Simulation Evaluation of Web Caching Hierarchies

A Thesis Submitted to the College of

Graduate Studies and Research

in Partial Fulfillment of the Requirements

For the Degree of Master of Science

in the Department of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan

by

Mudashiru Busari

June 2000

Permission To Use

Abstract

The rapid growth of the Web has sparked significant research activity on improving the performance and scalability of the Web. Much of this work focuses on Web caching: client-side caching, server-side caching, proxy-level caching, and multi-level caching hierarchies. Single-level caching and multi-level caching hierarchies combined with well-informed cache management strategies are obvious keys to the performance and scalability of the Web.

The first part of this thesis research describes the design and use of a synthetic Web proxy workload generator called ProWGen to evaluate the effectiveness of Web proxy caching architectures. The synthetic workload generator can be used to closely match the salient characteristics of empirical Web proxy workloads (e.g., one-timers, Zipf-like referencing behaviour, heavy-tailed file size distribution, temporal locality), or to generate multiple workloads that differ from each other in carefully controlled ways (e.g., Zipf slope, tail index, correlation between file size and popularity).

The second part uses the workload generator and trace-driven simulation to address two main research questions: "How sensitive are Web proxy caching algorithms to workload characteristics?" and "How can workload characteristics be exploited to improve the performance of multi-level Web caching hierarchies?"

The simulation results highlight the sensitivity of Web caching performance to Zipf slope, temporal locality, and the correlation (if any) between file size and popularity, but the relative insensitivity to one-timers and the heavy tail index. Several novel multi-level caching algorithms are also investigated, and their performance tradeoffs discussed. For example, simulation results of a two-level caching hierarchy show that using the LRU replacement policy at the lower level and the GD-Size policy at the upper level offer modest improvement in caching performance. Furthermore, a size-based partitioning strategy that keeps small files at the lower level and large files at the upper level can improve the performance of Web caching hierarchies.

Acknowledgements

I am indebted to a large number of people, all of whom have contributed in one way or the other to the successful completion of my Master's programme. First, I would specifically like to thank my supervisor, Prof. Carey Williamson, for his guidance, wisdom, valuable suggestions, promptness in reviewing my work, and ensuring that everything is on schedule. Carey, you are greatly appreciated. I also appreciate the contributions of each and every member of my committee, Prof. Rick Bunt, Prof. Herb Yang, and Prof. Chris Soteros (External) for improving the quality of this work with their valuable comments and suggestions.

I would also like to thank Greg Oster and Anirban Mahanti for their relentless willingness to help at different stages of my research. Greg, thanks also for always making those Web access logs available; Anirban, thanks for sparing your time to answer those numerous questions about your own thesis work.

A special appreciation to TR*Labs* (Telecommunication Research Laboratories) Saskatoon for funding my post-graduate education and providing a very conducive environment for carrying out this research. In particular, I would like to thank Trudy Gall and Jack Hanson for providing necessary assistances on numerous occasions. I would in addition like to thank TR*Labs* for the personal computer given to me at the end of this research.

Always in my mind, back home in Nigeria, is my mentor, Remi Okeyinka, for teaching me the values of hard work and determination throughout my undergraduate years. Thanks for your moral support and words of encouragement during this research. I would also like to thank Mr. J.A. Oladokun for sending encouraging messages at intervals.

Lastly, but always first in my heart, I appreciate the support of my wife. Aminat, thanks for your incredible support.

Finally, I give thanks to Almighty Allah, the beneficient, the merciful, for guiding me to this stage in my life.

Table Of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet continues to grow at an explosive rate. The growth has been attributed to many factors [7, 48, 49]: the ease with which people can access information on the Web through graphical user interface browsers; the increasing availability of tools for creating and publishing Web documents; and the flexibility of the protocols used for communication between heterogenous computing platforms. Also important are the decreasing cost of personal computers and the decreasing cost of accessing the Internet through competing Internet Service Providers. Services such as free electronic mail, free Web page hosting, and free disk storage space on the Internet are contributing to the staggering growth as well.

As the Web continues to grow, Web servers are becoming more overloaded and the Internet backbone is becoming more congested, with the resultant effect being an increase in end-user perceived latency. The performance degradation of the WWW (World-Wide Web) is frustrating users and a new meaning has been coined for the acronym WWW - “World-Wide Wait.”

The overall performance of the Web is determined by the performance of the components that make up the Web: the clients, the servers, the proxies, the networks, and the protocols used for communication [7]. From the client side, the access latency to remote data includes several components [20]: (1) the time for the client request to reach the server; (2) the time at the server to process the request; (3) the time for the reply to reach the client from the server; and (4) the time at the client to process the reply. The delays in (2) and (4) can be reduced by having enough resources at the server and the client ends to support the anticipated load. The delays in (1) and (3) are network-dependent and represent what is called the roundtrip delay between client and server. The roundtrip delays also include transmission and propagation delays at each intermediate router. To some extent roundtrip delays can be reduced by having enough resources (e.g., bandwidth and buffer) at the routers to support the anticipated load and reduce queueing. Another approach is to reduce the amount of data sent into the network. The latter can be done in several ways, one of which is caching, which can reduce the number of requests sent to distant servers.

Caching is a well-accepted means of improving the performance of the Internet [1, 7, 12, 13, 18, 20, 28, 62, 65, 76] and can be employed at different locations on the Web. At the client side [2, 18, 23, 33, 62, 74], browser software can utilize memory (non-persistent cache) or disk cache (persistent cache) to store copies of recently requested documents to satisfy subsequent access to the same document. At the server side [6, 7, 8, 9, 18], recently requested documents can be stored in memory to satisfy future requests to the same document by the same user or another user, thus eliminating disk I/O access time. Within the network [1, 10, 12, 13, 18, 20, 28, 56, 59, 62, 65, 74, 76], proxy caches can be employed to act as intermediaries between Web clients (browsers) and Web servers, accepting user requests and forwarding them to Web servers only as necessary. When a requested document is returned by a Web server, the proxy server sends the document to the requesting client and stores a copy of the document in the proxy's cache to satisfy future requests without contacting the server.

Caching proxies have gained widespread popularity on the Internet. By keeping local copies of documents requested by Web clients, and using them to satisfy future requests to the same documents, the amount of traffic flowing between Web clients and Web servers can be reduced. As a result, Web servers see a reduced load, unnecessarily duplicated traffic through the Internet is reduced, and the delays perceived by users are reduced.

In recent years, multi-level proxy cache configurations have received increasing research attention [27, 35, 38, 49, 50, 59, 64, 65, 68]. In a hierarchical configuration, proxies at or near the end user constitute the lowest level of the hierarchy. The lowest level proxies are configured to have a sibling-sibling relationship with one another and may have a child-parent relationship with a higher level proxy, usually a (geographically) regional proxy. Regional proxies can in turn connect to a higher level proxy, usually a national proxy. A request that cannot be handled at the lowest level is sent up the hierarchy until the requested document is located in one of the proxies or otherwise retrieved from the original server. A protocol called the Internet Cache Protocol (ICP) [35, 58, 60, 70] is employed by member proxies of a hierarchy to query peer proxies or a parent proxy for a copy of requested document.

The use of caching proxies on the Web introduces several performance issues. The key issues are cache management and cache consistency. Understanding the characteristics of traffic flowing through Web proxy servers is of paramount importance to how these issues can be addressed. Characterizing proxy workload helps identify characteristics that affect caching, which can then be explored in capacity planning and simulation studies [1]. A number of studies have characterized proxy workloads [1, 10, 48, 49, 50, 62, 71], and a number of characteristics have been identified in many of the proxy access logs used. Some common characteristics are: (1) the document popularity often follows the Zipf law [10, 48, 49, 50, 63]; (2) the file size distribution for Web documents is heavy-tailed [1, 10, 48, 49, 50]; (3) many (e.g., 50-70%) of the documents are referenced only once[1] [1, 48, 49, 50, 62]; and (4) temporal locality exists in the Web proxy reference stream [48, 49, 51, 63]. All these characteristics are important to proxy caching techniques and a number of caching policies have been designed based on these characteristics.

The issue of cache management is one of the issues that must be properly addressed in a caching proxy in order to improve Web performance. A Web proxy needs to make good replacement decisions each time a new document is coming into the cache. After deciding that a new document is to be cached, some of the old ones

[1] These documents are referred to as one-timers [8].

in the cache might need to be evicted to create enough storage space for the new document. It is therefore important to make good choices about which document (or documents) to remove (e.g., to ensure that only documents that are not likely to be requested in the near future are evicted from the cache).

Many cache management techniques have been proposed in the literature. The Least Recently Used (LRU) algorithm [2, 6, 7, 9, 23, 71, 74] is one of the most common, and is used in public domain proxy software (e.g., Squid) because of its simplicity of implementation. LRU removes the document that has spent the longest time in the cache without being accessed. Variations of this algorithm include LRU-MIN, LRU-THOLD [2], LRU-K, and SLRU [9]. Another common policy is the Least Frequently Used (LFU) algorithm [6, 7, 8, 9, 23, 71] which maintains a reference count for each document in the cache. LFU removes the document with the lowest reference count when a replacement decision needs to be made. Variations of this policy include LFU-Aging, LFU*, and LFU*-Aging [6, 8]. A third common policy, which takes the size of documents into consideration, is the SIZE policy [7, 9, 71]. The SIZE policy removes the largest document from the cache each time a document needs to be evicted.

There are also many hybrid policies, such as the LRV (Lowest Relative Value) policy [62], the LCP (Least Caching Profit) policy [13], page load delay estimate policy [73], cache partitioning policy [55, 73], and the GD-Size (Greedy Dual Size) policy [23]. The LRV policy computes the relative value of each document in the proxy's cache based on the cost of fetching the document, the benefit of purging the document, and the probability that the document is accessed again in the future. The LCP policy removes documents promising the least benefit with respect to loading latency. The page load delay estimate policy takes several factors into consideration, such as the time to connect to originating Web server, status of the network link, size of the requested document and its frequency of reference. The partitioning approach combines the LRU policy with several size-based strategies. The GD-Size policy associates, with each new document, a value H which represents the cost of bringing the document into the cache. A replacement decision removes the document with the lowest value of H, H_{min}, and reduces the H values for all documents by H_{min}. If a document accessed is found in the proxy's cache, its H value is restored to the original cost of bringing it into the cache. Variations of this algorithm include GD-Size(1), GD-Size(pkts), GD-Size(hops) and GD-Size(weightedhops) [23].

1.1 Motivation and Thesis Goals

Two major lines of research pertaining to Web proxy servers can be identified: proxy workload characterization, and performance improvement of proxy servers. The latter can be subdivided into cache replacement policy design issues and cache consistency issues. Neglecting the cache consistency issue, a natural question at this juncture is *How do workload characteristics affect the performance of the different cache replacement policies that have been proposed in the literature?* In addition, the goal of a multi-level caching hierarchy is to further reduce traffic and improve the performance of the Web. Hence a second research question arises: *Is it possible to improve the performance of a caching hierarchy with novel caching strategies?* These two questions form the basis of this thesis research.

Unfortunately, the first question is not as easy as it appears. For instance, in order to investigate how a particular workload characteristic (e.g., one-timers) affects different replacement policies, we need empirical workloads that differ only in that characteristic. Such workloads obviously may be difficult to obtain. Therefore, to answer the research questions above, there is a need to develop a synthetic Web proxy workload generator that offers the required flexibility. Furthermore, developing such a tool can overcome the drawbacks associated with using empirical workloads, such as trace collection difficulties, storage space requirements, anomalous behaviours, discontinuities, privacy concerns, and limited tunability.

The goals of this research can now be expressed as:

- Develop a synthetic proxy workload generator that will allow the synthesis of workloads exhibiting different selected workload characteristics.

- Make use of synthetic workloads from the workload generator in experimental studies seeking to answer the following research questions:
  - In a single-level proxy cache, how sensitive is Web proxy caching performance to certain workload characteristics (e.g., Zipf-ness, heavy-tail property, correlation, temporal locality)? How does the degree of sensitivity change depending on the cache replacement policy?
  - In a multi-level caching hierarchy, can overall caching performance be improved by using different cache replacement policies at different levels of the hierarchy (e.g., LFU at the lowest level and LRU at the highest level)?
  - In a multi-level caching hierarchy, can overall caching performance be improved by maintaining disjoint document sets at each level of the hierarchy (e.g., small documents at the lowest level, and large documents at the highest level)?

1.2 Thesis Overview

In the first part of this thesis research, a tool set called *ProWGen* (Proxy Workload Generator) was developed for synthetic generation of Web proxy workloads. ProWGen eliminates the shortcomings of using empirical access logs as outlined earlier by generating typical proxy workloads within a very short time, avoiding unnecessary information in the access logs (e.g., by representing URLs with unique integer values, rather than variable-length ASCII strings), and employing controllable parameters to represent the different characteristics that are relevant to cache management evaluation (e.g., popularity, file size distribution).

In the second part of the research, synthetic workloads from ProWGen are used to evaluate the sensitivity of Web proxy caching techniques to specific workload characteristics (e.g., Zipf distribution, heavy-tails, temporal locality). The LRU, LFU-Aging, and GD-Size cache replacement policies were used in the experiments.

The simulation experiments in the third part of the work focus on a multi-level caching hierarchy. In particular, the experiments investigate: (1) whether using different cache replacement policies at different levels of a multi-level caching hierarchy can improve the performance of the hierarchy; and (2) whether maintaining disjoint document sets at each level of the caching hierarchy can improve performance. The experiments used synthetic workloads from ProWGen to examine under what conditions these approaches work well.

The rest of this thesis is organized as follows: Chapter 2 presents some background on the history of the Internet and the way the WWW operates. It also provides an overview of related research studies on the Web. In Chapter 3, a description of the tool set for proxy workload synthesis is presented. Chapter 4 presents the simulation experiments and results for the single proxy server. Chapter 5 presents the experiments for the multi-level caching hierarchy simulations. A summary of the thesis work, contributions, and directions for future research are presented in Chapter 6.

Chapter 2

Background

2.1 Brief History of the Internet

The history of the Internet began in the late 1960s, when the US Advanced Research Projects Agency (ARPA, and later DARPA) of the Department of Defense began funding an experimental wide area network among major US universities. This network later became the ARPANET [3, 37, 40, 77]. The original goal of the ARPANET was to allow people to share expensive computing resources and work in a more collaborative manner. The necessity to agree on a standard protocol by which communication could exist among the different computing platforms led to the development of the TCP/IP (Transmission Control Protocol/Internet Protocol) protocol suite [67]. With time, the number of hosts on the ARPANET grew as more networks were linked, and the ARPANET became the backbone of the Internet. In 1988, DARPA discontinued funding this experiment and the National Science Foundation took over control with NSFNET replacing ARPANET as the backbone of the Internet [3].

The growth continued at an exponential rate, with more regional networks joining, and by 1990, the Internet had grown to 3000 networks and 200,000 computers [67]. Today, the NSFNET is not the only backbone of the Internet, as several commercial long distance carriers (e.g., MCI, AT&T, and Sprint) help to link millions of hosts around the world.

2.2 The World-Wide Web

Until the early 1990s, the primary users of the Internet were academic, government, and industrial researchers [67]. Traditionally, the main applications used on the Internet were electronic mail, news, remote login, and file transfer. One new "killer application," the WWW (World-Wide Web) changed all that and brought millions of non-academic users to the Internet.

Originally conceived by CERN (European Centre for Nuclear Research) physicist Tim Berners-Lee, the WWW was seen as a global hypertext space [17] in which any network-accessible information could be referred to by a single "Universal Document Identifier" (later URI – Universal Resource Identifier [36]). The design of the WWW was based on a few criteria: an information system must be able to record associations between any arbitrary objects; users should not be constrained to the use of a particular language or operating system; information must be available on all platforms, including future ones; information within an organization must be accurately represented and entering or correcting it must be trivial. In more general terms, information should be accessible in a seamless fashion [49].

The architecture of the Web is based on a client-server model [6] with the client accessing information on a remote server using Web browser software[1]. A number of browsers are now available, ranging from text-based to GUI-based (Graphical User Interface) software like Netscape Communicator and Microsoft Internet Explorer.

Web browsers communicate with Web servers using the HTTP (Hypertext Transfer Protocol) protocol [67]. The HTTP protocol relies on the underlying TCP/IP protocol while fetching a page from a server [48, 49]. TCP is a reliable connection-oriented protocol that allows a byte stream originating from one machine to be delivered to another machine within an intranet or on the Internet [67].

There are two different ways to fetch a web page (e.g., text file, audio, video, image) from a web server: direct connection from client to server or indirect con-

[1] The first Web browser was written by Tim Berners-Lee and was named "WorldWideWeb" but was later renamed to avoid confusion [17].

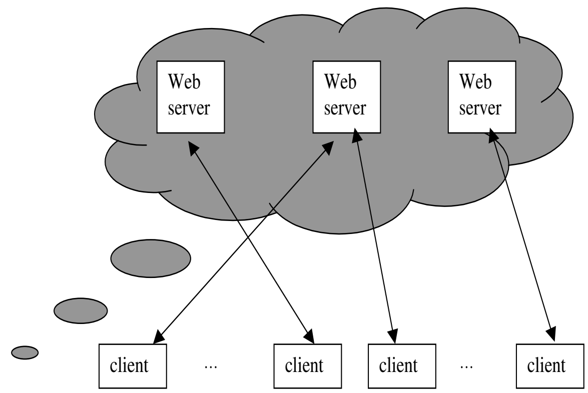


Figure 2.1: A Model of the Web with Clients Communicating Directly with Web Servers

nection through a Web proxy server. The two scenarios are illustrated in Figure 2.1 and Figure 2.2. In the first scenario (Figure 2.1), a Web client establishes a (HTTP) connection directly with the server. The server responds with the requested page, which is then displayed by the browser. In the latter case (Figure 2.2), the client's request is sent to a proxy server, which forwards the request to the server (if necessary), and when the requested page arrives, it is sent to the Web browser which displays it on the user's machine. The proxy server can also be configured to act as a caching proxy so that requested documents are cached locally to satisfy future requests. The remaining subsections elaborate on the three main entities depicted in Figure 2.1 and Figure 2.2.

2.3 Web Client

The WWW is a repository of documents of different types, usually referred to as *pages*. Each page may contain links to other (related or unrelated) pages anywhere in the world. Pages that point to other pages are said to use *hypertext*.

Web browsers (e.g., Netscape Navigator and Internet Explorer) are software programs that allow pages to be viewed. To display a Web page, a browser establishes a connection with a Web server and requests a Web page. When the page arrives, the

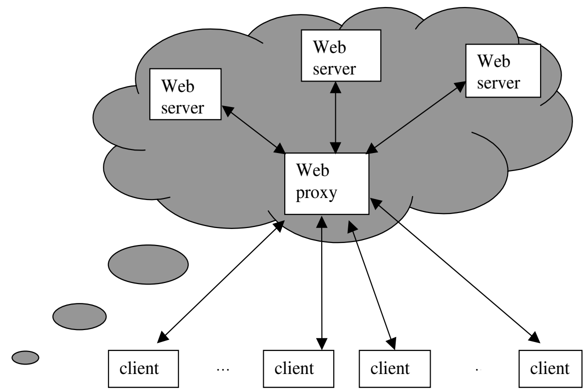


Figure 2.2: A Model of the Web with Clients Communicating Indirectly with Web Servers Through a Proxy Server

browser interprets the text and formatting commands contained within it and displays the page, properly formatted on the screen. With a GUI-based browser, links to other pages within a page are usually underlined and/or displayed in a different colour. These links are referred to as *hyperlinks*.

A Web page can also point to other pages through an icon, photograph etc. Some pages may contain non-viewable items such as video clips, audio clips etc. *Hypermedia* is used to refer to pages whose content is a mixture of text, images, audio, video, and any other type of media. Some Web pages may contain images that are very large, so it takes time for the browser to download them. To reduce the waiting time experienced by the user, an image may be displayed in a coarse resolution and the details filled in later as the text or body of the page is being displayed [67].

Some Web browsers employ a memory or disk cache on the client to store recently requested pages in case the user decides to revisit a previously visited page.

2.4 Web Server

A Web server is a software process running on the hosting machine. The process listens to TCP port 80 for incoming HTTP requests from a client. A browser will normally make a request for a Web page by establishing a connection to a host through port 80. After the connection has been established, the browser sends a *GET* request to fetch a page, the server process responds with an HTTP response message which has the following format [36, 48, 49]:

```
Response    =    Status-Line
                 [Header]
                 CRLF
                 [message-body]
```

`Status-Line` normally consists of the HTTP protocol version followed by a numeric status code and its associated description:

```
Status-Line  =    HTTP-Version  Status-Code Reason-Phrase
```

`Status-Code` is a 3-digit integer code representing the outcome of the server trying to process the client's request, while the `Reason-Phrase` is a short description of what the status code means. There are five basic classes for the status code represented by the first digit. The classes are [36]:

- 1xx: response codes starting with 1 indicate that the response has been received by the server and is being processed.
- 2xx: this code indicates that the server was able to interpret the client's request and respond successfully.
- 3xx: more actions need to be taken in order to handle the request successfully.
- 4xx: the server cannot successfully handle the client's request (e.g., user error).
- 5xx: although the request was valid, the server cannot fulfill it (e.g., server error).

After the server's response has been sent back to the client, the TCP connection is closed immediately in HTTP/1.0, but in HTTP/1.1 [36], it may be used by the client to make subsequent requests without necessarily having to establish another TCP connection. The *CERN httpd*, also known as *W3C httpd* [12, 26, 52, 70] is an example of one of the first widely available public domain WWW server software programs. CERN's final version has been released, with research focus now shifted to the Java-based *Jigsaw* server [42].

2.5 Web Proxy Server

As the WWW continued to grow, it was soon recognised that forwarding every client's request to Web servers directly places excessive load on the servers and the network. A simple scenario is a typical student laboratory with several dozen machines. One student might locate a particularly interesting document, and bring it to the attention of other students. Within minutes, every other student in the room is fetching copies of the same document from the originating server. This places an excessive burden on the server and the network.

Use of a proxy server is a recognised technique to lessen the burden on the Internet backbone and on Web servers. A proxy server refers to an intermediary program which acts as both a server and a client, and can make a request on behalf of other clients [36].

Web clients (browsers) can be configured to use a proxy when making WWW accesses. The proxy makes document retrievals on behalf of the client, and returns the result to the client. Such an approach is common for corporations with a network firewall, which provides a single stringent security point for controlling access to and from the Internet. By accepting and responding to a client's request a proxy server is acting as a server, and when forwarding the request to a Web server, it acts as a client.

A caching proxy extends this idea by having storage space (called a cache) at the proxy. When a user requests a particular document via the proxy, the proxy looks to see whether a (recent) copy of that document is available from its cache, and if so, returns it directly. If not, the proxy requests the document from the indicated server, and passes the retrieved document back to the client, at the same time keeping a copy in its cache.

In summary a proxy server could be designed to provide any combination of the following capabilities [57]:

Proxying The server provides access for internal clients through a firewall, to the Internet. This strategy is known as "forward proxying." Forward proxying allows clients in an intranet to go outside the firewall without compromising the integrity of the private network (e.g., Web servers on the Internet will see requests originating only from the proxy, and thus the identities of all computers and users inside the organization are concealed). A server can also provide access for external clients, through a firewall, to internal content. This service is known as "reverse proxying."

Caching The server can cache Web content locally, conserving bandwidth at network bottlenecks by storing frequently requested content locally. This content can be downloaded and regularly checked for changes in order to return up-to-date documents to all requests.

Filtering and Access Control A proxy server can be configured to block access to a particular Web site. For example, in an educational environment, access to certain content that is not supposed to be viewed by children may be controlled through a proxy server.

Logging A proxy server can be configured to record details about clients accessing the Web through the proxy. This may be for performance studies or for accounting purposes.

The most commonly used configuration is a caching proxy (often referred to by the generic term "Web proxy"). Proxy servers can be configured to interact with one another to improve network performance. In a tree-like fashion, referred to as hierarchical caching, proxy caches are placed at multiple levels of the network [48, 49, 50, 64]. At the bottom level of the hierarchy, there are the client caches.

When a request is not satisfied by the client cache, the request is redirected to the institutional cache. If the request is not found at the institutional level, the request is then forwarded to the regional cache, which in turn forwards unsatisfied requests to the national cache. If the request is not found at any cache level, the national cache contacts the origin server. When the document is found, either at a cache or at the origin server, it travels down the hierarchy, leaving a copy at each of the intermediate caches. Squid [52, 65, 70] and HARVEST [12, 27, 56, 70] are among the commonly used proxy software programs.

2.6 Related Work

The explosive growth of the Web has been the main reason for increasing research attention on the Web. Improving the scalability of the Web is much dependent on the components that make up the Web - the clients, the servers, the proxies, and the protocols. Understanding the workload characteristics of the first three components has provided some clues on how to improve performance. Some enhancements to Web protocols have also been suggested in a number of studies [16, 54]. Section 2.6.1 describes workload characterization studies of the Web, while Section 2.6.2 describes how workload characteristics can be exploited to enhance performance.

2.6.1 Web Server and Web Proxy Workload Characterization

Fundamental to the goal of improving Web performance is a solid understanding of Web server and Web proxy workloads. One of the well known characterization studies of Internet Web servers was done by Arlitt and Williamson [7]. Their objective was to characterize Web server workloads and identify common characteristics and their effects on server performance. Using Web server access logs ranging from one week to one year in duration, their analysis focussed on types of documents requested, popularity of documents, distribution of file sizes, etc. Several common characteristics were identified: (a) HTML and image are the dominant file types requested, accounting for over 90% of the activity in all access logs; (b) the median transfer size of documents is small, typically less than 5 KB; (c) 15-40% of files requested are accessed only once; (d) the file size distribution and the transfer size distribution are heavy-tailed; (e) about 10% of the files accessed on Web servers account for about 90% of the requests and bytes transferred.

In a similar study of Web proxy servers, Abdulla et al. [1] identified the following characteristics: (a) the mean file size of requested documents is less than 27 KB with the median around 2 KB; (b) 90-98% of files transferred are of types HTML and graphics; (c) graphic files account for most of the bytes transferred; (d) 25% of Web servers are responsible for 90% of requests and bytes transferred; (h) self- similarity is present in proxy access logs.

In a more recent study of proxy servers in a hierarchical caching configuration by Mahanti et al. [50], a number of workload characteristics that are consistent with those of Abdulla et al. [1] were identified. In addition, Mahanti et al. found that: (a) Web proxy document references do not strictly follow Zipf's law; (b) transfer size distributions are heavy-tailed; (c) concentration of references is stronger at lower-level proxies than at higher-level proxies in hierarchically configured proxies; (d) there appears to be no correlation between a document's modification rate and its popularity, although the modification rate tends to be higher at higher level proxies; (e) there is a gradual drift in "hot set", indicating that some documents have enduring popularity.

Several studies have used these workload characteristics to suggest ways of improving the performance of Web servers and Web proxies.

2.6.2 Approaches to Improving Cache Performance

One objective of workload characterization studies is to identify characteristics that are inherent in the network traffic seen by Web servers and Web proxies, and based on such characteristics, suggest ways in which performance can be improved. The workload characterization studies of Web servers and Web proxies have given some insight into how this can be done, particularly for caching.

As reported in server workload characterizations [7, 8, 18, 71] and proxy workload characterizations [1, 10, 48, 49, 62], some documents are requested more often than others. At the server side, it makes sense to move such documents from disk into memory, to reduce the latency resulting from disk I/O. Such documents can also be moved closer to the user through a caching proxy.

One issue that needs to be properly addressed is the issue of replacement policy. When the cache space is fully consumed and a new document is to be cached, which document (or documents) currently in the cache should be removed to make room for the incoming one? It is important that the server or the proxy makes good replacement decisions. This issue is of more concern for proxy server performance, because if a document cannot be served from the proxy's cache, the originating server must be contacted. The greater the number of requests (or volume of data) that can be satisfied from the cache, the better a replacement policy is. At the server side, this means fewer disk accesses and better response time. In a proxy server, this translates into reduced load on Web servers, fewer network accesses, and reduced latency for clients.

Replacement algorithms are commonly evaluated based on two metrics: *cache hit ratio* and *byte hit ratio* [6, 7, 8, 18, 48, 49, 62]. The cache hit ratio is the number of requests satisfied by the cache divided by the total number of requests from users, while the byte hit ratio gives the ratio of total bytes satisfied by the cache divided by the total bytes transferred to the user. Both metrics are needed since Web documents have highly variable sizes.

Researchers have proposed different algorithms in the literature. Some of the popularly known algorithms include the Least Recently Used algorithm (LRU) [2, 7, 9, 74]. In LRU, a list of documents is maintained in the cache and is ordered based on decreasing recency of access by the user. In other words, the most recently accessed document is the first in the list, while the least recently accessed document is at the bottom. When a new document is accessed, one or more documents in the list may have to be removed from the cache in order to make room for the newly accessed document [74]. Access to a document that is already in the cache brings the document to the top of the list. LRU is used in some public domain proxy software (e.g., Squid [52, 65, 70]).

The Least Frequently Used (LFU) policy [6, 7, 8, 9, 58, 71] is another popular algorithm, which makes replacement decisions based on frequency of access to each document. In LFU, each document has associated with it a count of the number of times it has been accessed. Access to a new document not in the cache may trigger a replacement decision, in which case the document with the lowest reference count is removed. If two documents have the same reference count, the one among these that was accessed least recently is removed. In other words, the LRU policy is used to break the tie.

LRU and LFU do not take document size into consideration. A size-based algorithm was studied in [7, 71]. With the *SIZE* policy, the largest file is removed when there is a need to remove a document from the cache. This often means that many small(er) files can be accommodated in the cache. A variation of the LRU policy referred to as LRU-MIN [71, 74] is also similar to the SIZE policy. LRU-MIN first tests whether there are any documents equal or larger in size than the incoming document; if there is, one of them is chosen by LRU. Otherwise, all documents larger than half the size of the incoming document are considered; if there are any, one of them is chosen by LRU. If not, this is repeated using one quarter the document size, and so on. This policy is like using $log_2(SIZE)$ as the primary key and LRU as the secondary key. A policy similar to the SIZE policy but that can be adapted to take the SIZE and/or the latency associated with fetching a document into consideration was proposed by Cao et al. [23] and is referred to as the *Greedy-Dual Size* (GD-Size) policy.

The LFU policy has some variations as well. For instance LFU-Aging, LFU*, and LFU*-Aging were studied in [6, 8]. The LFU* policy was targeted towards one-timers because of their high percentage in proxy workloads. This policy discriminates against one-timers by deciding to replace only documents with a reference count of one when a replacement decision is to be made. The aging policy is applied to both LFU and LFU* to avoid "cache pollution" – a situation which occurs when a document, after having accumulated a very high reference count becomes "unpopular" but never becomes a candidate for replacement because of its high reference count.

2.6.3 Cache Consistency Issues

A natural question arising from caching, especially in a Web proxy, is: *how does the proxy maintain cache consistency?* Techniques are needed for ensuring that the proxy does not issue out-of-date documents to clients. There are recognised mechanisms for maintaining cache consistency, but often heuristics are still involved. Some of the mechanisms for maintaining consistency are the following:

- The server indicates an expiry date and time when replying to a proxy's request for a document. The proxy does not fetch a new copy until the document expires. Certain documents that the content provider (server) does not want cached can be sent pre-expired.
- When a client requests a copy of a document that the proxy has in cache, the proxy asks the server for headers including last change date/time, and only fetches a new copy of the document if necessary.
- The proxy sends a "conditional request" to the server, "send me this document if it has changed since date/time" (this is often referred to as a *GET if-modified-since* request). The server sends a "Not modified" response (and no document bytes) if the document has not changed.

Cache consistency is also possible with the piggybacking technique suggested in [28] wherein the server groups its resources into volumes based on its directory structure or based on some probability that some resources are accessed together. When a request comes from a proxy, the server sends additional information, with the requested resource, about related resources that are likely to be accessed very soon. The proxy on the other hand could use this information to pre-fetch some of the resources. The piggybacked information can also be used to update resources in the proxy's cache, thus serving as a means for maintaining cache consistency.

The Piggyback Server Invalidation technique in [33] employs a similar mechanism for maintaining cache consistency. The basic idea is for servers to piggyback on a reply to a proxy client the list of resources that have changed since the last access by the client. The proxy client invalidates cached entries on the list and can extend the lifetime of entries not on the list. A similar invalidation approach for a caching hierarchy was proposed by Yu et al. in [75]. Cao et al. [25] conducted a performance evaluation study of the various heuristics for maintaining cache consistency. Their findings showed that strong cache consistency can be maintained with the invalidation-based approaches.

2.6.4 Hierarchical/Distributed Proxy Configuration

Proxies can be configured in a hierarchical manner to conserve bandwidth and reduce load on Web servers. The goals of the hierarchical cache are to distribute load away from server hot spots and to reduce access latency [59]. It aims to do this by reducing redundant document transfers using a mechanism that allows the caches of multiple organizations to be shared in a hierarchical arrangement of servers. The server at the top level of the hierarchy is called the *root* cache, while those nodes at the lowest levels of the hierarchy (closest to the clients) are called *leaf* caches. A node that is at the immediate level above a given cache is known as the *parent* of that cache. Caches that share the same parent are known as *siblings*, and the node that holds the original copy of the document is called the *primary site*.

Web clients are configured to request a document from a leaf cache using a standard HTTP proxy protocol. If the request cannot be resolved from its local set of documents, the cache then queries its parents and siblings using a datagram oriented protocol known as the Internet Cache Protocol (ICP) [58, 70]. A cache retrieves the object from the sibling, parent, or primary site that responds with a

'Hit'. In the case that all the caches queried have replied with a 'Miss', the parent cache is contacted and the process is repeated. This procedure continues recursively with each higher level of cache querying its parent and sibling until the request is satisfied, or the root node is reached. In this last case, the document is retrieved from the primary site. The Harvest Object Cache [12, 27, 56, 59, 70] and Squid [52, 59, 65, 70] all use this strategy.

There are several problems associated with a caching hierarchy: (i) every level of the hierarchy may introduce additional delays; (ii) higher level caches may become bottlenecks and have long query delays; (iii) multiple document copies are stored at different cache levels.

Because of the problems associated with a caching hierarchy, there are several other similar configurations proposed in the literature to address its shortcomings. A distributed approach was proposed by Tewari et al. [68] which separates *data paths* from *metadata paths* and maintains a hierarchy of metadata that tracks where copies of data are stored. A *location hint record* also tells each cache where to locate nearby copies of data without suffering network latencies. The approach suggested by Povey et al. in [59] operates in a similar manner. This approach avoids the use of servers at the root and upper levels of the hierarchy to resolve requests and store cached documents. Instead, only the leaf caches are responsible for retrieving objects, while the upper level caches are only used to maintain information about the contents of these caches.

The collaborative method proposed by Yu et al. [74] uses the PICS (Platform for Internet Content Selection) protocol to pass caching information down the proxy hierarchy for the lower level proxies to make better caching decisions. This method does not require message exchange or probing of cache contents among the proxy servers to get caching information. There is a cache manager at each proxy that takes into account the caching status at the higher level proxies in deciding whether or not to cache an object, and which objects to replace in a replacement policy.

The Cache Array Routing Protocol (CARP) [11, 69] allows for "queryless" distributed caching. With CARP, multiple proxy servers are configured in an array to appear as a single logical cache to the clients. By using hash-based routing, CARP provides a deterministic "request resolution path", allowing a web browser or downstream proxy to know exactly where each URL would be stored across the array of servers. CARP is more scalable and becomes more efficient and faster as more proxy servers are added.

In the summary cache scheme proposed by Fan et al. [35], each proxy stores a summary of URLs of documents cached at every other proxy. When a user request misses in the local cache, the proxy checks the stored summaries to see if the requested document might be stored on other proxies. If it appears so, the proxy sends out requests to the relevant proxies to fetch the document. Otherwise, the proxy sends the request directly to the Web server.

The geographical push caching technique of Gwertzman and Seltzer [38] uses a replication approach to reduce bandwidth consumption and server loads. A server tracks geographical access to its list of files. Using a modified HTTP protocol, the server can detect when the demand for a file exceeds a threshold level, at which point the file is replicated at another server location. The server on which the file is replicated is chosen from among a list of servers willing to replicate files. The new server on which a file is to be replicated is chosen to minimize the amount of bandwidth used in future. Once the primary server gives a file to another for caching, the primary server forgets about the other server. The primary server's load will drop as clients begin to access the file from the new server. Should the primary server's load begin to climb again due to a new popular file, another server will be chosen to replicate the popular file.

2.7 Summary

The Internet started in the late 1960s with ARPANET as the backbone. ARPANET was originally designed to link certain universities in the US for research purposes. NSFNET took over control two decades later. The arrival of the WWW caused staggering growth in the number of users on the Internet. The explosive growth caused serious performance degradation and scalability problems. This led to many research studies on how the performance of the Web can be improved. Understanding workload characteristics was seen as a primary issue to improving performance. Web client and server workloads have been characterized in several studies. The identified characteristics (e.g., re-referencing to the same document) have suggested the need for caching. Caching at the server side reduces disk I/O latencies. At the client side, bandwidth is effectively utilized and latency is reduced. Proxy caching is a third form of caching employed within the network. It reduces the load on Web servers, reduces the load on the Internet backbone, and reduces the latency perceived by users. Two important issues arising from caching are choice of replacement policy and cache consistency maintenance. Several studies have addressed each of these issues in the literature. Recently, the multi-level caching hierarchy configuration has been identified to further improve Web performance and is attracting increasing research attention.

Chapter 3

Proxy Workload Synthesis

The focus of the first part of this thesis research is on developing tools for synthesizing Web proxy workloads with controllable parameters. The goal is to synthesize the aggregate client workload as seen at a typical Web proxy server.

The tool set developed is called *ProWGen* (*Pro*xy *W*orkload *Gen*erator). ProWGen uses mathematical models to capture the salient characteristics of Web proxy workloads, as identified in previous studies of Web proxy servers [1, 48, 49, 62, 71]. Since the main purpose of ProWGen is to generate synthetic workloads for evaluating proxy caching techniques, only those workload characteristics deemed relevant to caching are incorporated into the synthesized workloads. This approach reduces the complexity of the models, as well as the time and space required for the generation and storage of synthetic workloads.

This chapter describes the design of ProWGen, and the techniques it uses to model Web proxy workload characteristics. Validation of the synthetic workloads generated by ProWGen is also discussed.

3.1 Description of ProWGen

There are two possible approaches to synthesizing a proxy workload [14]. The first approach starts with an empirical trace and either subsamples it or permutes the ordering of the requests to generate a new workload different in some respect from the original. This approach is called a *trace-based* approach. The second approach, called an *analytic* approach, uses mathematical models for the workload characteristics of interest, and uses random number generation to produce workloads that statistically conform to these models.

There are tradeoffs between these two approaches. While the trace-based approach is straightforward to implement, it has limited flexibility since the workload is inherently tied to a known system. In contrast, the analytic approach offers flexibility: each mathematical model can have one or more parameters to regulate the characteristics of the synthesized workload. Furthermore, many workloads can be synthesized within a short period of time, as opposed to collecting access logs over several days or months – a necessary step for the trace-based approach to work. However, choosing the workload characteristics to model, and incorporating them into a single workload with the desired characteristics may prove challenging.

ProWGen uses the analytic approach. The workload generator incorporates five selected workload characteristics, which have been identified as important in previous workload characterization studies of proxy servers [1, 48, 49, 62, 71].

The workload characteristics modeled in ProWGen are:

1. One-time referencing
2. File popularity
3. File size distribution
4. Correlation between file size and popularity
5. Temporal locality

Each characteristic is modeled using one or more controllable parameters, so that workloads with varying degrees of these characteristics can be synthesized easily. The descriptions of how each of these characteristics is modeled in ProWGen are provided in the subsections that follow.

3.1.1 Modeling One-time Referencing

Studies of Web server and Web proxy workloads have shown that many documents requested from a server or a proxy are requested only once, regardless of the duration of the access log studied [7, 8, 48, 49, 50]. These documents are referred to as "one-timers" in the literature. Clearly, there is no benefit to caching one-timer documents since they are never accessed again. In fact, caching algorithms need to discriminate against such documents so that they do not clutter the cache and reduce its effectiveness [7, 8].

The study of one-timers is important because of their prevalence in Web workloads. Arlitt and Williamson [7] report that 15-40% of the unique files accessed from a Web server are accessed only once. The situation is even worse in Web proxy access logs, where one-timers can account for 50-70% of the documents [1, 48, 49, 50]. Furthermore, in hierarchically configured proxies (from the client's side), the percentage of one-timers tends to increase [50] as you move up the hierarchy (away from the client), since files cached at the lower levels may never be requested again from the higher level proxies.

Modeling the one-time referencing characteristic is thus important when generating workloads for evaluating different caching algorithms. Fortunately, the modeling of one-timers is relatively straightforward.

The approach used to model one-time referencing is to determine how many of the distinct files in the synthetic workload should be one-timers. Representing the percentage of one-timers by a parameter allows the user to specify a desired value. Once specified, the actual number of one-timers in the workload can be determined and then their reference count can be fixed at 1, with the reference counts for the remaining distinct files determined from the file popularity model described in Section 3.1.2.

ProWGen uses the algorithm shown in Figure 3.1 to produce a synthesized workload with the desired number of one-timers. For example, assuming that $N = 1000$ requests are to be generated to 200 files ($c = 20\%$), half of which are one-timers

Given:

N,	the total number of requests to be generated
c,	the percentage of distinct files (expressed relative to N)
d,	the percentage of one-timers (expressed relative to c)

then, we can calculate from the above given values

a,	the number of distinct files as $a = (\frac{c}{100} \times N)$
b,	the number of one-timers as $b = \left(\frac{d}{100} \times a\right)$

Now, assign a popularity of 1 to each of the b one-timer files

Figure 3.1: Algorithm for Modeling One-time Referencing.

($d = 50\%$), ProWGen computes $a = 200$ and $b = 100$. Assuming that each distinct file has a unique ranking based on the reference count, with the most popular file (i.e., the file with the highest reference count) having a rank of 1, then the ranks range from 1 to a (the total number of distinct files). Since b is the number of one-timers among the a unique files, then all files ranked from $a - b + 1$ to a have a reference count of 1.

3.1.2 Modeling File Popularity

One common characteristic in Web workloads is the highly uneven distribution of references to files [10, 48, 49, 50, 63]. In many cases, Zipf's law [61] has been applied to model file popularity [4, 5, 21, 32]. Zipf's law expresses a power-law relationship between the popularity P of an item (i.e., its frequency of reference) and its rank r (i.e., relative rank among the referenced items, based on frequency of reference). This relationship is of the form $P = c/r^{\beta}$, where c is a constant, and β is often close to unity. The most famous example of Zipf's law is the frequency of occurrence of English words [61]; when the number of occurrences is plotted versus the rank, the result is a power-law function with exponent close to 1.

In the Web context, Zipf-like referencing behaviour has been observed in many studies [4, 5, 10, 32, 48, 49, 50, 63]. That is, if Web files are ranked r according to their popularity P, then the popularity of a file follows:

$$P = \frac{c}{r^{\beta}}, \quad 0 < \beta \leq 1.$$

Some researchers have found that the value of β is close to unity [5, 32], precisely following Zipf's law. Others [4, 21, 48, 49, 50] have found that the value of β is less than unity, and that the distribution can be described only as "Zipf-like" with the value of β varying from trace to trace. This behaviour typically results in a straight line of (negative) slope β on a log-log plot of P versus r. The linear fit is usually good for the main body of the distribution, though it may deviate slightly at both the most popular end (due to "hot" documents) and the least popular end (due to one-timers) [49].

