Differentiated Web Caching
– A Differentiated Memory Allocation Model on Proxies

by

Dong Zheng

A thesis submitted to the School of Computing

In conformity with the requirements for

the Degree of Master of Science

Queen's University

Kingston, Ontario, Canada

July 2004

Abstract

As the WWW proliferates nowadays in every aspect of modern life, there are more and more demands on the quality of service it provides to the users running various types of applications. Web caching is a proven way to alleviate the service bottleneck and network traffic load, improve request response time and improve web object availability by storing a copy of cacheable requested objects for future use. In recent years, much research has been dedicated to further improving the performance of the Web and providing differentiated services to its users. However, more work needs to be done on providing multiple levels of service in proxy caches, which play a key role in accelerating the Web performance.

This thesis makes two main contributions. First, we studied the workload at a cache server, compared the existing caching algorithms, and proposed webLRU-2, a variant of the LRU-K algorithm that takes into consideration the approximate frequency information, as well as the recency information of the K most recent accesses. Second, we proposed a simple and effective adaptive scheme, with the proposed webLRU-2 as the caching algorithm, to achieve a proportional differential service goal for different classes of requests in the workload.

Through simulations run on synthetic and real-world workloads, we evaluate and verify webLRU-2 against LRU, LRU-K, and LFU. As well, experimental results show that the adaptive model provides more desirable service differentiation than one with fixed-size compartments or fixed weights for classes of requests.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Patrick Martin, for his continuous guidance, support and patience during my graduate study and research. His expertise, insightful advice, and encouraging feedbacks have helped me greatly in my work.

I would also like to thank Wendy Powley, Bouning Niu, Wenhu Tian, Lei Wu, and other members in our database laboratory for their suggestions, co-operations and friendship. My thanks also go to Dali Zhang of Department of Electrical and Computer Engineering, for his help in workload generation and in-depth analysis.

I am grateful to the School of Computing at Queen's University, for providing me such an invaluable opportunity to pursue this degree. Financial support from them, and Communications Information and Technology Ontario (CITO), is much appreciated.

Table of Contents

List of Tables

List of Figures

Glossary of Acronyms

ACME	Adaptive Caching using Multiple Experts
ADSL	Asymmetric Digital Subscriber Line
AS	Autonomous System
BHR	Byte Hit Rate
CDN	Content Delivery Network
DiffServ	Differentiated Services
GDS	Greedy Dual Size
GDSF	Greedy Dual Size Frequency
HR	Hit Rate
HTML	HyperText Markup Language
IntServ	Integrated Services
LFU	Least Frequently Used
LIFO	Last In First Out
LRU	Least Recently Used
LRV	Lowest Relative Value
MF-LRU	Most Frequently – Least Recently Used
MFU	Most Frequently Used
MRU	Most Recently Used
NLANR	National Laboratory for Applied Network Research
PHB	Per-Hop Behavior
QoS	Quality of Service
URL	Uniform Resource Locator
WML	Wireless Markup Language

Chapter 1

Introduction

1.1 Motivation

Quality of Service is becoming an important property for users of the World Wide Web. As one of the most popular applications currently running on the Internet, the World Wide Web is experiencing exponential growth in size, which results in network congestion and server overloading [39]. The Web is used in different ways to access a wide variety of content, including multimedia data, which is real-time and places high value in consistent data access latency (e.g. Voice over IP, Video over IP), and e-business data, which stresses prompt response and minimal data loss. However, with the rapid growth in content and the number of users, the Internet can only provide "best-effort" content delivery to its users.

Web caching, a technique that temporarily stores Web objects (such as Hypertext documents) for later retrieval, has been deployed throughout the Internet and has proven to be an efficient way to alleviate traffic congestion and server overload, and hence improve web content availability and user response time. Proxy caches are strategically deployed at organization network boundaries, at the end of high-latency links, or at major autonomous system (AS) exchanges. Web servers are also replicated to achieve load-balancing. However, the inevitability of cache misses due to the limited capacity of a cache and changing workload determines that caching is only a best-effort network storage service.

To meet the challenges of QoS in a Web caching system we must recognize two main sources of contention. First, capacity on caches is very limited compared with the various and enormous user interest. Second, CPU time for each process becomes crucial in times of overload. A lot of work has been done in recent years to improve the performance of the Web caching systems. Caches have been distributed in various caching system architectures (hierarchical [38], fully distributed [42], or hybrid [36]), with different co-operative protocols (ICP [44], Cache Digest [39], Summary Cache [22], etc) to share the cache among clients more efficiently. Optimization problems like proxy placement [23][28][29] in the caching hierarchy, cache routing/switching [21][40], and cache allocation/replacement algorithms [9][14][31][34][46] have also been studied to achieve higher overall performance of the caching system.

Integrated Services (IntServ) [12] and Differentiated Services (DiffServ) [11] are the two methods from the Internet Engineering Task Force (IETF) to add Quality of Service to IP networks. With the IntServ approach, routers keep track of individual packet flows with specific levels of services. The network grants or rejects a traffic flow based on the availability of resources requested so as to guarantee the levels of services for each flow. This service model, however, suffers from major drawbacks in scalability, cost and deployment. Many researchers therefore resort to DiffServ, which focuses on aggregation of flows of different service classes rather than providing per-flow QoS management.

To add the equivalent function into the Web caching system, J. Chuang et al [20] propose the “stor-serv” framework to provide differential QoS to classified workloads. It tries to setup SLAs (Service Level Agreements) ranging from best-effort caching to object level replication with performance guarantees [19][33] similar to QoS in the

network domain. They also extend their work to propose a pricing scheme [18] for the multiple levels of QoS subscribed by the content publishers.

Many researchers focus their work on optimizing and/or differentiating the utilization of the two key resources in Web caching systems, namely, cache space and CPU time of the content hosts. To provide differentiated levels of service in Web content hosting, J. Almeida et al [7] prioritize processes for classified requests on both the application level and the kernel level by scheduling the order of request processing on a web server or proxy cache, and thus attain multiple levels of QoS with respect to the measure of latency in handling the HTTP request to the Web page. To dynamically allocate cache spaces among classes, Y. Lu et al [32] adopt adaptive control theory to approach the proportional QoS goal, A. Myers et al [33] design their QoS Web caching infrastructure with a publisher-centric approach, and T. Kelly et al attains differential QoS together with the proposed server-weighted LFU algorithm [26], and also evaluate value-sensitive policies with a user-centered approach [27].

The performance of Web proxies throughout the Web caching infrastructure is crucial to any guaranteed QoS that could possibly be provisioned. The former, however, depends largely on how well the running caching algorithms fit the ever-changing access patterns of the Web proxies. Many caching algorithms, viewed as sorting problems that vary in the sorting key used, have been proposed [31][2] in previous research. They originate from classic caching algorithms in file systems or databases, which use basic document attributes as sorting keys, mostly, recency (or the latest access time) in LRU (Least Recently Used), frequency in LFU (Least Frequently Used), and combination of the two in LRU-K. As an Internet workload is different from a workload of a file system or a database in several aspects, some Web caching algorithms take into account different

characteristics, for example, document size in GDS (GreedyDual-Size) and LRU-SIZE, retrieval cost of a missed object from server to proxy in LRV (Least Relative Value [31]).

Unfortunately, it is hard to find an optimal caching algorithm for a wide range of workloads. A successful caching algorithm should have the ability to adapt and at the same time maintain affordable bookkeeping overhead. The ACME model proposed by I. Ari et al [9] combines the efforts of various caching algorithms on a content host, and dynamically chooses the currently most suitable algorithm according to the changing workload. The model is adaptive to a wide range of both workloads and cache topologies. However, as this model is virtually hosting multiple caching algorithms, it suffers from a heavy overhead of bookkeeping information.

LRU-K [34][32] is a self-adaptive algorithm to evolving workloads. This algorithm was proposed for buffer pools in Database Management Systems to achieve higher hit rates, and has proven to be "essentially optimal among all replacement algorithms that are solely based on stochastic information about past references". LRU-K is self-tuning and incurs little bookkeeping overhead. In our work, we adapt the LRU-K algorithm to fit the Web workload better.

There is a view that disk space is currently not the limiting factor in proxy performance as the disk capacity and proxy's capability of serving requests per second grows at a remarkable pace. This, together with the fact that most other algorithms are either unable to fit all workloads or take too much book-keeping overhead, the most popularly used caching algorithm on today's Web caching system is still LRU.

However, it is widely observed that Web cache hit rates are proportional to the logarithm of cache size, and that replacement policies differ substantially in performance [26]. Also, in their research on changes in Web workload, Barford et al [10] compared

the dataset collected from the same computing facility at Boston University in 1995 and 1998 respectively, and found that cache replacement policies for the 1998 dataset benefit from the use of size less than for the 1995 dataset. The overall effectiveness of network caching is also lower in 1998 than in 1995. This means that a mere increase in cache size is no longer a satisfactory solution to the evolving Web traces. What and how we cache can make a big difference. We should also recognize the need in quality of service along with evolution of Web application and request [43], which will rely more on efficient caching. Moreover, the main memory in a proxy, unlike disk space, is still a limited resource that must be managed wisely.

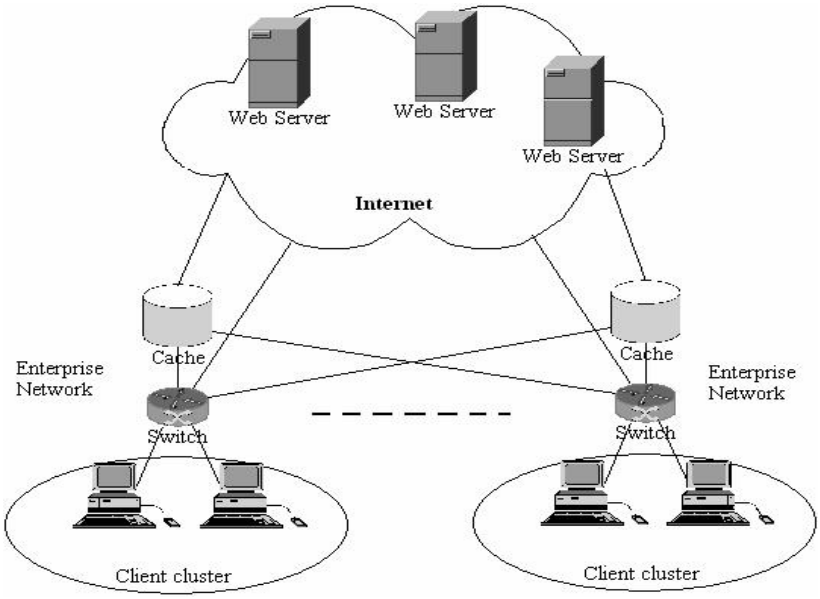

Figure 1-1 A Model of DiffServ Web Caching System

The work in this thesis is a part of the research on optimized and differentiated Web caching by members of the Database Systems Lab and Network Transmission Lab of Queen's University. Previous work includes the following: a transparent distributed load balancing layer 5 switching-based (LB-L5) Web caching architecture by Z. Liang et al [30][2]; the Minimum Response Time (MRT) switching-based Web caching scheme by

Q. Zou et al [48], which reduces the HTTP request response time and balances the workload among the caches; a differentiated switching model by J. Zhou et al [47] to classify, prioritize and redirect user requests; and the Eager-update Page Dynamic Caching (EPDC) scheme by W. Chen et al. to efficiently cache dynamic content [16]. Figure 1-1 shows the DiffServ network model depicted in their work. To complete their work, we focus on providing differentiated service at the cache servers by dynamically allocating cache capacity among classes of objects. We choose an approach based on LRU-K as the caching algorithm, and propose an adaptive model to approach dynamic differentiated services.

1.2 Objective

The main goal of the thesis is to develop an approach to integrate Web caching into the DiffServ environment. We study the workload characteristics at a

cache server, and try to find a suitable and low-cost caching algorithm for differentiated service by first comparing caching algorithms with a synthetic workload that follows the typical Zipf-like distribution, and with real workloads from the cache servers in National Laboratory for Applied Network Research (NLANR [2]). Our solution to differentiated Web caching with cache servers is to combine this caching algorithm with an adaptive model to provide differentiated services to classes of requests.

The objectives of this research are as follows:

- To study and compare existing caching algorithms.
- To propose a caching algorithm that adapts to changes of workload while requiring low overhead of the cache server.
- To design an adaptive model that dynamically reallocates the cache memory for

classes to achieve the desired proportional hit rate goal.

- To develop a trace driven simulation to compare the performance of the caching algorithms and evaluate the adaptive model.

1.3 Thesis organization

The rest of the thesis is organized as follows. Chapter 2 presents the background for research and related work. Chapter 3 presents the webLRU-K algorithm and its performance evaluation compared with LRU, the original LRU-K, In-Cache-LFU and Perfect-LFU. It also describes the simulation model and settings that are used in both this chapter and Chapter 4. Chapter 4 describes the cache server QoS problem, and presents an adaptive model to provide the dynamic allocation solution for proportional QoS, together with the simulation results. Chapter 5 draws conclusions and proposes possible future work.

Chapter 2

Background and Related Work

To fulfill the goal of differentiated QoS on a web cache with optimized resource allocation among classes of web objects, we examine characteristics of proxy workloads in Section 2.1, existing caching algorithms in DBMSs and caching systems in Section 2.2, and issues and solutions for QoS on a proxy server in Section 2.3.

2.1 Workloads at a Web Proxy

Since the performance of a caching policy depends largely on the access pattern of the workload, there has been a lot of research on the Web access workloads [5][8][10][13][24][35][37]. The statistical properties of Web client workloads are typically characterized by high variability due to the diversity of Web caching hierarchies, client groups, and various dynamic events. The following workload properties are among the most observed by researchers, and recognized as "rules of thumb" for workload at a Web proxy.

- "One-timers": 60~70% of the objects will not be accessed again, or will not be the same, and thus useless to cache.
- Object size distribution

Most accessed objects are small. It's a heavy-tailed size distribution. Barford et al [10] observed a mean distinct object size of 7.6 KBytes and a mean download size of around 7.2 Kbytes. They also observed the median sizes of distinct object and object

downloads to be 2.8 Kbytes and 2.4 Kbytes, respectively. Abdulla et al [5] reported that the mean file size ranges between 7K and 27K. Shriver et al [41] found the mean size to be 6.1 K, median size to be around 2 K, and over 90% of the references are for files smaller than 8K. Based on this data, we can assume that:

- The mean object size retrieved by clients is on the order of 10-20 KBytes.
- The median object size is much smaller than the mean, around 2-3 Kbytes.

- Object type distribution

G. Abdulla et al [5] and M. Rabinovich et al [37] drew almost the same conclusion on the percentage of accesses for file types in general Web workloads, namely that images are the most accessed file type followed by HTML. However, G. Abdulla found that, unlike the 90+% of accesses observed on Web servers, images and HTML take up to around 80% of the accesses on Web proxies.

As multimedia objects become more popular, the domination of image access might be changed. Presently, multimedia objects account for very few accesses but a significant fraction of downloaded bytes, varying from 6 to 24 percent.

Another interesting observation is that the number of accesses to dynamically generated objects is rather small, with reported values from under 1 percent to 10~20 percent. This sounds contradictive to the dynamic nature of Web content. The reason is that only the container page of dynamic documents is dynamic, while all embedded objects in the page are static. So accesses to dynamic objects only count as a small percentage of total accesses.

- Reference locality

Web accesses exhibit the properties of temporal and spatial locality. An accessed object exhibits temporal locality if it is likely to be accessed again in the near future. An

accessed object exhibits spatial locality, if objects near it are likely to be accessed in the near future.

- Temporal locality

In their study on Web reference locality, Shudong Jin et al [24] identified and characterized temporal locality from two sources: the long-term popularity and short-term correlations of references.

Population distribution of workloads to Web servers and proxies is observed by many researchers to conform to a Zipf-like distribution. Breslau et al [13] studied the Zipf-like distribution in web requests: $P_N(i) = \frac{\Omega}{i^{\alpha}}$, where $\Omega = \left(\sum_{i=1}^{N} \frac{1}{i^{\alpha}} \right)^{-1}$, with constant α between 0 and 1. They reported α to be in the range from 0.64 to 0.83, suggesting a somewhat more even object popularity distribution at proxies, especially large ones. These findings could be caused by the filtering effect of lower level caches, which service popular pages so that high-level proxies do not see them. Their simulations on Zipf-like workloads show that due to the nature of a Zipf-like distribution, perfect-LFU outperforms LRU, GD_Size and In-Cache-LFU. (An introduction to Zipf-like distributions is presented in Appendix B.1.)

Stack distance [10], or *temporal distance*, is used to observe the temporal locality of the Web accesses. Consider a stack of infinite depth, with elements corresponding to objects. When an object is requested, it is moved to the top of the stack, pushing objects above its old position one position down the stack. The stack distance of an object reference is equal to the position of the object in the stack at the time of the access, or, the distance between two accesses to the same object in the reference sequence. Figure 2-1 illustrates an example of a repeated reference.

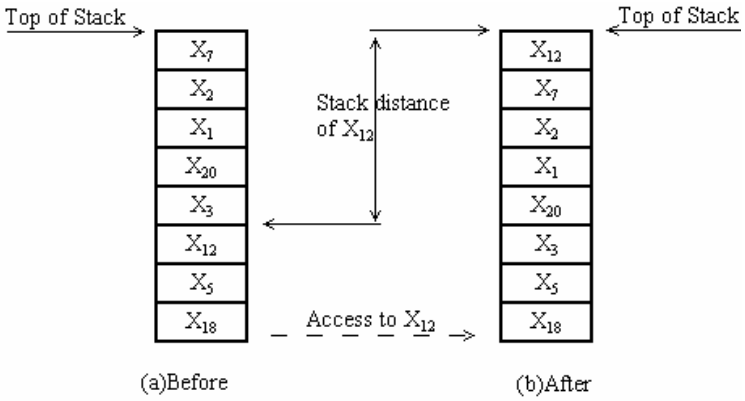

Figure 2-1 An Example of Stack Transformation

The mean or median value of the stack distance can characterize the level of temporal locality. A smaller stack distance indicates that requests to the same object arrive closer to each other, and hence the temporal locality is higher. Thus, the ratio of the average stack distance to the total number of unique objects in the trace can serve as a measure of the trace temporal locality. Barford et al [10] named this ratio *normalized stack distance*, and found the mean and median values of the ratio of their university proxy trace to be 0.2340 and 0.00399, indicating a significant level of temporal locality.

Zipf's popularity distribution of objects by itself captures long-term temporal locality of Web accesses. On the other hand, how the objects are spread throughout the trace is decided mostly by the short-term temporal locality.

- Spatial locality

Web requests are stateless, contrary to requests in an operating system or a DBMS, and spatial locality in the accesses are characterized by a request for an HTML page followed by requests for embedded objects. There is still no direct measure to characterize spatial locality. The set of images within a cacheable HTML page changes slowly over time. So, when a page is requested by a client, it makes

sense for the proxy to anticipate future requests by pre-fetching its historically associated images.

2.2 Caching algorithms

We take an overview of existing caching algorithms (Section 2.2.1) and in particular, for the ability to adapt to changing workloads, analyze a caching scheme of collective intelligence of individual caching algorithms (Section 2.2.2), and a special caching algorithm LRU-K (Section 2.2.3).

2.2.1 Overview of Caching Algorithms

Caching algorithms, or replacement policies have been described and analyzed since processor memory caching was invented. The combination of replacement policy and offered workload determine the efficiency of the cache in optimizing the utilization of system resources. Many caching algorithms have been designed for Web workloads [6][14][17][25][31] [34][37][45][46].

Prime replacement policies take into account critical criteria like recency, frequency and object size. The Least Recently Used (LRU) policy is one of the most popular replacement policies. It evicts the object that has not been accessed for the longest time. This policy caters to the temporal locality of reference in the workload and performs well when most recently referenced objects are most likely to be referenced again in the near future. The LRU policy can be easily implemented using a linked list ordered by last access time. Addition and removal of objects from the list is done in $O(1)$ (constant) time by accessing only the head and tail of the list, respectively. Updates of a referenced object can be accomplished in constant time through a pointer into a list node in object metadata.

Another common policy is Least Frequently Used (LFU). The policy keeps a reference counter of each object, and evicts the objects with the lowest reference count when it makes replacement decisions. The LFU policy can also be implemented with a single linked list, in a sequence of object frequency, and the addition and removal of objects from the list is done in $O(\log n)$ time. However, LFU suffers from cache pollution when a popular object becomes unpopular: it keeps the object in the cache for a long time, preventing newer popular objects from replacing it.

Log(Size)-LRU [6] evicts the document who has the largest log(size) and is the least recently used document among all documents with the same log(size).

A variant of the LFU policy, the LFU-Aging policy considers both the access frequency of an object and the age of the object in cache (the recency of last access). The aging policy addresses the cache pollution that occurs due to turnover in the popular object set. LFU-DA (LFU with Dynamic Aging) is based on LFU-Aging, but instead of using a tunable constant, its dynamic aging mechanism does not require parameterization. Hyper-G [45] is another refinement of LFU with last access time and size considerations;

The GreedyDual-Size (GDS [14]) policy takes into account size and a cost function for retrieving objects. It replaces the smallest key calculated by the function

$$K(p) = C(p)/S(p) + L,$$

where $C(p)$ is the retrieval cost, $S(p)$ is object size of object *p*, and L is the running age factor. GDS-F [17] is a refinement of the GDS policy to account for frequency as a replacement policy parameter. Its key function is

$$K(p) = C(p) * F(p)/S(p) + L,$$

where frequency $F(p)$ is added into the formula of the GDS. Another variation, GDS-P

[24][25], is designed to take into consideration long-term access frequencies in its cache replacement strategy. It uses a similar key function as GDS-F, except that the frequency F is calculated with a decay function to de-emphasize the significance of past accesses. On the *(i+1)-th* reference to object *p*, its frequency is iterated as:

$$F_{i+1}(p) = F_i(p) * 2^{-t/T} + 1,$$

where *t* is the elapsed time since the last reference and *T* is a time constant that controls the rate of decay. The three algorithms GDS, GDS-F and GDS-P, each maintaining a priority queue with key *K(p)*, have the same overhead of $O(\log n)$ handling a hit or a replacement.

The Lowest Relative Value (LRV [31]) policy replaces the object with the lowest relative value that is calculated using access time, frequency and size information.

Of the many Web cache replacement policies we described above, none attempt to provide variable levels of service to clients and servers, which is a very practical and urgent need of both content publishers and subscribers. Disk space in shared caches is a strategically-placed resource, and it is reasonable to suppose that QoS differentiation might be achieved by devoting it to those who value it most. So priority of different object classes can also be one of the factors in the caching policies' key calculation, as in server-weighted LFU [26] proposed by T. Kelly et al, which calculates in its key function the object's contribution to aggregate user value per unit of cache space:

$$\frac{W_u * size(u) * N_u}{size(u)} = W_u * N_u,$$

where W_u is the weight of object *u*, supplied by the server indicating how much the server values cache hits on object *u*, and N_u is the number of references to object *u*

since it entered the cache.

To sum up, different static policies may be more successful with different workload patterns, therefore choosing a single static policy can result in different losses with different workloads. A possible solution is to design a policy that better suits the Web workloads. Another one is an automated scheme that will be able to select the current best static policies.

