PROBABILISTIC METHODS FOR WEB CACHING AND PERFORMANCE PREDICTION OF IP NETWORKS AND WEB FARMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Konstantinos Psounis
December 2002

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Professor Balaji Prabhakar
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Professor Nicholas Bambos

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Professor Rajeev Motwani

Approved for the University Committee on Graduate Studies:

Preface

How to read this thesis

The thesis consists of two parts that are self contained. The introduction is kept brief to give a flavor of the work that follows.

The first part presents methods to improve the efficiency of the web and in particular of web caching. After reading Chapter 2 which gives an overview of web caching and the proposed methods, the reader can jump to any of the next three chapters to read more about each method.

The second part presents a novel technique to predict the performance of large-scale systems. After reading Chapters 6 and 7 that present background information and introduce the method, the reader may continue with Chapter 8 that applies the method to IP networks, or jump to Chapter 9 that applies the method to web farms.

Within each topic, an algorithm, method, or model is designed that is motivated by actual problems of today's Internet. Then, the performance of the design is analyzed using theoretical tools and simulations. Finally, comments on implementation issues and experimental results are provided.

Acknowledgments

I am grateful to my advisor Professor Balaji Prabhakar for his continued guidance and the numerous stimulating discussions. Balaji's belief in working on novel problems, and sharing new thoughts and ideas with colleagues shaped me as a researcher.

I would also like to thank my associate advisor Professor Nicholas Bambos for serving in my committees and for his support. Further, I would like to thank Professors Dawson Engler, Rajeev Motwani, and Peter Glynn for serving on my committees and for the fruitful research discussions we had at Stanford.

I would like to give thanks to Nat Kausik for giving me the opportunity to work on problems with direct application to the industry and to Jay Jawahar for the fruitful discussions we had at FineGround Networks.

I have had the privilege to collaborate, share office space, and discuss research topics with many talented people. I would like to thank Ayalvadi Ganesh, Paolo Giaccone, Pablo Molinero, Chandra Nair, Rong Pan, Devavrat Shah, Mayank Sharma, Elif Uysal, Damon Wichick, and An Zhu.

My life at Stanford would have been a lot less exciting without the Greek gang. For a long project like a Ph.D., this is essential. I had great fun with Athina, Christos, Dimitris, Nikos, Giannis, Kimonas, Savvas and Stavros. I also had great times with the newer generation, Katerina, Gianna, Giannis, Giorgos and Jpg.

These acknowledgments wouldn't be complete without mentioning the great pleasure that I have had wave-sailing California's coast.

Finally, I would like to express my deepest appreciation for my parents, my sister and Dora, for being a constant source of loving support and encouragement throughout the years. It is to them that I dedicate this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past ten years, the Internet has grown from a small-scale research network to an immense world-wide infrastructure. It is directly involved in hundreds of billions of dollars of economic activity annually, and each year its size more than doubles. But its great success is also the source of overwhelming problems; it is very hard to analyze, control, or even predict the performance of such a large-scale, heterogeneous network.

When trying to solve these problems, researchers and practitioners face the so-called "curse of dimensionality": Due to the very large number of users and the very high speeds of operation, there are too many things to do in too short a period of time. As a result, traditional methods often yield complex solutions that do not scale with size and speed. Probabilistic methods reduce the implementation complexity by requiring only a small random sample of the "state" and input. When this random sample has enough information for achieving good performance, the outcome is a large reduction in implementation complexity for a small sacrifice in performance.

We use this approach in two important areas of Internet research: web caching and performance prediction of networks and web farms.

We improve the efficiency of web caching by designing (i) a method that randomizes web cache replacement schemes, leading to reduced-complexity implementations at a small performance cost, (ii) a parsimonious model for web request sequences that provides guidelines for designing optimal replacement and prefetching algorithms, and (iii) a scalable scheme for exploiting the redundancy in dynamic web content.

Next, we reduce the complexity of measurement and analysis of large, complex systems, by introducing a scalable, widely applicable method for performance prediction for such systems. We apply this method to IP networks and web farms.

A brief summary of the concept and merits of probabilistic algorithms and methods, and an introduction to web caching and performance prediction are given in this chapter. More detailed background for the last two topics will be presented at the beginning of Parts I and II of this thesis.

1.1 Probabilistic Methods

A probabilistic algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random inputs and use these random inputs to influence its computation [65]. Berlekamp [9], Rabin [89], and Solovay and Strassen [101] gave early examples of concrete randomized algorithms in the seventies. Subsequently, a tremendous growth has taken place in this field. Randomized algorithms have evolved from being a tool in computational number theory to finding widespread application in areas like data structures [112], graph theory [12], pattern matching [57], and geometry [66]. [1]

There are two major advantages of probabilistic algorithms: They are simple to implement, and they terminate fast since they work with a sample of the "state" and input. Indeed, the space requirements and execution time of a randomized algorithm tend to be smaller than those of the best deterministic algorithm [65]. As a result of these benefits, randomized algorithms are becoming popular in the real world. For example, the success of Ethernet, the de facto standard for local area networks, is partially due to the probabilistic method that it uses to avoids collisions, which happen when multiple users want to send packets at the same time.

The price to pay for the simplicity and the fast execution is performance. Probabilistic algorithms are usually not optimal. However, a well-designed randomized algorithm can produce a solution that is "close" to the optimal solution with a high probability [65]. This is the case when the algorithm operates on an appropriate sample of the state and input that has enough information for achieving good performance.

Probabilistic methods are well-suited for problems where the optimal solutions are not implementable, as is often the case in networking problems. As a result, a number of randomized methods have been recently proposed to solve problems in various networking areas like bandwidth allocation [77, 83, 113], switching [43, 103], mobile computing [19, 22],

[1] For more information on randomized algorithms the interested reader is referred to [56, 65].

peer-to-peer systems [90], reliable multicasting [111], and denial of service attack [95]. This thesis applies this approach to web caching and to performance prediction for large systems.

1.2 Web Caching

The enormous popularity of the World Wide Web in recent years has caused a tremendous increase in network traffic due to HTTP [2] requests. HTTP traffic is the dominant traffic on the Internet and comprises more than 50% of the total traffic [40, 41].

Web traffic originates from clients that use their browsers to request web documents. These documents are hosted on web servers. Web caches often reside close to the clients and operate similarly to traditional CPU caches: they store frequently requested documents and respond to requests for these documents. Web caches may also reside next to servers, or in client's web browsers [59].

The web caching infrastructure provides a natural way to reduce network traffic, server load, and download latency. As a result, the performance of the Internet as a whole is significantly improved.

A critical component of a web cache is its replacement policy; that is, the policy it uses to evict older documents and replace them with newer documents that are more likely to be requested in the future. As a result, there has been a lot of research activity in designing novel document eviction schemes for web caches [1, 17, 55, 92, 106, 110]. The increasing speed of operation of the Internet, causes an increase in the load on web caches. Further, as the size of the Internet increases, there are more clients per cache and more distinct web documents. Thus, in order to maintain good performance, there is a need for larger web caches. (Cache sizes of many gigabytes are quite common nowadays [16].) This, in turn, poses implementation constraints on the eviction schemes, which motivates us to design a method that randomizes any deterministic replacement scheme, leading to reduced-complexity implementations at a small performance cost. The novelty of the method is a general procedure to improve the quality of a sample without increasing its size, that is applicable to a variety of iterative algorithms. In addition to being useful in practice, the method and its analysis are also of theoretical interest.

Another way to improve web performance is through traffic modeling. Traffic models are employed in two fundamental ways: either to drive discrete-event simulators, or as

[2]HyperText Transfer Protocol

part of an analytic model. Hence, they can be used to experiment with, and optimally design algorithms that exploit traffic correlations. Previous work on obtaining models of web traffic have observed that the distribution of the long term popularity of documents is Zipf-like [14, 45]. However, a Zipf-like popularity distribution alone doesn't suffice to capture temporal locality [55]; the request sequence needs to be correlated in time. To this end, we devise a simple, accurate, and analytically tractable Markovian model that simultaneously captures both temporal correlations and document popularity. The model is then used to generate synthetic traces, and to optimally design replacement algorithms.

Traditional web caching is applicable only to static documents, or to documents that change over large time-scales. However, a large proportion of web documents change frequently over time, which significantly reduces the gains from web caching [102, 109]. Even though these documents are not "cachable", the differences between consecutive versions of the same document are small. A natural way to exploit this observation is as follows: Both the clients and the server store the same version of a document, and upon a request for this document, the server sends to the clients the difference between the current version and the stored version. This technique is called "delta-encoding" [5, 50]. Despite its good performance, delta-encoding has not yet been deployed because it requires the server to store too many files: usually one file for every document for every client. Motivated by this deficiency, we introduce "class-based delta-encoding", a scheme that solves the storage problem by properly grouping documents into classes and storing one document per class.

1.3 Performance Prediction

Monitoring the performance of a large system like a popular web server farm or the Internet, and predicting its behavior under novel algorithms, protocols, architectures and load conditions are important research problems. Monitoring allows network and farm operators to ensure reliable operation and keep track of service level agreements, while performance prediction enables operators and researchers to do capacity planning and evaluate the effect of design changes on performance.

Despite their importance, there are currently no good solutions to these problems. This is not because researchers and industry have ignored them. On the contrary, there has been a lot of related research [6, 26, 30, 94] and industrial products [23, 81]. The problems of measurement and analysis of systems like the Internet are hard because of the systems' overwhelming size, complexity, heterogeneity, and speed of operation. For example, there are millions of packets per second traversing a link at the core of the Internet, where port speeds are as high as 10 gigabits per second. Further, there are millions of distinct flows in hourly traces, as recorded from Sprint's, MCI's and vBNS's links [31, 41]. Popular web farms suffer from similar issues: They receive millions of requests per second during peak times, and consist of hundreds of front-end web servers and hundreds of application servers at the back end. For example, America On Line (AOL) reported an average of 8 billion visits per day in the year 2000.

Traditional techniques for measurement and analysis, such as modeling or trace-driven simulation, are not appealing when applied to such systems. First, the heterogeneity and complexity of the Internet makes it very difficult and time consuming to devise realistic traffic models, and models for the network or a web farm. Second, it is very hard to take measurements because of the very high speeds of operation and the large amount of data. Even when such data are available, it is very expensive to run a realistic simulator due to the huge memory and CPU requirements. One way to reduce the sheer size of the traffic is sampling, but then it is hard to work backwards; that is, to infer the performance of the original system.

To sidestep these problems, we design a scalable, data-driven method for performance prediction of large, complex networks and web farms, which we call SHRiNK. [3] We first reduce the overall complexity of the system by using a sample of the actual traffic to drive a small-scale replica of the original system. Then, we observe and exploit a natural scaling law to extrapolate from the performance of the scaled system to that of the original. The challenge is to find suitable ways to sample the traffic and scale down the original system, so that performance extrapolation is possible.

Working with small-scale replicas is not a new idea in engineering: aeronautical and naval engineerers often study the behavior of aircraft and sailboat miniatures inside test-tubes, and extrapolate the results to the real scales. However, it is not a priori known whether such an approach would work in networking, neither is it clear what it means to scale down a network or a farm.

Our research agenda starts with answering whether such an approach has merit in networking: this is done by performing network simulations and web farm experiments. We find that the distribution of a number of performance metrics of the original system can

[3] SHRiNK: Small-scale Hi-fidelity Reproduction of Network Kinetics

be accurately predicted by the small-scale replica. Motivated by the positive results, we continue with devising theoretical arguments to understand why SHRiNK is working, and what is the range of applicability of the method. These arguments reveal that the method we suggest is widely applicable for a variety of performance measures, and for any network topology, transfer protocol, and queue management scheme.

Part I

Improving the Efficiency of Web Caching

Chapter 2

Caching on the Web

In its first decade of existence, the World Wide Web has altered the way information is created and distributed, and how business is conducted globally. Its enormous popularity has caused a tremendous increase in network traffic. Even after the emergent popularity of file sharing systems such as Napster in the last couple of years, the web is still the largest Internet application and web traffic comprises on average more than 50% of the total traffic [40, 41].

Web traffic originates from clients that use their browsers to request web documents, which are hosted on web servers. The web infrastructure also consists of intermediaries that reside between clients and servers. These intermediaries are called web proxies, and act as a server to clients and as a client to other proxies or origin servers [59]. The most commonly used proxies are web caches. Web caches usually reside close to clients and operate similarly to traditional CPU caches; they store frequently requested documents and respond to requests for these documents (see Figure 2.1). Web caches may also reside next to servers, or in client's web browsers [59].

The web caching infrastructure provides a natural way to reduce network traffic, server load, and download latency. As a result, the performance of the Internet as a whole is significantly improved.

The first part of this thesis investigates how to improve the performance and reduce the complexity of web caching. First, we introduce a method to randomize any web cache replacement policy, that reduces implementation complexity at a small performance cost. Second, we design a simple but accurate model for generating synthetic web traces, and use the model to design optimal and practical replacement policies. Third, we introduce

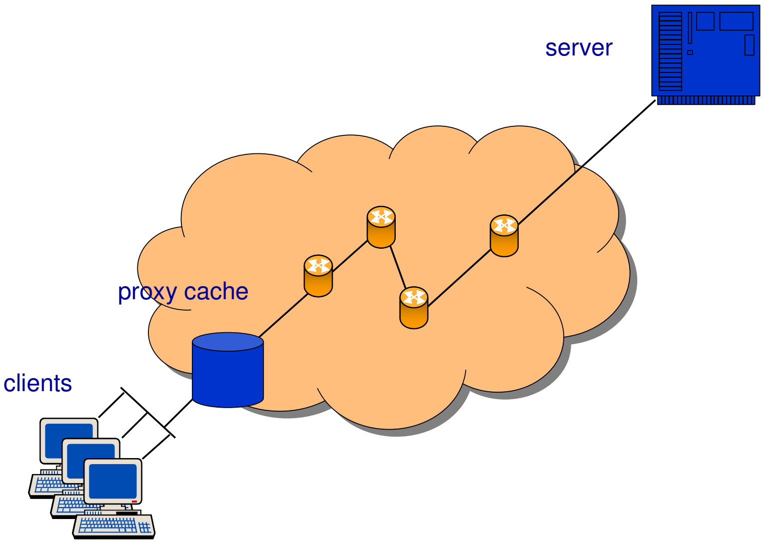

Figure 2.1: The web infrastructure.

a scalable scheme for exploiting the redundancy in web documents that change frequently over time. This scheme is based on the recently proposed idea of delta-encoding [5]. In the rest of this chapter we motivate our work on web caching, and give background information on each of the above three topics.

2.1 Randomized Replacement Policies

A critical component of a web cache is its replacement policy; that is, the policy that the cache uses to evict older documents and replace them with newer documents that are more likely to be requested in the future.

Traditionally, CPU caches employ the least recently used (LRU) algorithm, or a variant, as a replacement policy. By evicting the document that was requested further in the past, LRU exploits the observation that documents not recently requested have a low probability of being requested in the future, and hence they are good eviction candidates.

In web caches, there are more complicated criteria to decide the suitability of a document for eviction. In addition to recentness of use, frequency of use is also important. Further, document sizes are different, and the cost to fetch documents from different servers may vary significantly. Hence, more complicated policies that use all these criteria in their eviction policy, have better performance.

The problem with these policies is that their implementation can be complicated. In particular, they need priority queues to be deployed. Thus, the supporting computations become time consuming: For a cache of size K, [1] these policies require $O(\log K)$ operations per access time, while all this work is done to prepare for eviction times that can be a lot less frequent. This motivates us to seek for approximations to these high performance yet complicated schemes, that require no data structures for the eviction policy.

To this end, in Chapter 3 we introduce a method that randomizes any web cache replacement scheme, and leads to reduced-complexity implementations with a small sacrifice in performance. The novelty of the method is a general procedure to improve the quality of a sample without increasing its size. In particular, we exploit the iterative nature of the cache replacement problem, and use past eviction samples to improve current eviction decisions. Further, we study the tradeoff associated with this procedure using analysis and trace-driven simulations.

The main algorithmic contribution of this work is the demonstration that carrying information between iterations will greatly improve the performance of iterative randomized algorithms. While this is applied to web cache replacement policies, it is equally applicable to other iterative randomized algorithms of interest in networking (e.g. load balancing [97], switch scheduling [43, 96, 104]). Thus, the theoretical methods used may have wider applicability.

2.2 Modeling Web Request Sequences

A number of algorithms, such as replacement and prefetching policies, rely on correlations present in the sequence of requests for efficient performance. Further, the large size of real traces makes it very hard to evaluate the performance of new algorithms and architectures using these traces.

A simple but accurate web trace model can be used to synthetically generate realistic web traces, and hence alleviate the need to work with real traces. Further, such a model can provide design guidelines for replacement and other algorithms that exploit traffic correlations.

Previous work on obtaining web traffic models have observed that the distribution of the

[1] K can be many millions in large web caches as reported by product data sheets [16].

long term popularity of documents is Zipf-like (e.g. [14]). However, a Zipf-like popularity distribution is not enough to capture temporal locality [55], the request sequence needs to be correlated in time. The lack of a request sequence model that captures all the important correlations in web traces, motivates us to design such a model in Chapter 4. The proposed Markovian model is simple but accurate, analytically tractable and simultaneously captures temporal correlations and document popularity. The model is also extended to capture inter-document correlations.

A direct benefit of the model is the ability to generate synthetic traces with the desired statistical characteristics. As an important by-product, the model provides insights for designing efficient replacement algorithms. Using these insights, we obtain optimal algorithms and introduce a spectrum of practical, high-performance algorithms that adapt to the degree of temporal correlation present in the request sequence.

2.3 Exploiting Redundancy in Dynamic Traffic

Traditional web caching is applicable only to static documents, or to documents that change over large timescales. However, a large proportion of web documents, called dynamic documents, change frequently over time. This proportion is increasing day by day. As a result, the performance of current caching solutions cannot be significantly improved unless a mechanism is used to exploit the redundancy in dynamic traffic.

In particular, no matter what the replacement scheme, the cache size and the user population serviced by the cache, proxy cache hit rates are usually around 40% [109]. However, if proxy caches were equipped with mechanisms that exploit redundancy from static and dynamic documents, hit rates can increase up to 80% [102, 109].

Delta-encoding [5, 50] is a promising technique that exploits temporal correlation among different snapshots of a dynamic document as follows: Both the clients and the server store the same version of a document, called *base-file*, and upon a request for this document, the server sends to the clients the difference between the current version and the stored version. Then, the clients combine the base-file with the difference to generate the current snapshot of the document.

A major problem with delta-encoding is that it requires the server to store too many base-files: one file for every document, and for personalized web documents, one file for every document for every client. To solve this problem, in Chapter 5 we introduce *class-based delta-encoding*, a scalable scheme that groups documents into classes and stores one document per class. In essence, in addition to exploiting temporal correlation in a dynamically evolving document, we also exploit spatial correlation among different documents. The scheme also addresses the privacy concerns due to the maintenance of a single base-file per class, that is shared by many clients.

Chapter 3

Randomized Web Cache Replacement Schemes

A key component of a cache is its replacement policy, which is a decision rule for evicting a page currently in the cache to make room for a new page. A particularly popular rule for page replacement replaces the least recently used (LRU) page. This is due to a number of reasons: As an online algorithm it is known to have the best competitive ratio, [1] it only requires a linked list to be efficiently implemented as opposed to more complicated data structures required for other schemes, and takes advantage of temporal correlations in the request sequence.

Suppose that we associate with any replacement scheme a *utility function*, which sorts pages according to their suitability for eviction. For example, the utility function for LRU assigns to each page a value which is the time since the page's last use. The replacement scheme would then replace that page which is most suitable for eviction.

Whereas for processor caches LRU and its variants have worked very well [99], it has recently been found that LRU is not suitable for web caches [17]. This is because some important differences distinguish a web cache from a processor cache: (i) The size of web documents are not the same, (ii) the cost of fetching different documents varies significantly, and (iii) short-term temporal correlations in web request sequences are not as strong. Thus, a utility function that takes into account not only the recency of use of a web document, but also its size, cost of fetching, and frequency of use can be expected to perform significantly

[1]That is, there is no other deterministic online algorithm with a smaller competitive ratio [65].

better. Recent work proposes many new cache replacement schemes that exploit this point [1, 17, 55, 92, 106, 110].

However, the data structures that are needed for implementing these new utility functions turn out to be complicated. Most of them require a priority queue in order to reduce the time to find a replacement from $O(K)$ to $O(\log K)$, where K is the number of documents in the cache. Further, these data structures need to be *constantly updated* (i.e., even when there is no eviction), although they are solely used for eviction.

This prompts us to consider randomized algorithms which do not need any data structures to support the eviction decisions. For example, the particularly simple Random Replacement (RR) algorithm evicts a document drawn at random from the cache [65]. However, as might be expected, the RR algorithm does not perform very well.

We propose to combine the benefits of both the utility function based schemes and the RR scheme. Thus, consider a scheme which draws N documents from the cache and evicts the least useful document in the sample, where the "usefulness" of a document is determined by the utility function. Although this basic scheme performs better than RR for small values of N, we find a big improvement in performance by refining it as follows: After replacing the least useful of N samples, the identity of the next $M < N$ least useful documents is retained in memory. At the next eviction time, $N-M$ new samples are drawn from the cache and the least useful of these $N-M$ and M previously retained is evicted, the identity of the M least useful of the remaining is stored in memory, and so on.

Intuitively, the performance of an algorithm that works on a few randomly chosen samples depends on their quality. Therefore, by deliberately tilting the distribution of the samples towards the good side, which is precisely what the refinement achieves, one expects an improvement in performance. Surprisingly, we find that the performance improvement is significant for small values of M (e.g. 1, 2 or 3). As the value of M increases one expects a degradation in performance because bad samples are being retained and not enough new samples are being chosen. This suggests there is an optimal value of M. We analytically demonstrate the above observations and obtain an approximate formula for the optimal value of M as a function of N.

The rest of the chapter is organized to reflect the three main parts of this work: (i) a description of the proposed randomized algorithm for document replacement (Section 3.1), (ii) an analysis of its general features (Sections 3.2 and 3.3), and (iii) a simulation comparison, using web traces, of its performance relative to the deterministic algorithms it approximates (Section 3.4). Finally, Section 3.5 discusses implementation issues and further motivates the use of the randomized algorithm in practice.

3.1 The Algorithm

The first time a document is to be evicted, N samples are drawn at random from the cache and the least useful of these is evicted. Then, the next $M < N$ least useful documents are retained for the next iteration. And when the next replacement is to be performed, the algorithm obtains $N - M$ new samples from the cache, and replaces the least useful document from the $N - M$ new samples and the M previously retained. This procedure is repeated whenever a document needs to be evicted. In pseudo-code the algorithm is shown in Figure 3.1.

```
if (eviction) {
   if (first_iteration) {
      sample(N);
      evict_least_useful;
      keep_least_useful(M);
   } else {
      sample(N-M);
      evict_least_useful;
      keep_least_useful(M);
   }
}
```

Figure 3.1: The randomized algorithm.

An error is said to have occurred if the evicted document *does not* belong to the least useful n^{th} percentile of all the documents in the cache, for some desirable values of n. The goal of the algorithm we consider is to minimize the probability of error. We shall say that a document is *useless* if it belongs to the least useful n^{th} percentile. [2]

It is interesting to conduct a quick analysis of the algorithm described above in the case where $M = 0$ so as to have a benchmark for comparison. Accordingly, suppose that all the documents are divided into $100/n$ bins according to usefulness and N documents are

[2]Note that samples that are good eviction candidates will be called "useless" samples since they are useless for the cache.

sampled uniformly and independently from the cache. Then the probability of error equals $(1 - n/100)^N$, [3] which approximately equals $e^{-nN/100}$. By increasing N this probability can be made to decrease exponentially fast. (For example, when $n = 8$ and $N = 30$, the probability of error is approximately 0.08. By increasing N to 60, the probability of error can be made as low as 0.0067.)

But it is possible to do much better without doubling N! That is, even with $N = 30$, by choosing $M = 9$, the probability of error can be brought down to 2.4×10^{-6}. In the next few sections we obtain models to further understand the effect of M on performance. Rather surprisingly, we find that the reduction on the probability of error can be *exponential* for small values of M, leading to a significant improvement on hit rate.

We end this section with the following remark. Whereas it is possible for a document whose id is retained in memory to be accessed between iterations, making it a "recently used document", we find that in practice the odds of this happening are negligibly small. [4] Hence, in all our analysis, we shall assume that documents which are retained in memory are not accessed between iterations.

3.2 The Model and Preliminary Analysis

In this section we derive and solve a model that describes the behavior of the algorithm precisely. We are interested in computing the probability of error, which is the probability that none of the N documents in the sample is useless for the cache, for any given N and n and for all M $(0 \leq M < N)$.

We proceed by introducing some helpful notation. Of the M samples retained at the end of the $(m-1)^{th}$ iteration, let Y_{m-1} $(0 \leq Y_{m-1} \leq M)$ be the number of useless documents. At the beginning of the m^{th} iteration, the algorithm chooses $N - M$ fresh samples. Let A_m, $0 \leq A_m \leq N - M$ be the number of useless documents coming from the $N - M$ fresh samples. In the m^{th} iteration, the algorithm replaces one document out of the total $Y_{m-1} + A_m$ available (so long as $Y_{m-1} + A_m > 0$) and retains M documents for the next iteration. Note that it is possible for the algorithm to discard some useless documents because of the memory limit of M that we have imposed.

Define $X_m = \min(M + 1, Y_{m-1} + A_m)$ to be precisely the number of useless documents

[3] Although the algorithm samples without replacement, the values of N are so small compared to the overall size of the cache that $(1 - n/100)^N$ almost exactly equals the probability of error.

[4] Trace driven simulations in Section 3.4 support our observation.

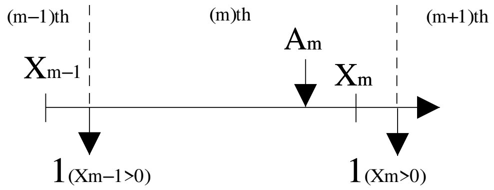

Figure 3.2: Sequence of events per iteration. Note that eviction takes place prior to resampling.

in the sample just prior to the m^{th} document replacement, that the algorithm would ever replace at eviction times. If $X_m = 0$, then the algorithm commits an error at the m^{th} eviction. It is easy to see that X_m is a Markov chain and satisfies the recursion

$$X_m = \min(M+1, X_{m-1} - 1_{(X_{m-1}>0)} + A_m),$$

and that A_m is binomially distributed with parameters $N-M$ and $n/100$. For a fixed N and n, let $p_k(M) = P(A_m = k)$, $k = 0, \ldots, N-M$, denote the probability that k useless documents for the cache, and thus good eviction candidates, are acquired during a sampling. When it is clear from the context we will abbreviate $p_k(M)$ to p_k. Figure 3.2 is a schematic of the above Markov chain.

Let T_M denote the transition matrix of the chain X_m for a given value of M. The form of the matrix depends on whether M is smaller or larger than $N/2$. Since we are interested in small values of M, we shall suppose that $M \leq N/2$. [5] It is immediate that T_M is irreducible and has the general form

$$T_M = \begin{pmatrix} p_0 & p_1 & p_2 & \ldots & p_M & 1-\sum_{i=0}^{M} p_i \\ p_0 & p_1 & p_2 & \ldots & p_M & 1-\sum_{i=0}^{M} p_i \\ 0 & p_0 & p_1 & \ldots & p_{M-1} & 1-\sum_{i=0}^{M-1} p_i \\ 0 & 0 & p_0 & \ldots & p_{M-2} & 1-\sum_{i=0}^{M-2} p_i \\ \vdots & & & \ddots & & \vdots \\ 0 & 0 & 0 & \ldots & p_0 & 1-p_0 \end{pmatrix}$$

As may be inferred from the transition matrix, the Markov chain models a system with

[5] Figure 3.3 suggests that the M at which the probability of error is minimized is less than $N/2$.

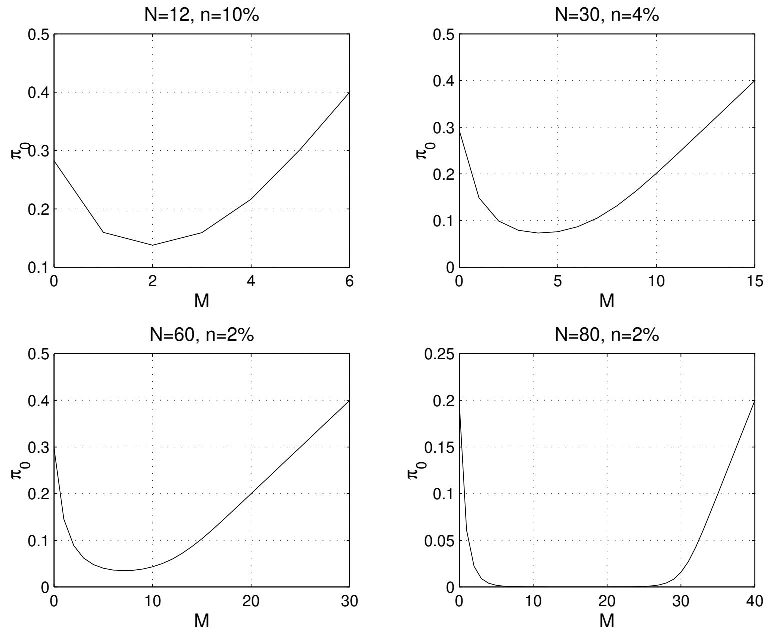

Figure 3.3: Probability of error (π_0=probability not a useless document for the cache is replaced) versus number of documents retained (M).

one deterministic server, binomial arrivals, and a finite queue of size M (the system's overall size is $M+1$). An interesting feature of the system is that as M increases, the average arrival rate, $E(A_m) = (N-M)n/100$, decreases linearly and the maximum queue size increases linearly.

Let $\pi = (\pi_0, \ldots, \pi_{M+1})$ denote the stationary distribution of the chain X_m. Clearly π_0 is the probability of error as defined above. Let $A = (a_{ij})$ be an $(M+2)\times(M+2)$ matrix, with $a_{ij} = 1$ for all i, j. Let $a = (a_i)$ be a $1\times(M+2)$ matrix with $a_i = 1$ for all i. Since T_M is irreducible, $I - T_M + A$ is invertible [71] and [6]

$$\pi = a \cdot (I - T_M + A)^{-1}. \tag{3.1}$$

Figure 3.3 shows a collection of plots of π_0 versus M for different values of N and n. We note that given N and n there are values of $M > 0$ for which the error probability is very small compared to its value at $M = 0$. We also observe that there is no need for N to

[6] Equation 3.1 has the same solution with the system of equations $\{\pi = \pi T_M, \sum_i \pi_i = 1\}$, but it is more convenient to work with.

be a lot bigger than the number of bins $100/n$ for the probability of error to be as close to zero as desired, since even for $N = 2 \cdot 100/n$ the minimum probability of error is extremely small. Finally, we notice that for small values of M there is a huge reduction in the error probability and that the minimum is achieved for a small M. As M increases further the performance deteriorates linearly.

The exponential improvement for small M can be intuitively explained as follows. For concreteness, suppose that $M = 1$ and that the Markov chain X_m has been running from time $-\infty$ onwards (hence it is in equilibrium at any time $m \geq 0$). The relationship $\{X_m = 0\} \subset \{A_m = 0; A_{m-1} \leq 1\}$ immediately gives that $P(X_m = 0) \leq P(A_m = 0)[P(A_{m-1} = 0) + P(A_{m-1} = 1)]$. Supposing that $N \geq 2 \cdot 100/n$, $P(A_m = 0) \approx e^{-3}$ and $P(A_m = 1) \approx 3e^{-3}$. Therefore $P(X_m = 0) \leq 4e^{-6}$. Compare this number with the case $M = 0$, where $P(X_m = 0) = P(A_m = 0) \approx e^{-3}$, and the claimed exponential improvement is apparent.

The linear increase of π_0 for large M, evident from Figure 3.3, can be seen from the following argument. As M increases, the average arrival rate decreases and the queue size increases. Thus, for large M, the queue virtually never overflows and good eviction candidates are not discarded due to a lack of storage. The problem is that the smaller arrival rate brings fewer good eviction candidates, making the linear decrease of the arrival rate the dominant phenomenon in performance degradation. Recall that for a queue with arrival rate λ and service rate μ when the overflow probability is negligible, the probability that it is empty, π_0, equals $1 - \lambda/\mu$. In our case $\pi_0 = 1 - (N - M)\frac{n}{100}$, showing the linear increase in π_0 with M.

3.2.1 A Closer Look at the Performance Curves

From the above discussion and Figure 3.3 we may deduce that the error probability as a function of M and for given N and n has the following features: As M increases from 0, the error probability decreases exponentially, flattens out, and then increases linearly. In particular, it appears that the error probability is a convex function of M. The rest of this section shows that this convexity is a general feature of the algorithm and holds for arbitrary values of N and n.

To establish convexity directly it would have helped greatly if $\pi_0(M)$ could be expressed as a function of the elements of T_M in closed form. Unfortunately, this is not the case and we must use an indirect method, which seems interesting in its own right. Our method consists of relating $\pi_0(M)$ to the quantity $D(t, M, \lambda(M))$, which is the number of overflows in the time interval $[0, t]$ from a buffer of size M with average arrival rate $\lambda(M)$.

Let $\overline{D}(M) \stackrel{\triangle}{=} \lim_{t\to\infty} D(t, M, \lambda(M))/t$.