The synthetic workloads produced by ProWGen model both Zipf and Zipf-like distributions for file popularity. After fixing the number of one-timers as explained in Section 3.1.1, the reference counts of the remaining distinct files are determined from a Zipf-like distribution. The value of β supplied to the workload generator determines whether the generated reference stream will follow the Zipf distribution perfectly ($\beta = 1$) or not ($0 < \beta < 1$).

The Zipf-like distribution is modeled as follows. In order to determine the popularity of the remaining distinct files, the first step is to estimate the proportionality constant c from the Zipf formula $P = c/r^{\beta}$ using the known parameters (i.e., number of references (N), number of unique files (a), and number of one-timers (b)). Since all the one-timers have a reference count of 1, the rank r of the first one-timer is $(a - b + 1)$, and the rank of the last one-timer is a. The constant c can be estimated[1] using the middle one-timer with rank $r = a - (b/2)$ and popularity $P = 1$. Substituting these values into the Zipf formula gives:

[1]Note that this provides only an estimate of c since there is no requirement for the linear part of the plot to intersect with the middle one-timer.

$$1 = \frac{c}{(a - b/2)^{\beta}}$$

and thus

$$c = (a - b/2)^{\beta}.$$

Having estimated c, the popularity of each remaining distinct file (from rank 1 to rank $a - b$) can be generated directly from the Zipf formula. This algorithm is given in Figure 3.2. Note that no random number generation is involved in this step; rather the popularities are generated directly from the formula (and rounded to the nearest integer, of course).

Given:

a,	the number of distinct files
b,	the number of one-timers
β,	the exponent of the Zipf formula

first, estimate c, the proportionality constant of the Zipf formula using:

$$c = (a - b/2)^{\beta}$$

then, generate the popularity P of each non-one-timer distinct file as follows:

$$P_r = \tfrac{c}{r^{\beta}}, \quad r = 1, 2, \ldots, a - b.$$

Figure 3.2: Algorithm for File Popularity Generation from Zipf Distribution.

Unfortunately, after determining the popularity of each distinct file as indicated here, the sum of the popularities of all the distinct files (i.e., $\sum_{i=1}^{a} P_i$) does not always add up exactly to N (the desired number of requests to be generated). There are two main reasons for this: the estimate of the constant c is not exact, and all file popularities must be rounded to the nearest integer. ProWGen has an additional step to adjust the popularity assigned to each distinct file. This is done by scaling the popularities of the non-one-timers (i.e., files ranked from 1 to $a - b$), while still preserving the slope β. The scaling factor s is determined using $s = (N-b)/\sum_{i=1}^{a-b} P_i$. Note that the computation of s excludes the one-timers because their popularities should not be altered. For the same reason, while adjusting the popularities of the non-one-timers, popularities lower than 2 are not permitted.

Following this step, the total number of references is very close to N (typically within 1-5%, and nearly always within 10%). Further adjustment could be made at this point to match N exactly, but matching the number of documents and the number of one-timers exactly, and modeling the Zipf-like distribution as faithfully as possible were deemed to be more important.

3.1.3 Modeling File Size Distribution

Previous workload characterization studies [6, 7, 8, 38, 48, 49] have shown that the file size distribution for Web transfers is heavy-tailed. This heavy-tail property has been shown to be partly responsible for the self-similarity noted in WWW traffic [31]. The heavy-tailed distribution implies that relatively few large files account for a significant percentage of the data volume (in bytes) transferred to Web clients.

Clearly, the distribution of file sizes affects the design and performance of caching strategies. Caching only small files at the proxies can reduce the number of requests sent to originating servers. This can result in a high document hit ratio, but a low byte hit ratio. On the other hand, caching large files can result in a higher byte hit ratio at the expense of document hit ratio (since many small documents may be forced out of the cache).

For effective evaluation of cache management strategies, the heavy-tailed file size distribution must be incorporated into synthetic workloads. In particular, the "heaviness" of the tail should be adjustable so that its impact on caching performance can be evaluated.

In ProWGen, the modeling of the file size distribution is done in three parts:

1. Modeling the tail of the distribution using the Pareto distribution (a well-known heavy-tailed distribution)

2. Modeling the body of the distribution using the lognormal distribution

3. Joining the two distributions together

3.1.3.1 Modeling the Tail

The tail of the file size distribution can be modeled with a heavy-tailed function. A distribution is said to be heavy-tailed if the asymptotic shape of the distribution is hyperbolic. That is:

$$P[X > x] \sim x^{-\alpha}, \quad \text{as } x \to \infty, \quad 0 < \alpha < 2.$$

The doubly-exponential Pareto distribution is an example of a heavy-tailed distribution that has been used to model the tail of the file size distribution [14, 30, 31, 48, 49, 50]. Its probability density function is:

$$p(x) = \alpha k^{\alpha} x^{-\alpha-1}, \quad \alpha, k > 0, x \geq k,$$

and its cumulative distribution function is:

$$F(x) = P[X \leq x] = 1 - \left(\frac{k}{x}\right)^{\alpha}.$$

The parameter α, referred to as the tail index, determines the heaviness of the tail of the distribution. This distribution has infinite variance, and if $\alpha \leq 1$ then the distribution has infinite mean. It follows that smaller values of α represent heavier tails (i.e., more of the "mass" is present in the tail of the distribution). k is a parameter that determines where the tail of the distribution begins (i.e., it represents the smallest possible value of the heavy-tailed random variable).

The parameters α and k that characterize the file size distribution can be determined using *log-log complementary distribution* (LLCD) plots [7, 30, 31, 48, 49]. An LLCD plot is a plot of the complementary cumulative distribution function $\overline{F}(x) = 1 - F(x)$ versus x, on a log-log scale. Since heavy-tailed distributions have the property that

$$\frac{d \log \overline{F}(x)}{d \log x} = -\alpha, \quad \alpha > k,$$

the slope α can be estimated over the range exhibiting linear behaviour. This method is employed to estimate α from synthesized workloads in this study (α and k are provided as parameters to the workload generation process).

Incorporating the heavy-tail model into ProWGen begins with partitioning the distinct files into two sets: those in the body and those in the tail. The percentage of files in the tail is specified as an input parameter to the workload generator.

An inverse mapping technique based on the Pareto cumulative distribution function is used to generate the heavy-tailed file sizes. This inverse mapping function is obtained as follows. Given

$$y = 1 - \left(\frac{k}{x}\right)^{\alpha}$$

as the cumulative distribution function of the Pareto distribution, the goal is to obtain, for some Uniform(0,1) random number y, the corresponding value of x. Rearranging the equation and solving for x gives:

$$x = \left(\frac{k^{\alpha}}{1-y}\right)^{\frac{1}{\alpha}}$$

which can be rewritten as:

$$x = \frac{k}{(1-y)^{\frac{1}{\alpha}}}.$$

Since y is a Uniform(0,1) random variable, $1 - y$ is also a Uniform(0,1) random value. Therefore, the above equation becomes:

$$x = \frac{k}{(y')^{\frac{1}{\alpha}}}$$

where $y' = 1 - y$.

Thus, once α and k are specified to the workload generator, heavy-tailed values can be generated using this formula. The algorithm in Figure 3.3 uses this inverse mapping function to incorporate the heavy-tail characteristic into the file size distribution of workloads generated in ProWGen. Note, however, that while implementing this algorithm in ProWGen, an artificial upper-bound of 60 MB was imposed, to match empirical traces and to avoid sensitivities to extreme values in the synthetic workloads. If a heavy-tailed value generated is greater than 60 MB, it is simply discarded and another one generated.

Given:

α,	the tail index of the Pareto distribution
k,	the beginning of the tail of the distribution
n,	the number of heavy-tailed values to generate

Repeat the following steps n times

1.	generate y' as IID U(0,1)
2.	compute x, the next heavy-tailed value as $x = \frac{k}{(y')^{\frac{1}{\alpha}}}$

Figure 3.3: Algorithm for Generating Heavy-Tailed Values from Pareto Distribution.

3.1.3.2 Modeling the Body

ProWGen uses the *lognormal distribution* [41, 46] to model the body of the file size distribution [15, 30]. The lognormal distribution has the property that if $X \sim$ N(μ, σ^2) (i.e., X is normally distributed with mean μ and variance σ^2), then e^X has a lognormal distribution with parameters μ and σ^2, denoted by LN(μ, σ^2). The probability density function of the lognormal distribution is:

$$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{\frac{-(lnx-\mu)^2}{2\sigma^2}}.$$

While the lognormal distribution has no closed form and hence no cumulative distribution function, there is a straightforward approach which exploits the property of the lognormal distribution to generate lognormal variates. Thus, to generate a lognormal variate, it suffices to generate a normally distributed variate x from a $\mathrm{N}(\mu, \sigma^2)$ distribution, and then return e^x as a lognormal variate. Note, however, that μ and σ^2 are the mean and variance of the normal distribution; they do not represent the mean and variance of the lognormal distribution (instead they are called parameters of the distribution). In fact, if we let μ_l and σ_l^2 represent the mean and variance of the lognormal distribution respectively, then $\mu_l = e^{\mu+\sigma^2/2}$ and $\sigma_l^2 = e^{2\mu+\sigma^2}(e^{\sigma^2} - 1)$ [46]. It follows that if we want to generate a lognormal random variate with given mean μ_l and variance σ_l^2, we should solve for μ and σ^2 first, using $\mu = \ln(\mu_l^2/\sqrt{\sigma_l^2 + \mu_l^2})$ and $\sigma^2 = \ln[(\sigma_l^2 + \mu_l^2)/\mu_l^2]$. Then, μ and σ^2 represent the mean and the variance of the normal variate used to generate the lognormal variate, as outlined in Figure 3.4.

ProWGen uses the *polar* method [46] to generate the normal variates from which the lognormal variates are generated. This method generates normally distributed random values $\mathrm{N}(0, 1)$ in pairs. Note that given $X \sim \mathrm{N}(0, 1)$, we can obtain $X' \sim \mathrm{N}(\mu, \sigma^2)$ by setting $X' = \mu + \sigma X$. The algorithm for generating values from the lognormal distribution when the mean and standard deviation are given is presented in Figure 3.4.

3.1.3.3 Joining the Distributions

Because ProWGen models the body and the tail of the file size distribution separately, there are several additional constraints needed when joining these distributions together. First, values in the body should not be in the tail. Second, values in the tail should not be in the body. Third, when the file size cumulative distribution function is plotted, the curve should show a smooth transition from the body to the tail.

A simple approach to achieve this is to place an upper bound on file sizes generated for the body of the distribution[2]. That is, if a value larger than k is generated,

[2]Fortunately, a lower bound of k already exists on the values generated for the tail by using the Pareto distribution.

Given:
μ_l, the mean of the lognormal distribution
σ_l, the standard deviation of the lognormal distribution

Estimate μ and σ^2, the parameters of the lognormal distribution as:

$\mu = \ln(\mu_l^2/\sqrt{\sigma_l^2 + \mu_l^2})$

$\sigma^2 = \ln[(\sigma_l^2 + \mu_l^2)/\mu_l^2]$

Now, repeat the following steps as many times as required to generate as many values as required from the lognormal distribution

1. Generate u_1 and u_2 as IID U(0,1)
2. Let $v_1 = 2u_1 - 1$ and $v_2 = 2u_2 - 1$
3. Let $w = v_1^2 + v_2^2$
4. If $w > 1$, go to step 1
5. Let $y = \sqrt{(-2\ln w)/w}$
6. Let $x_1 = v_1 y$ and $x_2 = v_2 y$, where x_1 and x_2 are IID N(0,1)
7. Return $e^{\mu+\sigma x_1}$ and $e^{\mu+\sigma x_2}$ as two lognormal values

Figure 3.4: Algorithm for Generating Values from Lognormal Distribution.

it is simply discarded and another one generated. The side effects of this restriction are two-fold: (1) after generating the lognormal variates, the mean and the standard deviation of the resulting values may be less than the specified input values; and (2) the overall file size distribution may show an obvious discontinuity where the two distributions meet. Another approach is to remove the upper bound restriction, solving these two problems, but possibly increasing the percentage of files in the tail of the distribution.

In ProWGen, a compromise solution was employed, by using the upper bound restriction, and compensating for this by artificially inflating the values of the mean and the standard deviation specified for the lognormal distribution. Note that this *ad hoc* step is only needed when fitting a synthesized workload to an empirical workload. In other cases, simply disabling the upper bound restriction solves the problems, with only minimal impact on the percentage of files in the tail of the distribution, and negligible impact on the tail index estimated from the synthetic workload using an LLCD plot.

3.1.4 Modeling Correlation Between File Size and Popularity

Proxy workload studies show that many of the files transferred on the Web are small [1, 8, 21, 48, 49, 50]. A natural question that arises is whether there is any statistical correlation between the frequency of access to a file and its size. Some studies [21, 49, 50] have shown that there is little or no correlation between the frequency of access to a file and its size, though the issue is still the subject of some debate.

For flexible modeling, ProWGen allows generation of workloads that exhibit positive, negative, or zero correlation between file popularity and file size. Positive correlation means that large files are more likely to be accessed, while negative correlation means that small files are more likely to be accessed. While zero correlation is arguably the closest approximation of reality, the other workload generation options (strong positive and strong negative correlations) allow exploration of the sensitivity of caching algorithms to these workload characteristics.

Modeling and incorporating correlation characteristics into the workload generator is accomplished in three stages: (1) generate a set of file popularities using the approach in Section 3.1.2; (2) generate a set of file sizes using the approach in Section 3.1.3; and (3) use a mapping technique to introduce either positive, negative, or zero correlation between file popularities and file sizes. The approach follows thus:

1. Generate a list P of popularities for n distinct files using the approach in Section 3.1.2. Sort the list P into ascending order.

2. Generate a list S of file sizes for n distinct files using the approach in Section 3.1.3. Sort the list S into ascending order.

3. Calculate the cumulative distribution values for the items in list P. Construct a new list P_{new} with entries of the form (v_i, p_i), where v_i is a unique popularity from the list P and p_i represents its cumulative probability. Note that P_{new} will have x entries ($1 \leq x \leq n$), and that $p_x = 1.0$.

4. Calculate the cumulative distribution values for the items in list S. Construct a new list S_{new} with entries of the form (v_i, p_i), where v_i is a unique file size from the list S and p_i represents its cumulative probability. Note that S_{new} will have y entries ($1 \leq y \leq n$), and that $p_y = 1.0$.

5. Repeat the following steps n times to generate a new list L to represent the popularity and size mapping for each unique file. The j^{th} entry ($1 \leq j \leq n$) in L is of the form (f_j, s_j), where f_j represents the popularity and s_j the file size. The entry (f_j, s_j) is determined as follows:

   (a) generate a random number r_1 which is IID U(0,1).

   (b) search the sorted list P_{new} of pairs (v_i, p_i) to find the first item i that satisfies $r_1 \leq p_i$. If $p_i = r_1$ or $i = 1$, then let $f_j = v_i$. Otherwise, interpolate linearly using $f_j = v_{i-1} + \frac{(r_1 - p_{i-1}) \times (v_i - v_{i-1})}{(p_i - p_{i-1})}$.

   (c) i. if positive correlation is desired, use r_1 to search the list S_{new} just like in step (b) to determine the value of the file size s_j.

      ii. if negative correlation is desired, use $1 - r_1$ to search the list S_{new} just like in step (b) to determine the value of the file size s_j.

      iii. if zero correlation is desired, generate another U(0,1) random number r_2 and use it to search the list S_{new} just like in step (b) to determine the value of the file size s_j.

   (d) add (f_j, s_j) to list L as the popularity and size of unique file j.

6. The list L now represents a new list of n distinct files each with its own popularity and file size with the desired correlation introduced.

3.1.5 Modeling Temporal Locality

After determining the file size and the popularity of each distinct file in the workload to be synthesized, it then remains to determine the relative order in which the file requests appear in the output reference stream. This step is where the issue of temporal locality comes into play.

Temporal locality refers to the tendency (in real Web workloads) for documents referenced in the recent past to be referenced in the near future. From the information already computed about each distinct file, the number of times each file should appear in the synthetic trace (the popularity) is already known. However, given that a reference is generated for a file in the reference stream at a particular virtual time t_0, it is not clear how soon the next reference to the same file should be generated. Needless to say, the presence (and strength) of the temporal locality induced in the synthetic workload can have a dramatic effect on caching performance.

The approach used in ProWGen to model temporal locality is based on the finite size LRU (Least Recently Used) stack model used in [49]. An LRU stack is an ordering of all files by recency of reference [5, 66], such that the most recently referenced file is at the top of the stack (position 1) and the least recently referenced file is at the bottom of the stack. The stack is updated dynamically as each reference is processed. In some cases, this update involves adding a new item to the top of the stack, pushing others down; in other cases, it involves extracting an existing item from the midst of the stack and moving it to the top again, pushing other items down as necessary.

A finite size LRU stack is an LRU stack that can hold only up to a specified maximum number of m files. It is unable to model temporal locality beyond the m most recently referenced documents. Experiments by Mahanti [49] suggest that $m = 1,000$ is adequate to capture the temporal locality present in Web proxy workloads.

The important aspect of an LRU stack is that each position in the stack has an associated probability of referencing it. Note that the probabilities are associated with the stack positions, and not the documents. The stack position probabilities can be provided by the models, or obtained from analyzing an empirical workload. For instance, assume that the popularities of the distinct files in the workload are represented by

$$D = \{x_1, x_2, \ldots, x_n\}$$

where

$$x_1 \geq x_2 \geq \ldots \geq x_n.$$

Then probabilities a_i can be computed for each distinct file i using

$$a_i = \frac{x_i}{\sum_{j=1}^{n} x_j} \quad \text{for } i = 1, 2, \ldots, n.$$

There are two different approaches by which these probabilities are used in ProWGen. The first approach is referred to as the *static stack model* and the second is referred to as the *dynamic stack model.*

In the static approach, if the finite stack has a specified size m such that $m \leq n$, the cumulative probability y_i of referencing stack position i $(1 \leq i \leq m)$ is computed as:

$$y_i = \sum_{j=1}^{i} a_j.$$

The cumulative probability computed for each stack position is assigned at the beginning of the reference stream generation and does not change for the duration of the generation. This technique generates statistically "homogeneous" temporal locality in all the (re-referenced) documents in the workload.

In the dynamic approach, each time the LRU stack is modified (either by moving a file from another position on the stack to the top of the stack or by bringing a new file onto the stack), the cumulative probability for each stack position is recomputed using the a_i values of the files currently occupying each stack position. This approach can model "heterogeneous" (i.e., document specific) temporal locality properties. For instance, if at virtual time t_0, the first two stack positions hold file 1 and 2 respectively, then their state can be represented by $S_{t_0} = \{a_1, a_2\}$, where a_1 is the probability of referencing file 1 and a_2 is the probability of referencing file 2. Then, the cumulative probabilities of referencing these two stack positions are a_1 and $a_1 + a_2$, respectively. Assume that at virtual time t_1, the file in position 2 on the stack is moved to position 1. This action will trigger a recomputation of the cumulative probabilities associated with each stack position, and then the stack state becomes $S_{t_1} = \{a_2, a_1\}$. Thus the cumulative probabilities of referencing these two stack positions become a_2 and $a_2 + a_1$, respectively (note that a_1 need not equal a_2). This is how the dynamic approach works in ProWGen.

Having described the two approaches used to model temporal locality, the reference stream generation process is now described. At the begining of the reference stream generation, the LRU stack is empty. If the static approach is being used, each stack position has an associated cumulative probability of referencing that position. In the dynamic approach, the probabilities are uninitialized. The reference stream generation process starts by generating a U(0,1) random number x_i. Now, three cases are possible: (1) the stack is empty; (2) the stack is not empty but the next file to reference is not on the stack; (3) the stack is not empty and the next file to reference is already on the stack.

- In case 1 and 2, a file is selected uniformly at random from the set of remaining distinct files requiring references. A reference is generated for this file in the reference stream, and its remaining reference count is decremented by 1. If there are no remaining references for this file, then the file is removed from the candidate set. Otherwise, the file is moved onto the top of the stack (pushing other files down one position in case 2).

- In case 3, the stack is searched from the top (position 1) until a position j is encountered such that $x_i \leq y_j$. A reference for the file at position j is generated in the reference stream and its reference count decremented by 1. If there are no remaining references to be generated for this file, the file is removed from the stack and all files below it on the stack are moved up one position. Otherwise, the file is moved onto the top of the stack, pushing other files (if any) down one position.

In the dynamic LRU stack model, each of these cases will trigger recomputation of the cumulative probabilities associated with stack positions unless the state of the stack remains unchanged.

3.1.6 Other Issues

It is also worth mentioning what is *not* modeled in ProWGen. In particular, ProWGen does not include models of time of day, document types, document modifications, or individual client behaviours. These decisions were made to simplify the design of ProWGen and the structure of the simulation experiments for the thesis.

3.2 Validation of ProWGen

The purpose of validation is to ensure that the synthetic workloads produced by ProWGen have the correct quantitative and qualitative characteristics. The validation process establishes confidence in the workload generator and the statistical properties of the workloads that it produces.

There are two different aspects in validating ProWGen. The first aspect, often referred to as verification, requires checking that the set of programs used for modeling performs as desired. During this process, ProWGen was used to synthesize a small workload of 100 requests. Each step of the synthesis process was followed by manual verification to ensure that the tool set behaves in the manner expected of it. The second aspect of the validation process is to ensure that synthetic workloads from ProWGen are representative of empirical workloads, both in terms of quantitative characteristics (e.g., statistical distributions produced) and in terms of performance under caching strategies. This validation approach is the same as used by Bodnarchuk [19].

The latter validation step was accomplished in two stages. In the first stage, a sample workload of one million requests was synthesized with the intent of observing its characteristics and examining the behaviour of the synthetic workload in a caching experiment using the LRU replacement policy. During this stage, a problem (a sensitivity issue) was found in one of the underlying models, and a fix was warranted. In the second stage of the validation process, an empirical workload from the University of Saskatchewan proxy server was used for comparison with a synthetic workload from ProWGen. In subsection 3.2.1, the first stage is presented, along with an explanation of the observed anomaly in the synthetic workload, and the modification made in one of the underlying models to fix the anomaly. Subsection 3.2.2 presents the second stage of the validation using an empirical workload.

3.2.1 Sample Synthetic Workload from ProWGen

This section provides a synopsis of the parameters and settings needed to generate a synthetic proxy workload using ProWGen. It also highlights qualitative and quantitative characteristics of the synthetic workload.

ProWGen requires twelve input parameters to produce a synthetic workload. A sample setting of these parameters to synthesize one million requests is shown in Table 3.1. This synthesis requests zero correlation between file size and popularity, and uses the dynamic LRU stack approach to model temporal locality (see Section 3.1.5).

Table 3.1: Parameter Settings to Illustrate Use of ProWGen for Synthesizing One Million Requests

Parameter	Value
Total number of requests	1,000,000
Percentage of unique documents (of total requests)	30%
Percentage of one-timers (of unique documents)	70%
Correlation between file size and popularity	Zero
Zipf Slope	0.75
Tail index	1.3
Beginning of the tail (k) in bytes	10,000
Percentage of documents in the tail	20%
Mean of the lognormal distribution (μ)	7,000
Standard deviation of the lognormal distribution (σ)	11,000
LRU Stack Model for temporal locality	Dynamic
LRU Stack Size for temporal locality	1,000

Table 3.2 shows the quantitative results from the analysis of a synthetic workload generated by ProWGen using these parameter settings. As pointed out in Section 3.1.2, the total number of requests generated is not exactly one million, but

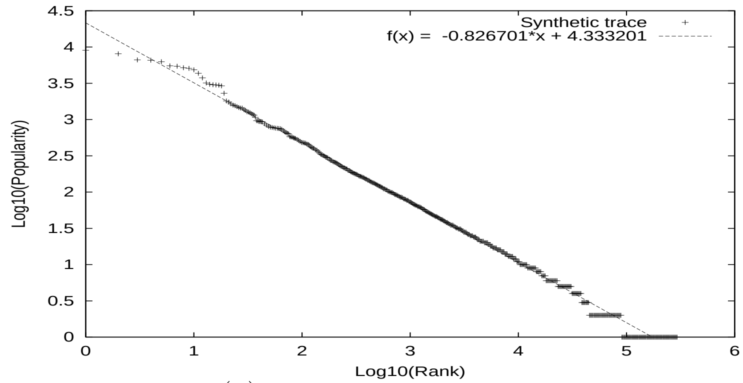
is within 1% in this case. Other values agree closely with the values of the input parameters (e.g., number of unique documents and number of one-timers). Correlation between file size and popularity is close to zero. The measured slope of the Zipf-like popularity curve and the tail index are close to the input values, and the R^2 values indicate the (strong) goodness of the fit used in estimating them. Table 3.2 also provides information about the file sizes generated. For example, the biggest file was 21 Megabytes (MB) while the smallest file was 32 bytes. The mean and the median file sizes are similar to those in empirical traces [49].

Table 3.2: Characteristics of Sample Synthetic Trace from ProWGen

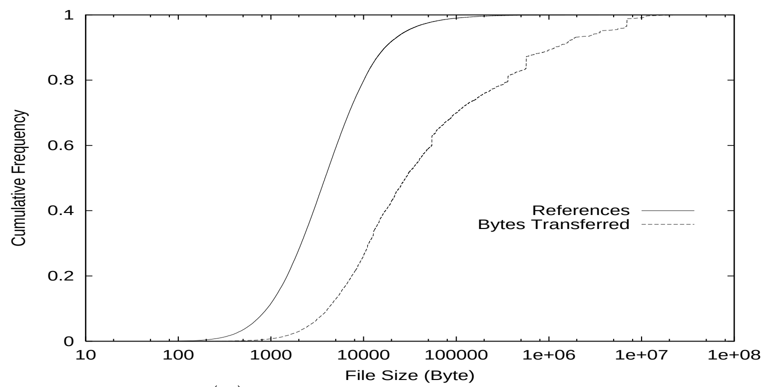
Item	Value
Total number of requests	991,099
Number of unique documents	300,000
Percentage of unique documents (of total requests)	30%
Number of one-timers	209,693
Percentage of one-timers (of unique documents)	69%
Total bytes of unique documents (GB)	3
Smallest file size (bytes)	32
Biggest file size (bytes)	21,809,968
Mean file size (bytes)	10,878
Median file size (bytes)	3,839
Correlation between file size and popularity	-0.000695
Zipf Slope	-0.826701
R^2	0.996158
Tail index	-1.293293
R^2	0.999592

The graphical characteristics of the synthetic workload are shown in Figure 3.5. The popularity ranking shown in Figure 3.5(a) resembles that seen in empirical traces [49].

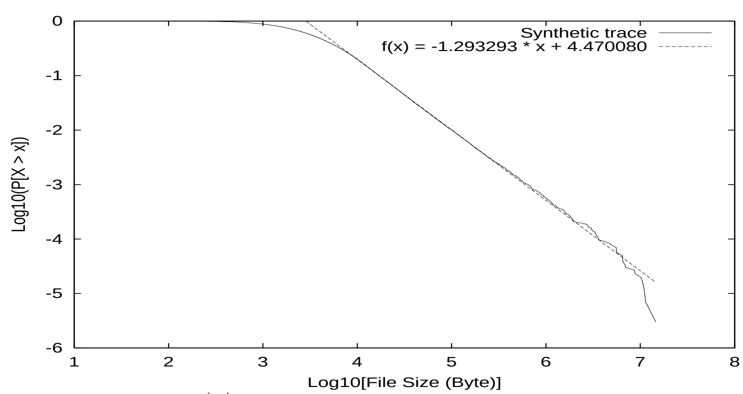
The file size cumulative distribution is shown in Figure 3.5(b). From this figure, the line representing references shows that about 80% of the references are for files smaller than 10,000 bytes. The second line representing bytes transferred shows the implication of a heavy-tailed file size distribution. While about 80% of the



(a) Popularity vs. Rank



(b) CDF plot for file sizes



(c) LLCD plot for file sizes

Figure 3.5: Popularity, CDF and LLCD Plots for Sample Synthetic Workload.

transfers send fewer than 10,000 bytes, these files account for only 30% of the data bytes transferred. Files in the extreme tail of the distribution (say, files larger than 100,000 bytes), which constitute roughly 1% of the total references are responsible for 30% of the total byte volume transferred. This is evidence of the well-documented heavy-tailed phenomenon observed in empirical workloads [6, 7, 8, 31, 38, 48, 49]. The difference between the mean and the median file sizes for the same synthetic trace shown in Table 3.2 is indicative of this heavy-tailed property.

The LLCD plot for estimating the tail index is shown in Figure 3.5(c). The tail is estimated starting from 10,000 bytes, which is the value supplied to ProWGen for synthesizing the trace. The measured tail index values matches the input value closely.

Having established the adequacy of the quantitative characteristics of the synthetic workload, the workload is next simulated with a single proxy server model using the LRU replacement policy. The hit ratios and byte hit ratios obtained are shown in Table 3.3.

One surprising result in Table 3.3 is the close match between the document hit ratios and the byte hit ratios across the wide range of cache sizes studied. This result is in stark contrast to the significantly lower byte hit ratios observed in empirical traces [6, 7, 8, 9, 10, 18, 21, 48, 49].

Upon further investigation, this anomaly was attributed to a set of files in the synthetic workload that were relatively large and relatively highly referenced. While the number of these files is relatively small, and statistically reasonable, given the assumption of zero correlation between file size and popularity, they are much more prevalent in the synthetic workload than in most empirical workloads, and have a dramatic effect on the byte hit ratio observed. In other words, the byte hit ratio results are very sensitive to the number of files in this category generated in the synthetic workload. Furthermore, the correlation coefficient provides very little hint about this problem, since these files constitute an insignificant fraction of the total files.

The chosen solution for this problem was to modify the model that maps file sizes

Table 3.3: Hit Ratios and Byte Hit Ratios with LRU Policy Using the Original Sample Synthetic Workload

Cache Size (MB)	Hit Ratio	Byte Hit Ratio
1	10.23	8.24
2	14.95	13.58
4	20.07	18.78
8	23.40	21.69
16	30.59	27.93
32	36.25	34.17
64	42.67	40.36
128	48.69	46.35
256	54.59	52.45
512	60.01	57.89
1024	64.94	62.88
2048	68.77	66.94
4096	69.72	67.97

and popularities (the correlation model), by putting a restriction on the popularity of large files. The actual modification to the underlying model in ProWGen was to restrict files larger than 10,000 bytes from having a popularity exceeding 100 (in fact, when an empirical workload was examined for the percentage of unique files larger than 10,000 bytes that were referenced more than 100 times, it was found to be insignificant). A correlation measurement on the resulting synthetic workload with this restriction showed that no undue correlation was introduced, while the objective of introducing this restriction was achieved. Also, no other workload characteristic was affected by this decision.

The same parameter settings as shown in Table 3.1 were used to re-synthesize the workload with this modification incorporated into ProWGen. The new workload was then used for the caching study, and the resulting hit ratios and byte hit ratios are shown in Table 3.4. The wider margin which now exists between the hit ratios and the byte hit ratios is similar to that observed for empirical traces [6, 7, 8, 9, 10, 18, 21, 48, 49]. The new workload was the same as the first one in all other respects, as shown in Table 3.5 (compare with Table 3.2).

Table 3.4: Hit Ratios and Byte Hit Ratios with LRU Policy Using the Modified Sample Synthetic Workload

Cache Size (MB)	Hit Ratio	Byte Hit Ratio
1	14.89	5.48
2	18.93	7.09
4	21.80	8.38
8	25.43	10.83
16	30.85	15.70
32	36.48	22.09
64	42.05	28.70
128	48.05	36.25
256	53.96	43.87
512	59.74	50.89
1024	64.84	57.33
2048	68.77	62.45
4096	69.72	63.68

Table 3.5: Characteristics of the Modified Sample Synthetic Workload from ProWGen

Item	Value
Total number of requests	991,099
Number of unique documents	300,000
Percentage of unique documents (of total requests)	30%
Number of one-timers	209,693
Percentage of one-timers (of unique documents)	69%
Total bytes of unique documents (GB)	3
Smallest file size (bytes)	32
Biggest file size (bytes)	21,809,968
Mean file size (bytes)	10,871
Median file size (bytes)	3,835
Correlation between file size and popularity	-0.000181
Zipf Slope	-0.826701
R^2	0.996158
Tail index	-1.292319
R^2	0.999586

3.2.2 Validation of ProWGen with Empirical Workload

The next step in the validation of ProWGen is to compare an empirical workload from the University of Saskatchewan proxy server to a synthetic workload generated by ProWGen (parameterized based on the empirical workload). A trace of five million requests was analysed to extract the input settings needed for the workload generator. The characteristics of the empirical workload are shown in Table 3.6.

Table 3.6: Characteristics of the Empirical Trace from UofS Proxy Server.

Item	Value
Total number of requests	5,000,000
Percentage of unique documents (of total requests)	34%
Percentage of one-timers (of unique documents)	72%
Total bytes of unique documents (GB)	19
Smallest file size (bytes)	0
Biggest file size (bytes)	53,857,877
Mean file size (bytes)	11,740
Median file size (bytes)	3,504
Mean file size (body only)	3256
Standard deviation of file size (body only)	2441
Zipf Slope	-0.807867
R^2	0.991546
Tail index	-1.322804
R^2	0.980057

Two different workloads are synthesized (*Trace1* and *Trace2*) from ProWGen to mimic the empirical trace. Table 3.7 presents the parameter settings used for the workload generation, and the analysis of the resulting workloads. As shown in the table, the temporal locality model was set to *Static* for *Trace1* and *Dynamic* for *Trace2*. This temporal locality model is the only difference between the two traces, so that the two approaches for modeling temporal locality can be evaluated and compared to the temporal locality characteristics in the empirical trace.

Other parameter settings are determined from Table 3.6, and are identical in the two synthetic traces (e.g., LRU stack size of 1000, zero correlation between file size and popularity). As noted in Table 3.7, the mean and the standard deviation

Table 3.7: Characteristics of the Two Synthetic Traces Used to Validate ProWGen

Item	Trace1	Trace2
Input to workload generator		
Total number of requests	5,000,000	5,000,000
Percentage of unique documents (of total requests)	34%	34%
Percentage of one-timers (of unique documents)	72%	72%
Zipf Slope	0.807867	0.807867
Tail index	1.322804	1.322804
Percentage of documents in the tail	22%	22%
Beginning of the tail (k) in bytes	10,000	10,000
Mean of the lognormal file size distribution	7,000	7,000
Standard deviation	11,000	11,000
Correlation between file size and popularity	Zero	Zero
LRU Stack Model for temporal locality	**Static**	**Dynamic**
LRU Stack Size for temporal locality	1,000	1,000
Summary of the workload generated		
Total number of requests	4,965,779	4,965,779
Number of unique documents	1,700,000	1,700,000
Percentage of unique documents (of total requests)	34%	34%
Number of one-timers	1,223,719	1,223,719
Percentage of one-timers (of unique documents)	71%	71%
Total bytes of unique documents (GB)	17	17
Smallest file size (bytes)	13	13
Biggest file size (bytes)	42,975,450	42,975,450
Mean file size (bytes)	11,157	11,157
Median file size (bytes)	3,962	3,962
Correlation between file size and popularity	-0.005042	-0.005042
Zipf Slope	-0.833727	-0.833727
R^2	0.998503	0.998503
Tail index	-1.326192	-1.326192
R^2	0.999874	0.999874

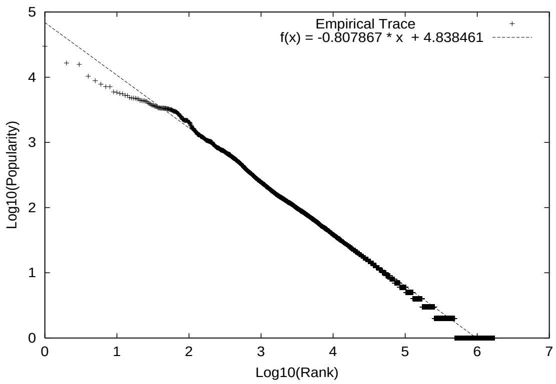
settings for file sizes are not exactly the same as the values in Table 3.6 measured from the empirical workload. The reason for inflating these values was discussed in Section 3.1.3.2.

The analysis of the two synthetic workloads presented in the remaining part of Table 3.7 confirms that all measured statistics for both traces are identical. That is, the traces differ only in the ordering of their references. Hence the graphical comparisons of the synthetic and the empirical traces in Figure 3.6 use only one of the synthetic traces (*Trace1*).

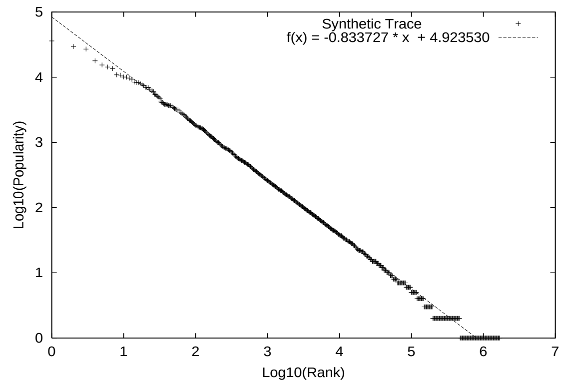
Figures 3.6(a) and (b) show the file popularity results for the empirical and the synthetic traces, respectively. The Zipf-like referencing behaviour is qualitatively similar in both graphs, and the estimated slopes using least-squares regression differ by only 3.2% (0.808 vs. 0.834). There is thus strong evidence that the desired Zipf-like referencing behaviour is present in the synthetic workloads.

Figures 3.6(c) and (d) show the tail index estimation using LLCD plots. Again, the graphs are shown for the empirical and the synthetic traces, respectively. The agreement between the two graphs is close, though the synthetic trace clearly has "better" straight line tail behaviour because of the mathematical model used for file size generation. The estimated slopes are almost identical (1.323 vs. 1.326), and the R^2 goodness of fit is high ($R^2 > 0.99$) for both graphs. As further confirmation of the fit, Figure 3.6(e) superimposes the cumulative distribution plots for file sizes and bytes transferred, and Figure 3.6(f) superimposes the two LLCD plots. There is thus strong evidence that workloads generated using ProWGen can match both the body and the tail of the empirical file size distribution.

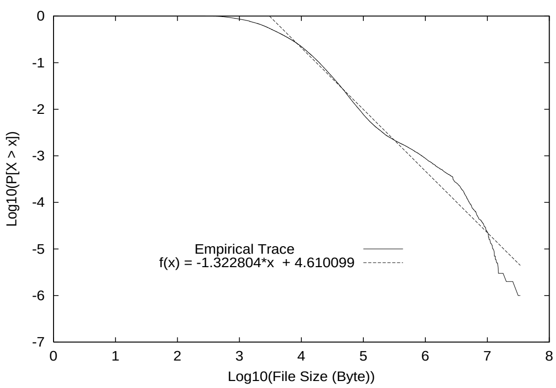
Having established the satisfactory quantitative characteristics of the synthetic traces, it remains to see how the synthetic traces behave compared to the empirical trace in terms of caching performance. The caching study uses three types of cache replacement policies (recency-based, frequency-based, and size-based) to understand the qualitative effects of the two temporal locality models. In particular, the three traces (*Trace1*, *Trace2*, and the empirical trace) are provided as input to a trace-driven simulation of a single proxy server model, using LRU, LFU-Aging, and the



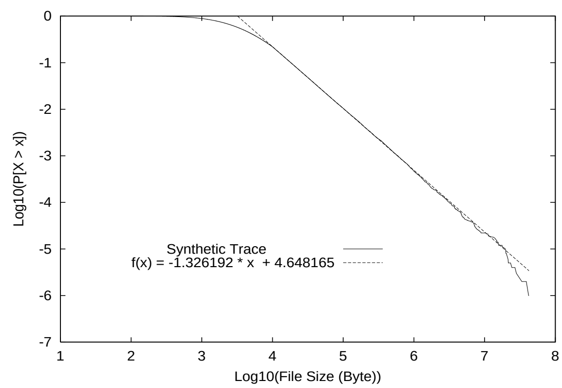
(a) Popularity ranking for UofS trace



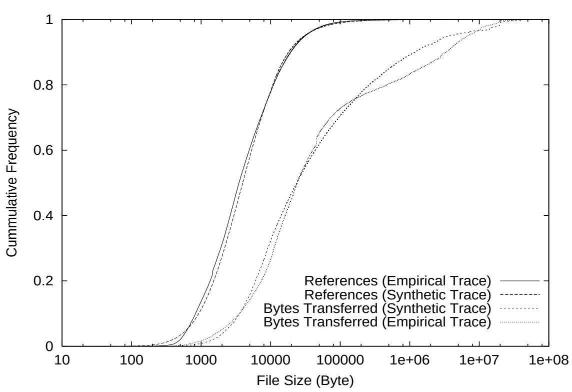
(b) Popularity ranking for Synthetic trace



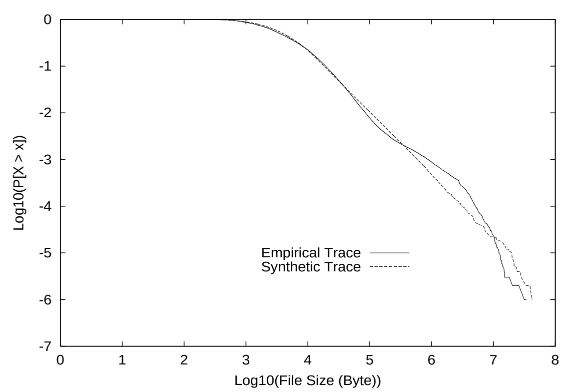
(c) LLCD Plot for UofS trace



(d) LLCD Plot for Synthetic trace



(e) CDF plot for UofS and Synthetic traces



(f) LLCD Plot for UofS and Synthetic traces

Figure 3.6: Comparison of Characteristics Between Empirical and Synthetic Traces

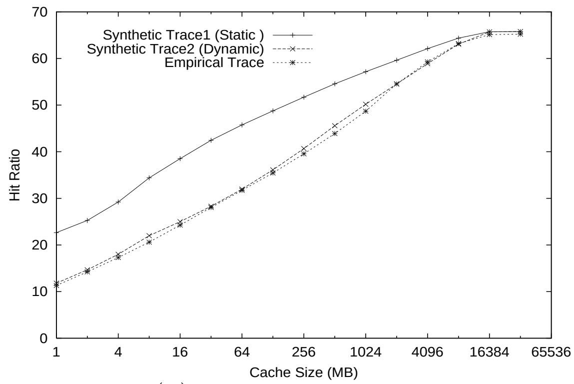
GD-Size replacement policies. Note that there is no cache warm-up for these simulations since the objective of the experiment is to investigate the temporal locality characteristics of the traces. Also, file size changes were not simulated.

The hit ratios and the byte hit ratios obtained under each policy for the three workloads are shown in Figures 3.7(a)-(f). Examining the hit ratios for all the policies (Figures 3.7(a), (c), and (e)), it is evident that the performance of *Trace2* (the synthetic trace with the dynamic stack model for temporal locality) more closely mimics the performance for the empirical trace, than does the performance of *Trace1* (static stack model for temporal locality).

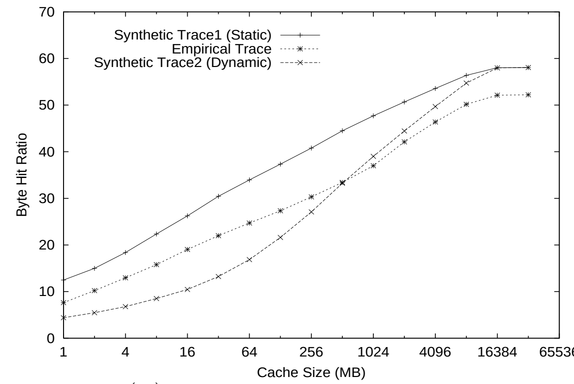
The static stack model (*Trace1*) seems to overestimate the hit ratio, perhaps because it provides homogeneous temporal locality in all documents; the dynamic stack model for temporal locality is less prone to this problem. Furthermore, it seems to provide robust estimates of cache hit ratios across a broad range of cache sizes (from a ridiculously small cache size of 1 MB to an "infinite" cache size of 32 GB). This behaviour, and the fact that the dynamic stack model seems to provide its best match on the (recency-based) LRU cache replacement policy suggests that it is a superior model for temporal locality.