2.2.2 Collective Intelligence: ACME (Adaptive Caching using Multiple Experts)

Ismail Ari et al [9] proposed an adaptive caching scheme that combines the efforts of multiple experts, or caching policies. Machine learning algorithms are used to rate and select the best policies or mixtures of policies for current workload via weight updates based on their recent success.

Criteria	**Algorithm**
-	Random, FIFO, LIFO
Recency	LRU, MRU, GDS, GDSF, LRV, MF-LRU
Frequency	LFU, MFU, GDSF, LRV, MF-LRU
Size	SIZE, GDS, GDSF, LRV
Retrieval cost	GDS, GDSF, LRV
ID	Hash, Bloom-Filter
Hop-count	-
QoS priority	Stor-serv

Table 2-1 Caching Policies and Their Criteria [9]

Static caching policies are registered in ACME's policy pool, and descriptions of their criteria are registered in the criteria pool. As shown in Table 2-1, one policy may have more than one criterion, and more than one policy may share a common criterion, for example, the GDSF policy shares a common criterion "Time" with the LRU policy, and shares a common criterion "Frequency" with the LFU policy.

ACME records and updates the information according to descriptions of the registered criteria on a per-object basis. Each static caching policy maintains a virtual cache of ordered object headers as if it owns the entire physical cache, and reports a hit or a miss on each request. By rewarding or punishing individual policies according to their prediction accuracy, the machine learning algorithm ensures the best ones for the current workload receive the highest weight and therefore have the largest effect on the cache management decisions.

Let vector x_t be the predictions made by the caching algorithms in the policy pool. The machine learning algorithm predicts with a weighted average y' of the experts' predictions: $y' = x_t \cdot w_t$. The loss, compared with the actual outcomes y, may be defined as $Loss(y'_t, y_t) = (y'_t - y_t)^2$, and is used to update the weights:

$$w_{t+1,i} = w_{t,i} \frac{e^{-\eta * Loss_i}}{\sum_{i=1}^{N} w_{t+1,i}} \quad \text{for i = 1, …, N}$$

where η is the learning rate and the denominator provides normalization of weights. Weights can be initialized equally as $w_{0,i} = 1/N$, where N is the number of experts, or past experience on the success of policies may be used to bias the initialization vector.

ACME combines the weighted policies to make adaptive decisions for current workload, favoring objects that are rated highly by many policies rather than simply favoring objects with high weight by a single, possibly complex policy. It is adaptive to a wide range of both workloads and cache topologies. The authors provide preliminary results of simulations running with two different cache policies in the ACME policy pool. The result justifies the adaptive scheme with the ability to look at recent success and

switch to currently successful policy and provide continuous high performance. However, in more general cases where the model is virtually hosting multiple caching algorithms, it suffers from heavy bookkeeping overhead, and may not be practical to most proxy servers.

2.2.3 The LRU-K algorithm

LRU-K [34] is a self-reliant page-replacement algorithm derived from classical Least Recently Used (LRU). It was proposed for managing buffer areas in database management systems. It incorporates both recency and frequency information when making replacement decisions. Since the LRU buffering algorithm drops the page from the buffer that has not been accessed for the longest time when a new buffer is needed, it limits itself to only the time of the last reference. Specifically, LRU does not discriminate well between frequently and infrequently referenced pages until the system has wasted a lot of resources keeping infrequently referenced pages in the buffer for an extended period. It was proven that LRU-K is essentially optimal among all replacement algorithms that are solely based on stochastic information about past references ([34]).

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

Assume we are given a set of disk pages, denoted by the set of positive integers $N = \{1,2,\cdots,n\}$, and that the database system under study makes a succession of

references to these pages specified by the reference string: $r_1, r_2, \cdots, r_i \cdots$, where $r_t = p (p \in N)$ means that r_t is a reference to disk page p. Clearly, each disk page p has an expected reference inter-arrival time, which is the time between successive occurrences of p in the reference string. The system then attempts to keep in memory buffers only those pages with the shortest access inter-arrival times, or equivalently the greatest probability of reference. The LRU-1 algorithm can be thought of as taking such a statistical approach, keeping in memory only those pages that seem to have the shortest inter-arrival time.

The LRU-K Algorithm specifies a page replacement policy when a buffer slot is needed for a new page from disk, the algorithm specifies that the page p to be dropped is the one whose Backward K-distance, $b_t(p, K)$, is the maximum of all pages in buffer. Given a reference string known up to time t, $r_1, r_2, \cdots, r_t$, the Backward K-distance $b_t(p, K)$ is defined as the distance backward to the K^{th} most recent reference to the page p:

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

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

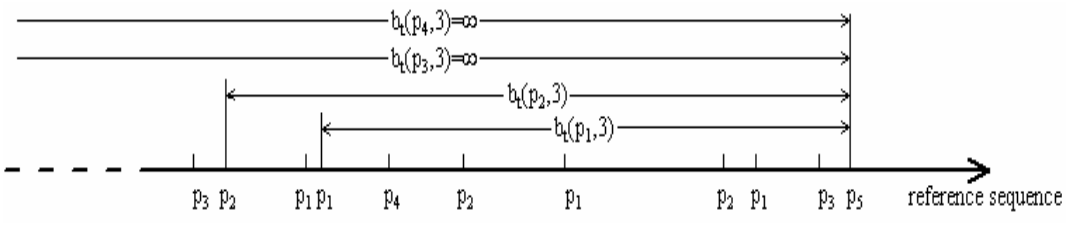

Figure 2-2 A Simplified Example of Backward K-distance (K=3)

Figure 2-2 shows a simplified example of LRU-3 for a sequence of accesses to pages $p_1, p_2, \cdots, p_n$, When a request for an absent page p_5 comes and the buffer is full, a victim is chosen based on the backward K-distance from the point of the new access. In the case of this example, both p_3 and p_4 have the backward K-distance of infinity, so a subsidiary policy is needed to break the tie.

LRU-K takes into account more of the access history for each page to better discriminate pages that should be kept in the cache, based on the assumption that the reference string is a sequence of random accesses with Zipfian distribution, and each disk page p has a well defined probability, β_p, to be the next page referenced by the system. Changing access patterns may alter these page reference probabilities, but the probabilities β_p have relatively long periods of stable values, and are assumed to be independent of t. From the assumption and definition of LRU-K, it's conjecturable that for $K > 2$, the LRU-K algorithm would provide somewhat improved performance over LRU-2 for stable patterns of access, but is less responsive to changes in access patterns, an important consideration for some applications.

Two important features, peculiar to the cases where $K \geq 2$, require careful consideration to ensure proper caching behavior. The first one, known as Early Page Replacement, arises in situations where a page recently read into memory buffer does not merit retention in the buffer by standard LRU-K criteria, for example, because the page

has a $b_t(p,K)$ value of infinity. We surely want to drop this page from the cache relatively quickly, to save memory resources for more deserving disk pages. However we need to allow for the fact that a page that is not generally popular may still experience a burst of correlated references shortly after being referenced for the first time. The LRU-K Algorithm addresses this issue with a Correlation Time-Out parameter, so that the system does not drop a page immediately after its first reference, but keeps the page around for a Correlated Reference Period (set by the Correlation Time-Out parameter) to eliminate the likelihood of a dependent follow-up reference.

The second feature is the fact that there is a need to retain history information of references for objects that are not currently present in the cache. This is a departure from current page replacement algorithms, and is referred to as the Page Reference Retained Information Problem. The LRU-K Algorithm addresses this problem with a *Retained Information Period* parameter so that the system maintains history information about any object for this period after its most recent access. It eliminates the possibility of repeatedly referencing a page soon after it is evicted and having no record of prior references and drops it because each time its backward K-distance is estimated as infinity. A pseudo-code description of the LRU-2 algorithm is given in Appendix A.

2.3 Differentiated Services on a Web Proxy

This section goes over object classification and resource management on a Web proxy. It then discusses two frameworks aimed at DiffServ in Web caching systems. We compare the stor-Serv framework [20] proposed by J. Chuang et al to implement the entire DiffServ architecture in the storage domain with a control-theoretical approach adopted from physical process control in the adaptive control framework by Y. Lu et al

[32] that tried to accomplish differentiated caching services on proxy caches.

2.3.1 Classification of the Web Objects

There are several immediate applications of caches with a performance differentiation mechanism [32]. The first one is to improve client-perceived performance by caching the most “important” content type. It has been observed that different types of Web content contribute differently to the user’s perception of network performance. For example, user-perceived performance depends more on the download latency of HTML pages than on the download latency of their embedded objects (mostly images). Thus prioritizing HTML text in a cache improves the experience of the clients for the same network load conditions and overall cache hit ratio.

The second application is differentiation between standard web content and the emerging wireless content. Web appliances such as PDAs and web-enabled wireless phones will soon make traditional PCs and Workstations a minority among information clients. Wireless appliances require new content types for which a new language, the Wireless Markup Language (WML [3]) was designed. Proxy caching will have a lower impact on user-perceived performance of wireless clients, because saving backbone roundtrips does not avoid the wireless bottleneck. Future proxies that support performance differentiation can get away with a lower hit rate on WML traffic to give more resources to faster clients (that are more susceptible to network delays) thus optimizing aggregate resource usage. This method can be easily implemented based on the IP address of the clients sending the request. For example, an ISP might have separate IP blocks allocated to low bandwidth wireless clients requesting WML documents and higher bandwidth ADSL clients requesting regular HTML content.

Another method is to classify content by the identity of the requested URL, so that an ISP can have agreements with the content providers on the level of QoS they require, for a negotiated price.

J. Zhou et al [47] classify objects based on object type in their research. Four classes are defined: Premium, Assured, Best-Effort (BE) and Others. Premium Class covers real-time application, including voice and/or video streaming, video conference; Assured Class includes mission critical data such as e-commerce data; BE class objects are non-critical data; Non-cacheable objects belong to Others. With Differentiated QoS, they provide Premium Class with guaranteed low delay, Assured Class with controlled delay (a given percentage of this class will be treated within the delay bound), BE Class with neither minimum rate nor loss guarantees.

2.3.2 A DiffServ Architecture for Web Caching: Stor-Serv

Differentiated Services (DiffServ [11]), together with Integrated Services (IntServ [12]), are the two architectures in the Integrated Services Architecture intended to support QoS offerings in the Internet and private intranets. Compared to IntServ, DiffServ is a simpler, easy-to-implement, low-overhead tool to support a range of network services that are differentiated on the basis of performance.

Several key characteristics of DiffServ contribute to its efficiency and ease of deployment:

- QoS classification information is kept in bit-field DSCP (DiffServ Codepoint), which uses the existing IPv4 Type-of-Service (ToS) field in every IP packet header.

- A service level agreement (SLA) established between the service provider and the customer prior to the use of DiffServ, so that there is no need to incorporate DS mechanisms in applications.
- DiffServ provides a built-in aggregation mechanism. All traffic of the same class (with the same codepoint) receives the same treatment at each router, or per-hop behavior (PHB). This supports scaling to larger networks and traffic loads.
- No per-flow information in the core of the network, but rather, per-class queues, is maintained in routers. Different treatments are performed among different queues.

Codepoints should be looked at as an index of a table of packet forwarding treatments at each router. Routers in a DiffServ (DS) domain are either boundary nodes or interior nodes. Typically, the interior nodes implement simple mechanisms for handling packets based on their DS codepoint values. This includes a queuing discipline to give preferential treatment depending on codepoint value, and packet-dropping rules to dictate specifications of the forwarding treatment provided at a router.

PHB must be available at all routers, and typically PHB is the only part of DS implemented in interior routers. The boundary nodes include PHB mechanisms and also more sophisticated traffic conditioning mechanisms to provide the desired service. Thus, interior routers have minimal functionality and minimal overhead in providing the DS service, while most of the complexity is in the boundary nodes.

Figure 2-3 illustrates the elements in the traffic conditioning function:

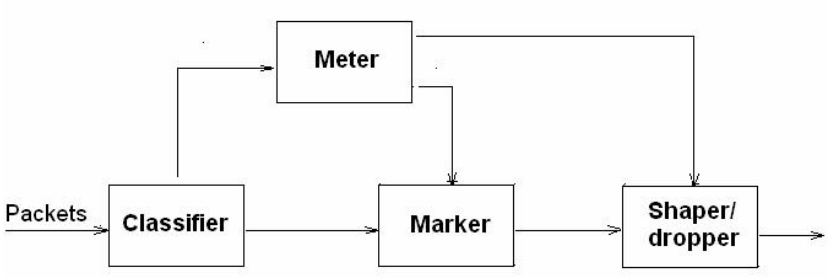

Figure 2-3 DS Traffic Conditioner

Classifier: Separates submitted packets into different classes.

Meter: Measures submitted traffic for conformance to a profile to determine whether a given packet stream class is within, or exceeds, the service level guaranteed for that class

Marker: Polices traffic by re-marking packets with a different codepoint as needed. This may be needed when packets exceed their profile, to remark for best effort handling; or at the boundary between two DS domains that give different priority value to the same traffic class, to enter the second domain.

Shaper: Polices traffic by delaying packets (for example, by means of token bucket) as necessary so that the packet stream in a given class does not exceed the traffic rate specified in the profile for that class.

Dropper: Drops packets when the rate of packets of a given class exceeds that specified in the profile for that class.

John Chuang et al proposed the Stor-serv framework [20], which brings the concept of quality-of-service from the network transmission domain to the network storage domain. With the two dimensions of storage QoS, namely the object placement policy and object replacement policy, four classes are defined, as shown in Table 2-2.

Service Class	Custom Placement	Custom Replacement
Best Effort Caching	No	No
Differential Caching	No	Yes
Push Caching	Yes	No
Object Level Replication	Yes	Yes

Table 2-2 Classification of the stor-serv Framework[20]

In best-effort caching, object placement is demand-driven, and object replacement follows the local policy of the cache node. In differential caching, object placement remains demand-driven, but the publisher can specify the replacement policy for the objects in place of the local one. Push caching is the opposite of differential caching, in that the publisher specifies the cache nodes to which objects are pushed, but once the objects arrive at the caches, they are subject to the local replacement policies of each cache. Finally, object level replication allows the publisher to specify cache nodes, as well as replacement policies for its objects.

Figure 2-4 illustrates the process of turning a publisher's performance requirements into performance realization within the stor-serv framework.

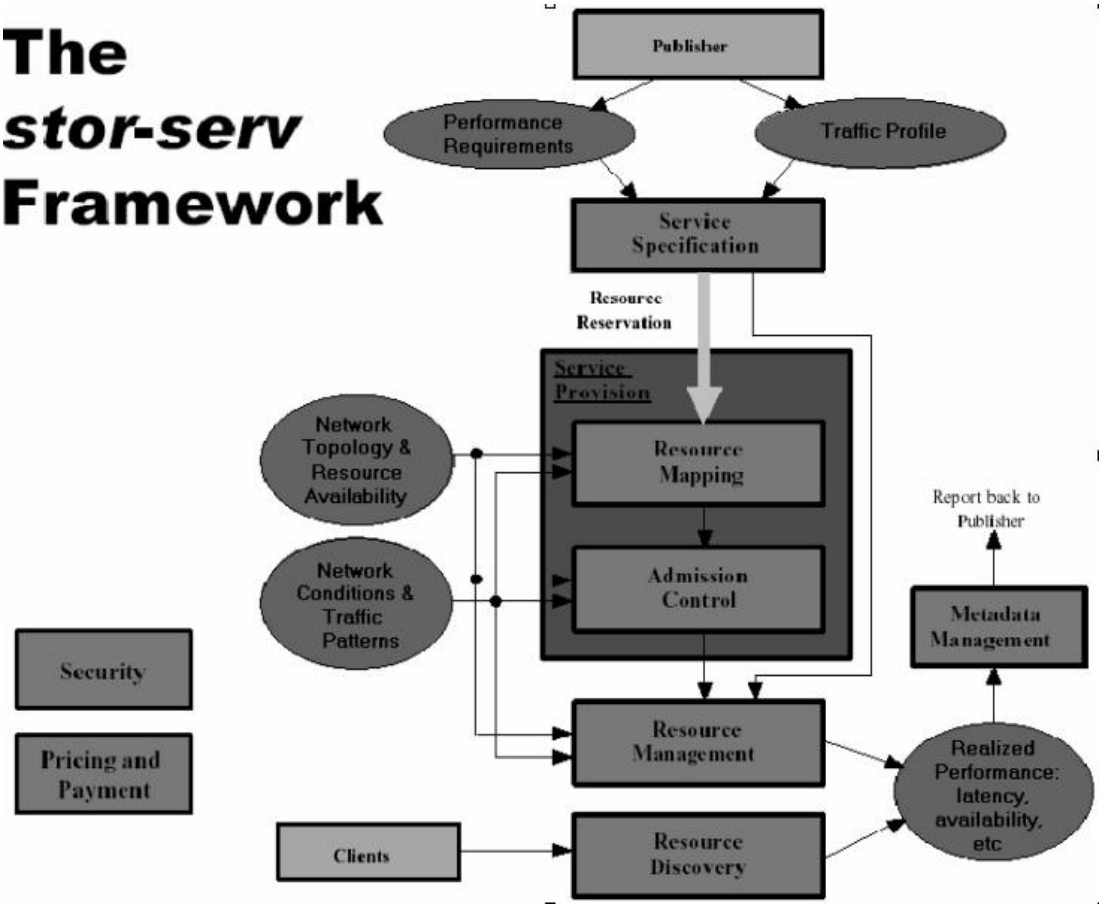

Figure 2-4 The stor-serv Framework: From Performance Requirements to Performance Realization ([20])

The key components of the framework include service specification, service provision, resource management, resource discovery, metadata management, security and economics. When a publisher requests object level replication of its data, the service specification is presented in forms of (1) its QoS performance requirements and (2) its traffic profile, including object size and even its information access pattern.

Service provision includes three steps, namely resource reservation, resource mapping, and admission control. In resource reservation, the publisher forwards the service specification to the network storage service provider. In resource mapping, the provider translates the high-level QoS requirements into low-level physical requirements.

Then each of the individual nodes is consulted to see if it is able to admit the service request. The service could be established only if all of the nodes accept the request.

Resource managers in individual stor-serv nodes perform real-time resource management, either to maximize local and global resource utilization, or to maintain service commitments when faced with changes in network conditions (e.g., congested or failed links, nodes). Differential caching replacement policies are also realized by the resource managers. In order to direct the clients to the closest copy of each requested object, the provider needs to fulfill the function of resource discovery by dealing with issues such as naming, name resolution and distance estimation

Metadata management component measures and logs requests and reports the aggregation of the logs back to the publisher for the purpose of accounting, billing and audit, and verification that the performance realization by the system is consistent with the requirement originally specified by the publisher.

The two additional components, security and economics, ensure the security of data objects at individual nodes and efficient use of the storage service infrastructure, respectively.

2.3.3 A Control-Theoretical Approach

Control theory was originally developed for physical process control. Y. Lu et al [32] are among the first to adopt it into use in the context of software performance control. It allows us to build self-tuning performance regulators, which will automatically build a system model and maintain the performance of the system at a level that satisfies QoS requirements whenever feasible.

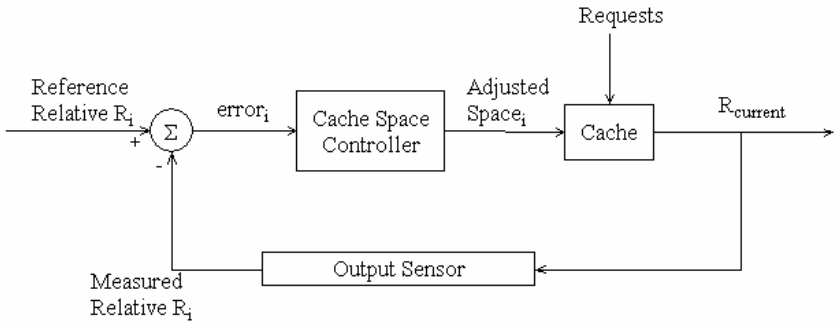

Figure 2-5 The Hit Rate Control Loop

In Figure 2-5, a service differentiation policy dynamically determines the desired performance level for the traffic class under consideration. In this case, hit ratio is the performance metric. Let there be n content classes in the system, and the measured average hit ratio of content class i be H_i. The differentiation policy specifies that the hit ratio of different classes should be related by the expression: $H_n = C_1 : C_2 : \cdots : C_n$, where C_i is a proportion constant or a weight for class i. The relative hit ratio, R_i, of content class i is defined to be $R_i = H_i / (H_1 + H_2 + \cdots + H_n)$. It determines how the class is performing relative to other classes. The desired performance of class i should be $R_{i_{desired}} = C_i / (C_1 + C_2 + \cdots + C_n)$. For the performance error $e_i = R_{i_{desired}} - R_i$ of class i, an appealing property of this model is that the aggregate performance error of the system is always zero, because:

$$\sum_{1 \le i \le n} e_i = \sum_{1 \le i \le n} (R_{i_{desired}} - R_i) = \frac{\sum_{1 \le i \le n} C_i}{C_1 + C_2 + \cdots + C_n} - \frac{\sum_{1 \le i \le n} H_i}{H_1 + H_2 + \cdots + H_n} = 1 - 1 = 0$$

With this property, the authors developed resource allocation algorithms in which resources of each class are heuristically adjusted independent of adjustments of other classes, yet the total amount of allocated resources remains equal to the total size of the cache.

However, the control approach according to feedback information is somewhat like chasing a moving target. The feedback it relies on may be obsolete and the adjustment made could be misleading. There is a tendency to shorten the interval between feedbacks, which may cause the loss of accuracy of statistical feedback within the interval. This is recognized as a trade-off between promptness and accuracy.

Chapter 3

LRU-K and webLRU-K

The LRU-K algorithm, which is originally used in DBMSs, has been proved to be "essentially optimal among all replacement algorithms that are solely based on stochastic information about past references" [34] by adapting in real time to changing patterns of access to a database. LRU-K is characterized by short-term popularity of objects, or short-term access patterns in the workload. On the contrary, the LFU algorithms base their replacement decision solely on the frequency information of history accesses, or long-term access pattern of the workload. To adapt the algorithm into the Web caching environment, on a web proxy particularly, alterations are necessary favoring access patterns at the web proxies.

We first analyze the differences between workloads to a database and those to a web proxy in Section 3.1, then in Section 3.2, we analyze the LFU algorithms and LRU-K algorithms and try to combine the strengths of these two approaches in the proposed webLRU-2 algorithm to efficiently cache web objects on proxies. We describe the simulation model and experiment settings in Section 3.3. Simulations (Section 3.4) are run with both synthetic Zipf-like workloads and real-world workloads to set proper parameter values for the algorithm and to compare the algorithm with the LFU and LRU-K algorithms.

3.1 Workloads of DBMSs and Web Proxies

Although the caching problem was first recognized and addressed in operating systems and DBMSs, Web caching is demanding a different treatment because of the different properties of Web traffic. We compare Web Proxy workloads with those in DBMSs to analyze the success of LRU-K and its potential in Web caching. The main differences relevant to caching algorithms include the following:

- Reference locality.

An important factor in DB workloads is that they show a strong correlative pattern. For example, before any data page is buffered, its index page usually has to be buffered first. The reference to the pages is: $I1, R1, I2, R2, I3, R3, \cdots$. As a result, DB workloads differentiate abruptly between index and tuple pages. In the example set by O'Neil et al [34], the index tree leaf pages are referenced with a probability of 0.005, while data pages have a reference probability of only 0.00005. Thus, there exists in DB workloads a significant short-term reference locality between index tree leaf pages and all data pages indexed by them. LRU-K meets this property of DB workloads by combining recency and short-term frequency. LRU-2 alone is very efficient to distinguish these two groups of pages, at the same time of being the most adaptive one in the LRU-K ($K \geqslant 2$) family.