Theorem 3.1 *The probability of error is convex in M.*

Proof: Let $A^M[0, t]$ be the number of arrivals in $[0, t]$. Then the probability the system is full as observed by arrivals, or equivalent the probability of drops, equals

$$P_{drop} = \lim_{t\to\infty} \frac{D(t, M, \lambda(M))}{A^M[0, t]} = \lim_{t\to\infty} \frac{D(t, M, \lambda(M))}{t} \frac{t}{A^M[0, t]} = \overline{D}(M)\lambda(M).$$

Proceeding, equating effective arrival and departure rates we obtain

$$\begin{aligned} \lambda(M) \cdot (1 - P_{drop}(M)) &= (1 - P_{empty}(M)), \\ \text{or } P_{empty}(M) &= 1 - \lambda(M) \cdot (1 - P_{drop}(M)), \\ \text{or } P_{empty}(M) &= 1 - \lambda(M) + \overline{D}(M). \end{aligned} \tag{3.2}$$

Since $\lambda(M) = (N - M)\frac{n}{100}$ is linear in M, and Lemma 3.3 below implies that $\overline{D}(M)$ is convex in M, Equation (3.2) implies $P_{error}(M) = P_{empty}(M)$ is convex in M. ∎

To complete the proof it remains to show that $\overline{D}(M)$ is convex in M. The proof of the convexity of $\overline{D}(M)$ is carried out in Lemmas 3.1, 3.2, and 3.3 below. In the following we abbreviate $D(t, M, \lambda(M))$ to $D(t, M)$ when the arrival process does not depend on M, and to $D(t, \lambda(M))$ when the buffer size is constant, regardless of the value of M. Lemma 3.1 shows that $D(t, M)$ is convex in M for all $t > 0$. Lemma 3.2 shows the convexity of $\lim_{t\to\infty} D(t, \lambda(M))/t$. Finally, Lemma 3.3 shows the convexity of $\overline{D}(M)$.

For now, we only give a sketch of the somewhat combinatorially involved proofs of these results. The full proofs can be found in the appendix at the end of the chapter.

Lemma 3.1 *$D(t, M)$ is a convex function of M, for each $t > 0$.*

Sketch of proof: To prove convexity it suffices to show that the second order derivative of the number of drops is non-negative; i.e., that $(D(t, M-1) - D(t, M)) - (D(t, M) - D(t, M+1)) \geq 0$. This can be done by comparing the number of drops $D(t, M-1)$, $D(t, M)$, and $D(t, M+1)$ from systems with buffer sizes $M-1$, M, and $M+1$ respectively, under identical arrival processes.

Essentially, the comparison entails considering four situations for buffer occupancies in the three systems, as illustrated in Figure 3.18 in the appendix. ■

Let $D(\lambda(M)) \triangleq \lim_{t\to\infty} D(t, \lambda(M))/t$.

Lemma 3.2 *$D(\lambda(M))$ is a convex function of M.*

Sketch of proof: We need to show $D(\lambda(M-1)) - 2D(\lambda(M)) + D(\lambda(M+1)) \geq 0$ by considering three systems with same buffer sizes and binomially distributed arrival processes with average rates $\lambda(M-1)$, $\lambda(M)$ and $\lambda(M+1)$. Thus, there will be common arrivals and exclusive arrivals as categorized below:

(a) An arrival occurs at all three systems.

(b) An arrival occurs only at the system with buffer size $M-1$.

(c) An arrival occurs at the two systems with buffer sizes $M-1$ and M and there is no arrival at the system with buffer size $M+1$.

Due to the arrival rates being as in the hypothesis of the lemma, category (b) and (c) arrivals are identically distributed. Using this and combinatorial arguments one can then show that $D(\lambda(M))$ is convex. ■

Lemma 3.3 *$\overline{D}(M)$ is a convex function of M.*

Sketch of proof: We consider three systems of buffer sizes $M-1$, M and $M+1$, whose arrival processes are Binomially distributed with rates $\lambda(M-1)$, $\lambda(M)$ and $\lambda(M+1)$. This is a combination of Lemma 3.1 and 3.2. ■

3.3 On the Optimal Value of M

The objective of this section is to derive an approximate closed form expression for the optimal value of M for a given N and n. This expression (Equation (3.6)) is simple and, as Table 3.2 shows, it is quite accurate over a large range of values of N and n.

Let $M^* = \arg\min\{\pi_0(M)\}$ be the optimal value of M. As remarked earlier, even though the form of the transition matrix, T_M, allows one to write down an expression for $\pi_0(M)$, this expression doesn't allow one to calculate M^*. Thus, we have to numerically solve

N	$min(\pi_0)$			M^*		
	n=10	n=20		n=10	n=20	
8	0.3643	0.0593		1	2	
10	0.2450	0.0110		1	3	
12	0.1378	0.0011		2	4	
	n=5	n=10		n=5	n=10	
20	0.1946	0.0013		2	5	
	n=4	n=8		n=4	n=8	
30	0.0732	2.4454^{-6}		4	9	
	n=3	n=6	n=9	n=3	n=6	n=9
40	0.0558	8.0595^{-8}	4.6629^{-15}	5	12	16
	n=2	n=4	n=6	n=2	n=4	n=6
50	0.1354	1.8678^{-6}	9.5368^{-14}	4	13	18
60	0.0350	8.3933^{-11}	-	7	19	-
70	0.0025	-	-	11	-	-
80	3.1553^{-5}	-	-	16	-	-

Table 3.1: Optimum values of π_0 and M for various N and n

Equation (3.1), compute $\pi_0(M)$ for all $M \leq N/2$, and read off M^* for various values of N and n, as in Table 3.1. This table is to be read as follows. Suppose N=30 and n = 4%. Then minimum value of π_0 is 0.0732 and it is achieved at $M^* = 4$.

Even though there is no exact closed form solution from which one might calculate M^*, we can derive an approximate close form solution using elementary martingale theory. Recall that X_m is the number of useless documents in the sample and that $X_{m+1} = ((X_m - 1 + A_m) \wedge (M+1)) \vee 0$. [7] The boundaries at 0 and $M+1$ complicate the analysis of this Markov Chain. The idea is to work with a Markov chain that has no boundaries and then use the *Optional Stopping Time Theorem* [29] to take into account the boundaries at 0 and $(M+1)$.

Since we want to operate away from the event $\{X_m = 0\}$, we assume that the Markov chain will have a positive drift towards the boundary at $(M+1)$. Let $Z'_m = (M+1) - X_m$. Then Z'_m will have a negative drift towards the boundary at 0.

[7] The symbols $\wedge$ and $\vee$ denote the minimum and maximum operations, respectively.

Let Z_m be the corresponding unreflected Markov chain which follows the equation $Z_{m+1} = Z_m + 1 - A_m$. Consider the exponential martingale

$$W_m = \exp(\theta Z_m),$$

where θ will be chosen so that $E(W_{m+1}|W_m) = W_m$. Since

$$\begin{aligned}
E(W_{m+1}|W_m) &= E(\exp(\theta Z_{m+1})|Z_m) \\
&= E(\exp(\theta(Z_m + 1 - A_m))|Z_m) \\
&= W_m \cdot \exp(\theta) \cdot E(\exp(-\theta A_m)),
\end{aligned}$$

we obtain that θ must satisfy the equation

$$E(\exp(-\theta A_m)) = \exp(-\theta).$$

Since A_m is binomially distributed with parameters $N-M$ and $p = n/100$, $E(\exp(-\theta A_m)) = (p \cdot \exp(-\theta) + 1 - p)^{N-M}$. Therefore, we require that θ solve the equation

$$(N - M) \cdot \ln(1 - p(1 - \exp(-\theta))) = -\theta. \tag{3.3}$$

Since there is no general closed form solution to the above equation, using a Taylor's approximation for $\ln(1 - x)$ and $\exp(x)$ and keeping terms up to θ^2 we get:

$$\theta = \frac{2}{p(1-p)} \left(p - \frac{1}{(N-M)} \right). \tag{3.4}$$

Let $T = \min\{m > 0 : Z_m \leq 0 \text{ or } Z_m = M + 1\}$. Then T is a bounded stopping time and we can use the Optional Stopping Time Theorem to get

$$E(W_0) = E(W_T).$$

Let $Z_0 = 0$. Then $W_0 = 1$ and thus

$$\begin{aligned}
& E(\exp(\theta Z_T)) = 1 \\
\Rightarrow\ & E(\exp(\theta Z_T) 1_{(Z_T = M+1)}) \leq 1 \\
\Rightarrow\ & \exp(\theta(M+1)) \cdot P(Z_T = M+1) \leq 1
\end{aligned}$$

N	M^o, M^+, M^*								
	n=10			n=20					
8	0	1	1	1.3	1	2			
10	0	0	1	2.6	2	3			
12	0.5	1	2	3.9	3	4			
	n=5			n=10					
20	0	0	2	5.5	5	5			
	n=4			n=8					
30	2.2	2	4	10.3	9	9			
	n=3			n=6			n=9		
40	3.0	3	5	13.8	12	12	18.7	15	16
	n=2			n=4			n=6		
50	0	0	4	14.3	13	13	20.8	18	18
60	4.8	5	7	20.9	18	19	-	-	-
70	10.4	10	11	-	-	-	-	-	-
80	16.4	15	16	-	-	-	-	-	-

Table 3.2: Comparison of optimum values of M for various N and n, calculated from the martingale approximation (M^o, M^+) and the Markov chain (M^*).

$$\Rightarrow \quad P(X_T = 0) \leq \exp(-\theta(M+1)). \tag{3.5}$$

From (3.5), to minimize the probability of error it suffices to maximize $\theta(M) \cdot (M+1)$ over M. Using Equation (3.4) and elementary algebra we conclude that the optimal value is given by

$$M^o = N - \sqrt{(N+1)100/n}. \tag{3.6}$$

Some comments about the approximations we have made in the above derivation are in order. We have dropped the term $E(\exp(\theta Z_T)1_{(Z_T \leq 0)})$ to obtain the bound in inequality (3.5). This term is not negligible, in general. However, it is negligible for positive θ, and Z_T sufficiently negative, [8] which holds for reasonably large values of $N \times n$. When the term is not negligible, the approximation for M^o in (3.6) can differ from the true optimum by 1 or 2.

[8] Note that θ is positive when the number of samples $N - M$ are more than the number of bins $100/n$, and Z_T becomes more negative the more and better samples we have.

In Table 3.2 we compare the results for the optimal M obtained by: (i) Equation (3.6), denoted by M^o, (ii) numerically solving Equation (3.3) for θ and then using the bound in (3.5), denoted by M^+, and (iii) the Markov chain model, denoted by M^*. This table is to be read as follows. For example, suppose $N = 40$ and $n = 6$, the optimal M equals (i) $M^o = 13.8$, (ii) $M^+ = 12$, and (iii) $M^* = 12$. Note that M^o and M^+ are very close to M^* unless the number of samples is very close to the number of bins.

So far we derived an approximate closed form expression for M^*. Here we comment on the performance improvement when a memory of M^* is used. We have mentioned that introducing memory leads to an exponential improvement in performance. A direct way of verifying this is to compare the number of samples needed to achieve a certain performance with and without memory. In particular, fix an n and for each $N > 0$, compute the minimum probability of error obtainable using the optimal value of memory. Then, compute the number of samples needed to achieve the same probability of error without memory. Denote by N^* and N^0 the number of samples needed to obtain the same error probability with and without memory, respectively. Figure 3.4 plots N^* versus N^0 for four different values of n as the error probability varies.

There is no closed form expression for the curves in Figure 3.4. However, one can obtain an upper bound on N^0 as follows. One of the events that will lead to an error is when all the $N - M$ fresh samples of all the consecutive $M + 1$ iterations will be bad samples. Thus $P_{error} > (1 - n/100)^{(N-M)(M+1)}$ or $N^0 < (N - M) \cdot (M + 1)$. In a different context, the authors in [97] proved that $N^0 > (N - M - \frac{M}{2}) \cdot (M + 1)$.

3.4 Trace Driven Simulations

We present simulations using web traces to study the performance of our algorithm under real traffic. In particular, we approximate deterministic cache replacement schemes using our randomized algorithm, and compare their performances. Recall that any cache replacement algorithm is characterized by a utility function that determines the suitability of a page for eviction. The main issues we wish to understand are:

- How good is the performance of the randomized algorithm according to realistic metrics like hit rate and latency? It is important to understand this because we have analyzed performance using the frequency of eviction from designated percentile bins

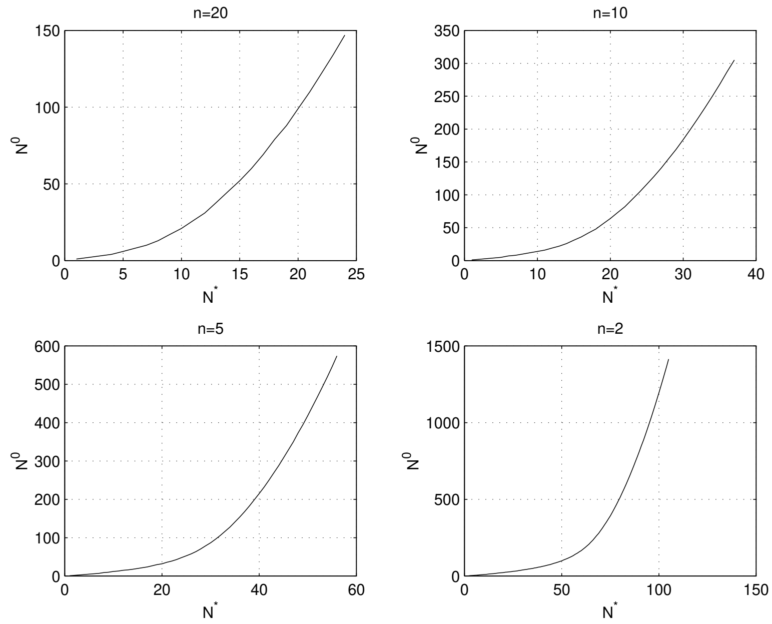

Figure 3.4: Comparison of number of samples required to achieve same probability of error with (N^*) and without (N^0)memory.

as a metric. This metric has a strong positive correlation with realistic metrics but doesn't directly determine them.

- Our analysis in the previous sections assumes that documents retained in memory are not accessed between iterations. Clearly, in practice, this assumption can only hold with a high probability at best. We show that this is indeed the case and determine the probability that a sample retained in memory is accessed between iterations.

- How long do the best eviction candidates stay in the cache? If this time is very long (on average), then the randomized scheme would waste space on "dead" items that can only be removed by a cache flush.

Of the three items listed above, the first is clearly the most important and the other two are of lesser interest. Accordingly, the bulk of the section is devoted to the first question and the other two are addressed towards the end.

3.4.1 Deterministic Replacement Algorithms

We shall approximate the following two deterministic algorithms: (i) LRU, and (ii) GD-Hyb, which is a combination of the GD-Size [17] and the Hybrid [110] algorithms. LRU is chosen because it is the standard cache replacement algorithm. GD-Hyb represents the class of new algorithms which base their document replacement policy on multiple criteria (recentness of use, size of document, cost of fetching documents, and frequency of use). We briefly describe the details of the deterministic algorithms mentioned above.

1. *LRU.* The utility function assigns to each document the most recent time that the document was accessed.

2. *Hybrid* [110]. The utility function f assigns eviction values to documents according to the formula
$$f = (L + \frac{W_1}{B})F^{W_2}/S,$$
where L is an estimate of the latency for connecting with the corresponding server, B is an estimate of the bandwidth between the proxy cache and the corresponding server, F is the number of times the document has been requested since it entered the cache (frequency of use), S is the size of the document, and W_1, W_2 are weights. [9] Hybrid evicts the document with the smallest value of f.

3. *GD-Size* [17]. Whenever there is a request for a document, the utility function f adds the reciprocal of the document's size to the currently minimum eviction value among all the documents in the cache, and assigns the result to the document. Thus, the eviction value for document i is given by
$$f_i = \min(f_j : j \text{ in cache}) + \frac{1}{S_i},$$
Note that the quantity $\min(f_j : j \text{ in cache})$ is increasing in time and it is used to take into account the recentness of a document. Indeed, since whenever a document is accessed its eviction value is increased by the currently minimum eviction value, the most recently used documents tend to have larger eviction values. GD-Size evicts the document with the smallest value of f.

[9] In the simulations we use the same weights as in [110].

4. *GD-Hyb* uses the utility function of Hybrid in place of the quantity $1/S$ in the utility function of GD-Size. Thus, its utility function is as follows:

$$\begin{aligned} f &= \min(f) + f', \quad \text{where} \\ f' &= (L + \frac{W_1}{B}) F^{W_2} / S. \end{aligned}$$

We shall refer to the randomized versions of LRU and GD-Hyb as RLRU and RGD-Hyb respectively. Note that the RGD-Hyb algorithm uses the $\min(f)$ among the samples, and not the global $\min(f)$ among all documents in the cache.

So far we have described the utility functions of some deterministic replacement algorithms. Next, we comment on the implementation requirements of those schemes.

LRU can be implemented with a linked list that maintains the order in which the cached documents were accessed so far. This is due to the "monotonicity" property of its utility function; whenever a document is accessed, it is the most recently used. Thus, it should be inserted at the bottom of the list and the least recently used document always resides at the top of the list. However, most algorithms, including those which have the best performance, lack the monotonicity property and they need to search all documents to find which one to evict. To reduce the computational overhead, they must use a priority queue to drop the search cost to $O(logK)$, where K is the number of documents in the cache. In particular, Hybrid, GD-Size, and GD-Hyb must use a priority queue.

The authors in [92] propose an algorithm called LRV (Lowest Relative Value). This algorithm uses a utility function that is based on statistical parameters collected by the server. By separating the cached documents into different queues according to the number of times they are accessed, or their relative size, and by taking into account within a queue only time locality, the algorithm maintains the monotonicity property of LRU *within* a queue. LRV evicts the best among the documents residing at the head of these queues. Thus, the scheme can be implemented with a constant number of linked lists, and finds an eviction candidate in constant time. However, its performance is inferior to algorithms like GD-Size [17]. Also, the cost of maintaining all these linked lists is still high.

The best cache replacement algorithm is in essence the one with the best utility function. In this work we don't seek the best utility function. Instead, we propose a low cost, high performance, robust algorithm that treats all the different utility functions in a unified way.

Traces	total requests	unique requests	one-timers	θ
Virginia	54434	27395	20603	0.59
Boston	104496	34178	22157	0.69
NLANR daily	295295	167862	122966	0.49
NLANR weekly	2876105	1012831	658319	0.72

Table 3.3: Trace workload characteristics.

3.4.2 Web Traces

The traces we use are taken from Virginia University, Boston University, and National Laboratory for Applied Network Research (NLANR). In particular:

- The *Virginia* [106] trace consists of every URL request appearing on the Computer Science Department backbone of Virginia University, with a client inside the department naming any server in the world. The trace was recorded for a 37 day period in September and October 1995. There are no latency data on that trace thus it can not be used to evaluate RGD-Hyb.

- The *Boston* [27] trace consists of all HTTP requests originating from 32 workstations in the Computer Science Department of Boston University. It was collected in February 1995 and contains latency data.

- The *NLANR* [70] traces consist of two sets, one daily trace, and a weekly trace. The daily trace was recorded the 23rd of September 2000, while the weekly trace was collected from the 22nd to the 28th of September 2000. [10] Both of them contain latency data.

We only simulate requests with a known reply size. Table 3.3 presents the workload characteristics of the traces, namely the total number of requests, the number of unique requests, the number of one-timers, and a popularity-parameter, θ, resulting from fitting the popularity distribution of the documents of each trace to a Zipf-like distribution. [11]

3.4.3 Results

We use three performance criteria:

[10] NLANR traces consist of daily and weekly traces of many sites; the one we used is the PA site.

[11] Let p_i be the probability of requesting the i^{th} most popular document of a trace. In practice, $\{p_i\}$ is known to be Zipf-like [14]; i.e., $p_i \propto 1/i^\theta$, where θ is the popularity-parameter of the trace.

Traces	HR	BHR	LR
Virginia	42.40%	32.57%	-
Boston	54.38%	36.05%	33.72%
NLANR daily	24.74%	16.00%	13.91%
NLANR weekly	31.13%	17.50%	23.28%

Table 3.4: Absolute values of HR, BHR, and LR for the four traces, for an infinite size cache.

(i) the hit rate (HR), which is the fraction of client-requested URLs returned by the proxy cache,

(ii) the byte hit rate (BHR), which is the fraction of client requested bytes returned by the proxy cache, and

(iii) the latency reduction (LR), which is the reduction of the waiting time of the user from the time the request is made till the time the document is fetched to the terminal (download latency), over the sum of all download latencies.

For each trace, HR, BHR, and LR are calculated for a cache of infinite size, that is a cache large enough to avoid any evictions. Then, they are calculated for a cache of size $0.5\%, 5\%, 10\%$, and 20% of the minimum size required to avoid any evictions. This size is 500MB, 900MB, 2GB, and 20GB, for Virginia, Boston, daily NLANR, and weekly NLANR trace respectively. Table 3.4 shows the absolute values of HR, BHR, and LR for the four traces, for an infinite size cache. Notice that the HR of the NLNAR traces is lower than the HR of the other traces, since NLANR caches are second-level caches. [12]

All the traces give similar results. Below, we show the performance of RLRU under the Virginia trace, the performance of RGD-Hyb under the Boston trace, and the performance of both under the longer and newer NLANR traces.

Figure 3.5 presents the ratio of the HR of LRU, RLRU, and RR over the HR achieved by an infinite cache, using the Virginia trace. As expected, the more the samples the better RLRU approximates LRU. Note that $N = 8$ and $M = 2$ are enough to make RLRU perform almost as good as LRU, and even $N = 3$, $M = 1$ give good results. Figure 3.6 presents the ratio of the BHR of LRU, RLRU, and RR, over the BHR achieved by an infinite cache. Note again that $N = 8$ and $M = 2$ are enough to make RLRU perform almost as good as LRU.

[12] First-level caches exploit most of the temporal locality in web-request sequences.

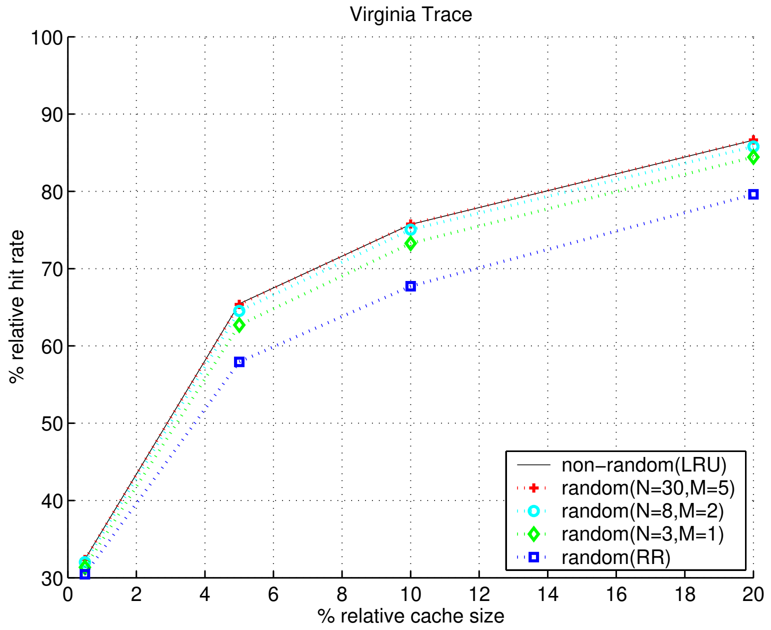

Figure 3.5: Hit rate comparison between LRU and RLRU.

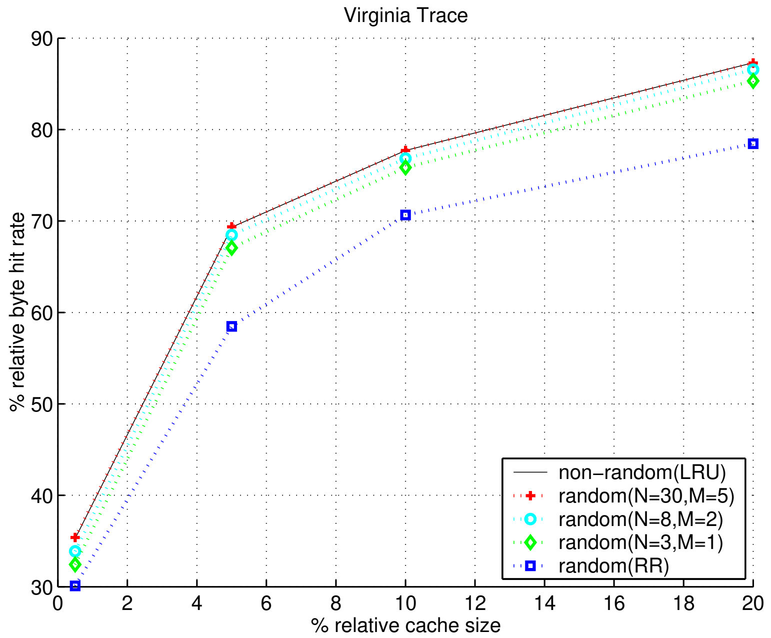

Figure 3.6: Byte Hit rate comparison between LRU and RLRU.

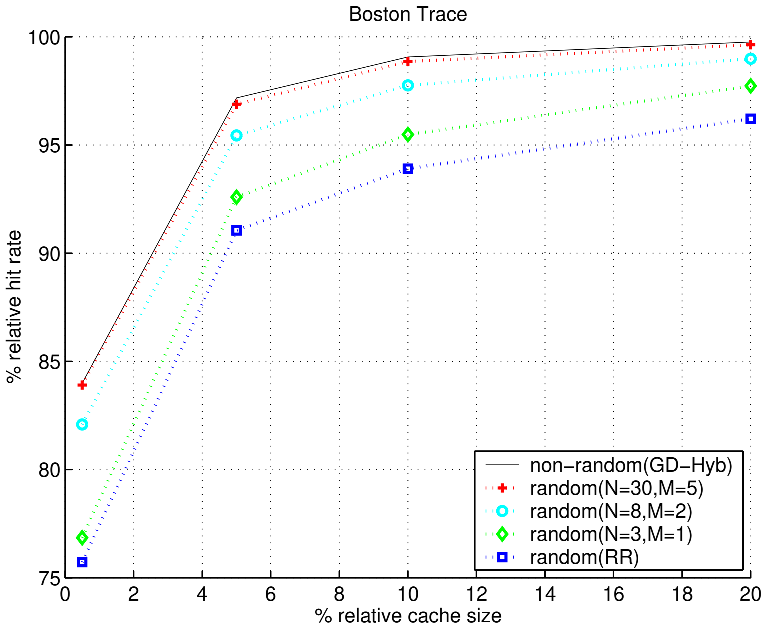

Figure 3.7: Hit rate comparison between GD-Hyb and RGD-Hyb.

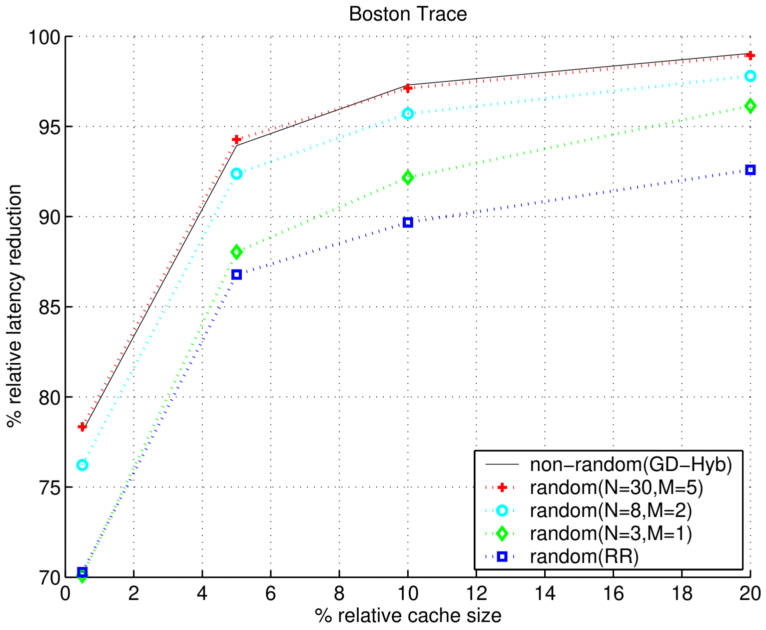

Figure 3.8: Latency reduction comparison between GD-Hyb and RGD-Hyb.

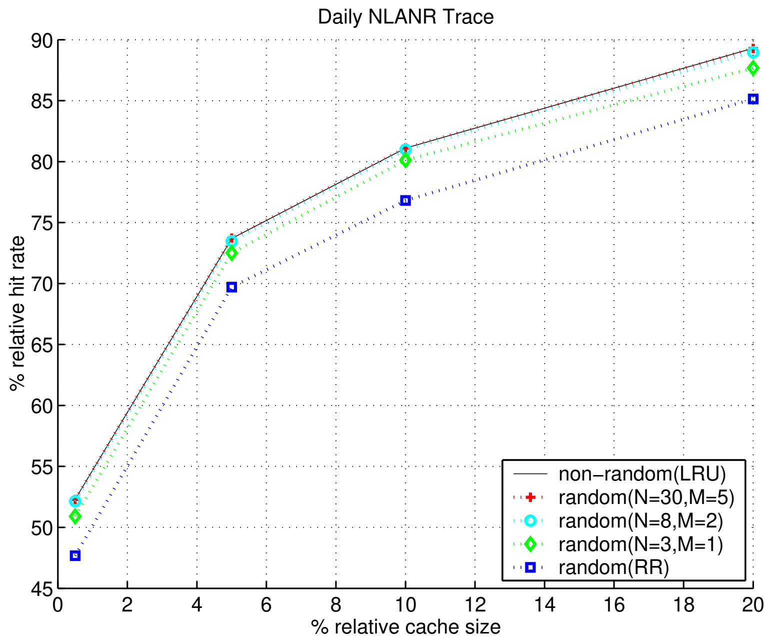

Figure 3.9: Hit rate comparison between LRU and RLRU.

Next, we examine how well the randomized algorithm can approximate GD-Hyb. We present results from the Boston trace. Figure 3.7 presents the ratio of the HR of GD-Hyb, RGD-Hyb, and RR, over the HR achieved by an infinite cache. As expected, the more the samples the better RGD-Hyb approximates GD-Hyb. The performance curve of RGD-Hyb for N=8 and M=2 is very close to the performance curve of GD-Hyb. For N=30 and M=5 the curves almost coincide.

Figure 3.8 presents the ratio of LR achieved by GD-Hyb, RGD-Hyb, and RR, over the LR achieved by an infinite cache. It is again the case that values of N and M as low as 8 and 2 respectively are enough for RGD-Hyb to perform very close to GD-Hyb.

The traces we used so far are taken from universities, and they are relatively small and old. Next, we present the results from the daily NLANR trace. Our goal is twofold: to evaluate the randomized algorithm under more recent and larger traces, and examine the performance of RLRU and RGD-Hyb in the same trace.

Figures 3.9 and 3.10 present the ratio of HR of various schemes over the HR achieved by an infinite cache. The former figure compares LRU to RLRU, and the latter GD-Hyb to RGD-Hyb. In Figure 3.9 RLRU nearly matches LRU for N and M as small as 8

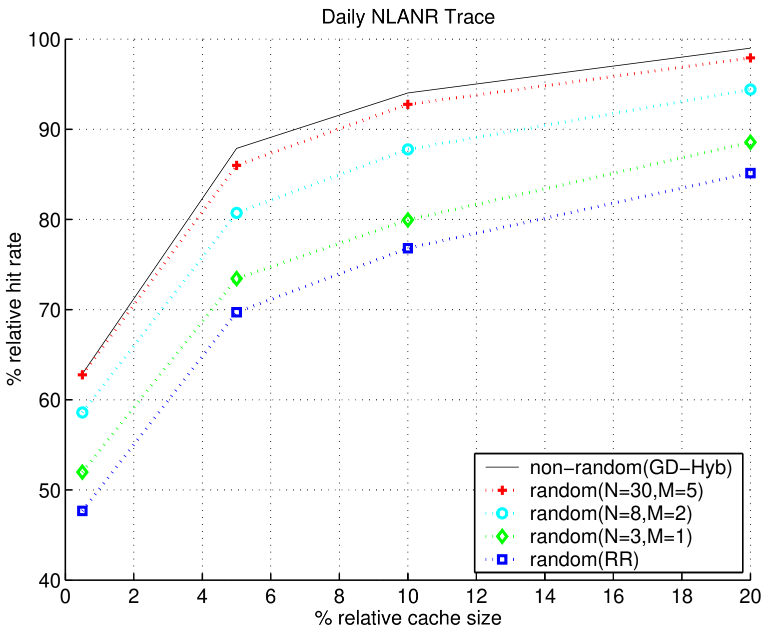

Figure 3.10: Hit rate comparison between GD-Hyb and RGD-Hyb.

and 2 respectively. In Figure 3.10 RGD-Hyb requires 30 samples and a memory of 5 to approximate GD-Hyb very closely, but its performance is superior to LRU. Indeed GD-Hyb achieves around 100% of the infinite cache performance while LRU achieves below 90%. Note that in Figure 3.10 RR's hit rate is 15% worse than GD-Hyb's whereas in Figure 3.7 it is only 4% worse. This is due to the different statistical characteristics of the two traces.

Figures 3.11 and 3.12 present the ratio of LR of various schemes over the LR achieved by an infinite cache. The former figure compares LRU to RLRU, and the latter GD-Hyb to RGD-Hyb. In Figure 3.11 RLRU nearly matches LRU for N and M as small as 3 and 1 respectively. In Figure 3.12 RGD-Hyb performs slightly better than GD-Hyb. This somewhat unexpected result is due to GD-Hyb's suboptimal eviction decisions with respect to LR under this trace, that could be removed by fine-tuning the two constants in its utility function. [13] Despite this, GD-Hyb performs better than LRU with respect to LR, since LRU does not take into account any latency information.

[13]Since all the deterministic cache replacement schemes rely on heuristics to predict future requests, their eviction decision may be suboptimal. Thus, the randomized approximations of these algorithms may occasionally perform better.