In terms of byte hit ratios (Figures 3.7(b), (d), and (f)), the differences are less clear cut, since both models are somewhat lacking. For example, the static stack model (*Trace1*) consistently overestimates byte hit ratios (except at infinite cache sizes), while the dynamic stack model seems to underestimate byte hit ratios at small cache sizes and overestimate byte hit ratios at large cache sizes. This behaviour is attributed to the earlier-mentioned sensitivity of the byte hit ratio, particularly to the number of large popular documents in a particular trace.

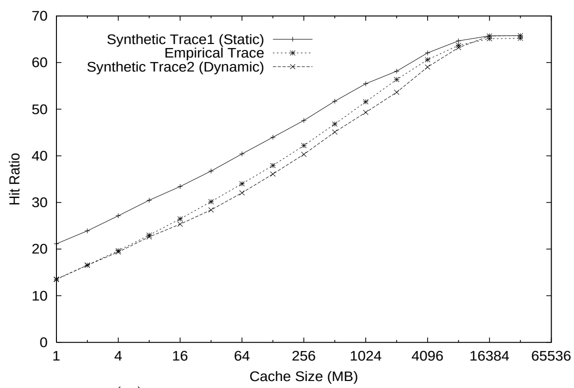
Nevertheless, the byte hit ratio results for *Trace2* capture the qualitative behaviour of the empirical trace, and that on average *Trace2* provides better estimates of the byte hit ratio than does *Trace1*. Thus, the *dynamic stack model* is used as the default temporal locality model in most of the subsequent experiments.



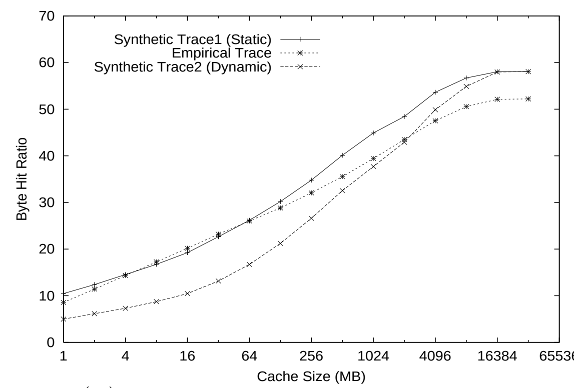
(a) LRU: Hit Ratio;



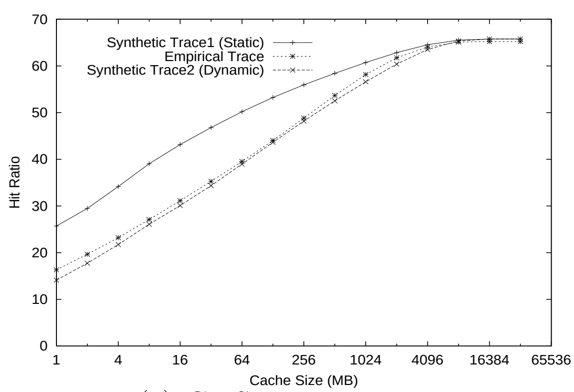
(b) LRU: Byte Hit Ratio



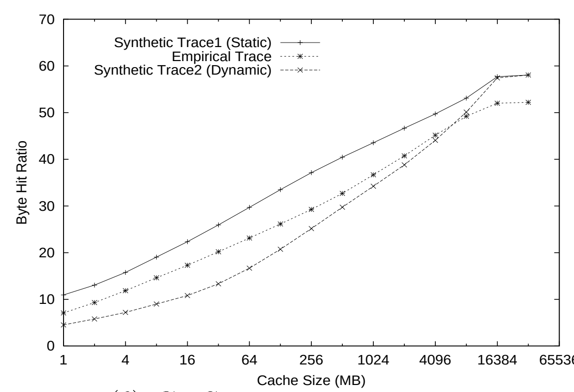
(c) LFU-Aging: Hit Ratio



(d) LFU-Aging: Byte Hit Ratio



(e) GDSize: Hit Ratio



(f) GDSize: Byte Hit Ratio

Figure 3.7: Comparison of Hit Ratios and Byte Hit Ratios Between Empirical and Synthetic Traces.

3.3 Summary

This chapter has presented the design of a proxy workload generator called ProWGen. ProWGen models selected workload characteristics that are expected to affect proxy cache performance, such as one-time referencing, file popularity, file size distribution, and temporal locality. In ProWGen, each characteristic is modeled with one or more controllable parameters to provide flexibility. Thus, the tool set can be used to synthesize workloads with a wide range of characteristics.

Validation of ProWGen was also discussed in this chapter. A sample empirical workload was used to validate a synthetic workload produced from the tool (parameterized to mimic the empirical one). The results of the validation experiments showed that ProWGen can generate workloads closely matching the characteristics configured using input parameters. Most importantly, the generated workloads can closely mimic the quantitative and qualitative characteristics of an empirical workload.

Flexibility is the key property of ProWGen. It allows the synthesis of workloads that differ in controlled ways in one or more characteristics, a property exploited by the experiments in the next chapter to evaluate the sensitivity of different cache replacement policies to different workload characteristics.

Chapter 4

Evaluation of Single-Level Caches

In this chapter, the focus is on evaluating cache management techniques in a single-level Web proxy server. The purpose is to address the following research question: *In a single-level proxy cache, how sensitive is Web proxy caching performance to certain workload characteristics (e.g., Zipf-ness, heavy-tail property, correlation, temporal locality), and how does the degree of sensitivity change depending on the cache replacement policy?* In other words, how important are the various characteristics observed in proxy workloads to caching strategies? Do these characteristics affect all replacement policies in the same or different ways? Are the current replacement policies exploiting these characteristics to enhance proxy performance? These are questions that will be answered by the experiments in this chapter.

These questions are answered using trace-driven simulations. In particular, ProWGen is used to generate workloads that differ in one characteristic at a time, and through trace-driven simulation, the behaviours of different cache policies under such workloads are determined.

The rest of the chapter is organized as follows. Section 4.1 presents the experimental methodology for the simulation experiments. Section 4.2 presents the simulation results, and Section 4.3 summarizes the chapter.

4.1 Experimental Methodology

This section describes the experimental methodology used for evaluating the sensitivity of proxy cache performance to workload characteristics. The simulation model is given in Section 4.1.1; factors and levels are described in Section 4.1.2; performance metrics are described in Section 4.1.3; and simulation validation is presented in Section 4.1.4.

4.1.1 Simulation Model

The simulation experiments in this chapter model a single Web proxy server, as shown in Figure 4.1. In the trace-driven simulation, all requests coming from the clients (i.e., the aggregate workload generated by ProWGen) are directed to the proxy server. The clients represent Web browsers that have been configured to make use of the proxy (though individual clients are not modeled in ProWGen). When the proxy receives a request from a client, it checks its cache to see if it has a copy of the requested file. If there is a copy in its cache, the proxy records a cache hit and returns the file to the user; otherwise the proxy records a cache miss, obtains a copy of the file from the (simulated) Web servers, stores a copy in its cache for future use, and returns a copy to the requesting client. If the cache is full when a file needs to be stored, then a replacement decision is triggered to decide which file (or files) to remove. Note that there are no coherence misses, since file modification events are not modeled in ProWGen; only cold misses (first reference to a file) and capacity misses (a previously referenced file which has been removed from the cache because of capacity constraints) can occur.

4.1.2 Factors and Levels

Three main factors were used in the trace-driven simulation experiments: cache size, replacement policy, and workload characteristics. The experiments employ a one-factor-at-a-time methodology. The factors and their levels are as follows:

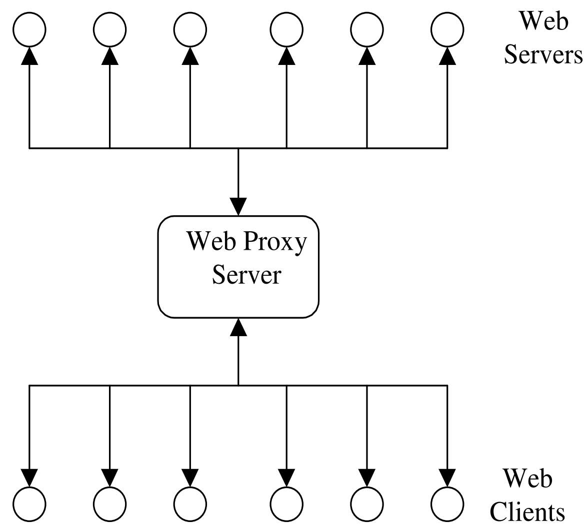


Figure 4.1: Single Web Proxy Server Simulation Model

4.1.2.1 Cache Size

Different cache sizes were used, from a minimum of 1 MB to a maximum of 16 GB. The maximum level for this factor represents an infinite cache size for the workloads considered (i.e., enough cache space to store all requested files without any replacements), so that the performance of a smaller cache size could be compared to that of an infinite cache.

4.1.2.2 Cache Replacement Policy

Three different cache replacement policies are considered: LRU, LFU-Aging, and GD-Size. These policies function as follows:

LRU The LRU policy is a recency-based policy that assigns priority to files based on their recency of reference only. When a replacement decision is required, LRU removes from the cache the file that has not been referenced for the longest period of time (i.e., Least Recently Used).

LFU-Aging LFU-Aging is a frequency-based policy, which tries to identify and keep popular documents in its cache. To describe the LFU-Aging policy, one must first describe the LFU policy. The LFU policy keeps a reference count for each file in the cache. Each time a requested file is found in the cache, the reference count is incremented. If a requested file is not found in the cache and there is the need to evict some files from the cache to make room for the incoming one, the file with the lowest reference count (Least Frequently Used) is removed. If two or more files have the same (lowest) reference count, LRU is used to break the tie. LFU-Aging is LFU plus an aging policy designed to address one of the shortcomings of LFU - "cache pollution." With the LFU policy, some files, having been referenced many times, build up a high reference count and are rarely chosen for replacement, even if these files are never requested again. What the aging policy does is to reduce the reference count of each file in the cache by one-half when the average reference count of all files in the cache reaches a specified limit. With this approach, the reference count of files not referenced for a long period of time decreases over time, making them possible candidates for removal.

GD-Size GD-Size is a size-based policy, which tries to keep "small" files in the cache. This policy, proposed by Cao and Irani [23], associates a value H with each file brought into the cache. The version of this algorithm implemented here sets H to $1/s$ where s is the size of the file. Whenever there is need to evict a file from the cache, the file with the lowest H value, say H_{min}, is selected for removal, and the H values of all other files in the cache are reduced by H_{min}. When a cache hit occurs, the H value of the file is restored to its original value. See [23] or [49] for a description of an algorithm that implements this policy efficiently.

4.1.2.3 Workload Characteristics

Five different workload characteristics are used in the simulation experiments. Each characteristic is represented by a controllable parameter in ProWGen, allowing workloads with different degrees of each characteristic to be synthesized. The selected characteristics are described below.

One-time referencing. Two different settings were used for the parameter controlling the one-time referencing characteristic in the workloads. These settings were 60% and 70% one-timers (expressed as a percentage of the number of unique files referenced). These values reflect typical one-time referencing behaviour seen in empirical workloads.

Zipf-ness. Workloads exhibiting three different levels of Zipf-ness were used. The parameter of the Zipf distribution β was fixed at 0.60 (low), 0.75 (medium), and 0.95 (high) to synthesize the workloads used to examine the sensitivity of the replacement policies to this characteristic.

Heavy-tail index. The tail index of the Pareto distribution controls the tail behaviour of the heavy-tailed file size distribution in proxy workloads. The tail index is a controllable parameter in ProWGen. Three different workloads with tail index of 1.2 (heavier heavy-tail), 1.3 (medium heavy-tail), and 1.4 (lighter heavy-tail) were used to examine the effect of this characteristic.

Correlation. Three different synthetic workloads exhibiting zero, positive, and negative correlation between file popularity and file size were used to examine how cache management policies would be affected. While the zero correlation is arguably the closest to reality, varying this parameter can establish the sensitivity of caching results to correlations, if they were present.

Temporal Locality. Synthesized workloads with two different temporal locality models were used. One uses the *static* LRU stack approach and the other uses the *dynamic* LRU stack approach to incorporate temporal locality into the workloads.

As described in Chapter 3, ProWGen requires twelve parameter settings to synthesize a workload. When the parameters for the five selected characteristics are varied from one experiment to the next to synthesize different workloads, the remaining parameters were kept constant at the values shown in Table 4.1.

Table 4.1: Parameters Kept Constant for Generating Synthetic Workloads in Single Proxy Simulation Experiments.

Parameter	Value
Total number of requests	5,000,000
Percentage of unique files (of total requests)	30%
Percentage of files in the tail (of unique files)	20%
Beginning of the tail (k) in bytes	10,000
Mean of the lognormal distribution (μ)	7,000
Standard deviation of the lognormal distribution (σ)	11,000
LRU Stack Size for temporal locality model	1,000

4.1.3 Performance Metrics

Two performance metrics are used to evaluate the three replacement policies, namely hit ratio and byte hit ratio.

4.1.3.1 Hit Ratio

The hit ratio is the ratio of the number of requests satisfied by the proxy's cache to the overall total number of requests seen by the proxy. The higher the hit ratio, the better a replacement policy is, because this means that fewer requests are forwarded to the Web servers, thus reducing the loads on the Web servers and on the network.

4.1.3.2 Byte Hit Ratio

The byte hit ratio is the total volume of data (in bytes) satisfied by the proxy's cache to the overall total volume of data requested from the proxy. The higher the byte hit ratio, the better a replacement policy is. A higher byte hit ratio means a lower volume of data flowing between the proxy and the Web servers, thus reducing the bandwidth consumed on the network between the proxy and the Web servers.

4.1.4 Validation of Simulator

Validation is an important step in any simulation study. For credibility in the simulation results, it is important to ensure that the simulator performs as expected. The simulator for the single server model was validated by running it with a small number of requests, which were then verified by hand to confirm that the hit ratios and byte hit ratios obtained were the same as expected. To establish more confidence in the simulator, the results obtained were compared with the results from the simulators used in [49].

4.1.5 Other Experimental Issues

In most of the simulation experiments in this chapter, a warm-up period of 500,000 requests was used to allow the simulated cache to reach steady state. That is, cold misses due to an empty cache are ignored during the warm-up period, and caching performance results (i.e., hit ratios and byte hit ratios) are reported only for the remainder of the simulation run, following the warm-up period. The length of the warm-up period (10% of the total trace) is consistent with that recommended in [8].

The temporal locality experiments, however, did not use a warm-up period. The reason for this decision is that the goal of the experiment was to investigate the impact of the order in which the requests appear in the workloads. Understanding and assessing this behaviour for the entire trace (even with a cold cache) was deemed more appropriate in this case. This approach also makes it easier to verify that each caching policy provides the same performance for the workloads on an infinite cache size, when no cache replacement decisions occur.

4.2 Simulation Results

4.2.1 Sensitivity to One-timers

Two synthetic workloads were used to examine the effect of one-timers on the three replacement policies. Characteristics of the synthetic workloads are shown in Table 4.2.

As shown in the first part of this table, one of the workloads (*Trace1*) has 60% one-timers, while the other (*Trace2*) has 70% one-timers. The analysis in the remaining part of the table shows that there is a close agreement between the input values and the measured output values of the generated workloads.

The popularity curves, the file size cumulative distribution, and the LLCD plots are shown in Figure 4.2. The traces are identical in most characteristics, except for the number of one-timers, which is impossible to see in Figure 4.2(a).

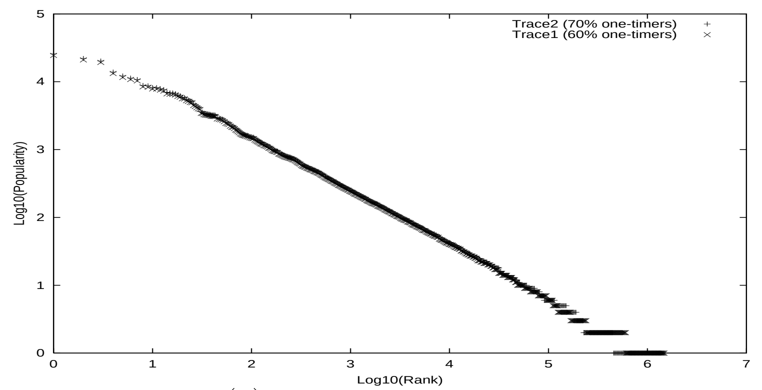
The performance results for the three replacement policies on these workloads are shown in Figure 4.3. The results show that the percentage of one-timers has little effect on the caching performance results. The difference in hit ratios and byte hit ratios is negligible at small cache sizes, and modest (1-4%) at larger cache sizes.

Surprisingly, caching performance is better for *Trace2* (70% one-timers) than for *Trace1* (60% one-timers). Intuitively, the expectation is that the replacement policies will perform better under *Trace1* because of the lower percentage of one-timers, but the result shown in Figure 4.3 is contrary to this intuition. In this figure, all the policies gave better performance in terms of both hit ratio and byte hit ratio for *Trace2*.

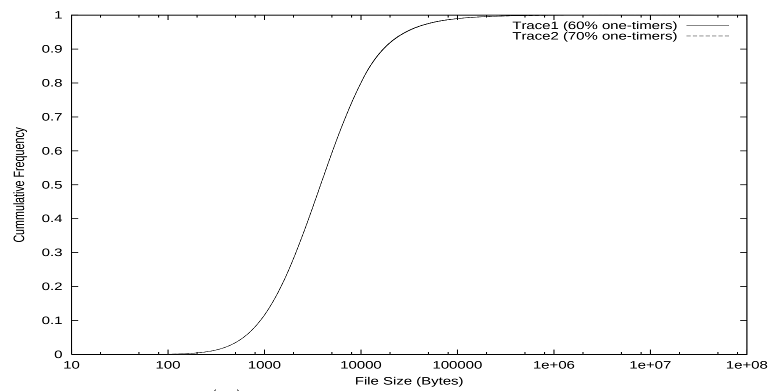
The reason for this behaviour is the higher concentration of references in *Trace2* than in *Trace1*. That is, because each trace has exactly the same number of references (as input to the workload generator), and the same number of documents (by construction), the traces differ only in how the references are distributed across the documents. Since *Trace2* has more one-timers (about 150,000 more) than *Trace1*, the number of references to non-one-timers is actually slightly higher on average, for *Trace2*, on a per document basis (e.g., 3.8 million references to 450,000 documents in *Trace2*, for an average of 8.5 per document, versus 4.0 million references to 600,000 documents in *Trace1*, for an average of 6.6 per document). This concentration difference is reflected in Figure 4.4, which plots the cumulative percentage of references versus the busiest fraction of the document set. From this figure, while 20% of the total unique files in *Trace2* account for 72% of the total references, the same

Table 4.2: Characteristics of the Workloads Used to Study the Effect of One-timers on Cache Replacement Policies.

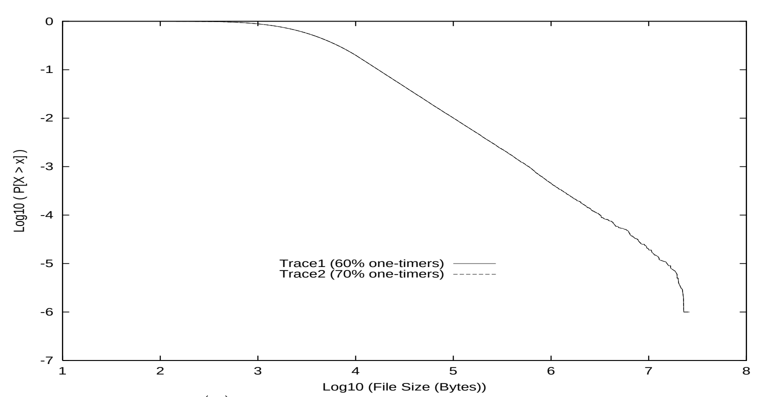
Item	Trace1	Trace2
Input to workload generator	"Fewer"	"More"
Zipf Slope	0.75	0.75
Tail index	1.3	1.3
Percentage of one-timers (of unique documents)	**60**	**70**
Correlation between file size and popularity	Zero	Zero
LRU Stack Model for temporal locality	Dynamic	Dynamic
Summary of the workload generated		
Total number of requests	4,866,266	4,886,334
Number of unique documents	1,500,000	1,500,000
Percentage of unique documents (of total requests)	30	30
Number of one-timers	898,882	1,049,393
Percentage of one-timers (of unique documents)	**59**	**69**
Total bytes of unique documents (GB)	14	14
Smallest file size (bytes)	13	13
Biggest file size (bytes)	27,987,309	27,987,309
Mean file size (bytes)	10,662	10,667
Median file size (bytes)	3,818	3,817
Correlation between file size and popularity	-0.004543	-0.004133
Zipf Slope	-0.779264	-0.780500
R^2	0.999705	0.998713
Tail index	-1.304922	-1.304177
R^2	0.999705	0.999706



(a) Popularity vs. Rank

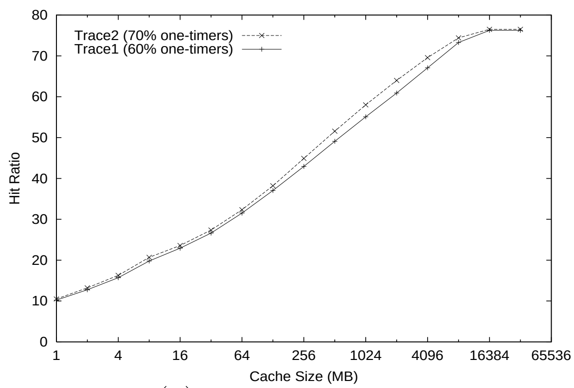


(b) CDF plots for file sizes

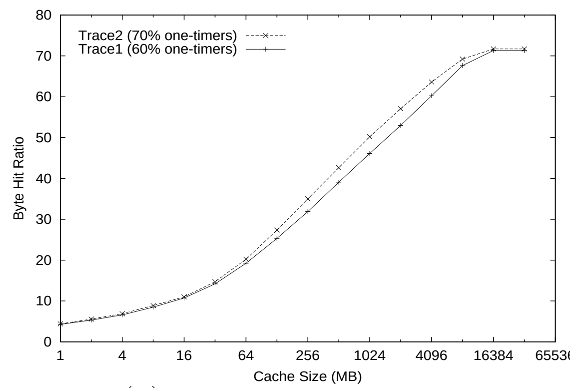


(c) LLCD plots for file sizes

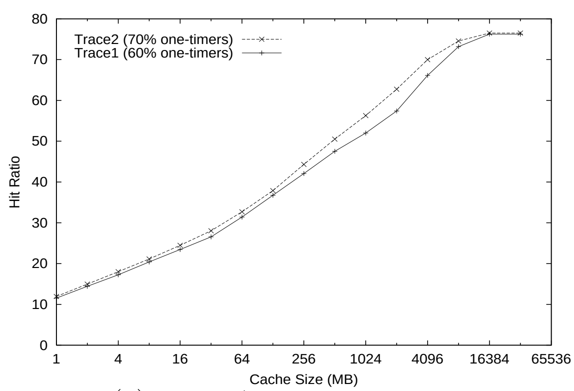
Figure 4.2: Popularity, CDF and LLCD Plots for the Synthetic Workloads Used to Study the Effects of One-timers on Cache Replacement Policies.



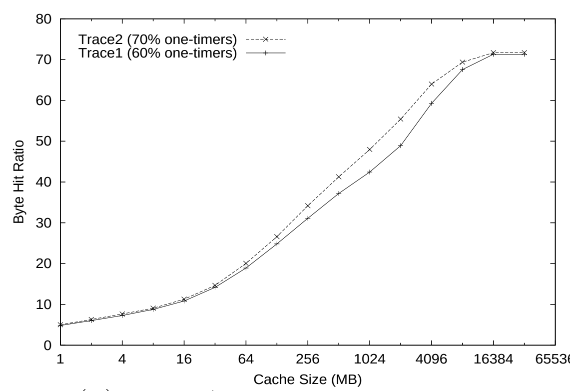
(a) LRU: Hit Ratio



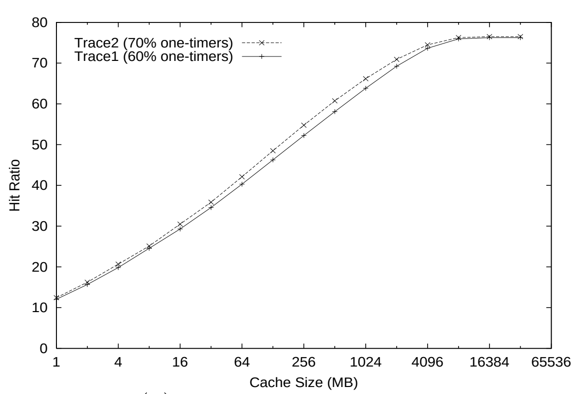
(b) LRU: Byte Hit Ratio



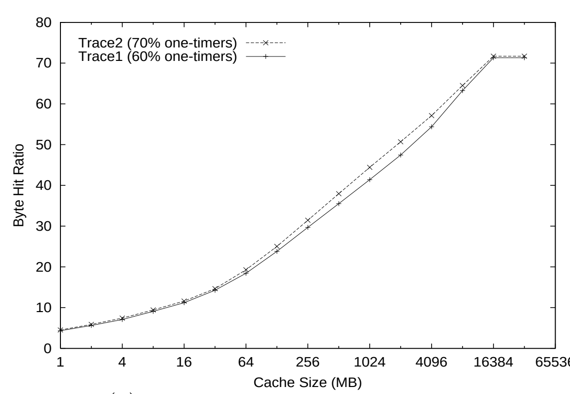
(c) LFU-Aging: Hit Ratio



(d) LFU-Aging: Byte Hit Ratio



(e) GDSize: Hit Ratio



(f) GDSize: Byte Hit Ratio

Figure 4.3: Sensitivity of Cache Replacement Policies to Changes in the Number of One-timers.

percentage of unique files in *Trace1* is responsible for 69% of the total references.

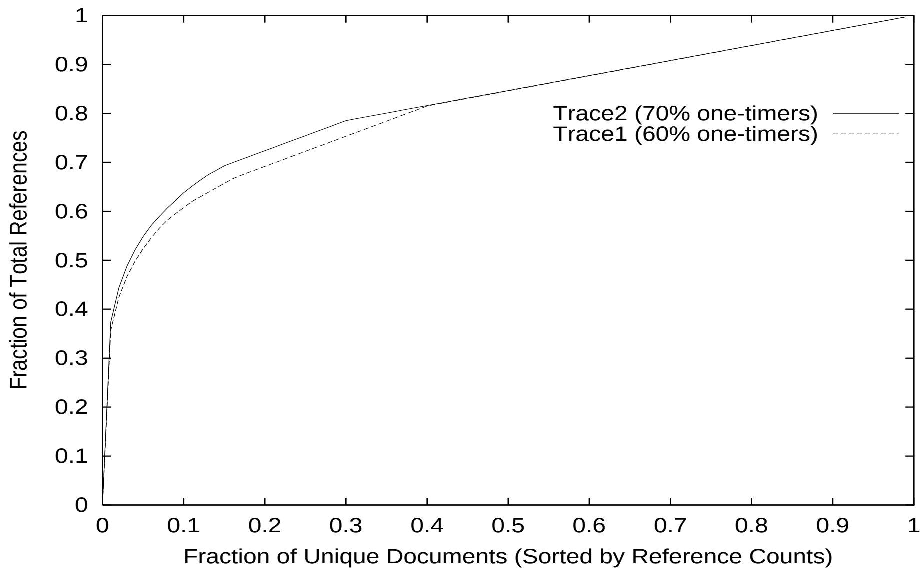


Figure 4.4: Concentration of References for Trace1 and Trace2.

In general, most "reasonable" caching algorithms are not very sensitive to the percentage of one-timers. In fact, assuming the algorithm can deal with one-timers reasonably effectively, a higher percentage of one-timers can lead to better caching performance, since there are lots of good candidates for removal. The LFU-Aging algorithm is most effective at this, since one-timers are immediate candidates for removal from the cache. Thus its performance increase is slightly greater than that of LRU and GD-Size (see Figure 4.3).

In a series of other experiments carried out to investigate the effect of increasing the percentage of one-timers (up to 90%), the same pattern of behaviour as in Figure 4.3 was observed. That is, as the percentage of one-timers increases[1], the replacement policies (even *random* and *first-in first-out* (FIFO) policies) perform better with workloads having a higher percentage of one-timers.

Another interesting observation from these experiments is that, at lower percentage of one-timers, LFU-Aging performs worse than LRU. However, as the per-

[1] A slight increase in the Zipf slope was noted as well. For 90% one-timers, the Zipf slope was 0.7858, compared to 0.7805 for 70% one-timers.

centage of one-timers increases, LFU-Aging starts to outperform all other policies except GD-Size. In fact, at 90% one-timers, LFU-Aging gave the best performance in byte hit ratio, and the second best, after GD-Size, in terms of hit ratio.

4.2.2 Sensitivity to Zipf Slope

In order to examine how sensitive the three replacement policies are to the Zipf-like referencing behaviour, three workloads (*Trace1, Trace2*, and *Trace3*) were synthesized using ProWGen with different values for the Zipf exponent β. Table 4.3 presents the parameter settings and the analysis of the generated workloads.

As shown in the first part of Table 4.3, the only parameter that differs for the three synthetic traces is the Zipf slope. *Trace1* has a low (flat) slope value of 0.6, *Trace2* has a slope of 0.75, while *Trace3* has a (high) slope of 0.95. Other parameters were kept constant.

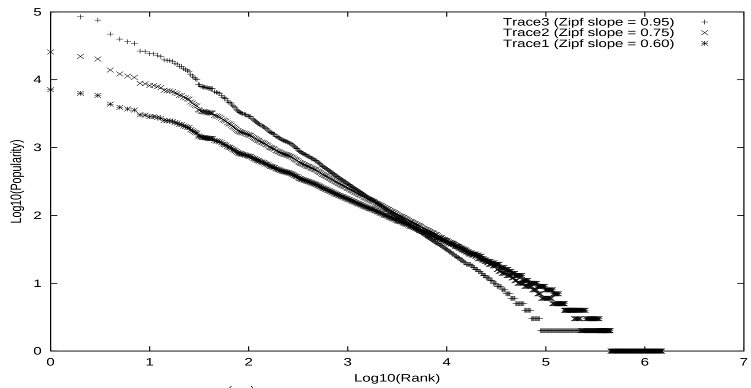
In the summary part of the table, the values reported for the three traces closely agree with the input values. The total number of requests generated, the percentage of distinct files, percentage of one-timers, and the correlation between file size and popularity, all show close agreement with the values supplied to the workload generator. Also shown on the table is the estimated Zipf slope and the heavy tail index. These values estimated from the synthesized workloads are very close to the desired values.

Several graphical characteristics of the synthetic workloads are presented in Figure 4.5. Figure 4.5(a) shows the popularity curves, the slopes of which were estimated using least-squares regression. The goodness of the fit is evident from the estimated value of R^2 in Table 4.3. The file size distributions are shown in Figure 4.5(b) for the three workloads; the three lines overlap since this characteristic has not been altered in the three traces. The LLCD plots are shown in Figure 4.5(c). The perfect overlap indicates the tail behaviour is the same for the three traces. The estimated tail index and the R^2 values are shown in Table 4.3.

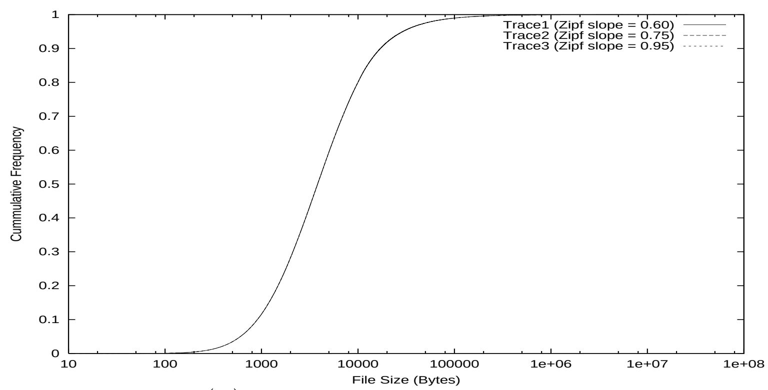
The three workloads were presented (one at a time) to the single server simulation

Table 4.3: Characteristics of the Workloads Used to Study the Effect of Zipf Slope on Cache Replacement Policies.

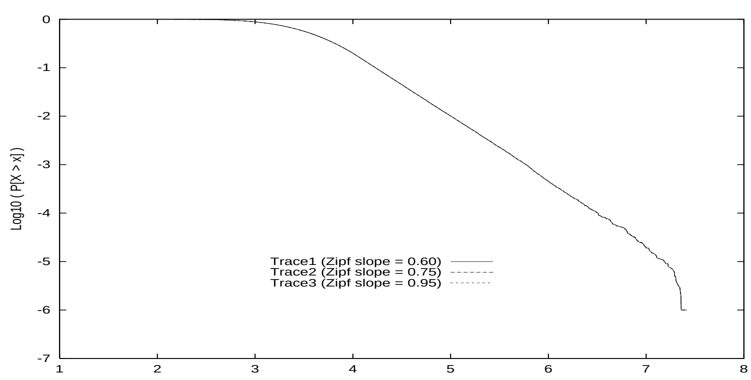
Item	Trace1	Trace2	Trace3
Input to workload generator	"Low"	"Medium"	"High"
Zipf Slope	**0.60**	**0.75**	**0.95**
Tail index	1.3	1.3	1.3
Percentage of one-timers (of unique documents)	70	70	70
Correlation between file size and popularity	Zero	Zero	Zero
LRU Stack Model for temporal locality	Dynamic	Dynamic	Dynamic
Summary of the workload generated			
Total number of requests	4,744,401	4,886,334	4,944,913
Number of unique documents	1,500,000	1,500,000	1,500,000
Percentage of unique documents (of total requests)	31	30	30
Number of one-timers	1,049,393	1,049,393	1,049,393
Percentage of one-timers (of unique documents)	69	69	69
Total bytes of unique documents (GB)	14	14	14
Smallest file size (bytes)	13	13	13
Biggest file size (bytes)	27,987,309	27,987,309	27,987,309
Mean file size (bytes)	10,672	10,667	10,660
Median file size (bytes)	3,819	3,817	3,817
Correlation between file size and popularity	-0.001297	-0.004133	-0.006223
Zipf Slope	**-0.639521**	**-0.780500**	**-0.976656**
R^2	0.996593	0.998713	0.998146
Tail index	-1.304420	-1.304177	-1.304375
R^2	0.999706	0.999706	0.999699



(a) Popularity vs. Rank



(b) CDF plots for file sizes



(c) LLCD plots for file sizes

Figure 4.5: Popularity, CDF and LLCD Plots for the Synthetic Workloads Used to Study the Effects of Zipf Slope on Replacement Policies.

model of Figure 4.1. The caching performance results for the three replacement policies on these workloads are shown in Figure 4.6.

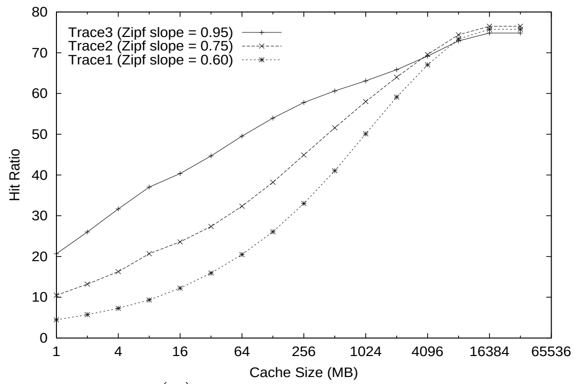
Figure 4.6 shows the sensitivity of the three cache replacement policies to the slope of the Zipf-like distribution. Figures 4.6(a) and (b) show the performance for the LRU policy, Figures 4.6(c) and (d) show the performance for the LFU-Aging policy, and Figures 4.6(e) and (f) show the results for the GD-Size policy.

A general trend observed from the hit ratios for the three policies ((a), (c), and (e)) is that as the exponent of the Zipf formula increases, the hit ratio increases as well, i.e., the steeper the slope of the popularity curve, the higher the hit ratio. This observation persists across the three replacement policies.

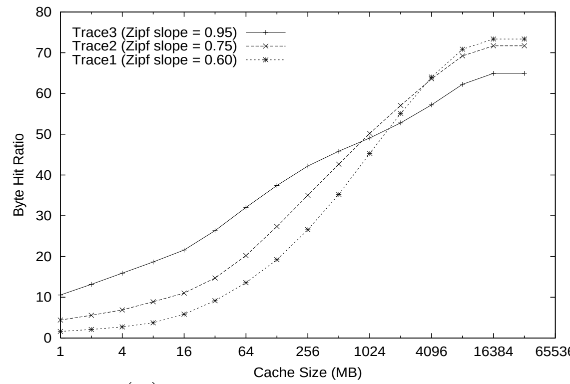
The reason for this behaviour is an increase in concentration of references as the slope becomes steeper. This concentration of reference behaviour can be explained from Figure 4.7, which plots the cumulative percentage of references accounted for by larger and larger fraction of the busiest files. In this graph, differences are particularly evident between 0 and 0.3 on the horizontal axis. Considering what fraction of the unique files in each trace account for 60% of the total references, it turns out that it takes 3% of the unique files in *Trace3*, 8% of the unique files in *Trace2*, and about 12% of the unique files in *Trace1*. In other words, the concentration of references is highest in *Trace3*, followed by *Trace2*, and then *Trace1*. This behaviour is also noticeable in Figure 4.5(a), where the busiest files in *Trace3* receive almost 100,000 requests as compared with fewer than 10,000 requests in *Trace1*. In simple terms, the steeper the Zipf-like popularity curve, the higher is the concentration of references, which translates into better caching performance.

A similar pattern of behaviour can be seen in the byte hit ratio graphs. That is, a steeper Zipf slope leads to a higher byte hit ratio, at least for small to medium cache sizes.

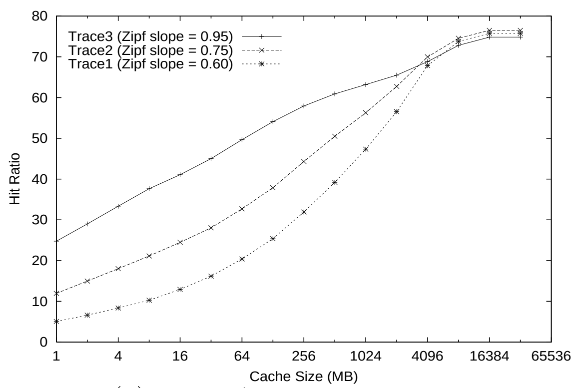
However, there is a distinct crossover effect in the byte hit ratio graphs, such that the opposite is true for large cache sizes (i.e., steeper Zipf slope provides worse byte



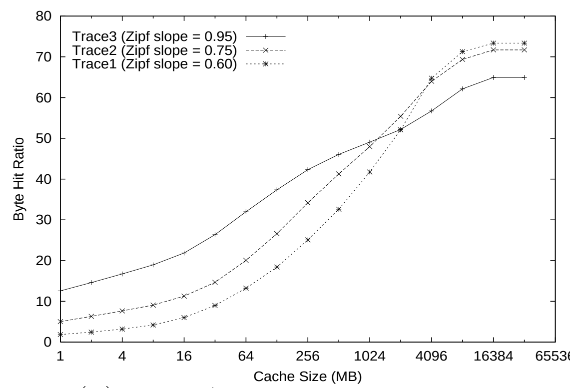
(a) LRU: Hit Ratio



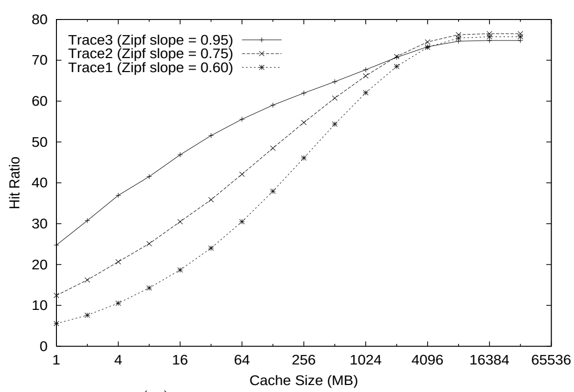
(b) LRU: Byte Hit Ratio



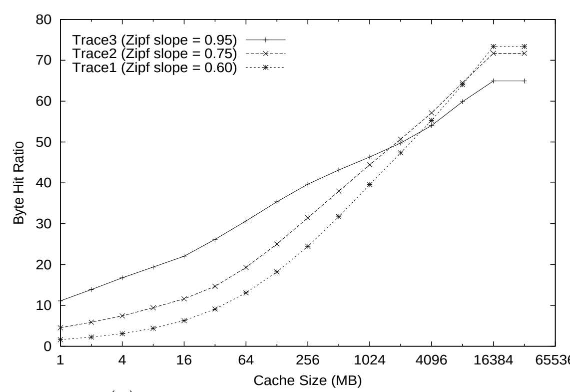
(c) LFU-Aging: Hit Ratio



(d) LFU-Aging: Byte Hit Ratio



(e) GDSize: Hit Ratio



(f) GDSize: Byte Hit Ratio

Figure 4.6: Sensitivity of Cache Replacement Policies to Variation in Zipf Slope.

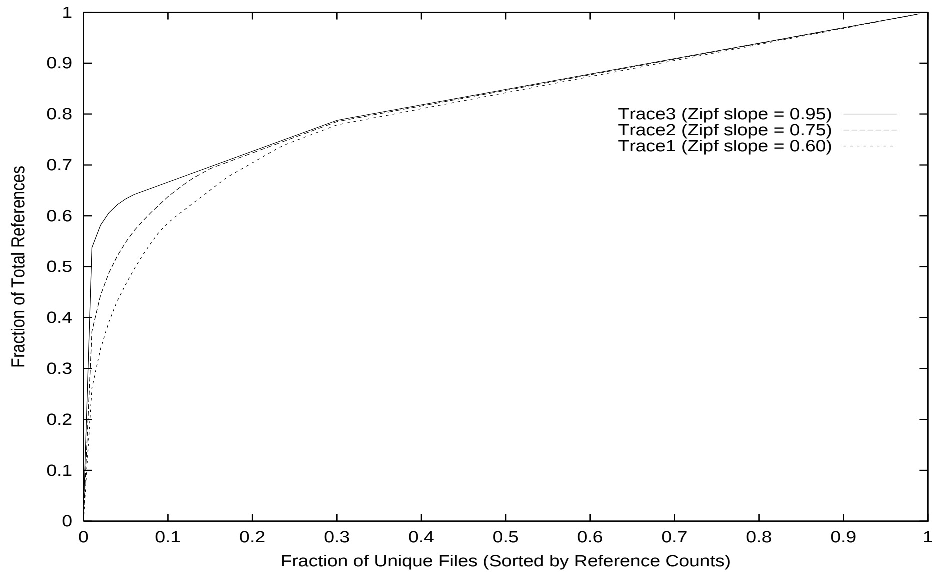


Figure 4.7: Concentration of References for Trace1, Trace2, and Trace3.

hit ratio[2]). This behaviour can be explained using the concentration of reference behaviour as well, combined with knowledge about file size distribution. A steeper slope means a very rapidly declining popularity: in *Trace3*, the popularity of files declines more rapidly than in *Trace2* and *Trace1*. In *Trace1*, the popularity decline is slowest, which means that more files (including some large files), can have similar (higher) popularity, and thus represent a larger share of the popularity distribution than in *Trace3*, for example, with a rapidly declining slope. At small cache sizes, few large files can fit in the cache (either at all, or before being evicted) for very long, but as the cache size increases, large files can stay in the cache longer. If they are re-referenced before being evicted, they contribute significantly to the byte hit ratio. The (same) large files in *Trace3*, even though they can stay longer in the cache as the cache size increases, tend to have fewer references than in *Trace1* or *Trace2* (because of Zipf slope), and thus contribute less significantly to the byte hit ratio.