On the other hand, Web proxy workloads, as discussed in Section 2.1, are stateless and the inherent spatial locality is mainly caused by HTML page accesses leading to embedded object accesses. Long-term popularity becomes a major characteristic of Web request streams. In our experiment with synthetic workloads (Section 3.4.1) and on real proxy workloads (Section 3.4.2), the results show that perfect-LFU generally has the best performance over the other algorithms, due to its full record of frequency information.

Also, the access stream seen by a database buffer comes from a limited group of users with similar access interest to certain pages within the database that has a predefined purpose, and thus exhibit strong access patterns. The access stream seen by a proxy cache is a union of access streams coming from the users (or lower level proxies) sitting behind the proxy. The number of users can be as high as several hundreds or thousands. These users could have vastly different access interests to the contents that reside on Web servers throughout the Internet.

- Granularity of Caching

The granularity of caching is different. In DBMSs, index pages and data pages are of a fixed size, while the HTTP protocol supports whole file transfers only, and Web objects are cached as individual files with various sizes. Statistics show that Web objects are of vastly different sizes, though 90% of the references are for objects smaller than 8K. Although size of an object does not decide its probability of reference, it is reasonable to favor smaller objects over larger objects. They are more popular in general and are less likely to harm the overall hit rate and byte hit rate even in case they turn out to be unpopular in the future. In practice, many Web caching algorithms include object size in their key calculation in favor of small Web objects. For example, GreedyDual-Size [14], LRV [31] and Hybrid [46] calculate their key to be inversely proportional to the object size. Some algorithms such as log(size)+LRU [6] categorize objects into log(Size) groups in comparing caching keys.

- Size of Cache

Unlike DBMSs that cache requested pages in main memory, cache proxies mainly use secondary memory to store the enormous amount of requested objects. Since most of the Web objects are small, the number of objects in a Web cache can vary and be huge,

adding much to the bookkeeping overhead of the caching algorithms. In fact, this has become a limitation on Web caching algorithms, and one of the reasons some algorithms with good theoretical performance, such as Perfect-LFU, LRU-K(K>>2) etc., are not practical. LRU is still the most widely used on the Internet.

- Types of Objects

As discussed in Section 2.1, multimedia objects are taking up a bigger and bigger portion of the evolving Web traffic, which is an important factor to consider when providing differential QoS.

Differences in request streams to Web proxies from those to DBMSs cause LRU-K to have worse performance in Web caching. This can be observed in comparison between simulation results of O'Neil et al's work [34] and ours. The former shows that on both random access with Zipfian distribution and OLTP traces to a DBMS disk page pool, LRU-2 outperforms LRU such that, on average, LRU must use twice as many buffer pages to achieve the same hit rate as LRU-2. In our experiment with Zipf-like workloads, as shown in Figure 3-7, for workloads with Zipf skews varying from 0.5 to 0.9, LRU needs only roughly 1.3 times as much cache space to achieve the same hit rate as LRU-2, and in Figure 3-9, for workloads with Zipf skew 0.7 (the general skew observed in Web request streams), LRU needs about twice as much cache space to achieve the same hit rate as LRU-2. However, on real Web traces, Figure 3-10 shows a weakened advantage of LRU-2 over LRU.

Also, simulation results on DBMS buffer cache hit ratios using an OLTP trace in [34] shows that LRU-2 outperforms LFU. However, on Web request streams,

Perfect-LFU meets the long-term popularity and outperforms LRU-K, as shown in Figure 3-10 with K=1, 2, 3, 4, 8, 16.

At this point, we can presume that:

- Short-term correlation of references is the dominant factor in DBMS workloads, but in Web request streams, it is out-weighted by long-term popularity of objects.
- LRU-K captures short-term locality very well, but turns out to be inefficient in trying to predict popularity of objects in the future by comparing their popularity in the short-term history.
- LRU-2 turns out to be the most adaptive in the LRU-K (K>1) family. Although it may not be the best one on synthetic Zipf-like workloads, it almost always outperforms the others on both real DBMS and Web workloads where the access pattern keeps changing and short-term locality exists. LRU-K, with bigger K, is less responsive to pattern changes in the sense that it can sense the changes only after K accesses to new popular objects.
- LFU gets better performance on Web request streams because it meets the long-term locality.

3.2 LFU, LRU-K and webLRU-K

In this section we propose a new caching algorithm that extends the LRU-K algorithm to better suit Web caching. The idea is to exploit the adaptability of LRU-K and incorporate more frequency information as in LFU, in order to meet both long-term and short-term reference locality in Web request streams. A balanced stress between long-term and short-term locality is crucial to performance of the new algorithm, and a

low bookkeeping overhead and computational complexity is part of our goal in the design. We name the proposed algorithm webLRU-K.

In Section 3.2.1 and 3.2.2, we analyze LFU and LRU-K, respectively, and how they meet the two aspects of reference localities in workloads. We describe webLRU-2 in Section 3.2.3.

3.2.1 LFU and Long-term Popularity

There are two versions of LFU, namely, In-Cache-LFU and Perfect-LFU. Perfect-LFU keeps page counters over the whole process, even if a page is evicted from the cache, while In-Cache-LFU removes the page counter together with the evicted page. Clearly, Perfect-LFU incurs more overhead than In-Cache-LFU and is less practical.

Perfect-LFU and In-Cache-LFU capture long-term popularity of requested objects in cache traces. In L. Breslau et al's simulations [13], Perfect-LFU has a preferable performance on both Zipf-like synthetic workloads and real workloads. However, the hit ratio of In-Cache-LFU is substantially worse. This can also be observed in our simulations in Section 3.4.1 and Section 3.4.2 on both workloads.

Compared with other caching algorithms, LFU involves low computational complexity, and In-Cache-LFU has a low overhead. The inherent performance drawback of Perfect-LFU is that it ignores short-term correlation of reference and does not adapt to evolving access patterns.

3.2.2 LRU-K and Short-term Correlation of Reference

As described in Section 2.2.3, LRU-K (K>1) is an algorithm that tries to incorporate frequency and recency in its caching decisions. When K is small, LRU-K emphasizes short-term characteristics of the workload, and is observed to be more adaptive to pattern

changes in the workload. LRU is an extreme case of LRU-K, with K=1. It replaces objects that are least recently accessed, but totally ignores the long-term popularity information of the objects. LRU-2 is better than LRU because it takes into account frequency of the objects, and as well keeps the prompt adaptability of LRU-1 to some degree. When K increases, LRU-K increases its emphasis on long-term characteristics of the workload, and fits stable workloads better than LRU or LRU-2, especially with a cache size big enough to keep long-term popular objects.

LRU-K algorithm has different results between synthetic and real workloads with opposite trends in performance with respect to the value of K. In the synthetic workload experiments, as shown in Figure 3-9, LRU-8 has a better performance than LRU-4, followed by LRU-3, and then LRU-2. LRU has the lowest hit rate among them. However in the real workload experiments, where short-term correlation of references takes place, the bigger the K value, the lower the hit rate in general, as shown in Figure 3-10. This is caused by the differences between synthetic workloads and real traces. The former are distributions of objects randomized with Zipf-like probability, and could be considered stable, while access patterns of the latter are variable, and short-term reference locality is present. Let $k_2 > k_1 > 1$, the trend of LRU-k_1 outperforming LRU-k_2 implies a major effect of relative short-term correlation over long-term frequency on the algorithms, and requires that the caching algorithms are more adaptive to workload changes. An algorithm like LRU-16 does not seem to hold enough frequency information to compare with Perfect-LFU or In-Cache-LFU, neither is it adaptive like LRU-2. That is why it has the poorest performance on real workloads.

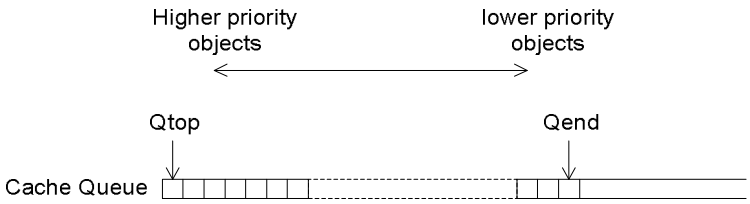

Figure 3-1 A Single Cache Queue

In our implementation of caching algorithms, we use cache queues to facilitate prioritized sequences of cached objects, as shown in Figure 3-1. Lower priority objects are located towards the end of the queue, so that they are first to be evicted.

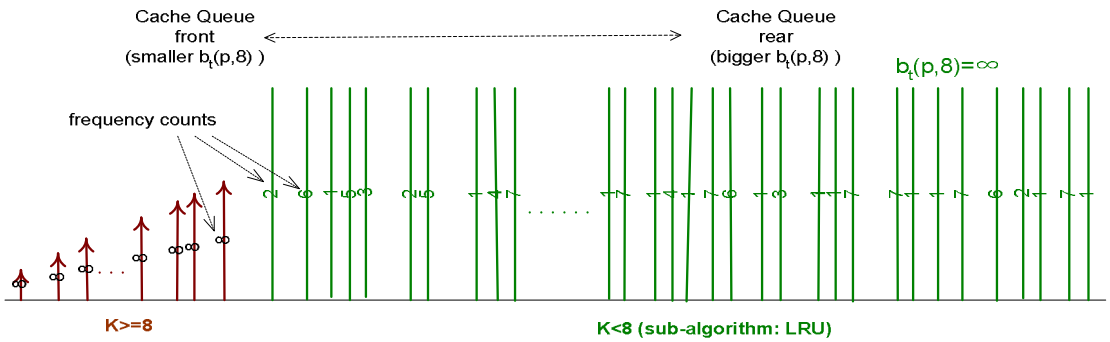

Figure 3-2 Cached Objects in LRU-8 Cache Queue

Figure 3-2 illustrates the cache queue and functioning of LRU-8, as a specific case of LRU-K. The axis represents the cache queue with the front of the queue at on the left, and the rear of the queue on the right. Every column in the queue stands for a cached object, with its frequency count on the column, varying from 1 to 8 in the case of LRU-8. The height of a column represents its backward K-distance, in this case backward 8-distance $b_t(p,8)$. We can see that cached objects in the queue are ordered so that lower $b_t(p,8)$ objects are located to the front of the queue, while higher $b_t(p,8)$ objects are located to the rear of the queue. So objects accessed less than 8 times are in the lower part of the queue because of their infinite $b_t(p,8)$. These objects are sub-ordered with their latest access time.

In LRU-8 with LRU as subsidiary algorithm, the following are true: (a) all cached objects with less than 8 accesses ($f < 8$) are cached in the sequence of latest access time, or backward 1-distances: $b_t(p,1)$. (b) Cached objects with at least 8 accesses ($f \geq 8$) are in the higher part of the queue. Only the latest 8 access records are held in bookkeeping, representing relatively short-term instead of long-term popularities. This portion of objects are cached in the sequence of backward 8-distance $b_t(p,8)$. (c) When objects are evicted starting from the rear towards the top of cache queue, objects with their latest access time beyond the *Correlation Reference period* are evicted from the cache queue, with no regards to their access frequencies. (d) Evicted objects' history information will be kept by the algorithm for a *Retain Information Period* from their latest accesses. This prevents the cases that objects' previous accesses never build up. For example, when object *p* is evicted, without a *Retain Information Period,* its history information is removed together with it. If soon after that object *p* is re-accessed and cached again, then it is treated as a new object requested only once. Without knowledge of its previous accesses, object *p* is very likely to be evicted soon after, and re-accessed, and so on. Even though object *p* is quite popular over time, caching it does not add to the hit rate in the cache.

We can see that, compared with the LFU algorithms, LRU-K is designed to capture relatively short-term rather than long-term popularity of requested objects, the latter of which is an outstanding feature of Web request streams. For Web request streams, we need a method to incorporate more frequency information into LRU-K that better distinguishes objects with various popularities in the long run. To accomplish this goal,

the algorithm should be adaptive like LRU-2 when workload pattern changes, while relying more on frequency information in its eviction decisions.

3.2.3 Design of webLRU-2

Based on the analysis of LRU-K and LFU, we propose webLRU-2 by combining the two. The design of webLRU-2 considers two aspects of Web proxy traces: short-term locality and long-term population of objects.

LRU-K and short-term locality

LRU-K is self-adaptive to changing workloads, especially with K=2, which excels on workloads with more significant short-term locality than long-term locality. Although LRU-K fits stable workloads better with some bigger K, depending on the degree of the workload's stability, bigger K certainly incurs greater overheads to the algorithm. Also, since real-world workloads on various proxies evolve over time, LRU-2 has been shown to have the best overall performance among the group, with a performance rather close to that of Perfect-LFU.

In our design of webLRU-K, we choose K=2, that is, webLRU-2, so that short-term locality in proxy traces is handled as in LRU-2. Since only 2 history access records of objects are kept by the algorithm, the algorithm has the advantages of low book-keeping and computational overhead, which is preferable and practical for large-size caches. This also makes possible that an extra control method to be added on top for differential QoS.

Frequency information

Perfect-LFU seems to generally have the best performance among the algorithms tested. However, its heavy book-keeping overhead is exhausting on caching and

computation resources, and makes it impractical on real proxies. In-Cache-LFU trims the book-keeping to a subset of the currently in-cache objects, and substantially lowers the overhead. However, the performance is considerably lowered as well.

We can see that the merit in Perfect-LFU is the knowledge of the most popular objects over time. A possible optimization is to trim the book-keeping of Perfect-LFU to a subset of objects that are still popular, instead of all objects accessed, and of a shorter history. This is known to previous researchers as the "aging problem" of frequency-based caching algorithms. We address this problem with a variable *Retain Information Period* depending on the popularity of individual objects:

- History information of evicted objects is kept for a period of time (*Retain Information Period*) before being erased. Thus we forget frequency information that is too long ago, unless the objects are continuously popular.
- As objects are of various popularities, the *Retain Information Period* for their history information should be varying as well, favoring more popular objects. Thus book-keeping is also narrowed to a subset of accessed objects that are either accessed recently, or most frequently requested over a longer period of time.

Without a proper *Retain Information Period*, there are two potential extremes of object book-keeping that may misrepresent the relative popularities of objects. The first extreme is continuously accessed objects that keep their history information over the whole span of caching, and may have an unfair advantage over other objects. The second extreme is that objects accessed between intervals that are not much bigger than their *Retain Information Period* are treated as "one-timers". We set the *Retain Information Period retn_TO* of individual objects to be proportional to the constant *retain_info_timeout*, and a function of the object's frequency count:

$$retn\_TO(f) = retain\_info\_timeout * fLev(f),$$

where *fLev(f)* is a function of *f*, the number of accesses to the object.

Cached objects in queue

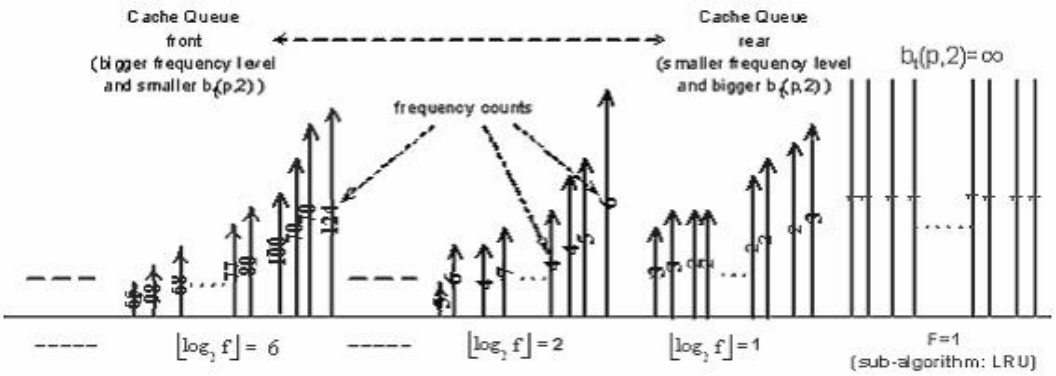

Figure 3-3 Cached Objects in webLRU-2 Cache Queue

Figure 3-3 illustrates the cache queue of webLRU-2. Cached objects are categorized into frequency groups with frequency level function $fLev(f) = \lfloor \log_2 f \rfloor$. Lower-level groups are located towards the rear of the cache queue, while higher-level groups are located towards the front. Within each group, objects are located in subsequence of backward 2-distance $b_t(p,2)$, so that objects with bigger $b_t(p,2)$ are located towards the queue rear. Once an object is evicted from the cache, it is removed from the cache queue. However, its record will be maintained within a retention period that is proportional to its frequency level *fLev(f)*. At the time it is re-accessed within the retention period, its access counter is updated before it is put back to the cache queue.

We follow the settings of correlation_timeout and retain_info_timeout in E. O'Neil's work [34], 5 seconds and 200 seconds, respectively. In Section 3.4.3 we test four methods of categorizing frequency levels with different functions fLev(f): $fLev(f) = f$,

$fLev(f) = \lfloor \log_2 f \rfloor$, $fLev(f) = \lfloor \log_5 f \rfloor$, and $fLev(f) = \lfloor \log_{10} f \rfloor$, as illustrated in Figure 3-4. $fLev(f) = f$ tends to stress more on object popularity as in LFU, while $fLev(f) = \lfloor \log_{10} f \rfloor$ caters more to short-term patterns in the workload, as in LRU-2. Also, $fLev(f) = \lfloor \log_{10} f \rfloor$ helps alleviate cache pollution better than $fLev(f) = f$ by de-emphasizing the gap between popularity levels.

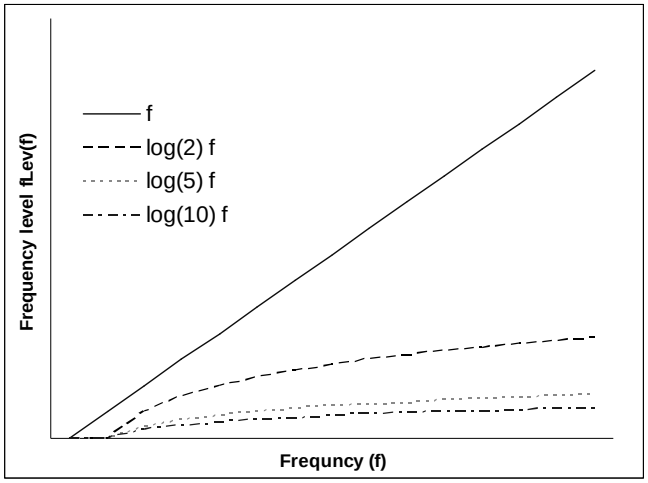

Figure 3-4 Frequency Level Function *fLev(f)* in Setting *Retain Information Period*

Definition of webLRU-2

To sum up, the webLRU-2 algorithm is defined as the following:

1) maintains a frequency counter for each object
2) group objects by their frequency level calculated as $fLev(f) = \lfloor \log_2 f \rfloor$, where f is the current frequency count. Object of lower frequency levels are first considered for eviction
3) within each frequency level group, objects with the biggest backward 2-distance $b_t(p,2)$ is first considered for eviction.
4) *retain information period* of individual objects are calculated proportional to their

frequency level:

$$retn\_TO(f) = retain\_info\_timeout * fLev(f),$$

where *fLev(f)* is a function of *f*, the number of accesses to the object.

A subsidiary policy, such as classic LRU, may be used to select a replacement victim among the pages with infinite Backward 2-distance. The pseudo code is provided in Appendix A for both unclassified and classified workloads.

webLRU-2 has a similar book-keeping overhead as LRU-2, with only an extra counter of in-cache objects. Retained history information of evicted objects are much less than those of Perfect-LFU, while more than those of LRU-2 to keep records of long-term popular objects for longer time.

Through comparisons between size-sensitive algorithms such as GDS, SIZE-threshold and LRV and size-insensitive ones, previous studies [5][14][31] have shown that the size factor does benefit Web caching algorithms, We also observe in preliminary experiments an improvement in webLRU-2 performance in terms of hit rates, in cost of some deterioration in byte hit rates. However, in this thesis, our prime objective is to exam webLRU-2's ability of combining short-time and long-term reference locality through a fair comparison with LRU-K and LFUs. We put it off to future research to seek proper value of size threshold of cached objects.

3.3 Simulation Model

The simulation in this chapter evaluates the proposed webLRU-K algorithm and verifies the adaptability inherent in the original LRU-K algorithm. We use both real proxy workloads and synthesized workloads to feed in the simulation model we established.

The simulator is designed to model a Web proxy with caching algorithms perfect-LFU, in-cache-LFU, LRU, LRU-K and webLRU-K. The simulation model used is shown in Figure 3-5. There are four groups of configurable parameters, namely, proxy setting, caching policy parameters, classification settings, and adaptor settings:

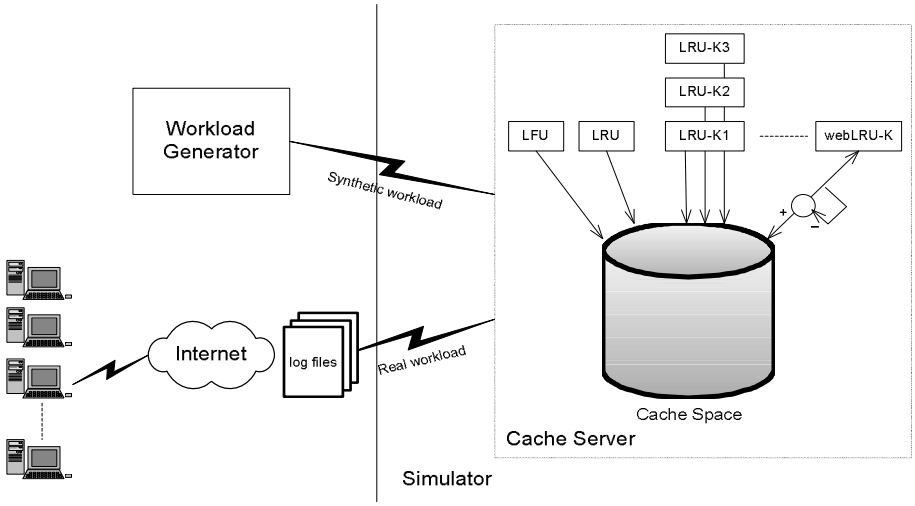

Figure 3-5 Simulation Model

1. Proxy setting

Cache space sets the total cache size of the proxy in mega-bytes.

2. Caching policy parameters

Parameters related to the caching algorithm include

- *correlation_timeout* that sets the value of *Correlation Reference Period* to eliminate the possibility of correlated access so that two accesses to the same object within this time-out period are treated as one. Any cached object is kept in the cache for this period from its most recent access so that correlated accesses are not missed.
- *retain_info_timeout* that sets the parameter in calculating *Retain Information Period* retn_TO(f) for the algorithm to maintain history information about an object since its latest access.

These parameters are as discussed in Section 3.2.3.

3. Classification settings

These parameters are for differential QoS and explained in Chapter 4.

4. Adaptor settings

These parameters define how the adaptor acts in monitoring/adjusting the model and are described in Chapter 4.

Appendix C.1 lists a table of the parameters in the simulator, and provides examples of settings of these parameters.

