transport Control Protocol
URL: Uniform Resource Locator
UDP: User Datagram Protocol
USC: University of Southern California
WWW: World Wide Web

Appendix B

Simulator

The simulator used in the experiments in chapter 7 is written in Perl using Perl's object oriented facilities. It was developed under Solaris, but being written in Perl, it can be run in any environment where Perl is available. The simulations were conducted under Solaris, Linux and Windows NT.

The simulator implements a Squid-like cache where the cache is filled up until a configurable high-water mark is reached. When this happens, the cache is emptied until the space used is below a configurable low-water mark. By default the high- and low-water marks are the same as in Squid, i.e. 95 % and 90 %, respectively, but they can be modified by command line options.

The simulator implements a wealth of different hard and soft caching schemes to make it easy to study the effects of different parameters. Replacement and refreshment policies, number of recoding levels, user models, etc. are fully configurable from an extensive set of options and new ones can be easily added.

B.1 Cache Size

The cache size is fully configurable and determines the base value used with the high- and low-water marks. These marks are also configurable as a percentage of the base size. Given the way the cache operates, it should be noted that the cache size should never reach the configured size, since it is emptied when the amount of data in the cache hits the high-water mark.

B.2 Replacement Policies

The possible replacement policies are:

1. LRU

2. Size

3. Optimal soft replacement policy

4. Combination of LRU and Size

All of these replacement policies exist in both normal, hard variants as well as in recoding versions. For LRU and Size, there exists also the second chance recoding version of the base algorithm. The last option, combination of LRU and Size, is still under study and was therefore not used in the experiments performed.

B.3 Refreshment Policies

As the choice for the adjustment of the object's access time after recoding in LRU-based policies, there are two choices:

1. Current time

2. Linear model

These models have been explained in section 7.2.5.

B.4 Recoding Levels

The number of recoding levels is fully configurable via this command line option. It can be set to any number of levels desired.

B.5 User Model

As far as the access type is concerned, all user accesses are assumed hard accesses. There is no option to configure soft access.

Three user models were programmed were targeted at the user's preferences towards recoded images. These models were:

1. Accept everything

2. Reject recoded

3. Accept slightly recoded

The first user would always accept any object given, regardless of the number of recodings performed on the object. The second user is a picky one, refusing all recoded images. This refusal is made always when a recoded image is presented, even if it has been recoded only once.

The last user represents the most realistic scenario. In this model, if an object was recoded only a few times, it was accepted, and if the object had been recoded more than a few times, it was rejected. The threshold for "a few" was set at a level where 60 % of the recoding levels, not bytes, were still left in the image. For a 10-level JPEG, this would mean, that anything with at least 6 layers left, would be accepted, less than 6 layers, and it would be rejected.

B.6 Recoding Model

The two recoding models available are:

1. JPEG-like
2. Linear

These options are as described in section 7.2.2.

B.7 Recodable Objects

There are also three options for setting the recodable object types:

1. JPEG only
2. JPEG and GIF
3. Everything

The first two were used in the simulations. The last option is for simulating separate proxies, where a proxy would only see recodable traffic. Even though the logfiles normally contain information about the object type, and they could thereby be separated in advance, this option guarantees the desired behavior because sometimes, the logfiles are lacking in information.

B.8 Flow of Execution

The simulator reads as input a logfile in the format produced by the Squid-proxy. In this format, there are ten fields on each line, and these fields are as shown in table B.1.

The simulator read the logfile line by line and decides what to do with the object. The next line is read only when the processing has been finished for the previous line.

When the line has been read, it is parsed and some sanity checks are performed. These sanity checks are to verify that the request was completed without errors in the real operation, and that the object size is below a threshold. This threshold is hardcoded as the same as in Squid's default, 4 megabytes. This threshold is used so that extremely large objects do not hog the space in the cache.

Field	Content
Time stamp	Time when request was completed
Duration	Duration of request in milliseconds
Client	Client IP-address
Action	What action the proxy took
Size	Amount of data written to the client
Method	What HTTP-method was used (GET, HEAD, POST,...)
URL	URL requested
Ident	Ident string (if enabled)
Hierarchy	How the object was retrieved (direct, parent,...)
MIME-type	MIME-type as given by server

Table B.1: Squid logfile fields

After the request has been validated using the above checks, the execution continues. The simulator then decides what to do with the request. This decision is shown in pseudo-code in figure B.1.

```
if (object in cache) {
    /* Object in cache */
    /* Check if (possibly recoded) object OK */
    if (object accepted) {
        /* Recoded object OK. Hit. */
        Update statistics.
    } else {
        /* Recoded object not OK. Miss. */
        Retrieve missing bits.
        Update statistics.
    }
} else {
    /* Object not in cache. Miss. */
    Insert object into cache
    Update statistics
}

if (cache size > high-water mark) {
    Empty cache until cache size below low-water mark
}
```

Figure B.1: Pseudo-code of decision-making process

The decision-making process in figure B.1 is performed for each validated request.

When all entries in the logfile have been processed, the simulator prints out a summary of the input data, how many requests and bytes were present as well as the statistics of the performance. These statistics include the number of hits, hit-rate, how many bytes were found in cache, byte hit-rate, average download time for a request, and what the average download time would have been without the presence of caching.

These statistics are printed out with unique labels making it easy to extract the results and compare the results from a large set of simulation runs.