In summary, an increase in the Zipf slope generally translates into better caching

[2]The crossover is weakly evident in the hit ratio results also.

performance, except for the crossover effect for byte hit ratio at large cache sizes (due to the sensitivity of byte hit ratio to the number of large popular files). All three cache replacement policies seem equally affected by the Zipf slope.

4.2.3 Sensitivity to Heavy Tail Index

The next experiment examines the sensitivity of the three replacement policies to the heavy tail index. Three workloads were synthesized differing only in the values of the tail index parameter as shown in Table 4.4. Tail index values of 1.2, 1.3 and 1.4 were used to synthesize *Trace1*, *Trace2* and *Trace3* to reflect "heavier", "medium", and "lighter" heavy-tailed file size distributions in the workloads, respectively.

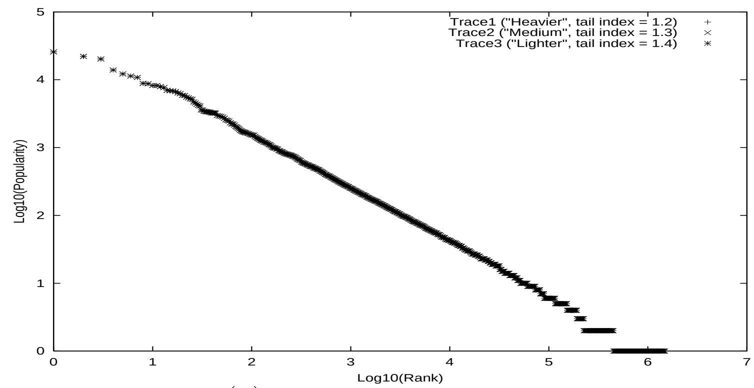
The rest of Table 4.4 presents other characteristics of the three workloads, which all show close agreement with the input values in the first part of the table. One thing to note from the table is the impact of the tail index parameter on the largest file size generated in each trace. *Trace1* has a 42 MB file in its workload, while *Trace2* has 27 MB and *Trace3* has 20 MB as the largest file. The impact of the heavy-tail can also be seen in the mean file size for each trace, as shown in the table. The median file size is unaffected.

Figure 4.8 presents a graphical synopsis of the three workloads, and how they differ. In Figure 4.8(a), the popularity graph for the three traces is shown with the fitted regression line. The perfect overlap shows that there is no difference in terms of the Zipf behaviour for the traces. Figure 4.8(b) shows the cumulative file size distribution with a slightly noticeable difference towards the tail – an indication of the difference in the heavy tail index. This difference is shown more clearly in the LLCD plot in Figure 4.8(c). The estimated tail index values are shown in this figure and in Table 4.4. They all agree closely to the desired values. The regression lines were omitted from Figure 4.8(c) for clarity.

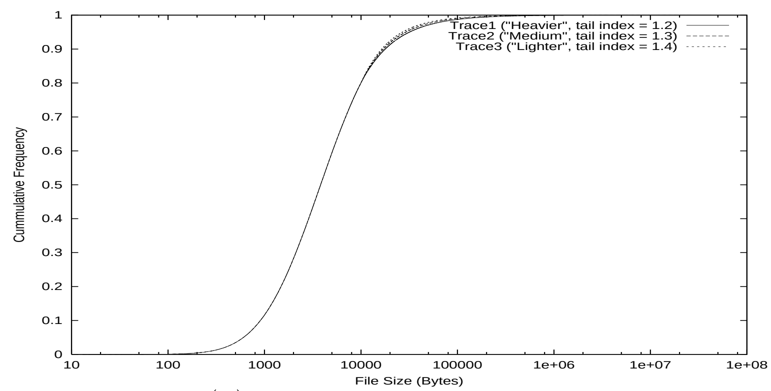
Figure 4.9 shows the performance results of the three cache replacement policies on these workloads. In terms of hit ratios (Figures 4.9(a), (c), and (e)), the tail index has little impact on the three caching policies. The reasons for this are twofold: (1)

Table 4.4: Characteristics of the Workloads Used to Study the Effect of Heavy Tail Index on Cache Replacement Policies.

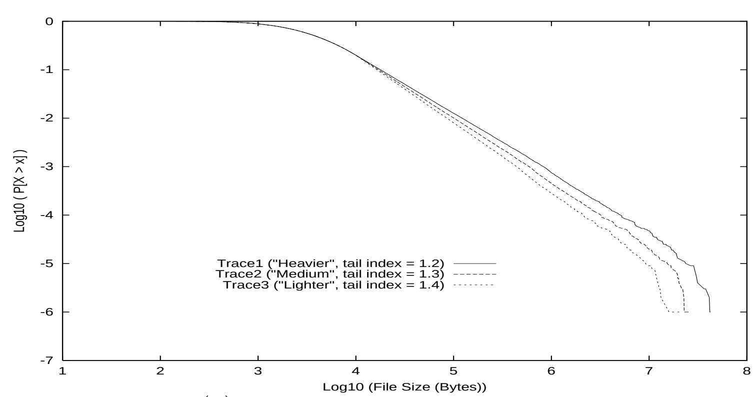
Item	Trace1	Trace2	Trace3
Input to workload generator	"Heavier"	"Medium"	"Lighter"
Zipf Slope	0.75	0.75	0.75
Tail index	**1.2**	**1.3**	**1.4**
Percentage of one-timers (of unique documents)	70	70	70
Correlation between file size and popularity	Zero	Zero	Zero
LRU Stack Model for temporal locality	Dynamic	Dynamic	Dynamic
Summary of the workload generated			
Total number of requests	4,886,334	4,886,334	4,886,334
Number of unique documents	1,500,000	1,500,000	1,500,000
Percentage of unique documents	30	30	30
Number of one-timers	1,049,393	1,049,393	1,049,393
Percentage of one-timers (of unique documents)	69	69	69
Total bytes of unique documents (GB)	17	14	13
Smallest file size (bytes)	13	13	13
Biggest file size (bytes)	42,493,776	27,987,309	20,744,557
Mean file size (bytes)	12,545	10,667	9,452
Median file size (bytes)	3,817	3,817	3,817
Correlation between file size and popularity	-0.004256	-0.004133	-0.003989
Zipf Slope	-0.780500	-0.780500	-0.780500
R^2	0.998713	0.998713	0.998713
Tail index	**-1.204699**	**-1.304177**	**-1.404271**
R^2	0.999649	0.999706	0.999736



(a) Popularity vs. Rank



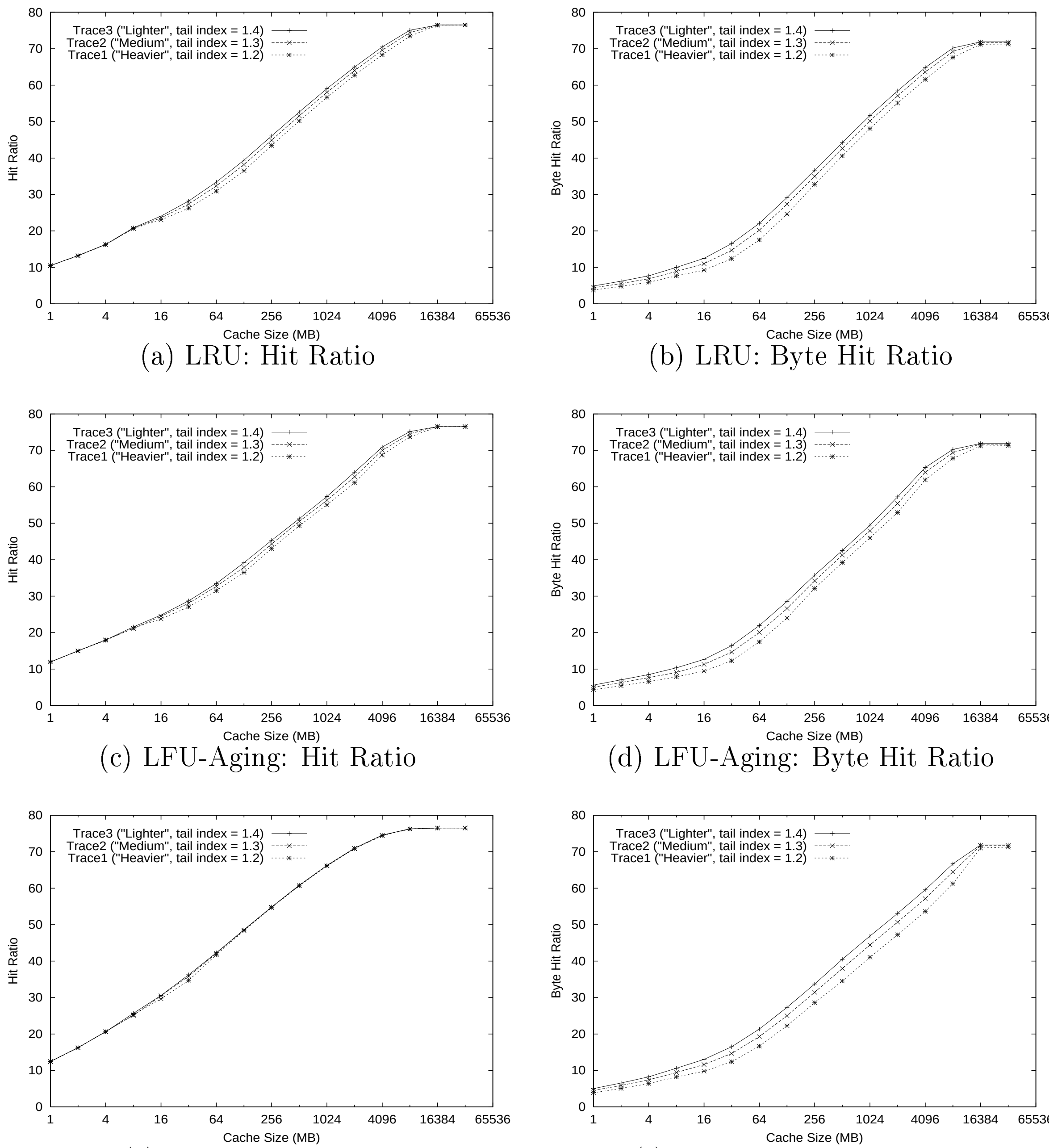
(b) CDF plots for file sizes



(c) LLCD plots for file sizes

Figure 4.8: Popularity, CDF and LLCD Plots for the Synthetic Workloads Used to Study the Effects of Heavy Tail Index on Replacement Policies.

the number of files affected by the tail index parameter is few; and (2) most of these files are so large that they do not fit (or stay long) in the cache, especially at small cache sizes.



(a) LRU: Hit Ratio

(b) LRU: Byte Hit Ratio

(c) LFU-Aging: Hit Ratio

(d) LFU-Aging: Byte Hit Ratio

(e) GDSize: Hit Ratio

(f) GDSize: Byte Hit Ratio

Figure 4.9: Sensitivity of Cache Replacement Policies to Variation in Heavy Tail Index.

As a general observation, all replacement policies provide lower hit ratios and lower byte hit ratios when the heaviness of the tail increases. That is, the heavier the tail (the smaller the value of the tail index), the lower the hit ratios and the byte hit ratios.

As the cache size increases, the impact of the heavy tail index becomes more pronounced. The impact on byte hit ratio is greater than the impact on hit ratio, but the impact is still modest. In general, the GD-Size policy shows the least sensitivity to the tail index in terms of hit ratio, while LFU-Aging and LRU are equally sensitive. For example, at 256 MB cache size, the difference in hit ratios for *Trace1* and *Trace3* for the LRU and the LFU-Aging policies is 2-3%, while it is 0.2% for the GD-Size policy. However, a different pattern of behaviour can be observed from the byte hit ratio graphs, where the GD-Size policy shows more sensitivity to the tail index than do LRU and LFU-Aging.

The behaviour of the GD-Size policy needs more explanation here. A natural question is: *why is GD-Size insensitive to changes in heavy tail index in terms of hit ratio but sensitive when it comes to byte hit ratio?* This question can be answered by recognizing that GD-Size is the only policy that takes file sizes into consideration when evicting files from the cache. Small files are favoured in its replacement decisions at the expense of large files, thus the GD-Size policy tends to keep more smaller files in the cache. The tail index parameter is a factor determining how large the files at the tail of the file size distribution are, but the GD-Size policy seems to be indifferent to this characteristic in terms of hit ratios as it does not keep large files in the cache long. Therefore, there is hardly a significant difference when traces with different heavy tail index are examined with this policy. The observed sensitivity of this policy in terms of byte hit ratio is the price it has to pay for its bias against large files.

Generally, this experiment has shown that the heavy tail property is not as important as the Zipf exponent in caching performance (at least over the range of parameter settings studied). While a difference of 0.2 in the Zipf exponent could translate into a 10-15% difference in hit ratio and byte hit ratio, a difference of 0.2

in the tail index changes the hit ratio and the byte hit ratio less than 3%.

4.2.4 Sensitivity to Correlation Between File Size and Popularity

The experiment in this section studies the impacts of correlation between file size and popularity on the different replacement policies. In particular, strong positive and strong negative correlation were studied, in addition to the normal close-to-zero correlation. Recall that positive correlation means that large files tend to be more popular, while negative correlation means that small files tend to be more popular. Zero correlation means that file popularity is unrelated to file size.

Three different workloads were synthesized to exhibit negative (*Trace1*), zero (*Trace2*), and positive (*Trace3*) correlation between file size and popularity. The characteristics of the synthesized workloads are presented in Table 4.5. The graphical characteristics of the workloads are shown in Figure 4.10, where (a) represents the popularity curve for the three traces, (b) is the file size distribution curve, and (c) is the LLCD plot. All three workloads show similar characteristics, with the only perceptible difference in the extreme tail of the file size distribution (since each workload generates file sizes in slightly different ways).

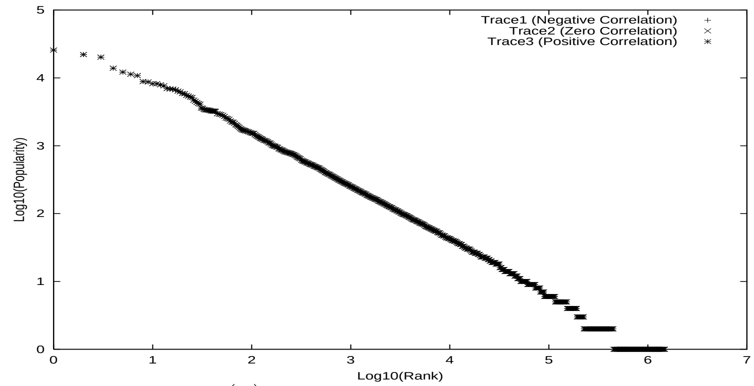
The performance results for the three replacement policies with these workloads are shown in Figure 4.11. These results show the dramatic impact of the correlation property on the caching performance.

One general observation is that all the policies considered provided better hit ratios, but worse byte hit ratios, for *Trace1*, which exhibits negative correlation. On the other hand, for *Trace3* with positive correlation, all the policies gave their worst hit ratio performance but the best byte hit ratio performance. For *Trace2*, the most realistic trace, the performance of the policies is always in-between the other two traces, in terms of both hit ratios and byte hit ratios.

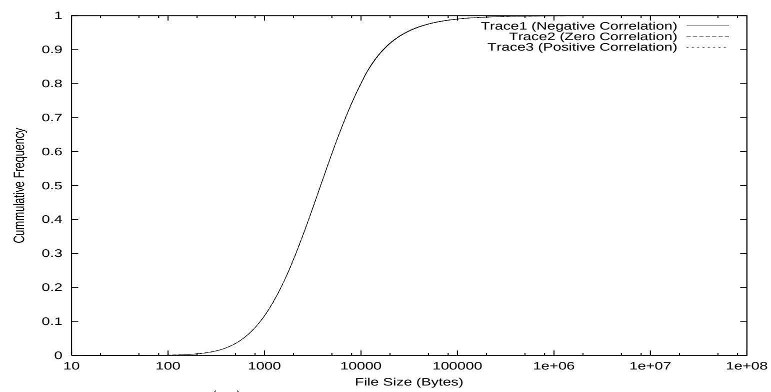
The observed behaviours are not unexpected. When there is negative correlation between file size and popularity, such as in *Trace1*, small files are very popular while

Table 4.5: Characteristics of the Workloads Used to Study the Effect of Correlation Between File Size and Popularity on Cache Replacement Policies.

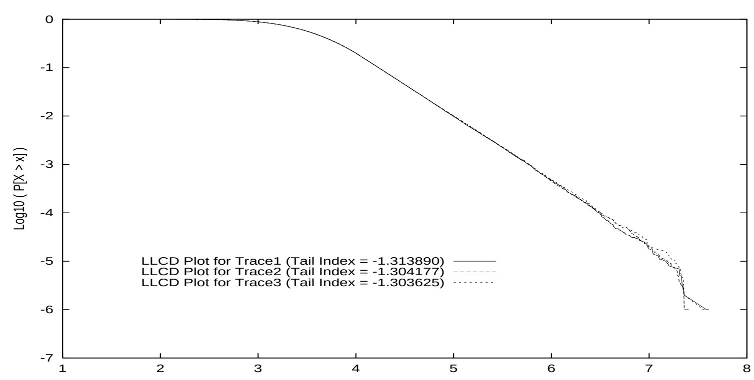
Item	Trace1	Trace2	Trace3
Input to workload generator	"Negative"	"Zero"	"Positive"
Zipf Slope	0.75	0.75	0.75
Tail index	1.3	1.3	1.3
Percentage of one-timers (of unique documents)	70	70	70
Correlation between file size and popularity	**Negative**	**Zero**	**Positive**
LRU Stack Model for temporal locality	Dynamic	Dynamic	Dynamic
Summary of the workload generated			
Total number of requests	4,886,334	4,886,334	4,886,334
Number of unique documents	1,500,000	1,500,000	1,500,000
Percentage of unique documents (of total requests)	30	30	30
Number of one-timers	1,049,393	1,049,393	1,049,393
Percentage of one-timers (of unique documents)	69	69	69
Total bytes of unique documents (GB)	14	14	15
Smallest file size (bytes)	14	13	13
Biggest file size (bytes)	44,381,786	27,987,309	40,027,158
Mean file size (bytes)	10,598	10,667	10,792
Median file size (bytes)	3,817	3,817	3,828
Correlation between file size and popularity	**-0.734314**	**-0.004133**	**0.809784**
Zipf Slope	-0.780500	-0.780500	-0.780500
R^2	0.998713	0.998713	0.998713
Tail index	-1.313890	-1.304177	-1.303625
R^2	0.999803	0.999706	0.999851



(a) Popularity vs. Rank

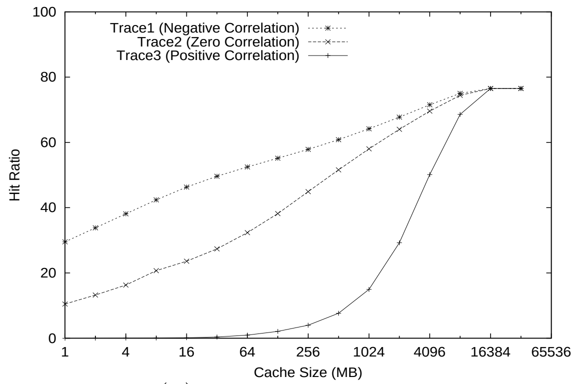


(b) CDF plots for file sizes

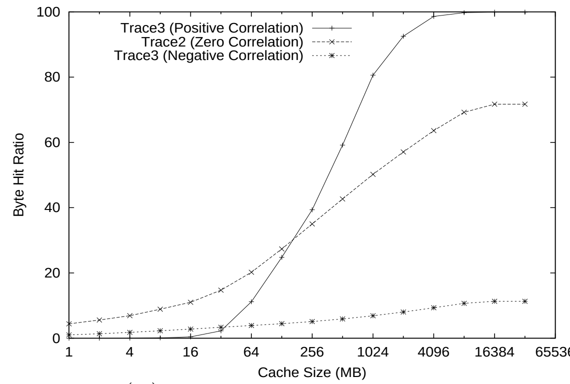


(c) LLCD plots for file sizes

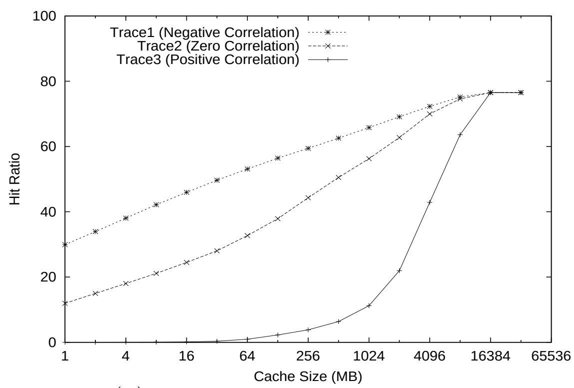
Figure 4.10: Popularity, CDF and LLCD Plots for the Synthetic Workloads Used to Study the Effects of Correlation Between File Size and Popularity on Replacement Policies.



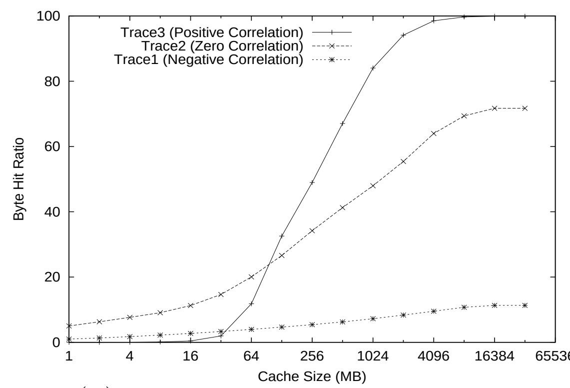
(a) LRU: Hit Ratio



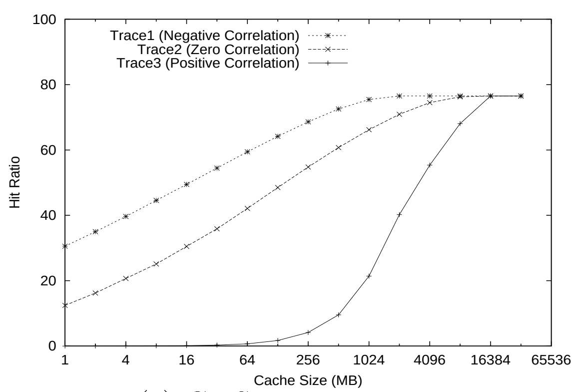
(b) LRU: Byte Hit Ratio



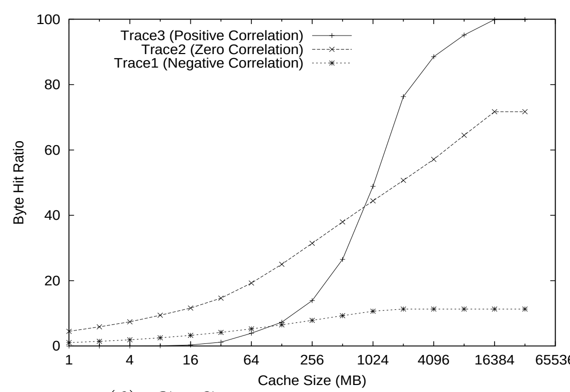
(c) LFU-Aging: Hit Ratio



(d) LFU-Aging: Byte Hit Ratio



(e) GDSize: Hit Ratio



(f) GDSize: Byte Hit Ratio

Figure 4.11: Sensitivity of Cache Replacement Policies to Correlation Between File Size and Popularity.

large files are not. The effect is that the small files, lots of which can fit in the cache, receive many re-references, producing a high reference hit ratio. The penalty for this success is a poor byte hit ratio, since the large files (responsible for many of the bytes transferred) are rarely accommodated in the cache, except at large cache sizes.

On the other hand, in a workload with positive correlation between file size and popularity, such as *Trace3*, small files have low popularities and large files have high popularities. It follows that at small cache sizes (say, up to 256 MB), the observed hit ratio is poor. Similarly, the byte hit ratio is poor (say, up to a cache size of 64 MB), because few of the large files with many references can be accommodated in the cache. The small files that do fit in the cache are not heavily re-referenced, while the large files with many re-references compete with each other for cache space.

The qualitative behaviour for the positive correlation trace is quite similar for all three caching policies. Beyond some critical cache size (e.g., above 1024 MB), the cache hit ratio increases sharply. The same pattern of behaviour is observed for the byte hit ratio, which at 64 MB starts increasing rapidly because large files now stored in the cache are being re-referenced, and these files contribute significantly to the total bytes transferred.

One additional observation about the byte hit ratio graphs for *Trace3* is that for an infinite cache, the byte hit ratio asymptotically approaches 100%. This result can be explained from the fact that the total content bytes of small files in the workload constitute a negligible fraction of the total bytes requested from the proxy server.

In summary, the presence of correlation between file popularity and file size can have a dramatic impact on caching performance (at least for the extreme correlation values considered).

4.2.5 Sensitivity to Temporal Locality

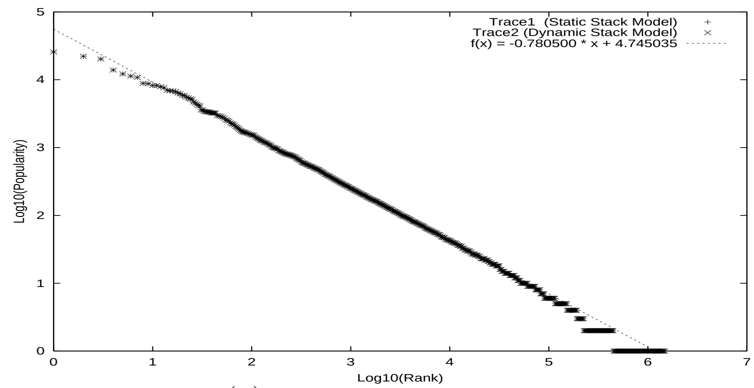
The next experiment is designated to assess the impact on caching performance of two different temporal locality models. The first workload (*Trace1*) uses the *static*

LRU stack model, while the second (*Trace2*) uses the *dynamic* LRU stack model. As indicated in Chapter 3, the static model seems to introduce "more" temporal locality into a trace than the dynamic model, perhaps producing optimistic estimates of caching performance. The experiments in this section seek to determine how sensitive the three replacement policies are to the degree of temporal locality in a trace.

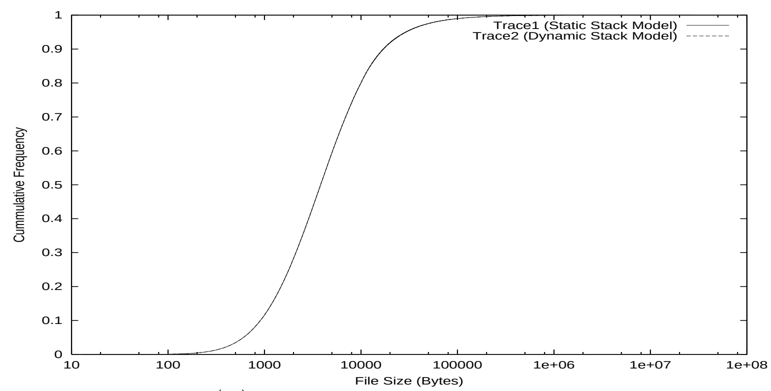
Table 4.6: Characteristics of the Workloads Used to Study the Effect of Temporal Locality on Cache Replacement Policies.

Item	Trace1	Trace2
Input to workload generator	"Higher"	"Lower"
Zipf Slope	0.75	0.75
Tail index	1.3	1.3
Percentage of one-timers (of unique documents)	70	70
Correlation between file size and popularity	Zero	Zero
LRU Stack Model for temporal locality	**Static**	**Dynamic**
Summary of the workload generated		
Total number of requests	4,886,334	4,886,334
Number of unique documents	1,500,000	1,500,000
Percentage of unique documents (of total requests)	30	30
Number of one-timers	1,049,393	1,049,393
Percentage of one-timers (of unique documents)	69	69
Total bytes of unique documents (GB)	14	14
Smallest file size (bytes)	13	13
Biggest file size (bytes)	27,987,309	27,987,309
Mean file size (bytes)	10,667	10,667
Median file size (bytes)	3,817	3,817
Correlation between file size and popularity	-0.004133	-0.004133
Zipf Slope	-0.780500	-0.780500
R^2	0.998713	0.998713
Tail index	-1.304177	-1.304177
R^2	0.999706	0.999706

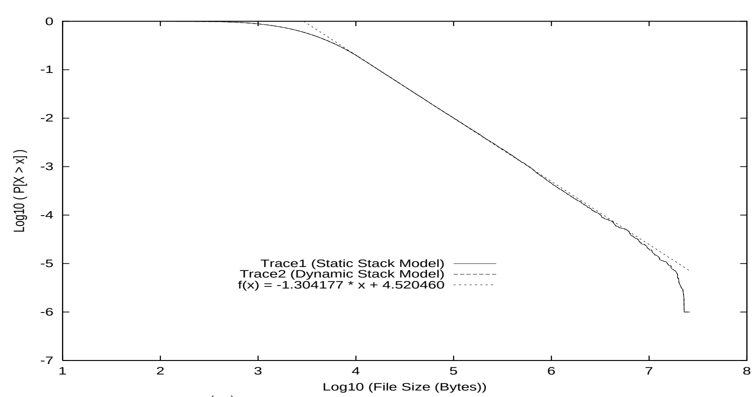
The characteristics of the two workloads used are presented in Table 4.6. The first part of the table shows that the only input parameter to ProWGen that differs is the *LRU stack model* parameter. As shown in the remaining part of the table, and in Figure 4.12, all the measured statistics of the two traces are the same, as expected.



(a) Popularity vs. Rank



(b) CDF plots for filesizes



(c) LLCD plots for filesizes

Figure 4.12: Popularity, CDF and LLCD Plots for the Synthetic Workloads Used to Study the Effects of Temporal Locality on Cache Replacement Policies.

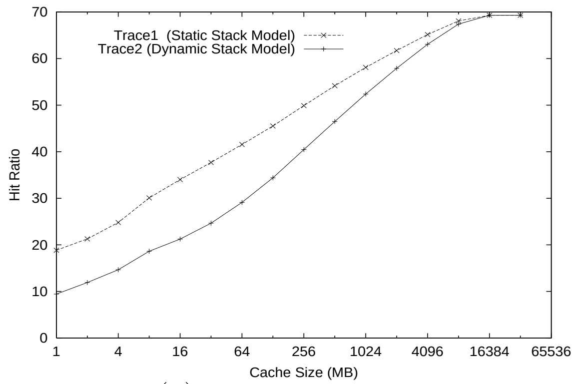
The temporal locality model used changes only the ordering of the requests in the traces.

Figure 4.13 shows the performance of the three cache replacement policies for the two synthetic traces. A general observation from the figure is that all the policies provided higher hit ratios and byte hit ratios for *Trace1*, which arguably has "higher" (but not necessarily more realistic) temporal locality.

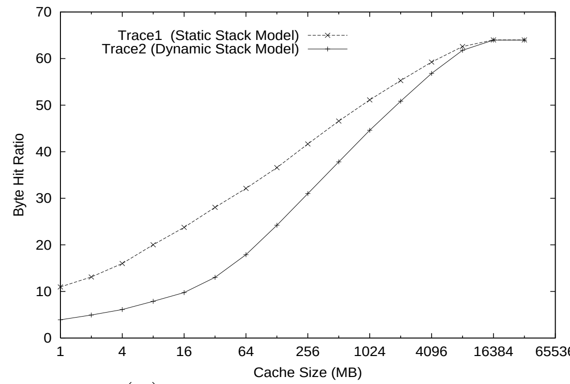
Another observation from Figure 4.13 is that the LRU policy seems to be the most sensitive to the degree of temporal locality in the traces, while the LFU-Aging policy is the least sensitive. For example, the difference in hit ratios for *Trace1* and *Trace2* for the LRU policy at 64 MB (Figure 4.13(a)) is 12.43%, while GD-Size has a difference of 12%, and LFU-Aging has a difference of 7.43%.

The performance differences between LRU and LFU-Aging are more pronounced on *Trace1* than on *Trace2*. For example, at a 64 MB cache size, LRU and LFU-Aging provide hit ratios of 41.55% and 36.87% for *Trace1*, respectively. At the same cache size on *Trace2*, LRU and LFU-Aging provide hit ratios of 29.12% and 29.44%, respectively. However, LRU is not even superior to GD-Size for *Trace1*, because at the same cache size, the hit ratio for GD-Size on *Trace1* is 49%.

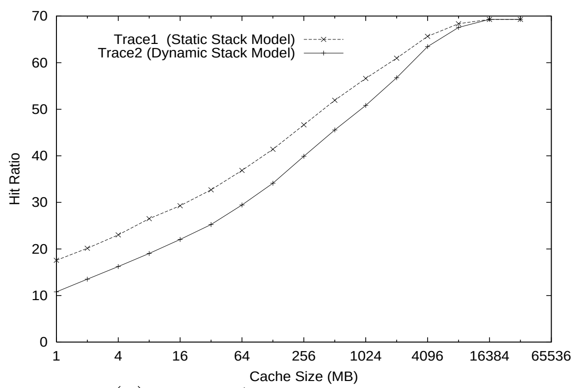
In general, the sensitivity results for GD-Size are similar to those for LRU. The behaviour of GD-Size can be explained from the way this policy operates when a new file is brought into the cache, or when a file already in the cache is re-referenced. In the former case, a newly accessed file has a cost H attached to it, which is calculated as $1/s$, where s is the size of the file. If the file is small, the H value computed for it may be high enough to place it at the head of the list of all files currently in the cache at that time; in this way, GD-Size mimics the LRU policy. In the latter case, where a file currently in the cache is re-referenced, GD-Size restores the file's H value to its original value. Similarly, if the size of the file is very small, it may end up at the head of all other files currently in the cache, thus behaving like the LRU policy again. LFU-Aging, on the other hand, may never dramatically move any file to the head of the list, whether the file is just coming into the cache for the first time or is already in the cache, unless all the files currently in the cache have



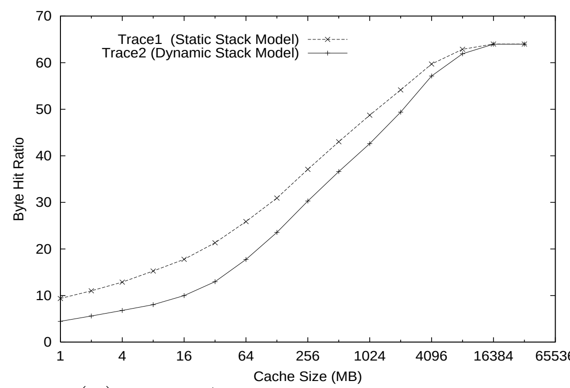
(a) LRU: Hit Ratio



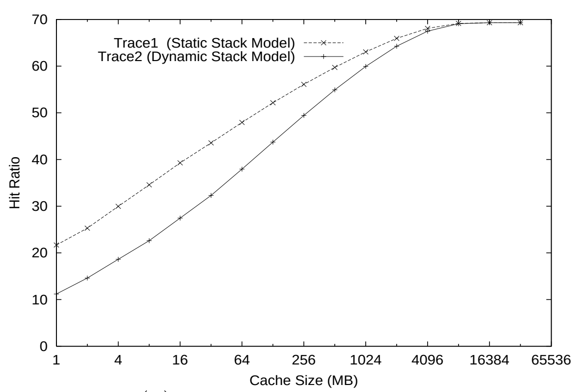
(b) LRU: Byte Hit Ratio



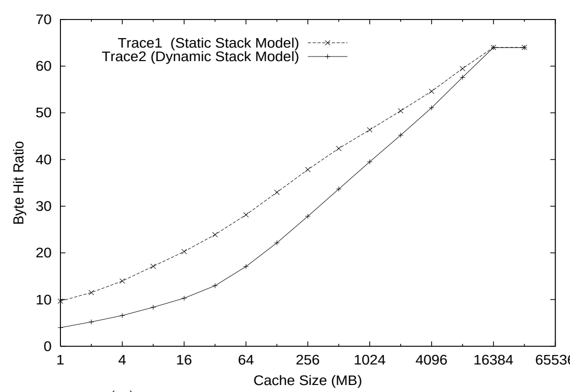
(c) LFU-Aging: Hit Ratio



(d) LFU-Aging: Byte Hit Ratio



(e) GDSize: Hit Ratio



(f) GDSize: Byte Hit Ratio

Figure 4.13: Sensitivity of Cache Replacement Policies to Temporal Locality.

a reference count of 1 when a new file is coming into the cache, or incrementing the reference count of a re-referenced file will make it as popular as the currently most popular file. This situation might not occur as frequently as in LRU or GD-Size, a reason why this policy shows the least sensitivity to temporal locality.

In general, all three policies are sensitive to temporal locality with LRU being the most sensitive, followed by GD-Size and then LFU-Aging. The higher the degree of temporal locality, the better the caching performance. The sensitivity follows the same pattern for both hit ratio and byte hit ratio.

4.3 Summary

The experiments in this chapter focussed on the sensitivity of cache replacement policies to the characteristics in Web proxy workloads. Several workloads were synthesized in which the characteristics were varied one at a time to evaluate the replacement policies.

The first experiment that studied the effect of one-timers indicates that there is little sensitivity. Generally, all cache management policies considered performed slightly better as the percentage of one-timers increased, because of a corresponding increase in the concentration of references.

From the simulation results, LRU, LFU-Aging and GD-Size show the same pattern of sensitivity to variation in Zipf slope. The general observation was that a slight difference in Zipf slope could mean a difference of 10-15% in hit ratios and byte hit ratios. The steeper the slope, the higher the cache hit ratios.

For the heavy tail index, GD-Size is the most insensitive (in terms of hit ratio), while LRU and LFU-Aging are equally sensitive. The heavier the tail of the file size distribution, the worse the caching performance of the policies, both in terms of hit ratio and byte hit ratio, with byte hit ratio showing greater sensitivity. In contrast to Zipf slope, tail index does not seem to have significant impact on caching policies.

The experiment on correlation indicates extreme sensitivity. All three policies provided excellent hit ratios, but poor byte hit ratios, for a workload exhibiting negative correlation between file popularity and file size. On the other hand, a workload with positive correlation provided the best byte hit ratios and the worst hit ratios.

In terms of temporal locality, the LRU replacement policy is the most sensitive, both in terms of hit ratio and byte hit ratio, to the degree of temporal locality in the workload.

In conclusion, the experiments in this chapter show how important some of the characteristics identified in proxy workloads are to caching policies and the extent to which each of them affects the different replacement policies. These results are summarized in Table 4.7. The next chapter explores caching sensitivity and tradeoffs in a multi-level Web proxy caching hierarchy.

Table 4.7: Summary of Sensitivity of Replacement Policies to Workload Characteristics.

Characteristic	LRU	LFU-Aging	GD-Size
One-timers	Slightly sensitive. The higher the number of one-timers the better the performance in hit ratio and byte hit ratio.	More Sensitive than LRU and GD-Size. The higher the number of one-timers the better the performance in hit ratio and byte hit ratio.	Same as LRU.
Zipf slope	Very Sensitive. Difference of 0.2 in Zipf slope can impact performance by 10-15%. The higher the Zipf slope, the better the performance in hit ratio and byte hit ratio due to increasing concentration of references.	Same as LRU.	Same as LRU.
Tail index	Low sensitivity. Difference of 0.2 in tail index can impact performance by less than 3%. The higher the tail index value (lighter heavy tail), the better the performance in hit ratio and byte hit ratio.	Same as LRU.	Insignificant sensitivity in terms of hit ratio (less than 1%), but a little more sensitivity than LRU and LFU-Aging in terms of byte hit ratio. Difference of 0.2 in tail index can impact performance by 3-5%. The higher the tail index value (lighter heavy tail), the better the performance in hit ratio and byte hit ratio.
Correlation between file size and popularity	Negative correlation: significantly higher hit ratios but worse byte hit ratios. Positive correlation: poor hit ratios at very small cache sizes but significant hit ratios and excellent byte hit ratios at large cache sizes. Zero correlation: moderate hit ratios and byte hit ratios, in-between the positive and negative correlation results.	Same as LRU.	Same as LRU.
Temporal locality	Very sensitive. More temporal locality, better performance. Could outperform LFU-Aging in the presence of high temporal locality.	Less sensitive than LRU and GD-Size. More temporal locality, better performance.	More sensitive than LFU-Aging but less sensitive than LRU. More temporal locality, better performance.

Chapter 5

Evaluation of Multi-Level Caches

The experiments in this chapter focus on how to improve caching techniques in a caching hierarchy. It uses trace-driven simulations, with synthetic workloads from ProWGen, to explore the following research questions:

Question 1 *In a multi-level caching hierarchy, can overall caching performance be improved by using different cache replacement policies at different levels of the hierarchy (e.g., LFU at the lowest level, LRU at the highest level)?*

Question 2 *In a multi-level caching hierarchy, can overall caching performance be improved by maintaining disjoint document sets at each level of the hierarchy (e.g., small documents at the lowest level, and large documents at the highest level)?*

5.1 Simulation Model

The simulation experiments consider a two-level hierarchical Web proxy configuration as shown in Figure 5.1. The clients interact directly with the lower level proxies (children), which in turn forward any misses to the upper level proxy (parent). The parent contacts an originating server if it cannot satisfy a request from its cache. Each time a new file is fetched from a Web server by the upper level proxy server, the caching strategy determines whether the file should be cached or not. A similar decision is made at the lower level proxies when a file is requested from the upper level proxy. It is important to note that there is no communication between peer lower level proxies.

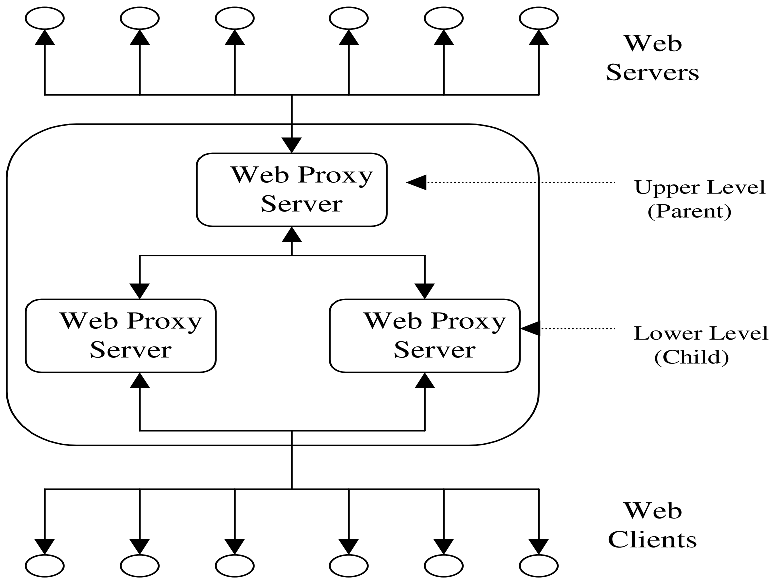


Figure 5.1: Two-Level Hierarchical Web Proxy Server Simulation Model

For the simulation model presented, the first difficulty lies in deciding how to synthesize the workload to be used for the experiments. Should individual workloads be synthesized for the two lower level proxies? Or does it make sense to synthesize a single workload and then split it between the two proxies during the simulation?