3.3.1 Simulation Assumptions

We make the following assumptions in our simulations:

- For synthetic workloads, we assume random access with Zipf-like distributions on a fixed number of cacheable files, with slope α varying from 0.5 to 0.9, generalizing at 0.70. For simplicity we assume the accesses are independent of each other, and there is no correlation between request frequency and response size or rate of change.
- Although it is possible to distinguish users making object references, we assume for simplicity that references are not distinguished by user, so any reference pairs within the correlation period (*correlation_timeout)* are considered correlated.
- It is assumed that no pages are invalidated by the cache during the experiments.
- Uncacheable objects include dynamic objects and objects whose sizes fall out of the size range set for cacheable objects. They also count for hit rate of the caching algorithms.
- Though there are several possible classification methods, as described in Section

2.3.1, we run our simulation with classification on file type, and the evaluation and analysis is based on this method.

- Each request is met with the whole file, as HTTP protocol supports whole file transfers only.
- Each experiment starts from initial state with an empty cache and finishes after processing the whole trace file.

3.3.2 Simulation Workloads

The ever-changing workload at the proxies is the challenge to both quality and stability of performance. We use both Zipf-like synthetic workload and real-world proxy workload to evaluate our caching model.

We described the Zipf-like distribution workload in Section 2.2. As the skew of the workload represents the degree to which statistical user interest is concentrated on a certain set of "hot objects", we synthesize a Zipf-like workload to simulate the workloads with different skew values, representing the degrees of user interest in the objects. With this synthetic workload we observe the adaptability and QoS differentiation the proposed scheme provides.

The synthetic workload is generated according to the Zipf-like distribution and size/popularity distribution rules. It is generated in three steps. First, a Zipf-like sequence of 300,000 accesses on 3,000 (workload 1) or 5,000 (workload 2) unique files is generated with Zipf-like distribution $P_N(i) = \frac{\Omega}{i^{\alpha}}$, with a certain value of α. Second, the unique files are extracted from a real workload from a Web proxy of NLANR, sorted according to their popularity and size. The final step is done when the workload is fed to

the simulation model, where the Zipf-like sequence and the actual files are combined together.

Item	Workload_1	Workload_2
Unique documents	3,000	5,000
Total bytes of unique documents (MB)	77.16	108.22
Minimum(size) (bytes)	117	117
Maximum(size) (bytes)	42,547,568	42,547,568
Mean(size) (bytes)	14,469.73	21,645.26
Median(size) (bytes)	320	2,987
Requests	300,000	300,000
Zipf Slope	0.5~0.9	0.7

Table 3-1 Characteristics of Synthetic Simulation Workloads in the Simulation

The synthetic workload represents the popularity distribution of accesses with Zipf-like rule, and the size/popularity distribution of objects by abstracting and sorting files from a real-world workload based on their frequency and size information. The synthetic workloads are tailored to meet the assumptions about Web proxy workloads discussed in Section 2.1. Appendix B.2 contains details on the generation of the synthetic workload.

There is, however, a deficiency in characterizing temporal locality and spatial locality in the synthetic workload, which has a slight effect on the performance evaluation of the algorithms. The impact is more on LRU-K with smaller Ks than bigger Ks. We try to offset the effect with a properly set *correlation_timeout* parameter.

The real-world workload for our simulation is obtained from sanitized access logs on some proxy caches of the National Laboratory for Applied Network Research (NLANR) [2]. Each log entry represents an access to a particular web object, and contains the following fields:

1. Timestamp: The time when the client socket is closed. The format is “Unix time” (seconds since Jan 1, 1970) with millisecond resolution.

2. Elapsed Time: The elapsed time of the request, in milliseconds.

3. Client Address: A random IP address identifying the client.

4. Log Tag and HTTP Code: The Log Tag describes how the request was treated locally (hit, miss, etc).

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

6. Request Method: The HTTP request method.

7. URL: The requested URL.

8. User Identity.

9. Hierarchy Data and Hostname: A description of how and where the requested object was fetched.

10. Content Type: The Content-type field from the HTTP reply.

Workloads	Requests	ReqSize (MB)	Files	FileSize (MB)
BO1 2003.11.18~2003.11.20	354293	6127.55	224832	3494.01
BO1 2003.11.21~2003.11.23	206101	4315.45	127380	2282.78
BO1 2003.12.29~2003.12.31	317160	5858.4	172009	2692.69
BO2 2003.12.30~2003.12.31	340816	5214.39	206200	2373.59

(a) Before classification

Workloads	Class IMAGE		Class TEXT		Class APP		Class Others	
	#Reqs	#Files	#Reqs	#Files	#Reqs	#Files	#Reqs	#Files
BO1 2003.11.18~2003.11.20	124248	53937	151327	126133	24258	9062	54373	35700
BO1 2003.11.21~2003.11.23	67081	31305	92556	71457	17226	5571	29185	19047
BO1 2003.12.29~2003.12.31	95362	36478	145669	99752	36447	10438	39634	25341
BO2 2003.12.30~2003.12.31	109284	46940	146208	111291	37360	15989	46804	31980

(b) Classified

Table 3-2 Real-world Workloads Used in the Simulation

*** All statistic data on cacheable files/requests**

We collect 11 days (from 18/11/2003 to 31/12/2003) of traces from NLANR

proxies BO1 and BO2 and use them as input to drive our simulation. Table 3-2 (a) and (b) outline the factors of the workloads with object type as the classification method. In this chapter our experiments treat the workloads without classification, while in Chapter 4, experiments classify objects by their object type, as being one possible classification method out of many.

3.3.3 Experiment Design and Setup

The experiments in this chapter set *Retain Information Period* for webLRU-2, and evaluate it against LRU, LRU-K, In-Cache-LFU and Perfect-LFU. Through experiments we try to find answers to the following questions:

- How different are the performance of LRU, LRU-K and LFU on synthetic workloads with varying skews, and on real traces?
- With what method should we set *Retain Information Period* of webLRU-2?
- How well does webLRU-2 suit the synthetic workloads and real traces, compared with LRU, LRU-K and LFU?

Four sets of experiments are carried out to address the questions. These experiments also serve as the basis for the experiments on the adaptor in the next chapter,

Experiments	**Algorithms**	**Workload**
Set 1. (Section 3.4.1) General performance of LRU-Ks on workloads with various skews	LRU, LRU-Ks, LFUs	Synthetic workload 1 (α=0.5~0.9)
Set 2. (Section 3.4.2) Algorithm performance on synthetic and real workloads	LRU, LRU-Ks, LFUs	Synthetic workload 2 (α=0.7) Real traces
Set 3. (Section 3.4.3) Setting *Retain Information Period* for webLRU-2	webLRU-2	Synthetic workload 2 (α=0.7)
Set 4. (Section 3.4.4) Performance evaluation of webLRU-2	LRU, LRU-2, webLRU-2, LFUs	Synthetic workload 2 (α=0.7) Real traces

Table 3-3 Experiment Sets for webLRU-K

Experiment Set 1. General performance of LRU-Ks on workloads with various skews

This experiment runs algorithm LRU, LRU-K (K=2, 3, 4, 8, 16), In-Cache-LFU and Perfect-LFU on synthetic Zipf-like workloads with α varying from 0.5 to 0.9 and cache size varying from 1% to 32% of the total unique file size of the workload, or 0.77 to 12.35MB.

Experiment Set 2. Algorithm performance on synthetic and real workloads

This experiment tries to evaluate the algorithms on real proxy traces, and also a synthetic workload that has the general proxy trace skew α=0.7. By comparing the performance of the algorithms, we analyze the effect of short-term locality on the workloads, suggesting preferable characteristics of the proposed caching algorithm to meet the need of Web caching. On the 11 days of recent proxy traces collected from two of NLANR's cache servers, the algorithms are run with cache spaces varying from 0 to 1.5GB; on the synthetic workload, the algorithms are run with cache spaces varying from 2% to 43.8% of the total unique file size of the workload, or 2.19 to 47.35MB.

Experiment Set 3. Setting *Retain Information Period* for webLRU-2

This experiment runs webLRU-2 on a synthetic Zipf-like workload on a fixed cache size of 13.83MB (12.78% total unique file size) to evaluate the algorithm with different methods of setting *Retain Information Period*, suggesting the best one among them.

Experiment Set 4. Performance evaluation of webLRU-2

This set of experiment runs webLRU-2 on both synthetic Zipf-like workloads and real proxy traces on varying cache sizes and evaluates it against the other algorithms.

3.4 Simulation Results and Analysis

3.4.1 General Performance of LRU, LRU-Ks and LFUs on Skewed Workloads

With this set of experiments we study the performance of algorithm LRU, LRU-K (K=2, 3, 4, 8, 16), In-Cache-LFU and Perfect-LFU on synthetic Zipf-like workloads with different skew values. Figure 3-6 shows the results of individual experiments on each algorithm with skewed workloads and varying cache sizes from 1% (0.77MB) to 16% (12.35MB) of the unique file size (77.16MB) of the workload..

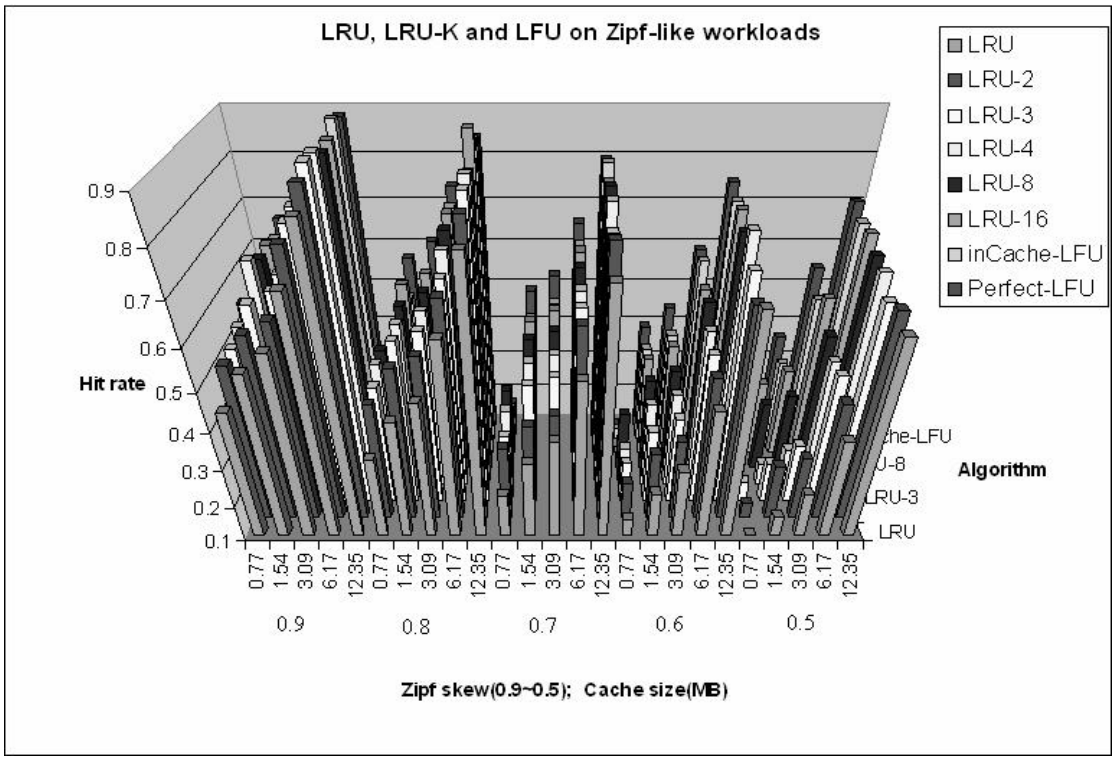

Figure 3-6 Algorithm Performance on Synthetic Proxy Workloads

With a fixed cached size, hit rates of the algorithms increase as workload skew α increases. With skew α fixed, hit rates of the algorithms increase with the cache size. Perfect-LFU generally has the highest hit rate and LRU has the lowest among the

algorithms. For each individual algorithm, the bigger the Zipf skew α is, the more obvious the gain of hit rate is as cache size increases. This is because when skew α is close to 1, more requests are concentrated on a small set of objects, the algorithms can achieve good performance with a relatively small cache, and the increase of cache size does not have as big an effect as when skew α is smaller. Axis Y plots the average hit rate of each experiment; Axis Z plots the caching algorithms involved; Axis X plots the varying cache sizes from 0.77MB to 12.35MB, grouped in every workload skew α from 0.9 to 0.5.

Figure 3-7 plots the hit rates of each algorithm against various cache size, averaged over the spectrum of workload skews varying from 0.5 to 0.9. Similarly, Figure 3-8 plots the hit rates of each algorithm against various workload skew α, averaged over different cache sizes varying from 0.77MB to 12.35MB.

We can see that in terms of hit rate, LRU-16 is better than LRU-8, which is better than LRU-4, then LRU-3, LRU-2. Let integers $k_2>k_1>1$, the gain in performance of LRU-2 from LRU-1 is much bigger than those of LRU-k_2 from LRU-k_1. LRU and LRU-K seem to have similar trends as Perfect-LFU and In-Cache-LFU. This indicates a coincidence between short-term and long-term localities in this random accesses sequence with Zipf-like distribution.

When the cache size is very small compared to individual file sizes, the number of cached files is very limited, the time for most of the cached objects to stay in is also much shorter. In such cases, short-term reference correlation plays a more important role and has greater effect on an algorithm's performance. LFUs tend to have an inferior performance to those of LRU-Ks. This is because. LFUs will cache those with top

reference counts over time, while LRU-Ks will choose those mostly accessed in recent history, which will more likely be popular than the former in the short-term.

As shown in Figure 3-7, Perfect-LFU and In-Cache-LFU have much lower hit rates when cache size is too small (0.77MB) to hold enough of the most frequently accessed objects to get most cache hits, while their hit rates have a drastic increase at cache size 1.54MB where they begin to have superior performance over LRU-Ks. We also observe generally lowered performance than expected at cache size 3.09MB in the algorithms tested. These mild gains in hit rates are caused by the fact that the cache size is still small compared with big objects in the traces, and the increase in cache size at this point does not favor the big objects as significantly as those before and after it.

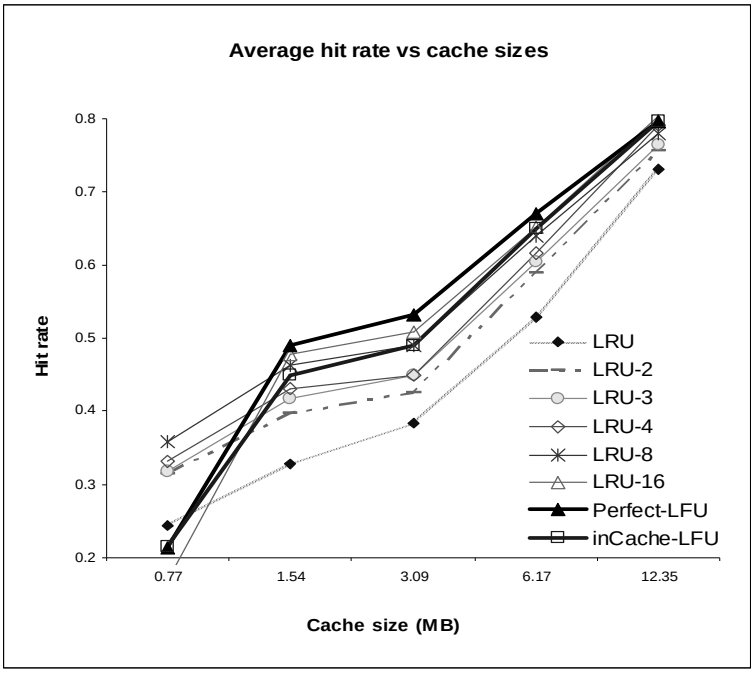

Figure 3-7 Average Hit Rate vs. Cache Sizes

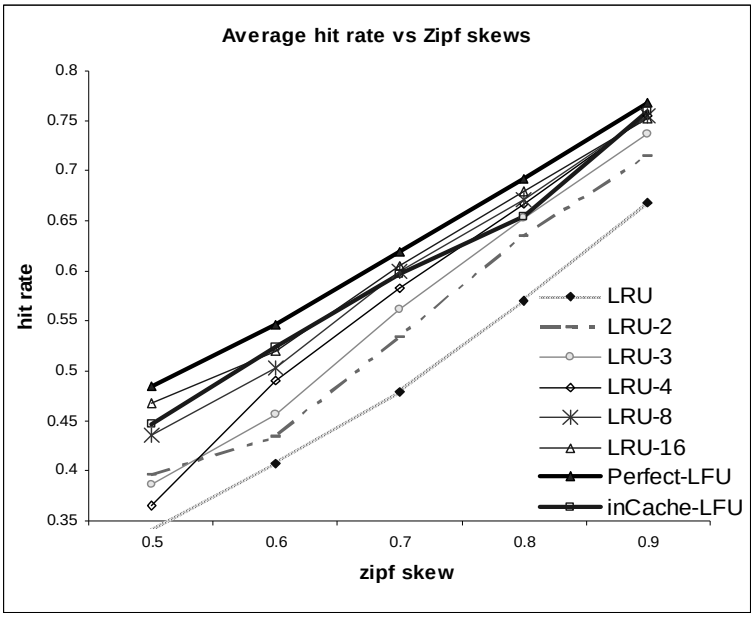

Figure 3-8 Average Hit Rates vs. Zipf Skews

3.4.2 Performance of LRU, LRU-Ks and LFUs on Synthetic and Real Workloads

With this experiment set we study the performance of the algorithms on real proxy traces, compared with results on a synthetic workload that has the general proxy trace skew α=0.7. We analyze the effect of short-term locality on the workloads to guide our design of the new caching algorithm.

Figure 3-9 shows the algorithm comparison on the second synthetic workload with α=0.7. Similar to the results of Experiments Set 1, Perfect-LFU still has the highest hit rate and LRU the lowest. In-Cache-LFU has a substantially lower performance compared with PerfectLFU, however, it is comparable to the best of LRU-K algorithms, that is, LRU-4 in this experiment. The trend of LRU-K performance versus value K is different than in Experiment Set 1. With $K \leqslant 4$, hit rate of LRU-K increases as K increases; With $K>4$, hit rate of LRU-K decreases as K increases. This indicates some degree of conflict

between short-term and long-term localities. LRU-8 and LRU-16 do not adapt well to the short-term locality, and cannot capture long-term popularity within 8 or 16 history access records, so their performance degrades.

Figure 3-10 shows algorithm performance on 4 sets of real workloads, each of which consists of 2 to 3 sequential days' traces to a proxy. Perfect-LFU still has the highest hit rate. In-Cache-LFU has a lower performance compared with Perfect-LFU, but is comparable to LRU-2, which is the best among LRU-K algorithms. LRU-K performance decreases as K increases, so that LRU-2 has the highest hit rate among the group, while LRU-16 and LRU-8 generally have lower hit rates than LRU. The trends of the algorithms in these experiments seem almost linear with cache size smaller than 246.46MB, and then the increasing rates of hit rates over cache sizes rise quite significantly. This could be caused by the relatively small cache sizes, compared with the total file sizes of traces, and also the limited length of traces we run our experiments on.

Another issue of LRU-16 and LRU-8, which also applies to Perfect-LFU, is their heavy book-keeping overhead. As seen in experiment on trace 20031229~20031231 of proxy BO1, the simulator runs out of memory with Perfect-LFU and LRU-16, LRU-8 with 1.5GB of cache size, and in experiments on trace 20031230~20031231 of proxy BO2, the simulator runs out of memory with Perfect-LFU with 1.5GB of cache size.

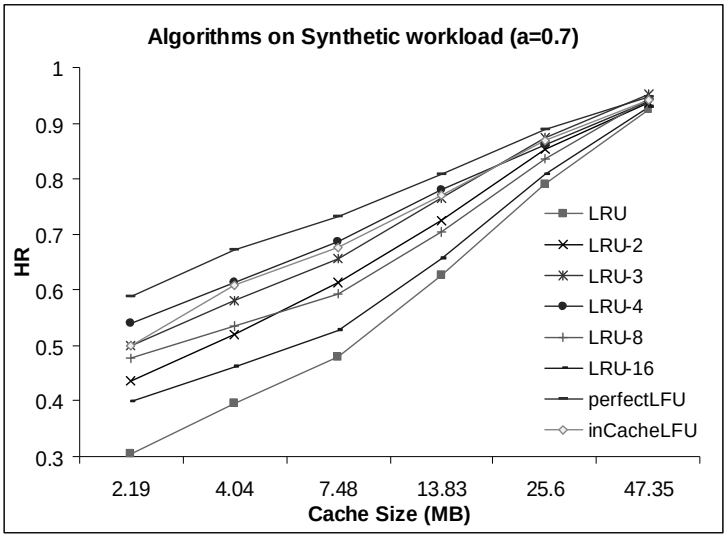

Figure 3-9 LRU-K, LFU on Synthetic Workload with α=0.7

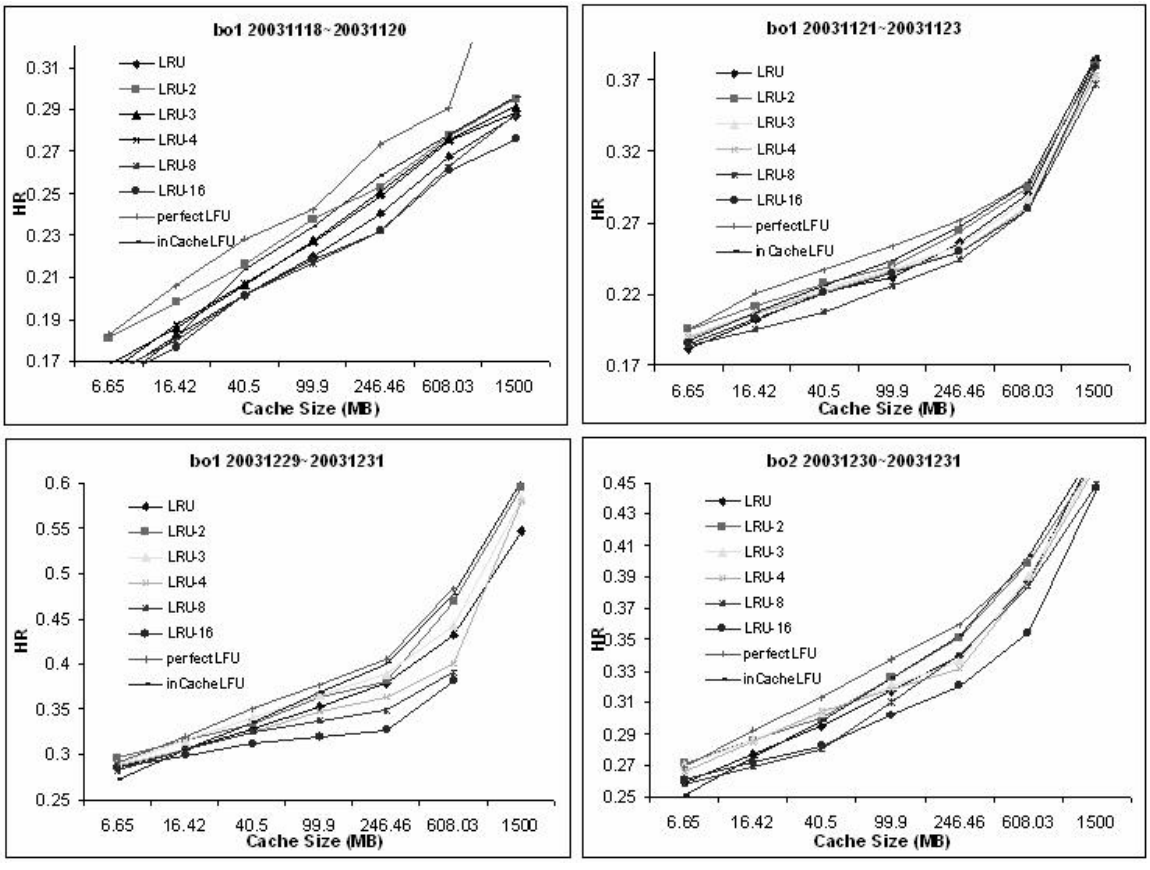

Figure 3-10 LRU-K, LFU on Real Traces

3.4.3 Setting *Retain Information Period* for webLRU-2

This experiment runs webLRU-2 on a synthetic Zipf-like workload with skew α=0.7, on a fixed cache size of 13.83MB (12.78% total unique file size) to evaluate the algorithm with different methods of dynamically setting *Retain Information Period* according to the object's frequency count, suggesting the best one among them. The four frequency level functions are $fLev(f) = f$, $fLev(f) = \lfloor \log_2 f \rfloor$, $fLev(f) = \lfloor \log_5 f \rfloor$, and $fLev(f) = \lfloor \log_{10} f \rfloor$, where *f* is the frequency count of the object. Figure 3-11 plots hit rate of webLRU-2 with different methods of calculating its *Retain Information Period* (*retn_TO(f)*), together with hit rates of Perfect-LFU, In-Cache-LFU and LRU-2, which serve as reference values in our selection of the best method. As indicated, among the four methods, the method with $fLev(f) = \lfloor \log_2 f \rfloor$ achieves the best hit rate, which is much better than that of LRU-2, and between those of In-Cache-LFU and Perfect-LFU.