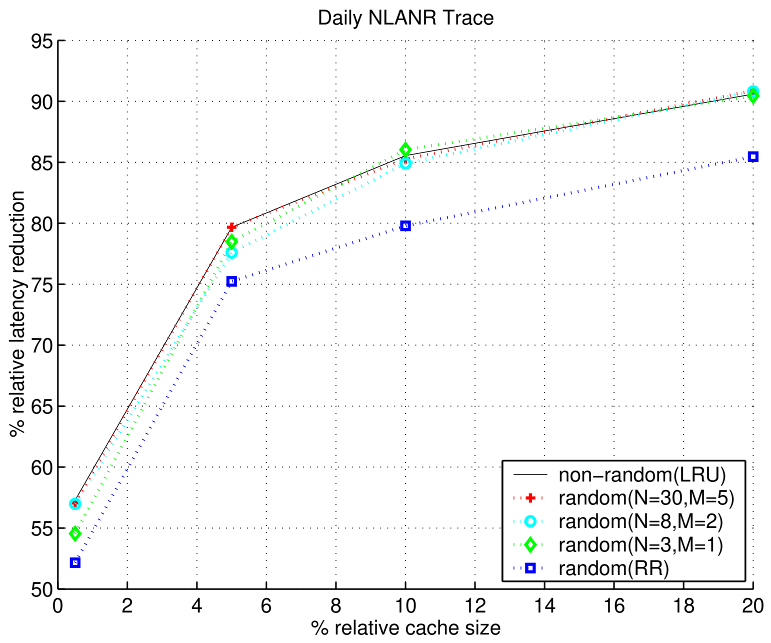

Figure 3.11: Latency reduction comparison between LRU and RLRU.

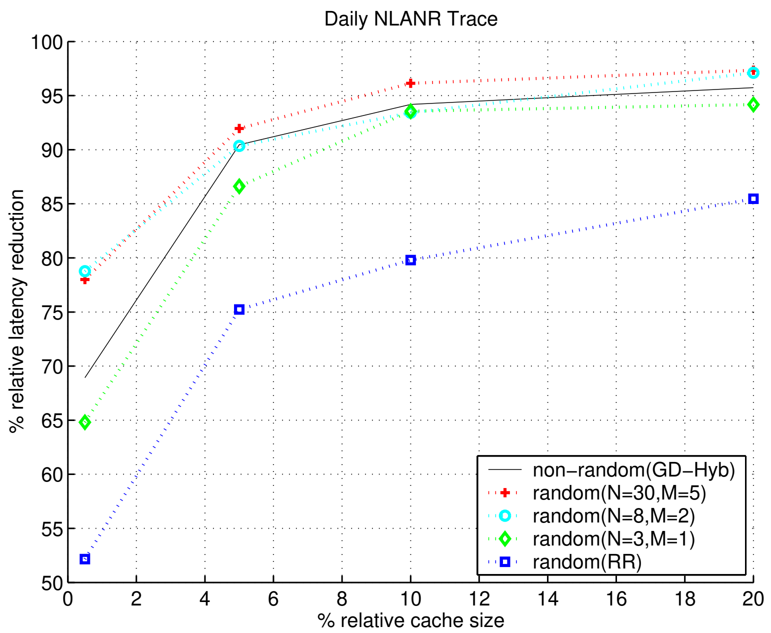

Figure 3.12: Latency reduction comparison between GD-Hyb and RGD-Hyb.

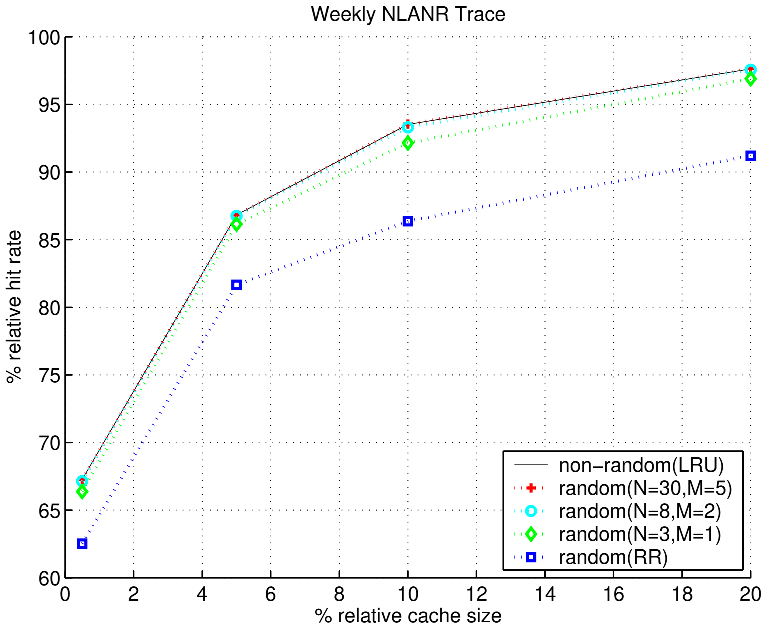

Figure 3.13: Hit rate comparison between LRU and RLRU.

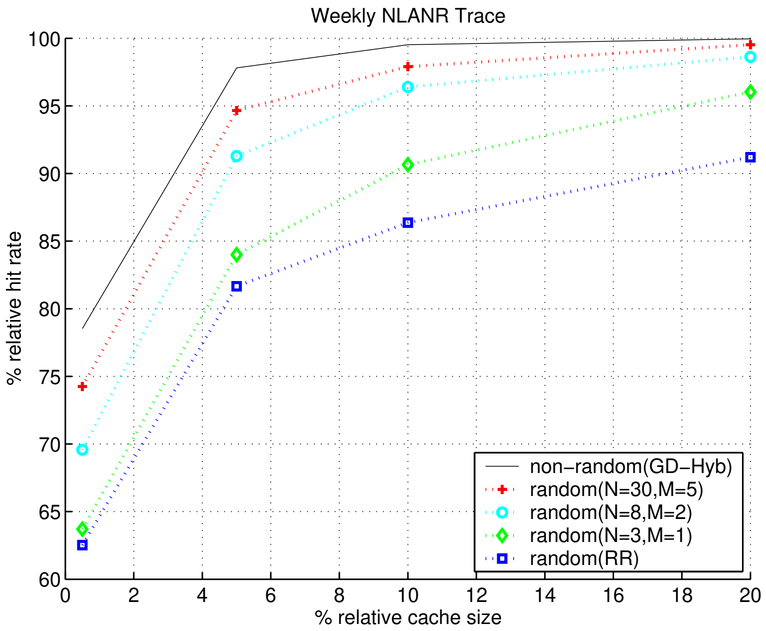

Figure 3.14: Hit rate comparison between GD-Hyb and RGD-Hyb.

The last trace from which we present results is the weekly NLANR trace. This trace consists of roughly 3 million requests and thus approximates reality better. The cache sizes used in the simulations are now quite large. For example, 20% of the minimum size required to avoid any evictions corresponds to 4GB.

Figures 3.13 and 3.14 present the ratio of HR of various schemes over the HR achieved by an infinite cache. The former figure compares LRU to RLRU, and the latter compares GD-Hyb to RGD-Hyb. Since these figures are similar to Figures 3.9 and 3.10, the performance of the randomized algorithm appears to be unaffected by the trace size. Note that the difference in the performance of LRU and GD-Hyb is not significant for a relative cache size of 20%, but for smaller cache sizes it is.

Figures 3.15 and 3.16 present the ratio of BHR of various schemes over the BHR achieved by an infinite cache. The former figure compares LRU to RLRU, and the latter compares GD-Hyb to RGD-Hyb. The randomized algorithm works well with respect to BHR too. Note that RGD-Hyb performs slightly better than GD-Hyb for smaller cache sizes and more importantly, LRU performs better than GD-Hyb. This somewhat unexpected result is caused because GD-Hyb makes relatively poor choices in terms of BHR by design, since it has a strong bias against large size documents even when these documents are popular. This suboptimal performance of GD-Hyb is inherited from SIZE [106] and Hybrid [110] and could be removed by fine-tuning. All the three schemes trade in HR for BHR.

Recently, an algorithm called GreedyDual* (GD*) has been proposed [55], that achieves superior HR *and* BHR when compared to other web cache replacement policies. This algorithm is a generalization of GD-Size. GD* adjusts the relative worth of long-term popularity versus short-term temporal correlation of references. It achieves this by dynamically adapting to the degree of temporal correlation of the web request streams. GD* requires a priority queue to be efficiently implemented and thus it is a good candidate to be approximated by our randomized algorithm.

From the figures above, it is evident that the randomized versions of the schemes can perform competitively with very small number of samples and memory. Since all the online cache replacement schemes rely on heuristics to predict future requests, it appears that it is not necessary to exactly mimic their behavior in order to achieve high performance. Instead, it usually suffices to evict a document that it is within a reasonable distance from the least useful document.

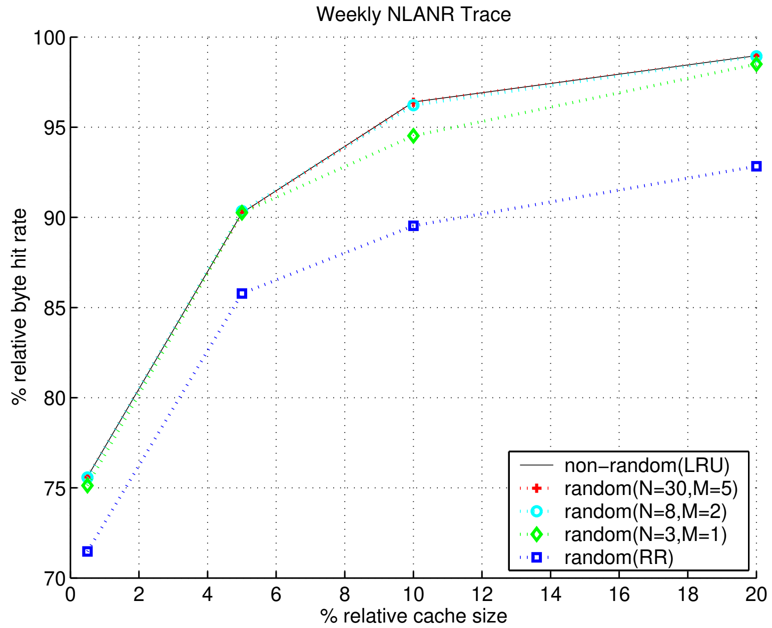

Figure 3.15: Byte hit rate comparison between LRU and RLRU.

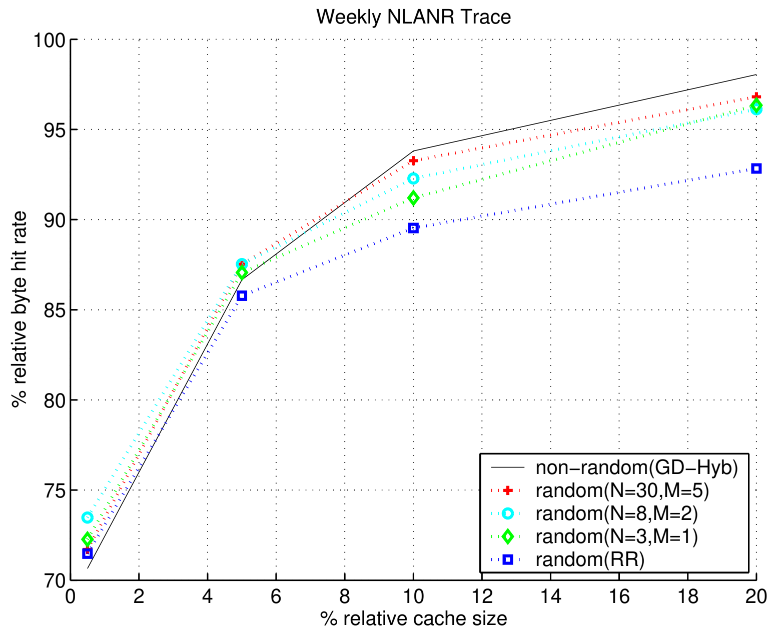

Figure 3.16: Byte hit rate comparison between GD-Hyb and RGD-Hyb.

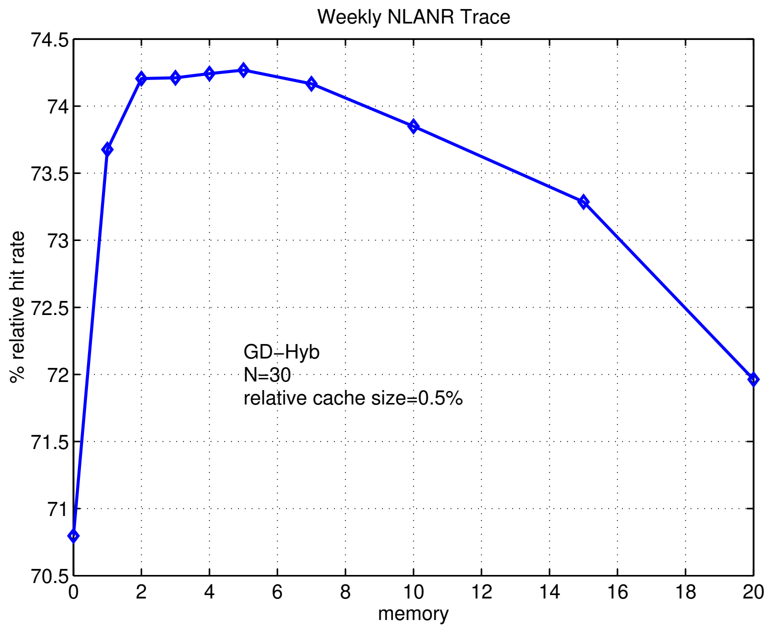

Figure 3.17: Hit rate as a function of memory.

In Sections 3.2 and 3.3, we studied the probability of error of the randomized algorithm as a function of the memory, and found that for small values of the memory there is a large reduction in the error probability, while as the memory increases further the performance deteriorates. Figure 3.17 plots the relative hit rate of RGD-Hyb as a function of the memory. In this plot, the randomized scheme uses a total of 30 samples, and operates to a cache of relative size 0.5%. For small values of the memory the hit rate increases rapidly, while as the memory increases further the hit rate deteriorates, which is in accordance with the theoretical findings. The plots are similar for LRU, other values of N, and different cache sizes.

There are two more issues to be addressed. First, we estimate the probability that documents retained in memory are accessed between iterations. This event very much depends on the request patterns and is hard to analyze exactly. Instead, we use the simulations to estimate the probability of its occurrence and find it to be less than 10^{-3}. The reason why this probability is so small is that our policy for retaining samples in memory, deliberately chooses the best eviction candidates which are not likely to be requested in the future.

Second, we verify that the randomized versions of the schemes do not produce "dead" documents. Due to the sampling procedure, the number of sampling times that a document is not chosen follows a geometric distribution with parameter roughly equal to N over the total number of documents in the cache. This is around 1/100 in our simulations. Hence, the probability that the best eviction candidates are never chosen is zero, and they are chosen once every 100 sampling times or so.

3.5 Implementation Issues

In the previous sections we established that a small number of samples, 6 fresh and 2 carried from one iteration to the next, is sufficient for the randomized algorithm to have good performance. In this section we discuss in detail the implementation savings by using the randomized algorithm.

There are two main operations that a web cache has to support; access to arbitrary cached documents, and eviction of relatively useless documents to make room for new documents.

State-of-the-art web caches access documents through a hash table [16], which has constant time lookups. Also, this hash table is stored in the RAM [16], which ensures that any document can be obtained in a single disk read. Thus, accessing arbitrary cached documents is done very efficiently. This operation is orthogonal to the eviction scheme used.

Different eviction schemes have different implementation requirements. As previously discussed, LRU can be implemented with a linked list that maintains the temporal order in which the cached documents were accessed. High performance algorithms like GD-Hyb require a priority queue to be implemented efficiently. Thus, for every insertion, they perform $O(logK)$ operations, where K is the number of documents in the cache. The size of a typical web cache today is tens of GBs. Since the average size of web documents is close to 10KB, K is typically many millions [16]. These schemes perform an insertion operation at every access time, since even at hit times the utility value of the document changes and the document should be reinserted at a new position.

The implementation savings due to the randomized eviction scheme are the following: First, the randomized scheme saves memory resources by not maintaining a data structure for eviction purposes. However, the parameters used by the utility function, like the frequency and recency of use, still need to be stored in order to be available when the document is chosen as a sample. Thus, for every document, there will be a corresponding object in RAM holding its utility parameters.

Second, the proposed randomized algorithm draws about 6 fresh samples per eviction time, instead of performing one insertion and one deletion operation in a priority queue per access time. A sample can be easily obtained in constant time by randomly choosing one of the objects that hold the utility parameters of the documents, or one of the entries of the hash table used to access the documents. [14] Thus, drawing random samples is cheaper than updating a priority queue.

Suppose the hit rate is around 50%. Then, there is one miss in every two accesses. Assuming there is no correlation between the size of a document and its popularity [15], every miss causes on average one eviction. Thus, priority queue updates take place twice as often as random sampling. For higher hit rates the CPU savings increase further, for example, for hit rates close to 75%, priority queue updates take place four times as often as random sampling. [15]

Finally, the randomized algorithm does not need to recompute the utility value of a cached document every time it is accessed. It performs this operation only with the samples obtained at eviction times if their utility value is not up-to-date. However, as a consequence, whenever RGD-Hyb computes utility values, it uses the minimum utility value among the samples instead of the global minimum among all documents. The simulations in Section 3.4 take this into account and show that it causes no performance degradation.

This work focuses on implementation savings with respect to CPU and memory resources. In some systems, the disk I/O resources are more important. Eviction schemes in such systems typically aim to reduce the number of and the time taken by disk reads/writes, and ignore parameters like the recency of use of a document. Using the ideas presented here it is possible to take such parameters into account at a minimal overhead, and thus increase the hit rate without trading-off disk I/O resources.

3.6 Conclusions

Motivated by the need to reduce the complexity of data structures required by good document replacement schemes, we have introduced a randomized algorithm that requires no

[14] Sampling is done without accessing the disk. It is only after the best sample is identified, that the corresponding document is accessed on the disk in order to be evicted.

[15] Hit rates up to 70% are quite common [55], [92].

data structure. This saves both the memory used up by the data structure and the processing power needed to update it.

Being randomized, the performance of our algorithm depends crucially on the quality of the samples it obtains. We observe that retaining the best unused samples from one iteration to the next improves the quality of the samples and leads to a dramatic improvement in performance. An analysis of the algorithm provides insights about its features and a formula for determining the right parameters for approximating the performance of any deterministic replacement scheme as closely as desired. Trace-driven simulations show that a small number of samples, 6 fresh and 2 carried from one iteration to the next, is sufficient for a good performance.

We believe that the idea of using memory to improve the quality of samples in iterative randomized algorithms is of general interest. While we have applied this idea to cache replacement policies [85, 86, 87], it has also been applied to other iterative randomized algorithms of interest in networking (e.g. load balancing [97], switch scheduling [43, 96, 104]).

3.7 Appendix

Proof of Lemma 3.1: Convexity follows from showing that the second derivative of the cumulative number of drops is non-negative:

$$D^{''}(t) \stackrel{\triangle}{=} D(t, M-1) - 2D(t, M) + D(t, M+1) \geq 0 \text{ for all } t \geq 0. \tag{3.7}$$

Essentially, this reduces to comparing the cumulative number of drops up to time t, $D(t, M-1)$, $D(t, M)$, and $D(t, M+1)$, from systems with buffer sizes $M-1$, M, and $M+1$, respectively. The three systems have identical arrival processes and start with empty buffers at time 0. The proof is an induction on arrival times, since these are common to all three systems, and hence this is equivalent to inducing over time. Define $d^{''}(a)$ to be the instantaneous difference in drops caused by the common arrival a which occurs at time t_a. Then $D^{''}(t) = \sum_{a:t_a \leq t} d^{''}(a)$.

To determine $d^{''}(a)$, consider the situation depicted in Figure 3.18. The three buffers are placed next to each other, in increasing order of size. The shaded boxes in the figure represent occupied buffer spaces and the white boxes represent empty buffer spaces. If all three buffers have empty spaces, then no drops will occur. Thus, it suffices to consider cases where at least one of the buffers is full.

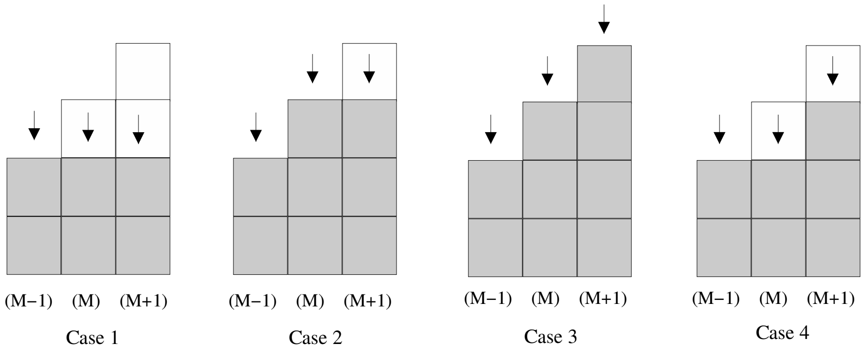

Figure 3.18: Possible cases of buffer occupancies for Lemma 3.1.

We claim that there are only four possible cases to consider where at least one buffer is full, and these are depicted in Figure 3.18. The following general observation, whose proof is inductive, easily verifies the claim. Consider any two systems A and B with buffer sizes B_A and $B_B = B_A + 1$, respectively. Suppose that A and B have identical arrival processes and there is exactly one departure from each non-empty system in each time slot. Then the buffer occupancies of system A and B, denoted by BO_A and BO_B respectively, will always satisfy $BO_A = BO_B$ or $BO_A = BO_B - 1$. Therefore, the only cases to consider are the ones shown in Figure 3.18.

An inspection of the four cases shows that for each common arrival a the values of $d^{''}(a)$ are given by (1) $d^{''} = 1$, (2) $d^{''} = -1$, (3) $d^{''} = 0$, and (4) $d^{''} = 1$. Thus, under Cases 1, 3 and 4 $d^{''} \geq 0$, and the only troublesome case is 2. Note, however, that every instance of Case 2 is *preceded* by at least one instance of Case 1. Therefore, the negative values of Case 2 are offset by the positive values of Case 1 and it follows that $D^{''}(t) \geq 0$ for all t. ■

Proof of Lemma 3.2: We need to show

$$\lim_{t\to\infty} \frac{1}{t}(D(t, \lambda(M-1)) - 2D(t, \lambda(M)) + D(t, \lambda(M+1))) \geq 0 \tag{3.8}$$

by considering three systems with same buffer sizes, and arrival rates that drop linearly with M.

In particular, the arrival processes A_{M-1}, A_M, and A_{M+1} to these systems are Binomially distributed with average rates $\lambda(M-1) = \lambda(M+1) + 2\frac{n}{100}$, $\lambda(M) = \lambda(M+1) + \frac{n}{100}$, and $\lambda(M+1) = (N-M-1)\frac{n}{100}$ respectively. Note that the processes A_{M-1} and A_M stochastically dominate [16] A_{M+1}. Thus, we can use a coupling argument [62] to categorize arrivals as follows:

(a) *Common.* An arrival occurs at all three systems. Common arrivals are Binomially distributed with average rate $(N-M-1)\frac{n}{100}$.

(b) *Single.* An arrival occurs only at the system with buffer size $M-1$. Single arrivals are Bernoulli(n/100).

(c) *Double.* An arrival occurs at the two systems with buffer sizes $M-1$ and M and there is no arrival at the system with buffer size $M+1$. Double arrivals are also Bernoulli(n/100).

Define $d''(a)$ to be the instantaneous difference in drops caused by arrival a which occurs at time t_a. Then, $D(t, \lambda(M-1)) - 2D(t, \lambda(M)) + D(t, \lambda(M+1)) = \sum_{a:t_a \leq t} d''(a)$. If all three buffers have empty spaces, then no drops will occur. Thus, it suffices to consider cases where at least one of the buffers is full.

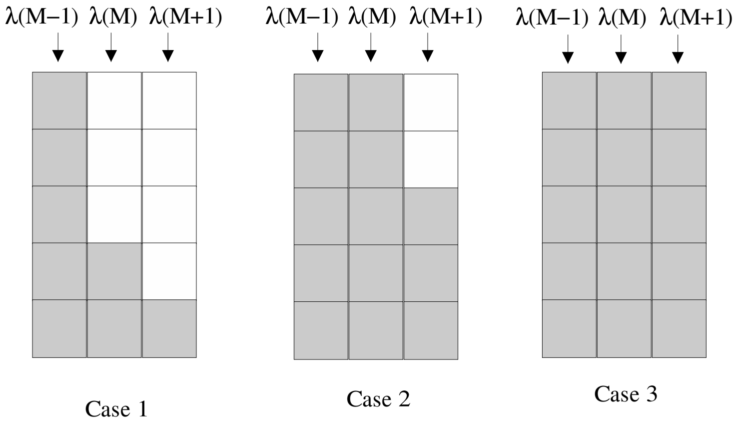

Figure 3.19: Possible cases when buffer sizes same.

Let $BO(M-1)$, $BO(M)$, and $BO(M+1)$ be the buffer occupancy of a buffer with average arrival rate $\lambda(M-1)$, $\lambda(M)$, and $\lambda(M+1)$ respectively. Note that for any time t,

[16] A process X stochastically dominates Y if and only if $P(Y < x) \leq P(X < x)$ for all x.

$BO(M-1) \geq BO(M) \geq BO(M+1)$. There are three cases to consider as shown in In Figure 3.19. Let C be the common buffer size. Case 1 is characterized by $BO(M-1)-BO(M) > 0$, $BO(M-1) = C$, Case 2 by $BO(M) - BO(M+1) > 0$, $BO(M) = C$, and Case 3 by $BO(M+1) = C$.

In Case 1, $d^{''}(a) = 1$ for common, single, and double arrivals. In Case 2, $d^{''}(a)$ equals 1 for single arrivals, and -1 for double and common arrivals. In Case 3, $d^{''}(a)$ equals 1 for single, 0 and common, and -1 for double arrivals. Since single and double arrivals are identically distributed, as $t \to \infty$ their $d^{''}$'s cancel out in Cases 2 and 3. Thus, it suffices to show that the negative effect of common arrivals in Case 2 is cancelled out by the positive effect of common, single, and double arrivals in Case 1.

We say a single departure takes place if $BO(M-1) = 1$, $BO(M) = 0$, $BO(M+1) = 0$, and a there is a departure. We say a double departure takes place if $BO(M-1) = 1$, $BO(M) = 1$, $BO(M+1) = 0$, and there is a departure.

Assertion 1: $BO(M-1) - BO(M)$ changes at most by one per arrival or departure. $BO(M-1) - BO(M)$ is increased by one, if and only if a single arrival is not dropped. $BO(M-1) - BO(M)$ is decreased by one, if and only if a common or a double arrival comes in Case 1, or there is a single departure.

Assertion 2: $BO(M) - BO(M+1)$ changes at most by one per arrival or departure. $BO(M) - BO(M+1)$ is increased by one, if and only a double arrival is not dropped, or a double arrival comes in Case 1. $BO(M) - BO(M+1)$ is decreased by one, if and only if a common arrival comes in Case 2, or there is a double departure.

For an arrival or a departure event A, let $\langle A \rangle \triangleq \lim_{t\to\infty} \#(A[0,t])/t$ be the number of times the event takes place in the interval $[0, t]$ divided by t as $t \to \infty$. Using this notation, let

$\langle SD \rangle \triangleq \langle$single departure$\rangle$,
$\langle DD \rangle \triangleq \langle$double departure$\rangle$,
$\langle SAND \rangle \triangleq \langle$single arrival not dropped$\rangle$,
$\langle DAND \rangle \triangleq \langle$double arrival not dropped$\rangle$,
$\langle SA1 \rangle \triangleq \langle$single arrival in Case 1$\rangle$,
$\langle DA1 \rangle \triangleq \langle$double arrival in Case 1$\rangle$,
$\langle CA1 \rangle \triangleq \langle$common arrival in Case 1$\rangle$,
$\langle CA2 \rangle \triangleq \langle$common arrival in Case 2$\rangle$.

Assertion 3: $\langle SD \rangle < \langle DD \rangle$, $\langle SAND \rangle = \langle DAND \rangle$, $\langle SA1 \rangle = \langle DA1 \rangle$.

Assertion 1 implies

$$\langle SAND\rangle = \langle CA1\rangle + \langle DA1\rangle + \langle SD\rangle.$$

Assertion 2 implies

$$\langle DAND\rangle + \langle DA1\rangle = \langle CA2\rangle + \langle DD\rangle.$$

Using the above two equations and Assertion 3, we get

$$\langle CA1\rangle + \langle DA1\rangle + \langle SA1\rangle > \langle CA2\rangle \tag{3.9}$$

and thus the negative effect of common arrivals in Case 2 is cancelled out by the positive effect of common, single, and double arrivals in Case 1. ■

Proof of Lemma 3.3: We need to show

$$\begin{aligned}\lim_{t\to\infty}\frac{1}{t}(\quad D(t,M-1,\lambda(M-1)) \quad & - \\ 2D(t,M,\lambda(M)) \quad & + \\ D(t,M+1,\lambda(M+1)) \quad) \quad & \geq \quad 0\end{aligned} \tag{3.10}$$

by considering three systems of buffer sizes $M-1$, M and $M+1$ and average rates $\lambda(M-1)$, $\lambda(M)$ and $\lambda(M+1)$ respectively. Let $D(M,\lambda(M)) \stackrel{\triangle}{=} \overline{D}(M) = \lim_{t\to\infty} D(t,M,\lambda(M))/t$.

Lemma 3.2 implies

$$D(M-1,\lambda(M-1)) - 2D(M-1,\lambda(M)) + D(M-1,\lambda(M+1)) \geq 0. \tag{3.11}$$

We need to show

$$D(M-1,\lambda(M-1)) - 2D(M,\lambda(M)) + D(M+1,\lambda(M+1)) \geq 0. \tag{3.12}$$

By adding and subtracting $2D(M-1,\lambda(M))$ and $D(M-1,\lambda(M+1))$ in Equation (3.12) and using Equation (3.11), it is easy to see that it suffices to show

$$\begin{gathered}2(D(M-1,\lambda(M)) - D(M,\lambda(M))) \geq \\ D(M-1,\lambda(M+1)) - D(M+1,\lambda(M+1)).\end{gathered}$$

By adding and subtracting $D(M,\lambda(M+1))$ in the right hand side of this equation, it is easy to see that is suffices to show

$$D(M-1,\lambda(M)) - D(M,\lambda(M)) \geq$$
$$D(M-1,\lambda(M+1)) - D(M,\lambda(M+1)) \quad \text{and} \tag{3.13}$$
$$D(M-1,\lambda(M+1)) - D(M,\lambda(M+1)) \geq$$
$$D(M,\lambda(M+1)) - D(M+1,\lambda(M+1)). \tag{3.14}$$

Lemma 3.1 implies Equation (3.14) holds. To see why Equation (3.13) holds, categorize arrivals to basic arrivals with rate $\lambda(M+1)$ and excess arrivals with rate $\lambda(M) - \lambda(M+1)$, and notice that there is no way for excess arrivals to increase $D(M, \lambda(M))$, without also increasing $D(M-1, \lambda(M))$. ■

Chapter 4

A Model that Generates Web Request Streams

Algorithms devised for web caches need to take advantage of specific characteristics present in the request sequence for efficient performance. For example, cache replacement policies exploit the temporal locality between successive requests and the long-term popularity of web documents during eviction times. Prefetching policies take advantage of correlations present in inter-document requests (see [74], for example). Since algorithm design relies on the features inherent in HTTP request sequences, it is important to obtain a simple and accurate model of such sequences. This allows one to understand the impact of various characteristics (like temporal correlations, document popularity, etc) on the performance of the cache, and guides the design of better algorithms.

Broadly, there are three key features of web traffic that we wish to model. They are: (i) temporal correlations – this captures the likelihood of requesting a document in the near future, given that it has been currently requested, (ii) long-term document popularity – this captures the probability that a document is likely to be requested relative to other documents, and (iii) spatial (inter-document) correlations – this models the chance that document j will be requested next, given that document i has been requested now. In this work we focus on the relationship between temporal correlations and popularity as it relates to designing cache replacement schemes, and only briefly discuss spatial correlations.

Previous work on obtaining models of web traffic have observed that the distribution of the long-term popularity of documents is Zipf-like (see [14] and the references therein). The authors in [55] point out that a Zipf-like popularity distribution alone doesn't suffice to capture temporal locality, the request sequence needs to be correlated in time. Our approach simultaneously models both temporal correlations and document popularity. Temporal correlations are captured using a Markov model for the request sequence. We do not assume specific parameters for the Markov chain, instead we infer them from the real traces we wish to model. Similarly, we also infer the distribution of the long-term popularity from the traces (and find that it is indeed Zipf-like).

Observations about the characteristics of web trace sequences have influenced the design of document replacement schemes. For example, the least recently used (LRU) algorithm is known to exploit temporal correlations present in the request sequence [99]. Hence, it is suitable for use when the trace has a high degree of temporal correlation (or locality). On the other hand, when the trace has a low temporal correlation, the least frequently used (LFU) algorithm exploits long-term document popularity to achieve high hit rates. Indeed, it is easy to see that when request sequences are independent and identically distributed, LFU achieves the highest hit rates [1] [25].

Real web request sequences, however, are influenced both by temporal correlations and document popularity. This observation and clever heuristics have led to the proposal of replacement algorithms which combine features of both LRU and LFU [3, 17, 54, 55, 61]. Our model adds to this body of work in three ways: (a) It allows us to quantify the degree to which real traces are influenced by temporal correlations and document popularity, (b) it allows us to explore extreme cases by considering traces with different mixtures of temporal correlation and document popularity, and (c) this, in turn, points to a way of designing algorithms that trade-off locality and popularity.