Upon critical examination, the first approach was found to be unattractive because of the difficulty of controlling the level of overlap between the two workloads. The second approach does not have this drawback, because the level of overlap can be controlled by splitting the requests in the aggregate workload in some determined manner. The questionable issue about the second option is whether the splitting technique would destroy the characteristics inherent in the original workload (e.g., Zipf-ness, heavy-tail property, temporal locality) as it is seen by each of the lower level proxies.

Further work found that the workload characteristics were quite robust across the splitting process, and thus the second option was adopted. Three different strategies are employed to distribute the requests in the aggregate synthetic workload across the two lower level proxies. Fortunately, each method ensures that the effective workload seen by each lower level proxy is representative of a Web proxy workload. These strategies are described below.

Complete Overlap The complete overlap scenario models the situation where the aggregate client workloads seen by each lower level proxy are equally likely to reference any of the documents in the entire document space available from the Web servers. In this scenario, the cache contents at the two lower level proxies are likely to be statistically similar, on average. For example, proxy caches at SaskTel in Saskatoon and at SaskTel in Regina might see very similar request streams from Internet subscribers. To mimic this scenario, the following algorithm is used to dispatch the requests in the aggregate synthetic workload.

```
WHILE there are more requests
    generate a random number x which is IID U(0,1)
    IF (x ≤ 0.5)
         send request to the lower level proxy on the left
    ELSE
         send request to the lower level proxy on the right
    ENDIF
ENDWHILE
```

On average, this approach will result in half of the total requests being sent to each lower level proxy, and a typical document with R total requests in the aggregate workload will have about $R/2$ of its requests directed to each lower level proxy (on average).

No Overlap The no overlap scenario models the situation where the sets of document requests seen by each lower level proxy are disjoint, i.e., the set of files requested by the users of each lower level proxy are entirely different. For example, proxy caches at the University of Saskatchewan and at Toshiba in

Japan might see very different request streams. For this model, the *file id* is used to partition the requests into two distinct sets as follows:

```
WHILE there are more requests
     IF (fileId % 2 == 0)
          send request to the lower level proxy on the left
     ELSE
          send request to the lower level proxy on the right
     ENDIF
ENDWHILE
```

Partial Overlap The partial overlap scenario represents an intermediate situation between the two extremes already discussed (Complete Overlap and No Overlap). That is, there is a partial overlap in the workloads of the two lower level proxies, in terms of the set of documents requested. It signifies some level of commonality in the interests of the clients of the lower level proxies. For example, proxy caches at the University of Saskatchewan and the University of Toronto might see some overlap in content requested, but not complete overlap. To mimic this scenario, the (flexible) algorithm below is used to dispatch requests from the aggregate workload such that there is 50% overlap in the document sets.

```
WHILE there are more requests
     IF (fileId % 4 == 0) /* 25% of docs */
          send request to the lower level proxy on the left
     ELSE IF (fileId % 4 == 1) /* 25% of docs */
          send request to the lower level proxy on the right
     ELSE /* 50% of docs */
          generate a random number x which is IID U(0,1)
          IF (x ≤ 0.5)
               send request to the lower level proxy on the left
```

```
        ELSE
            send request to the lower level proxy on the right
        ENDIF
    ENDIF
ENDWHILE
```

The algorithm can be easily generalized to provide other degrees of overlap, if desired (e.g., 33%, 60%). However, the experiments in this thesis consider only 50% partial overlap.

5.2 Other Experimental Issues

For the experiments in this chapter, the factors employed are cache replacement policy and cache size, just like in the single server simulations. Similarly, the same performance metrics used for the single proxy server simulation, namely hit ratio and byte hit ratio, are employed to evaluate all the caching strategies proposed. The description of these items is the same as in the single server experiments in Chapter 4. In addition, a simulation warm-up of 500,000 requests is used for each proxy in the hierarchy.

5.3 Characteristics of the Aggregate Workload

Table 5.1 presents the input parameters used to synthesize the aggregate workload. As shown in the table, a trace with 10 million requests was synthesized. The analysis of the synthetic workload is presented in Table 5.2. The graphical characteristics of the workload are shown in Figures 5.2(a), (b), and (c).

5.4 Simulation Results

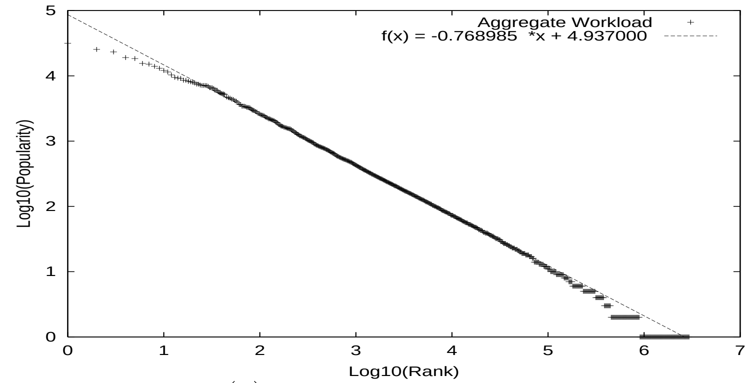
The experiments in this chapter are grouped into three categories. In the first set of experiments, the performance benefits of using different combinations of the three

Table 5.1: Parameter Settings for ProWGen to Synthesize Ten Million Requests

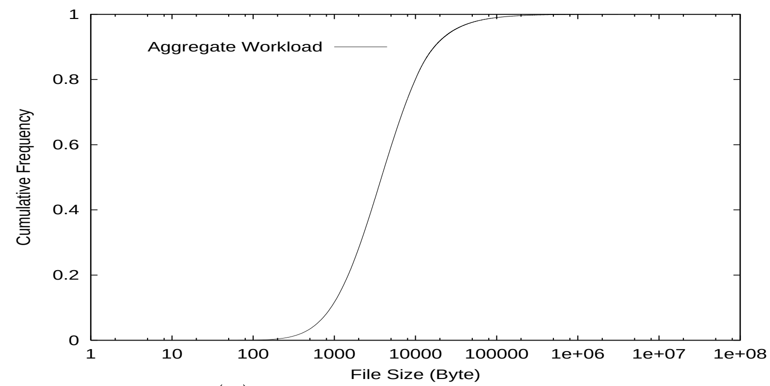
Parameter	Value
Total number of requests	10,000,000
Percentage of unique documents (of total requests)	30%
Percentage of one-timers (of unique documents)	70%
Correlation between file size and popularity	Zero
Zipf Slope	0.75
Tail index	1.3
Beginning of the tail (k) in bytes	10,000
Percentage of documents in the tail	20%
Mean of the lognormal distribution (μ)	7,000
Standard deviation of the lognormal distribution (σ)	11,000
LRU Stack Model for temporal locality	Dynamic
LRU Stack Size for temporal locality	1,000

Table 5.2: Characteristics of the Aggregate Synthetic Workload Used for the Hierarchical Caching Simulation

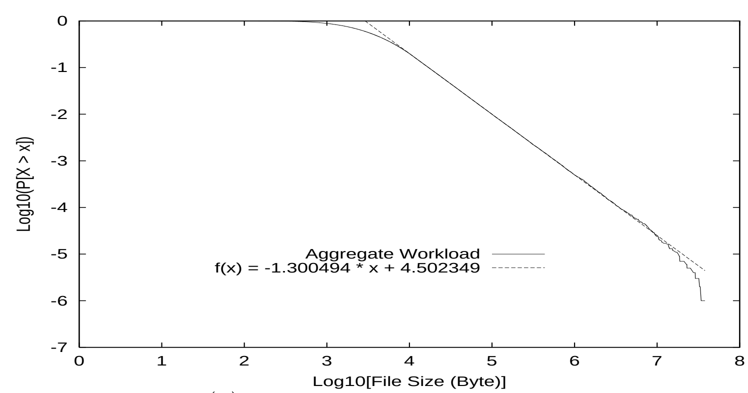
Item	Value
Total number of requests	9,749,703
Number of unique documents	3,000,000
Percentage of unique documents (of total requests)	30%
Number of one-timers	2,099,045
Percentage of one-timers (of unique documents)	69%
Total bytes of unique documents (GB)	30
Total content bytes of all requests (GB)	89
Smallest file size (bytes)	9
Largest file size (bytes)	51,600,679
Mean file size (bytes)	10,850
Median file size (bytes)	3,815
Correlation between file size and popularity	-0.004982
Zipf Slope	-0.768985
R^2	0.997980
Tail index	-1.300494
R^2	0.999887



(a) Popularity vs. Rank



(b) CDF plot for file sizes



(c) LLCD plot for file sizes

Figure 5.2: Popularity, CDF and LLCD Plots for the Aggregate Workload Used in the Hierarchical Caching Simulation.

replacement policies at the two levels of the hierarchy are examined; the results are presented in Section 5.4.1. In the following experiments, the effects of keeping shared documents at the upper level of the hierarchy are examined; the results are presented in Section 5.4.2. The last set of experiments looks at the performance implications of keeping disjoint documents at different levels of the hierarchy, e.g., small files at the lower level and large files at the upper level, or vice versa; these results are presented in Section 5.4.3.

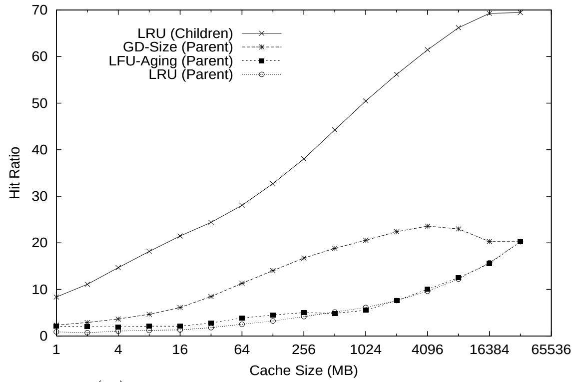
5.4.1 Experiment 1: Different Replacement Policies at Different Levels of the Hierarchy

The main objective of the experiments in this section is to see if using different replacement policies at different levels of the hierarchy can improve caching performance. Using the two-level hierarchical proxy model in Figure 5.1, and considering the *complete, partial,* and *no overlap* situations, with different combinations of the three replacement policies, the simulation results are shown in Figures 5.3, 5.4, and 5.5.

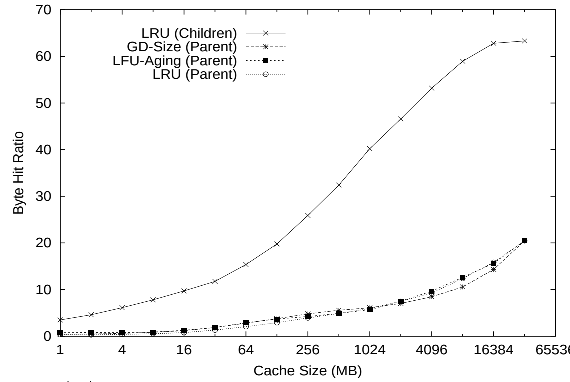
The performance results for the complete overlap scenario are shown in Figure 5.3. The results for the partial overlap scenario are shown in Figure 5.4, and the results for the no overlap scenario are shown in Figure 5.5. On each graph, the replacement policy at the lower level proxies is fixed, while that at the upper level proxy is varied. Also, for each simulation, the children and the parent proxies are configured with the same cache sizes.

In all simulation results, the two lower level proxies provide practically identical hit ratios and byte hit ratios, hence only one result is plotted against the results for the parent cache on each graph. The reason for the same hit ratios and byte hit ratios at the two lower level proxies is the use of the same replacement policy at each lower level proxy, and the randomized (uniform) distribution of the requests in the aggregate workload by the dispatching mechanisms used.

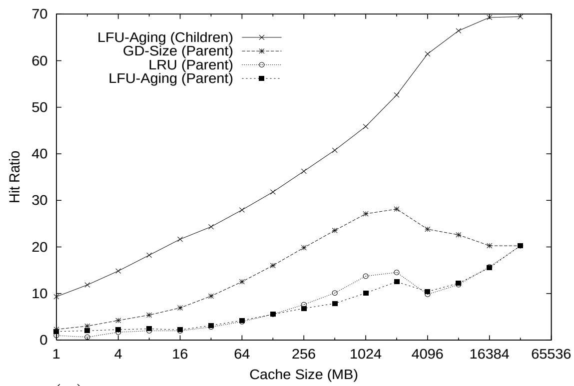
Concentrating on Figure 5.3, which shows the performance of the hierarchy for



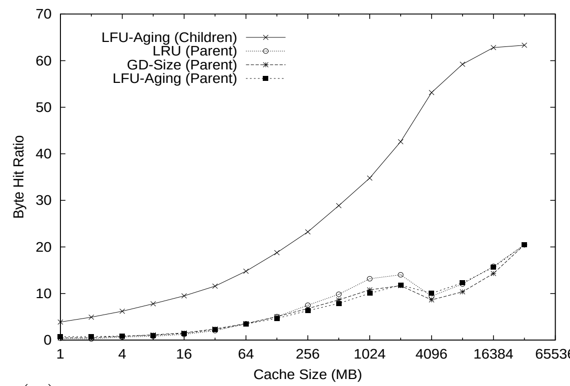
(a) Hit Ratio (Children: LRU)



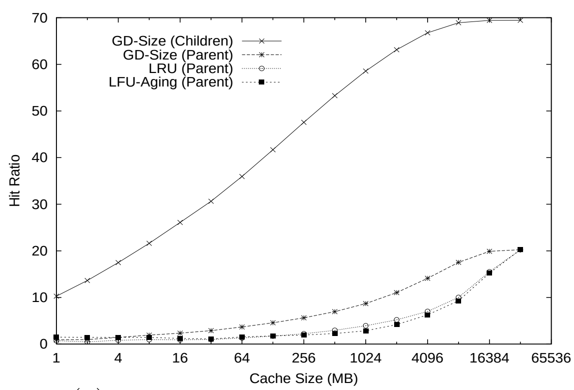
(b) Byte Hit Ratio (Children: LRU)



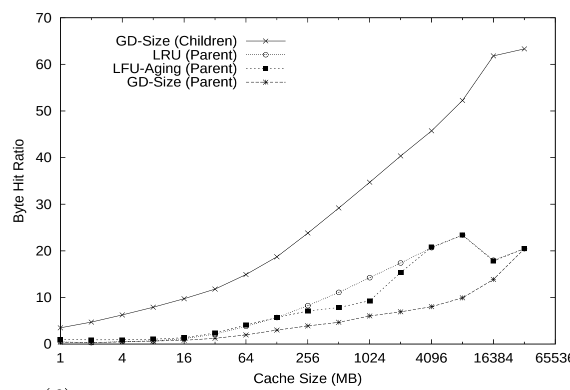
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

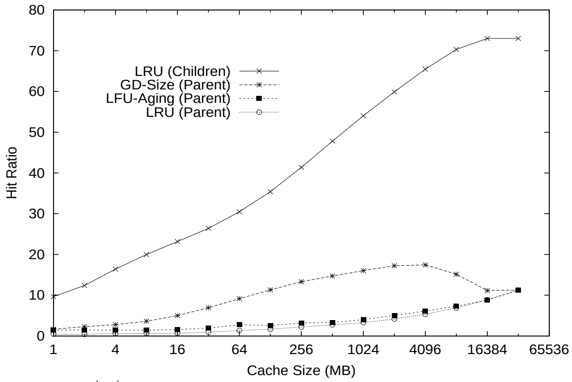


(e) Hit Ratio (Children: GD-Size)

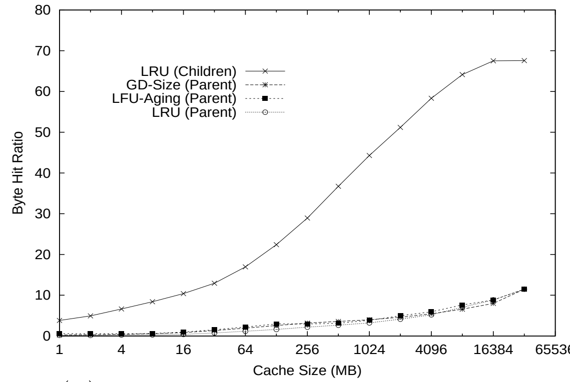


(f) Byte Hit Ratio (Children: GD-Size)

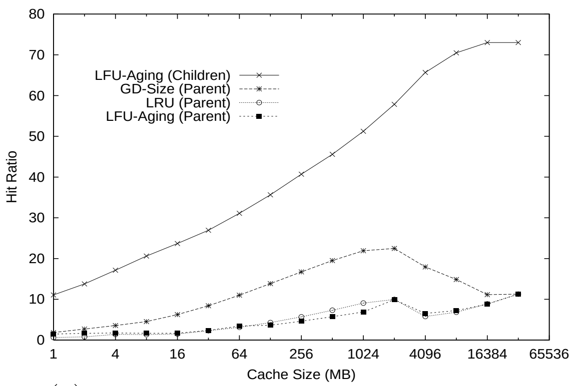
Figure 5.3: Performance Results for Different Combinations of Replacement Policies in a Two-Level Hierarchical Proxy Configuration, with COMPLETE Overlap in the Workloads of the Lower Level Proxies.



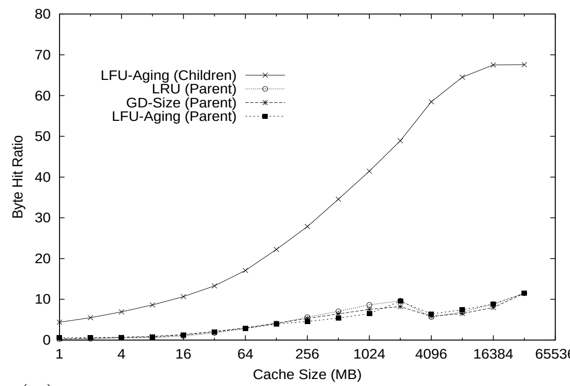
(a) Hit Ratio (Children: LRU)



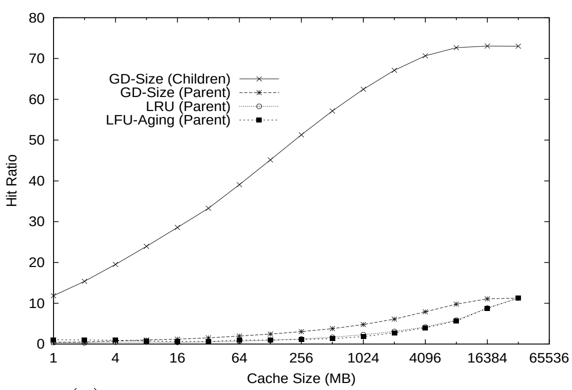
(b) Byte Hit Ratio (Children: LRU)



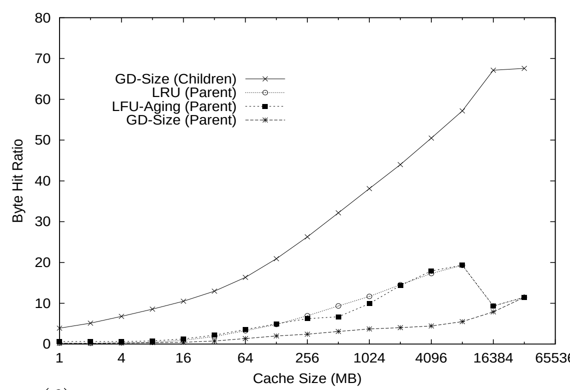
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

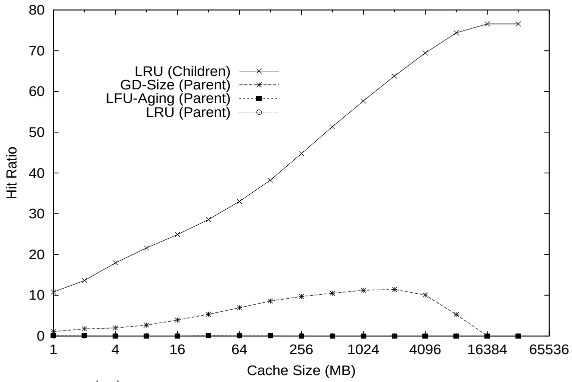


(e) Hit Ratio (Children: GD-Size)

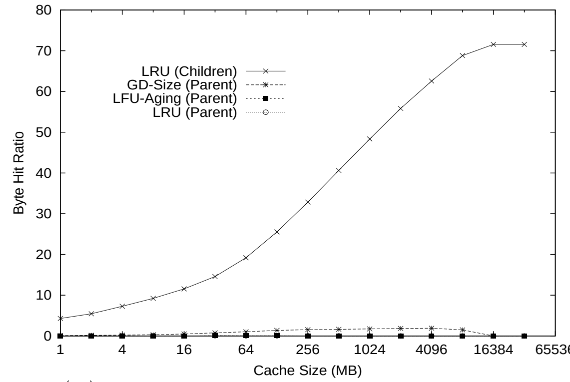


(f) Byte Hit Ratio (Children: GD-Size)

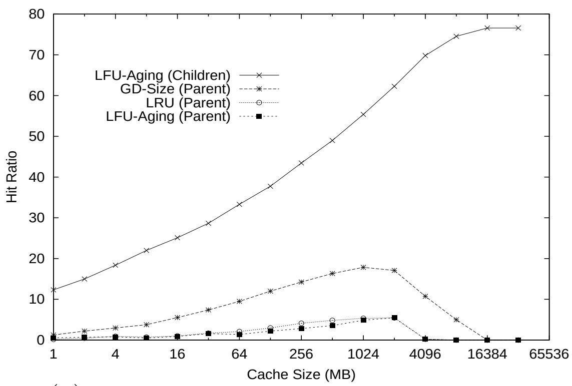
Figure 5.4: Performance Results for Different Combinations of Replacement Policies in a Two-Level Hierarchical Proxy Configuration, with PARTIAL Overlap in the Workloads of the Lower Level Proxies.



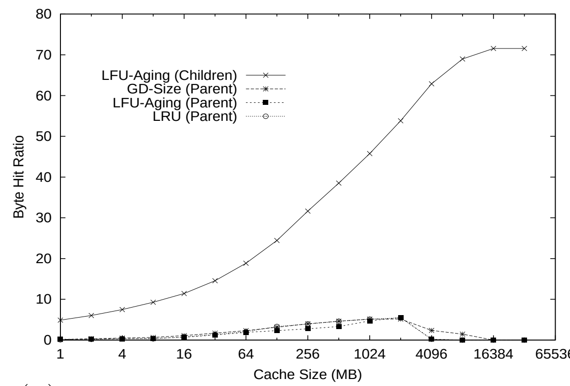
(a) Hit Ratio (Children: LRU)



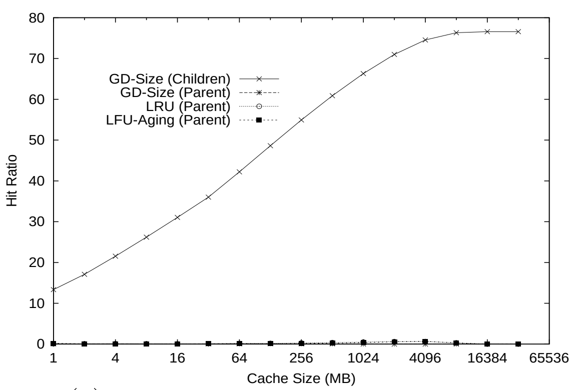
(b) Byte Hit Ratio (Children: LRU)



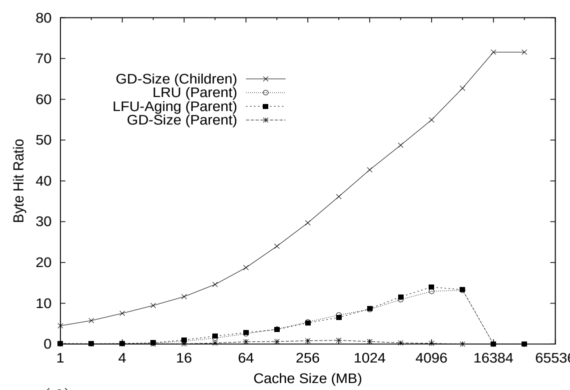
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)



(e) Hit Ratio (Children: GD-Size)



(f) Byte Hit Ratio (Children: GD-Size)

Figure 5.5: Performance Results for Different Combinations of Replacement Policies in a Two-Level Hierarchical Proxy Configuration, with NO Overlap in the Workloads of the Lower Level Proxies.

the *complete overlap* scenario, Figures 5.3(a) and (b) show the hit ratios and the byte hit ratios when the children use the LRU policy, and the parent uses either LRU, LFU-Aging, or GD-Size. Similarly, Figures 5.3(c) and (d) show the performance results when the replacement policy at the children is LFU-Aging, while Figures 5.3(e) and (f) show the results when the children use the GD-Size policy.

In general, the child proxy caches have much higher hit ratios than the parent proxy. This observation is not surprising, given that the parent proxy only sees the requests that miss at the lower level caches (i.e., the request stream is filtered by the lower level proxies).

An observation evident from Figures 5.3(a) and (c) is that using GD-Size at the upper level provides much better hit ratios than using either LRU or LFU-Aging at the upper level. In Figures 5.3(e) and (f), which use GD-Size at the lower level, GD-Size at the upper level still outperforms other combinations shown on these graphs, in terms of hit ratios (though the advantage is less pronounced), but in terms of byte hit ratios, the GD-Size policy at the parent gives lower byte hit ratios than LRU and LFU-Aging. Further analysis of which combination is "better" is deferred for later discussion in Section 5.5.

The performance results for the *partial overlap* scenario are shown in Figure 5.4. These results follow the same trend as in the complete overlap case, except for the noticeable drop in the hit ratios of the parent cache, though with slight improvement in performance of the children. The reduced overlap in the workloads of the children is responsible for the decrease in performance of the parent. In other words, in the partial overlap scenario, the probability that a file requested by one child can also be requested later by the other child is about 50% as opposed to about 100% in the complete overlap (ignoring one-timers). The reason for the slight improvement in the performance of the children can be attributed to the remaining unshared 50% of the unique files in the aggregate workload, which is partitioned into two distinct sets, ensuring that a distinct 25% of the files consistently goes to a particular child thus preserving the concentration of references and temporal locality of such files. In other words, each lower level cache sees *all* the references to a distinct 25% of the document set, and shares the references to the remaining 50% of the document set. Overall, the portion of the aggregate document set seen by each lower level proxy is around 75%.

Lastly, in the no overlap scenario (Figure 5.5), the same observations as in the complete and the partial overlap cases can be drawn, thus the same explanations hold. With no shared files in the workloads of the children, it is rare for the parent to satisfy a request from its cache (it would do so only if the child had a capacity miss while the parent did not). This is also the reason why the performance of the parent starts to drop as the cache size at the children approaches infinite size. On the other hand, the performance of the children is also better than in the previous case of partial overlap because the portion of the aggregate document set seen by each lower level proxy in this case is around 50%. That is, all references to a unique 50% of the document set consistently go to a particular child, thus preserving concentration of references and temporal locality for such files.

In summary, the general observation is consistent for the three typical scenarios considered in this experiment. The general observation is that if the lower level proxies use the LRU or the LFU-Aging policy, the GD-Size policy is the best policy at the upper level. It often provides double the hit ratios of LRU or LFU-Aging at the upper level, without any significant penalty in byte hit ratios.

5.4.2 Experiment 2: Shared Files at the Upper Level of the Hierarchy

One drawback of a caching hierarchy is the replication of cache contents that can occur. That is, a single Web document may be present in multiple caches, and at multiple levels of the hierarchy. As a result, the aggregate cache space is not utilized as effectively as possible.

Intuitively, in a caching hierarchy, it makes sense to keep files that are shared by two or more lower level proxies in the upper level of the hierarchy to avoid caching duplicate copies of such files in the hierarchy. In such a case, the effectiveness of this strategy will depend on the level of sharing in the workloads of the lower level proxies.

The experiments in this section examine the effect of keeping shared files at the upper level of the hierarchy. For simplicity, only the LRU policy is considered at each level of the hierarchy, assuming that previous results about combining different policies still hold with this caching strategy.

To further reduce complexity, it is assumed that the upper level proxy has perfect knowledge about shared files, i.e., files that are requested by the two lower level proxies. This information is obtained by first running the simulation experiment to record, in a log file, all requests sent to each lower level proxy. Thereafter, the files are then processed to extract information about common files. Then the simulation proceeds as usual to obtain hit ratios and byte hit ratios.

During the simulation, each request sent to the lower level proxy is first checked in the set of shared files. If the requested file is found in the list of shared files, a miss is recorded for the child proxy and the request is sent to the upper level proxy (a request for a shared file cannot be satisfied in the lower level proxies because only the upper level proxy is allowed to cache shared files). At the upper level proxy, if the request cannot be satisfied from the cache, it is obtained from the originating server and a copy is stored in the cache. Such files, when finally returned to the client, are not cached in the lower level proxy. On the other hand, if a requested file cannot be satisfied from a lower level proxy's cache, but the file is not among the shared files, the requesting child proxy can store a copy in its cache when the file is finally received from the parent. In this case, the parent is not allowed to cache the file.

In practice, implementing this algorithm will require the parent to keep some information about the files each child currently has in its cache. Furthermore, it requires that each proxy in the hierarchy have a local copy of the list of shared files. Thus if a file is to be added to the shared list by the parent because the file has been cached by one of the children, and it is now being requested by another child, the child currently caching a copy would have to be notified to expunge the file from its cache and update its local copy of the shared files. This experiment uses the perfect knowledge approach to eliminate this complexity, and to examine the maximum benefit derivable from this caching technique, if implemented.

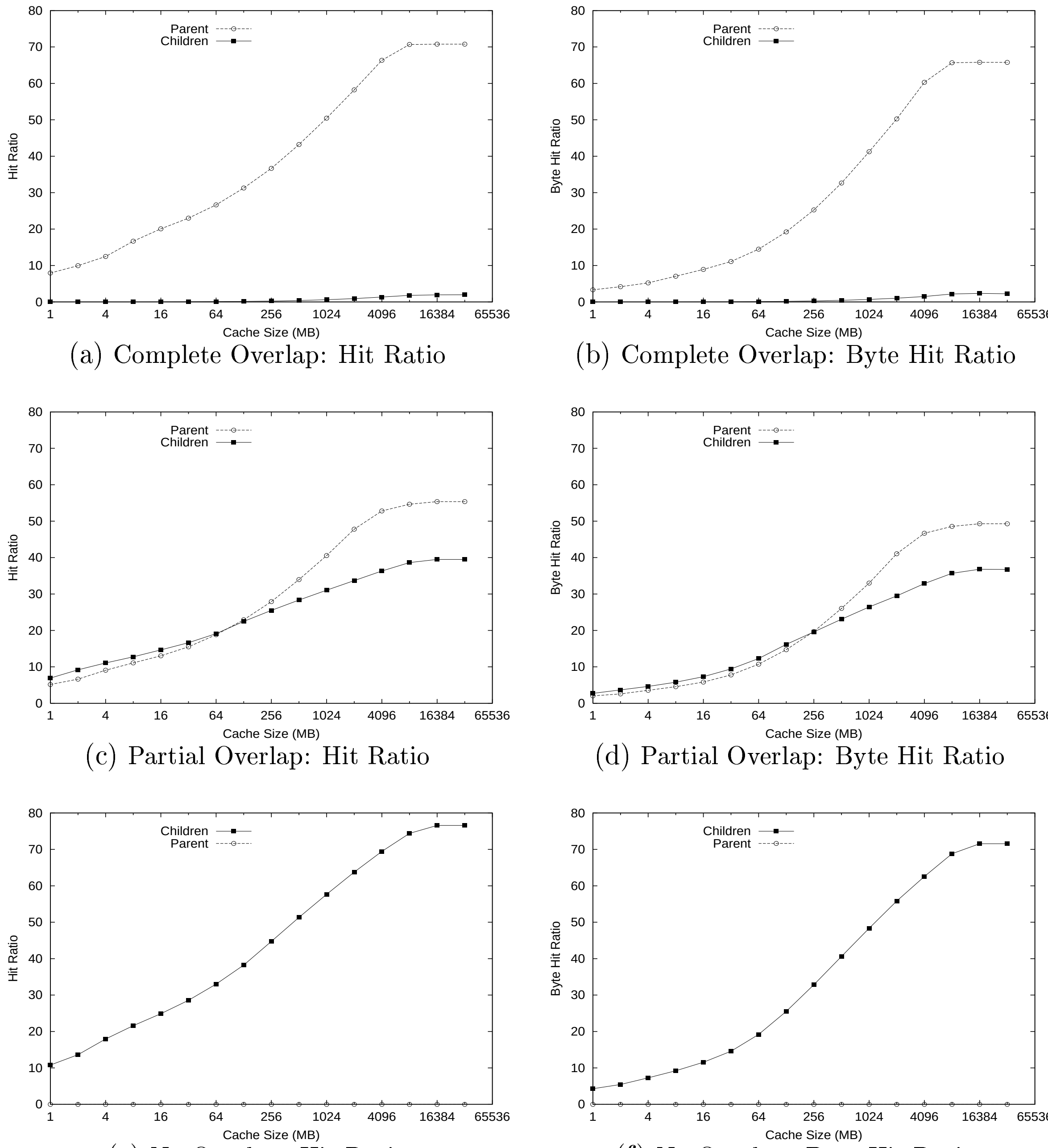
The performance results for the three overlap scenarios are presented in Figure 5.6. The graphs in Figures 5.6(a) and (b) show the hit ratios and byte hit ratios for the complete overlap workload, Figures 5.6(c) and (d) are the results for the partial overlap case, while Figures 5.6(e) and (f) present the results for the no overlap scenario.

The results for the complete overlap case in Figures 5.6(a) and (b) are not surprising. Since almost all (non-one-timer) files are shared files in this workload, the hit ratios and byte hit ratios at the parent cache are high, but this is at the expense of the lower level proxies where the hit ratios are less than 3% for the whole range of cache sizes considered. The reason for this behaviour is the high percentage of shared files in the workloads of the lower level proxies. Since most of these files are cached at the upper level, only a few files remain for caching at the lower level, thus limiting the hit ratios achieved by the children.

At the other extreme are the results in Figures 5.6(e) and (f) where there is no overlap in the workloads of the lower level proxies. These results are consistent with what is expected. In this case, the parent cannot cache any file, as no files are shared. Thus the hit ratios and the byte hit ratios at the parent remain constant at zero. In other words, the proxies operate as if the parent is non-existent.

Figures 5.6(c) and (d) show the results for a 50% overlap in the workload of the lower level proxies. These results represent the intermediate case between complete overlap and no overlap. The graphs show that the hit ratios and the byte hit ratios for the parent are lower (compared to the complete overlap results in Figures 5.6(a) and (b)), while the hit ratios and the byte hit ratios for the children have improved. Both levels of the caching hierarchy are about equally effective for this workload scenario.

The overall caching performance with the "shared documents at parent cache only" approach is arguably better than the unrestricted caching approach. For



(a) Complete Overlap: Hit Ratio

(b) Complete Overlap: Byte Hit Ratio

(c) Partial Overlap: Hit Ratio

(d) Partial Overlap: Byte Hit Ratio

(e) No Overlap: Hit Ratio

(f) No Overlap: Byte Hit Ratio

Figure 5.6: Performance Results for Keeping Shared Files at the Upper Level in a Two-Level Caching Hierarchy, with the LRU Policy at Both Levels.

example, the results in Figures 5.6(c) and (d) can be compared to Figures 5.4(a) and (b) (using the lines for LRU at both levels from these graphs). Summing the aggregate hit ratios and byte hit ratios from Figures 5.4(a) and (b), the combined value is about 85% for the largest cache size considered, as opposed to 95% in Figures 5.6(c) and (d). However, such an estimation can be misleading, since in the former, most of the requests are satisfied at the children level, but in the latter, the reverse is the case. The choice of which policy to use depends on the extra network cost and latency incurred by forwarding a request to the upper level proxy. This issue is discussed later in the chapter.

In summary, the general observation from the above results show that as the level of sharing between the workloads of the lower level proxies decreases, the performance of the parent decreases, while the performance of the children increases. However, the results show that with a medium level of sharing, this caching strategy may provide better performance.

5.4.3 Experiment 3: Disjoint Files at Different Levels of the Hierarchy Based on Size Thresholds

The experiments carried out in this section are designed to see whether keeping small files at the lower level of the hierarchy and large files at the upper level, or vice versa, can improve the performance of a caching hierarchy.

In order to impose a size restriction, first a reasonable threshold file size must be chosen. To understand the impact of different threshold sizes, three values are considered: a threshold of 5,000 bytes, 10,000 bytes, and 100,000 bytes. For each threshold size t, the experiments consider the effect of keeping files smaller than t at the lower level proxies and the remaining files at the upper level proxy, and vice versa.

Since this caching strategy is size-based, it is necessary to examine the effect of combining it with different caching algorithms at different levels of the hierarchy. Therefore, the experiments examine the combination of this caching strategy with different combinations of the three cache replacement policies. The three different workload overlap scenarios are also considered.

The trace-driven simulation experiments use the two-level hierarchical model in Figure 5.1. During the simulation, each proxy respects the size threshold condition while making a decision on whether or not to store a file in its cache. For each new file requested by a particular proxy, the size is checked to ensure that caching it does not violate the restriction imposed. If the size of the file is outside the range of sizes permitted in its cache, the file is simply discarded after returning a copy to the requesting client (or proxy) below it, without keeping a copy, even if there is space in the cache. The results of the simulation experiments are presented in the subsections that follow.

5.4.3.1 Results for Threshold Size of 5,000 Bytes

The results obtained using a threshold size of 5,000 bytes[1] are shown in Figures 5.7 - 5.12. Figures 5.7, 5.8, and 5.9 are the results obtained by keeping files smaller than 5,000 bytes at the lower level of the hierarchy and files larger than that at the upper level, for complete, partial, and no overlap, respectively. The performance results for reversing this policy (i.e., keeping larger files at the lower level and smaller files at the upper level) are shown in Figures 5.10, 5.11, and 5.12, also for the complete, partial, and no overlap situations, respectively.

Starting with Figure 5.7, for the complete overlap workload, each graph shows the results for using a fixed replacement policy at the lower level and using all three replacement policies at the upper level. For instance, Figures 5.7(a) and (b) show the hit ratios and byte hit ratios for using the LRU policy at the lower level and the three different replacement policies at the upper level. Similarly, Figures 5.7(c) and (d) show the results for the LFU-Aging policy at the lower level and Figures 5.7(e) and (f) show the results for using GD-Size at the lower level.

The general observation from Figure 5.7 is that, while the child proxies always

[1] 60% of the references are for files below this threshold size, based on the CDF graph in Figure 5.2(b) on page 96.

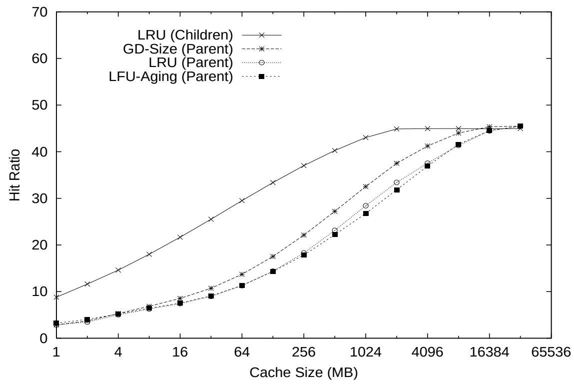
achieve higher hit ratios than the parent proxy, the parent always achieves higher byte hit ratios. The result for the parent proxy is particularly interesting, in that it is able to achieve significant hit ratios as well as byte hit ratios. The pattern of behaviour noted in the hit ratios (Figures 5.7(a), (c), and (e)) can be attributed to the high proportion of small files in the workload. The penalty for not keeping the large files (files $\geq$ 5,000 bytes) in the lower level cache is the lower byte hit ratios, as observed in Figures 5.7(b), (d), and (f). The significant hit ratios and byte hit ratios achieved by the parent indicate that many references occur to files stored in its cache, and these files are responsible for a significant fraction of the total volume of data transferred as well.

Another interesting observation from the graphs is that using the GD-Size policy at the upper level with any of the three replacement policies at the lower level can provide slightly better performance in terms of hit ratios, although with slightly lower byte hit ratios, compared to LRU or LFU-Aging, at large cache sizes.

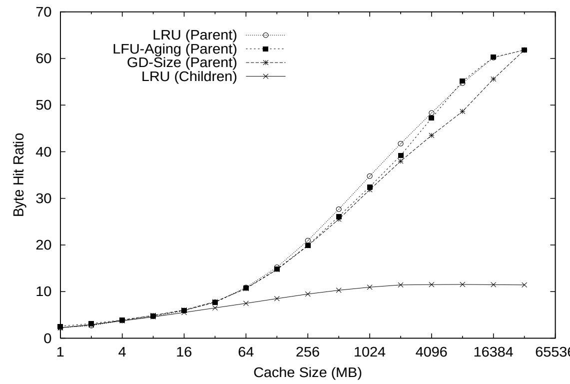
The flattening behaviour of the hit ratio and byte hit ratio graphs for the children beyond a cache size of 2 GB indicates a form of "cache ineffectiveness" beyond this point. The explanation is that this cache size represents the "infinite cache size" for the lower level proxies. That is, all the requested files smaller than 5,000 bytes can conveniently be accommodated in a 2 GB cache without having to replace any of them. Increasing the cache size beyond 2 GB increases the hit ratios and the byte hit ratios for the parent, but unfortunately, it has no further impact on the performance of the children.

The performance results for the partial overlap scenario shown in Figure 5.8 follow the same pattern as in the complete overlap case. The only noticeable difference is the slight increase in hit ratios for the parent and the children, and a wider margin between them than in Figure 5.7. Surprisingly, the byte hit ratios do not show any significant difference compared to the complete overlap scenario.