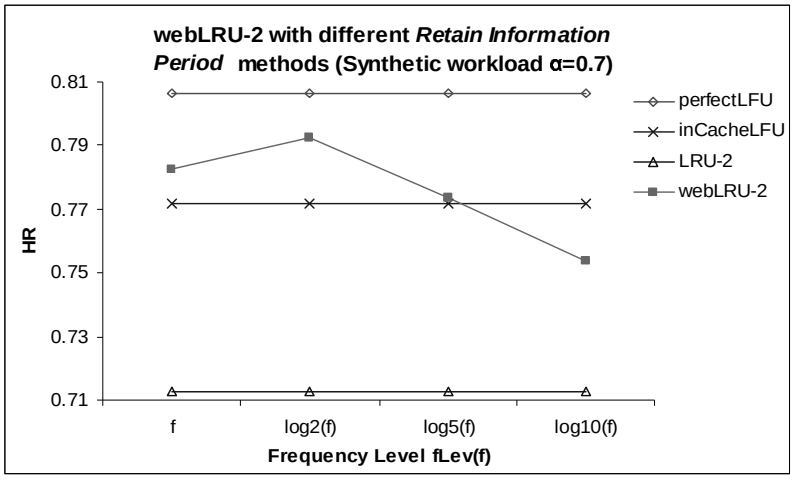

Figure 3-11 *Retain Information Period* methods of webLRU-2

Synthetic workload α = 0.7; cache size = 13.83MB

This is resulted by the facts that 1) although the bigger the value of *Retain Information Period*, the more accurate the frequency information is kept for every

accessed object, the setting of *Retain Information Period* with $fLev(f) = \lfloor \log_2 f \rfloor$ bears a reasonable tradeoff; and 2) the sequencing in cache queue (Figure 3-3) with $fLev(f) = \lfloor \log_2 f \rfloor$ gets a preferable balance between long-term and short-term locality of objects, and thus has a better performance than the other methods.

Based on this result, we choose $fLev(f) = \lfloor \log_2 f \rfloor$ in ordering the cached objects in the cache queue, and set *Retain Information Period* of webLRU-2 with

$$retn\_TO(f) = retain\_info\_timeout * \lfloor \log_2 f \rfloor,$$

where *f* is the reference count of the object, and *retain_info_timeout* is a preset parameter of webLRU-K. Although this result is obtained on a specific experiment (synthetic workload α = 0.7, cache size = 13.83MB), we generalize it base on the foregoing analysis.

3.4.4 Performance Evaluation of webLRU-2 on Synthetic and Real Workloads

With the chosen optimal method of setting object *Retain Information Period* in the previous experiment, this set of experiments runs webLRU-2 on both synthetic Zipf-like workloads and real proxy traces on varying cache sizes and evaluates it against the major algorithms Perfect-LFU, In-Cache-LFU, and LRU-2.

Figure 3-12 shows the performance of the algorithms run on the synthetic Zipf-like workloads. The hit rates of the algorithms are approximately linear on various cache sizes, increasing as the cache size does. Compared with LRU-2, the gain in terms of hit rate of webLRU-2 is substantial, especially on small cache sizes where it is up to about 35 percent higher. The hit rate of webLRU-2 is between those of In-Cache-LFU and

Perfect-LFU, Perfect-LFU having the highest hit rate.

Figure 3-13 compares the performance of the algorithms on 4 sets of real proxy workloads. The overall performance of webLRU-2 compared with the LFUs is similar to that on the synthetic workload, with exception that webLRU-2 outperforms Perfect-LFU on small cache size. In most cases, webLRU-2 outperforms LRU-2 quite substantially when the cache size is smaller than 246.46MB (about 10% of the total file size). When the cache size is bigger, hit rate of webLRU-2 is similar to that of LRU-2. This indicates that on real proxy workloads, webLRU-2 captures both long-term and short-time localities in the workloads and has a quite preferable performance, especially on a small cache.

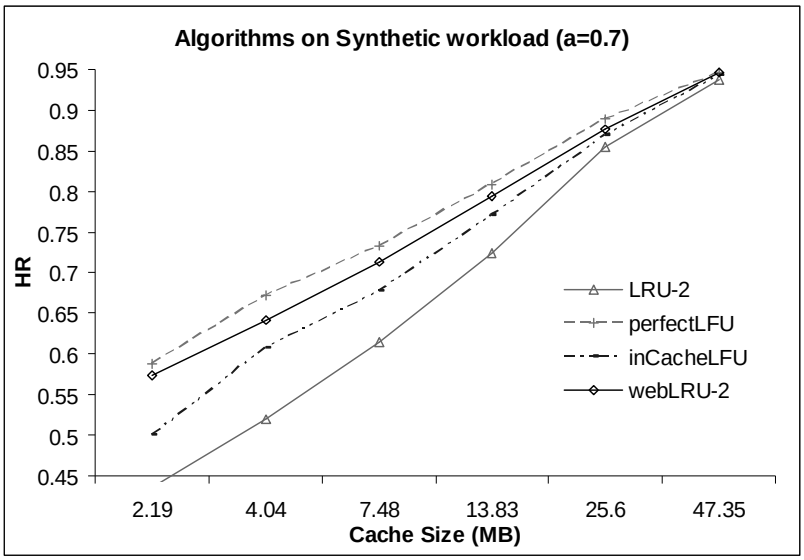

Figure 3-12 webLRU-2 on Zipf-like Synthetic Workload with α=0.7

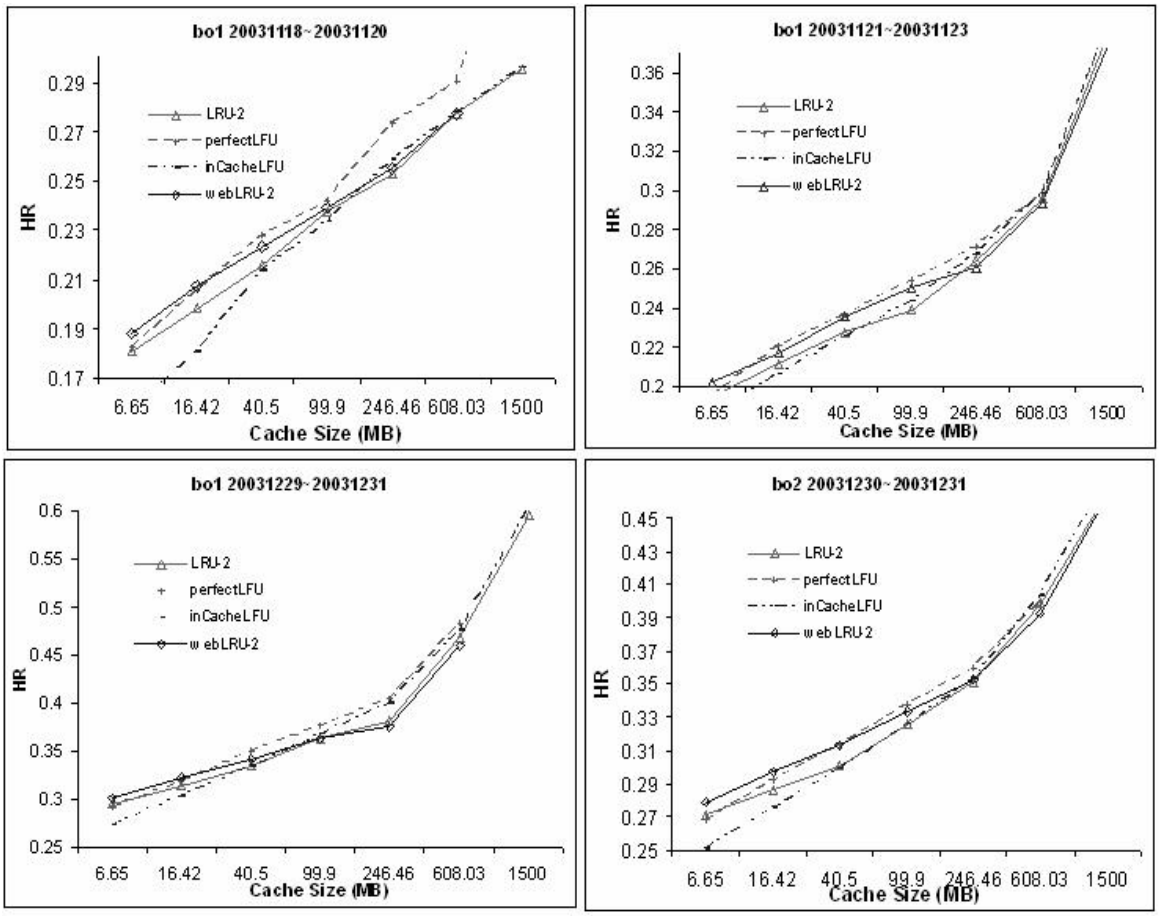

Figure 3- 13 webLRU-2 on Real-world Proxy Workloads

3.4.5 Discussion and Summary

We ran experiments on algorithms LRU, LRU-K with different K values, and two versions of LFU with both synthetic workloads and real proxy traces. We designed webLRU-2 based on an analysis of simulation results, to better suit Web request streams. We also ran a set of experiments to evaluate webLRU-2 against LRU, LRU-K, and LFUs. The following are our observations on the simulation results and considerations in our design of webLRU-2:

- Algorithm performance differs between synthetic workloads and real workloads due to the different workload characteristics. Perfect-LFU outperforms In-Cache-LFU,

LRU-K, and LRU on synthetic workloads much more than on real workloads since synthetic workloads have more constant access patterns than real workloads.

- Perfect-LFU succeeds in capturing the long-term popularity of accessed objects. However, it incurs high book-keeping overhead, and is not practical for large caches. In-Cache-LFU, on the other hand, requires only a limited overhead but has significantly worse performance. As short-term locality is significant in real workloads, LFUs' ignorance to recency information becomes a drawback to their performance.
- LRU-2 is generally the most efficient of the LRU-K algorithms on real proxy workloads. LRU-K with bigger K is less adaptive but better suited to stable workloads. However, as K increases, the algorithm incurs much more book-keeping overhead in keeping K access records of the objects, and the computation complexity increases as well.
- We hypothesize that webLRU-2 is better for Web caching systems because it captures both short-term and long-term localities inherent in Web request streams. It has the following features:
  - Sensitive to long-term popularity of objects, by keeping frequency information with an access counter as LFUs do.
  - Adaptive to evolving workloads, as inherited from LRU-2.
  - Addresses the "aging problem" of the frequency information with a dynamically set *Retain Information Period* for each evicted object as a function of its number of accesses. So popular objects are kept track of in long-term, instead of short-term as in In-Cache-LFU and LRU-K. On the other hand, not retaining object information over the whole process help

resetting records of previously popular objects and preventing cache pollution.

- Relatively light in overheads. Compared with LRU-2, extra overhead of webLRU-2 comes from the frequency counter for each object, and retained history records of popular objects. With a carefully chosen method to set *Retain Information Period*, we come to a balance between more awareness of long-term popularity and reasonable overhead.

In evaluating webLRU-2 against the other algorithms, the results show that by adding frequency information of in-cache objects and relative popular objects, webLRU-2 has a preferable performance, outperforming LRU-2 and In-Cache-LFU on relatively small cache sizes (smaller than 10% of total file size of our simulation workloads, which is quite practical to real proxies).

We notice that although, as a rule of thumb, hit rate of an algorithm increases logarithmically as the cache size increases, the trends of the algorithms in our experiments seem a little different. This should be caused by the limited objects number and sizes of the workloads and the relatively small cache sizes. We expect the hit rates of the algorithms to converge as the cache size increases further.

Chapter 4

An Adaptive Scheme with webLRU-K

Previous researchers have proposed many ways to provide differential QoS in space allocation on a proxy. In this chapter we present an adaptive scheme and adopt the webLRU-2 algorithm proposed in Chapter 3.

A simple weighted replacement policy can provide differential QoS in caching to some extent, as does the server-weighted LFU proposed by T. P. Kelly et al [26]. Fixed weights are pre-set to each URL, and the key of each cached object is proportional to its weight. However, fixed weights alone do not guarantee user-perceived improvement. For instance, for a workload with most popular objects from low weight URLs, the proxy cache will be occupied by these objects because they have the highest key calculated by $W_u * N_u$, where W_u is the weight of object *u*, and N_u is the number of references to object *u* since it entered the cache. Although the scheme is good in the sense that it prioritizes objects from high weight URLs, there is no guarantee of the differentiated services. Without an adaptive method, the performance of the proxy can not approach its QoS goal.

In their adaptive control scheme, Lu et al [32] achieve QoS portability with a self-tuning performance regulator, based on approximate linear difference equation models. For a given pair of classes, let $C_i : C_j$ be the desired ratio of their hit rates. It serves as the target performance reference in the corresponding adaptive control loop.

The objective of the control loop is to make the system output (the measured relative hit ratio $\frac{H_i}{H_j}$) to converge to $\frac{C_i}{C_j}$. In the process of feedback control, estimated storage space is calculated for each class to guide the two separate processes of caching: de-allocation and allocation. However, this scheme has some drawbacks, such as impractical design for cases of more than 2 classes, strict assumption versus possible parameter variations in the dynamics, single cache queue for all class objects and complexity in adaptive control.

Their controller design is based on a linear approximation of the intrinsically nonlinear web cache system. The authors experimentally show that this approach is robust and leads to a significant system performance improvement over non-adaptive schemes.

In the scheme, the adaptive control loop is designed for each pair of classes. A larger number of classes would entail a control loop per pair. This will cause a complicated iteration in cases with more than 2 classes. Also, the design of one list for cached objects of all classes makes it impossible to precisely compare objects of different classes in space de-allocation under the QoS specification, and in space allocation the server would not cache objects of a class that exceeds the class's estimated cache space, which may very possibly hurt the overall performance of that class.

In this chapter we design and implement an adaptive model to provide differentiated services in the sense of relative average hit rate among classes with webLRU-2 proposed in Chapter 3. Per-class caching queues are maintained for objects of each class, so that better allocation/de-allocation decision is possible. Our model can be viewed as a simplified variation of the work by Lu et al in [32]. In Section 4.1, we describe the design

and implementation of our model. Then in Section 4.2, we describe the simulation model and experiment settings. We present experiments and results with the scheme, compared with the non-control model and the fixed-weight model, in Section 4.3.

4.1 The Adaptive Scheme

4.1.1 The Model

Figure 4-1 illustrates our adaptive model. The adaptor of the model periodically monitors the output performance of each class, updates the class weights accordingly, and then decides upon the space allocation needed to reach target relative hit ratio. The adaptor accepts as input a specification of proportional hit ratios, which is independent of the platform and incoming workload. System administrators merely set the desired QoS specification.

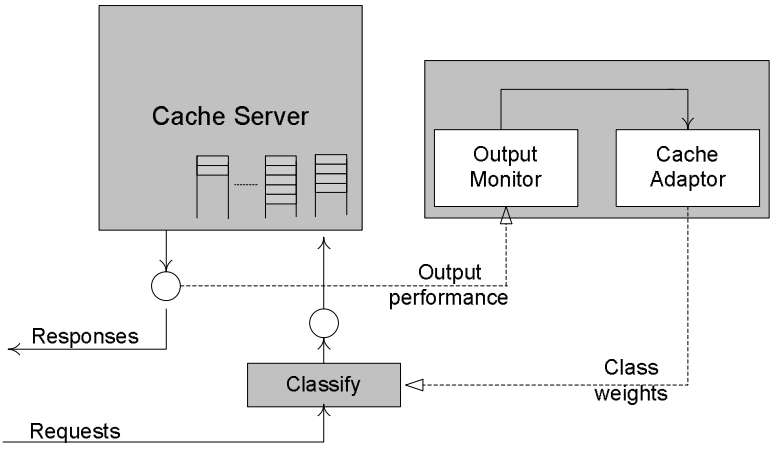

Figure 4-1 The Adaptive Model

In our design, the cache server maintains per-class caching queues for different class objects, and the adaptor estimates the new weights before normalization that assures their sum equals 1. Thus, only one adaptor is needed for all the classes, unlike the design of Lu et al ([32]), which employs one controller for each successive pair of classes. The control

method is simplified. We neglect the controller design block in Lu's model, and map the hit rate proportional deviation of each class into adjustments to its weight.

The limitation of the feedback loop is that its success in achieving its QoS goal is contingent on the feasibility of the specified target relative hit ratio. Assume the average hit rate of the unmodified cache is $X\%$. In general, when space is divided equally among classes, the maximum multiplicative increase in space that any one class can get is bounded by the number of classes N. Since hit rate increases logarithmically with cache size, in a cache of total size S, the maximum increase in hit rate for the highest priority class is upper-bounded by $\ln N$. After some algebraic manipulation, this leads to $X_{\max} \le X/(1-\ln N/\ln S)$. If the relative hit ratio between the top and bottom classes is q, the maximum hit rate of the bottom class is bounded by $X_{\max}/q$. This gives some orientation for specifying the desired relative hit ratio q. We set upper-bounds for relative weights between top and bottom classes to prevent the adaptor from decreasing the bottom class hit rate too much in order to converge to the desired hit rate proportion.

4.1.2 Implementing Adaptability in Cache Space Allocation

Output Monitor

The output monitor periodically collects performance data on the cache proxy. It calculates deviations of each class hit rate for input to the cache adaptor. Let the preset QoS specification for N classes be a proportional hit rate, $set\_pp\_hitrates[N]$ (e.g., $set\_pp\_hitrates[4] = (15,4,7,2)$ means hit rates of Class0:Class1:Class2:Class3 = 15:4:7:2), and the compound average hit rate of each class in interval t be $cls\_HR_t[N]$. We map the class deviation in a proportional way so that class weights could be adjusted accordingly. For a class with hit rate proportion beyond its preset goal

$set\_pp\_hitrates[i]$, its class weight does not need to change, the class deviation is 0; For a class with hit rate proportion lower than its preset goal $set\_pp\_hitrates[i]$, its class weight need to be adjusted proportionally according to the deviation of its current hit rate proportion from its desired proportion. With this adaptation through proportion, $cls\_DEV_t[i]$ works as a lever between the class weights and the class hit rates.

Then correspondently, the compound class deviation $cls\_DEV_t[i]$ for class *i* in interval *t*, can be calculated as follows:

$$cls\_DEV_t[i] = \begin{cases} 0, & \text{If } cls\_HR_t[i] / \sum_j^N cls\_HR_t[j] \geq set\_pp\_hitrates[i], \\ \dfrac{cls\_HR_t[i] / \sum_j^N cls\_HR_t[j] - set\_pp\_hitrates[i]}{set\_pp\_hitrates[i]}, & \\ & \text{If } cls\_HR_t[i] / \sum_j^N cls\_HR_t[j] < set\_pp\_hitrates[i]. \end{cases} \quad (1)$$

where the compound class hit rate $cls\_HR_t[N]$ is the proportional average hit rate over the whole caching process, rather than one interval, in order to eliminate the fluctuating effect in the adaptor in the face of the changing workload pattern. Let $cls\_hitrates_t[i]$ be the average hit rate of class *i* in interval *t*, and λ ($0<\lambda<1$) be the averaging factor, the long-term compound hit rate of class *i* at the end of interval *t* is calculated by the output monitor as:

$$cls\_HR_t[i] = (1-\lambda) * cls\_HR_{t-1} + \lambda * cls\_hitrates_t[i] \quad (2)$$

λ represents the proportion of hit rate of current interval *t*, in the compound hit rate. With a smaller value of λ, the compound hit rate is more stable, reflecting the overall hit rate over the long run. With a bigger value of λ, the compound hit rate tends to fluctuate with the current performance of the proxy. Based on the compound hit rate monitored, the

adaptation may be retarded or over-reacting, both impairing the effectiveness of the adaptor. In Experiment Set 2 (Section 4.3.2), the impact of average factor λ is observed in order to set a proper value for it. In our simulation, the interval is set to be 10 minutes, which is reasonable for continuous adaptation by the feedback loop.

Cache Adaptor

The cache adaptor makes adjustments to class weights based on the class deviations monitored. If a class is below its desired relative hit rate in the previous interval, its weight is increased in the next. If a class meets its desired relative hit rate, its weight is not changed. Then the adaptor normalizes the class weights so that they sum to 1. Thus, in the next interval, the currently underperforming classes will have a higher priority in cache space allocation, while currently well-performing classes may have a slightly decreased priority after the normalization.

Let $cls\_weights'[i]$ be the current class weight for class i, and $cls\_weights[i]$ be the updated class weight to converge to the desired relative hit rate. The adaptor then maps class deviations into weighted deviation, such that

$$weight\_deviations[i] = \frac{cls\_weights'[i] - cls\_weights[i]}{cls\_weights[i]}, \tag{3}$$

$$weight\_deviations[i] = cls\_DEV_t[i], \tag{4}$$

and adjusts accordingly for the next interval. The adjustment is made only when the class deviation from its hit rate proportion is greater than a deviation threshold *h*, so as to eliminate the fluctuation effect. From (3) and (4), we derive:

$$cls\_weights[i] = \frac{cls\_weights'[i]}{1 + cls\_DEV_t[i]}, \quad \text{when} \quad cls\_DEV_t[i] > h \tag{5}$$

At the end of the adjustment, the class weights are normalized so that $\sum_{i}^{N} cls\_weights[i] = 1$. Thus the class weights are adjusted towards the desired relative hit ratio, though only one iteration of adjustment is needed. However, since class weights are calculated separately before normalization, there is a deviation from the real proportion of each class. This deviation increases as the number of classes increases.

Cache Server (Caching Policy)

The cache server maintains one cache queue for each class, as illustrated in Figure 4-2. Higher priority objects are located towards the top of its class queue, while lower priority objects are located towards the end of the queue, and are first considered for eviction when a request comes for an object not currently cached and bigger than the available cache space.