Another contribution of our work is for designing good workload generation tools [8, 15]. An important challenge for these tools is to capture temporal correlations present in web request traces. This is commonly done using the LRU stack model [98]. But this model is hard to analyze; as a result, it is difficult to scale a trace of some length into a longer trace while retaining the same degree of short-term temporal correlation. With our model one can first capture the degree of temporal correlation in the short trace and then generate longer traces with the same characteristics.

It has been observed that in addition to short-term correlation and long-term popularity, other attributes of a document, such as its size and the cost of fetching it from the origin-server, play an important role in the performance of web replacement schemes. Since the

[1] This result holds under the assumption that all documents are of the same size.

focus of this work is on the trade-off between locality and popularity, we initially ignore these other attributes. Later, we comment on how to incorporate these parameters in designing good eviction schemes. As a final comment, we limit our discussion to static documents and do not consider documents that change dynamically over time.

The chapter is organized as follows. Section 4.1 introduces the model. Section 4.2 presents a method to extract the model parameters from real traces. In Section 4.3 we generate synthetic traces and verify that they exhibit the desired statistical properties. Further, we validate the model by comparing the hit rates for real traces and their synthetic counterparts under various caching algorithms. Extensions to the basic model are presented in Section 4.4. Based on the trace model, in Section 4.5 we formulate the problem of obtaining an optimal replacement algorithm with respect to hit rate as a Markov decision process, and obtain an online replacement algorithm that is optimal conditioned on the state of the Markov process. This algorithm, however, relies on knowledge of trace characteristics and it is hard to use in practice. Thus, we introduce γ-LRU, a class of practicable replacement policies parameterized by γ with LRU and LFU obtained for extreme values of γ, and study its performance. Finally, we address implementation issues related to γ-LRU, like taking into account its data structure requirement and the variability in document sizes.

4.1 The Model

In this section we introduce our model for capturing long-term popularity and short-term temporal correlation in web request sequences.

Index all documents in decreasing order of popularity. Let p_i be the probability of requesting the i^{th} most popular document. We shall call the probability distribution $\{p_i\}$ the distribution of the long term popularity of the documents. In practice, $\{p_i\}$ is known to be Zipf-like [14]; i.e., $P[Y_n = i] = p_i \propto 1/i^\theta$, $0 < \theta < 1$.

Let X_n be the document requested at time $n \geq 1$ (time is slotted, one request per slot), and Y_n, $n \geq 1$, be a sequence of random variables that are independent and identically distributed according to $\{p_i\}$. We propose the following model for web requests for $n > h$:

$$X_n = \begin{cases} X_{n-1} & \text{with probability } \alpha_1 \\ X_{n-2} & \text{w.p. } \alpha_2 \\ \vdots & \\ X_{n-h} & \text{w.p. } \alpha_h \\ Y_n & \text{w.p. } \beta, \end{cases}$$

where

$$\beta + \sum_{i=1}^{h} \alpha_i = 1. \tag{4.1}$$

In the model we keep a history of the h most recent requests to capture short-term correlations. Y_n injects documents not recently requested and captures long-term popularity. The probability that document i is going to be requested at time n, given that it was not requested in the last h requests is simply βp_i. The probability that document i is going to be requested at time n, given that it was requested only once in the last h requests at time $n - j$ $(1 \leq j \leq h)$, equals $\alpha_j + \beta p_i$. In general,

$$P(X_n = i | X_{n-1}, \ldots, X_{n-h}) = \beta p_i + \sum_{j=1}^{h} \alpha_j 1_{(X_{n-j}=i)}. \tag{4.2}$$

For mathematical convenience, another way to interpret the model is the following. At each time slot, toss an $h + 1$-sided, biased coin to decide the value of X_n. Let T_n be a random variable indicating the outcome of the toss at time $n > h$. Then, $P(T_n = j) = \alpha_j$ for $1 \leq j \leq h$ and $P(T_n = h + 1) = \beta$. Using this notation, the model is described by the following equation:

$$X_n = \sum_{j=1}^{h} X_{n-j} 1_{(T_n=j)} + Y_n 1_{(T_n=h+1)}. \tag{4.3}$$

The parameters of the model are the history window h, the α_i's, the β, and the p_i's. Let $\overline{X}_n = (X_n, X_{n-1}, \ldots, X_{n-h+1})$ and $\overline{i} = (i_1, i_2, \ldots, i_h)$ be a vector of h requested documents. Since $P(\overline{X}_{n+1} = \overline{i} \,|\overline{X}_n, \overline{X}_{n-1}, \ldots) = P(\overline{X}_{n+1} = \overline{i} \,|\overline{X}_n)$, $\overline{X}_n$ is a Markov chain or equivalent X_n is an h^{th}-order Markov chain. The $\overline{X}_n$, also X_n, has a unique stationary distribution since its state space is finite and it is irreducible and aperiodic (for $\beta > 0$).

Lemma 4.1 *In stationarity, X_n is distributed like Y_n.*

Proof: Using Equation (4.3) we get

$$P(X_n = i) = \sum_{j=1}^{h} \alpha_j \cdot P(X_{n-j} = i) + \beta \cdot P(Y_n = i).$$

Now, in stationarity, X_n equals in distribution some random variable X. Therefore, $P(X_n = i) = P(X_{n-j} = i) = P(X = i)$ for all $j = 1, \ldots, h$, and we get

$$P(X = i) \cdot (1 - \sum_{j=1}^{h} \alpha_j) = \beta \cdot P(Y_1 = i)$$

for all i. Since $1 - \sum_{j=1}^{h} \alpha_j = \beta$, X has the same distribution as Y_1. (Recall that Y_n is i.i.d.) ■

4.2 Inferring Model Parameters

In this section we present techniques for inferring the parameters of the model, and use these techniques to obtain the parameters from real traces.

We infer $\{p_i\}$ from its empirical distribution; i.e. set p_i to equal the sample frequency of document i in the trace.

Inferring the α_i's and β is more involved. Let R be the number of requests in the trace. Suppose, for now, that a suitable value of the history parameter, h, has been chosen. (We present a procedure for determining h at the end of this section.) Let C_i, $1 \leq i \leq h$ be a counter that counts how many times the n^{th} request, $h < n \leq R$, is the same with the $(n-i)^{th}$ request. Denote by $c_i, 1 \leq i \leq h$, the ratio $\frac{C_i}{R-h}$. In other words,

$$c_i = \frac{\sum_{n=h+1}^{R} 1_{(X_n = X_{n-i})}}{R-h} \quad \text{for } 1 \leq i \leq h.$$

If the piece of real trace was indeed generated by the proposed Markov chain model, then for $1 \leq i \leq h$:

$$c_i \approx P(X_n = X_{n-i})$$

$$
\begin{aligned}
&= \sum_{j=1}^{h} P(X_{n-i} = X_{n-j}; T_n = j) + P(X_{n-i} = Y_n; T_n = h+1) \\
&= \sum_{j=1}^{h} P(X_{n-i} = X_{n-j}) P(T_n = j) + P(X_{n-i} = Y_n) P(T_n = h+1) \\
&= \alpha_i + \sum_{j=1, j \neq i}^{h} c_{|i-j|}\alpha_j + \beta P(X_{n-i} = Y_n). \qquad (4.4)
\end{aligned}
$$

Using the independence of Y_n and X_{n-i}, and the fact that X_{n-i} is distributed like Y_n in stationarity, $P(Y_n = X_{n-i}) = \sum_j P(Y_n = j | X_{n-i} = j) P(X_{n-i} = j) = \sum_j p_j^2$ for all $1 \leq i \leq h$.

Equation (4.4), for $1 \leq i \leq h$, together with Equation (4.1) form a linear system of $h+1$ equations with $h+1$ unknowns that can be solved to calculate α_i's and β. Use Equation (4.1) to substitute β with $1 - \sum_{i=1}^{h} \alpha_i$ in Equation (4.4). Then, the α_i's are computed by solving the linear system $A \cdot \alpha = B$ where $\alpha = (\alpha_i)$ is a $1 \times h$ vector, $B = (c_i - \sum_j p_j^2)$ is a $1 \times h$ vector, and A is an $h \times h$ matrix given by the following expression:

$$
A = \begin{pmatrix}
1 - \sum p_j^2 & c_1 - \sum p_j^2 & c_2 - \sum p_j^2 & \dots & c_{h-1} - \sum p_j^2 \\
c_1 - \sum p_j^2 & 1 - \sum p_j^2 & c_1 - \sum p_j^2 & \dots & c_{h-2} - \sum p_j^2 \\
c_2 - \sum p_j^2 & c_1 - \sum p_j^2 & 1 - \sum p_j^2 & \dots & c_{h-3} - \sum p_j^2 \\
\vdots & & & \ddots & \vdots \\
c_{h-1} - \sum p_j^2 & c_{h-2} - \sum p_j^2 & c_{h-3} - \sum p_j^2 & \dots & 1 - \sum p_j^2
\end{pmatrix}
$$

Once the α_i's are determined, β is computed from Equation (4.1).

What remains is a procedure for determining the proper history h. In theory, overestimating h, which results in a larger matrix, would still find the correct α_i's and history ($\alpha_i = 0$ for all $i > h$). However, in practice, larger history leads to more rounding and statistical errors, hence to small negative α_i's in the solution of the linear system. A solution to this is to start from an overestimated history, and lower its value until all the α_i's are positive.

To determine the proper history h we execute the following steps: (i) We start with a large value of h, traverse the trace, and compute the c_i's, (ii) randomly permute the trace to cancel any short-term correlations, traverse the permuted trace, and compute the corresponding normalized counters, denoted by $c_i^{'}$, (iii) output as an overestimated history

h	β	$\hat{\beta}$
500	0.789	0.787
1000	0.708	0.707
2000	0.616	0.617

Table 4.1: β values for various histories.

the number of c_i's that are larger than the average value of the $c_i^{'}$'s, [2] and (iv) lower the history until all the α_i's, as obtained from the solution of the linear system, are positive.

4.3 Generating Synthetic Traces

Synthetic traces can be generated using the model after inferring the parameters h, α_i, β, and p_i from a real trace. We experiment with daily proxy cache traces from the National Laboratory for Applied Network Research (NLANR) [70] recorded from September '99 to January '02. The length of the traces varies from 300000 to 600000 requests. All traces are from the PA site. [3]

Using the procedure described above, the proper history for most of the traces is computed to be around 5000. We also experiment with shorter histories, namely 500, 1000, 2000 and 3000. In the rest of this section we present results from one trace only, since there are no differences in the results obtained from the various traces.

Figure 4.1 presents the α_i's and β inferred from the trace for $h = 1000$. Notice that the α_i's are rapidly decreasing. Therefore, using a history that is smaller than the one obtained through the procedure above, suffices to capture the most essential short-term correlations.

To determine whether the synthetic traces exhibit the same long-term popularities and short-term temporal correlations with their real counterparts, we perform the following procedure: (i) We verify that they possess the same history with the real traces, (ii) infer the rest of the model parameters from the synthetic traces, denoted by $\hat{\alpha}_i$, $\hat{\beta}$, and $\hat{p}_i$, and (iii) compare these parameters to the parameters inferred from the real traces.

[2] To see why this gives an upper bound for the history, first note that, in theory, $c_i^{'} = \sum p_j^2$ for all i. Now, since the original trace exhibits some positive correlation, $c_i \geq \sum p_j^2$ for all i. Hence, since the c_i's are decreasing, one can deduce an upper bound for the history by this lower bound on the c_i's. Finally, note that we use the average value the $c_i^{'}$'s because, in practice, the $c_i^{'}$'s fluctuate around $\sum p_j^2$.

[3] NLANR traces are a collection of traces from many sites. We use the PA site because it has relatively long traces.

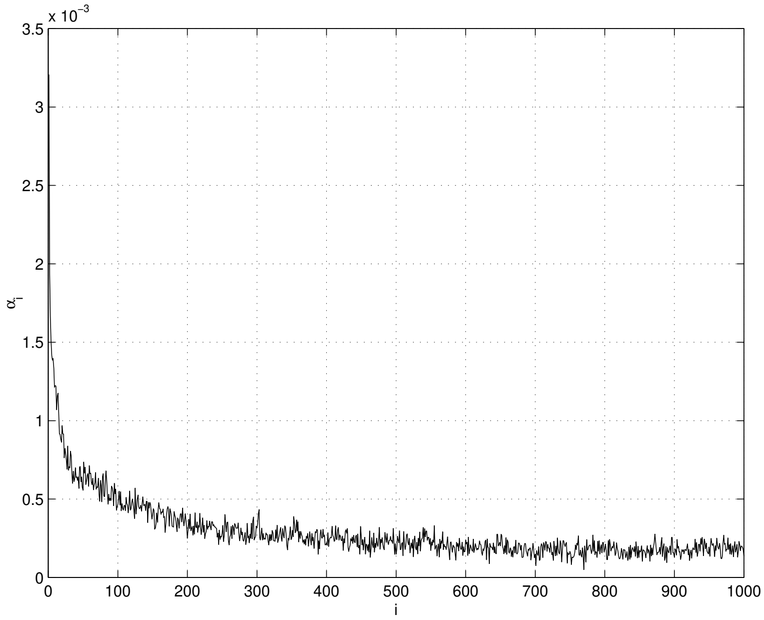

Figure 4.1: α_i's for the original trace (h=1000, $\beta = 0.708$).

In Table 4.1 we compare the values of β as obtained from the real and the corresponding synthetic trace. The values of β match very well for all histories shown. Figure 4.2 plots the $\hat{\alpha}_i$'s for the synthetic trace that is generated using the parameters of the real trace. By comparing Figures 4.1 and 4.2, it is evident that the plots match very well. [4]

In Figure 4.3 we plot the $\alpha_i' s$ of the real trace in logarithmic scale. The values of α_i roughly follow a power law with $\theta = 0.46$. We will use this observation to generate $\alpha_i' s$ for simulation purposes.

As expected, for popular documents $p_i \approx \hat{p}_i$ as it is shown in Figure 4.4. However, this is not the case for very unpopular documents since estimating the p_i's of such documents from the sample frequencies is not accurate. The insufficient statistics for unpopular documents, and in particular one-timers, lead us to treat them in a special manner. We will first elaborate on the problems associated with one-timers, and then, we will describe a heuristic to circumvent these problems.

The first issue with unpopular documents is a well-known problem of statistics, often referred to as the problem of estimating the number of classes [46] or distinct values [21]

[4] If α_i's and $\hat{\alpha_i}$'s are plotted in the same plot, the plots are nearly indistinguishable with small differences only in the oscillations.

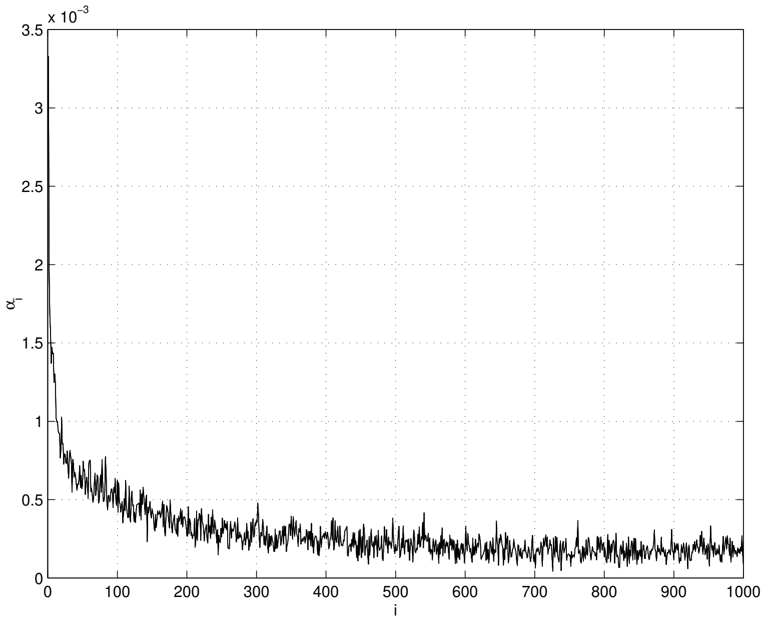

Figure 4.2: $\hat{\alpha}_i$'s for the synthetic counterpart of the original trace (h=1000, $\hat{\beta} = 0.707$).

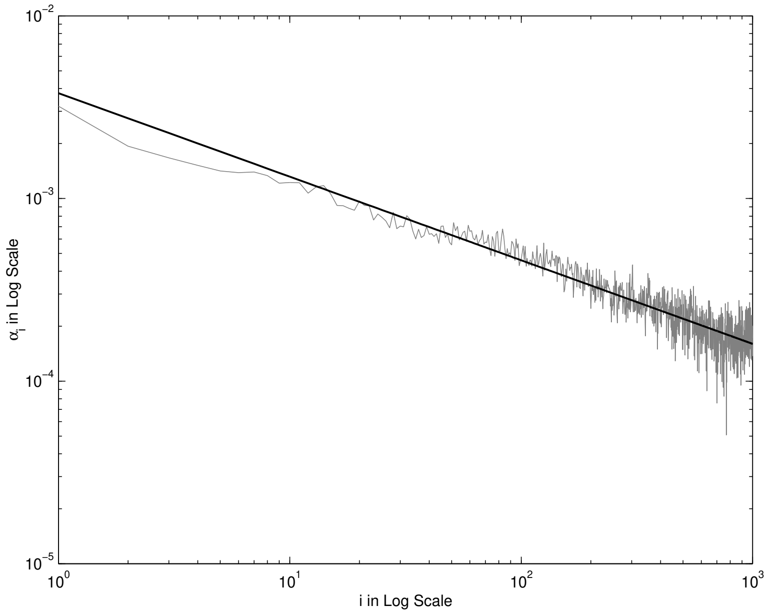

Figure 4.3: α_i's in logarithmic scale.

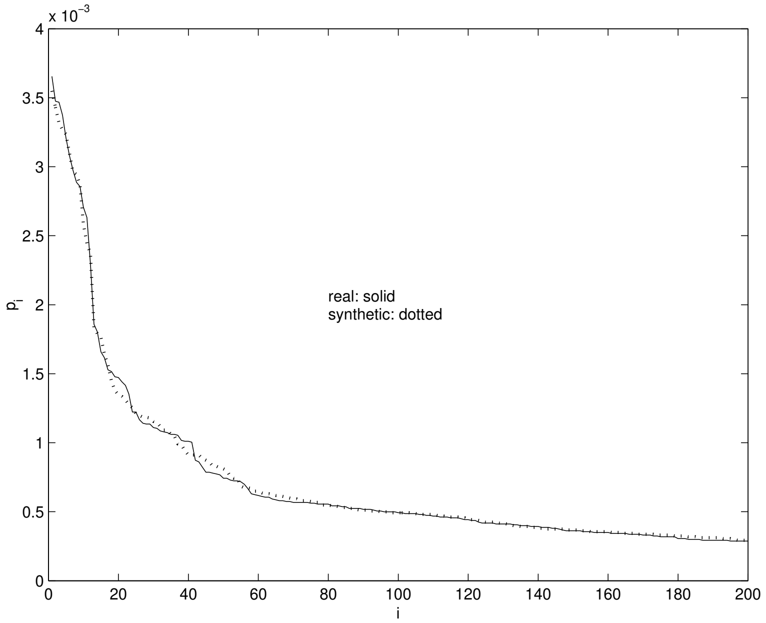

Figure 4.4: Long-term popularity comparison between the original trace and its synthetic counterpart for popular documents.

in a population. In the context of web traces, the problem is that when a trace is not long enough, there are many unpopular documents that are not present in the trace. If one uses the sample frequencies to generate a trace with the same length as the original one, the one-timers in the generated trace are going to be less than in the original. This is because the pool of available one-timers, from where the i.i.d sequence Y_n takes values, is a lot smaller than in reality. To fix this problem we need to estimate the actual number of unpopular documents available to the users from which the trace is recorded.

The second issue with unpopular documents is the following. Recall that in stationarity X_n is distributed like Y_n. However, since the available real traces are short, the statistics for unpopular documents are far from their stationary values, and the correlations inherent in the model force unpopular documents to be more popular. The problem is well exemplified with one-timers: Suppose a one-timer, say document j, is requested during the generation of a synthetic trace. Then, based on the model, the probability of requesting document j again in the next h time slots equals $1-\prod_{i=1}^{h}(\beta(1-p_j)+(1-\beta)(1-\alpha_i)) \approx 1-\prod_{i=1}^{h}(1-\alpha_i) \approx \sum_{i=1}^{h}\alpha_i = 1-\beta$. Hence, since this probability is positive, documents that appear only once in the real trace may appear more than once in the synthetic trace.

	LRU		LFU	
Trace	HR (real)	HR (synthetic)	HR (real)	HR (synthetic)
May 21^{st}, '01	32.2%	32.1%	29.4%	29.1%
May 23^{rd}, '01	31.7%	31.4%	28.5%	28.1%
January 19^{th}, '02	30.7%	31.1%	28.9%	28.7%
January 20^{th}, '02	32.7%	32.4%	31.1%	30.8%

Table 4.2: Hit rates comparison for real and synthetic traces under LRU and LFU.

We solve these problems as follows: Whenever a one-timer is requested during the generation of a synthetic trace, we record a special identifier instead of the identifier corresponding to the document. After the trace is generated, we parse it and replace all the occurrences of the special identifier with one-timers. [5]

4.3.1 Model Validation

A word on model validation is in order. The proposed model captures two important properties of web request streams: (i) the Zipfian nature of long-term popularity, and, (ii) the short-term temporal correlations. There are other properties that are not captured, e.g. inter-document correlations.

To validate the model, we should investigate if it captures all the important properties of request streams for a given application. In this work we use web caching as our application. Hence, we should compare the hit rates achieved by real traces and their corresponding synthetic traces, under various cache replacement schemes. Table 4.2 compares the hit rates for four traces under LRU and LFU. (We use online LFU; i.e. the popularity/frequency of a document at time t is inferred from the requests that occured at times $t' \leq t$.) The hit rates are close for all traces. In Section 4.5.3 we compare the hit rates under more sophisticated eviction schemes, after a thorough discussion on cache replacement policies that takes place in Section 4.5.

4.4 Extensions of the Basic Model

In this section we briefly comment on how to extend the basic model to take into account inter-document correlations. Further, we show how to extend the model to capture different degrees of temporal correlations for each document.

[5]It is easy to see that the number of occurrences of the special identifier will be very close to the total number of one-timers.

A simple way to model inter-document correlations is to associate with each document i a set of neighboring documents $N_i = \{j_1, j_2, \ldots\}$ that are strongly correlated with i. Let $|N_i|$ denote the cardinality of that set. Further, assume that all pairs of neighbors exhibit the same degree of inter-document correlation, and that these correlations die after one time slot. Then, it is straightforward to extend the basic model as follows:

$$X_n = \begin{cases} X_{n-1} & \text{with probability } \alpha_1 \\ X_{n-2} & \text{w.p. } \alpha_2 \\ \vdots & \\ X_{n-h} & \text{w.p. } \alpha_h \\ Y_n & \text{w.p. } \beta_1 \\ Z_n & \text{w.p. } \beta_2 \end{cases}$$

where $\alpha_1 + \cdots + \alpha_h + \beta_1 + \beta_2 = 1$, $Z_n = j_k$ w.p. $1/|N_{X_{n-1}}|$ for all $k = 1, \ldots, |N_{X_{n-1}}|$, and $N_{X_{n-1}} = \{j_1, j_2, \ldots, j_{|N_{X_{n-1}}|}\}$ is the set of neighbors of the document requested at time $n-1$. Notice that even this simplistic way to account for inter-document correlations requires us to identify a set of neighbors for each document, which can be quite burdensome in practice.

Recall from Equation (4.2) that the α_i's are the same for all documents. We generalize this by assuming that the probability of requesting document i, given the last h requests, equals

$$P(X_n = i | X_{n-1}, \ldots, X_{n-h}) = \beta_i p_i + \sum_{j=1}^{h} \alpha_{j,i} 1_{(X_{n-j}=i)},$$

where $\alpha_{1,i} + \cdots + \alpha_{h,i} + \beta_i = 1$ for all i. If document i_1 is more popular than document i_2, then α_{j,i_1} may be larger than α_{j,i_2} for some j's. In particular, the $\alpha_{j,i}$'s corresponding to one-timers are equal to zero for all j. Thus, the issue with the basic model where one-timers appear more frequently in generated traces than in original ones due to correlations, is resolved. Notice that this extension requires to keep track of h parameters for each document, which is also burdensome in practice.

In the interest of simplicity and tractability, we do not study further these extensions to the basic model. The closeness of the hit rates for real and synthetic traces under various replacement schemes (Tables 4.2, 4.6, and 4.7) justify this decision.

4.5 Caching Algorithms

As mentioned previously, researchers have proposed various caching algorithms that exploit both short-term temporal correlations and long-term popularity. Despite the good performance of these algorithms, the precise trade-offs between temporal correlations and popularity are not well-understood. Using the request model introduced in Section 4.1 we can shed light on these trade-offs, formulate the problem of obtaining an optimal algorithm with respect to hit rate as a Markov decision process, and obtain an online replacement algorithm that is optimal conditioned on the state of the Markov process. We also introduce γ-LRU, a class of practicable replacement policies parameterized by γ, $0 < \gamma \leq 1$, with LRU and LFU obtained for extreme values of γ. When γ is properly chosen, the performance of γ-LRU is close to the performance of the optimal policy.

4.5.1 Optimal Replacement Policies

We start by introducing some notation and terminology. Let K be the size of the cache. Fix a replacement policy r. Let $\overline{Z}_n^r$ be a random vector with the K elements $Z_{n,i}^r$, $i = 1, \ldots, K$, each of which holds the document at the i^{th} position of the cache at time n.

Let H_n^r be a random variable equal to 1 if there is a hit at time n, and zero otherwise. The long-term hit rate of a replacement policy r equals

$$HR^r \triangleq \lim_{N \to \infty} \frac{\sum_{n=1}^{N} H_n^r}{N}.$$

The H_n^r is a function of $\overline{Z}_n^r$ and X_n. Since $\overline{Z}_n^r$ depends only on the process X_n, and X_n is stationary and ergodic, the ergodic theorem [29] states that

$$\frac{\sum_{n=1}^{N} H_n^r}{N} \overset{a.s.}{\to} E(H_1^r) = P(H_1^r = 1). \tag{4.5}$$

Recall that $\overline{X}_n = (X_n, \ldots, X_{n-h+1})$. It is easy to see that $(\overline{Z}_n^r, \overline{X}_n)$ is a Markov chain with a finite state space S. Denote by π_s^r the stationary probability of being at state $s \in S$. [6] Then,

$$HR^r \overset{(4.5)}{=} P(H_n^r = 1) = \sum_{s \in S} \pi_s^r P(H_n^r = 1|s). \tag{4.6}$$

[6] The Markov chain $(\overline{Z}_n^r, \overline{X}_n)$ is aperiodic for $\beta > 0$ hence it has a stationary distribution.

A replacement policy r^* is optimal with respect to hit rate if it maximizes the stationary probability $P(H_n^r = 1)$ over all policies r. Finding r^* is the object of the theory of Markov Decision Processes [88], [93]. Using the vocabulary of this field, the Markov chain $(\overline{Z}_n^r, \overline{X}_n)$ is observed to be in a particular state s_n at time n. After observation of the state, an *action* must be chosen: which document to evict from the cache in case of a miss. [7] Based on the state s_n and the action chosen, a *reward* $R(s_n, r) = P(H_{n+1}^r = 1|s_n)$ is earned and the probability distribution for the next state is determined.

The action for every state is dictated by a stationary policy r. The problem of interest is to determine the policy that maximizes the average reward over an infinite time horizon, $\lim_{N\to\infty} E(\frac{1}{N}\sum_{i=1}^{N} R(s_i, r))$. It is easy to see by interchanging the limit and the expectation, [8] and using ergodicity, that the average reward is identical to the long-term hit rate. Hence,

$$r^* = \arg\max_r [HR^r] = \arg\max_r \left[\lim_{N\to\infty} E\left(\frac{1}{N}\sum_{i=1}^{N} R(s_i, r)\right)\right].$$

The theory of Markov Decision Processes provides a mathematical framework to identify optimal policies. However, usually only numerical solutions are possible. Furthermore, in many problems including the one we study, the computational requirements to obtain a numerical solution are overwhelming, because the number of states and/or policies is very large. In such situations, suboptimal approximate solutions are derived [11]. Investigating optimality along these lines is out of the scope of this work. Instead, we set for a weaker optimality criterion, namely we wish to identify the policy r that maximizes the reward $R(s, r)$ for all states $s \in S$. This policy is, by definition, optimal with respect to the hit rate conditioned on the current state. In particular, it maximizes $P(H_n^r = 1|s)$ in Equation (4.6) for all s but does not necessarily maximize $P(H_n^r = 1)$.

An optimal algorithm: Suppose the cache is full. A new request arrives, and the cache must evict one out of the $K + 1$ documents. Order the documents in the cache according to their probability of being requested at the next request time, given the last h requests. Call by *Optimal*, the algorithm which evicts the document with the smallest probability of being requested at the next iteration. Optimal maximizes $P(H_n^r = 1|s)$ for all s by definition.

The probability of requesting document i at the next request time, given the last h requests, is given by Equation (4.2) for all i. We assume that Optimal either knows the

[7] In case of a miss, $X_n \neq Z_{n,i}^r$ for all i, $1 \leq i \leq K$, and there are $K + 1$ documents out of which one should be evicted. In case of a hit, the action is null.

[8] This is justified by the bounded convergence theorem [29].

	$\beta = 0.5$	$\beta = 0.75$	$\beta = 0.95$
Optimal	65.34%	47.98%	34.09%
LRU	59.01%	38.55%	22.20%
LFU	34.00%	32.23%	31.23%

Table 4.3: Hit rate comparison for Optimal, LRU, and LFU under various values of β.

p_i's, α_i's and β, or uses the ideas of Section 4.2 to estimate them. Thus, the algorithm can identify the document to evict at each eviction time. In the rest of the section we use Optimal as a benchmark for other algorithms.

How much better is Optimal than LRU and LFU? In particular, we are interested in comparing the hit rates of Optimal, LRU, and LFU under various request sequences with different temporal correlation characteristics. Thus, we use our model to generate such sequences.

We generate request sequences consisting of 5 million requests, drawn from a pool of 10000 documents whose popularity distribution is Zipf-like with parameter 0.5. We set h to 100, β to 0.5, 0.75 and 0.95, and assign to α_i's the rest of the probability according to a Zipf-like distribution with parameter 0.5. The cache size is set to 1000.

A relatively small value of β indicates a request sequence where short term correlations are more significant as compared to a request sequence with a larger β. Thus, smaller values of β correspond to larger hit rates.

Table 4.3 shows the hit rate of Optimal, LRU, and LFU. When β is relatively small, LRU's performance is quite good (the hit rate of LRU is at least $1 - \beta$) while LFU's performance is bad. This is expected since the primary characteristic of the request sequence is the strong short-term correlations. On the other hand, when β is relatively large, LFU's performance is improved, while LRU's performance is bad. Therefore, both LRU and LFU fail to adapt to the degree of short-term correlation versus long-term popularity.

4.5.2 The γ-LRU Class of Replacement Policies

Given a trace, which algorithm performs better depends on the degree of temporal correlation versus popularity. Optimal adapts to the degree of correlation, but requires the knowledge of α_i's and β which is hard to compute in practice. This motivates us to introduce γ-*LRU*, a class of practicable replacement policies parameterized by γ with LRU and

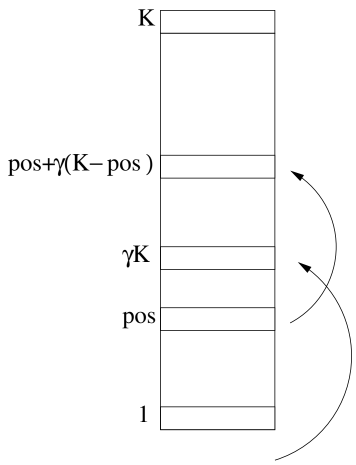

Figure 4.5: How documents move in a cache that uses γ-LRU.

LFU obtained for extreme values of γ, that also adapts to the degree of correlation in a trace.

γ-LRU algorithm: Consider the following variation of LRU: Whenever a page is requested, instead of moving the page to the top of the linked-list move it half way up. More generaly, whenever a page is requested move it up by a fraction γ, $0 < \gamma \leq 1$. This is how γ-LRU performs. In particular, name the positions in the cache $1, 2, \ldots, K$ moving from bottom to top, as illustrated in Figure 4.5. If the requested page is not in the cache, then insert it at position γK. Else, if it is currently at position *pos*, move it up to position $pos + \lceil\gamma(K - pos)\rceil$. [9]

How well does γ-LRU perform? Table 4.4 shows the hit rate of γ-LRU for the same parameters as in Table 4.3. The entries of Table 4.3 are repeated for comparison. It follows from this table that γ-LRU performs closely to Optimal for all values of β. In Table 4.4 we show the performance of γ-LRU for $\gamma = h/K$ and for the optimal γ, i.e., the γ value that maximizes hit rate, denoted by γ^*. We note that the performance of γ-LRU for $\gamma = h/K$ is close to the performance of $\gamma^* - LRU$. Later in this section we discuss this issue in more detail.

[9] In a different context, the authors in [4] introduce POS(k), another parameterized algorithm that performs Transpose [91] in the k topmost positions of the cache, and LRU in the rest of the positions. POS(k) has the same behavior with γ-LRU for the two extreme values of k and γ. However, it behaves very differently from γ-LRU for intermediate values of k and γ, and has worse performance. The problem with POS(k) is that it requires an impractically large period of time to gather popular documents at the top of the cache.

	$\beta = 0.5$	$\beta = 0.75$	$\beta = 0.95$
Optimal	65.34%	47.98%	34.09%
$\gamma^* - LRU$	62.81%	45.61%	32.25%
γ-LRU ($\gamma = \frac{h}{K}$)	61.77%	44.87%	31.86%
LRU	59.01%	38.55%	22.20%
LFU	34.00%	32.23%	31.23%

Table 4.4: Hit rate for γ-LRU and other schemes using the same parameters as in Table 4.3.