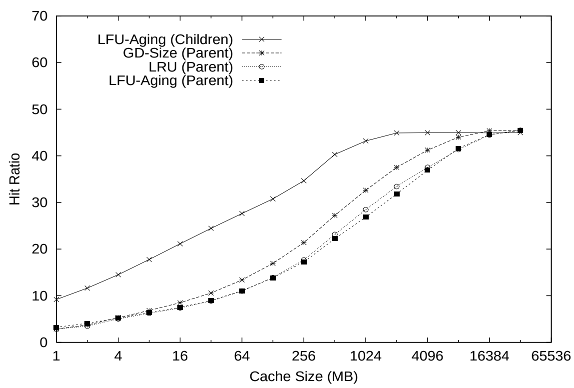
The situation for the no overlap scenario in Figure 5.9 is no different: an increase (up to 5%) in hit ratios compared to Figure 5.7, and a wider difference between the hit ratios of the parent and the children, but no significant difference in the byte



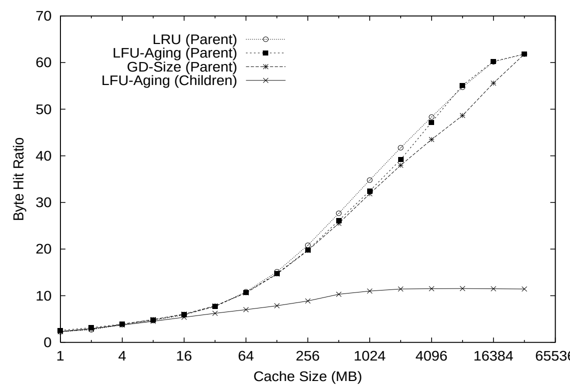
(a) Hit Ratio (Children: LRU)



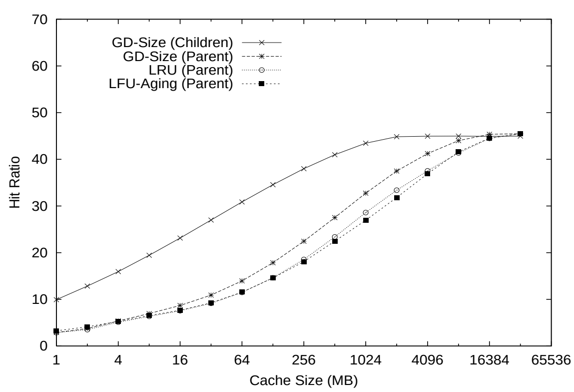
(b) Byte Hit Ratio (Children: LRU)



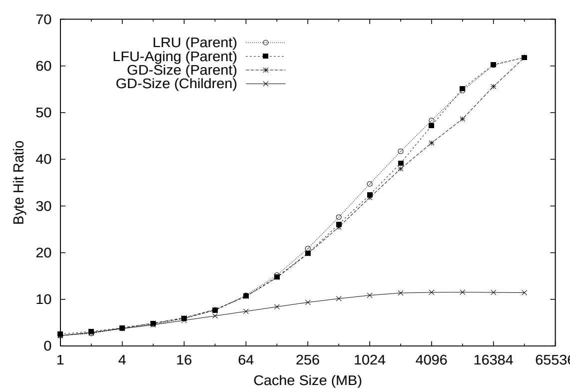
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

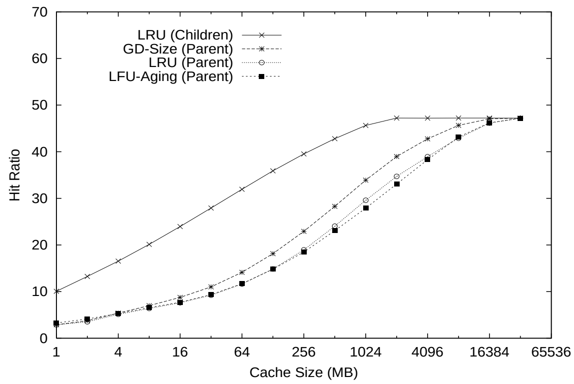


(e) Hit Ratio (Children: GD-Size)

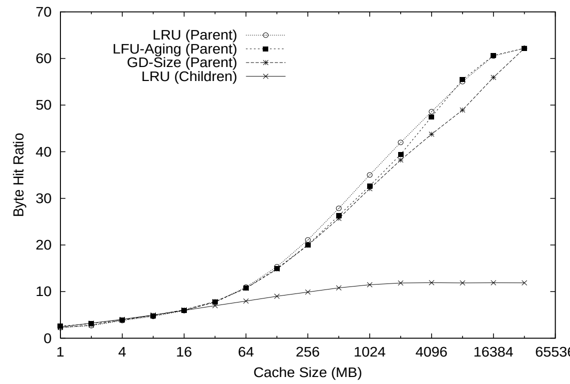


(f) Byte Hit Ratio (Children: GD-Size)

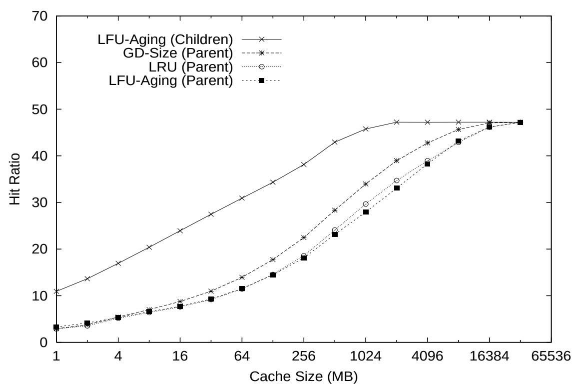
Figure 5.7: COMPLETE Overlap, Size Threshold = 5,000 Bytes: Small Files at the Lower Level; Large Files at the Upper Level.



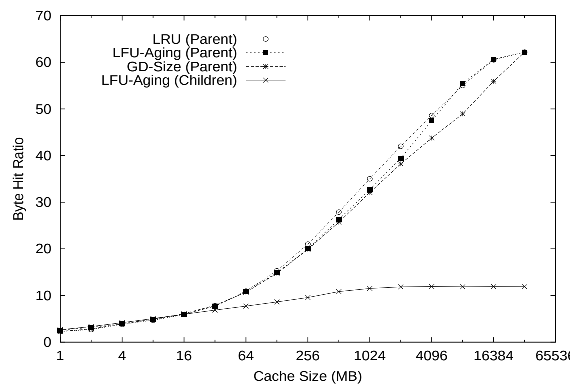
(a) Hit Ratio (Children: LRU)



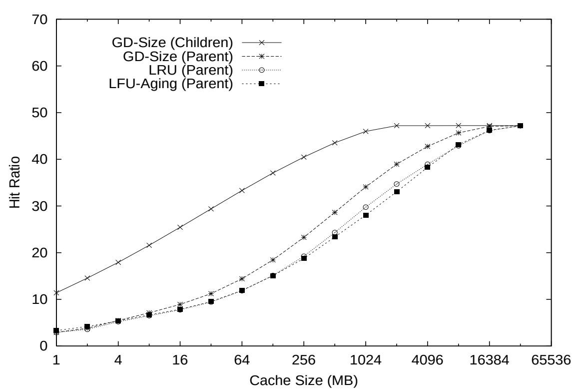
(b) Byte Hit Ratio (Children: LRU)



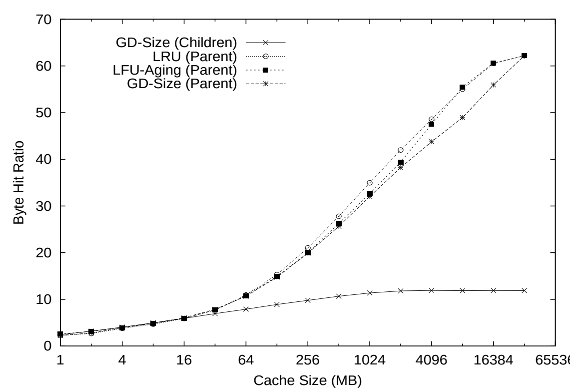
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

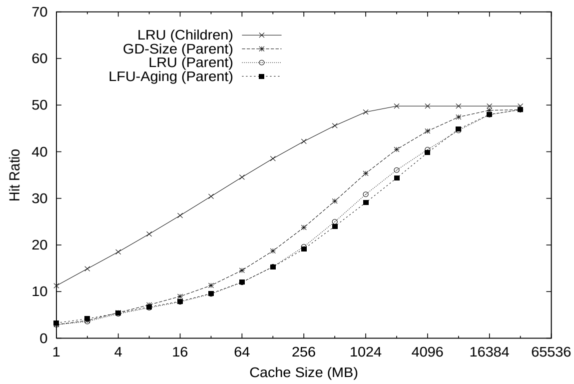


(e) Hit Ratio (Children: GD-Size)

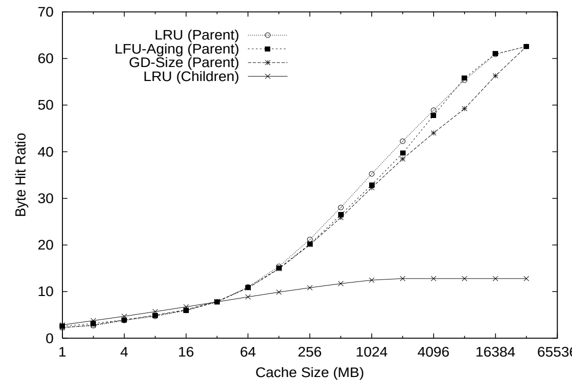


(f) Byte Hit Ratio (Children: GD-Size)

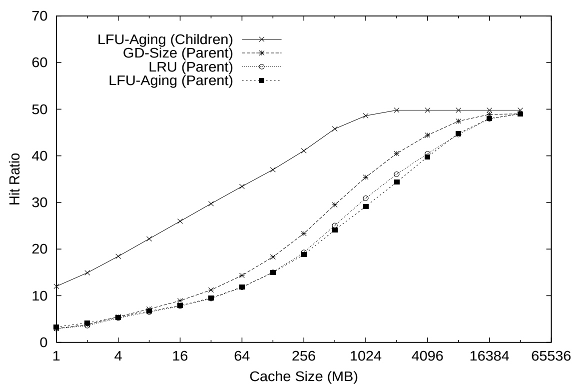
Figure 5.8: PARTIAL Overlap, Size Threshold = 5,000 Bytes: Small Files at the Lower Level; Large Files at the Upper Level.



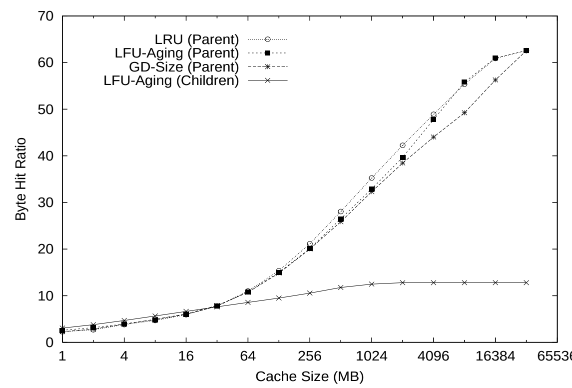
(a) Hit Ratio (Children: LRU)



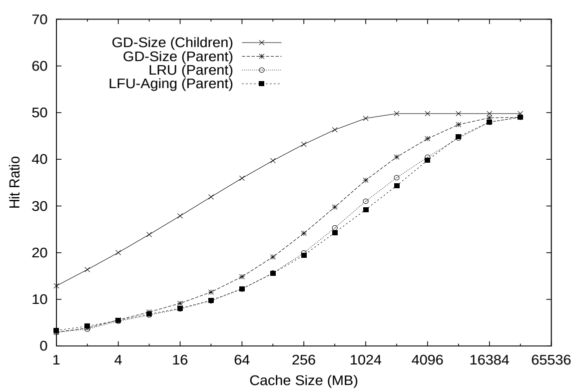
(b) Byte Hit Ratio (Children: LRU)



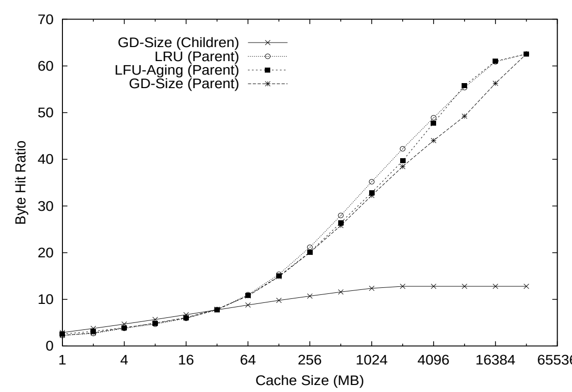
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)



(e) Hit Ratio (Children: GD-Size)



(f) Byte Hit Ratio (Children: GD-Size)

Figure 5.9: NO Overlap, Size Threshold = 5,000 Bytes: Small Files at the Lower Level; Large Files at the Upper Level.

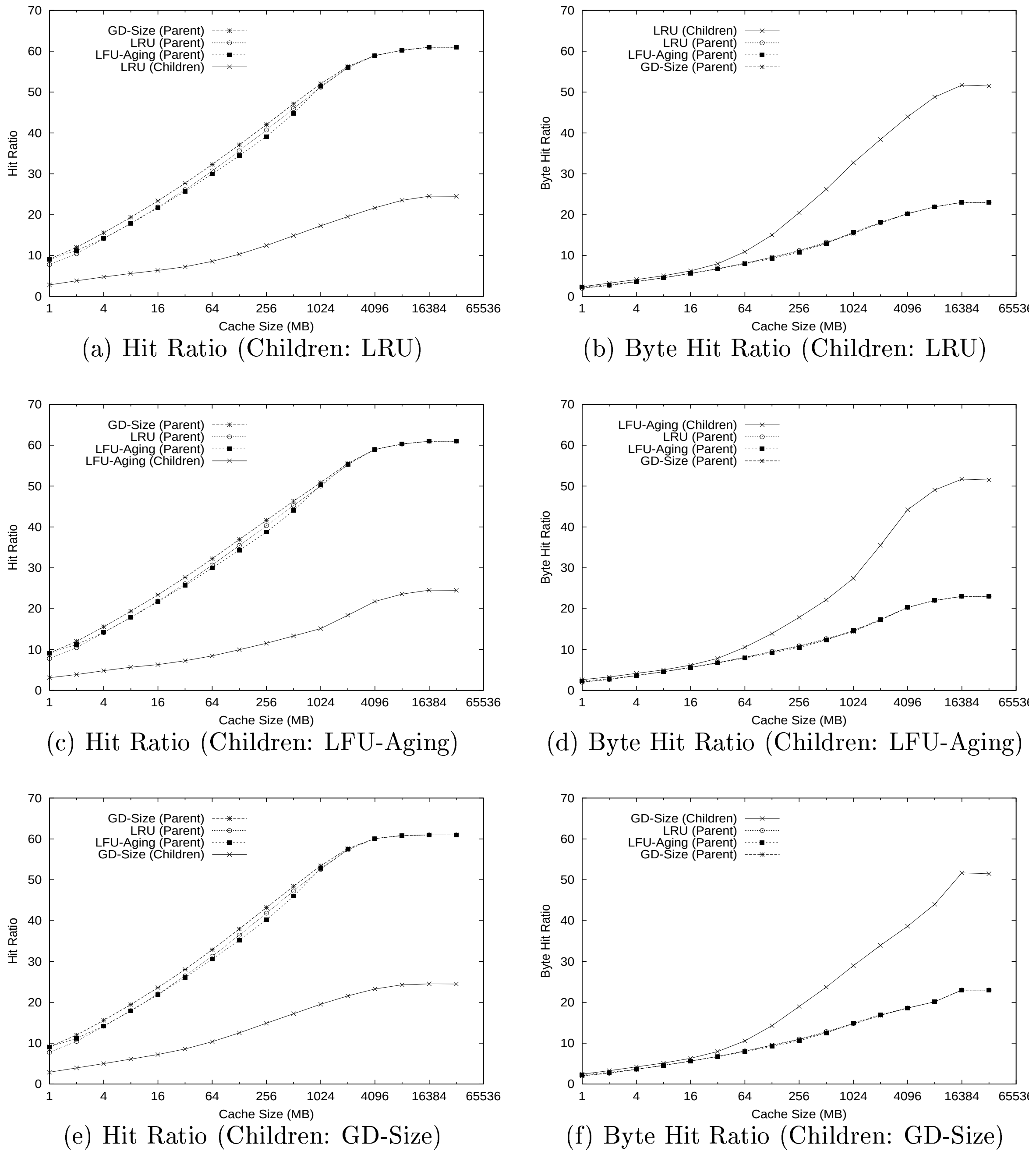
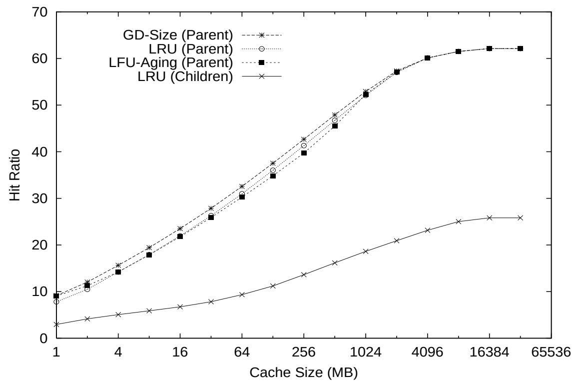
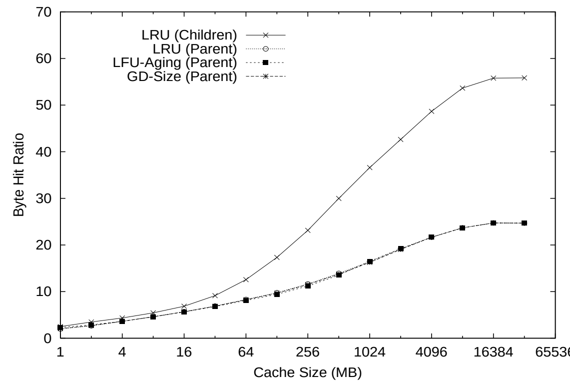


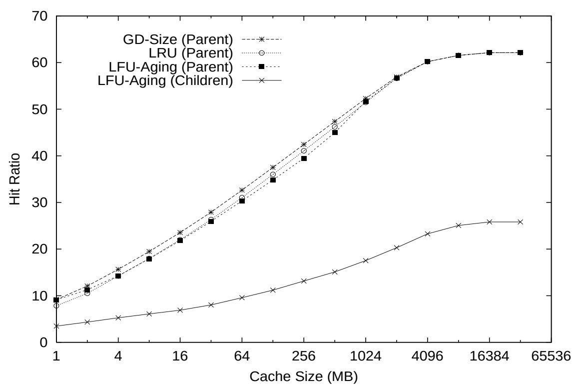
Figure 5.10: COMPLETE Overlap, Size Threshold = 5,000 Bytes: Large Files at the Lower Level; Small Files at the Upper Level.



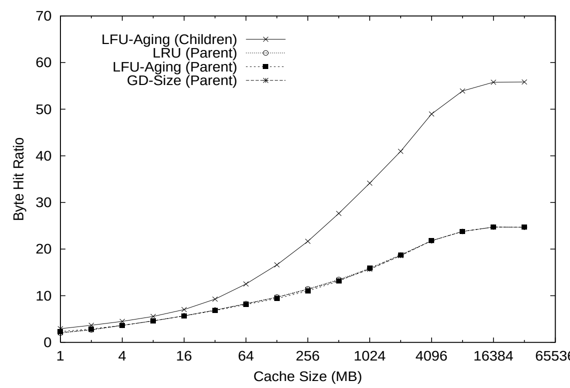
(a) Hit Ratio (Children: LRU)



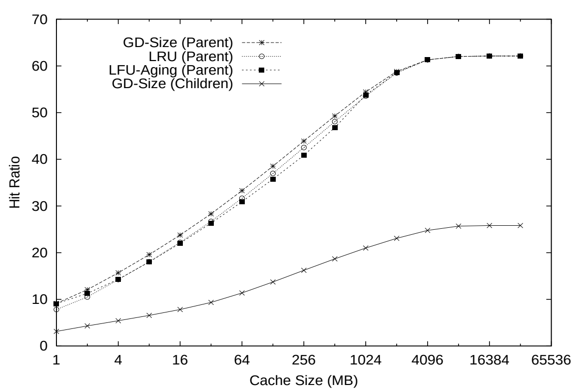
(b) Byte Hit Ratio (Children: LRU)



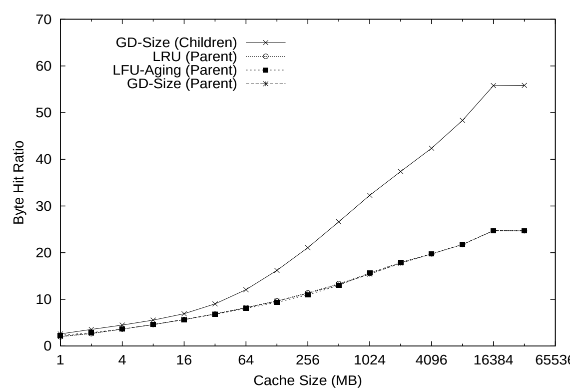
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

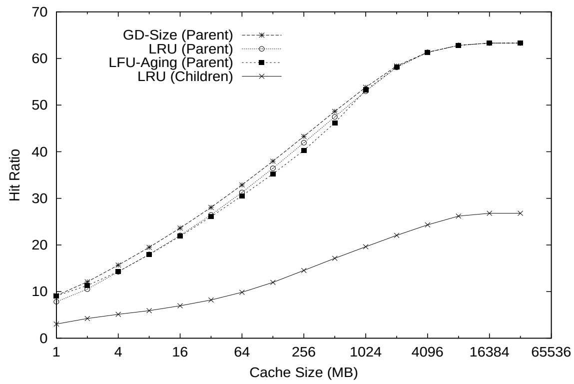


(e) Hit Ratio (Children: GD-Size)

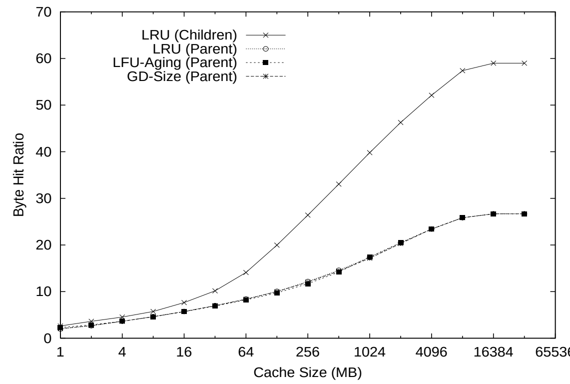


(f) Byte Hit Ratio (Children: GD-Size)

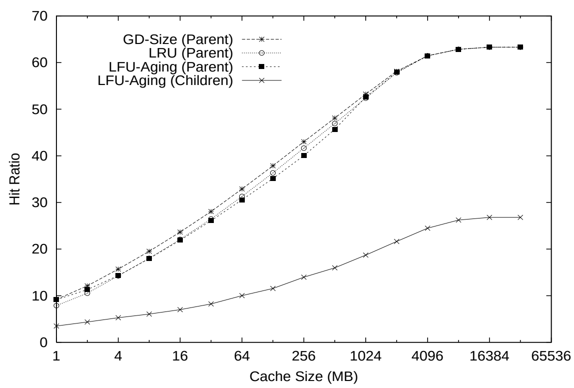
Figure 5.11: PARTIAL Overlap, Size Threshold = 5,000 Bytes: Large Files at the Lower Level; Small Files at the Upper Level.



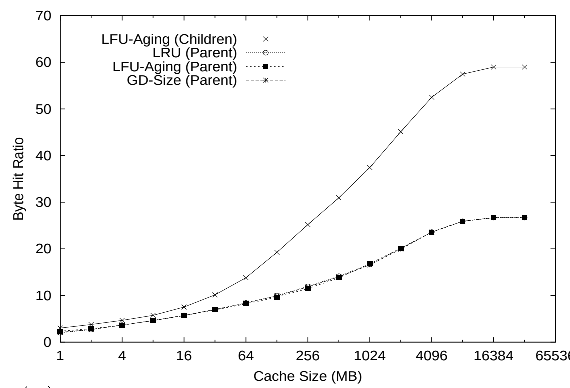
(a) Hit Ratio (Children: LRU)



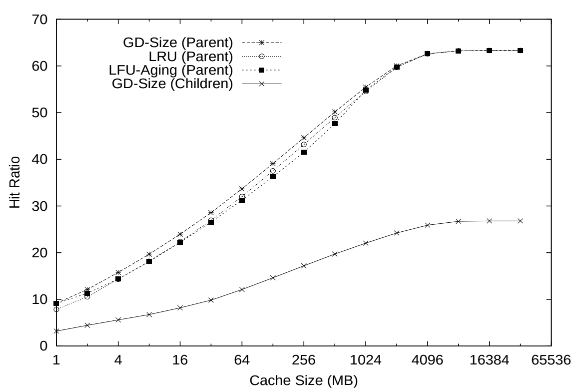
(b) Byte Hit Ratio (Children: LRU)



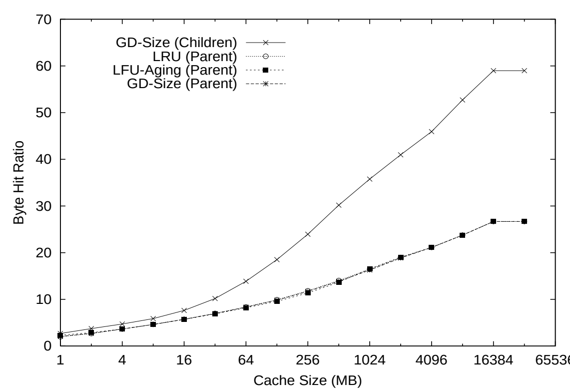
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)



(e) Hit Ratio (Children: GD-Size)



(f) Byte Hit Ratio (Children: GD-Size)

Figure 5.12: NO Overlap, Size Threshold = 5,000 Bytes: Large Files at the Lower Level; Small Files at the Upper Level.

hit ratios. The reason for the increase in hit ratios of the lower level proxies as the level of overlap changes from complete to partial to no overlap is due to the "proxy affinity" of references to some files. In other words, references to certain (small) files consistently go to a particular child proxy.

The effect of reversing the size threshold policy (i.e., keeping larger files at the lower level proxies and smaller files at the upper level proxy) can be observed in Figures 5.10, 5.11, and 5.12, again for complete, partial, and no overlap scenarios, respectively.

In Figures 5.10(a), (c), and (e), which represent the hit ratio graphs for complete overlap, the parent proxy seems to show consistently better performance than the children. This is not surprising: the children are not allowed to cache files smaller than 5,000 bytes; this implies that all files in this category end up in the parent's cache, and since they are responsible for a significant fraction of the requests, they tend to have a dramatic impact on the observed hit ratios at the parent. While the observed hit ratios at the children are low, the byte hit ratios (Figures 5.10(b), (d), and (f)) are better than for the parent cache. Another interesting observation in the byte hit ratios of the parent is that there is no difference between the performance of the three cache replacement policies. This is due mainly to the parent's discrimination against large files. In other words, the performance impact of the size-based threshold scheme between levels of the proxy hierarchy is so dominant that the precise replacement policy used at the upper level is irrelevant. LRU, LFU-Aging, and GD-Size are equally effective at the parent level.

In the partial overlap results in Figure 5.11, and the no overlap case in Figure 5.12, the same observations hold, with hardly any noticeable difference, in both hit ratios and byte hit ratios.

5.4.3.2 Results for Threshold Size of 10,000 Bytes

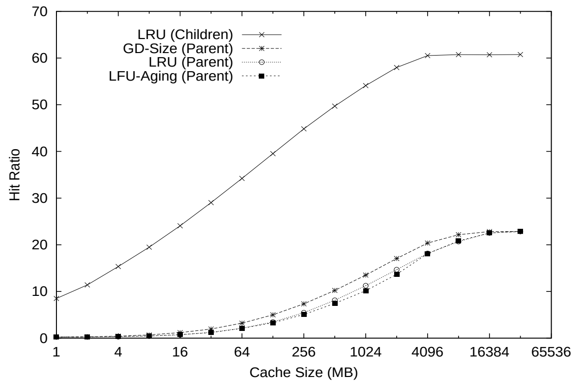
The performance results for a threshold size of 10,000 bytes[2] are shown in Figures 5.13 - 5.18. Figures 5.13, 5.14, and 5.15 show the results of keeping files smaller than 10,000 bytes at the lower level of the hierarchy and larger files at the upper level, for complete, partial, and no overlap scenarios, respectively. The performance results for reversing this policy (i.e., keeping files larger than 10,000 bytes at the lower level, and smaller files at the upper level) are shown in Figures 5.16, 5.17, and 5.18, also for the complete, partial, and no overlap situations, respectively.

By contrasting Figures 5.13(a) and (b) with Figures 5.7(a) and (b), a dramatic increase in the performance of the lower level proxies is noticed, up to 15%, in both hit ratios and byte hit ratios, when the larger threshold size is used. The performance of the parent dropped by more than 20% in hit ratios and 5% in byte hit ratios. As the threshold size increases in this case, the hit ratios drop significantly for the parent, because the percentage of files that can be cached at the lower level of the hierarchy increases, and decreases for the upper level, thus increasing the performance of the children while reducing that of the parent. However, the performance implication on the byte hit ratios for the parent is not as much as on the hit ratios, because the small set of the large files being cached at the upper level is responsible for a significant fraction of the total bytes.

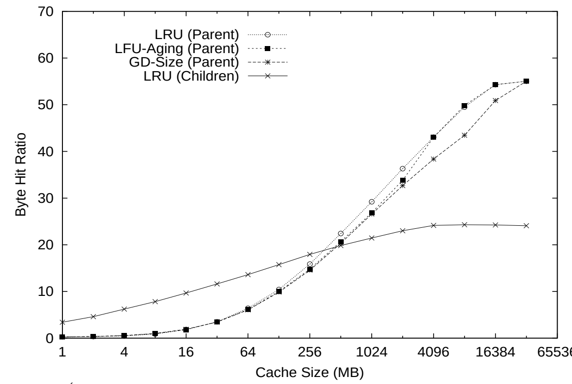
The GD-Size policy still shows better performance in hit ratios as noted earlier in Figure 5.7, although the difference is small. As before, GD-Size has byte hit ratios slightly lower than the other policies at large cache sizes. The cache ineffectiveness is also evident as observed in the previous results (e.g., Figure 5.7), but now at cache sizes larger than 4 GB (because additional large files are permitted in the cache of the lower level proxies).

The results for the partial overlap case (Figure 5.14) and the no overlap case (Figure 5.15) follow the same pattern as for the complete overlap case in Figure 5.13, with a slight increase in the performance of the lower level proxies (say, 1-2%).

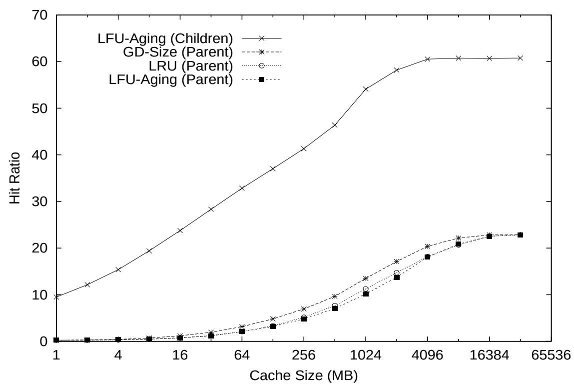
[2] 80% of the references are for files below this threshold size.



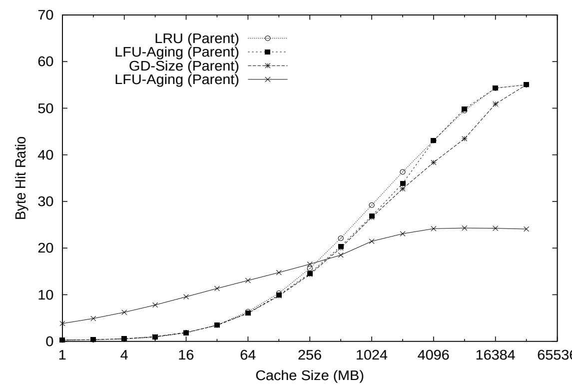
(a) Hit Ratio (Children: LRU)



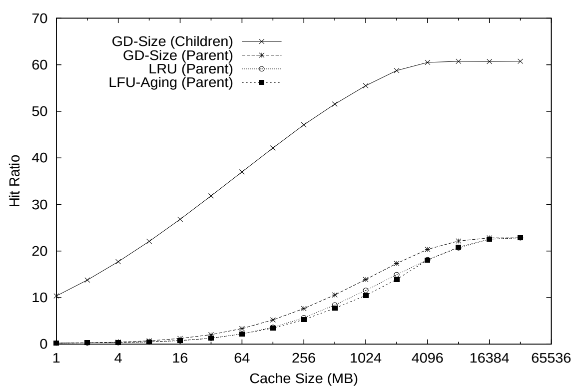
(b) Byte Hit Ratio (Children: LRU)



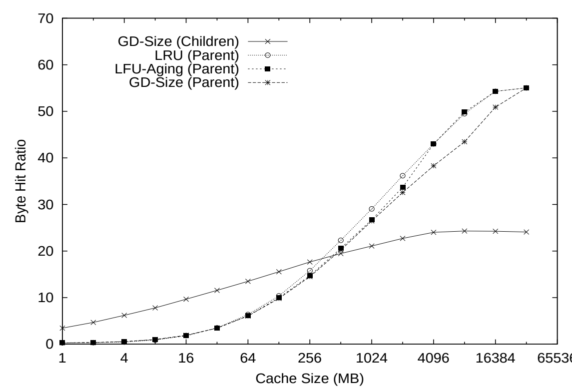
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

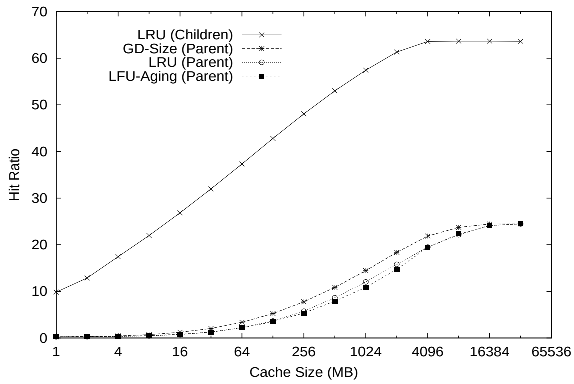


(e) Hit Ratio (Children: GD-Size)

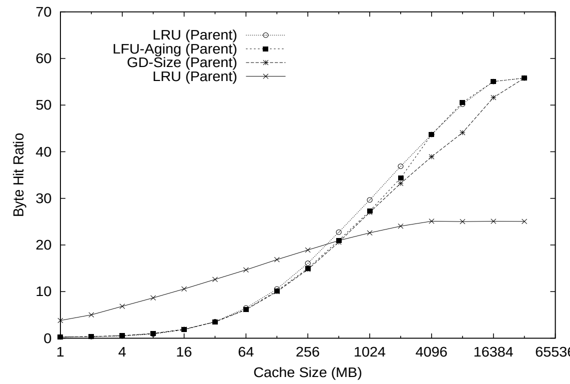


(f) Byte Hit Ratio (Children: GD-Size)

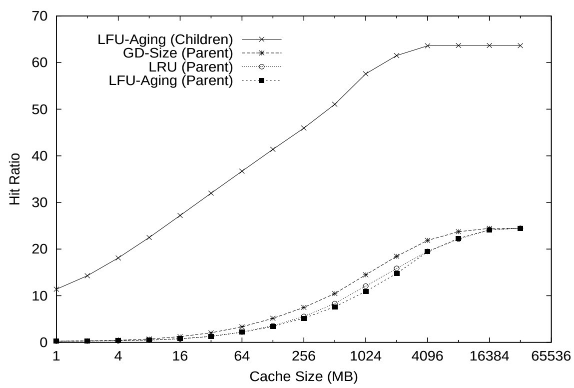
Figure 5.13: COMPLETE Overlap, Size Threshold = 10,000 Bytes: Small Files at the Lower Level; Large Files at the Upper Level.



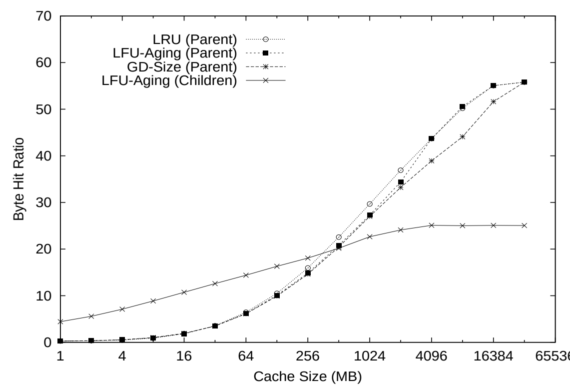
(a) Hit Ratio (Children: LRU)



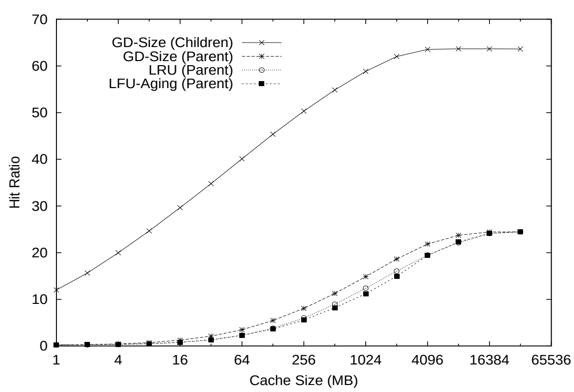
(b) Byte Hit Ratio (Children: LRU)



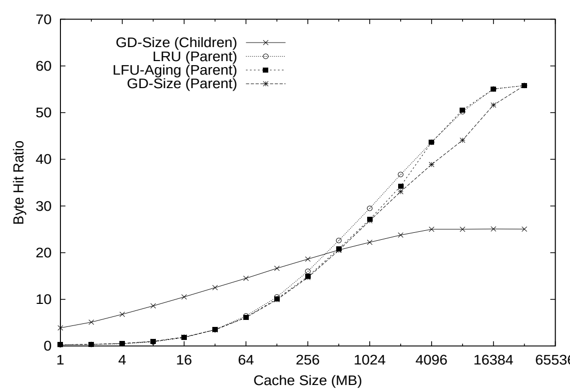
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

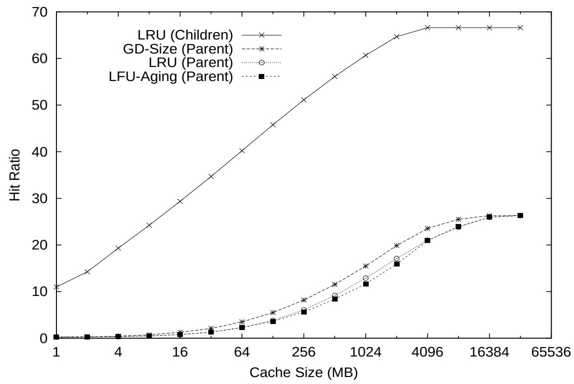


(e) Hit Ratio (Children: GD-Size)

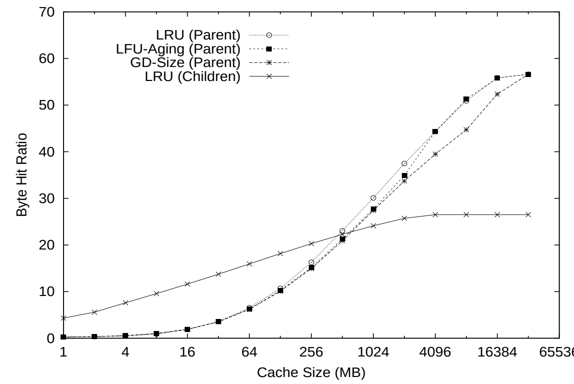


(f) Byte Hit Ratio (Children: GD-Size)

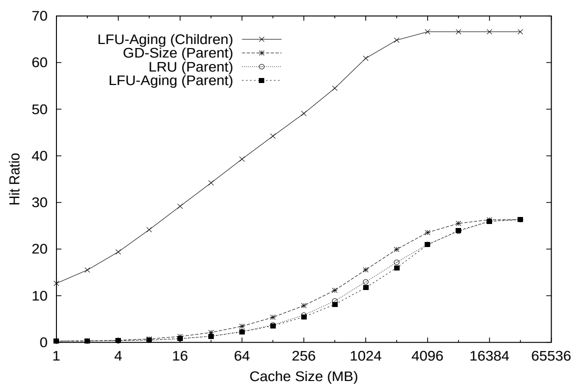
Figure 5.14: PARTIAL Overlap, Size Threshold = 10,000 Bytes: Small Files at the Lower Level; Large Files at the Upper Level.



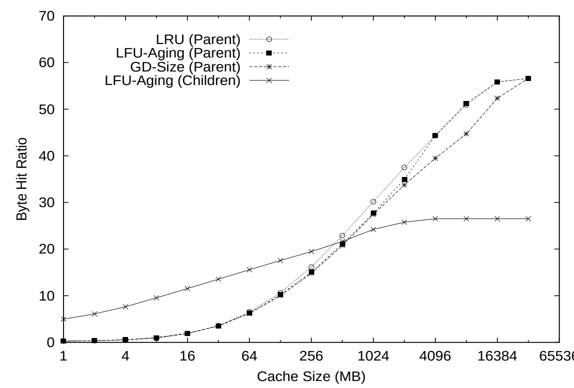
(a) Hit Ratio (Children: LRU)



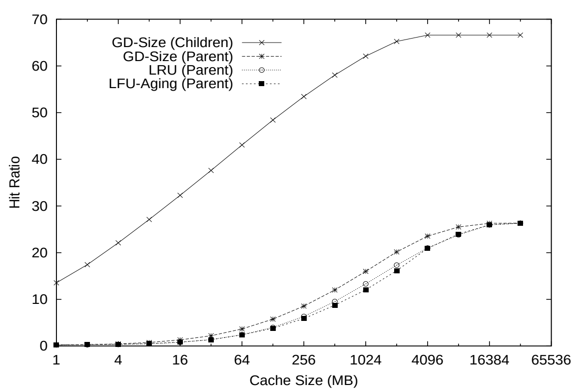
(b) Byte Hit Ratio (Children: LRU)



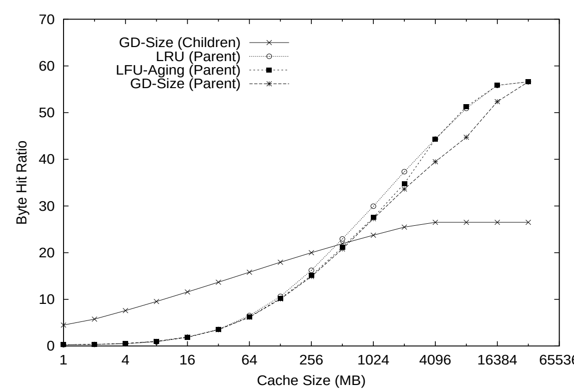
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

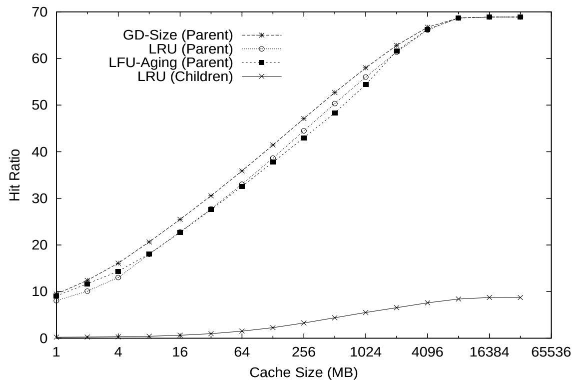


(e) Hit Ratio (Children: GD-Size)

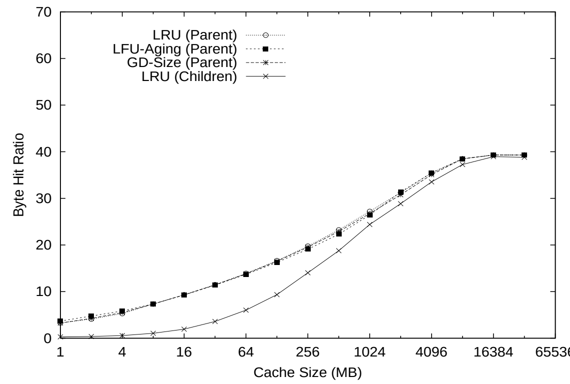


(f) Byte Hit Ratio (Children: GD-Size)

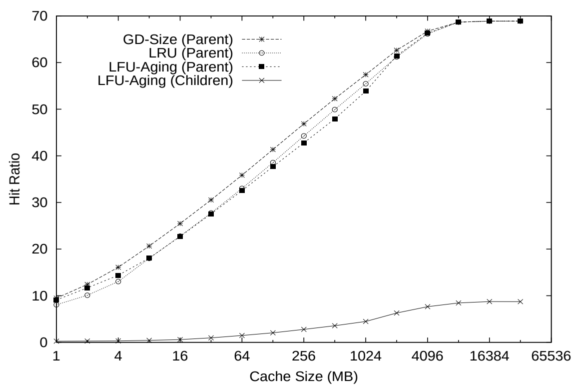
Figure 5.15: NO Overlap, Size Threshold = 10,000 Bytes: Small Files at the Lower Level; Large Files at the Upper Level.