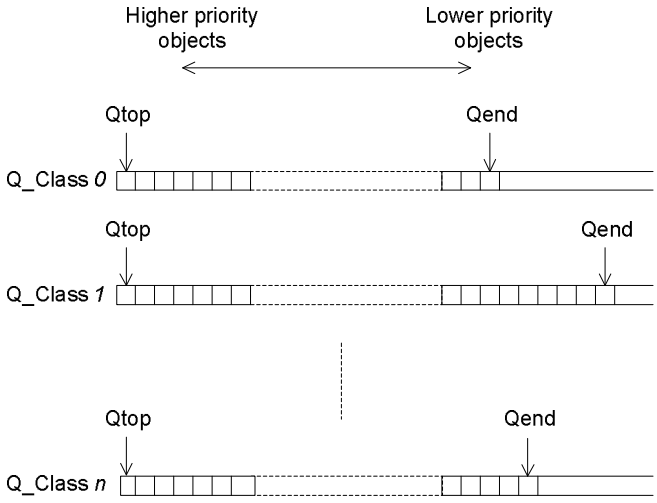

Figure 4-2 Classified Cache Queues for Resident Objects

The cache space management includes two processes: de-allocation and allocation. In the space de-allocation process, objects at the end of each class queue and that are beyond the correlation timeout are compared by their weighted keys. For example with

webLRU-2, suppose objects *p[0], p[1],…, p[n-1]* are beyond correlation timeout and at the ends of the cache queues of classes 0, 1, …, n-1, respectively. Their weighted keys, $b_t(p[i],2)/(cls\_weights[i]* fLev(f[i]))$ are compared, and the object with the biggest weighted key is evicted. This process repeats until there is enough space for the requested object. In the space allocation process, requested object *p* is allocated in its class queue in a position according to the sequence of object keys, in the case of webLRU-2, frequency level $fLev(f)$ and then backward 2-distance $b_t(p,2)$.

Adaptor Evaluation

To evaluate the performance of the adaptor, accumulated weighted deviation is defined as

$$w\_deviation_t = \sum_i^N (set\_pp\_hitrates[i] * cls\_DEV_t[i]), \qquad (6)$$

where $set\_pp\_hitrates[i]$ serves as the desired weight of class i in calculation of weighted deviation, so that deviation of a premium class that is assigned with a bigger relative hit rate would also contribute more to the weighted deviation.

In our simulations (Section 4.3) we use weighted deviation as a measure to evaluate the system performance in pursuing the QoS specification. Also, the hit rates of each class and the whole workload are used in the evaluation, to avoid the possibility of the adaptor pursuing the QoS goal by dramatically lowering the overall hit rate in order to achieve higher hit rates for the premium class.

4.2 Simulation Model

The simulator in this chapter is similar to that in Chapter 3, as shown in Figure 3-5, except that an adaptor is added and webLRU-2 maintains multiple queues in the cache for classes of objects. The assumptions described in Section 3.3.1 also apply for this simulation. Parameters of classification settings and adaptor settings are in the third and fourth group in the table of Appendix C.1:

- Classification settings

These are the parameters of QoS specification in terms of classes of objects (*set_cls_num*), the classification method (*set_cls_method*), the keywords of the first $set\_cls\_num - 1$ classes (*set_cls_keywords)* in their object types (as the classification method chosen with the workload, as shown in Table 3-2), and the goal of proportional hit rates (*set_pp_hitrates*). The (*set_cls_num*)-th class consists of all objects that fall out of the specified classes.

- Adaptor settings

These are the adaptor parameters that determine how often the system collects statistical data of the caching algorithms and adjusts the cache space allocation in its feedback loop:

  - *Interval* and *min_request* set the interval for iteration of the feedback loop. For real-world proxy workloads, the *interval* sets the time in minutes, with a minimum number (*min_requests)* of requests in every interval to assure that the statistical data is meaningful even in times of scarce requests. For synthetic workloads, where requests come in a sequence without temporal information, the intervals of statistical data collection is set by number of requests with

min_requests;

- *Max_pp_weight* sets the maximum size ratio between the object class with the highest priority and that with the lowest priority. This parameter makes sure that even in situations where higher priority class objects are in great need, the most used objects in the lower priority class will still be cached.
- *Wtd_deviation* sets the desired weighted deviation. When the weighted sum of the class deviations is lower than this value, no adjustment to the cache allocation is needed.

4.2.1 Simulation Workloads

Each of the experiments of the simulation is run on the 3 days of real trace from NLANR cache server BO1 from 2003.11.21 to 2003.11.23, as listed in Table 3-2. We use two classification methods. The first one classifies the workload into two classes: image objects and other objects. The second one classifies the workload into four classes: image, text, application and other objects. The number of requests to, and unique files in, the various classes are listed in Table 4-1. Experiments in this chapter are run on a cache size of 100MB, with which webLRU-2 had performance close to perfect-LFU, and better than LRU-2.

<table>
<tr><th></th><th>Class IMAGE</th><th>Class TEXT</th><th>Class APP</th><th>Class Other</th></tr>
<tr><td rowspan="2">Requests</td><td rowspan="2">92556</td><td>67081</td><td>17226</td><td>29185</td></tr>
<tr><td colspan="3">113545</td></tr>
<tr><td rowspan="2">Files</td><td rowspan="2">71457</td><td>31305</td><td>5571</td><td>19047</td></tr>
<tr><td colspan="3">55923</td></tr>
</table>

Table 4-1 Classified Workload on Proxy BO1 2003.11.21 to 2003.11.23

4.2.2 Experiment Design and Setup

We design and run four sets of experiments to evaluate the adaptive model against

the non-controlled model, and the fixed-weight controlled model. Each experiment set is run with the two classification methods. We first run the non-control model to obtain the relative hit rates of the classes without differentiation. Then our adaptive model is run with various values of the averaging factor λ to get the best performance. Two proportional goals are set to evaluate the models. The first goal is set to further favor the top class, which has the highest hit rate, to a reasonable degree. The second goal is set to be far away from the relative hit rates without differentiation. Both the weighted deviation from the QoS goal and the class hit rates are measured.

Experiment Set 1. Relative hit rates of classes with the non-control model

This experiment runs the non-control model on the workload classified into 2 and 4 classes respectively, and obtains the relative hit rates of each class without differentiation in space allocation. This relative hit ratio serves as a reference point for setting QoS goals for the models.

Experiment Set 2. Setting the averaging factor λ

In this experiment our adaptive model is run with different values of the averaging factor λ, which represents the method the output monitor of the feedback loop uses to calculate class deviations. We observe the effect on cache performance of different values of λ, and select a value for the later experiments.

Experiment Set 3. Adaptive model with a reasonable QoS goal

This experiment sets a QoS goal that further favors the top class, which has the highest hit rate, to a reasonable degree, and compares the performance of our adaptive model with those of the non-control model and the fixed-weight model.

Experiment Set 4. Adaptive model with an “unrealistic” QoS goal

This experiment sets a QoS goal that is far away from the relative hit rates without

differential control. Again, our model is compared with the non-control model and the fixed-weight model to show the performance of the models pursuing an almost unachievable goal.

Experiments	Models	Purpose
Set 1. (Section 4.3.1) Relative hit rates of non-control model	Non-control model Fixed-weight model Our model	To obtain the relative hit rates of the classes without differentiation.
Set 2. (Section 4.3.2) Setting the averaging factor λ	Our model	To observe the effect of averaging factor, and set a proper value for the simulator.
Set 3. (Section 4.3.3) Adaptive model with a reasonable QoS goal	Non-control model Fixed-weight model Our model	To show that our model outperforms the others with an achievable QoS goal.
Set 4. (Section 4.3.4) Adaptive model with an "unrealistic" QoS goal	Non-control model Fixed-weight model Our model	To observe the adaptor's effect on the proxy with an in-achievable QoS goal.

Table 4-2 Experiment Sets for the Adaptor

4.3 Simulation Results

4.3.1 Relative Hit Ratio of Classes without Differential Control

In this experiment set we run webLRU-2 with multiple cache queues on the 3 days of classified workload, in both 2-class and 4-class cases. Table 4-3 lists the average hit rates of both the overall system and individual classes. The average hit rates are calculated over all the 10-minute intervals on the cache server without differential control. As the success of adaptive models in converging to their QoS goals without significantly compromising the hit rates of bottom class(es) relies largely on the feasibility of the specified target relative hit ratio, the original hit rates without differential control provides a good reference for setting the QoS goal.

	Overall	Class IMAGE	Class TEXT	Class APP	Class Others
2-class	0.248	0.170	0.318		
4-class	0.252	0.173	0.276	0.581	0.253

Table 4-3 Average Hit Rates without Differential Control (2-class and 4-class)

In Section 4.3.2, we set these original relative hit ratios as the proxy's QoS goals. In Section 4.3.3, we slightly tune the original relative hit ratios to further favor the top class(es). In Section 4.3.4, we set the proxy's QoS goal against its original relative hit ratio.

4.3.2 Setting the Averaging Factor

2-class Experiment

The QoS goal of this experiment is set to be approximately the original relative hit ratio, 0.17:0.32, which normalizes to 0.35:0.65. Figure 4-3 shows the performance of our adaptive model with different hit rate averaging methods, with value λ varying from 1/10 to 1/100, compared with the performance of an un-controlled proxy. Figure 4-4 plots the average weighted deviations ($w\_deviation_t$) of the models. We can see that with a smaller λ, the weighted deviation becomes smoother, reflecting a tuned differentiation in accordance with the long-term trend of the workload. However, when λ is too small, for instance 1/100, the adaptation is retarded, and the weighted deviation increased. When λ is too big, for instance 1/10, the adaptor fluctuates with over-reaction, the weighted deviation increase as well. The average weighted deviation reaches its minimum value with λ=1/60, at about 0.061, which is about 36 percent lower than that of the un-controlled model, or 0.099.

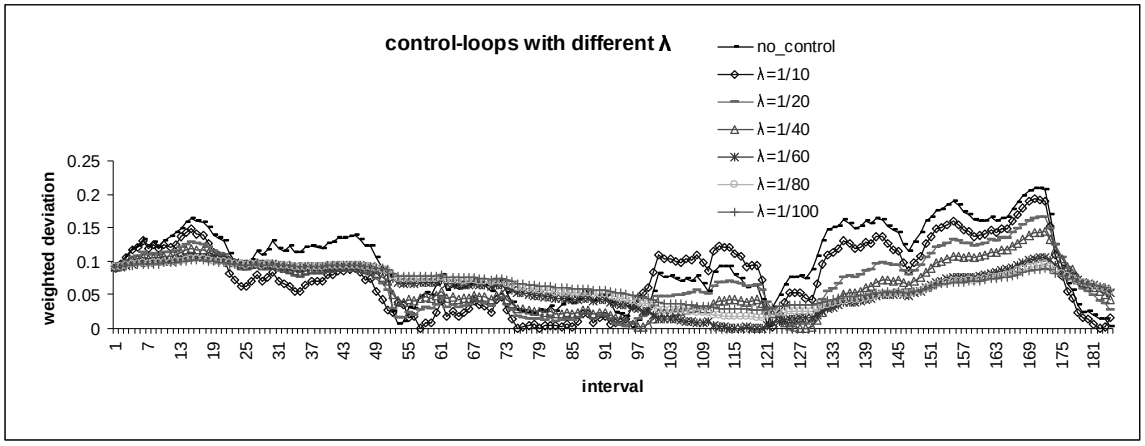

Figure 4-3 Performance of Adaptors with Different Averaging Factor λ (2-class)

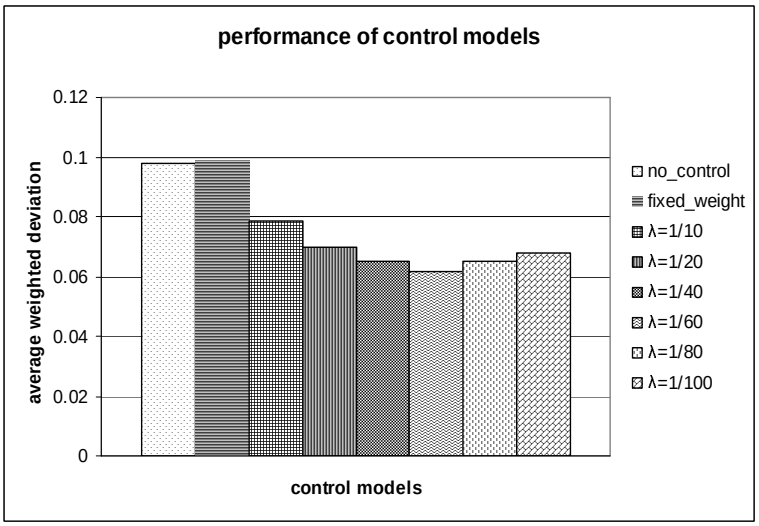

Figure 4-4 Average Weighted Deviations of Adaptors with Different Averaging Factor λ (2-class)

Figure 4-5 plots our adaptive model with λ=1/60 together with the un-controlled model and the fixed-weight model. Our model outperforms the other two in achieving the QoS goal. The smaller weighted deviation shows that it has a closer relative hit ratio to the pre-set QoS goal than the other two models. The fixed-weight model does not show much efficiency in differentiating the classes, but has a similar performance as the un-controlled model. This can be explained by the limitation of its cache space allocation method, which sets the space for each class equal to its desired relative hit ratio and never

adjusts, resulting in a similar effect with the un-controlled model in both the quality of the space allocation and the performance.

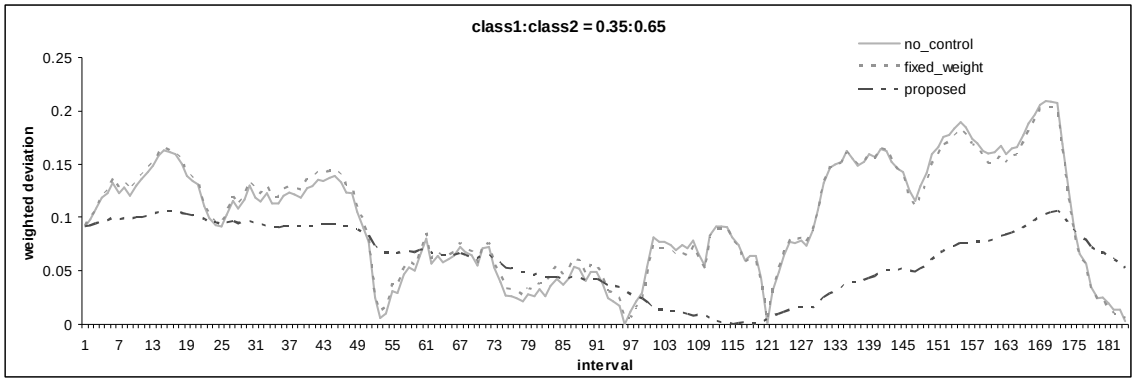

Figure 4-5 Performance of the Models Pursuing Original Relative Hit Ratio (2-class)

4-class Experiment

In this experiment on the four classes, the QoS goal is set to be approximately the original relative hit ratio, 0.17:0.27:0.58:0.25. Figure 4-6 shows the performance of the adaptor with different hit rate averaging methods, with value λ varying from 1/10 to 1/100, compared with the performance of an un-controlled proxy. Figure 4-7 plots the average weighted deviations of the models. We see similar trends of weighted deviation as in the 2-class experiment as we vary the value of λ, and the average minimum is achieved with λ=1/80. The advantage of our adaptive model in achieving the QoS goals in the 4-class case is about 5 percent, which is minor compared to the benefit in the 2-class case. This is expected with the update method of the class weights, as discussed in the implementation of the adaptor (Section 4.1.2). When the number of classes increases, the deviation of each class weight increases in the calculation, resulting in a bigger deviation from the QoS goal.

Figure 4-8 plots weighted deviation of the three models from the pre-set QoS goal. Although they have similar average weighted deviations, our adaptive model differs from

the other two with a more smooth and stable weighted deviation, in spite of the fluctuation in the workload variation, which shows its effect on weighted deviations of the other two models.

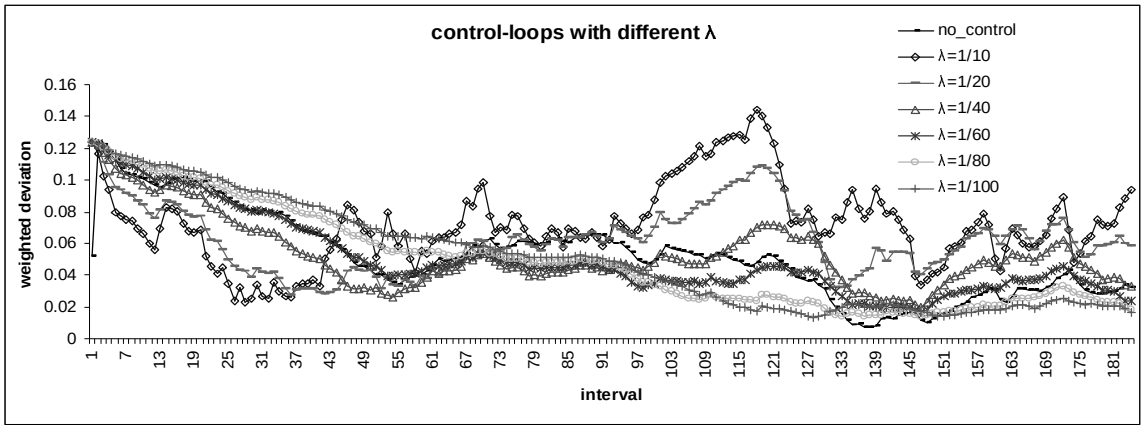

Figure 4-6 Performance of the Adaptor with Different Averaging Factor λ (4-class, original goal)

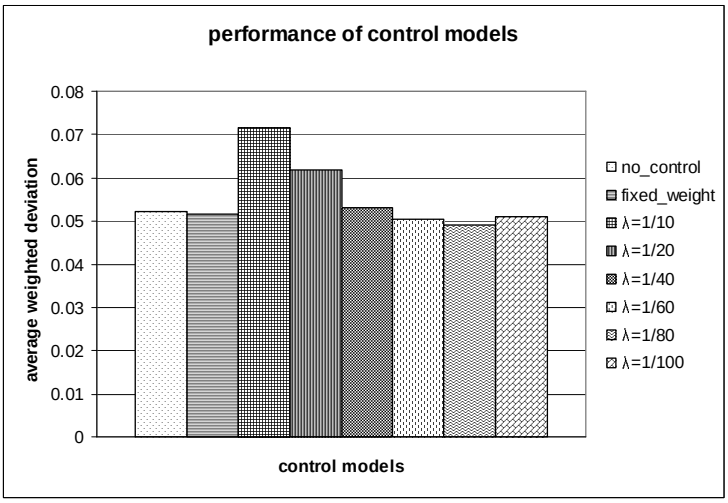

Figure 4-7 Average Weighted Deviations of the Adaptor with Different Averaging Factor λ (4-class)

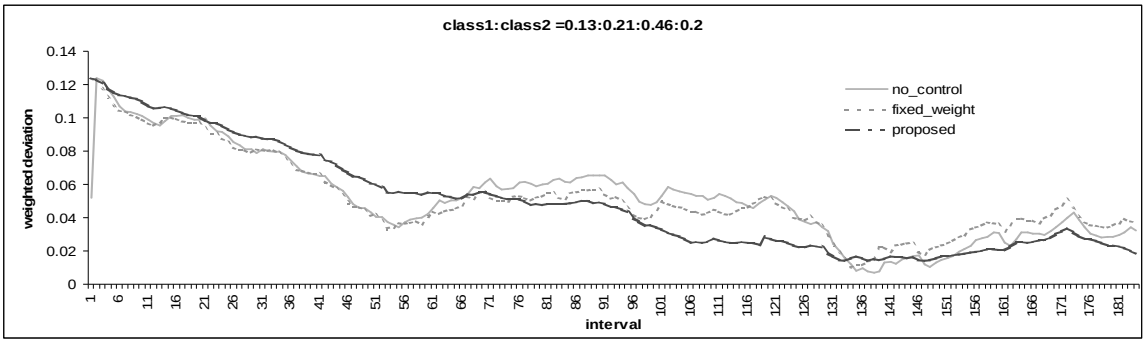

Figure 4-8 Performance of the Models Pursuing Original Relative Hit Ratio (4-class, original goal)

Besides weighted deviation, which measures the differentiated services given to classes, the average overall hit rate and class hit rates are also important in evaluating the differential models. Table 4-4 lists the average hit rates of the three models in both cases.

For both cases, the average overall hit rate of the adaptor is slightly decreased compared to the un-controlled model. However, the higher priority class has an increased average hit rate along with the impaired average hit rate of the lower priority class(es), which results in a relative hit ratio among classes that approximates the QoS goal.

		Overall	Class IMAGE	Class TEXT	Class APP	Class Others
2-class	No-control	0.248	0.170	0.318		
	fixed-weight	0.248	0.169	0.319		
	λ =1/60	0.236	0.181	0.303		
4-class	No-control	0.252	0.173	0.276	0.581	0.253
	fixed-weight	0.246	0.168	0.273	0.586	0.242
	λ =1/60	0.236	0.156	0.251	0.590	0.252

Table 4-4 Hit Rates in Pursuing the Original Relative Hit Ratio (2-class & 4-class)

4.3.3 Adaptive Model with a Reasonable Differential Goal

In this set of experiments, we further tune the relative hit ratio slightly in favor of the higher priority class(es) and evaluate the models' efficiency in approximating the new QoS goal. For the 2-class case, the relative hit ratio goal between Class 1 and Class 2 is set to be 0.3:0.7. For 4-class case, it is set to be 0.11:0.22:0.52:0.15.

2-class Experiment

Figure 4-9 and Figure 4-10 show the weighted deviations of the three models and their averages with the two classes. Again, our adaptive model outperforms the other two with gentle curves and smaller values for weighted deviation. The gain in approaching the specified QoS goal is quite significant, as shown in Figure 4-9. The weighted deviation of the no-control model averages about 0.067. The average weighted deviation of the fixed-weight model is about 7 percent smaller than that of the no-control model, while the weighted deviation of our adaptive model averages at about 0.036, about 46 percent smaller than that of the no-control model.