The closeness in the performance of γ-LRU and Optimal, for a large range of β values, can be intuitively explained by the documents that they cache. Optimal retains two classes of documents: (i) recently accessed documents, because their probability of being requested is high due to the α_i's, and (ii) very popular documents, because their probability of being requested is large due to βp_i. γ-LRU also retains these two classes of documents; the former in the lower part of the cache, and the later in the upper part of the cache.

Extreme Values of γ

Let K be the size of the cache. Then, $1/K \leq \gamma \leq 1$. For $\gamma = 1$ γ-LRU coincides with LRU by definition. We will show that for $\gamma = 1/K$, γ-LRU closely resembles LFU in stationarity.

When $1/K$-LRU is used, [10] documents that get requested while in the cache swap positions with the ones that are above them, and documents that get requested while being outside the cache swap position with the document at the bottom of the cache.

Let the total population of documents in the universe be N. Index the documents in decreasing order of their popularity. Denote by S_n the cache occupancy just prior to the n^{th} request. S_n is a Markov chain. Depending on the eviction scheme used, this chain may have up to $\binom{N}{K}K!$ states, corresponding to all possible orderings of all possible K-tuples of the N documents.

Call by *Offline-LFU* the algorithm which knows the overall popularity of the documents in advance and always keeps the most popular documents in the cache. For long request sequences, the empirical distribution that LFU uses is quite close to the actual distribution, and LFU and Offline-LFU have nearly the same behavior. When Offline-LFU is used, $S_n = (1, 2, \ldots, K)$ for all n. We will analyze $1/K$-LRU and show that this is the highest probability state of the corresponding Markov chain, and that most of the other states have

[10] $1/K$-LRU is sometimes referred to in the literature as Transpose [91].

significantly lower probability.

Let $1/K$-LRU be the eviction scheme used. In this case, S_n is time-reversible [11] [58] and it is easy to find the steady-state distribution in closed-form. Denote by $\pi_{(i_1,i_2,\ldots,i_K)}$ the steady state probability of state $i = (i_1, i_2, \ldots, i_K)$, i.e., document i_1 occupies position 1 in the cache, i_2 occupies position 2 in the cache and so on. Further, denote by P_{ij} the transition probability from state i to state j. One may verify that the condition

$$\pi_i P_{ij} = \pi_j P_{ji} \tag{4.7}$$

is satisfied by the stationary distribution:

$$\pi_{(i_1,i_2,\ldots,i_K)} = C \cdot p_{i_1}^K p_{i_2}^{K-1} \cdots p_{i_K}, \tag{4.8}$$

where C is a normalizing constant. For example, if $\pi_i = \pi_{(1,2,\ldots,K-1,K)}$ and $\pi_j = \pi_{(1,2,\ldots,K-1,K+1)}$, then $P_{ij} = p_{K+1}$ and $P_{ji} = p_K$, and Equation (4.7) holds.

Since the p_i's are decreasing as i increases, the highest probability state is $(1, 2, \ldots, K)$. Also, since p_i's are Zipf-like distributed, the probability of the states where unpopular documents reside in the cache is very low.

On the Optimal Value of γ

It is interesting to further investigate the behavior of γ-LRU as γ varies from $1/K$ to 1. Let γ^* denote the optimal value of γ with respect to hit rate given the history h, α_i's, β, p_i's, and the cache size K. We first argue that $\gamma^* \leq \frac{h}{K}$.

To see this, name the positions in the cache $1, 2, \ldots, K$ moving from bottom to top. Recall that $Z_{n,i}$ is a random variable indicating the document occupying position i at time n. Also, recall that in stationarity the hit rate of a replacement policy equals $P(H_n = 1) = \sum_{i=1}^K P(X_n = Z_{n,i})$, where H_n equals 1 if there is a hit at time n and zero otherwise, and X_n represents the document requested at time n.

If $\gamma K \geq h$ then all documents stay in the cache for at least h time slots, after their last request before they leave the cache. Since $\alpha_{h+i} = 0$ for all $i \geq 1$, these replacement policies take full advantage of short-term temporal correlation. Thus, $P(H_n = 1) = \sum_{j=1}^h a_j + \beta \sum_{i=1}^K P(Y_n = Z_{n,i})$ where Y_n, $n \geq 1$, is independent and identically distributed according

[11] When γ is larger than $1/K$ the corresponding Markov chain is not time-reversible.

to $\{p_i\}$.

The sum $\sum_{i=1}^{K} P(Y_n = Z_{n,i})$ is maximized by Offline LFU, since Offline LFU places in the cache the K most popular documents. Also, the more popular documents a replacement policy places in the cache, the larger this sum is. Now, the smaller the value of γ, the more popular are the documents in the cache. [12] Thus, this sum is larger for $\gamma K = h$ than for $\gamma K > h$, and the same is true for the hit rate.

Very small values of γ suffer from very large mixing times of the corresponding Markov chains. In practice, this means that it takes too much time for the cache to be populated with the most popular documents in the upper part of the cache. We choose to use γ values close to h/K since we find their performance to be close to optimal, see for example Table 4.4, and the corresponding mixing times to be small.

Figure 4.6 plots the hit rate of γ-LRU under the second synthetic trace used in Table 4.3 ($\beta = 0.75$), as a function of γ. For $\gamma = 1$ the hit rate equals that of LRU, and for small γ it is close to LFU's hit rate. Also, the maximum hit rate is achieved for a value of γ less than h/K while for $\gamma = h/K$ the performance is very close to optimal.

4.5.3 Performance Under Real Traces

In this section we study the performance of γ-LRU under real web traces from *NLANR* [70]. We also compare the hit rate obtained for real traces and their synthetic counterparts under various algorithms, to investigate whether the request model captures the essential properties of real traces as they relate to web caching.

We present results from 4 traces, recorded in May '01 and January '02. For the generation of the synthetic traces we use the procedure presented in Section 4.1. The history is set to 1000, and the parameters α_i's, β, and p_i's are inferred from the traces. The cache size is set to 5% of the total number of distinct requests in each trace, and the value of γ is set to 0.1 which is close to h/K in all the simulations. Finally, the cache is warmed up in all simulations. We don't take document size into account for now. Section 4.5.4 addresses this issue.

Table 4.5 reports the total requests and total distinct requests of each of the traces, as well as the cache size used in each case.

[12] We don't have a formal proof for this statement. Intuitively, larger values of γ cause larger perturbations whenever unpopular documents are requested. This intuition is verified by simulations, see for example Figure 4.6.

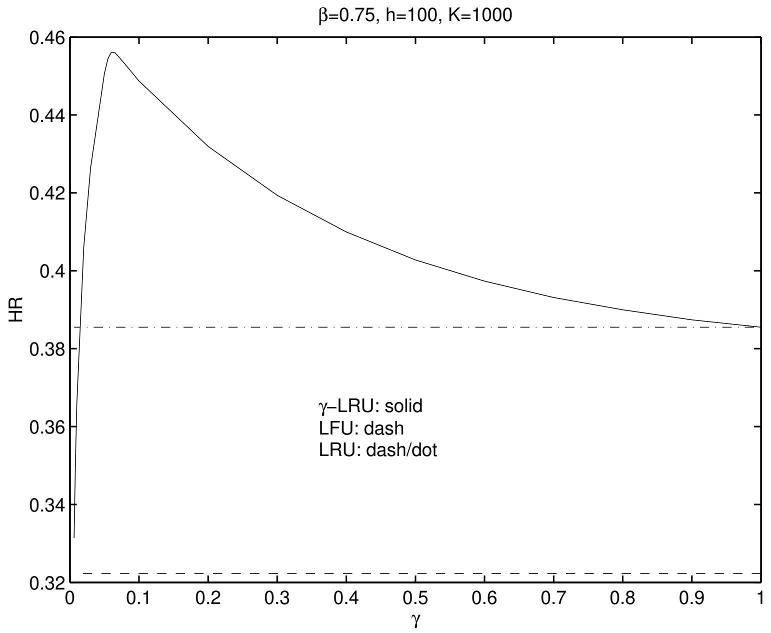

Figure 4.6: Hit rate of γ-LRU as a function of γ under the second synthetic trace (β=0.75) of Table 4.3. $\frac{h}{K} = 0.1$.

Trace	Total Requests	Distinct Requests	Cache Size
May 21^{st} '01	558826	275136	13500
May 23^{rd} '01	508980	261480	13000
January 19^{th} '02	512458	256486	12800
January 20^{th} '02	507827	238785	11900

Table 4.5: Trace characteristics and cache size.

	May 21^{st}, '01		May 23^{rd}, '01	
Scheme	HR (real)	HR (synthetic)	HR (real)	HR (synthetic)
Optimal	37.8%	41.1%	38.8%	40.2%
γ-LRU	35.3%	36.6%	35.1%	36.1%
LRU	32.2%	32.1%	31.7%	31.4%
LFU	29.4%	29.1%	28.5%	28.1%

Table 4.6: Hit rates under May '01 traces and their synthetic counterparts.

	January 19^{th}, '02		January 20^{th}, '02	
Scheme	HR (real)	HR (synthetic)	HR (real)	HR (synthetic)
Optimal	37.8%	41.2%	39.6%	42.5%
γ-LRU	34.9%	36.5%	36.6%	37.7%
LRU	30.7%	31.1%	32.7%	32.4%
LFU	28.9%	28.7%	31.1%	30.8%

Table 4.7: Hit rates under January '02 traces and their synthetic counterparts.

Table 4.6 presents the hit rates of Optimal, γ-LRU, LRU and LFU under the two May traces and their synthetic counterparts. Table 4.7 presents the same quantities for the January traces. As expected, Optimal is superior to all other algorithms. Also, γ-LRU is competitive against both LRU and LFU. Results are similar for all the other daily NLANR traces that we tried.

The hit rates obtained for the real traces and their synthetic counterparts are close. Optimal performs better under synthetic traces since it is designed to exploit the statistical characteristics that the synthetic traces exactly possess. The small differences in the hit rates are likely to be due to (i) the difficulty to estimate the popularity of the unpopular documents, (ii) the inter-document correlations present in the real traces that are not modeled, and (iii) the fact that the values of the α_i's are the same for all the documents (see Section 4.4 for a related discussion). These modeling decisions do not introducing significant inaccuracies, yet they simplify the model substantially.

4.5.4 Implementation Concerns

To be able to use γ-LRU in practice, we must address some implementation concerns. First, the size of the documents should be taken into consideration. It is straightforward to extend γ-LRU to take size into account, by making γ depend on the document size. Let $\gamma(S)$-LRU be such an extension where S denotes size and $\gamma(S)$ is a decreasing function of S.

We use a specific function $\gamma(S)$ and calculate the performance of $\gamma(S)$-LRU using real traces. The goal is to show that $\gamma(S)$-LRU is competitive against other high performance replacement schemes that take into account the size and the frequency of use in their eviction decision, in addition to recency of use.

We avoid doing an elaborate design of $\gamma(S)$, or optimizing its design. For S = 30KB, $\gamma(S)$ is set to 0.1. (We use 0.1 as the typical γ value. This value works well for all the conducted simulations, since the performance of γ-LRU under real traces is good for a wide range of γ values around h/K.) To penalize large documents, as S grows from 30KB to 250KB, γ drops quadratically to 0.05. Documents larger than 250KB are never cached. As S decreases from 30KB to 1KB, γ increases linearly to 0.2 and retains this value for smaller document sizes.

We compare $\gamma(S)$-LRU to Greedy-Dual-Size-Frequency-Connection (GD-SFC), a simple extension of the Greedy-Dual-Size (GD-Size) algorithm [17]. GD-SFC associates each document in the cache with a value, called eviction value, and evicts the document with the minimum such value. Whenever there is a request for a document i, the new eviction value for that document is set to

$$f_i = \min(f_j : j \text{ in cache}) + \frac{C_i \cdot F_i}{S_i},$$

where C_i is an estimate of the latency for connecting to the corresponding server, F_i is the number of times the document has been requested since it entered the cache, and S_i is the document size. Note that the quantity $\min(f_j : j \text{ in cache})$ is increasing in time and is used to take into account the recentness of a document. Indeed, whenever a document is accessed, its eviction value is increased by the currently minimum eviction value. Thus, the most recently used documents tend to have larger eviction values.

The traces we use are taken from (NLANR) [70]. We only present results from a daily trace of length 500000, since there are no significant differences in the results obtained from the various traces. We only simulate requests with a known reply size.

The performance criteria used are three:

(i) the hit rate (HR), which is the fraction of client-requested URLs returned by the proxy cache,

(ii) the byte hit rate (BHR), which is the fraction of client requested bytes returned by the proxy cache, and

(iii) the latency reduction (LR), which is the reduction of the waiting time of the user from the time the request is made till the time the document is received (download latency), over the sum of all download latencies.

For each trace, the HR, the BHR and the LR are calculated for a cache of infinite size. Then, they are calculated for a cache of size $0.5\%, 5\%, 10\%$, and 20% of the maximum size required to avoid any evictions. This size is around 2GB.

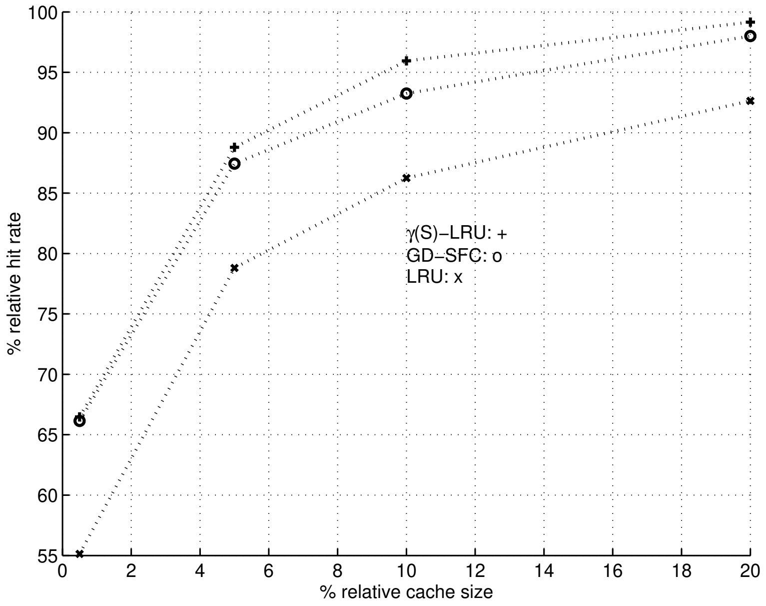

Figure 4.7: Hit rate comparison between $\gamma(S)$-LRU and GD-SFC. LRU's hit rate also shown.

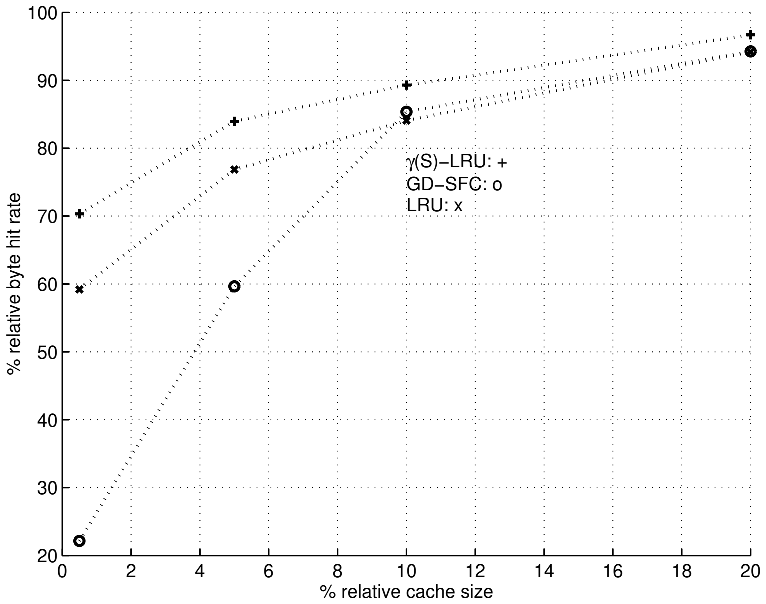

Figure 4.8: Byte hit rate comparison between $\gamma(S)$-LRU and GD-SFC. LRU's byte hit rate also shown.

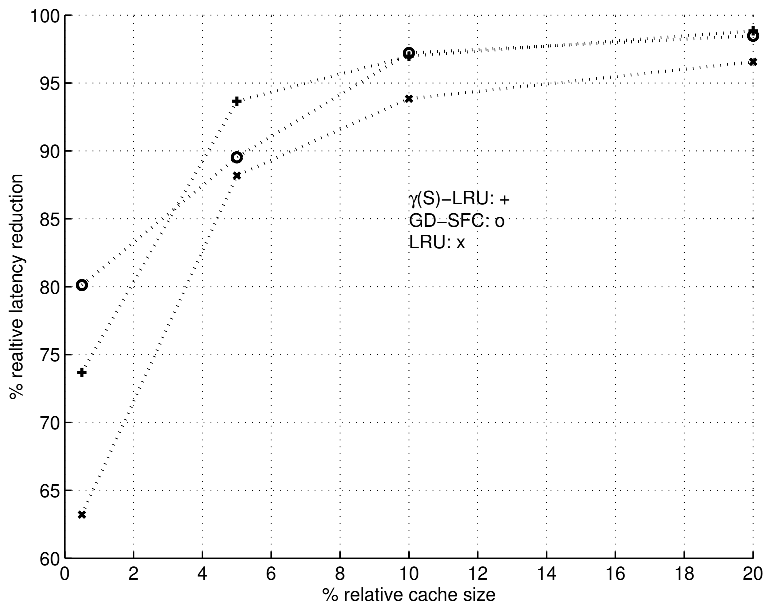

Figure 4.9: Latency reduction comparison between $\gamma(S)$-LRU and GD-SFC. LRU's latency reduction also shown.

Figure 4.7 presents the ratio of HR of $\gamma(S)$-LRU and GD-SFC over the HR achieved by an infinite size cache. The performance of the two schemes is nearly the same. Figure 4.8 presents the ratio of BHR of $\gamma(S)$-LRU and GD-SFC over the BHR achieved by an infinite size cache. The performance of $\gamma(S)$-LRU is superior to the performance of GD-SFC for small cache sizes. This is expected since GD-SFC has a strong bias against large size documents even when these documents are popular. Figure 4.9 presents the ratio of LR of $\gamma(S)$-LRU and GD-SFC over the LR achieved by an infinite cache. The performance of the two schemes is more or less the same.

Note that algorithms like GD-SFC usually estimate the frequency of use by recording the number of times documents are requested once in the cache, since it is not practicable to keep counters for all documents. Thus, they follow an approach similar to in-cache-LFU, which has worse performance than LFU. On the other hand, by design, $\gamma(S)$-LRU identifies popular documents as efficiently as LFU.

In addition to presenting the performance of $\gamma(S)$-LRU and GD-SFC, Figures 4.7, 4.8, and 4.9 also present the performance of LRU. The difference in the performance of the three schemes decreases rapidly as the relative cache size increases. Indeed, it has been observed that for very large caches, the performance of different replacement schemes is similar [17, 86]. The model introduced in Section 4.1 provides the following explanation for this phenomenon: (i) due to the Zipf-like distribution of the popularity of web documents, the fraction of documents that corresponds to most of the hits is small enough to fit in the cache, and (ii) due to the rapid decrease of temporal correlations (Figure 4.3), large caches store documents long enough to fully exploit these correlations.

Another implementation concern is the data structure required to implement γ-LRU. LRU uses a linked list to maintain the order of the documents in the cache. However, using a simple linked list for γ-LRU is not efficient. Indeed, linked lists are efficient when insertion and deletion operations take place at the top and the tail of the list only. In γ-LRU it is required to perform insertions at arbitrary places, which has complexity $O(\log n)$. To maintain constant-time complexity, we propose to only allow insertions to happen at predefined positions properly distributed along the length of the list. Each list entry will have a pointer pointing to the predefined position that the entry should jump to, if it is accessed. It is easy to see that whenever a jump takes place, only two of these pointers need to be updated.

The focus of this work is on the modeling of web trace correlations. The γ-LRU algorithm serves to illustrate the implications of these correlations on designing replacement policies. Since it is not our goal to fully investigate the performance of γ-LRU as a replacement scheme in practice, we refrain from further elaboration on $\gamma(S)$-LRU with predefined insertion positions, and from a comparison of its performance with other high-performance replacement schemes.

4.6 Conclusions

In this chapter we have proposed a model that precisely captures the degrees to which temporal correlations and document popularity influence web trace requests. We also described a mathematical procedure to extract the parameters of the model from real traces. The model was used to generate synthetic traces of arbitrary length exhibiting any degree of correlation, and provide guidelines for designing efficient replacement algorithms.

We formulated the problem of obtaining an optimal replacement algorithm with respect to hit rate as a Markov decision process, and obtained an online replacement algorithm that is optimal conditioning on the state of the Markov process. We also introduced γ-LRU, a class of practicable replacement policies parameterized by γ with LRU and LFU obtained for extreme values of γ. Finally, we showed that γ-LRU performs close to the optimal policy, under any degree of temporal correlation present in the request sequence, and extended the algorithm to take into account documents of varying sizes.

Chapter 5

Exploiting Redundancy in Dynamic Web Content

In the previous two chapters we focused our attention to static documents, or to documents that change over large time-scales. Since the fraction of web documents that belong to this category is decreasing day by day, traditional caching solutions have reached a point where their performance cannot be significantly improved. For example, no matter the replacement scheme, the cache size and the user population serviced by the cache, proxy cache hit rates are usually less than 40%. However, if proxy caches were equipped with mechanisms that exploit redundancy from all documents, static and dynamic, hit rates can increase up to 80% [109]. Another study reports that after proxy-caching has been applied, an additional 40% of web traffic is found to be redundant [102].

Even though documents that change frequently over time are not "cachable", the differences between consecutive versions of such documents are usually small. Motivated by this observation, researchers have recently proposed a variety of methods to exploit the redundancy in dynamic traffic.

Delta-encoding for web documents was introduced in [50], and independently in a more general context in [5]. The idea is that both ends of a slow link store the same snapshot of a web document, to be called *base-file* onwards, and upon a request for that document: (i) the end towards the server gets the current snapshot of the document from the web server, (ii) it computes the difference, to be called *delta* onwards, between the current and the stored snapshot, (iii) it sends the delta over the slow link to the other end towards the client, and (iv) the end towards the client reconstructs the current snapshot by combining the delta and the stored snapshot, and (v) it sends the response to the client. In [64] it is shown using traces that delta-encoding can provide remarkable improvements in the response size and delay for an important subset of HTTP content types.

The authors in [28] take an application-specific view, and separate the static and dynamic portions of a document. Static parts are cached as usual, while dynamic parts are obtained on each access from the server. They also provide an HTML extension to support this scheme. According to their simulations, the size of network transfers are typically 2 to 8 times smaller than the original sizes. This idea is simpler than delta-encoding, but it is less efficient. Clearly, delta-encoding exploits more redundancy than this scheme. Also, it is feasible to deploy delta-encoding without requiring HTML or other support from the existing infrastructure.

Another interesting work is Active Cache [18]. The Active Cache scheme allows servers to supply cache applets to be attached with documents, and requires proxies to invoke cache applets upon cache hits to furnish the necessary processing without contacting the server. Even though this approach is efficient for tasks like rotating advertisements, it is not efficient as a general mechanism that fully exploits temporal correlation in dynamic traffic. There are situations where one cannot avoid direct communication with the server that hosts the current snapshot of a document. Similar ideas with [18] are also used in [67] where caches dynamically generate content for clients by running code provided by publishers.

Finally, the authors in [100] propose a new protocol to allow individual content-generating applications exploit query semantics and specify how their results should be cached and delivered. Applications may declare a dynamic request to be identical, equivalent, or partially equivalent. In the first two cases cached results are up to date, while in the third case content can be immediately delivered as an approximate solution while actual content is generated and delivered. This is not a complete solution to the problem of exploiting temporal correlation in dynamic traffic; there exist dynamic documents that share many data, yet their redundancy cannot be exploited with this simple scheme.

In another direction, there has been a lot of research activity in the area of precomputing and handling dynamic traffic on the server-side [20, 49, 53, 114]. The idea is to offload servers from computing dynamic documents on the fly. These schemes reduce user latency only when the bottleneck is server's CPU, and do not reduce bandwidth consumption. However, they can be deployed in conjunction to delta-encoding mechanisms, since they are orthogonal to the previous schemes.

Delta-encoding could provide an efficient solution to the problem of exploiting temporal correlation in dynamic traffic. However, the basic delta-encoding scheme suffers from scalability problems at the server side. In particular, the storage requirements at the server side grow enormously due to the increasing number of dynamic documents. This problem becomes even more intense due to personalized web documents, since the scheme requires to store many personalized versions of each dynamic document.

Motivated by the storage concerns, we design a scalable scheme to exploit information redundancy in dynamic documents, called *class-based delta-encoding*. The idea is to group documents into classes, and store one document per class on the server side. In essence, in addition to exploiting temporal correlation in a dynamically evolving document, we also exploit spatial correlation among different documents.

The outline of the chapter is as follows. In the next section we describe the class-based delta-encoding scheme. Section 5.2 elaborates on how to group requests into classes. Section 5.3 proposes an online algorithm to choose a base-file for each class. Section 5.4 addresses the security issues raised by associating with every class one base-file that is shared by many clients. Finally, in Section 5.5 we present simulation results to evaluate the efficiency of class-based delta-encoding, and comment on implementation issues.

5.1 Class-Based Delta-Encoding

Delta-encoding is the process of generating a difference file, called *delta*, between two files such that the combination of the delta and one of the files, called *base-file*, suffices to reproduce the other file. An efficient delta-encoding mechanism has the following two properties: (i) the size of the delta is as small as possible, and (ii) the delta generation and the file reconstruction phases require as less time as possible to execute. For a thorough discussion on delta-encoding algorithms see [51].

In the context of HTTP, delta-encoding can be used to exploit *temporal* correlations between consecutive snapshots of a dynamic web document. As shown in Figure 5.1, the client and the server, for example an end user and a content provider respectively, share a common base-file which is a snapshot of the dynamic document at some point in time. Whenever the client requests the document from the server, the server computes the delta between the current snapshot of the document and the base-file, and sends the delta to the client. Upon receipt of the delta, the client computes the current snapshot of the document

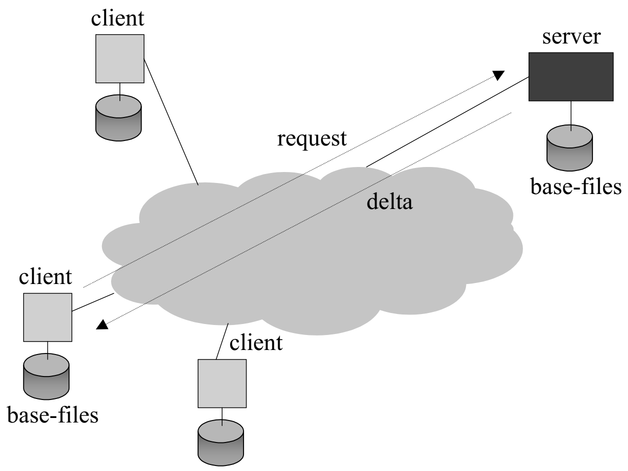

Figure 5.1: Using delta-encoding for web documents.

by combining the delta and the locally stored base-file.

In section 5.5 we show that this scheme is very efficient in reducing bandwidth consumption and user latency. We also show that the computational requirements to calculate the deltas on the server side are reasonable. However, this simple approach is not scalable due to the enormous storage requirements on the server side that are caused by the large number of dynamic documents. The problem becomes even more intense with personalized web documents. When documents are personalized (e.g. my.yahoo.com) the server must store one base-file for every dynamic document if hosts, for every client.

The solution to the scalability problem is to exploit *spatial* correlation, i.e. correlation between different documents, in addition to *temporal* correlation, i.e. correlation between different snapshots of the same document. This leads us to introduce *class-based delta-encoding*. Under class-based delta-encoding, dynamic documents are grouped into classes, and a single base-file per class is stored at the server.

There are three issues related to class-based delta-encoding that we address in this work:

- How to design an automated mechanism that groups documents into classes, based on the proximity of their content.

- How to design an efficient online algorithm that identifies a good base-file for each

URL	hint-part	rest
www.foo.com/laptops?id=100	laptops	id=100
www.foo.com/?dept=laptops&id=100	dept=laptops	id=100
www.foo.com/laptops/100	laptops	100

Table 5.1: URL parts for differently organized web sites.

class.

- How to design a mechanism that removes private information from the base-file of each class. This is important since the base-file of a class is stored locally by many clients.

5.2 Grouping Documents into Classes

In this section we describe a mechanism to group documents into classes. The mechanism aims to quickly identify an appropriate class for each document such that the difference between the base-file of the class and the document is small, i.e. the size of the delta, measured in bytes, is small.

Call *delta-server*, an engine that implements class-based delta-encoding and services the contents of some web servers. Since every document corresponds to a URL request, we group documents by grouping their corresponding URL requests.

All requests are processed by the delta-server before they are forwarded to the web servers. Initially, there are no classes formed in the delta-server. Whenever an ungrouped URL request arrives at the delta-server, the scheme groups it into an existing class, or creates a new class. As the number of classes grows, it is not practical to perform an exhaustive search among all existing classes to identify a good class. We shall use the observation that a similarity between two URLs is an indication of a similarity between their corresponding contents, and perform a search over a small and appropriate subset of the existing classes.

Partition the URLs in three parts, the server-part, the hint-part, and the rest. The server-part is the string from the beginning of the URL till the first slash, as usual. The portion of the URL that is used as the hint-part differs among web servers and depends on how the web server organizes its content. For example, let www.foo.com be a web site that sells computers, laptops and desktops. Assume that documents corresponding to laptops are similar, while they differ from documents corresponding to desktops. Table 5.1 shows the hint-part for three different cases. Depending on the web site, the administrator describes to the grouping mechanism how to partition URLs into parts using regular expressions. Then, the mechanism uses these parts to expedite the grouping process.

We now describe the grouping mechanism. A matching is said to occur if the delta between the requested document and the base-file of the class is smaller than a threshold. Since it is very unlikely that two documents originating from different servers will be close enough to join the same class, a new class is created in case there are no classes with members whose server-part is the same with the request's server-part. Else, some heuristics are used to tradeoff between search time and goodness of matching:

- If some classes have members whose hint-parts are the same with the request's hint-part, the mechanism only considers those as potential classes to group the request.
- The mechanism never considers more than N existing classes as potential classes to group a request. If no matching is found after N tries, a new class is created.
- The mechanism first attempts to group the request into classes with many members, and then into less popular classes. In particular, the first $a \cdot N$ tries consist of the most popular classes, and the last $(1 - a) \cdot N$ consist of random selections among the rest of the eligible classes. [1]
- Since for grouping purposes it is not required to generate a precise delta between the requested document and the base-file of a candidate class, but rather to estimate how close they are, a light version of the delta algorithm is used to reduce computation cost. [2]

We described an automated mechanism to group documents into classes. Sometimes, this mechanism is not efficient, for example, for web sites that are organized in a completely ad-hoc manner. In this case, the administrator has the option to manually group URLs into classes.

[1] Here it is implied that the procedure stops as soon as there is one matching. Another option is to always consider N classes and choose the best matching. The former is preferable since, in practice, it is more critical to reduce search time. Typical N values are less than 10, typical a values are around 0.5.

[2] A specific implementation of a delta-encoding algorithm is *Vdelta*[51]. *Vdelta* uses a hash table approach with enough indexes into the base-file for fast string matching. Each index is a position which is keyed by the four bytes starting at that position. Thus, the file is partitioned in four-byte chunks. Further, in order to identify the maximally long matching prefix, the algorithm traverses the file both forwards and backwards. We use a light version of this algorithm that uses larger byte-chunks and only traverses the file in the forward direction.

5.3 Choosing a Base-file

Once classes are formed, it is required to identify an appropriate base-file for each one of them. The simplest scheme would use as a base-file the document corresponding to the request creating the class. From a performance point of view, the best scheme would construct an artificial base-file consisting of all those document parts that are popular among the members of each class. However, this is too expensive. A reasonable compromise is to only consider entire documents as potential base-file candidates, and choose the best among those. Despite its relative simplicity, this choice performs well in practice.

Ideally, if all future requests $r_1, r_2, \ldots, r_n$ where known in advance, an offline algorithm would choose as a base-file the document that minimizes the sum of deltas between itself and every other document. The exhaustive online algorithm that considers all documents seen so far as base-file candidates is not practicable due to memory and computational constraints. Instead, we propose the following scheme:

1. Sample each request with probability p i.e. consider the corresponding document a base-file candidate and store it.

2. Use as a base-file the best of the stored documents i.e. the document that minimizes the sum of deltas between itself and all the other stored documents.

3. Store up to K documents. Thus, after acquiring K samples, whenever a new sample is drawn evict the document that maximizes the sum of deltas. [3]

Call a *rebase* the process of changing a base-file. After a rebase, the new base-file should be distributed to all clients before they can benefit from delta-encoding. To control the number of rebases, a rebase takes place if both a better base-file candidate exists, and a rebase-timeout, since the previous rebase, has expired. Rebases caused by this algorithm are called group-rebases to distinguish them from basic-rebases that are triggered when the generated deltas are relatively large. When a basic-rebase takes place, all K stored documents are flushed. Group-rebase and basic-rebase are orthogonal operations.

[3]To avoid storing K documents that are very close to each other but not close to most of the members of the class, at periodic intervals we evict a random sample among the stored documents excluding the current base-file, instead of evicting the document that maximizes the sum of deltas (worst document). Another option is to maintain two sets of K documents: one holding the base-file candidates and one holding K random samples against which deltas are calculated for each base-file candidate. At eviction times, the worst document is evicted from the candidates' set, and a random sample is evicted from the other set. Both options work well in practice.