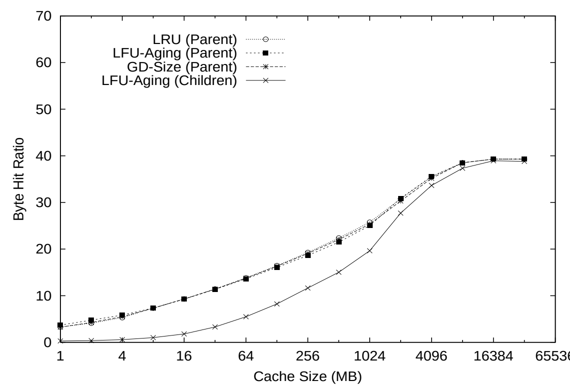
(a) Hit Ratio (Children: LRU)



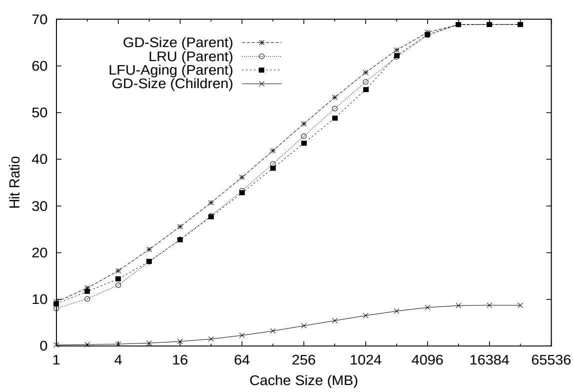
(b) Byte Hit Ratio (Children: LRU)



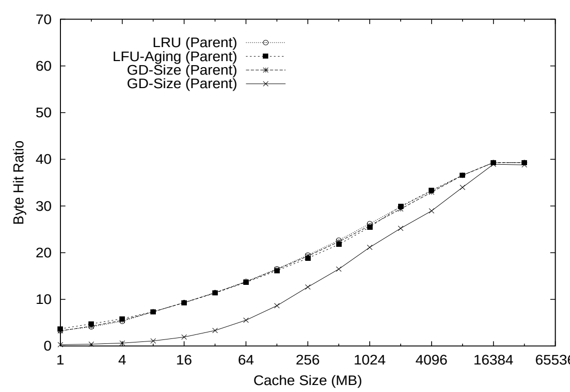
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

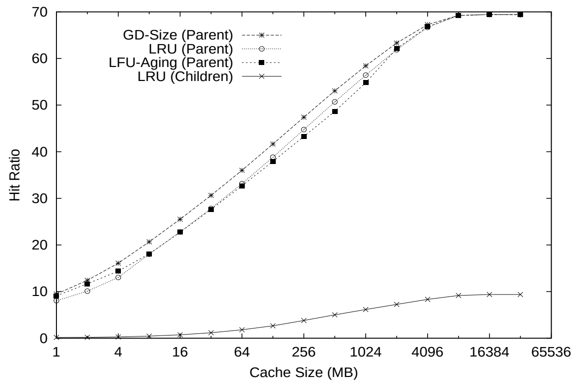


(e) Hit Ratio (Children: GD-Size)

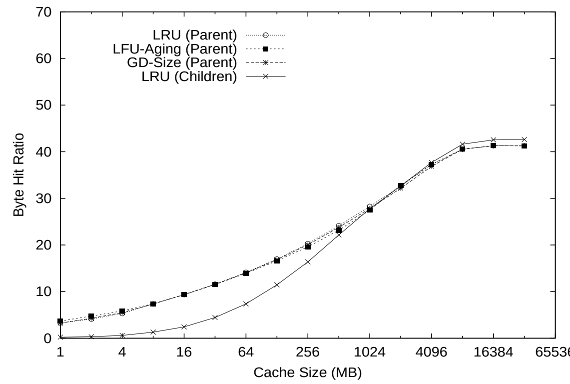


(f) Byte Hit Ratio (Children: GD-Size)

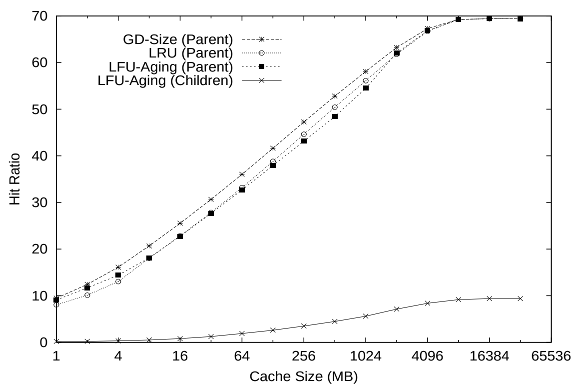
Figure 5.16: COMPLETE Overlap, Size Threshold = 10,000 Bytes: Large Files at the Lower Level; Small Files at the Upper Level.



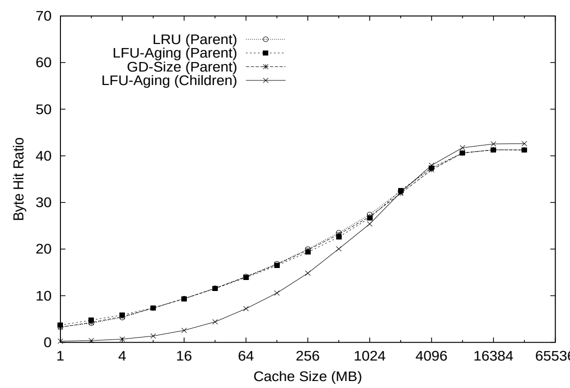
(a) Hit Ratio (Children: LRU)



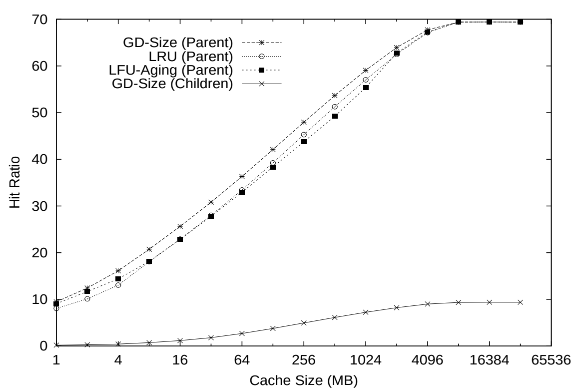
(b) Byte Hit Ratio (Children: LRU)



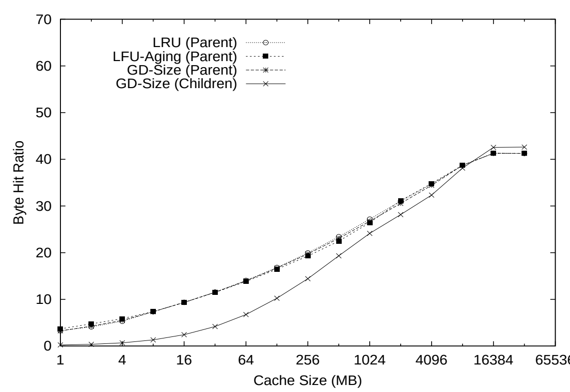
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

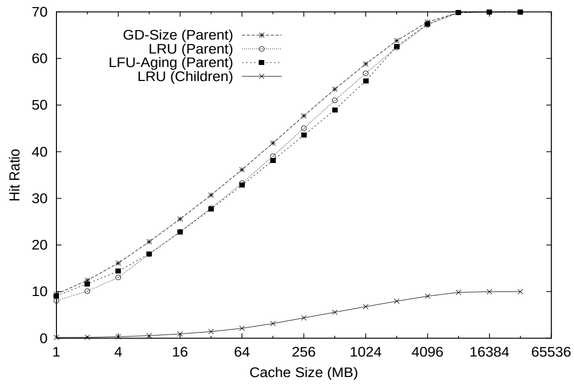


(e) Hit Ratio (Children: GD-Size)

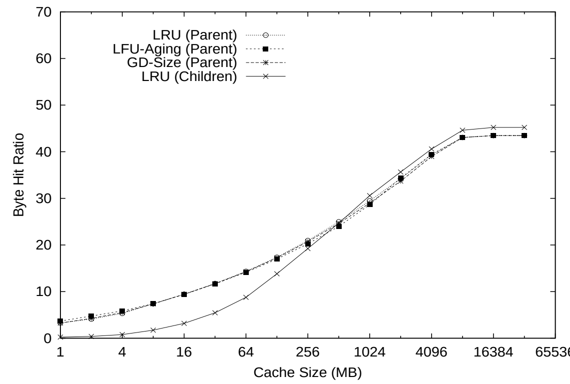


(f) Byte Hit Ratio (Children: GD-Size)

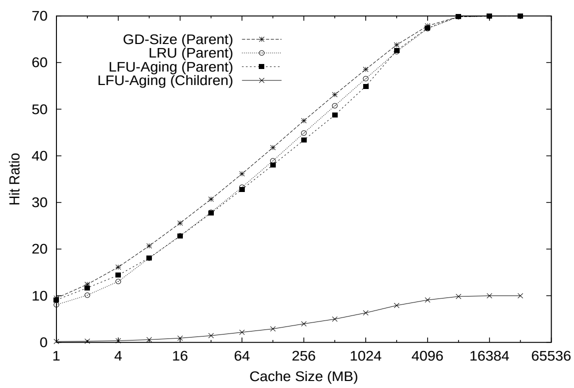
Figure 5.17: PARTIAL Overlap, Size Threshold = 10,000 Bytes: Large Files at the Lower Level; Small Files at the Upper Level.



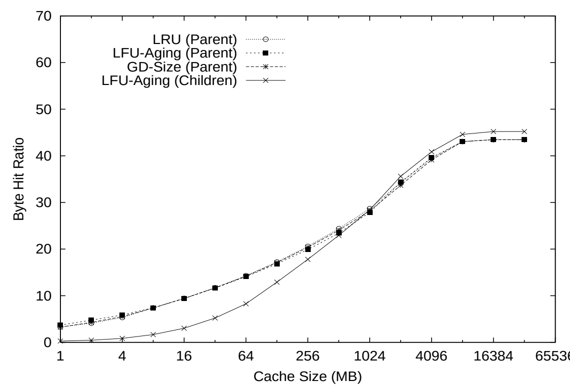
(a) Hit Ratio (Children: LRU)



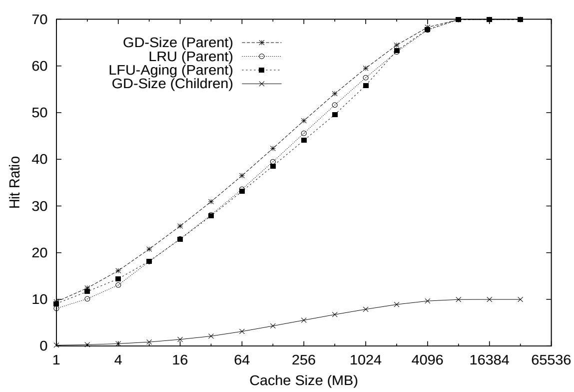
(b) Byte Hit Ratio (Children: LRU)



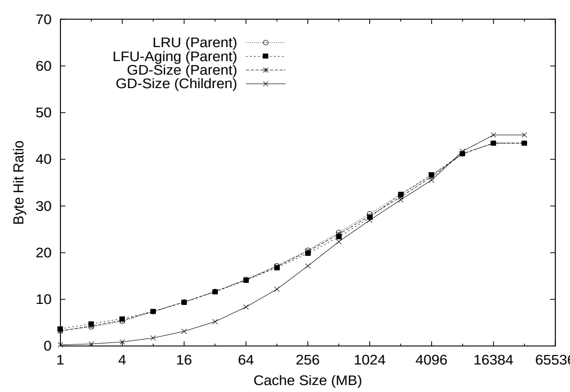
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)



(e) Hit Ratio (Children: GD-Size)



(f) Byte Hit Ratio (Children: GD-Size)

Figure 5.18: NO Overlap, Size Threshold = 10,000 Bytes: Large Files at the Lower Level; Small Files at the Upper Level.

The results for the reversed threshold policy (i.e., only files larger than 10,000 bytes are allowed in the lower level proxies, with smaller files kept in the parent's cache), are presented in Figures 5.16, 5.17, and 5.18.

The result in Figure 5.16 for complete overlap is comparable with Figure 5.10, except that the hit ratios and the byte hit ratios for the parent increase significantly, while the performance of the children drops. By keeping only large files in the lower level proxies, the byte hit ratio is still significantly high, but the hit ratio is affected by the fewer references to the large files. The GD-Size policy still provides better performance at the upper level, in terms of hit ratios, without significant loss in byte hit ratios.

5.4.3.3 Results for Threshold Size of 100,000 Bytes

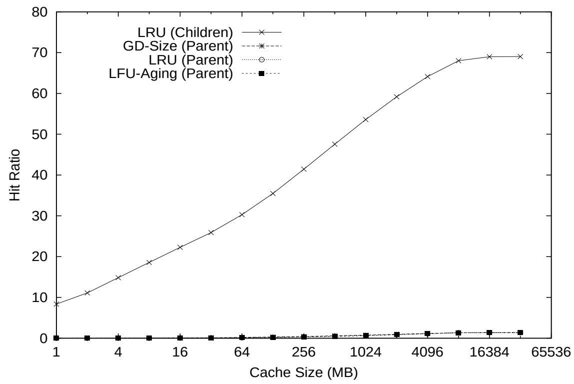
The performance results for a threshold size of 100,000 bytes[3] are presented in Figures 5.19 - 5.24. Figures 5.19, 5.20, and 5.21 show the results of keeping files smaller than 100,000 bytes at the lower level of the hierarchy and larger files at the upper level, for complete, partial, and no overlap scenarios, respectively. The performance results for reversing this policy are shown in Figures 5.22, 5.23, and 5.24, also for complete, partial, and no overlap situations, respectively.

By contrasting Figures 5.19(a) and (b) with Figures 5.13(a) and (b), a further increase in the performance of the lower level proxies is noticed, and a decrease in the performance of the parent. In fact, the hit ratios at the parent are close to zero throughout the range of cache sizes considered. But despite the poor performance in hit ratios, the byte hit ratios are still significant.

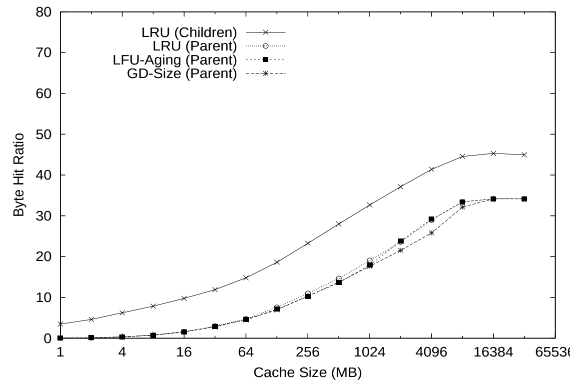
The results for the partial overlap and the no overlap cases in Figures 5.20 and 5.21 follow the same pattern as for the complete overlap case in Figure 5.19, with a slight increase in the performance of the lower level proxies.

The results in Figures 5.22, 5.23, and 5.24 show the effect of reversing the size threshold policy such that only files larger than 100,000 bytes can be cached at

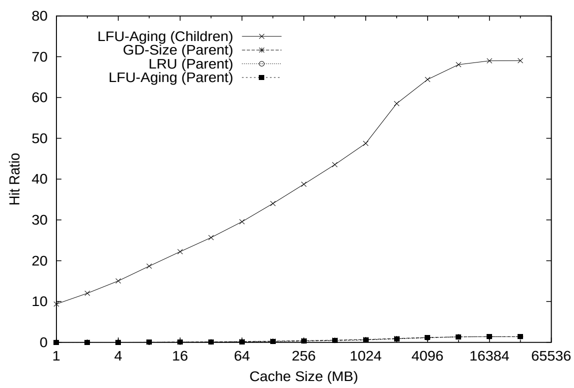
[3] 99% of the references are for files below this threshold size.



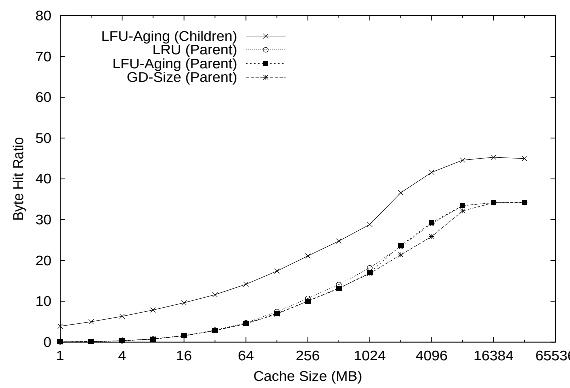
(a) Hit Ratio (Children: LRU)



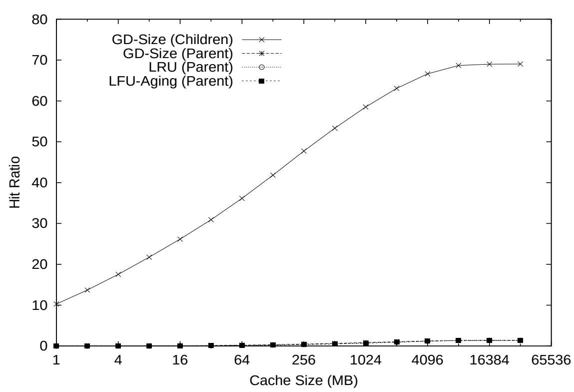
(b) Byte Hit Ratio (Children: LRU)



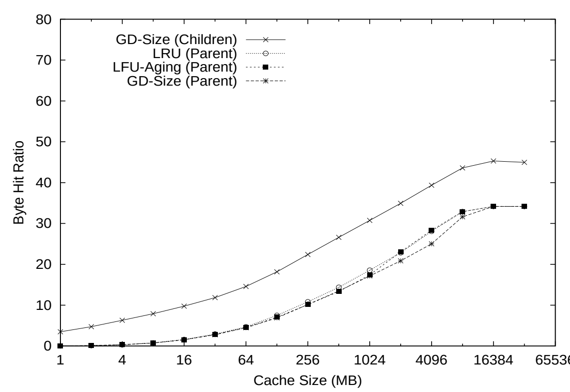
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

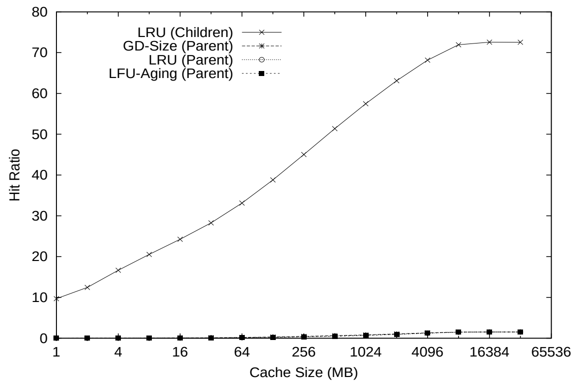


(e) Hit Ratio (Children: GD-Size)

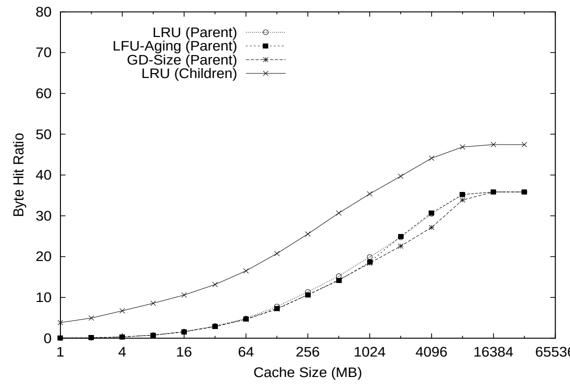


(f) Byte Hit Ratio (Children: GD-Size)

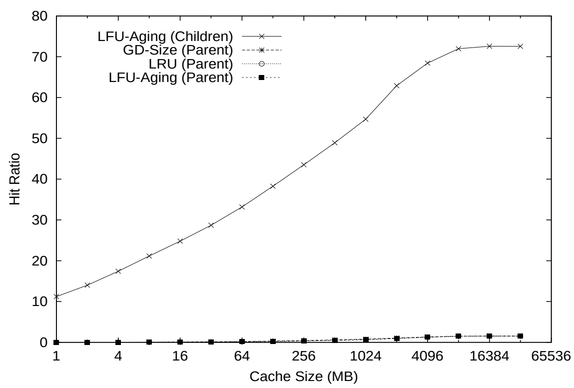
Figure 5.19: COMPLETE Overlap, Size Threshold = 100,000 Bytes: Small Files at the Lower Level; Large Files at the Upper Level.



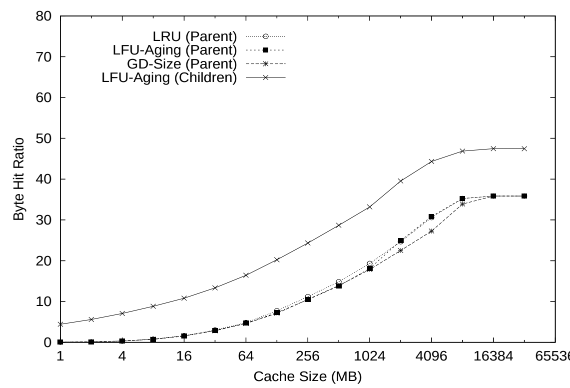
(a) Hit Ratio (Children: LRU)



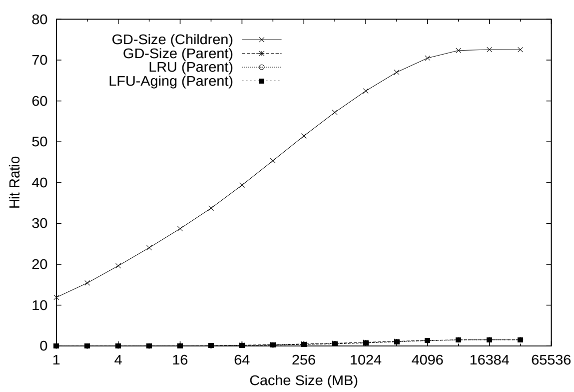
(b) Byte Hit Ratio (Children: LRU)



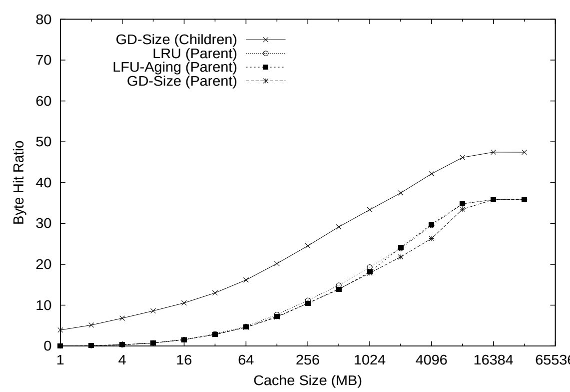
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

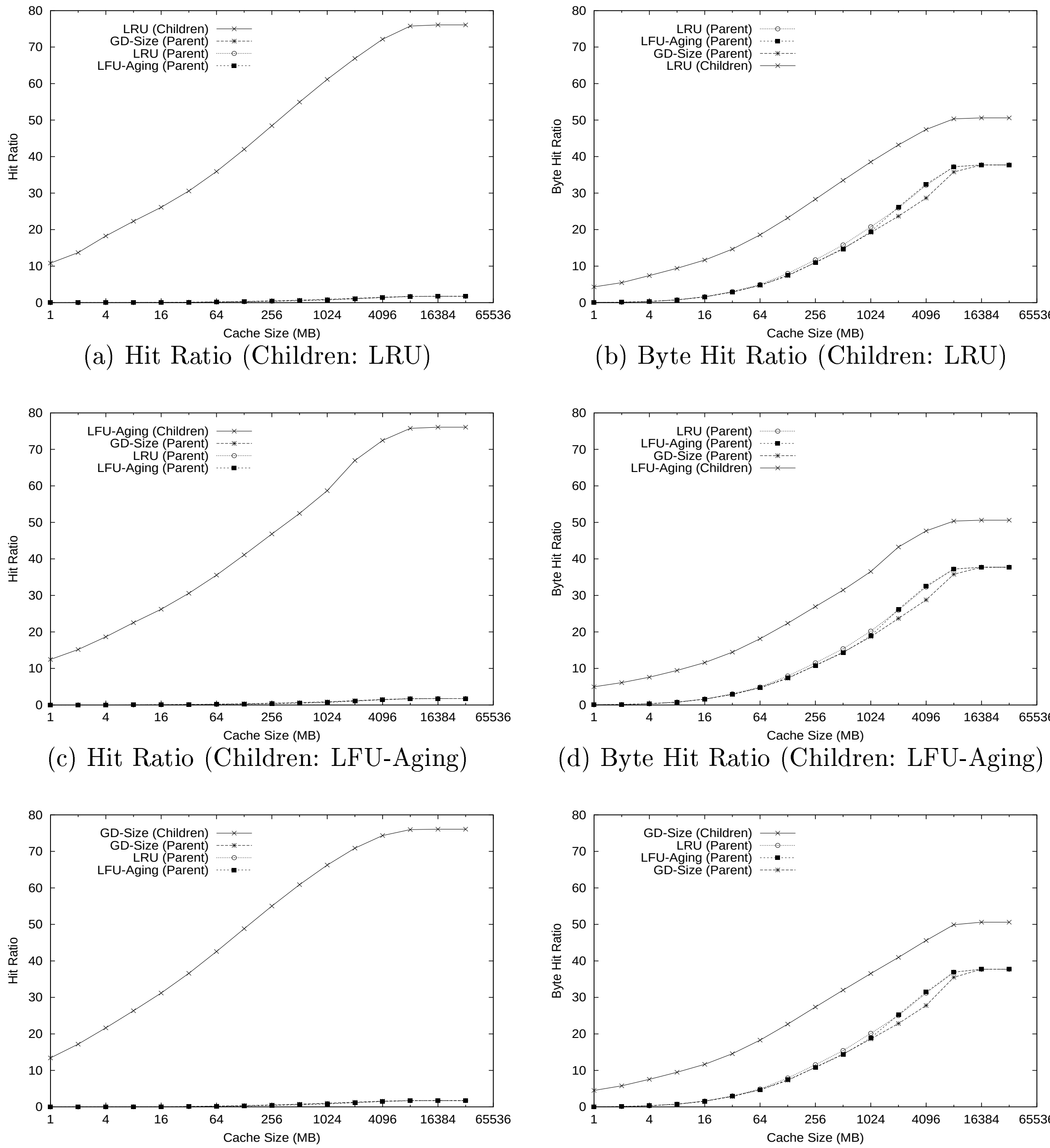


(e) Hit Ratio (Children: GD-Size)



(f) Byte Hit Ratio (Children: GD-Size)

Figure 5.20: PARTIAL Overlap, Size Threshold = 100,000 Bytes: Small Files at the Lower Level; Large Files at the Upper Level.



(a) Hit Ratio (Children: LRU)

(b) Byte Hit Ratio (Children: LRU)

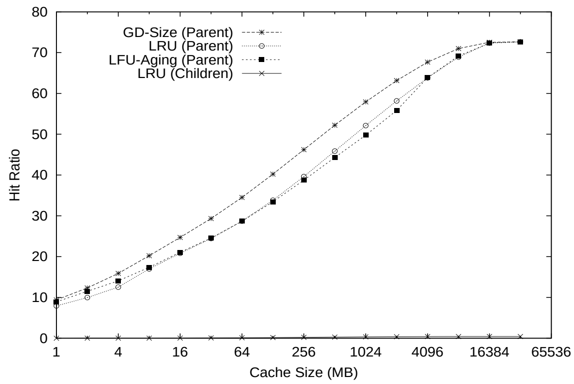
(c) Hit Ratio (Children: LFU-Aging)

(d) Byte Hit Ratio (Children: LFU-Aging)

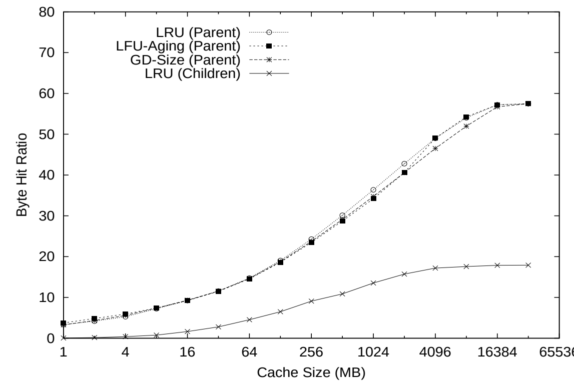
(e) Hit Ratio (Children: GD-Size)

(f) Byte Hit Ratio (Children: GD-Size)

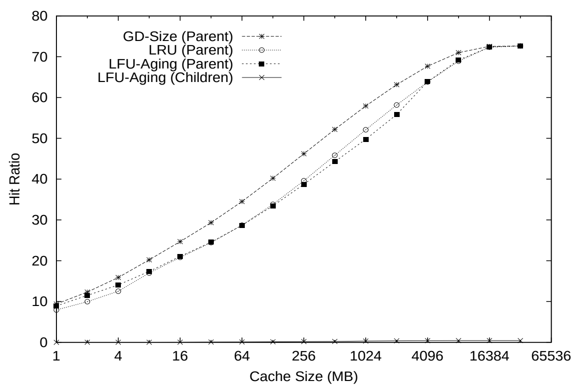
Figure 5.21: NO Overlap, Size Threshold = 100,000 Bytes: Small Files at the Lower Level; Large Files at the Upper Level.



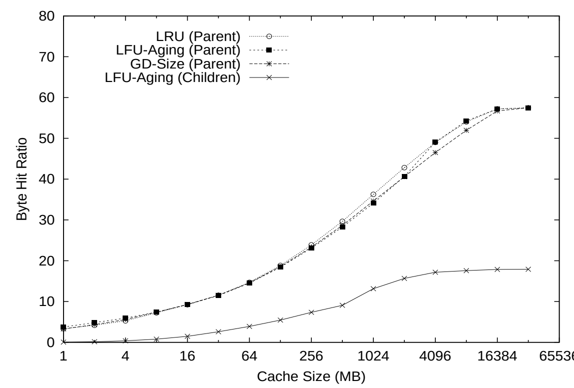
(a) Hit Ratio (Children: LRU)



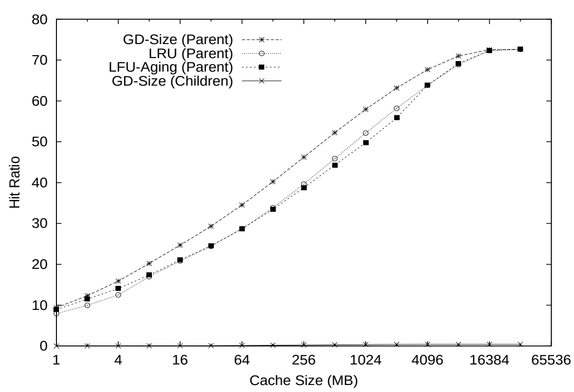
(b) Byte Hit Ratio (Children: LRU)



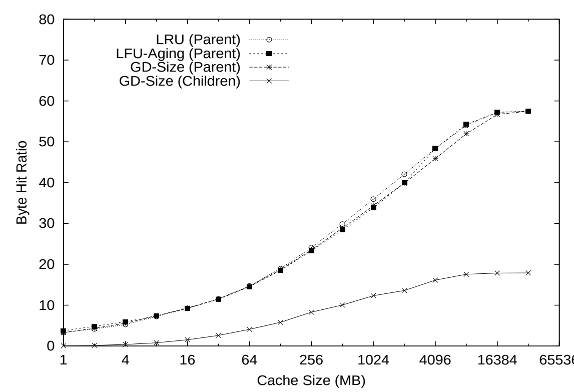
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

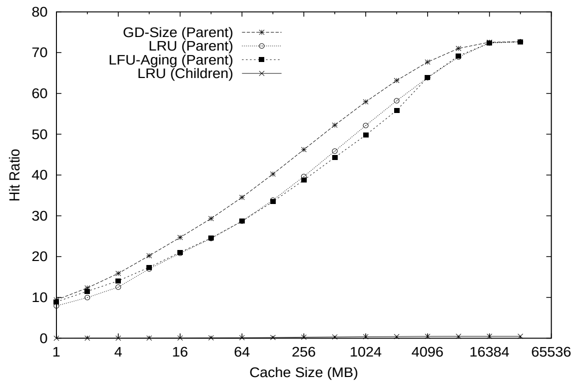


(e) Hit Ratio (Children: GD-Size)

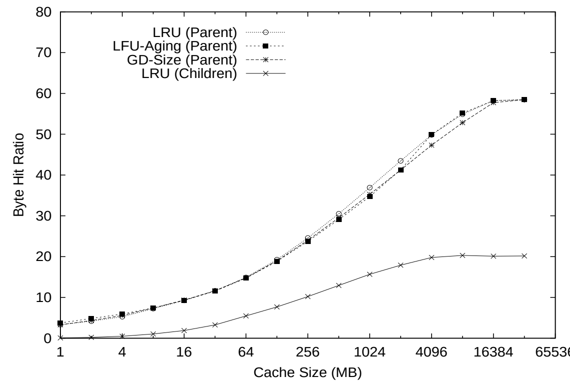


(f) Byte Hit Ratio (Children: GD-Size)

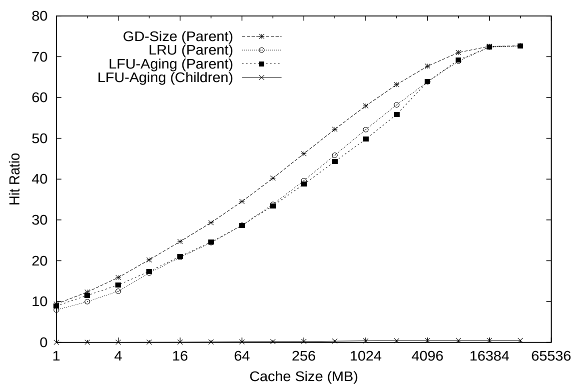
Figure 5.22: COMPLETE Overlap, Size Threshold = 100,000 Bytes: Large Files at the Lower Level; Small Files at the Upper Level.



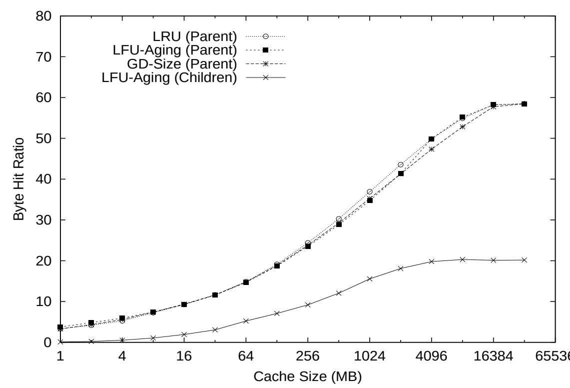
(a) Hit Ratio (Children: LRU)



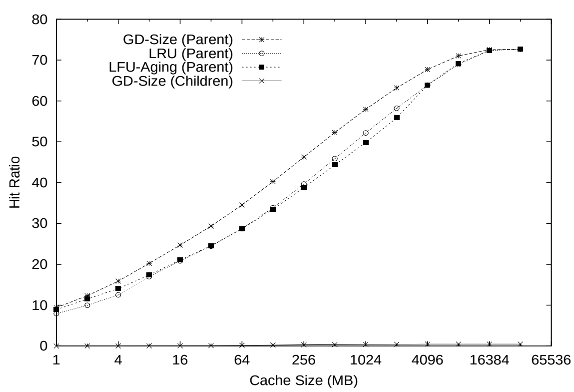
(b) Byte Hit Ratio (Children: LRU)



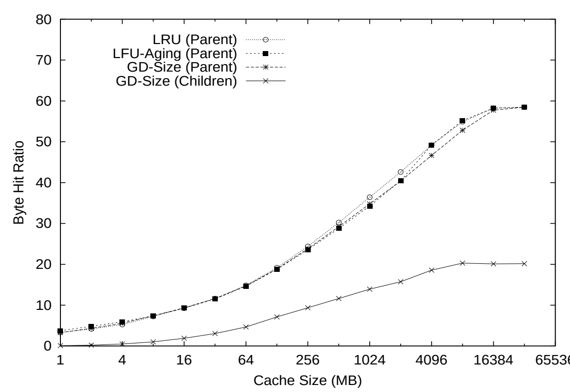
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)

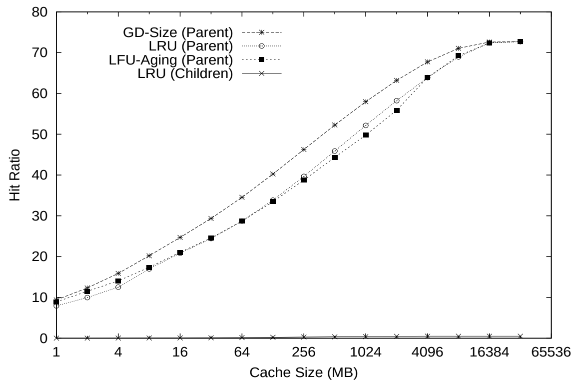


(e) Hit Ratio (Children: GD-Size)

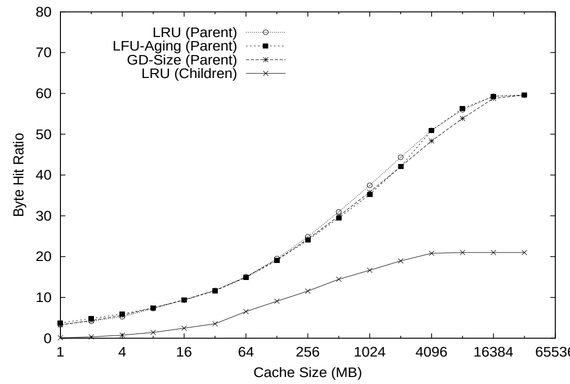


(f) Byte Hit Ratio (Children: GD-Size)

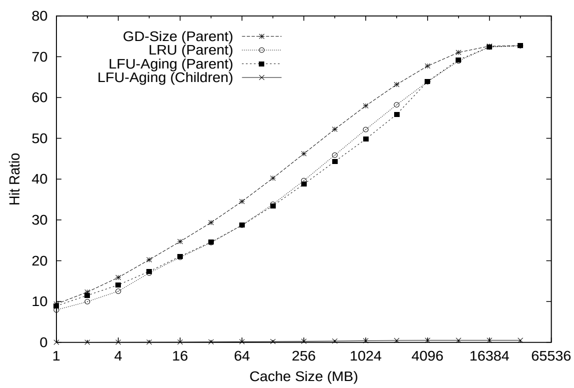
Figure 5.23: PARTIAL Overlap, Size Threshold = 100,000 Bytes: Large Files at the Lower Level; Small Files at the Upper Level.



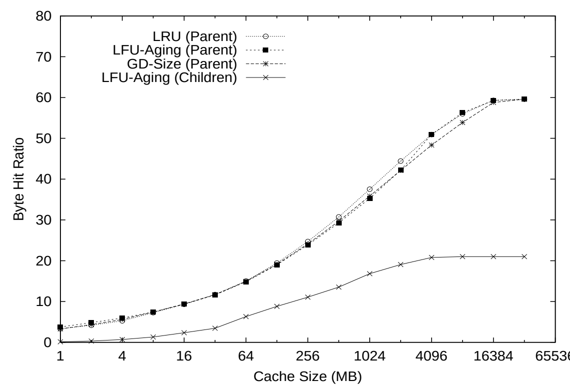
(a) Hit Ratio (Children: LRU)



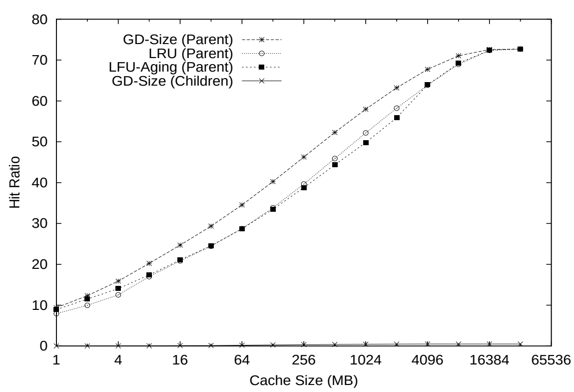
(b) Byte Hit Ratio (Children: LRU)



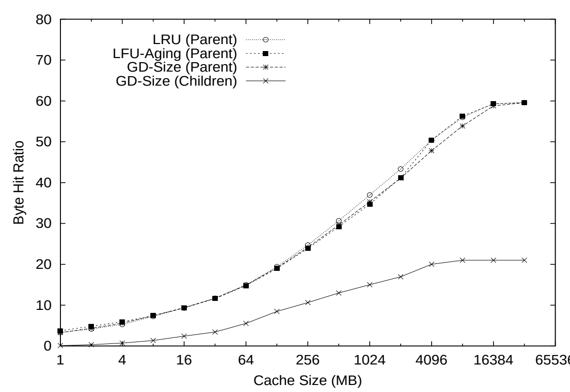
(c) Hit Ratio (Children: LFU-Aging)



(d) Byte Hit Ratio (Children: LFU-Aging)



(e) Hit Ratio (Children: GD-Size)



(f) Byte Hit Ratio (Children: GD-Size)

Figure 5.24: NO Overlap, Size Threshold = 100,000 Bytes: Large Files at the Lower Level; Small Files at the Upper Level.

the lower level proxies, while only smaller files are cached at the upper level of the hierarchy. In this case (e.g., in Figure 5.22), the hit ratios at the lower level proxies are almost zero throughout, and the byte hit ratios are less than 20%. The performance of the parent shows that a significant percentage of the requests and the bytes transferred are handled at this level.

In summary, from the threshold sizes studied, keeping smaller files at the lower level provides good hit ratios (not necessarily better) and worse byte hit ratios than using no threshold policy at all. This decreases the hit ratios at the upper level although the byte hit ratios there are still high. Increasing the threshold size to accommodate larger files at the lower level cache increases performance at the lower level and decreases it at the upper level. Reversing this strategy so that only large files are stored at the lower level while small files are stored at the upper level results in a reversal of the previous results.

The analysis provided in the next section offers more insight into which of the strategies presented so far can provide an overall improvement in Web caching performance.

5.5 Cost-Benefit Analysis of Caching Strategies

Analysis of the experimental results presented so far in this chapter show that several caching strategies could improve the performance of a caching hierarchy. While some of these strategies improve performance at the lowest level, others can only provide improvement at the upper level. The choice of which strategy to use depends on whether to maximize the hit ratios or the byte hit ratios at the lower level or the upper level of the hierarchy. This invariably depends on the costs associated with the two levels of the hierarchy, in terms of data transfer or delay that could result from sending a request from the lower level proxies to the upper level proxy. This section gives further analysis (using the only performance statistics obtained from the experiments, namely the hit ratios and the byte hit ratios) to show the relative benefit of choosing a particular caching strategy from the options evaluated so far.