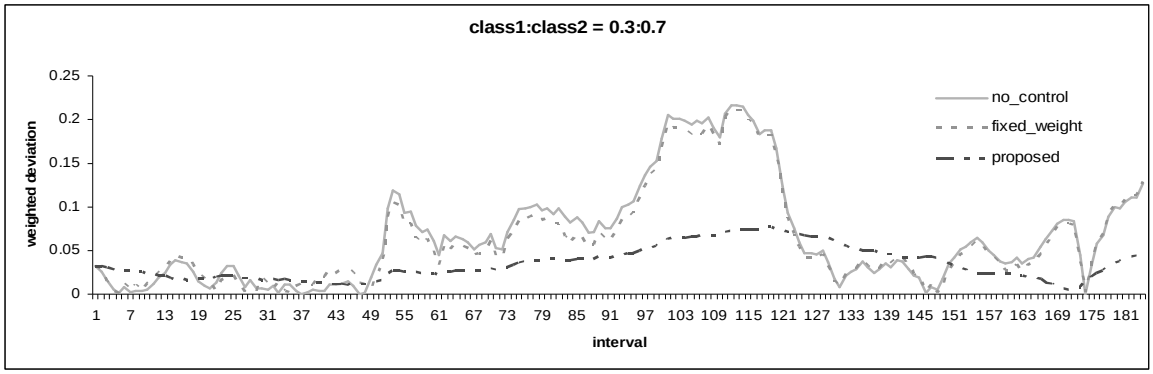

Figure 4-9 Performance of the Models Pursuing a Slightly Tuned Differential Goal (2-class 0.3:0.7)

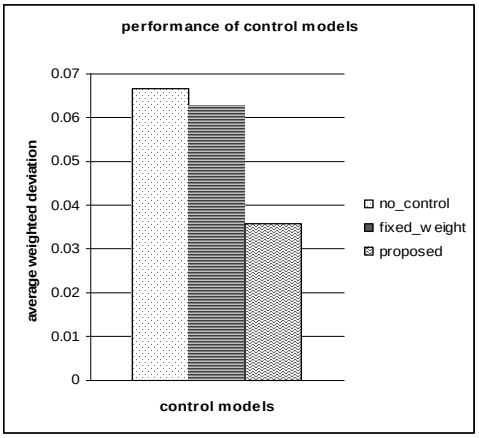

Figure 4-10 Average Weighted Deviations of the Models (2-class 0.3:0.7)

4-class Experiment

Figure 4-11 and Figure 4-12 show the weighted deviations of the three models and their average values respectively. As can be seen, the average weighted deviations in this case are considerably bigger, to the scale of 0.1, than in the 2-class case, where they are around 0.05. The gain of our model is relatively less than the 2-class case as well, averaging at about 22.4 percent smaller than that of the no-control model. However, the trend of each of the three models is similar to what is shown in the previous experiments.

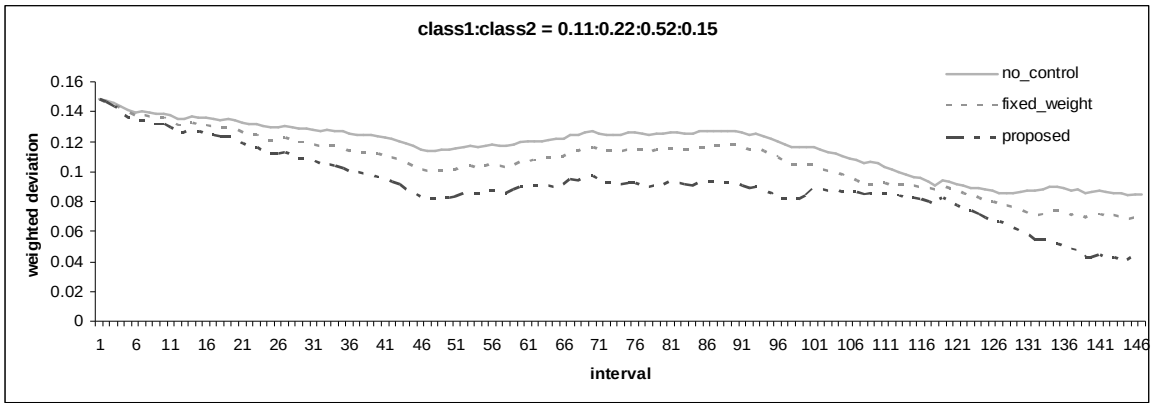

Figure 4-11 Performance of the Models Pursuing a Slightly Tuned Differential Goal (4-class 0.11:0.22:0.52:0.15)

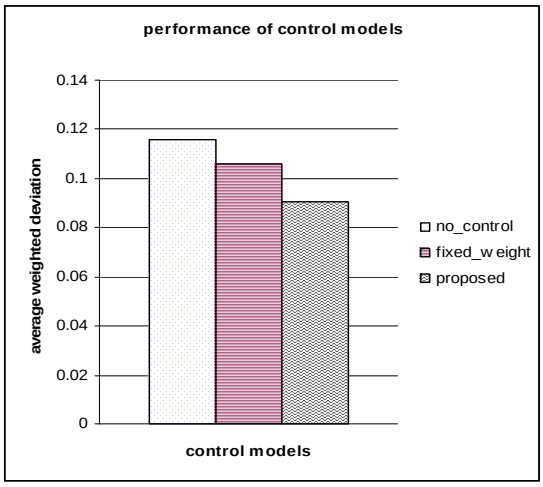

Figure 4-12 Average Weighted Deviations of the Models (4-class 0.11:0.22:0.52:0.15)

Table 4-5 lists the average hit rates of each model in both 2-class and 4-class cases.

In the 2-class case, both DiffServ models increase the average hit rate of the top class by compromising that of the bottom class, due to the differential goal. Our model differentiates the two classes even more than the fixed-weight model. However, the gain in top class's hit rate is smaller than the loss in bottom class, and the overall average hit rate is lowered a little bit.

		Overall	Class IMAGE	Class TEXT	Class APP	Class Others
2-class	**no-control**	0.248	0.170	0.318		
	fixed-weight	0.247	0.167	0.319		
	λ =1/60	0.244	0.159	0.321		
4-class	**no-control**	0.252	0.173	0.276	0.581	0.253
	fixed-weight	0.246	0.168	0.273	0.585	0.237
	λ =1/60	0.236	0.152	0.231	0.534	0.222

Table 4-5 Average Hit Rates in Pursuing the Slightly Tuned Relative Hit Ratio (2-class & 4-class)

In the 4-class case, both DiffServ models approach the QoS goal with a lowered average overall hit rate. However, our model seems to have a slightly worse performance in the measure of average hit rates, although they are adjusted to approach the desired proportion. This may be caused by the complexity of adjustment among four classes, and also, the pre-set QoS goal may still be quite far away from the natural hit ratio of the un-controlled model, unlike what it seems to be.

4.3.4 Adaptive Model with an "Unrealistic" Differential Goal

In this set of experiments, we change the relative hit ratio against its original ratio to observe the effect on the models' performance. For the 2-class case, the relative hit ratio goal is set to 0.7:0.3, and for 4-class case, it is set to 0.41:0.17:0.17:0.25. The original higher performance class(es) has a lower desired hit rate proportion in the QoS goal, while original lower performance class(es) aims at higher hit rate(s).

2-class Experiment

Figure 4-13 and Figure 4-14 show the weighted deviations and their average values respectively. We can see that due to the conflict between the QoS goal and the workload's natural hit ratio, the weighted deviations are around 0.34, which is much higher compared with those of the experiments with original hit ratio as the QoS goal. Both the two DiffServ models have a relatively smaller weighted deviation, about 3 percent better for the fixed-weight model, about 5 percent better for our model than the un-controlled one.

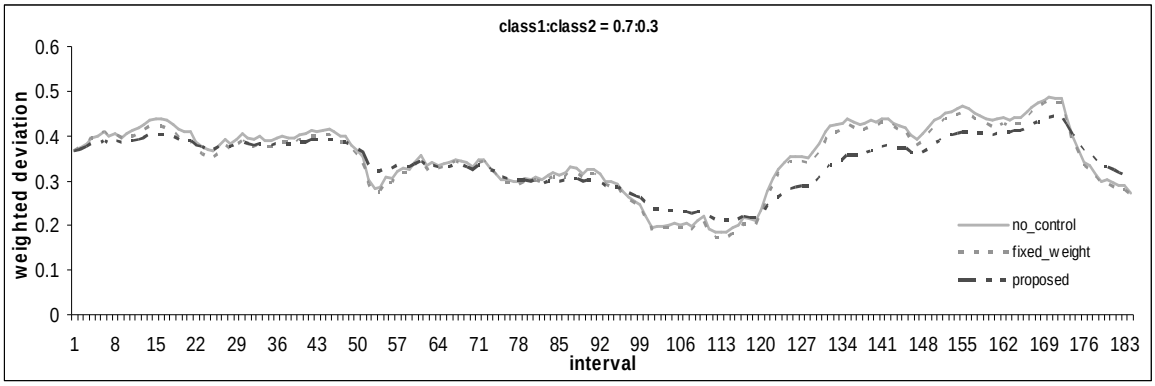

Figure 4-13 Performance of the Models Pursuing an "Unrealistic" Differential Goal (2-class)

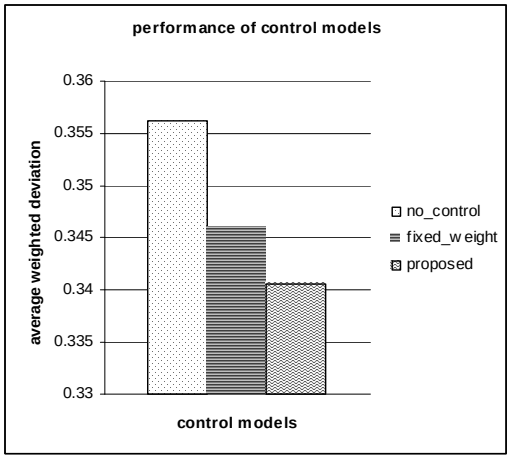

Figure 4-14 Average Weighted Deviations of the Models (2-class 0.7:0.3)

4-class Experiment

As shown in Figure 4-15 and Figure 4-16, the weighted deviations in this case are also big, around 0.33. The gain of the fixed-weight model and our model in the sense of

average weighted deviation are 1 and 4 percent, respectively.

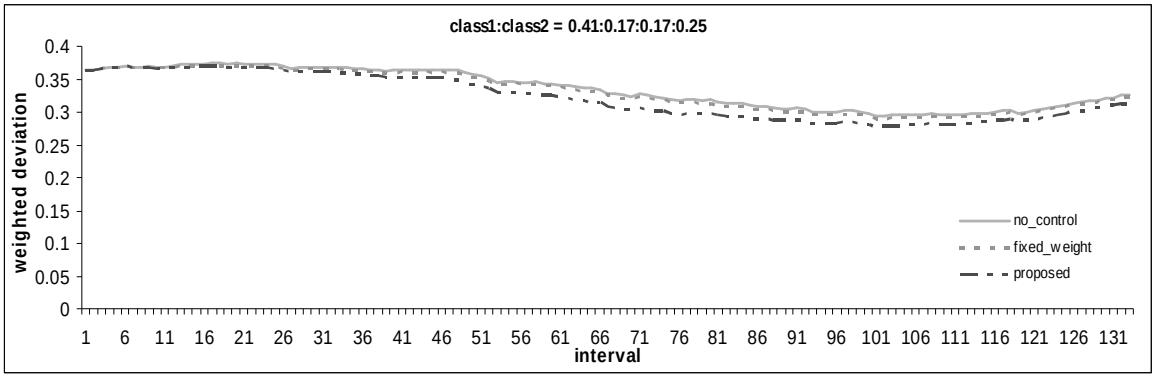

Figure 4-15 Performance of the Models Pursuing an "Unrealistic" Differential Goal (4-class 0.41:0.17:0.17:0.25)

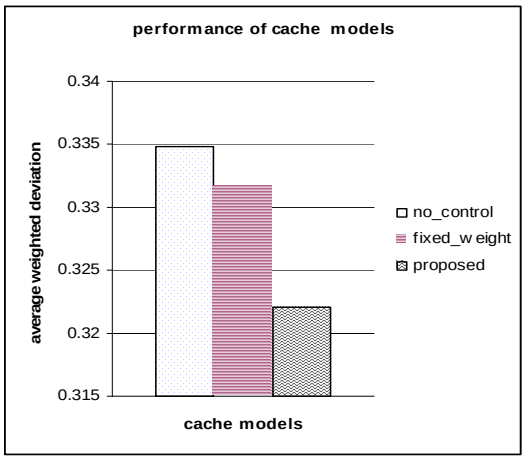

Figure 4-16 Average Weighted Deviations of the Models (4-class 0.41:0.17:0.17:0.25)

Table 4-6 lists the average hit rates of the three models. In the 2-class case, both of the DiffServ models try to lower the average hit rate of the top class and raise that of the bottom class to achieve the "unrealistic" QoS goal, our model having a further adjustment than does the fixed-weight model, and both maintain an average overall hit rate almost unchanged.

In the 4-class case, however, our model largely increases the hit rate of Class 4 ("Others"), and lowers those of the other three classes to approach the desired hit ratio. Class 2 ("Text") and Class 3 ("Application") are considerably decreased in hit rate to get

a low proportion in the hit ratio. The average overall hit rate is thus decreased 13 percent from that of the uncontrolled model.

		Overall	Class IMAGE	Class TEXT	Class APP	Class Others
2-class	**no-control**	0.248	0.170	0.318		
	fixed-weight	0.249	0.176	0.314		
	λ =1/60	0.249	0.178	0.312		
4-class	**no-control**	0.252	0.173	0.276	0.581	0.253
	fixed-weight	0.251	0.176	0.270	0.579	0.251
	λ =1/60	0.220	0.151	0.222	0.503	0.268

Table 4-6 Average Hit Rates in Pursuing the "Unrealistic" Relative Hit Ratio (2-class & 4-class)

4.3.5 Discussion and Summary

In this chapter we discussed the design of a simple adaptive model that dynamically adjusts cache space among classes to achieve pre-set QoS goals specified in terms of a relative hit ratio. Results of the simulations show that:

- Our adaptive model succeeds generally in achieving the QoS goal with multiple classes, with the smallest weighted deviation among the three models tested in our simulation. It provides preferable service differentiation in situations when the workload pattern approaches or departs from the QoS goal. When the natural hit ratio of the workload among classes are similar to the pre-set QoS goal, the average deviation of our adaptive model can reach up to about 36 percent smaller than the uncontrolled model and the fixed-weight model. Even in the case where the natural hit ratio of the workload departs from the pre-set QoS goal, our adaptive model maintains a lower average weighted deviation than the other two models. Since the workload pattern changes over time, and the pre-set QoS goal is inevitably inaccurate in some periods, a generally low deviation is preferable.

- Our model also simplifies the computation of weight updates in cases with more classes so that only one iteration, for all the classes is needed. However, the performance in terms of both average weighted deviation and hit rates are inferior compared with that in 2-class cases. This is cased by the approximate adjustments among classes that is not as accurate as 2-class cases.
- In pursuing the QoS goal, the hit rates of our model are lowered to some degree, as the cost of service differentiation. Especially in experiments with an "unrealistic" QoS goal on more than two classes, the hit rates are considerably lower.
- We compare the performance of three models, which all run webLRU-2 as the caching policy. This may have been one of the major reasons that the differences between our model and the other two are not large.
- The fixed-weight model can be considered as a static DiffServ model. Once the weights of the classes are pre-set according to the desired hit ratio, they never change in the course of caching, no matter how the workload changes. Thus, this model is simple, and inefficient in achieving a differential service goal. On the other hand, our model makes replacement decisions dynamically, and suffers from delay and inaccuracy in the adaptor.

Another issue is the setting of the QoS goal, especially in cases with more classes, which complicated the setting of relative hit ratio. Although it is inevitably inaccurate, a QoS goal set by an experienced administrator is essential to the performance of the proxy.

Chapter 5

Conclusions and Future Work

The limited storage space on proxies and Web servers is a crucial resource for guaranteed differential QoS in Web caching. In much of the research in this field, the main challenges come from the overhead, adaptability, and complexity in implementation of the caching algorithm, and portability and the computational complexity of the QoS control scheme. In this thesis we design and implement a simple, self-adaptive, and portable differentiated Web caching scheme for proxy servers.

We adopt the LRU-K algorithm from DBMSs' buffer management system and combine it with the LFU algorithm, based on an analysis on general request streams to a proxy server. The proposed algorithm, webLRU-2, takes into account both long-term workload characteristics at a proxy server and short-term fluctuation in the access pattern.

We design an adaptive DiffServ model that adopts the webLRU-2 algorithm in pursuing a differential QoS for cache space management. The adaptive model is designed to dynamically adjust class weights based on the feedback on hit rate deviation of each class. The model easily adjusts among 2 or more classes.

We implement a proxy simulator to evaluate caching policies including LRU, LRU-K, Perfect-LFU, In-Cache-LFU and webLRU-2. The adaptor is added to the simulator, together with the webLRU-2 algorithm on per-class queues. Through

trace-driven simulations on both synthetic Zipf-like workloads and real-world workloads from NLANR's proxy servers, we show that webLRU-2 has a generally higher hit rate than LRU, LRU-K, and In-Cache-LFU, approaching to that of Perfect-LFU. Also, with the adaptive scheme, the proxy obtains preferable differential QoS with a smaller weighted deviation from its proportional QoS goal, compared with the uncontrolled model and the fixed-weight model.

5.1 Conclusions

Based on the experiment results on both webLRU-2 and our adaptive model, we come to the following conclusions:

- The webLRU-2 algorithm maintains frequency information of the subset of both long-term and short-term popular objects with a reasonable book-keeping overhead, while being adaptive to workload changes. webLRU-2 generally approaches Perfect-LFU in terms of hit rate, and outperforms LRU-2 and In-Cache-LFU especially on relatively small cache sizes, around 10% of the total file size of our simulation workloads, which is reasonable for real proxies.
- Our adaptive model with webLRU-2 on multiple class queues provides portable differentiated services. With a reasonably pre-set QoS goal, it obtains a weighted deviation about 36 percent lower than those of the un-controlled model and the fixed-weight model. Even in case with an "unrealistic" pre-set QoS goal that departs from the natural hit rate proportion among classes, although the efficiency of the adaptive model is degraded, especially in cases with more than two classes, it is still the best differential model among the three.
- The proportional QoS goal facilitates portable differentiation over platforms,

however, it needs to be set by an experienced administrator so that the goal is approached at a reasonable cost, in terms of overall performance.

- The synthetic workloads in our simulations characterize the Zipf-like distribution of objects over time, but ignore the spatial locality and short-term correlation of locality in the real proxy workloads. This may have become a drawback in the evaluation of our caching scheme.

5.2 Future Work

Our work in this thesis suggests several potential topics for further study:

- Size factor in webLRU-2

To better evaluate webLRU-2's ability to capture reference locality in the workloads, we left out the size factor in our algorithm design. However, the size factor has been shown to benefit Web caching algorithms in previous studies [5][14][31]. We can add it to webLRU-2 and set the proper value for size threshold.

- Adjusting method for multiple classes

Our adaptive model on multiple levels of services provides a simple and fast adjustment to class weights. However, for QoS goals with more than 2 classes, we can continue our work to better tune the proportion of class weights, given the feedback on the class hit rates.

- Reference locality in synthetic workloads

The workload generator could be further developed to better approximate temporal and spatial locality in network traffic [8][35], so that more accurate evaluation of the performance of caching algorithms could be obtained.

- Compatibility with Web servers

Finally, although our work is based on proxy servers, it could be extended to run on Web servers and reverse proxy servers. The difference lies mainly in the workload patterns, with different skews in the long-term popularity [13], so the design of webLRU-2 and the idea of adaptive differentiation of caching should still apply, though some parameters, including the method of *Retain Information Period* may need to change.

References

[1] Akamai: http://www.akamai.com

[2] NLANR (National Laboratory for Applied Network Research) traces: ftp://ftp.ircache.net/Traces/

[3] The Wireless Developer Network: www.wirelessdevnet.com

[4] T. F. Abdelzaher and N. Bhatti. Web server QoS management by adaptive content delivery. In *International Workshop on Quality of Service*, London, UK. Jun 1999.

[5] G. Abdulla, E. A. Fox, M. Abrams and S. Williams. Www proxy traffic characterization with application to caching. Technical Report, Computer Science Department Virginia Technology, (CS-97-03):1--20, Mar 1998.

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

[7] J. Almeida, M. Dabu, A. Manikutty and P. Cao. Providing Differentiated Levels of Service in Web Content Hosting. In *Workshop on Internet Server Performance*, Madison, WI. Jun 1998.

[8] V. Almeida, A. Bestavros, M. Crovella and A. Oliveira. Characterizing reference locality in the WWW. In *IEEE International Conference in Parallel and Distributed Information Systems*, Miami Beach, FL, USA, Dec 1996.

[9] Ismail Ari, Ahmed Amer, Robert Gramacy, Ethan L. Miller, Scott A. Brandt, and D. E. Long, Who is more adaptive? ACME: adaptive caching using multiple experts, *Workshop on Distributed Data and Structures (WDAS 2002)*, Paris, France, Mar 2002.

[10] P. Barford, A. Bestavros, A. Bradley, and M. Crovella. Changes in Web Client Access Patterns: Characteristics and Caching Implications. *World Wide Web,* 2(1):15-28,

Jan 1999.

[11] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. *IETF RFC 2475*, Dec 1998.

[12] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. *IETF RFC 1633*, Jun 1994.

[13] L. Breslau, P. Cao, L. Fan, G. Philips, and S. Shenker, Web caching and Zipf-like distributions: evidence and implications, In *Proceedings of Infocom,* 1999.

[14] P. Cao, J. Zhang, and K. Beach, Cost-aware WWW proxy caching algorithms, In *Proc. of the USNIX Symposium on Internet Technologies and Systems (USITS '97)*, 193-206, Dec 1997.

[15] X. Chen and P. Mohapatra. Providing differentiated service from an Internet server. In *IEEE Int'l Conf. on Computer Communications and Networks*, Boston, MA. Oct. 1999.

[16] W. Chen, P. Martin and H. S. Hassanein. Differentiated Caching of Dynamic Content using Effective Page Classification. In *Proc. of the 23rd IEEE International Performance, Computing, and Communications Conference (IPCCC '04)*, 293-298, Phoenix, AZ. Apr 2004.

[17] L. Cherkasova. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy, *Technical Report* HPL-98-69R1, Hewlett-Packard Laboratories, Nov 1998.

[18] J. Chuang, K. Hosanagar, and R. Krishnan. Pricing Caching Services with multiple levels of QoS. In *Proc. of the IEEE Conference on Systems, Man and Cybernetics*, Tucson, AZ. Oct 2001.

[19] J. Chuang and M. Sirbu. Distributed Network Storage with Quality-of-Service

Guarantees. *Journal of Network and Computer Applications* 23(3): 163-185, July 2000.

[20] J. C.-I. Chuang and M. A. Sirbu, Stor-serv: Adding quality-of-service to network storage, In *Proc. of Workshop on Internet Service Quality Economics*, Cambridge MA. Dec 1999.

[21] M.E. Crovella and R.L. Carter, Dynamic server selection in the Internet. In *Proc. of the 3rd IEEE Workshop on the Architecture and Implementation of High Pref. Comm. Subsystems* (HPCS'95), 158-162. Aug. 1995.

[22] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proc. of ACM SIGCOMM*, Sep 1998.

[23] M. Feldman and J. Chuang. Service Differentiation in Web Caching and Content Distribution. http://citeseer.nj.nec.com/554650.html.

[24] S. Jin and A. Bestavros. Popularity-Aware GreedyDual-Size Algorithm for Web Access. In *Proc. Of IEEE ICDCS'00,* April, 2000. Computer Science Technical Report BUCSTR-1999-009, Boston University.

[25] S. Jin and A. Bestavros. GreedyDual* Web Caching Algorithm: Exploiting the Two Sources of Temporal Locality in Web Request Streams. In *Proc. of the 5th International Web Caching Workshop, 2000. Computer Communication (Elsevier) special issue*, 24(2), 2001.

[26] T. P. Kelly, Y. M. Chan, S. Jamin and J. K. MacKie-Mason Biased Replacement Policies for Web Caches: Differential Quality-of-Service and Aggregate User Value. In *Proc. of the 4th International Web Caching Workshop,* San Diego, CA, Mar 1999.