By design, the algorithm stores good base-file candidates and uses the best out of those as a base-file. Also, the computational and space requirements of the algorithm are low: Whenever a new sample is acquired, the algorithm calculates only the deltas between this document and the rest of the stored documents, and this calculation can be done *offline*. The memory requirements are low since a value of K less than 10 leads to good performance.

In Section 5.5 we evaluate the performance of the algorithm in practice. Here, we perform some simple analytic calculations. Assume there is a sequence of requests for distinct documents that are members of a class. We shall calculate the probability that the proposed scheme will discard the best base-file candidate. Let R be the total number of requests. Then, the expected number of base-file candidates is $N = R \cdot p$. Let S_N denote the set of all base-file candidates, and S_K denote the set of all K stored documents at some point in time. Finally, let d_{ij} denote the delta between documents i and j.

Associate with every base-file candidate i a global utility value, equal to $\sum_{j \in S_N} d_{ij}$. Index the documents in increasing order of their global utility value, i.e. document 1 minimizes $\sum d_{ij}$. Associate with every stored document i a local utility value, equal to $\sum_{j \in S_K} d_{ij}$. Notice that the algorithm uses the local utility values to order the documents.

Let i_1, i_2 be two documents such that $i_1 < i_2$, i.e. $\sum_{j \in S_N} d_{i_1 j} < \sum_{j \in S_N} d_{i_2 j}$. Suppose that the probability of the event $\{\sum_{j \in S_K} d_{i_1 j} > \sum_{j \in S_K} d_{i_2 j}\}$ equals $c/|i_1 - i_2|$ where c is a normalizing constant such that $c \cdot \sum_{i=1}^{N-1} \frac{1}{i} = 1$. This event corresponds to the situation where document i_1 is a better base-file candidate than document i_2, but since the algorithm uses the local utility values it would output the opposite. [4] The best candidate, document 1, can only be evicted if $\sum_{j \in S_K} d_{1j} > \sum_{j \in S_K} d_{ij}$ for all other stored documents $i \in S_K$. In this case the algorithm will make an error.

The probability of error at an eviction time is at most $c^{K-1}/(K-1)!$. This probability corresponds to the event where the K best base-file candidates are stored and $\sum_{j \in S_K} d_{1j} > \sum_{j \in S_K} d_{ij}$ for $i = 2 \ldots K$. Thus, the probability of keeping the best base-file candidate throughout the process is at least $\left(1 - \frac{c^{K-1}}{(K-1)!}\right)^{N-K} \approx 1 - (N-K)\frac{c^{K-1}}{(K-1)!}$, and since $c \approx 1/\ln(N-1) \approx 1/\ln N$, the probability of error is bounded above by

$$P_{error} \leq \frac{(N-K)}{(\ln N)^{K-1}(K-1)!},$$

[4] The probability expression $c/|i_1 - i_2|$ is used as an example. Its rationale is that the closer the global utility values of two documents are, the more likely it is for the ordering based on their local utility values to be wrong.

which is very small for all practical cases. For example, for $R = 10^5$, $p = 10^{-2}$, and $K = 10$, $N = 1000$ and $P_{error} \leq 8 \cdot 10^{-11}$.

5.4 Privacy Concerns

Class-based delta-encoding raises privacy concerns since many clients share the same base-files. In particular, each base-file is stored locally at more than one client. Therefore, it is critical to remove private information, for example credit card numbers, from shared base-files.

Before we present a mechanism that removes private information from base-files, it is useful to elaborate on a specific implementation of delta-encoding called *Vdelta* [51]. *Vdelta* uses a hash table approach with enough indices into the base-file for fast string matching. Each index is a position which is keyed by the four bytes starting at that position. The algorithm compares four-byte chunks from the base-file and the requested document and finds maximally long matchings between the two documents. Some byte-chunks are common, while others are not present in both documents.

A simple but efficient mechanism to identify and remove private information from base-files is the following: (i) execute delta-encoding between the base-file and N other documents that are members of the class, (ii) record for each four-byte chunk of the base-file how many times it was common between the base-file and another document, and (iii) remove from the base-file these byte-chunks which were not common between the base-file and any of the N other documents. We call this the *anonymization* process. The rationale behind this scheme is that private information is unique among users. Thus, by making sure that during the anonymization process the base-file is compared against documents corresponding to different users, any private information will be removed. Also, by doing the comparison with more than one document, the process does not remove potentially useful information from the base-file.

Note that until the base-file is properly anonymized, it should not be distributed to clients. Thus, delta-encoding should not be used until the anonymization process terminates. However, if there is already an anonymized base-file and a rebase is triggered, the previous base-file can be used until the new one is properly anonymized. Therefore, the performance penalty is not significant. Finally, the anonymization process can take place concurrently with the mechanism for finding a base-file, since both require the computation of deltas between some documents that are members of the same class.

A point of concern is that the same user may be falsely regarded by the server as a different user. Indeed, the standard way to distinguish users is by distributing to them user identifications through cookies. However, for example, Netscape and Internet Explorer do not share cookies and thus user identifications. As a result, if a user opens one Netscape and one Internet Explorer session at the same time and uses his/her credit card to make two transactions, the system will interpret these transactions as originating from different users.

Another point of concern is that sometimes private information is shared among a small set of different users, for example corporate credit card numbers. If two employees use a shared corporate credit card to make a transaction at the same web site, while base-file anonymization is in progress for the class in which the corresponding requests belong to, the credit card number is at risk.

Even though the odds of these situations occurring are small, we wish to protect private data from such cases. To solve this problem, we introduce a second parameter M, and only include in the anonymized based-file byte-chunks that are common between the base-file and at least M documents. $M = 0$ corresponds to no privacy, $M = 1$ corresponds to the basic anonymization scheme, and larger values of $M \leq N$ increase privacy. Values of M close to N significantly reduce the size of the base-file and thus compromise performance since deltas tend to be large. A rule of thumb is that N should be at least twice as large as M.

The steps of the algorithm are as follows:

1. Choose a base-file.

2. Associate with each byte-chunk of the base-file one counter to record the number of times that each byte-chunk is common between the base-file and another document.

3. For the next N requests that belong to the specific class and originate from distinct users, [5] increment the counters of the byte-chunks that are common between the base-file and the requested document.

4. Remove all byte-chunks with counter values less than M.

[5] These users should also be different from the user that corresponds to the base-file.

We perform some elementary calculations to estimate how likely it is to accidentally include in an anonymized based-file private information. The exercise consists of starting with a non-anonymized base-file, drawing N documents out of which X may share private information with the base-file, and calculating the probability of the event $\{X \geq M\}$. Let X_i, $i = 1 \ldots N$, be a random variable equal to 1 if the i^{th} sample shares private information with the base-file, and zero otherwise. Then, $X = \sum_{i=1}^{N} X_i$. Assume the X_i's are i.i.d random variables equal to one with probability p and zero otherwise. Then, X is binomially distributed with parameters N and p, and the probability of error, $P(X \geq M)$, equals $\sum_{i=M}^{N} \frac{N!}{i!(N-i)!} p^i (1-p)^{N-i}$. Assuming $P(X \geq M) \approx P(X = M)$ [6] and $(1-p)^{N-M} \approx 1$, the probability of error is bounded above by

$$P_{error} \leq (Ne/M)^M p^M.$$

For $p = 0.01$, $N = 10$ and $M = 5$, the upper bound equals $4.7 \cdot 10^{-7}$ while the exact probability of error equals $2.4 \cdot 10^{-8}$.

In the preceding analysis we assumed that the X_i's are i.i.d. Here, we assume that $P(X_2 = 1|X_1 = 1) < P(X_2 = 1|X_1 = 0)$. In general, if p_j is the probability of the j^{th} occurrence of a document sharing private information with the base-file, we assume $p_j = p\ j$, i.e. a decreasing function of j. Recall that for an error to occur, there should be at least M out of N documents sharing private data with the base-file. Since the probability of getting the $(M+1)^{th}$ such document is very low, we can ignore the cases where $X > M$ and only compute the probability of the event $\{X = M\}$. Since $P(X = M) \leq \frac{N!}{M!(N-M)!} \cdot p \cdot p^2 \cdots p^M \cdot (1-p^M)^{N-M}$, and $(1-p^M)^{N-M} \approx 1$, the probability of error is bounded above by

$$P_{error} \leq (Ne/M)^M p^{M(M+1)/2},$$

which is exponentially smaller than the previous bound.

5.5 Simulation Results and Deployment Issues

In this section we present simulation results to evaluate the performance of class-based delta-encoding. First, we report bandwidth savings achieved by delta-encoding using real traces, and convert bandwidth into latency savings. Second, we evaluate the performance

[6] p^i decreases a lot faster than $\frac{N!}{i!(N-i)!}$ increases as a function of $i < N/2$, for typical values of p and N.

Web sites	Total requests	Direct KB	Delta KB	Savings
1	16407	736495	38308	94.8%
2	1476	49536	2474	95.0%
3	7460	230840	6640	97.1%

Table 5.2: Bandwidth savings using access-logs from three commercial web sites.

of the class-related operations, namely grouping, choosing base-files, and anonymizing base-files. Finally, we propose an architecture to deploy class-based delta-encoding, and report experimental results from a specific implementation of that architecture.

5.5.1 Delta-encoding Performance

Delta-encoding is very efficient in reducing bandwidth consumption. Using traces from commercial web sites, we calculate the total outbound traffic when delta-encoding and compression of the generated deltas is used, [7] and compare it to the total direct outbound traffic. Notice that the access-logs of web sites represent HTTP requests after any proxy caches, and thus correspond to traditionally uncachable traffic. Table 5.2 shows the results obtained by three commercial web sites. [8] From this table we infer that delta-encoding combined with compression, reduces bandwidth consumption by at least a factor of 20, and sometimes by a factor of 30 or more. In these numbers a factor of 2 on average is due to compression. These result are in accordance with previous studies [64, 75]. Notice that the experimental results reported here correspond to actual savings when using an implementation of the architecture presented in Section 5.5.3. In contrast, the previous studies assume the existence of a delta-encoding mechanism and report projected savings by applying delta-encoding on the documents of a web server.

Since delay is what matters most to end users, it is interesting to investigate the relationship between bandwidth and latency savings. Web documents that benefit significantly from delta-encoding have an average size around 30 to 50KB, as observed from access-logs of real web servers. Gzipped deltas tend to be 1 to 3KB. Without loss of generality, suppose that two documents of size $S_1 = 30KB$ and $S_2 = 1KB$ are sent from a server to a client using TCP. Let L_1 and L_2 be the user latency corresponding to document 1 and 2 respectively. Given that $S_1/S_2 = 30$, we wish to estimate L_1/L_2.

[7]Deltas are compressed using gzip.

[8]Due to privacy concerns, we are unable to provide the URLs of the three sites.

Assume the transmission takes place over a high-bandwidth connection. Due to TCP slow-start the number of round-trip times (RTTs) required to send document 1 is roughly $\log S_1/S_2$ times the number of RTTs to send document 2. This can be easily verified by taking into account the additive increase of the TCP window when counting RTTs in both cases. Thus, L_1/L_2 is roughly equal to 5. The latency gain from delta-encoding is further increased when the client is connected through a low-bandwidth connection. Without loss of generality assume the connection is through a 56Kb/s modem with 100ms RTT. The transmission time of a single packet is roughly equal to twice the RTT, thus the advantage of larger TCP windows is reduced since transmission time is now the bottleneck. In this case L_1/L_2 is a linear function of S_1/S_2. Due to the cost associated with setting up a TCP connection, the queueing delay, the timeouts and the retransmissions caused by packet losses, L_1/L_2 is estimated to be around 10. More information on how to derive such estimations can be found in [7]. Using measurement tools [68], we measured L_1 and L_2 and verified the estimated values for L_1/L_2 in case of high- and low-bandwidth connections.

Delta-encoding is a CPU-intensive process since it involves compressing one file using the contents of another file. However, there are quite efficient algorithms, like Vdelta [51], to perform this task. In Section 5.5.3 we report experimental results which show that the CPU overhead associated with delta-encoding is reasonable.

5.5.2 Performance of Class-Based Operations

We have implemented the grouping, choosing a base file, and anonymization functionalities, and integrated them with the delta-encoding routines to evaluate their performance.

The grouping mechanism aims to find a good class for all documents as quickly as possible. The automated mechanism introduced in Section 5.2 groups requests in classes after a couple of tries, provided that a similarity between two URLs implies a similarity between the corresponding web documents, and proper regular expressions are given to the scheme to partition the URLs into parts. Further, under the traces used in Table 5.2, the number of produced groups are between 10 and 100 times less than the number of the original dynamic documents. Finally, no noticeable increase in the size of the deltas is observed when documents are grouped into classes.

To evaluate the performance of the proposed algorithm for choosing base-files, we use various algorithms to choose a base-file, and calculate the average size of the delta obtained

Permutations	First Response	Randomized	Online Optimal
1	1704	1559	1406
2	1774	1636	1540
3	1785	1599	1515
4	1876	1626	1542
5	2025	1679	1575

Table 5.3: Average delta sizes, in bytes, resulting from various algorithms that identify base-files for classes.

M	N	Base (plain)	Base (anon)	Delta (plain)	Delta (anon)
2	5	84213	73434	5224	6520
4	12	84213	72714	5224	6097
4	8	84213	71090	5224	6505

Table 5.4: Base-file and delta sizes, in bytes, for various anonymization (anon) levels. Deltas represent average values among a large pool of documents.

under the same sequence of requests for each of the algorithms. In particular, we compare (i) the scheme that uses the first response as a base-file, (ii) the randomized online algorithm proposed in Section 5.3, and (iii) the online optimal algorithm that uses as a base-file the document that minimizes the average delta so far. Table 5.3 presents the results for five different random permutations of the sequence of requests. The randomized algorithm uses a total of 8 samples, and a probability of choosing a sample as a base-file candidate equal to 0.2. It performs close to the optimal. The performance of the scheme that uses the first response as a base-file is worse than the other two, and it can be very bad, which is never the case for the randomized algorithm, since the latter adapts to the characteristics of the request sequence.

Anonymization is required to provide privacy. The downside is that deltas tend to be larger. Table 5.4 presents the size of base-files before and after anonymization for various anonymization levels, and the average sizes of the corresponding deltas with and without anonymization. Even when high anonymization levels are used, deltas tend to increase by a very small amount. Thus, anonymization is achieved at a minimal cost.

Even though anonymization increases delta sizes, combined with class-based delta-encoding may improve bandwidth savings in comparison to classless delta-encoding. Indeed, anonymized based files are cachable, and thus many different users will download the same base-files from a proxy cache. Since base-files are a lot larger than deltas, the gain from cachable base-files is expected to be larger than the loss from slightly larger deltas.

5.5.3 Architecture and Deployment

We have established that class-based delta-encoding is a scalable scheme that significantly reduces bandwidth consumption and user latency. However, in order to be widely deployed, it should require no changes to the current web architecture. Web servers, proxy caches, and clients' browsers should be able to interoperate with the class-based delta-encoding functionality without any changes. An architecture that satisfies this requirement is introduced at [34]. We will briefly present the main components of this architecture.

On the server side, there should be a delta-server responsible to generate the deltas and store all base-files. The delta-server should be as close to the web server as possible to minimize delays due to the transmission of current document snapshots from the web server to the delta-server. Thus, the delta-server is placed next to the web server. When class-based delta-encoding is used, the scheme groups similar documents and personalized versions of the same document in one class and uses a single base-file per class. Hence, the storage requirements on the delta-server are reasonable.

On the client-side, one may use the browser's cache to store base-files, and rely on Java-scripts, enabled at the browser, to combine deltas and locally stored base-files in order to compute the current document snapshot. Another option is to use specialized plug-ins. Notice that base-file anonymization guarantees that shared base-files do not contain private information and cannot be associated with any particular user.

Base-files can be marked cachable. Thus, proxy caches can cache them as usual, resulting in the known benefits of proxy-caching.

Therefore, class-based delta-encoding can be deployed transparently to clients, proxy caches, and web servers, with the only necessary addition being a delta-server on the server-side, as shown in Figure 5.2.

A specific realization of the architecture shown in Figure 5.2 has been implemented and tested. We use this implementation to comment on the CPU overhead of such a mechanism. The numbers reported correspond to a PC configured with a Pentium III at 866MHz and 512MB of RAM, where the delta-server has been integrated to an Apache 1.3.17 [2] web server, running under Linux 2.2.19.

A plain Apache web server is measured to have a capacity of 175 to 180 requests per second, with a maximum number of concurrent connections equal to 255. We will compare

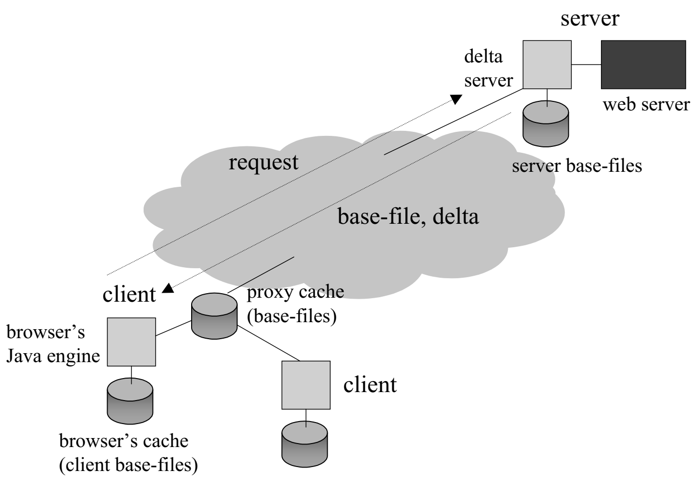

Figure 5.2: Transparent deployment of class-based delta-encoding.

these numbers to the respective numbers achieved by the system of the delta- and web server.

For a base-file of around 50 to 60KB, and a generated delta of around 8KB (uncompressed) or 3KB (compressed), the delta generation process requires 6 to 8ms. This latency is insignificant for a client. [9] However, the delta generation is a CPU-intensive process. As a result, the capacity of the system of the delta-server together with the web server is around 130 requests per second. Even though the system of the delta- and web server has lower capacity than a stand-alone web server, it can sustain a higher number of concurrent connections. In particular, thanks to an offloading effect on the web server from the delta-server, the system is possible to sustain 500 or more concurrent connections.

From the above we conclude that the CPU overhead of the delta-encoding mechanism is reasonable. In addition, CPU is cheap in comparison to the cost of access links. Thus, in practice it is very common that the bottleneck resource at a web server is the access link out of the web site and not the CPU. This further reduces the significance of the CPU overhead.

[9] Insignificant is also the latency associated with the document reconstruction on the client side.

5.6 Conclusions

In this chapter we introduced class-based delta-encoding [82], a scheme for exploiting the redundancy in web documents that change frequently over time. By combining the basic delta-encoding technique with the idea to group documents into classes, our scheme benefits from both temporal and spatial correlations found in dynamic traffic.

Class-based delta-encoding significantly reduces bandwidth consumption and latency perceived by users. Further, its computational and storage requirements are small, and it can be deployed without changing the current web infrastructure.

Part II

Performance Prediction for IP Networks and Web Farms

Chapter 6

Techniques for Performance Prediction

Monitoring the performance of a network or a web farm in real time, and predicting their behavior under novel protocols, architectures and load conditions, are important research problems. Network and farm operators find it useful to look at a live representation of their entire system: this lets them identify potential problems at a glance, ensure reliable operation, and monitor their service level agreements. Researchers like to evaluate the effect of design changes on system performance.

Despite their importance, there are currently no good solutions to these problems. This is not because researchers and industry have ignored them. On the contrary, there has been a lot of related research [6, 26, 30, 94] and industrial products [23, 81]. The problems of measurement and analysis of systems like the Internet are hard because of the systems' overwhelming size, complexity, heterogeneity, and speed of operation.

In this chapter we will review the existing techniques for measurement and analysis, and explain their limitations. In the following chapters, we will present our approach and apply it to IP networks and web farms.

6.1 Existing Techniques for Measurement and Analysis

Broadly speaking, there are two ways to predict the performance of a large, complex system. The first is to devise models and analytical tools to work with summary statistics. The second is to collect traffic measurements, and then feed the measured traffic into a simulator.

For example, one may use the network simulator [72] with a real trace as the input traffic.

Unfortunately, neither of the two methods is appealing when applied to the Internet. First, working with summary statistics rather than full traces is troublesome because the heterogeneity of the Internet means that each scenario might potentially require a new traffic model. For example, a model for backbone traffic would differ significantly from a model describing edge traffic. As a result, even though researchers have succeeded in modeling special types of traffic [38, 63, 105], there is no general model for the actual Internet traffic. Heterogeneity also makes it difficult and time consuming to devise models for the network or a web farm. For example, large commercial web farms have different architectures and one might need a different model for each architecture.

Second, it is very hard to take measurements because of the very high speeds of operation and the large amount of data: Millions of packets per second traverse backbone links, and recording packet headers from such a link results in terabytes of data per day [41]. Even when such data are available, the very detail and volume make them nearly impossible to analyze in real time. Further, it is very expensive to run a realistic simulator offline, due to the huge memory and CPU requirements.

The sheer size of the traffic has prompted work on traffic sampling [24, 30]. Sampling certainly reduces the volume of data, though it can be hard to work backwards; that is, to infer the performance of the original system. Another step to reduce the volume of measured traffic is to monitor flows [52]. However, the number of flows in a network is still quite large. For instance, it has been recently reported that there are millions of distinct flows traversing a backbone link during a period of one hour [31, 41].

6.2 A New Direction

To sidestep the above problems, we explore a new method which combines sampling, modeling and simulation, and which we call SHRiNK. [1] The basic hypothesis is this: If we take a *sample* of the traffic, and feed it into a *suitably scaled* version of the system, we can *extrapolate* from the performance of the scaled system to that of the original.

This has two benefits. First, by relying only on a sample of the traffic, SHRiNK reduces the amount of data we need to work with. Second, by using samples of actual traffic, it short-cuts the traffic characterization and model-building process while ensuring the relevance of

[1] SHRiNK: Small-scale Hi-fidelity Reproduction of Network Kinetics

the results.

This approach also presents challenges. At first glance, it appears optimistic: might not the behavior of a large network with many users and higher link speeds be intrinsically different to that of a smaller network? Somewhat surprisingly we find that, in several essential ways, one can mimic a large network or web farm using a suitably scaled-down version. The key is to find suitable ways to sample the traffic, scale down the system, and extrapolate performance.

In Chapter 7 we present the proposed method in detail. In Chapter 8 we apply SHRiNK to IP networks, and in Chapter 9 we apply the method to web farms.

Chapter 7

SHRiNK: A Method for Performance Prediction

In this chapter we present SHRiNK, a scalable data-driven method for performance prediction of large systems. The main idea of the method is to reduce overall complexity by using a sample of the actual traffic to drive a small-scale replica of the original system. Then, the method exploits a natural scaling law to extrapolate from the performance of the scaled system to that of the original. The challenge is to find suitable ways to scale down the original system, so that performance extrapolation is possible.

7.1 The Method

The proposed method consists of three main steps:

(i) Take a *sample* of the traffic. For example, in the case of IP networks one may sample in an i.i.d fashion network flows. In the case of web farms, one may sample HTTP requests. [1]

(ii) Feed the sampled traffic into a *suitably scaled* version of the original network or web farm. For example, a scaled web farm would consist of a fraction of the web servers of the original farm, and a scaled network would have links with lower capacities than the original network.

[1] As discussed in Chapter 9, one does not sample embedded requests, but rather requests for HTTP documents.

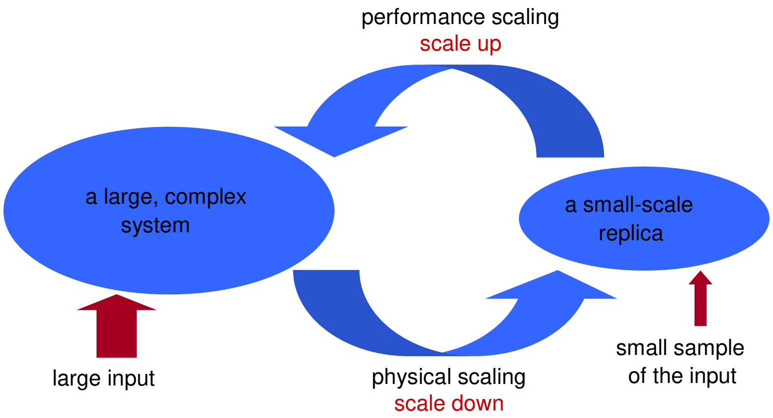

Figure 7.1: Physical and performance scaling.

(iii) *Extrapolate* from the performance of the scaled system to that of the original.

There are two types of scaling associated with SHRiNK: physical and performance scaling as shown in Figure 7.1. A system is physically scaled down to produce a small-scale replica. Then, we observe the performance of the replica and do performance scaling by scaling up the results. Our hypothesis is that there are appropriate ways to scale the system, such that performance extrapolation is possible.

SHRiNK not only simplifies the performance prediction of existing systems, but also predicts their behavior if they were to further increase in size and speed.

7.2 Scaling Hypothesis on Traditional Models

We start with a simple example which illustrates the key points: the $M/M/1$ queue. Suppose jobs arrive at a queue according to a Poisson process of rate λ, and that service requirements are independent and exponential with rate $\mu > \lambda$. Let $Q(t)$ be the number of jobs in the system at time t. Then, in equilibrium, the probability of having n jobs in the system equals $P(Q = n) = (1 - \lambda/\mu)(\lambda/\mu)^n$.

Now scale the system as follows: Sample the arriving jobs, keeping each job with probability α, independent of the others, so that sampled arrivals form a Poisson process of rate $\alpha\lambda$. Consider feeding the sampled arrivals to a separate queue whose server runs slower than the first by a factor α as illustrated in Figure 7.2(ii). Slowing down the server is equivalent

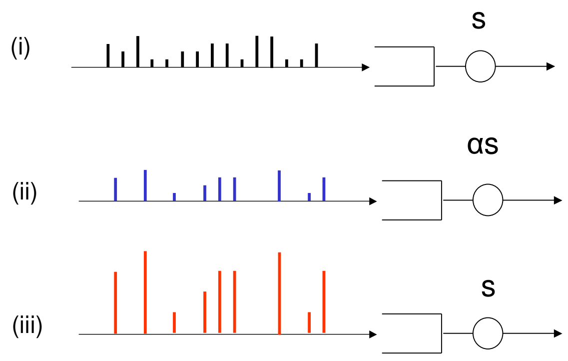

Figure 7.2: Scaling a system down: (i) original system, (ii) scaling down service rate, (iii) inflating jobs' sizes.

to multiplying the service requirements by a factor $1/\alpha$ as illustrated in Figure 7.2(iii), so that service requirements are rate $\alpha\mu$ exponentials. The slower queue is also $M/M/1$.

Let $\tilde{Q}(t)$ be the number of jobs in the slower queue at time t. The ratio of the arrival rate over the service rate is the same in the original and slower queue, and hence in equilibrium, the marginal distributions of the two queues are equal:

$$P(Q = n) = (1 - \lambda/\mu)(\lambda/\mu)^n = (1 - \alpha\lambda/\alpha\mu)(\alpha\lambda/\alpha\mu)^n = P(\tilde{Q} = n).$$

Thus, we have inferred the distribution of queue-size, and hence of delay, in the original high-speed system by looking at a smaller-scale version.

Further, it is not hard to see that $Q(t) \stackrel{d}{=} \tilde{Q}(t/\alpha)$, i.e. equal *in distribution*. That is, the evolution of the slower queue is statistically equivalent to that of the original queue slowed down in time by a factor α. This is because the queue-size process in an $M/M/1$ queue is a birth-death chain [108], and the ratio of the birth and death rate are the same in the two systems.

This result can be easily extended to a network of queues. In a network of queues with Poisson arrivals at the network entry points, independent exponential service requirements and independent routing of packets, Jackson's theorem [10, 108] asserts that the queue lengths in the system are distributed as if each queue were an $M/M/1$ queue, independent of the other queues. Hence, the probability of having n_1 packets in the first queue, n_2 packets in the second, and so on equals

$$P(Q_1 = n_1, Q_2 = n_2, \ldots) = \prod_{j=1,2,\ldots} (1 - \lambda_j/\mu_j)(\lambda_j/\mu_j)^{n_j},$$

where λ_j and μ_j are the arrival and service rate of the j^{th} queue.

Consider now the scaled version of the original network of queues where the arrivals *at the network entry points* are sampled in an i.i.d fashion with probability α, and the speed of all servers is reduced by the same factor. Sampling arrivals at the network entry points makes the aggregate rate of arrivals at queue j to equal $\alpha\lambda_j$ for all j. [2] Hence, the ratio of the arrival and departure rates remains unchanged for all queues, and the same holds for the joint and marginal distributions of the queues.

The class of queueing systems for which there exist closed-form solutions is rather small and depends on the assumption of exponential service requirements, which is unrealistic for data networks. We will now prove that a queue with Poisson arrivals and general independent service requirements observes the scaling hypothesis when one samples the arrivals in an i.i.d. fashion and scales the service rate in proportion. This result is independent of the queue discipline.

The easy argument that worked for $M/M/1$ queues is not applicable now: M/GI queues don't give rise to a simple birth-death chain. We will give a more subtle argument, involving two steps.

Theorem 7.1 *Let $Q(t)$ be the number of jobs in an M/GI queue with arrival rate λ, and average service requirement $1/\mu$. Let $\tilde{Q}(t)$ be the number of jobs in the scaled queue. Then $Q(t) \stackrel{d}{=} \tilde{Q}(t/\alpha)$.*

Proof: First, consider stretching time by a factor $1/\alpha$: if two consecutive events, arrivals or departures, of the original queue are t units of time apart, they are now t/α units of time apart. Let $\hat{Q}(t)$ be the number of jobs in this "time-stretched" queue at time t. It is clear that $Q(t) = \hat{Q}(t/\alpha)$ *exactly* for every t (i.e. on every sample path, not just in distribution).

[2] If r_j is the external arrival rate in queue j, and p_{ij} is the probability that a job serviced from server i will join queue j, then $\lambda_j = r_j + \sum_i \lambda_i p_{ij}$. Thus, if $\tilde{r}_j = \alpha r_j$, $\tilde{\lambda}_j = \alpha\lambda_j$.

Notice that the effect of stretching time on the input process is equivalent to multipling the arrival times by a factor $1/\alpha$ and letting the service requirements intact. The effect on the server is the same as slowing down its rate.

Second, consider applying a "sampled" input process to the slower queue: Each flow is chosen independently with probability α and its service requirement is as in the original system. It is not hard to see that this sampled input process is equal to the time-stretched input process *in distribution*. This follows from the fact that independent sampling causes the sampled input process to be Poisson of rate $\alpha\lambda$ and does not affect the distribution or the i.i.d. nature of the service requirements.

According to our notation, $\tilde{Q}(t)$ is the number of jobs in the slower server fed by the sampled input process at time t. The above distributional equivalence of the time-stretched and sampled input processes implies that $\hat{Q}(t) \stackrel{d}{=} \tilde{Q}(t)$, since queue-size processes are a fixed, deterministic function of the input processes, and both these queues run slower than the original by the same factor α. Hence $Q(t) \stackrel{d}{=} \tilde{Q}(t/\alpha)$. ■

In Chapter 8 we extend this result to the case of networks with Poisson arrivals at the entry points and general independent service requirements. We then use the latter result to argue about the scaling behavior of IP networks.

It is natural to be skeptical of the relevance of these results. After all, they assume Poisson input traffic, whereas Internet packet traffic exhibits long-range dependence. Even more, these are open networks (the rate of arrivals is independent of current network congestion), quite different from the window flow-controlled Internet.

Nevertheless we find in the next chapter that the SHRiNK approach can be applied to IP networks, because it relies on factors other than packet level statistics.

Chapter 8

Applying SHRiNK to IP Networks

In this chapter we first show how to physically scale IP networks. Then, in Section 8.2 we investigate how good performance scaling is using simulations. We find that a large variety of performance measures can be predicted from a small replica of the original network. After that, in Section 8.3 we provide theoretical justification for our findings. Quite interestingly, the theory reveals that SHRiNK is widely applicable for any network topology, transport protocol, and queue management scheme. In [76], SHRiNK has been applied to networks with special input traffic. Section 8.4 briefly comments on the physical and performance scaling for this type of networks.

8.1 Physical Scaling

We start with sampling: Due to the tremendous increase in the volume and speed of network traffic, it is very expensive to sample packets. Further, as it will become clear in Section 8.3, sampling packets would not preserve the scaling property that the method exploits. At the other end of the spectrum, one may sample network sessions, e.g. modem calls, ftp, telnet, or web sessions. However, sampling sessions is hard in practice, because only end users have enough information to distinguish between different sessions. Hence, we choose to sample network flows. This reduces sampled traffic, is easy to implement in practice, and gives rise to performance scaling. (Notice that in accordance with the usual practice [31, 40, 41], packets are said to belong to the same flow if they have the same source and destination IP address, and source and destination port number. A flow is “on” if its packets arrive more frequently than a timeout of some seconds. This timeout is usually set to something less than 60 seconds.)

A second issue related to sampling is how to sample. To keep things simple, the method samples all flows in an i.i.d. fashion with some probability α. It has been argued in [42] that network flows are *as if* they were uncorrelated, hence i.i.d. sampling is appropriate. The last issue with sampling is where to sample. The method samples at network entry points, e.g. at edge routers.

What is left is to describe how to obtain the small replica of the original network. This is done as follows: (i) link capacities are reduced by a factor α, (ii) propagation delays are scaled up by a factor $1/\alpha$, and (iii) protocol timeouts are also scaled up by the same factor. Intuitively, these steps aim to slow down the speed of the network. This will become more clear in Section 8.3.