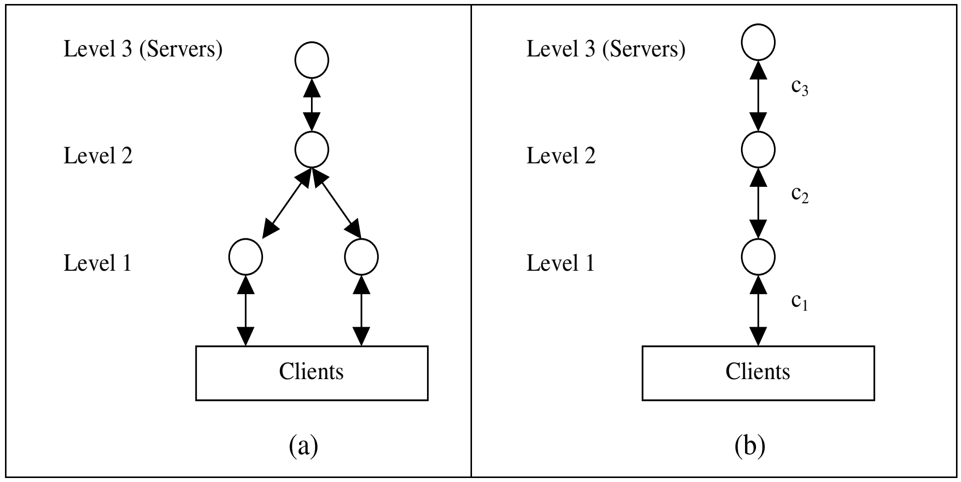


Figure 5.25: Analysis of Two-Level Hierarchical Simulation Model.

First, the two-level hierarchical proxy model in Figure 5.25(a) is represented by the model in Figure 5.25(b). Note that the two lower level proxies in Figure 5.25(a) are represented with a single node in Figure 5.25(b), since the two proxies provide similar hit ratios and byte hit ratios for the same cache sizes. For instance, if the hit ratio for a given cache size is HR, and a total of N requests originate at the client level, of which $N/2$ are sent to each of the lower level proxies, then the total number of requests N_l satisfied at the left proxy is $N_l = HR \times N/2$, while the total number of requests N_r satisfied at the right proxy is $N_r = HR \times N/2$. It follows that $N_l = N_r$ and the overall total number of requests N_1 satisfied at the first level of the hierarchy is $N_1 = N_l + N_r$. In other words, $N_1 = (HR \times N/2) + (HR \times N/2) = HR \times N$. This implies that if if the two lower level proxies are represented by a single node as in Figure 5.25(b), calculating the total number of requests satisfied at the first level requires only multiplying the hit ratio of one of the proxies by the total requests originated at the client level.

By representing the total number of requests originated at the client level by N, the total content bytes of the N requests by B, the hit ratio at level j by HR_j, and the byte hit ratio at level j by BHR_j for $1 \leq j \leq 3$ (Note that $HR_3 = BHR_3 = 1$, by assumption), then N_j and B_j, the number of requests and bytes satisfied at level

j can be calculated as

$$N_j = HR_j \times \left(N - \sum_{i=1}^{j-1} N_i \right)$$

$$B_j = BHR_j \times \left(B - \sum_{i=1}^{j-1} B_i \right).$$

In other words, the requests (or bytes) satisfied at each level j is determined from the hit ratio (or the byte hit ratio) at level j multiplied by the total requests (or bytes) originated from the client level, less those requests (or bytes) already satisfied at previous levels of the caching hierarchy.

If the cost (in network hops or delay) associated with the link between two adjacent levels is denoted by c_j, for $1 \leq j \leq 3$ as shown in Figure 5.25, then the cost associated with satisfying N_i requests at level i is

$$(c_1 + c_2 + \cdots + c_i)N_i = (\sum_{j=1}^{i} c_j)N_i.$$

Note that this cost estimate ignores the document size associated with each request. Similarly, the cost associated with satisfying B_i bytes at level i is

$$(c_1 + c_2 + \cdots + c_i)B_i = (\sum_{j=1}^{i} c_j)B_i.$$

In general, if the total cost associated with satisfying all requests and bytes requested by the clients are denoted by TRC and TBC, respectively, then

$$TRC = (c_1)N_1 + (c_1 + c_2)N_2 + (c_1 + c_2 + c_3)N_3 = \sum_{i=1}^{3} \left\{ \left(\sum_{j=1}^{i} c_j \right) N_i \right\}$$

$$TBC = (c_1)B_1 + (c_1 + c_2)B_2 + (c_1 + c_2 + c_3)B_3 = \sum_{i=1}^{3} \left\{ \left(\sum_{j=1}^{i} c_j \right) B_i \right\}.$$

In the worst case, all requests and bytes are satisfied at level 3, in which case, the total cost associated with the N requests is $(c_1 + c_2 + c_3)N = (\sum_{i=1}^{3} c_i)N$. Similarly for the B bytes, the cost is $(c_1 + c_2 + c_3)B = (\sum_{i=1}^{3} c_i)B$. Normalizing TRC and

TBC with the costs corresponding to each worst case scenario, respectively, yields:

$$\overline{TRC} = \frac{\sum_{i=1}^{3}\left\{\left(\sum_{j=1}^{i} c_j\right) N_i\right\}}{\left(\sum_{i=1}^{3} c_i\right) N}$$

$$\overline{TBC} = \frac{\sum_{i=1}^{3}\left\{\left(\sum_{j=1}^{i} c_j\right) B_i\right\}}{\left(\sum_{i=1}^{3} c_i\right) B}.$$

For simplicity in this analysis, it is assumed that $c_1 = c_2 = c_3 = c$, then $\overline{TRC}$ and $\overline{TBC}$ become:

$$\overline{TRC} = \frac{N_1 + 2N_2 + 3N_3}{3N}$$

$$\overline{TBC} = \frac{B_1 + 2B_2 + 3B_3}{3B}.$$

The smaller the value of the measure $\overline{TRC}$ and $\overline{TBC}$, the better a policy is. Note that the best performance occurs when all requests are satisfied at level 1, i.e., $N_1 = N$ and $B_1 = B$. In this case, $N_2 = B_2 = N_3 = B_3 = 0$ and $\overline{TRC} = \overline{TBC} = 0.3333$. The worst performance has $\overline{TRC} = \overline{TBC} = 1.0$.

5.5.1 Computation of $\overline{TRC}$ and $\overline{TBC}$ for Selected Caching Strategies

This section presents the computation of $\overline{TRC}$ and $\overline{TBC}$ for some selected caching strategies involving combinations of the LRU and GD-Size policies. The LRU policy was chosen for analysis over the LFU-Aging policy for two reasons: (1) It provides a simple basis against which to compare other caching policies, because of its use in most publicly available proxy software; (2) LFU-Aging is a parameterized policy, while LRU is not. In fact, the experimental results presented so far indicate no significant benefit of the LFU-Aging policy over LRU, perhaps because of a poor choice of the aging parameter or exaggerated temporal locality in the workload used. However, it is arguable that in this analysis, wherever the LRU policy is a good choice, the LFU-Aging policy would provide at least the same performance.

From the analysis of the results presented earlier, it was evident that the general pattern of behaviours observed was consistent, regardless of whether there is complete, partial, or no overlap in the workloads of the lower level proxies. For all the computations in this section, as a matter of choice, results from the partial overlap scenarios are used.

Also, because of the difficulty of generalizing the hit ratios and the byte hit ratios obtained across the cache sizes, $\overline{TRC}$ and $\overline{TBC}$ are computed for only three different cache sizes. The chosen sizes are 64 MB, 512 MB, and 1024 MB, to represent small, medium, and large cache sizes, respectively. The results of the computations are presented in Tables 5.3 - 5.10.

5.5.1.1 Analysis of $\overline{TRC}$ and $\overline{TBC}$ for Different Policies at Different Levels of the Hierarchy

The computed values of $\overline{TRC}$ and $\overline{TBC}$ for different combinations of LRU and GD-Size are tabulated in Table 5.3. The first combination (i.e., LRU at both levels) can be used as a standard to gauge the performance of other combinations for caching strategies. Any caching strategy that provides lower values of $\overline{TRC}$ and $\overline{TBC}$ across all the cache sizes considered is regarded as superior to the LRU-LRU combination.

Table 5.3: Cost-Benefit Analysis for Different Replacement Policies at Different Levels of the Hierarchy

No	Policy at Level 1	Policy at Level 2	$\overline{TRC}$			$\overline{TBC}$		
			64 MB	512 MB	1024 MB	64 MB	512 MB	1024 MB
1	LRU	LRU	0.7936	0.6765	0.6347	0.8838	0.7495	0.6988
2	LRU	GD	0.7755	0.6557	0.6151	**0.8817**	**0.7475**	**0.6974**
3	GD	LRU	0.7380	0.6167	0.5807	0.8818	0.7644	0.7218
4	GD	GD	**0.7354**	**0.6136**	**0.5776**	0.8872	0.7785	0.7382

A critical look across the four combinations from Table 5.3 shows that combinations involving the GD-Size policy seem to provide lower values of $\overline{TRC}$ and $\overline{TBC}$, especially the second combination in this table, which uses LRU at the lower level and GD-Size at the upper level. This combination gives lower values of $\overline{TRC}$ and $\overline{TBC}$ across all the cache sizes, compared to the first combination of LRU at both levels. This result is consistent with what was observed in Figures 5.4(a) and (b) on page 99. The third combination shows that using GD-Size at the lower level and LRU at the upper level provided even better performance than the first two combinations in terms of $\overline{TRC}$, but lags behind them in terms of $\overline{TBC}$ as the cache size increases. The last combination (GD-Size at both levels), improves $\overline{TRC}$ over the third one, but is worse in terms of $\overline{TBC}$. The values highlighted in bold font in Table 5.3 show the lowest values of $\overline{TRC}$ and $\overline{TBC}$ achieved for each cache size, among the policies considered.

In general from Table 5.3, the second combination provides better performance in all respects than the first combination, but the choice of whether the third or the fourth combination should be preferred over the first will depend on whether or not reducing $\overline{TRC}$ at the expense of $\overline{TBC}$ is a good tradeoff.

5.5.1.2 Analysis of $\overline{TRC}$ and $\overline{TBC}$ for Keeping Shared Files at the Upper Level of the Hierarchy

$\overline{TRC}$ and $\overline{TBC}$ computations for keeping shared files at the upper level of the hierarchy are shown in Table 5.4. A single LRU-LRU combination was considered in this experiment.

Table 5.4: Cost-Benefit Analysis for Keeping Shared Files at the Upper Level of the Hierarchy

No	Policy at Level 1	Policy at Level 2	$\overline{TRC}$			$\overline{TBC}$		
			64 MB	512 MB	1024 MB	64 MB	512 MB	1024 MB
1 (P)	LRU	LRU	0.8217	0.7299	0.6800	0.8867	0.7793	0.7428
2 (C)	LRU	LRU	0.9107	0.8539	0.8288	0.9511	0.8887	0.8589
3 (N)	LRU	LRU	0.7799	0.6577	0.6156	0.8720	0.7293	0.6777

Comparing the results from the first entry of this table (for partial workload overlap) with Table 5.3 shows that this caching strategy does not provide any performance gain. In fact, it is worse than every combination in Table 5.3. However, since this caching strategy depends heavily on the level of sharing between the workloads of the lower level proxies, it may perform better than some of the combinations in Table 5.3 if the workload overlap changes.

To clarify this supposition, $\overline{TRC}$ and $\overline{TBC}$ are computed again, but for the complete overlap and the no overlap scenarios, as shown in the second and the third entry of Table 5.4, respectively. As can be seen from these results, the complete overlap scenario worsens the situation. This is because as the level of sharing increases, the usefulness of the lower level proxies decreases because more and more requests go to the parent proxy before being satisfied. On the other hand, as the level of sharing decreases, the lower level proxies become more useful, gradually rendering the upper level proxy useless. At the extreme end, when there is no overlap (the third entry of the table) the hierarchy operates as if the upper level proxy does not exist.

At first glance, the results shown in the third entry of Table 5.4 seem better than some of the results in Table 5.3. However, the results in Table 5.3 are for the partial overlap scenario, while the results in the third entry of Table 5.4 are for the no overlap scenario. In fact, if similar results as in Table 5.3 were to be computed for the no overlap scenario, the third entry of Table 5.4 would provide the worst performance. Evidence for this can be deduced by comparing Figures 5.6(e) and (f) on page 105 with Figures 5.5(a) and (b) on page 100. Note that, in these figures, the hit ratios and the byte hit ratios of the lower level proxies are the same, but in addition, Figures 5.5(a) and (b) provide additional hits because of the upper level proxy, which Figures 5.6(e) and (f) do not provide. Thus, there seems to be *no performance advantage to the sharing-based approach.* Furthermore, the extra cost associated with the extra information and communication resulting from keeping the list of shared files consistent in the hierarchy makes this strategy less desirable than the policies in Table 5.3.

5.5.1.3 Analysis of $\overline{TRC}$ and $\overline{TBC}$ for Partitioning Files across the Hierarchy based on Size

The computed values of $\overline{TRC}$ and $\overline{TBC}$ for the size threshold strategies are shown in Tables 5.5 - 5.10. A quick comparison of the values in these tables with those in Table 5.3 shows that strategies that keep *larger files at the lower level* and smaller files at the upper level (Tables 5.6, 5.8, 5.10) *cannot provide any performance improvement* over any of the values in Table 5.3. That is, all $\overline{TRC}$ and $\overline{TBC}$ values computed in these tables are greater than the corresponding values in Table 5.3. Thus, these policies can be disregarded while focusing on the remaining strategies.

Table 5.5: Cost-Benefit Analysis for Keeping Small Files at the Lower Level and Large Files at the Upper Level (Size Threshold = 5,000 Bytes)

No	Policy at Level 1	Policy at Level 2	$\overline{TRC}$			$\overline{TBC}$		
			64 MB	512 MB	1024 MB	64 MB	512 MB	1024 MB
1	LRU	LRU	0.7606	0.6688	0.6421	**0.9132**	**0.8451**	**0.8202**
2	LRU	GD	0.7549	0.6607	0.6343	0.9140	0.8516	0.8289
3	GD	LRU	0.7516	0.6640	0.6399	0.9138	0.8461	0.8209
4	GD	GD	**0.7459**	**0.6559**	**0.6321**	0.9145	0.8525	0.8295

Table 5.6: Cost-Benefit Analysis for Keeping Large Files at the Lower Level and Small Files at the Upper Level (Size Threshold = 5,000 Bytes)

No	Policy at Level 1	Policy at Level 2	$\overline{TRC}$			$\overline{TBC}$		
			64 MB	512 MB	1024 MB	64 MB	512 MB	1024 MB
1	LRU	LRU	0.8441	0.7617	0.7346	**0.8921**	**0.7676**	**0.7213**
2	LRU	GD	0.8393	0.7584	0.7323	0.8922	0.7680	0.7217
3	GD	LRU	0.8306	0.7451	0.9191	0.8953	0.7898	0.7496
4	GD	GD	**0.8258**	**0.7419**	**0.7167**	0.8954	0.7902	0.7500

Table 5.7: Cost-Benefit Analysis for Keeping Small Files at the Lower Level and Large Files at the Upper Level (Size Threshold = 10,000 Bytes)

No	Policy at Level 1	Policy at Level 2	$\overline{TRC}$			$\overline{TBC}$		
			64 MB	512 MB	1024 MB	64 MB	512 MB	1024 MB
1	LRU	LRU	0.7466	0.6331	0.6000	**0.8839**	**0.8003**	**0.7729**
2	LRU	GD	0.7442	0.6296	0.5966	0.8849	0.8060	0.7797
3	GD	LRU	0.7281	0.6208	0.5907	0.8849	0.8029	0.7754
4	GD	GD	**0.7257**	**0.6173**	**0.5873**	0.8859	0.8085	0.7823

Tables 5.5, 5.7, and 5.9 exhibit a particular trend worth mentioning. A critical look at the values for $\overline{TRC}$ and $\overline{TBC}$ in these three tables shows that changing the threshold size from 5,000 bytes (Table 5.5) to 10,000 bytes (Table 5.7) results in significant reductions in $\overline{TRC}$ and $\overline{TBC}$ values for corresponding rows of the tables.

Table 5.8: Cost-Benefit Analysis for Keeping Large Files at the Lower Level and Small Files at the Upper Level (Size Threshold = 10,000 Bytes)

No	Policy at Level 1	Policy at Level 2	$\overline{TRC}$			$\overline{TBC}$		
			64 MB	512 MB	1024 MB	64 MB	512 MB	1024 MB
1	LRU	LRU	0.8794	0.8061	0.7825	**0.9073**	**0.7898**	**0.7469**
2	LRU	GD	0.8700	0.7986	0.7762	0.9075	0.7907	0.7480
3	GD	LRU	0.8736	0.7988	0.7754	0.9114	0.8082	0.7704
4	GD	GD	**0.8642**	**0.7912**	**0.7691**	0.9116	0.8091	0.7715

Table 5.9: Cost-Benefit Analysis for Keeping Small Files at the Lower Level and Large Files at the Upper Level (Size Threshold = 100,000 Bytes)

No	Policy at Level 1	Policy at Level 2	$\overline{TRC}$			$\overline{TBC}$		
			64 MB	512 MB	1024 MB	64 MB	512 MB	1024 MB
1	LRU	LRU	0.7788	0.6566	0.6158	**0.8765**	**0.7603**	**0.7212**
2	LRU	GD	0.7787	0.6564	0.6156	0.8772	0.7625	0.7245
3	GD	LRU	0.7371	0.6179	0.5827	0.8787	0.7704	0.7346
4	GD	GD	**0.7370**	**0.6177**	**0.5826**	0.8793	0.7726	0.7379

Table 5.10: Cost-Benefit Analysis for Keeping Large Files at the Lower Level and Small Files at the Upper Level (Size Threshold = 100,000 Bytes)

No	Policy at Level 1	Policy at Level 2	$\overline{TRC}$			$\overline{TBC}$		
			64 MB	512 MB	1024 MB	64 MB	512 MB	1024 MB
1	LRU	LRU	0.9036	0.8455	0.8244	**0.9165**	**0.8252**	**0.7918**
2	LRU	GD	0.8842	0.8244	0.8051	0.9168	0.8281	0.7964
3	GD	LRU	0.9032	0.8452	0.8241	0.9217	0.8334	0.8026
4	GD	GD	**0.8839**	**0.8240**	**0.8048**	0.9220	0.8364	0.8072

However, further increasing the partition size to 100,000 bytes (Table 5.9) makes the $\overline{TRC}$ values worse while making the $\overline{TBC}$ values slightly better. The reason for this behaviour is that as more and more large files are allowed to be cached at the lower level (by changing the partition size from 5,000 to 10,000 bytes), the hit ratios and the byte hit ratios increase, because most of the files in this category are still highly referenced, and the large size means they contribute significantly to the total bytes transferred. Increasing the partition size from 10,000 to 100,000 bytes makes the $\overline{TRC}$ values worse because the larger files allowed to be cached at the lower level will limit the total number of files in the cache, thus reducing hit ratios but increasing byte hit ratios, hence improving $\overline{TBC}$ values.

In general, comparing Table 5.5 with Table 5.3, an algorithm that keeps files less than 5,000 bytes at the lower level and files larger than 5,000 bytes at the upper level gives no overall performance improvement. Similarly, comparing Table 5.9 with Table 5.3, a restriction of 100,000 bytes gives no overall gain. Lastly, comparing Table 5.7 with Table 5.3 shows a consistent reduction in $\overline{TRC}$ values, for the policies considered, with the reduction most pronounced on the smaller cache sizes, and least pronounced when GD-Size is used at the children; however, there is often an increase in $\overline{TBC}$ values.

While the reductions in $\overline{TRC}$ seem modest (e.g., 5% lower for the LRU-LRU case), recall that this is for a linear cost model with constant cost for each level of the hierarchy traversed. If the cache miss penalties were more severe (e.g., $c_1 = 1, c_2 = 10, c_3 = 100$), then the $\overline{TRC}$ values for the LRU-LRU case for a 64 MB cache would change from 0.69 (for plain LRU-LRU) to 0.62 (when the size thresholding approach is used), a difference of 10%. The corresponding $\overline{TBC}$ would change from 0.82 to 0.81, thus making the size thresholding approach a preferable strategy over the plain LRU-LRU combination.

Note that the size restriction in Table 5.7 is 10,000 bytes, which corresponds to the beginning of the tail of the file size distribution in the workload. The implication of this is that maximizing performance in terms of document hits requires keeping files in the body of the file size distribution at the lowest level of the hierarchy. This will give the best performance and the best reduction in the cost associated with request misses. The questionable issue about this policy is how an on-line algorithm can detect the tail of the file size distribution.

One final experiment considers heterogeneous cache sizes. The results in Table 5.11 show the cost-benefit analysis when the two child caches are configured to be half the size of the parent cache, so that the sum of the total cache space at the child level is the same as the cache size at the parent. That is, there is a factor of two savings in the cost of cache space at each child proxy (note that the cache size indicated in Table 5.11 is the size of the parent cache).

Table 5.11: Cost-Benefit Analysis for Configuring Each Child Cache to be Half the Size of the Parent Cache

No	LRU-LRU at both Levels	$\overline{TRC}$			$\overline{TBC}$		
		64 MB	512 MB	1024 MB	64 MB	512 MB	1024 MB
1	Plain LRU-LRU	0.8167	0.7124	0.6695	0.9073	0.7934	0.7403
2	Size-based (10,000 bytes)	0.7820	0.6659	0.6293	0.8978	0.8136	0.7842
3	Sharing Approach	0.8383	0.7486	0.7177	0.9059	0.8020	0.7652

The first entry of Table 5.11 represents the plain LRU-LRU combination. The second entry represents the LRU-LRU combination with the size thresholding approach, where files smaller than 10,000 bytes are kept at the lower level of the hierarchy and larger files at the upper level. The third entry is for keeping shared files at the upper level of the hierarchy, also with the LRU-LRU combination at both levels. All the results in this table are for the partial overlap workload scenario.

By comparing the first entry of Table 5.11 with the first entry of Table 5.3, there seems to be a significant difference, with an increase in the $\overline{TRC}$ and $\overline{TBC}$ values when the lower level caches are each configured with half the cache size of the upper level proxy. Similarly, the size thresholding result in the second entry of Table 5.11 also gave slightly worse performance compared to the first entry of Table 5.7, indicating no significant benefit to using this strategy (though the result is better, in terms of $\overline{TRC}$, than the plain LRU-LRU in the first entry of the table, and even for plain LRU-LRU on the larger caches in Table 5.3), which is consistent with what was earlier observed from previous results.

The third entry of Table 5.11 for the sharing scenario again seems to be the worst overall in this table, and comparing its result with the corresponding partial scenario (first entry) of Table 5.4, the saving in cache cost does not seem to justify the observed differences in $\overline{TRC}$ and $\overline{TBC}$.

5.6 Summary

This chapter explored two possible techniques for improving the performance of a Web caching hierarchy: first, by using different replacement policies at different levels of the hierarchy, and second, by partitioning files based on sizes at different levels of the hierarchy. To investigate these policies, a two-level simulation model of a hierarchy involving three Web proxy servers (a parent at the upper level, and two children at the lower level of the hierarchy) was used.

Simulation results show that combining different replacement policies at different levels of the hierarchy can improve the performance of a caching hierarchy. The best performance was typically provided by the use of LRU or LFU-Aging at the lower level, combined with GD-Size at the upper level.

In terms of file partitioning at different levels of the hierarchy, simulation experiments show that caching strategies that keep smaller files at the lower level and larger files at the upper level can improve the performance of a caching hierarchy (at least, for the cache miss penalties considered). The converse, however, is not true from the simulation results. That is, there is no benefit to caching large files at the lower level and small files at the upper level. Furthermore, while caching small files at the lower level and large files at the upper level is capable of improving performance, the success of this caching strategy depends very much on the choice of the threshold size. Simulation results in this chapter show that a threshold size near the beginning of the heavy tail in the file size distribution seems to give the most benefit.

Chapter 6

Summary, Contributions, and Future Work

6.1 Thesis Summary

The goals of the thesis were: (1) to develop a set of tools for synthetic generation of Web proxy server workloads for evaluating cache replacement policies; (2) to investigate the sensitivity of proxy cache replacement policies to different workload characteristics; and (3) to evaluate a proxy caching hierarchy based on some novel caching techniques suggested in this research.

Chapter 2 presented a brief history of the Internet, followed by a discussion of the components of the World-Wide Web – the Web servers, the Web clients, and the Web proxy servers. A discussion of how these components communicate with one another and the protocol used for communication was also presented. The performance issues for the WWW were also highlighted in a review of prior research, and the approaches being employed to alleviate performance problems. Some of these studies focussed on workload characterization of the Web. Research in this area has demonstrated the benefit of caching Web documents. Several caching techniques were discussed. Single level proxy caching and multi-level proxy caching were among the relevant caching strategies considered.

Chapter 3 describes the design and development of the proxy workload synthesis tool called ProWGen. ProWGen was based on an analytic approach, using mathematical models for selected proxy workload characteristics believed relevant to caching performance evaluation. The selected workload characteristics, the models on which they were based, and the way they were incorporated into the synthesis tool were discussed in this chapter. Techniques used in validating synthetic workloads from the tool set were also presented.

Chapter 4 presented the results of a single-level proxy server simulation in which the sensitivity of three cache replacement policies (LRU, LFU-Aging, and GD-Size) to different workload charecteristics was evaluated. All the experiments in this chapter used synthetic workloads from ProWGen. The results of the experiments showed the relative impacts of the workload characteristics in the traces.

Chapter 5 presented the results for new caching strategies in a multi-level caching hierarchy. This chapter presented the results of using different replacement policies at different levels of a caching hierarchy and keeping disjoint files at different levels. The results showed that these caching strategies can improve the performance of a caching hierarchy.

6.2 Relevance and Contributions

The proxy workload generator tool set (ProWGen) is a valuable tool. Validation of synthetic workloads from ProWGen shows that its outputs can closely mimic empirical workloads, making it suitable for cache performance evaluation. With this property, it eliminates the shortcomings associated with the use of empirical workloads for evaluating cache replacement policies.

The benefits of ProWGen include:

Low storage space requirements ProWGen reduces the required storage space by representing URLs with integers, and avoiding unnecessary information in the generated workloads. For example, a 1 million synthetic workload requires around 20 MB of disk space, with the data in uncompressed ASCII format, while a raw proxy access log would require around 150 MB. Furthermore, there is no need to store synthetic traces permanently, since they can be regenerated at will in a matter of minutes.

Fast workload generation As opposed to configuring real proxy servers to record accesses in a log file over days, months, or years, ProWGen produces synthetic workloads within a short time period. For instance, generating 1 million requests takes 1 to 3 minutes (for static and dynamic stack model of temporal locality, respectively, with a stack size of 1,000) on a 256 MB memory, 400 MHz Pentium II system running NetBSD-1.4.1. In contrast, recording 1 million accesses in a log file in real life proxies could take hours (in a moderately busy Web proxy server) to weeks (in a less busy proxy server).

Tunability ProWGen can be parameterized to produce output which mimics a given empirical workload, both quantitatively and qualitatively.

Flexibility ProWGen represents each workload characteristic with one or more parameters. By specifying different values for these parameters, workloads exhibiting different characteristics can be produced.

Extensibility ProWGen can easily be extended to incorporate new workload characteristics as desired. In other words, a new model can be incorporated into the tool set such that synthetic outputs can reflect such characteristics.

Reproducibility Workloads generated from ProWGen are fully reproducible, given that the parameters have the same settings (and the same random number seeds are used) during each synthesis.

The single server simulation experiments showed that workload characteristics affect different cache replacement policies in different ways. Without an experiment of this nature, it would be very hard to answer questions like: *what performance impact does the Zipf slope have on cache replacement policies?* and *is the impact the same for the tail index?* The experiments on the single proxy server model showed that, while a small difference in Zipf slope could translate to as much as 10-15% difference in cache performance, the same difference in tail index has a significantly lower impact, translating to less than 3% in performance. The experiments on one-timers showed that the concentration of reference property can override the impact of one-timers, such that a replacement policy tends to perform better for workloads with many one-timers.

In the multi-level caching experiments, it was shown that using LRU or LFU-Aging at the lower level of the hierarchy and GD-Size at the upper level provides better performance than employing combinations of LRU and LFU-Aging at both levels. The experiments also showed that better performance in terms of hit ratio can be obtained by keeping small files at the lower level of the hierarchy and large files at the upper level, especially with a threshold size near the beginning of the tail of the file size distribution. Strategies that keep large files at the lower level and small files at the upper level do not offer any performance benefits. Similarly, there is no benefit from keeping shared files at the upper level of the hierarchy.

6.3 Future Research

With the proxy workload generator several other directions are open for research. First, the workload generator can be extended to incorporate file modification into synthetic workloads. This would be useful if workloads from ProWGen are to be used for evaluating cache consistency mechanisms. Secondly, ProWGen can equally generate Web server workloads. Only the parameter settings need be configured with the appropriate values to reflect values measured in Web server workloads. In other words, synthetic workloads from ProWGen can equally be used to evaluate Web server caching strategies.

On the issue of temporal locality, ProWGen uses two different approaches to incorporate this characteristic into synthetic workload, one of which shows a close approximation of this characteristic as observed in empirical traces. Quantifying this characteristic has been a difficult task for researchers, and in fact to date, there is no generally agreed upon methodology to capture or measure this characteristic like others, such as the file size distribution. This difficulty still leaves an open avenue for possible further research.

In the single proxy server experiment, the sensitivity of different replacement policies to workload characteristics shows that some characteristics are more important than others when making replacement decisions. This information could be useful in designing new cache replacement strategies for proxy servers (or Web servers).

In the multi-level caching hierarchy, more research needs to be done. The two level hierarchical simulation model used in this thesis research assumes that there is no communication between the two lower level proxies. Configurations in which the lower level proxies interact can be explored. Other interesting techniques such as push caching and replication techniques [38] are obvious candidate strategies that could be explored, in combination with the various strategies investigated in this work. In addition, the two-level model can be extended to three or more levels with more proxies introduced at the lowest level to ascertain the limits of applicability of some strategies suggested in this work.

Another possible extension that could be made to ProWGen is to incorporate a response time model so that outputs from it can be employed for response time studies of Web proxy servers or Web servers.

Another interesting possibility is to consider the Cache Array Routing Protocol (CARP) [11, 69]. The size partitioning strategies suggested in this research can be investigated in a caching hierarchy with CARP configuration.

References

[1] G. Abdulla, E. Fox, M. Abrams, and S. Williams, "WWW Proxy Traffic Characterization with Application to Caching," Technical Report TR-97-03, Computer Science Department, Virginia Tech., March 1997.

[2] M. Abrams, C. Standridge, G. Abdulla, S. Williams, and E. Fox, "Caching Proxies: Limitations and Potentials," in *Electron. Proceedings 4th World-Wide Web Conference 95*, pp. 119-133, Boston, MA, Dec. 1995.

[3] P. Albitz and C. Liu, DNS and BIND, O'Reilly and Associates, Inc., January 1997.

[4] V. Almeida, M. Cesario, R. Fonseca, W. Meira Jr., and C. Murta, "Analysing the Behavior of a Proxy Server in Light of Regional and Cultural Issues," in *Proceedings of the 3rd International WWW Caching Workshop*, Manchester, England, June 1998.

[5] V. Almeida, A. Bestavros, M. Crovella, and A. Oliveira, "Characterizing Reference Locality in the WWW," in *Proceedings of the 1996 International Conference on Parallel and Distributed Information Systems (PDIS 96)*, pp. 92-103, Dec. 1996.

[6] M. Arlitt and C. Williamson, "Trace-Driven Simulation of Document Caching Strategies for Internet Web Servers," in *Simulation J.*, vol. 68, no. 1, pp. 23-33, Jan. 1997.

[7] M. Arlitt and C. Williamson, "Internet Web Servers: Workload Characterization and Performance Implications," in *IEEE/ACM Trans. on Networking*, vol. 5, no. 5, pp. 631-645, Oct. 1997.

[8] M. Arlitt, *A Performance Study of Internet Web Servers*, M.Sc. Thesis, Department of Computer Science, University of Saskatchewan, 1996.

[9] M. Arlitt, R. Friedrich, and T. Jin, "Performance Evaluation of Web Proxy Cache Replacement Policies," Technical Report HPL-98-97, Hewlett-Packard Laboratories, June 1998.

[10] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and Tai Jin, "Evaluating Content Management Techniques for Web Proxy Caches," in *2nd Workshop on Internet Server Performance*, Atlanta, Georgia, May 1999.

[11] Cache Array Routing Protocol (CARP) and Microsoft Proxy Server 2.0. Available at http://www.msdn.microsoft.com/library/backgrnd/html/carp.html.

[12] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm, "World-Wide Web Caching: The Application Level View of the Internet," in *IEEE Communications*, pp. 170-178, June 1997.

[13] M. Baentsch, L. Barum, G. Molter, S. Rothkugel, and P. Sturm, "Enhancing the Web's Infrastructure: From Caching to Replication," in *IEEE Internet Computing*, vol. 1, no. 2, pp. 18-27, March 1997.

[14] P. Barford and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," in *Proceedings of the 1998 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 151-160, July 1998.

[15] P. Barford, A. Bestavros, A. Bradley, and M. Crovella, "Changes in Web Client Access Patterns: Characteristics and Caching Implications," World Wide Web Journal, Special Issue on Characterization and Performance Evaluation, vol. 2, no. 1, pp. 15-28, January 1999.

[16] P. Barford and M. Crovella, "A Performance Evaluation of Hyper Text Transfer Protocols," in *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 188-197, Atlanta, Georgia, May 1999.

[17] T. Berners-Lee, "WorldWideWeb, the First Web Client". Available at http://www.w3.org/People/Berners-Lee/WorldWideWeb.html.

[18] A. Bestavros, R. Carter, M. Crovella, C. Cunha, A. Heddaya, and S. Mirdad, "Application-Level Document Caching in the Internet," in *Proceedings of the 2nd International Workshop on Services in Distributed and Networked Environments (SDNE 95)*, pp. 166-173, Whistler, BC, June 1995.

[19] R. Bodnarchuk, *Characterizing and Modeling the Workload of a Distributed System File Server*, M.Sc. Thesis, Department of Computer Science, University of Saskatchewan, August 1990.

[20] J. Bolot and P. Hoschka, "Performance Engineering of WWW: Applications to Dimensioning and Cache Design," in *Electron. Proceedings of 5th Inter. World-Wide Web Conference*, Paris, France, May, 1996.

[21] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," in *Proceedings of the IEEE Infocom '99 Conference*, New York, NY, March 1999.

[22] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich, "Web Proxy Caching: The Devil is in the Details," in *Performance Evaluation Review*, vol. 26, no. 1, pp. 11-15, December 1998.

[23] P. Cao and S. Irani, "Cost-aware WWW Proxy Caching Algorithms," in *Proceedings USENIX Symp. on Internet Technologies and Systems*, pp. 193-206, Dec. 1997.

[24] P. Cao, J. Zhang, and K. Beach, "Active Cache: Caching Dynamic Contents (Objects) on the Web," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pp. 373-388, The Lake District, England, Sept. 1998.

[25] P. Cao and C. Liu, "Maitaining Strong Cache Consistency in the World-Wide Web," in *Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, May 1997.

[26] CERN - European Laboratory for Particle Physics, CERN httpd Web Server. Available at http://www.w3.org/Daemons/Status.html.

[27] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz and K. Worrell, "A Hierarchical Internet Object Cache," in *Proceedings of the 1996 USENIX Technical Conference*, pp. 153-163, San Diego, CA, Jan. 1996.

[28] E. Cohen, B. Krishnamurthy, and J. Rexford, "Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters," in *Proceedings ACM SIGCOMM 98*, Vancouver, B.C., pp. 241-253, Sept. 1998.

[29] M. Crovella and P. Barford, "The Network Effects of Pre-fetching," in *Proceedings IEEE INFOCOM*, pp. 1232-1240, April 1998.

[30] M. Crovella and M. Taqqu, "Estimating the Heavy Tail Index from Scaling Properties," in *Methodology and Computing in Applied Probability*, vol. 1, no. 1, 1999.

[31] M. Crovella and A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," in *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, pp. 835-846, Dec. 1997.

[32] C. Cunha, A. Bestavros, and M. Crovella, "Characteristics of WWW Client-based Traces," Technical Report TR-95-010, Department of Computer Science, Boston University, July 1995.

[33] C. Dodge, B. Marx, and H. Pfeiffenberger, "Web Cataloguing Through Cache Exploitation and Steps Toward Consistency Maintenance," in *3rd International World-Wide Web Conference*, April 1995.

[34] B. Duska, D. Marwood, and M. Feeley, "The Measured Access Characteristics of World-Wide Web Client Proxy Caches," in *Proceedings USENIX Symp. on Internet Technologies and Systems*, pp. 23-35, Dec. 1997.

[35] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," in *Proceedings of ACM SIGCOMM 98*, pp. 254-265, Vancouver, BC, Feb. 1998.

[36] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol - HTTP/1.1," HTTP Working Group, Internet Draft, Nov. 1998.

[37] Global Internet Project, "Internet Foundations: Breaking Technology Bottlenecks." Available at http://www.gip.org/gip10.htm

[38] J. Gwertzman and M. Seltzer, "An Analysis of Geographical Push-Caching," in *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, pp. 51-55, Orcas Island, WA, May 1995.

[39] J. Gwertzman and M. Seltzer, "People, Places, and Things: The Next Generation Web," in *Proceedings of 1996 CompCON*, pp. 65-70, Santa Clara, Feb. 1996.

[40] R. Hassett, "Governing the Internet."
Available at http://www.internetlegal.com/governin.htm.

[41] R. Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, John Wiley & Sons Inc., USA, 1994.

[42] Jigsaw – The W3Cs Web Server. Available at http://www.w3.org/Jigsaw/.

[43] B. Krishnamurthy and C. Wills, "Piggyback Server Invalidation for Proxy Cache Coherency," in *Proceedings World-Wide Web Conference*, pp. 185-194, April 1998.

[44] P. Krishnan and B. Sugla, "Utility of Cooperating Web Proxy Caches," in *Proceedings World-Wide Web Conference*, pp. 195-203, April 1998.

[45] T. Kwan, R. McGrath, and D. Reed, "NCSA's World Wide Web Server: Design and Performance," in *IEEE Computer*, vol. 28, no. 11, pp. 68-74, November 1995.

[46] A. Law and W. Kelton, Simulation Modeling and Analysis, Second Edition, McGraw-Hill, Inc., 1991.

[47] C. Liu and P. Cao, "Maintaining Strong Cache Consistency for the World-Wide Web," in *The 17th International Conference on Distributed Computer Systems*, pp. 12-21, May 1997.

[48] A. Mahanti and C. Williamson, "Web Proxy Workload Characterization," Technical Report, Department of Computer Science, University of Saskatchewan, Feb. 1999.

[49] A. Mahanti, *Web Proxy Workload Characterization and Modeling*, M.Sc. Thesis, Department of Computer Science, University of Saskatchewan, Sept. 1999.

[50] A. Mahanti, C. Williamson, and D. Eager, "Traffic Analysis of a Web Proxy Caching Hierarchy," to appear, IEEE Network, vol. 14, no. 3, pp. 16-23, May/June 2000.

[51] A. Mahanti, D. Eager, and C. Williamson, "Temporal Locality and its Impact on Web Proxy Cache Performance," to appear, Performance Evaluation, September 2000.

[52] C. Maltzahn, K. Richardson, and D. Grunwald, "Performance Issues of Enterprise Level Web Proxies," in *Proceedings of SIGMETRICS '97*, pp. 13-23, Seattle, Washington, June 1997.

[53] S. Manley and M. Seltzer, "Web Facts and Fantasy," in *USENIX Symposium on Internet Technology and Systems*, Monterey, California, USA, December 1997.

[54] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential Benefits of Delta-Encoding and Data Compression for HTTP," in *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 181-194, Cannes, France, Sept. 1997.

[55] C. Murta, V. Almeida, and W. Meira Jr., "Analyzing Performance of Partitioned Caches for the WWW," in *Proceedings of the 3rd WWW Caching Workshop*, Manchester, England, June 1998.

[56] D. Neal, "The Harvest Object Cache in New Zealand," in *Proceedings of the 5th International World-Wide Web Conference*, May 1996.

[57] Netscape Proxy Server Deployment Guide. Available at http://www.netscape.com/.

[58] N. Nishikawa, T. Hosokawa, Y. Mori, K. Yoshika, and H. Tsuji, "Memory-based Architecture for Distributed WWW Caching Proxy," in *Proceedings World-Wide Web Conference*, pp. 205-214, April 1998.

[59] D. Povey and J. Harrison, "A Distributed Internet Cache," in *Proceedings of the 20th Australian Computer Science Conference*, Sydney, Australia, Feb. 1997.

[60] F. Quaglia, B. Ciciani, and M. Colajanni, "An Analytical Comparison of Cooperation Protocols for Web Proxy Servers," in *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '98)*, College Park, Maryland USA, Oct. 1999.

[61] References on Zipf's Law. Available at http://linkage.rockefeller.edu/wli/zipf/.

[62] L. Rizzo and L. Vicisano, "Replacement Policies for a Proxy Cache," Technical Report RN/98/13, Department of Computer Science, University College London, 1998.

[63] C. Roadknight, I. Marshall, and D. Vearer, "File Popularity Characterization," in *Proceedings of the 2nd Workshop on Internet Server Performance (WISP 99)*, Atlanta, Georgia, May 1999.

[64] P. Rodriguez, C. Spanner, and E. Biersack, "Web Caching Architectures: Hierarchical and Distributed Caching," in *Proceedings of the 4th International Web Caching Workshop*, pp. 37-48, San Diego, California, April 1999.

[65] A. Rousskov and V. Soloviev, "On Performance of Caching Proxies," in *Proceedings of the ACM SIGMETRICS '98 Conference*, June 1998.

[66] J. Spirn, "Distance String Models for Program Behaviour," *IEEE Computer*, vol. 9, no. 11, pp. 14-20, Nov. 1976.

[67] A. Tanenbaum, Computer Networks, Third Edition, Prentice-Hall, Inc., 1996.

[68] R. Tewari, M. Dahlin, H. Vin, and J. Kay, "Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet", in *Proceedings of the 19th International Conference on Distributed Computing Systems*, Austin, Texas, June 1999.

[69] V. Valloppillil and K. Ross, "Cache Array Routing Protocol v1.1," Internet Draft, Feb. 1998. Available at http://ds1.internic.net/internet-drafts/draft-vinod-carp-v1-03.txt.

[70] D. Wessels and K. Claffy, "ICP and the Squid Web Cache," in *IEEE Journal on Selected Areas in Communication*, vol. 16, no. 3, pp. 345-357, April 1998.

[71] S. Williams, M. Abrams, C. Standridge, G. Abdulla, and E. Fox, "Removal Policies in Network Caches for World-Wide Web Documents," in *Proceedings ACM SIGCOMM 96, Stanford*, CA, pp. 293-305, Aug. 1996.

[72] D. Willick, D. Eager, and R. Bunt, "Disk Cache Replacement Policies for Network File Servers," in *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS)*, Pittsburgh, PA, pp. 2-11, May, 1993.

[73] R. Wooster and M. Abrams, "Proxy Caching That Estimates Page Load Delays," in *6th International World-Wide Web Conference*, Santa Clara, California, USA, April 1997.

[74] P. Yu and E. MacNair, "Performance Study of a Collaborative Method for Hierarchical Caching in Proxy Servers," in *Proceedings World-Wide Web Conference*, pp. 215-224, April 1998.

[75] H. Yu, L. Breslau, and S. Shenker, "A Scalable Web Cache Consistency Architecture," in *Proceedings of the 1999 ACM SIGCOMM Conference*, Cambridge, Massachusetts, USA, Sept. 1999.

[76] L. Zhang, S. Floyd, and V. Jacobson, "Adaptive Web Caching," in *Proceedings of the NLANR Web Cache Worshop*, Boulder, Colorado, USA, June 1997.

[77] What is The Internet. Available at http://www.cas.usf.edu/lis/lis5937/time.html.

[78] WWW Cache Briefing. Available at
http://ppewww.ph.gla.ac.uk/ flavell/cache.html.