[27] T. P. Kelly, S. Jamin, and J. K. MacKie-Mason. Variable QoS from Shared Web Caches: User-Centered Design and Value-Sensitive Replacement. In *Proc. of the MIT Workshop on Internet Service Quality Economics (ISQE 99),* Cambridge, MA. Dec 1999.

[28] P. Krishnan, D. Raz, and Y. Shavitt. The cache location problem. *IEEE/ACM Transactions on Networking,* 8(5): 568-582. Oct 2000.

[29] B. Li, M. J. Golin, G. F. Italiano, X. Deng, and K. Sohraby. On the optimal placement of web proxies in the Internet. In *Proceedings of the IEEE Infocom 1999,* 1282-1290, NY, USA, Mar 1999.

[30] Z. Liang, H. Hassanein, P. Martin, Transparent distributed Web Caching. In *Proc. Of the IEEELocal Computer Network Conference,* 225-233. Nov 2001.

[31] P. Lorensetti, L. Rizzo, L. Vicisano. Replacement Policies for Proxy Cache. Manuscript, 1997.

[32] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An Adaptive Control Framework for QoS Guarantees and its Application to Differentiated Caching Services. In *10th IEEE Int. Workshop on Quality of Service,* 23-32. May 2002

[33] A. Myers, J. C.-I Chuang, U. Hengartner, Y. Xie, W. Zhuang, and H. Zhang. A Secure, Publisher-Centric Web Caching Infrastructure. In *Proc. of IEEE INFOCOM 2001*, Anchorage AL. Apr 2001.

[34] E. J. O'Neil, P. E. O'Neil and G. Weikum, The LRU-K Page Replacement Algorithm for Database Disk Buffering, In *Proc. of ACM SIGMOD Int. Conference on Management of Data,* 1993

[35] V. Paxson, Fast Approximation of Self-Similar Network Traffic. Technical Report LBL-36750, Lawrence Berkeley Lab, Berkeley, CA, 1995.

[36] M. Rabinovich, J. Chase, and S. Gadde. Not All Hits Are Created Equal: Cooperative Proxy Caching Over a Wide-Area Network. In *Proc. Of the 3rd Int. WWW Caching Workshop.* Jun 1998.

[37] M. Rabinovich, O. Spatscheck. Web Caching and Replication. *Addison Wesley*, Dec

2001.

[38] P. Rodriguez, C. Spanner, and E. Biersack, Web caching architectures: Hierarchical and distributed caching. In *Proc. of the 4th Int. Web Caching Workshop*, Apr 1999.

[39] A. Rousskov and D. Wessels. Cache Digests. In *Proc. of the 3rd Int'l WWW Caching Workshop*, Manchester, England. Jun 1998.

[40] M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek, Selection Algorithms for Replicated Web Servers. In *Performance Evaluation Review*, 26(3): 44-50, Dec 1998.

[41] E. Shriver, E. Gabber, L. Huang and C. Stein, Storage Management for Web Proxies, In *Proc. of the 2001 USENIX Annual Technical Conference*, Boston, MA, June, 2001.

[42] R. Tewari, M. Dahlin, H. Vin and J. Kay. Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet. Technical Report TR98-04, Department of Computer Sciences, University of Texas at Austin, Feb 1998.

[43] J. Wang. A Survey of Web Caching Schemes for the Internet. *ACM Computer and Communication Review*, 36-46, Oct 1999.

[44] D. Wessels and K. Claffy. Application of Internet Cache Protocol (ICP), version 2. IETF *RFC 2187.* Sep 1997.

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

[46] R. Wooster, M. Abrams. Proxy Caching that Estimates Page Load Delays. In *Proc. Of 6th International WWW Conference*, Santa Clara, CA, April 1997.

[47] J. Zhou, P. Martin, and H. Hassanein. QoS Differentiation in Switching-based Web Caching. In *Proc. of the 23rd IEEE International Performance, Computing, and Communications Conference (IPCCC '04)*, 453-460, Phoenix, AZ. Apr 2004.

[48] Q. Zou, P. Martin and H. S. Hassanein. Transparent Distributed Web Caching with Minimum Expected Response Time. In *Proc. of the IEEE International Performance, Computing, and Communications Conference (IPCCC '03)*, Phoenix, AZ. Apr 2003.

Appendix A

Pseudo-code of LRU-k and webLRU-2

Notations

HIST(p) denotes the history control block of object p; it contains the time stamps of the k most recent references to object p, with HIST(p,1) denoting the latest reference. Correlated references are discounted.

Correlation_Timeout denotes the constant of correlation reference period.

Retain_Info_Period denotes the constant of retain information period.

LRU-K: on request for object p at time t

```
/* scan cache queue to see if p is already in cache */
q := the object at queue end
hit := false
While (q != null) do
    If (q.url equals p.url) then       // hit
        hit := true
        break
    Endif
    q := next object before q
Enddo
If (hit) then              // hit
    /* update history information of p */
    If (t-HIST(p,1)> Correlation_Timeout) then      // a new, uncorrelated reference
        For i =2 to K do
            HIST(p,i) = HIST(p,i-1)
        Endfor
```

```
        HIST(p,1) = t
    Else                // a correlated reference
        HIST(p,1) = t
    Endif
    hits += 1
Else                    // miss
    /* select replacement victims */
    q = the object at the Cache Queue end
    While (Free Space < p.size) do
        If (t-HIST(q,1) > Correlation_Timeout) then     // eligible for replacement
            evict victim q from cache
            Free Space += q.size
            put HIST(q) into the Evict Table
        Endif
        q = next object before q          // object with next max Backward K-distance
    Enddo
    /* cache the referenced object*/
    fetch p into the cache and append p at the end of Cache Queue
    Free Space -= p.size
    misses += 1
    check the Evict Table for object p
    If (p does not exist) then              // initialize history control block
        allocate HIST(p)
        For i := 2 to K do HIST(p,i) := 0
    Else
        retrieve stored HIST(p)
        For i = 2 to K do HIST(p,i) = HIST(p,i-1)
    Endif
    HIST(p,1) = t
Endif
/* Relocate p in the cache queue with its Backward K-distance and HIST(p,1)*/
```

```
q := next object before p
While (q != NULL && HIST(q,K) ≥  HIST(p,K)) do
    q := next object before q
Enddo
If (q == NULL) then move p to Cache Queue top
Else move p into the position after q
```

webLRU-2 (one class): on request for object p at time t

// **FLEV(p)** denotes the frequency level of object p. It is a function of p's number of references, as discussed in Section 3.2.3

```
/* scan cache queue to see if p is already in cache */
q := the object at queue end
hit := false
While (q != null) do
    If (q.url equals p.url) then        // hit
        hit := true
        break
    Endif
    q := next object before q
Enddo
If (hit) then               // hit
    /* update history information of p */
    If (t-HIST(p,1)> Correlation_Timeout) then       // a new, uncorrelated reference
        HIST(p,2) = HIST(p,1)
        HIST(p,1) = t
        p.frequency += 1
    Else                    // a correlated reference
        HIST(p,1) = t
    Endif
```

```
    hits += 1
Else                    // miss
    /* select replacement victims */
    q = the object at the Cache Queue end
    While (Free Space < p.size) do
        If (t-HIST(q,1) > Correlation_Timeout) then      // eligible for replacement
            evict victim q from cache
            Free Space += q.size
            put HIST(q) into the Evict Table
        Endif
        q = next object before q          // object with next max Backward K-distance
    Enddo
    /* cache the referenced object*/
    fetch p into the cache and append p at the end of Cache Queue
    Free Space -= p.size
    misses += 1
    check the Evict Table for object p
    If (p does not exist) then                // initialize history control block
        allocate HIST(p), p
        HIST(p,2) := 0
        p.frequency := 1
    Else
        retrieve stored HIST(p), p
        p.frequency += 1
        HIST(p,2) = HIST(p,1)
    Endif
    HIST(p,1) = t
Endif
/* Relocate p in the cache queue with its Backward K-distance and HIST(p,1)*/
q := next object before p
While (q != NULL &&
```

```
    (FLEV(q) < FLEV(p) || FLEV(q) ≥ FLEV(p) && HIST(q,2) ≥  HIST(p,2))) do
    q := next object before q
Enddo
If (q == NULL) then move p to Cache Queue top
Else move p into the position after q
```

webLRU-2 (classified): on request for object p of class c at time t

```
// FLEV(p) denotes the frequency level of object p. It is a function of p's number of
references, as discussed in Section 3.2.3
// N denotes the number of classes
// weights[0..N-1] denotes the weight for each class

/* scan cache queue to see if p is already in cache */
q := the object at the end of Cache Queue for class c
hit := false
While (q != null) do
    If (q.url equals p.url) then       // hit
        hit := true
        break
    Endif
    q := next object before q
Enddo
If (hit) then              // hit
    /* update history information of p */
    If (t-HIST(p,1)> Correlation_Timeout) then      // a new, uncorrelated reference
        HIST(p,2) = HIST(p,1)
        HIST(p,1) = t
        p.frequency += 1
    Else                   // a correlated reference
        HIST(p,1) := t
    Endif
```

```
    hits += 1
Else                    // miss
    /* select replacement victims */
    For i := 0 to N-1
        o[i] := the object at the end of Cache Queue of class 0
    Endfor
    While (Free Space < p.size) do
        q := o[N-1]
        max := HIST(q, 2)/weights[N-1]/FLEV(q)
        For i := N-1 to 0    //find the victim with maximum key
            While (o[i] != NULL && t-HIST(o[i],1) ≤ Correlation_Timeout) do
                o[i] := next object before o[i]
            Enddo
            If (o[i] != NULL && HIST(o[i],2)/weights[0]/FLEV(o[i]) > max) then
                q := o[i]
                max := HIST(q, 2)/weights[N-1]/FLEV(q)
            Endif
        Endfor
        evict victim q from cache
        Free Space += q.size
        put HIST(q) into the Evict Table
    Enddo
    /* cache the referenced object*/
    fetch p into the cache and attach p at the end of Cache Queue of class c
    Free Space -= p.size
    misses += 1
    check the Evict Table for object p
    If (p does not exist) then              // initialize history control block
        allocate HIST(p), p
        HIST(p,2) := 0
        p.frequency := 1
```

```
    Else
        retrieve stored HIST(p), p
        p.frequency += 1
        HIST(p,2) = HIST(p,1)
    Endif
    HIST(p,1) = t
Endif
/* Relocate p in the cache queue with its Backward K-distance and HIST(p,1)*/
q := next object before p
While (q != NULL &&
    (FLEV(q) < FLEV(p) || FLEV(q) ≥ FLEV(p) && HIST(q,2) ≥  HIST(p,2))) do
    q := next object before q
Enddo
If (q == NULL) then move p to Cache Queue top
Else move p into the position after q
```

Appendix B

Zipf-like Distribution Workload Generator

B.1 Zipf-like Distribution

Zipf's law was originally applied to the relationship between a word's popularity in terms of rank and its frequency of use. It states that if one ranks the popularity of words used in a given text (denoted by ρ) by their frequency of use (denoted by P), then

$$P \sim \rho^{-\beta}$$

with $\beta \approx 1$. More general cases are Zipf-like laws that relate frequency of symbol use to popularity rank via a power-law relationship.

Applied to the Web, Zipf-like distribution states that the relative probability of a request for the i'th most popular page is proportional to $1/i^{\alpha}$, for some constant α between 0 and 1. Zipf's Law is considered as a particular case, with $\alpha = 1$. In a popularity distribution of objects that comforms to Zipf's Law, the most popular Web object is twice as popular as the second most popular object.

Figure B-1 shows a series of Zipf-like distributions with the value of α varying from 0.05 to 1. When $\alpha = 0$, it's a uniform distribution, and objects are receiving equal attention. As α approaches 1, popular objects receive greater fraction of requests.

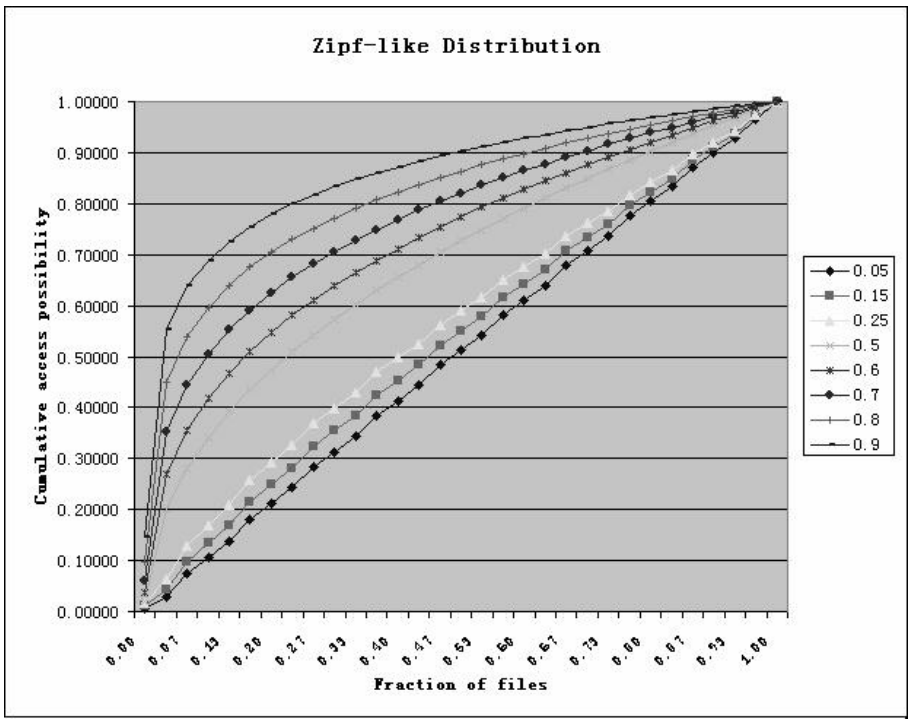

Figure B-1 Zipf-like Distributions

B.2 The Synthetic Workload Generator

The synthetic workload is generated according to the Zipf-like distribution and size/popularity distribution rules. It's generated in three steps. First, a Zipf-like sequence of 300,000 accesses on 5,000 unique files is generated with Zipf-like distribution $P_N(i) = \frac{\Omega}{i^{\alpha}}$, with a certain value of α. Second, the 5,000 unique files are extracted from a real workload from a Web proxy of NLANR, sorted according to their popularity and size. The final step is done when the workload is fed to the simulation model, during the course the Zipf-like sequence and the actual files are combined together.

- Pseudo-code of generator of the Zipf-like distribution access sequence

Main Procedure

```
N := 5000;              //total # of unique files
RN := 300000;           //total # of requests in the workload
Alfa := 0.75;           //the Zipf slope α
seed := 3
r := 0.0
p[N+1] := {0.0…0.0}     //probability array

For i := 1 to N         //individual probability
    p[i] := zipf( i )
Endfor
For i := 2 to N         //accumulative probability
    p[i] += p[i-1]
Endfor
For i := 1 to RN        //generating the access sequence with a random seed
    r := mrand( seed )  //random number between 0 and 1
    If (the random number r falls into accumulative probability p[i]) then
        append an access to file i to the sequence;
    Endif
Endfor
```

Module zipf (in: fileNo)

Local: omega, index

```
omega := 0.0
For index := 1 to N
    omega += 1 / (pow(index, Alfa))
Endfor
omega := 1/omega
Return (omega/(pow(fileNo, Alfa)))
```

Appendix C

Simulator Settings and Pseudo-codes

C.1 Simulation Parameters

The following table lists the parameter in the simulator. Example settings and explanations are provided.

Parameters	Example Values	Notes
Cache space	200	Total cache space the proxy server in MB
Correlation_timeout	300 sec	Timeout for the algorithm to keep any new page after its reference
Retain_info_timeout		Timeout for the algorithm to maintain history information about any page after its most recent access.
set_cls_num	4	Number of classes
set_cls_method	1	Classification method method 1: content category (RequestedObject.category) method 2: content type (RequestedObject.contentType) method 3: client address (RequestedObject.clientAddress) method 4: server address (RequestedObject.serverAddress)
set_cls_keywords	text; image; application;	Classification keywords, and "others" for un-matched objects
set_pp_hitrates	15;4;7;2;	Desired proportional Hitrate C1:C2:C3:...
interval	600	Adapting interval (in seconds)
Min_requests	1000	Minimum requests per adapting interval (in # of requests)
Wtd_deviation	0.10	Desired weighted deviation bound; when weighted sum of class deviation is lower than this value, no adjustment to cache allocation is needed
Max_pp_weight	50	Maximum adapted proportional weight (or memory allocation) between first and last classes (priority w_1: w_n)

Table C-1 Simulation Parameters

Parameters are grouped into four fields: Cache settings to set cache size f the proxy; Caching policy parameters are related to individual caching algorithms run on the proxy; Classification settings assign object classification methods and proportional class goal; Adaptor settings determine how often the adaptor monitors and adjusts cache space allocation among object classes according to feedback from the system.

C.2 Simulator Pseudo-codes and the Classes

```
Simulator
//M: number of caching policies
//policy[0..M-1]: policies run in the simulator

read from configuration file and initialize the simulator
create and run Client Cluster, Adaptor and Output Log
create and run caching policies policy[0..M-1]
While (has more requests) do
    Read in a request r from Client Cluster
    For i :=0 to M-1
        policy[i] on request r // individual caching policies
    Endfor
Enddo
```

```
Adaptor: on feedback with hr[0..N-1]
//p_hr[0..N-1]: the desired proportional hit ratio among classes, normalized with a sum of 1.
//hr[0..N-1]: hit rate of each class
//w_dev: weighted deviation from desired proportional hit ratio p_hr[0..N-1]

sum := 0
w_dev := 0
For i := 0 to N
```

```
    sum += hr[i]
Endfor
For i := 0 to N
    w_dev += weight[i] * (hr[i]/sum – p_hr[i])/p_hr[i]
Endfor
If (w_dev ≤ threshold) then
    For i := 0 to N
        If (p_hr[i] - hr[i]/sum > threshold) then
            weight[i] := weight[i]/(1+ (hr[i]/sum - p_hr[i])/p_hr[i])
        Endif
    Endfor
Endif
normalize weight[0..N-1]
```

Cache Policies:

LRU, Perfect-LFU, In-Cache-LFU, LRU-K (Appendix A), and webLRU-2 (Appendix A)

LRU: on request for object p at time t

//HIST(p) contains the time stamp of the latest reference to object p

```
/* scan cache queue to see if p is already in cache */
q := the object at queue end
hit := false
While (q != null) do
    If (q.url equals p.url) then       // hit
        hit := true
        break
    Endif
    q := next object before q
Enddo
If (hit) then              // hit
    HIST(p) = t
    hits += 1
Else                  // miss
    /* select replacement victims */
```

```
    q = the object at the Cache Queue end
    While (Free Space < p.size) do
        If (t-HIST(q) > Correlation_Timeout) then  // eligible for replacement
            evict victim q from cache
            Free Space += q.size
        Endif
        q = next object before q   //object with the latest access longest back from now
    Enddo
    /* cache the referenced object*/
    fetch p into the cache
    Free Space -= p.size
    misses += 1
    /* initialize history control block */
    allocate HIST(p)
    HIST(p) = t
Endif
/* Relocate p to the top of Cache Queue*/
move p to Cache Queue top
```

Perfect-LFU: on request for object p at time t

// **Retain_Info_Period** denotes the constant of retain information period.

```
/* scan cache queue to see if p is already in cache */
q := the object at queue end
hit := false
While (q != null) do
    If (q.url equals p.url) then       // hit
        hit := true
        break
```

```
    Endif
    q := next object before q
Enddo
If (hit) then              // hit
    p.frequency += 1
    hits += 1
Else                       // miss
    /* select replacement victims */
    q = the object at the Cache Queue end
    While (Free Space < p.size) do
        evict victim q from cache
        Free Space += q.size
        put q into the Evict Table
        q = next object before q    // object with the next least accesses
    Enddo
    /* cache the referenced object*/
    fetch p into the cache and append p at the end of Cache Queue
    Free Space -= p.size
    misses += 1
    check the Evict Table for object p
    If (p does not exist) then
        p.frequency := 1
    Else
        retrieve stored p
        p.frequency += 1
    Endif
Endif
/* Relocate p in the cache queue with its frequency*/
q := next object before p
While (q != NULL && q.frequency ≤ p.frequency) do
    q := next object before q
```

```
Enddo
If (q == NULL) then move p to Cache Queue top
Else move p into the position after q
```

In-Cache-LFU: on request for object p at time t

// **Retain_Info_Period** denotes the constant of retain information period.

```
/* scan cache queue to see if p is already in cache */
q := the object at queue end
hit := false
While (q != null) do
    If (q.url equals p.url) then        // hit
        hit := true
        break
    Endif
    q := next object before q
Enddo
If (hit) then                // hit
    p.frequency += 1
    hits += 1
Else                    // miss
    /* select replacement victims */
    q = the object at the Cache Queue end
    While (Free Space < p.size) do
        evict victim q from cache
        Free Space += q.size
        q = next object before q    // object with the next least accesses
    Enddo
    /* cache the referenced object*/
    fetch p into the cache  and append p at the end of Cache Queue
```

```
    Free Space -= p.size
    misses += 1
    p.frequency := 1
Endif
/* Relocate p in the cache queue with its frequency*/
q := next object before p
While (q != NULL && q.frequency ≤ p.frequency) do
    q := next object before q
Enddo
If (q == NULL) then move p to Cache Queue top
Else move p into the position after q
```

Class *AdminConsole*

- Initializes the simulation with parameter settings, starts the whole model.

Class *WebProxy*

- Initializes the proxy with input (ClientCluster), output (LogOutput), cache policies (CachePolicy) and adaptor (CacheAdaptor).

- Divides request sequence into sections (or time intervals) and synchronizes the policies and the adaptor.

Class *CachePolicy*

- Caching policies include LRU, LRU-K, webLRU-2, Perfect-LFU, In-Cache-LFU. Each policy establishes and maintains cache queues (Class CacheQueue) for in-cache objects, in accordance to its placement/replacement algorithm. webLRU-2 on classified workloads maintains multiple cache queues.

- For policies that retain history information, evict tables are maintained to hold the evicted objects for a certain time. These policies include LRU-K, webLRU-2 and

Perfect-LFU (Perfect-LFU is exceptional that all the evicted objects are kept without a time-out).

- Statistic data are collected at the end of each section, and an average is calculated for each class when all the sections finish.

Class *CacheQueue*

- As a key component of the cache policy, maintains cached objects in one or multiple queues in a sequence of caching key.

Class *CacheAdaptor*

- Monitor the performance of each caching policy against pre-set class goals and deviation thresholds, and makes adaptation to class weights at the end of each caching section, if the performance deviation exceeds pre-set thresholds,.

Class *ClientCluster*

- Initializes the client cluster with log file as input. Read from the log file, and parse it into a request to the proxy.
- Eliminates un-cacheable objects.
- The log file can be either a real trace or a synthetic workload.

Class *RequestedObject*

- The classified requests parsed from a log file and sent to the proxy. The structure of a real workload request is described in Section 3.3.2. The structure of a synthetic workload is simplified mostly due to ignorance of client and detailed request information: it's simply a sequence of access to file IDs with the sequence number as the time stamp.

Class *CachedObject*

- Extended from Class RequestedObject, this class holds access records, access

frequency, calculated key of an in-cache object, and also, as cached objects are stored in cache queues, the pointers to form a linked-list queue. Methods are provided to maintain access records and calculate object keys.

Class *LogOutput*

- Logs statistic data to output files.