8.2 Performance Scaling of IP Networks

In this section we investigate performance scaling using the network simulator ns-2 [72]. We start by motivating the simulation decisions related to input traffic.

It has been shown that the size distribution of flows on the Internet is heavy-tailed [107]. Hence, Internet traffic consists of a large fraction of short flows, and a small fraction of long flows that carry most of the traffic. Also, it has been recently argued that since network sessions arrive as a Poisson process [33, 73, 80], [1] network flows are *as if* they were Poisson [42]. (In particular, the equilibrium distribution of the number of flows in progress is *as if* flows arrive as a Poisson process.) We take these observations into account and study the scaling behavior of IP networks carrying heavy-tail distributed, Poisson flows.

For simplicity, consider the topology illustrated in Figure 8.1. In Section 8.3 we establish that the results are independent of the particular topology. There are three routers, $R1$, $R2$ and $R3$, two links in tandem, and three groups of flows, $grp1$, $grp2$, and $grp3$. The link speeds are 10Mbps.

Routers use either DropTail or Random Early Detection (RED). DropTail refers to the queue management mechanism that drops packets once the queue is full, while RED drops packets in proportion to the current average queue size [36]. The RED parameters are $min_{th} = 100$, $max_{th} = 250$ and $w = 0.00005$. When using DropTail, the buffer can hold

[1] That network sessions are Poisson is not surprising since a Poisson process is known to result from the superposition of a large number of independent user processes.

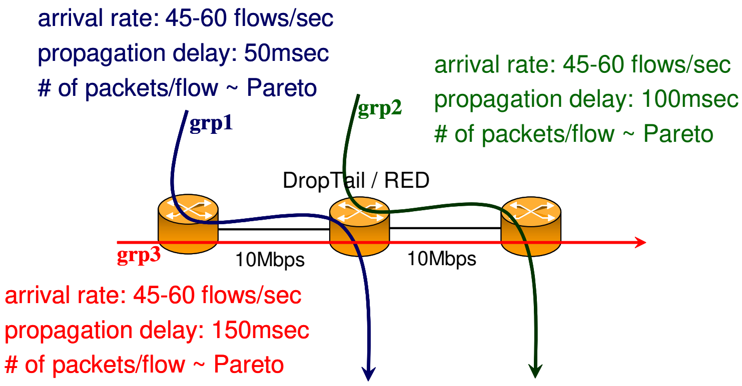

Figure 8.1: Basic network topology and flow information.

200 packets.

Within each group, flows arrive as a Poisson process with rate λ. We vary λ to study both uncongested and congested scenarios. (We use the ns-2 built-in routines to generate web sessions consisting of a single object each. This is what we call a flow in the simulations.) Each flow consists of a Pareto-distributed number of packets with average size 12 packets and shape parameter equal to 1.2. The packet size is set to 1000 bytes. The propagation delay of each flow of $grp1$, $grp2$, and $grp3$, is 50msec, 100msec, and 150msec respectively.

We run the experiments for scale factors $\alpha = 1$ and 0.1, and compare the distribution of the number of active flows as well as the histogram of the normalized delays of the flows in the original and the scaled system. (The normalized delays are the flow transfer times multiplied by α.) We also compare more detailed performance measures such as the distribution of active flows that are less than some size and belong to a particular group, and the distribution of the queue occupancies. As it will be shown in Section 8.3, the method can predict the marginal and joint distributions of a large number of performance measures.

8.2.1 Uncongested Networks

We start with simulations where no drops occur. The flow arrival rate is set to 45 flows/sec within each group. Routers use RED as a queue management scheme.

Figure 8.2 plots the distribution of the number of active flows in the first link between routers $R1$ and $R2$. It is evident from the plot that the two distributions match.

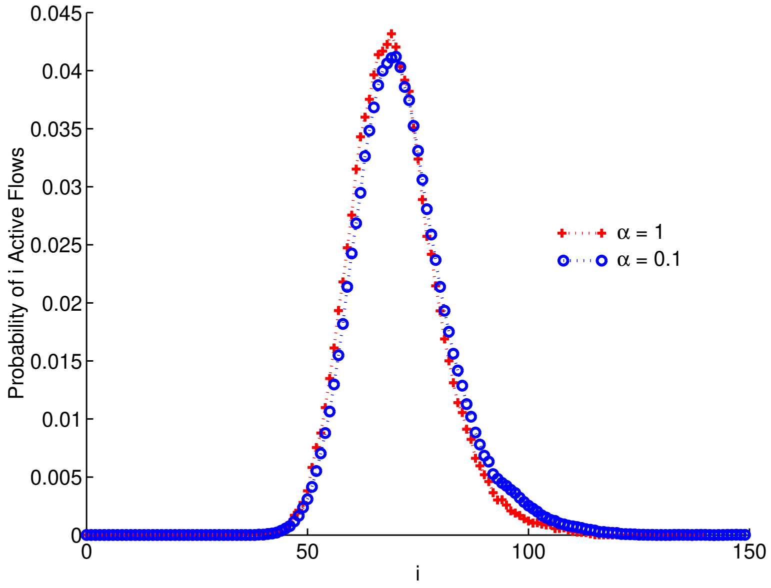

Figure 8.2: Distribution of number of active flows on the first link when no drops occur.

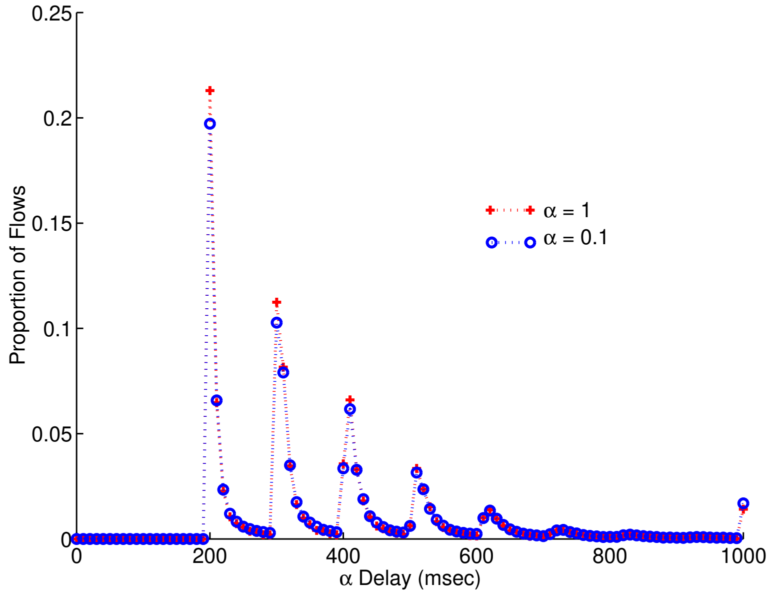

Figure 8.3: Histogram of normalized delays of $grp1$ flows when no drops occur.

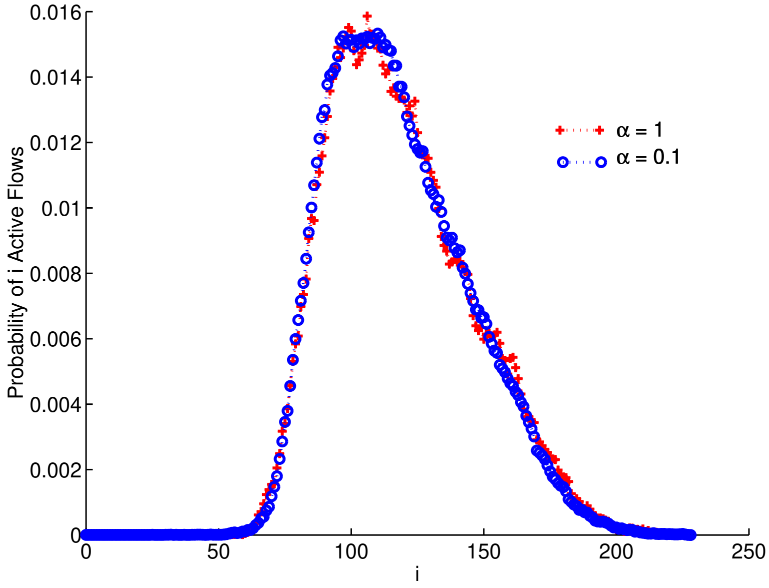

Figure 8.4: Distribution of number of active flows on the first link.

Figure 8.3 plots the histogram of the normalized delays of the flows of $grp1$. To generate the histogram, we use normalized delay chunks of 10msec each. There are 100 such delay chunks in the plot, corresponding to flows having a normalized delay of 0 to 10msec, 10msec to 20msec, and so on. The last delay chunk is for flows that have a normalized delay of at least 1sec. It is evident from the plot that the distribution of the normalized delays match.

Similar are the results for the second link, and for the other two groups of flows. Also, performance scaling occurs when routers use DropTail instead of RED. We do not present these plots here. Instead, we move to the more challenging and realistic case of congested networks, where we presents all the plots in detail.

8.2.2 Congested Networks

The flow arrival rate is set to be 60 flows/sec within each group. Flows experience drops that account for up to 5% of the total traffic. We first present simulations where all three routers use RED.

Figure 8.4 plots the distribution of the number of active flows in the first link. The two distributions match. A similar scaling holds for the second link as it is shown in Figure 8.5. The reason why the distribution of active flows in the second link is more to the right than

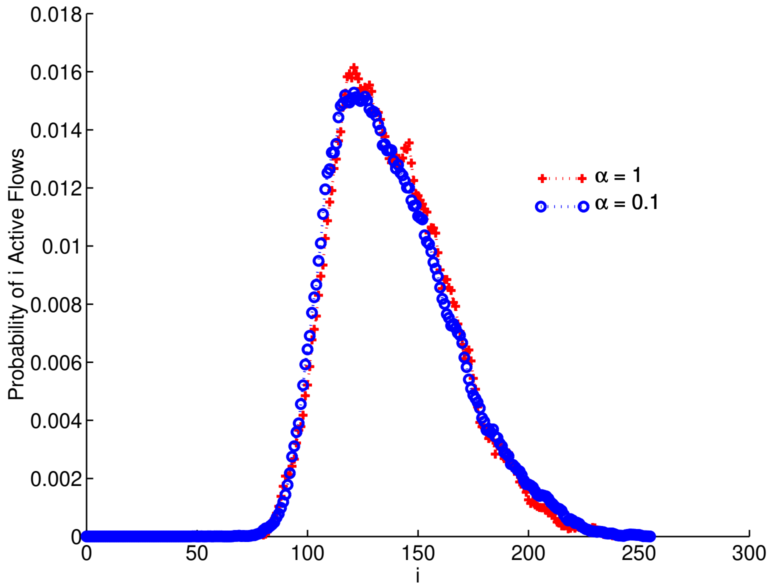

Figure 8.5: Distribution of number of active flows on the second link.

in the first link, is the larger propagation delays of the flows traversing the second link.

Figure 8.6 plots the histogram of the normalized delays of the flows of $grp1$. To generate the histogram, we use again normalized delay chunks of 10msec each. It is evident from the plot that the distribution of the normalized delays match. The results for the other two groups of flows are similar, as illustrated in Figures 8.7 and 8.8.

The peaks in the delay plots are due to the TCP slow-start mechanism. The left-most peak corresponds to flows which send only one packet and face no congestion. These flows only have to wait for the setup of the TCP connection. (Hence, for example, in Figure 8.6 where propagation delays are 50msec, the normalized delay for these flows is a bit more than 200msec accounting for SYN, SYN-ACK, the data packet, the ACK for the packet, and insignificant transmission and queueing delays.) The portion of the curve between the first and second peaks corresponds to flows which send only one packet and face congestion (but no drops). The next peak corresponds to flows which send two or three packets and face no congestion. These flows have to wait for an additional round trip time for the acknowledgment for the first packet to arrive. The third peak corresponds to flows which send between four and seven packets and face no congestion, and so on. [2]

[2]Whenever an acknowledgment arrives, the sender doubles its window.

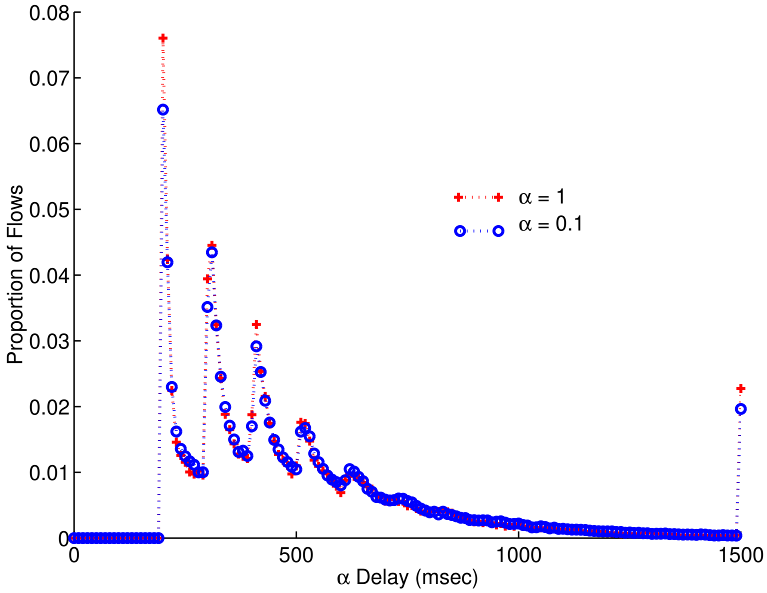

Figure 8.6: Histogram of normalized delays of $grp1$ flows.

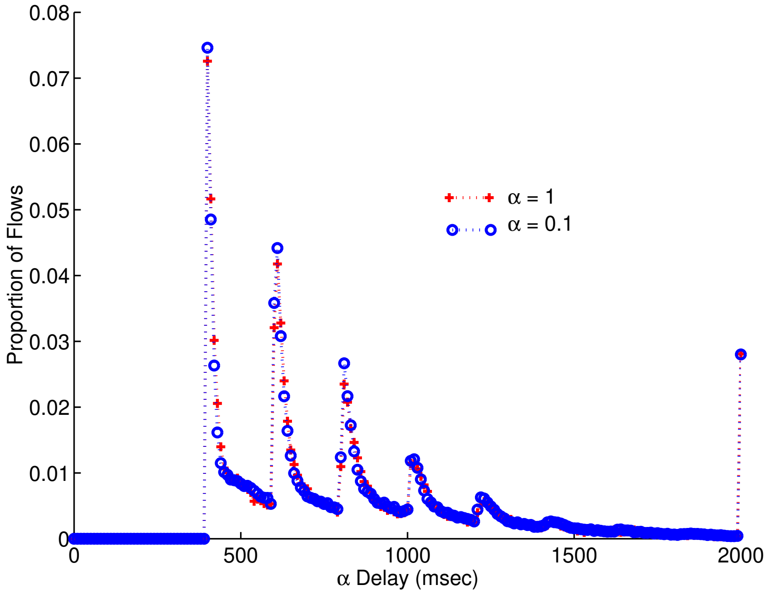

Figure 8.7: Histogram of normalized delays of $grp2$ flows.

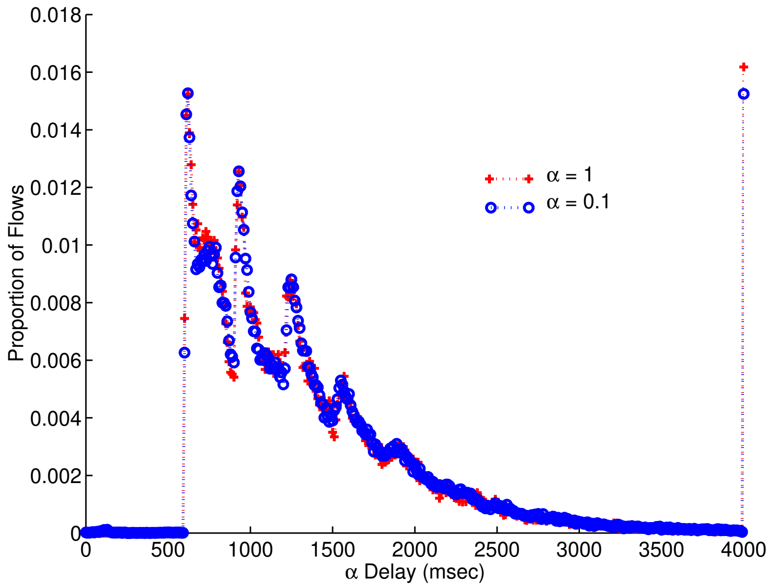

Figure 8.8: Histogram of normalized delays of $grp3$ flows.

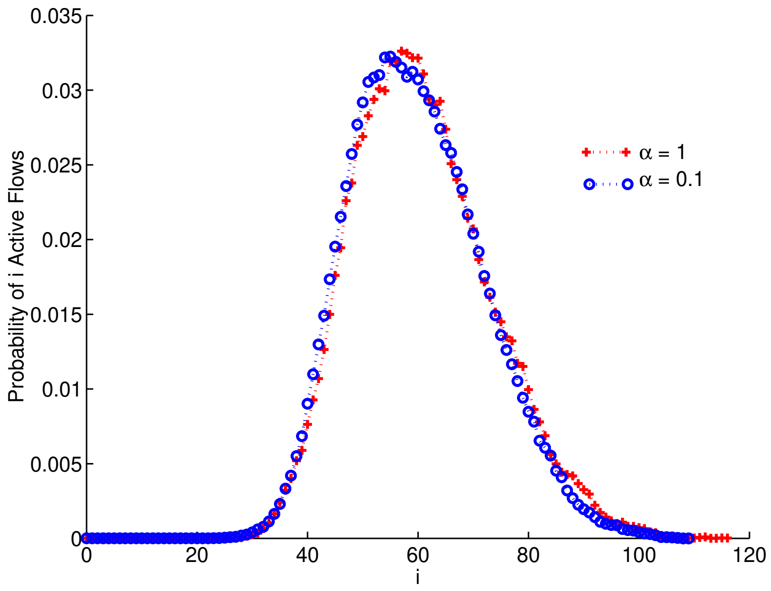

Figure 8.9: Distribution of number of active $grp3$ flows with size less than 12 packets.

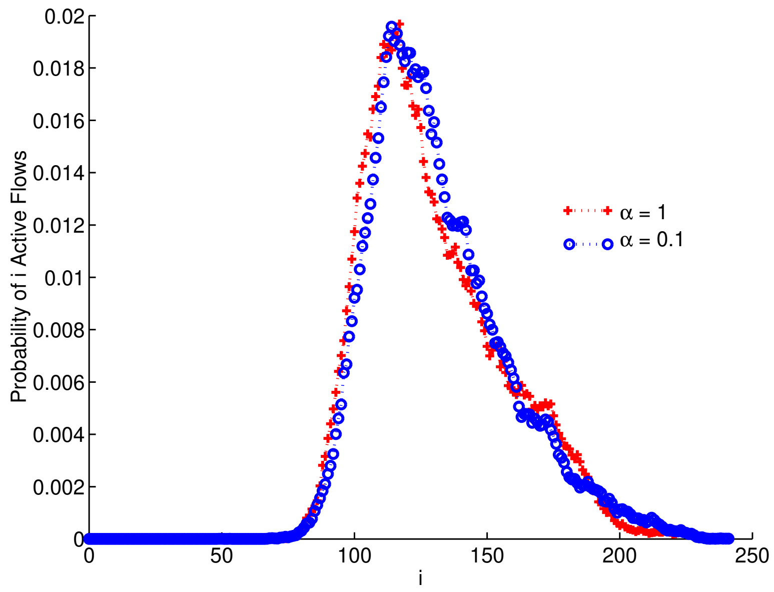

Figure 8.10: Distribution of number of active flows on the second link when DropTail is used.

What about more detailed performance measures? As an example, we compare the distribution of active flows belonging to $grp3$ that are less than 12 packets long. Figure 8.9 compares the two distributions from the original and scaled system. Again, the plots match.

We will now investigate if distributions scale when DropTail is used. Figure 8.10 plots the distribution of the number of concurrently active flows in the second link between routers $R2$ and $R3$ when all routers use DropTail. It is evident from the plot that the two distributions match as before. A similar scaling holds for the other link.

Figure 8.11 plots the histogram of the normalized delays of the flows of $grp2$ when DropTail is employed. The distributions match as before. A similar scaling holds for the other two groups of flows.

So far, the method has successfully predicted the distribution of various performance measures at the *flow* level. Figure 8.12 compares the distribution of the number of *packets* at the first queue (just before the first link) in the original and scaled network. As it is evident from the plot, the method can also predict the distribution of the queue occupancies.

A word on the convergence of the results is in order. In the implementation of ns-2,

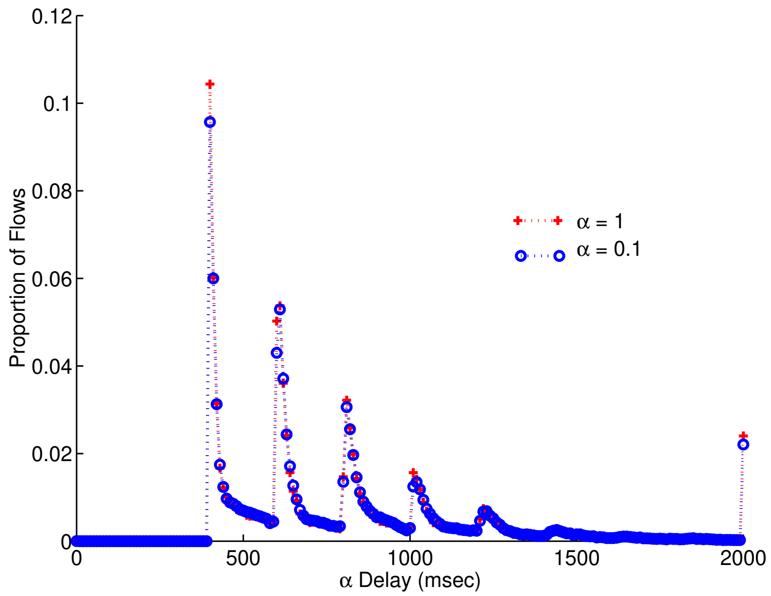

Figure 8.11: Histogram of normalized delays of $grp2$ flows when DropTail is used.

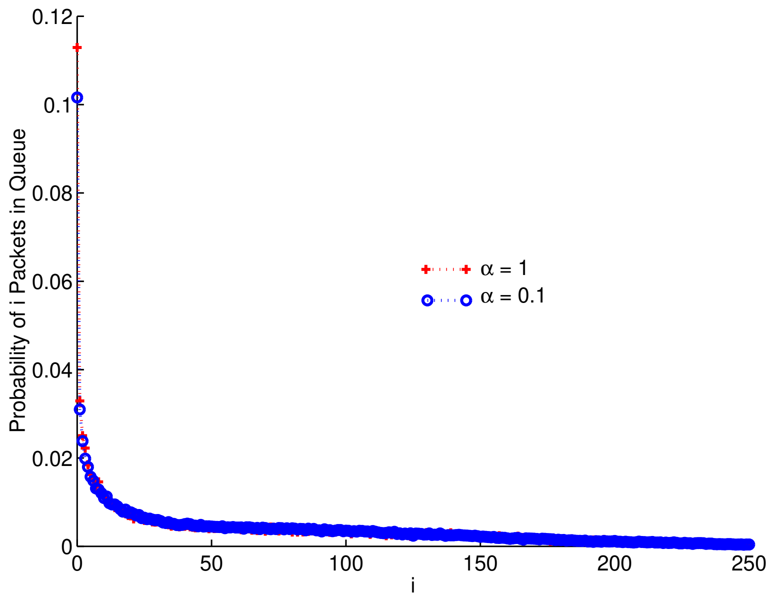

Figure 8.12: Distribution of number of packets in $R1$.

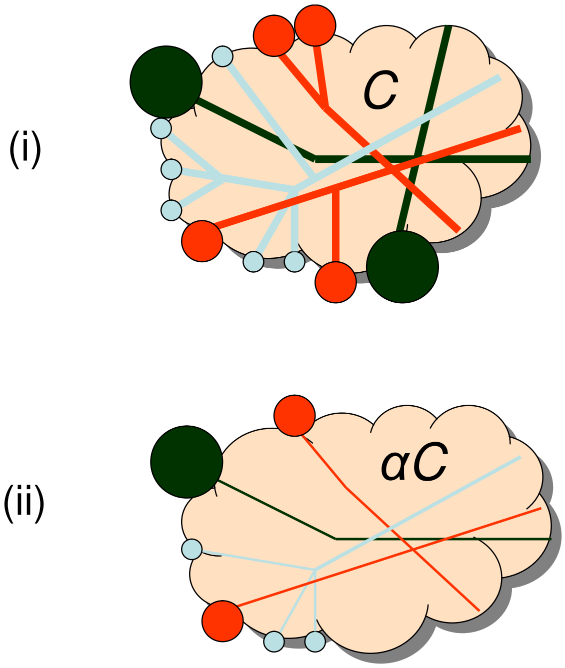

Figure 8.13: Picturing the network as a server servicing flows: (i) original network, (ii) scaled network.

every scheduled flow requires the allocation of some memory in advance. This limits the total number of flows that one can schedule per experiment, and hence limits the duration of the experiment. This, together with the heavy-tailed nature of the traffic, makes it hard to achieve very good convergence. We believe this to be the source of the very small disagreements in the distributions of the various performance measures of the original and scaled system. This belief is justified by the theoretical argument that follows in the next section and explains the observed results.

8.3 Understanding SHRiNK

It helps to think of the network as a cloud that services flows as illustrated in Figure 8.13. Flows are denoted by bubbles that sit at the edge of the network. The network services flows by transferring their data from one place to another, and quantities like number of flows under service, and mean service time of flows are well defined. Figure 8.13(i) corresponds to the original network, and figure 8.13(ii) to the scaled network.

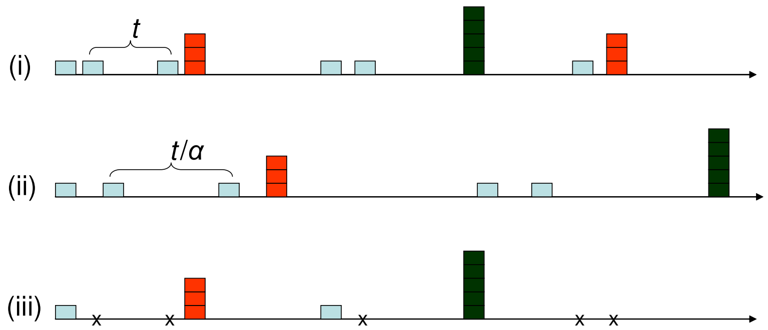

Figure 8.14: Input processes of: (i) the original, (ii) the time-stretched, and (iii) the scaled system.

Recall that flows arrive as a Poisson process. Further, distinguish between the service requirements or sizes of flows which are independent with each other, and the service times or delays, which are usually dependent. The above observations allow the use of M/GI-type queueing models to analyze SHRiNK.

Call *state of the network* at some time t the information that is needed to resume the time evolution of the network from this time on. For example, the state consists of information about currently active flows, their transfered packets, their packets in transit and where they are at time t, etc. We use a similar argument to that used in Theorem 7.1 to prove the following:

Theorem 8.1 *Let $S(t)$ be the state of the original network at time t, and $\tilde{S}(t)$ be the state of the scaled network at time t. Then $S(t) \stackrel{d}{=} \tilde{S}(t/\alpha)$, i.e. the same in distribution.*

Proof: Let a system be a network and its input and consider three systems. First, the original system. Without loss of generality, let Figure 8.14(i) illustrate the input process, $I(t)$, of the this system. The graph denotes flow arrival times and their corresponding sizes.

Now, scale up time by a factor $1/\alpha$, and construct a time-stretched system. The input process, $\hat{I}(t)$, of this system is a time-stretched version of $I(t)$ as illustrated in Figure 8.14(ii). Whatever happens in the original system at time t, happens in this system at time t/α. This is illustrated in Figure 8.15 that shows the time evolution of the network between a source-destination pair for the three systems. (In each sub-figure, the corresponding input process is shown on the left. In the time-stretched system packets have larger transmission and propagation delays, denoted by "fatter" parallelograms going from left to right and

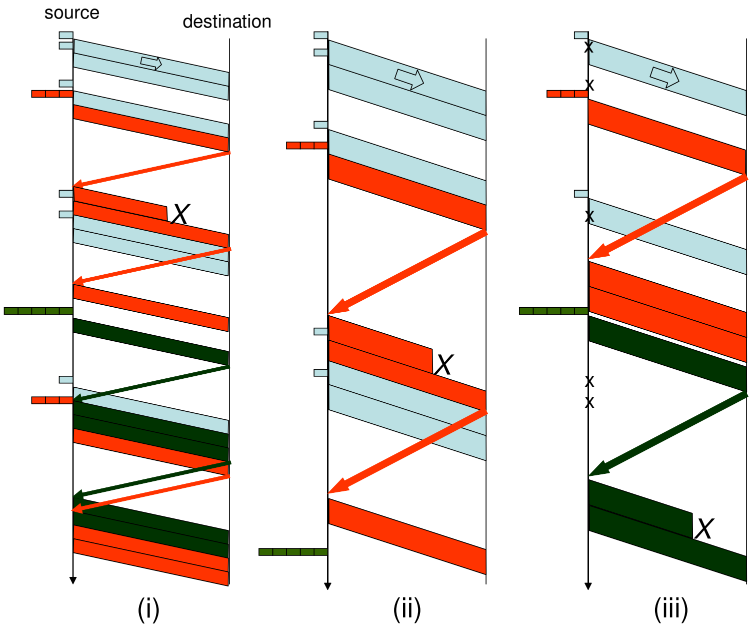

Figure 8.15: Time evolution of: (i) the original, (ii) the time-stretched, and (iii) the scaled system.

larger slopes respectively. The thin lines going from right to left denote acknowledgments. Finally, the large "X"'s denote packet drops.) By stretching time, the original network is effectively slowed down by a factor α. Notice that the time-stretched system is just a devise for the proof, it does not exist.

The third system is the scaled system. Figure 8.14(iii) illustrates the input process, $\tilde{I}(t)$, of this system that occurs after sampling i.i.d the flow-arrivals of the original system. The scaled network also runs slower than the original by a factor α. This is by construction: the physical scaling consists of all necessary steps for this to hold, since propagation delays and protocol timeouts are scaled up by a factor $1/\alpha$, and link speeds are reduced by α.

Now, observe that the state of any of the three systems is a function of the input process *only*. This function describes what the network does: the TCP mechanism, the protocol timeouts, the active queue management schemes, etc, and acts at the inputs to produce the state of the network. It is a rather complicated function, but it exists. [3] Let F, $\hat{F}$, and $\tilde{F}$ be this function for the original, the time-stretched, and the scaled system respectively.

[3] Notice that there is no need to compute this function, it suffices to know that it exists.

Hence, for example, $S(t) = F\left(I(\tau), 0 \le \tau \le t\right) = F(I, t)$.

In the rest of the proof we relate the state of the original and scaled system, using the time-stretched system as the glue between them. First, it is easy to see that

$$F(I,t) = \hat{F}(\hat{I}, t/\alpha),$$

for all t.

Second, observe that, by construction, $\hat{F} = \tilde{F}$, i.e. they are the same functions. Also, $\hat{I}(t) \stackrel{d}{=} \tilde{I}(t)$. This follows from the fact that independent sampling with some probability α of a Poisson process causes the sampled input process to be Poisson of rate α-times the original rate, and does not affect the distribution or the i.i.d. nature of the service requirements. Thus,

$$\hat{F}(\hat{I}, t) \stackrel{d}{=} \tilde{F}(\tilde{I}, t).$$

Therefore,

$$S(t) = F(I,t) \stackrel{d}{=} \tilde{F}(\tilde{I}, t/\alpha) = \tilde{S}(t/\alpha).$$

■

This theorem explains why the distributions of various performance measures are matching in the simulations. Further, it shows that performance scaling involves speeding up the time, and this is why we compare normalized delays rather than delays.

The proof of the theorem only relies on the assumptions on the inputs: Poisson flow arrivals, and independent service requirements. Hence, the argument reveals that SHRiNK is widely applicable for marginal and joint, steady-state and transient distributions of a large family of performance measures, for any network topology, transport protocol, and queue mechanism.

Another consequence of Theorem 8.1 is that SHRiNK works for any value of α. Thus, networks can be slowed down arbitrarily. However, the smaller the α, the slower the network is, and the longer it takes for distributions to converge.

A comment on the assumption of Poisson flow arrivals is in order. (A brief motivation for this assumption was given at the beginning of Section 8.2.) Network flows don't have to be Poisson; it suffices that network sessions are Poisson and the network consists of links and nodes that can be modelled as "symmetric queues" [58]. Then, network flows are *as if* they were Poisson (see Chapter 3 in [58]) and the proof goes through.

It is a well documented fact that network sessions arrive as a Poisson process [33, 73, 80]. Whether the network can be modelled by symmetric queues is a question that was recently answered affirmatively in [42]. First, the authors model a network session as a sequence of flows separated by think times. Then, they argue that the think time between successive flows of a session and the links that carry TCP flows which share the link bandwidth, can be both modelled as symmetric queues. In [13], it is shown that this model can lead to self-similar traffic which is in accordance with real traces.

As a final comment, in practice, it suffices that session arrivals are close to Poisson, and the properties of network links and nodes are close to those of a symmetric queue, for performance scaling to occur with reasonable accuracy.

8.4 SHRiNKing Networks with Long-lived Flows

SHRiNK has been also applied to networks in which the traffic consists of long-lived TCP-like flows arriving in clusters. In contrast, the results we presented in Sections 8.2 and 8.3 are for IP networks with flows that arrive at random times and whose size is heavy tailed.

The physical scaling in the long-lived flows case is a bit different and consists of the following steps: (i) sample flows in an i.i.d fashion with probability α, (ii) reduce link speeds by the same factor α, and (iii) multiply buffer sizes and various AQM-specific parameters, like the minimum and maximum thresholds of RED, by α.

Performance scaling is also different. In the case of long-lived flows, we find using simulations and theory that performance measures such as queueing delay and drop probability are left virtually unchanged as a function of *time* [79, 84]. (In Section 8.2 it is the *distribution* of performance measures that scales.)

Theorem 8.1 reveals that SHRiNK is widely applicable in the case of heavy-tailed distributed flows that arrive at random times. In the long-lived flows case, the required assumptions are more restrictive and as a result the applicability of the technique is narrower. For example, for reasons explained in [78], performance scaling does not occur when DropTail is used.

Networks with long-lived flows arriving in clusters have been used in the literature to test the behavior of control algorithms and queue management schemes[44, 48, 60], but they are not a realistic representation of today's Internet. The networks that we study in this thesis consist of flows that arrive at random times and whose size is heavy tailed, and they are a more accurate representation of the actual Internet. The issue of which traffic models are useful in Internet research is controversial (see, for example, [37]), but it is unrelated to the SHRiNK methodology: with a different physical scaling, performance extrapolation occurs for both types of traffic models.

For more information on the long-lived flows case the interested reader is referred to [76, 79].

8.5 Conclusions and Applications

In the previous chapter we have described an approach, called SHRiNK, for scalable performance prediction of large systems. In this chapter we applied the method to IP networks in which flows come and go at random times, and the size distribution of the flows is heavy-tailed. These networks are a plausible representation of the actual Internet.

Using simulations, we have found that performance measures like the distribution of active flows and flow service times can be inferred quite accurately from measurements of a scaled-down system. A theoretical argument reveals that the method is widely applicable for marginal and joint distributions of a large family of performance measures, for any network topology, transport protocol, and queue management mechanism.

Since the method provides a way to deduce the performance of a fast network from a slowed-down replica, it can be used to reduce the cost of experimentation: Imagine a test-network with slow network interfaces, slow switches and routers, and cheap links, that is fed with a sample of the actual network traffic. [4] In this network one may experiment with new algorithms, protocols, and architectures, and extrapolate performance.

Another use of the method is the following: There has been a recent development of research prototypes and products [23] that record partial information about the network by sampling incoming traffic. SHRiNK offers a systematic way to reproduce offline the whole behavior of the network using this sample.

[4] This network should also have larger propagation delay than the original. This can be achieved in software, or with delay-loops.

Chapter 9

Applying SHRiNK to Web Farms

In the previous chapter SHRiNK has been applied to IP networks. In this Chapter we apply the method to another large-scale system, namely to web farms.

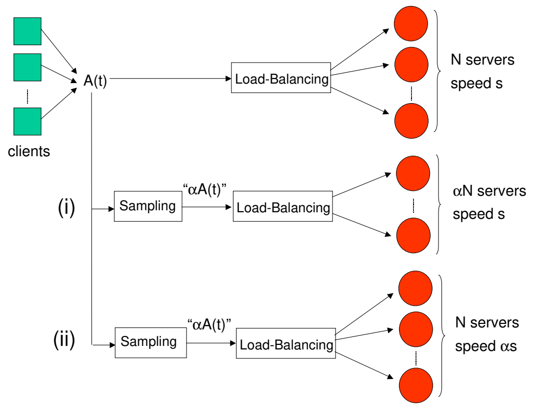

Figure 9.1: Scaling a web server farm:[1] (i) scaling the number of servers, (ii) scaling the speed of the servers.

Consider a web server farm with N servers each having speed s, as in Figure 9.1. Sample in an i.i.d. fashion with probability α the requests destined for the original farm. (We elaborate more on sampling in the following sections, where for every experiment we discuss whether sampling takes place at the session, document, or embedded request level. A web browsing session is a series of document requests initiated by a user, with small silent

[1]This is a simplified picture of a farm, since the application-servers, the databases, and the switches used to interconnect the various components are absent.

periods between consecutive requests. Each document request usually initiates a bunch of embedded requests. These requests are usually for images embedded in the document.)

Feed the sampled traffic into a farm consisting of either (i) a fraction α of the original web servers, or (ii) the same number of servers each having speed αs (see (i) and (ii) of Figure 9.1). We wish to investigate how close the average response time, the server throughput, and the capacity (maximum throughput) in the scaled system are to those in the original system.

9.1 Scaling the Speed of Servers

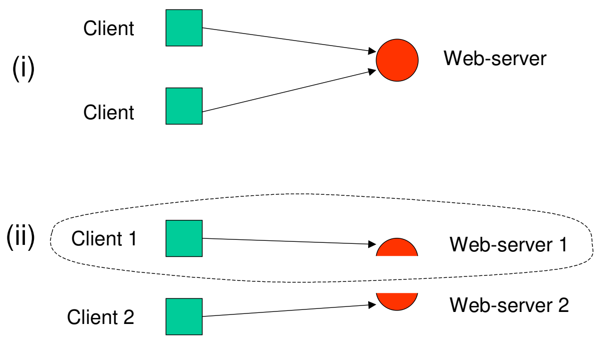

Figure 9.2: Experimental setup: (i) the unscaled system, and (ii) effectively running a server at a fraction $\alpha = .5$ the speed, by running two servers on a single machine.

In practice, it is not straightforward to simply run a server at a slower speed. Instead, we use the setup shown in Figure 9.2: Three Linux machines, configured with a Pentium II at 300MHz and 256MB of RAM, are connected to a 100Mbits/sec switch. A variable number of Apache 1.3.9 [2] web servers is hosted at one of the machines, and the other two machines run Surge [8] HTTP traffic generators. Running two web servers on a single machine has the effect of halving the speed of each: when the load is light, speed is not a constraint either for the two web servers or for one at half the speed; when load is heavy, each of the two servers runs most of the time with its maximum number of concurrent httpd processes, so they share the speed of the machine between them equally.

We compare the average response time of the following two systems:

I. Original system, illustrated in Figure 9.2(i): The hosting machine hosts a single web server, which serves requests from both client machines.

II. Scaled system, illustrated in Figure 9.2(ii): The hosting machine hosts two web servers, each listening on a different IP address. Each of the client machines generates requests destined for only one of the servers. One client-server pair is used only to consume half of the resources while the other pair is used to record the results for the scaled system.

The web files of both systems are generated using Surge, configured with the default parameters. Surge generates web traffic by creating a number of *userequivalents*, each of which generates HTTP requests in the following manner: First, an HTTP object is requested. Objects consist of a set of embedded requests and model a user's "click", i.e. a request for a web document. Then, the user equivalent waits till it receives the responses for all the embedded requests. After that, it sleeps for some random time. Finally, a new object is requested and so on and so forth. [2] Notice that in the scaled system described above, we essentially sample user equivalents.

We have performed experiments with HTTP/1.0, HTTP/1.1 without pipelining, and HTTP/1.1 with pipelining. The results are similar, so we only show results for the latter case. Further, we show results for the cases where the bottleneck is the maximum number of concurrent httpd processes, or the server CPU. Unless otherwise stated, the Apache servers use the default configuration parameters.

9.1.1 Experimental Results

Apache's default limit on the number of concurrent httpd processes is 150. In the scaled system, we set a limit of 75 for each of the two servers. Figure 9.3 plots the mean response time of HTTP objects as a function of the normalized load, that is, the load multiplied by α^{-1}. The load is measured in user equivalents. Figure 9.4 is the same plot zoomed in. We see that scaling the system leaves the mean response time virtually unchanged. (So,

[2] User equivalents are like web sessions with an important difference: they are all created at the beginning of the experiment and terminate at the end, instead of arriving and terminating at random times.

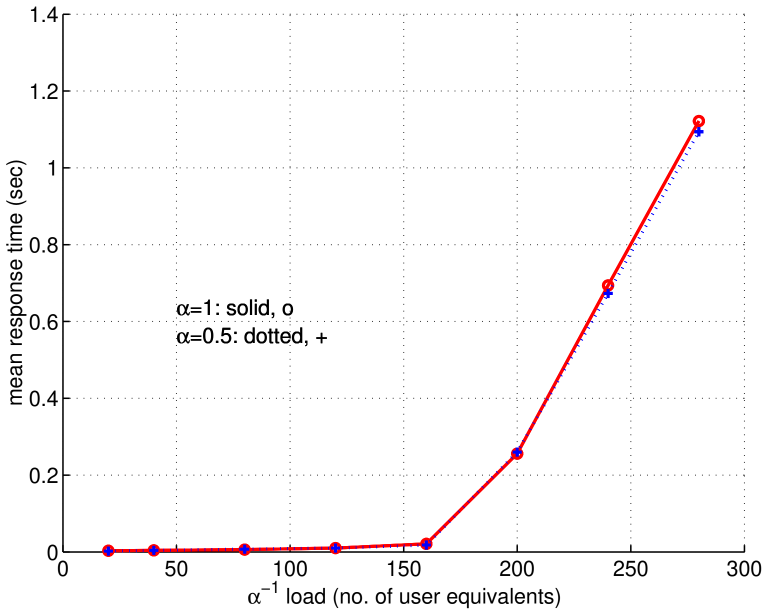

Figure 9.3: Average response time, when the bottleneck is the number of concurrent processes.

for example, the mean response time for the original system serving 200 user equivalents is the same as that for the scaled system serving 100 user equivalents.) Figure 9.5 shows the normalized server throughput and capacity, that is, the server throughput and capacity multiplied by α^{-1}, as a function of the normalized load. Again, scaling the system changes these quantities by very little.

To make the CPU the bottleneck, we recompile Apache to allow a large number of concurrent httpd processes, and set their maximum number to 800 in the original server and to 400 in each of the scaled servers. Figures 9.6 and 9.7 show the average response time and the normalized server throughput as a function of the normalized load. Again, scaling the system leaves these quantities virtually unchanged. Notice that, as expected, the server capacity is increased due to allowing more concurrent connections.

An intuitive way to see why performance scaling occurs in these experiments is the following: Let s be the speed of the server and n be the number of concurrent httpd processes in the original system. Then, since in the scaled system there are two web servers hosted on the same machine that receive half of the traffic each, each server has speed $s/2$, and the number of concurrent httpd processes on each server is $n/2$. Hence, since CPU

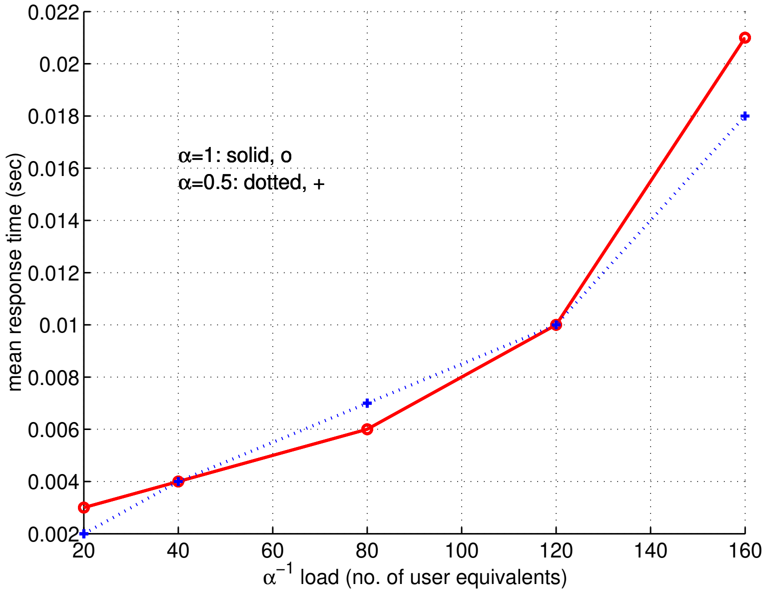

Figure 9.4: Average response time, when the bottleneck is the number of concurrent processes (zoomed in).

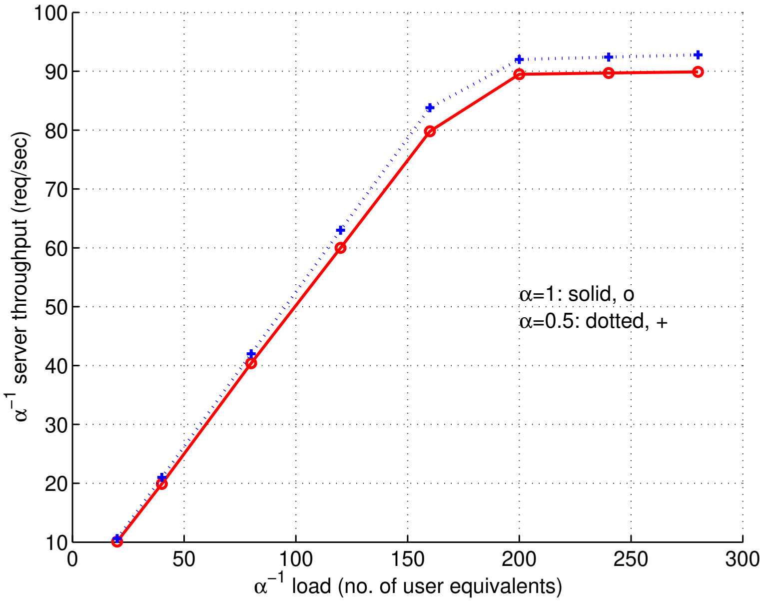

Figure 9.5: Server throughput, when the bottleneck is the number of concurrent processes.

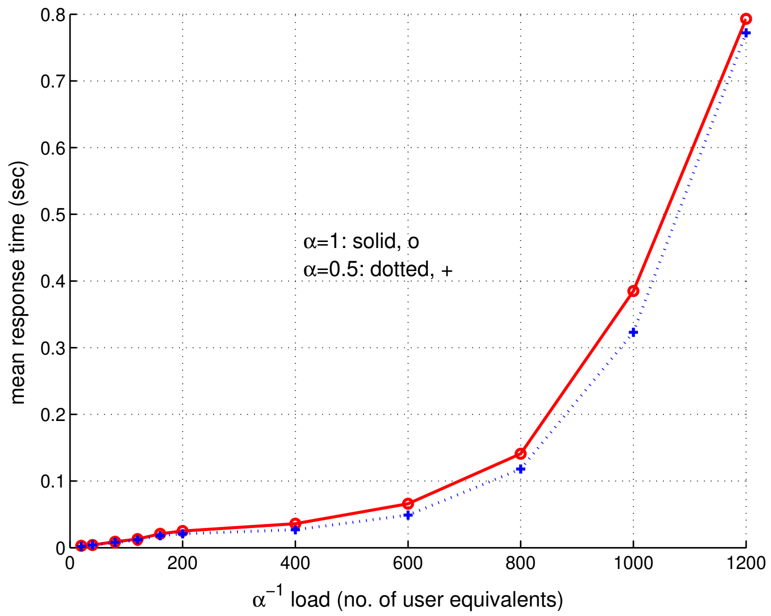

Figure 9.6: Average response time, when the CPU is the bottleneck.

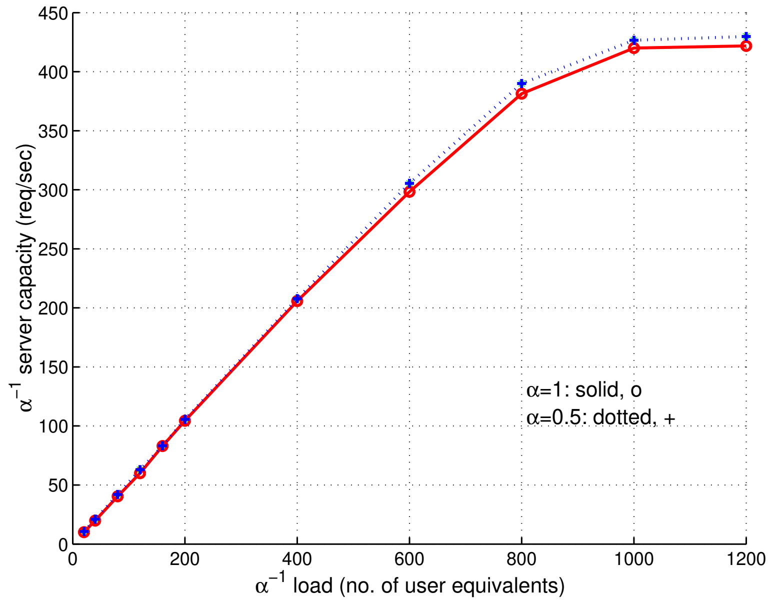

Figure 9.7: Server throughput, when the CPU is the bottleneck.

operates in a processor sharing manner, each job gets the same service rate in the two systems and mean response times remain unchanged.

9.2 Scaling the Number of Servers

We investigate the validity of SHRiNK in server farms when one scales the number of servers of the farm.

Consider the farm illustrated in Figure 9.1. When servers are identical, it is straightforward to scale their number. If there are K types of servers, we simply scale down by the same proportion the servers of each type.

Sampling is more involved. There are two levels at which one may perform the sampling, namely at the object or the user-equivalent level. [3] When HTTP1.1 is used, it is important to sample at the same level at which load balancing takes place. To see this, suppose load balancing takes place at the user-equivalent level, but sampling takes place at the object level (see Figure 9.1). Then, since HTTP1.1 maintains persistent connections between the clients and the servers, a server will have more concurrent connections in the scaled system than in the original system. (This is because there are more user equivalents per server in the scaled system than in the original.) We perform experiments and verify that SHRiNK is valid when load-balancing and sampling take place at the same level, while when this constraint is not met the mean response time in the two systems differ. This is not an issue when HTTP1.0 is used, since HTTP1.0 does not use persistent connections.

We illustrate the above points by presenting some experimental results. In the experiments we use eight Linux machines configured with a Pentium III at 550MHz and 384MB of RAM, connected to a 100Mbits/sec switch. Machines acting as servers host one Apache 1.3.9 web server, and machines acting as clients run Surge to generate HTTP requests.

9.2.1 Experimental Results

In the first experiment the clients use HTTP1.1. Load-balancing is a simple round-robin scheme. Both load-balancing and sampling take place at the user-equivalent level. We compare the average response time of the following two systems:

[3] Sampling at the embedded request level would not work well because the sample doesn't capture the structure of the traffic.

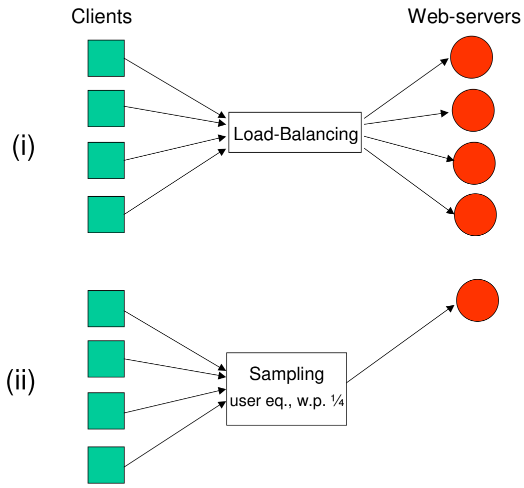

Figure 9.8: Experimental setup using HTTP1.1: (i) the unscaled system, and (ii) scaling the number of servers and sampling incoming user equivalents, $\alpha = .25$.

I. Original system, illustrated in Figure 9.8(i): Four machines act as servers, and four as clients.

II. Scaled system, illustrated in Figure 9.8(ii): The user equivalents of the four clients are sampled in an i.i.d. fashion with probability 1/4 and are directed to a single server machine.

Figures 9.9 and 9.10 show the average response time and the normalized server throughput as a function of the normalized load. Again, scaling the system leaves these quantities virtually unchanged. Note that we treat the farm of the four servers as a single entity. The normalized load is the total normalized load directed into the farm, and the normalized throughput is the sum of the normalized throughputs of the servers of the farm.

Intuitively, performance scaling occurs in this experiment because the total number of user equivalents directed to the stand-alone server of the scaled system is the same with the total number of user equivalents directed to each of the servers of the original farm.

In the second experiment the servers receive HTTP1.0 requests. In practice, this happens either when clients use HTTP1.0., or more frequently due to a proxy that may reside

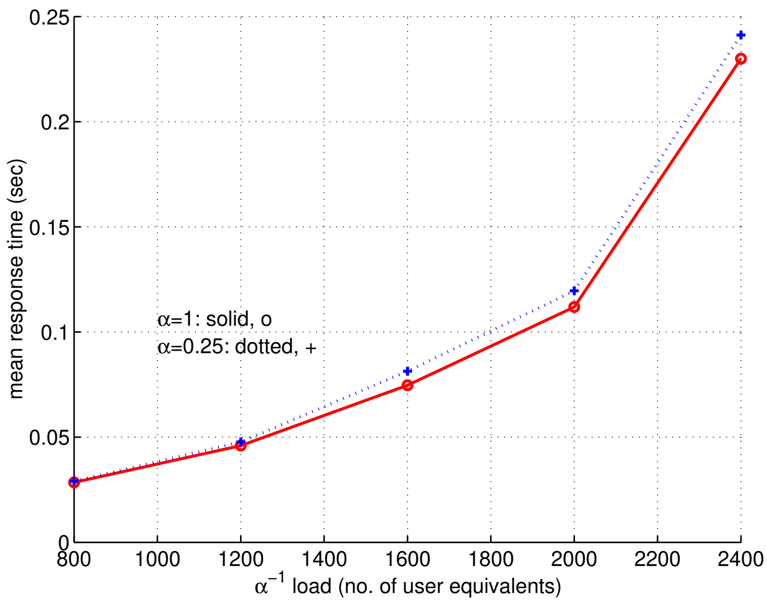

Figure 9.9: Average response time, when sampling user equivalents.

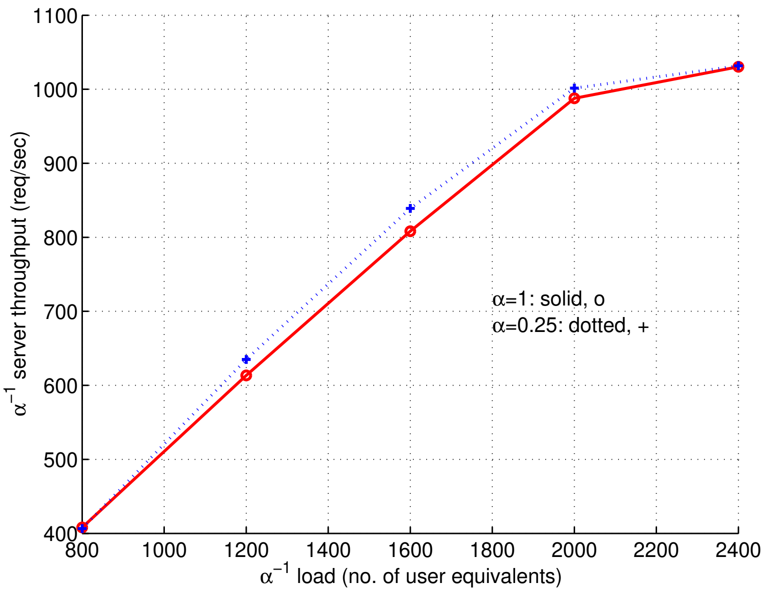

Figure 9.10: Server throughput, when sampling user equivalents.

at the entry point of a farm and convert HTTP1.1 requests to HTTP1.0 requests.

(The rational of using this proxy is the following: The persistent connections that HTTP1.1 maintain reduce the average response time, when the network delay between the client and the server is large. However, if the client and the server are connected at the same LAN, it is faster to use HTTP1.0 because as the number of clients increases the overhead of persistent connections is large. Hence, using a proxy at the entry point of a farm to convert HTTP1.1 requests to HTTP1.0 requests and visa versa, exploits the advantages of both versions of the HTTP protocol.)

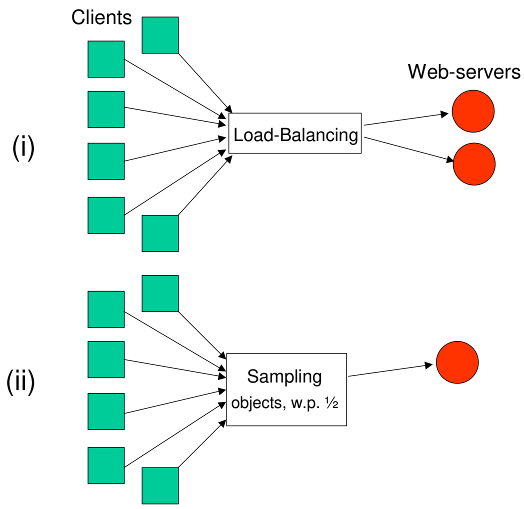

Figure 9.11: Experimental setup using HTTP1.0: (i) the unscaled system, and (ii) scaling the number of servers and sampling incoming HTTP objects, $\alpha = .5$.

Load-balancing is again a simple round-robin scheme at the user-equivalent level, while sampling takes place at the object level. [4]

We compare the average response time of the following two systems:

I. Original system, illustrated in Figure 9.11(i): Two machines act as servers, and six machines act as clients. (We want to be able to saturate the servers. In the previous experiment four client machines are enough to achieve that, since HTTP1.1 is used

[4]Due to the close-loop manner by which Surge generates requests, when an object is not sampled, we make the corresponding user equivalent to sleep for some time to account for the object service time and the sleep time that it would wait if the object were requested.

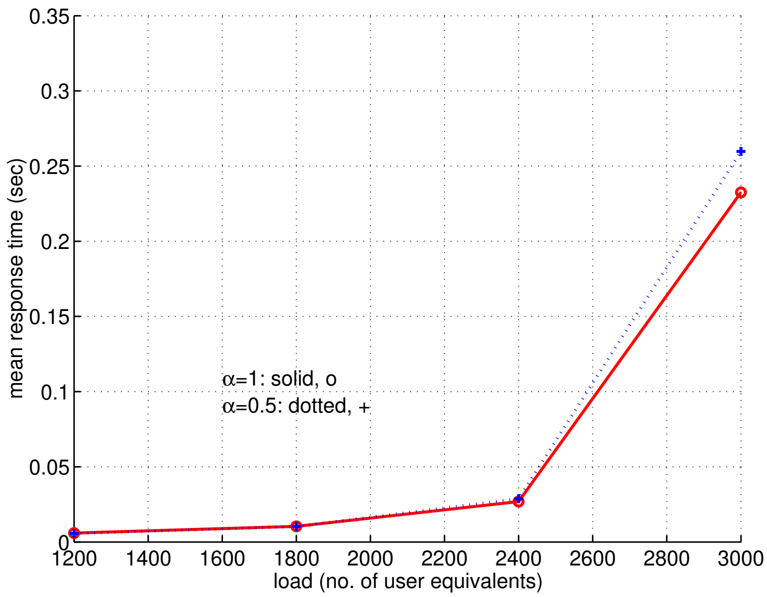

Figure 9.12: Average response time, when sampling HTTP objects.

and we artificially saturate the farm by setting the maximum number of concurrent connections to 500. Here, since HTTP1.0 is used, we need to saturate the CPU utilization of the server machines and hence we need more client machines.)

II. Scaled system, illustrated in Figure 9.11(ii): The HTTP objects of the six clients are sampled in an i.i.d. fashion with probability 1/2 and are directed to a single server machine.

Figures 9.12 and 9.13 show the average response time and the normalized server throughput as a function of the load. [5] Again, these quantities remain virtually unchanged after scaling. Note that we treat the farm of the two servers as a single entity.

9.2.2 Simple Analysis

Consider the special case where the servers of the original farm operate independently and one samples web sessions. If the load-balancer assigns sessions to servers uniformly at random then SHRiNK is valid. To see this, note that web sessions are Poisson[32] and let

[5] The number of user equivalents sending requests at the two systems is now the same hence the horizontal axis is not multiplied with α^{-1} as before. It is the number of objects and requests directed at the two systems that differ due to object sampling.

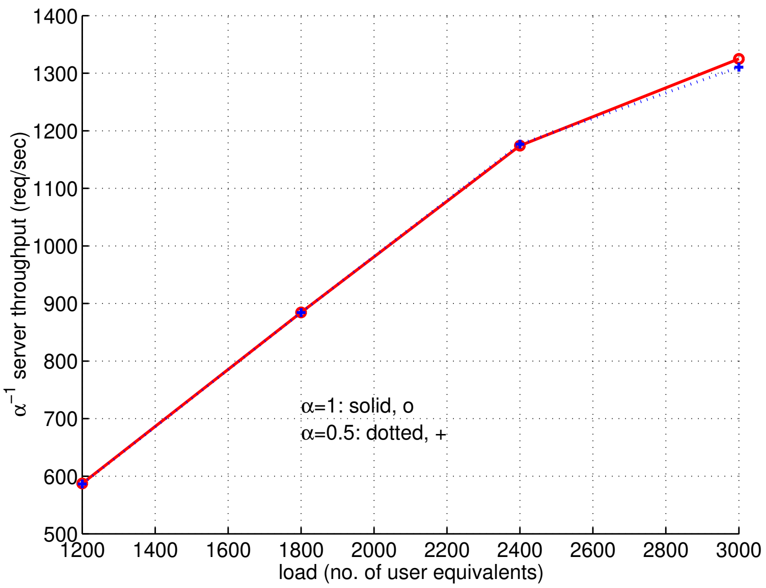

Figure 9.13: Server throughput, when sampling HTTP objects.

the session arrival rate be λ. Sample a proportion α of the sessions, and keep a proportion α of the N servers in the original web farm. Then the arrival process to each server in the original system is Poisson with rate λ/N, and the arrival process to each server in the scaled system is Poisson with the same rate $\alpha\lambda/\alpha N = \lambda/N$. Hence, mean response times are the same.

Is this analysis relevant to practice? Web servers may not be independent, either by virtue of sharing resources such as databases and bandwidth, or due to load balancing. However, usually databases and bandwidth are not the bottleneck on a farm. Further, the most commonly used load-balancing scheme is round-robin. Hence, the dependencies among web servers are usually insignificant. Another element of real web farms that is missing from our experiments is large network delays and packet losses due to slow or congested connections between the clients and the servers. Recently [69], researchers have shown that these factors may significantly affect server performance and response times. We believe the validity of SHRiNK will not be affected by wide-area conditions, since both the original and scaled web-farms face the same conditions.

The analysis above can be also used to shed light on the experimental results when one samples HTTP objects. First, the round-robin load balancing mechanism that is used does not introduce significant dependencies among the servers. Second, the sampled traffic is close to Poisson. To see this, note that HTTP objects are generated by independent user equivalents. Further, since the sleep time is usually significantly larger than the transfer time, the object interarrival times can be considered more or less independent. Hence, the aggregate object arrival times are close to Poisson, and the analysis goes through.

As a final comment, note that Figure 9.1 implies that one fast server is equivalent to many slow servers. This is not always the case. For example, if incoming requests are serviced in a FIFO manner, the fast single server can be clogged up by a large job and hence give larger mean transfer times than the many slow servers, especially when service requirements are heavy-tailed [47]. However, web servers service requests in a processor sharing manner and server clogging does not occur.

9.3 Conclusions and Applications

In this chapter we have applied SHRiNK to web server farms. Experimental results show that the mean HTTP response time, the throughput, and the capacity of a farm, can be predicted by a small-scale replica. This replica may either run slower than the original farm, or consist of a smaller number of web servers.

SHRiNK is particularly appealing in the server farm case for practical reasons: It is very convenient to run a prototype farm, consisting of a small number of servers, next to an actual farm, and extrapolate performance from the prototype. Practitioners use such scaling techniques in an ad-hoc manner when they want to try a novel architecture or software in a large farm, but to the best of our knowledge there has been no systematic way to do this before SHRiNK. Notice that it is important to service the sampled requests from the prototype farm, not the original farm. This is so because users proceed to the next request after they receive the response to their previous request, and this feedback must depend on the algorithms and architectures tested on the prototype farm.

Chapter 10

Concluding Remarks

We conclude with an annotated summary of the results presented in this thesis, and some remarks on future directions of research both on practical and theoretical issues.

In Part I of the thesis, three methods have been introduced that improve web performance and efficiency. The first is a method to randomize any deterministic web cache replacement scheme, leading to reduced-complexity implementations at a small performance cost. The novelty of the method is a general procedure to improve the quality of a sample without increasing its size. This procedure exploits the iterative nature of replacement schemes by using past eviction samples to improve current eviction decisions. Another contribution of this work is an analysis of the tradeoff associated with using past samples. While this procedure is applied to web cache replacement policies, it is equally applicable to other iterative algorithms of interest in networking (e.g. load balancing [97], switch scheduling [104]). Thus, the theoretical methods used in this analysis may have wider applicability.

The second work is about a simple, accurate, and analytically tractable model for web request sequences. The model simultaneously captures all the important correlations in web traces. A direct benefit from the model is the ability to generate synthetic traces with the desired statistical characteristics. As an important by-product, the model provides guidelines for designing efficient replacement algorithms. It would be interesting both from a theoretical and practical perspective to use the model for designing prefetching algorithms.

The third work in the first part of the thesis is a scalable scheme that exploits the redundancy in dynamic web content. The scheme builds upon the technique of delta-encoding by grouping many dynamic documents into one class. By doing so, it significantly reduces the storage requirements associated with delta-encoding. The key idea is to exploit spatial correlation among different documents in addition to temporal correlation in a dynamically evolving document. This work was part of an industrial internship at FineGround Networks [35].

Part II of the thesis proposes a scalable, widely applicable method, called SHRiNK, for predicting the performance of large-scale, complex systems, from small-scale replicas. First, the method is applied to IP networks in which flows come and go at random times, and the size distribution of the flows is heavy-tailed. These networks are a plausible representation of the actual Internet. Our main finding is that SHRiNK is widely applicable for marginal and joint, steady-state and transient distributions of a large family of performance measures, for any network topology, transport protocol, and queue management mechanism. This result is validated via simulations and a theoretical argument.

SHRiNK reduces the complexity of the original network by slowing down its speed and sampling the input traffic. However, the network topology remains unchanged. Further investigation into scaling down the network topology might reveal new useful ways to scale a network and extrapolate performance.

The method is also applied to web server farms. Two ways of physically scaling a web farm are explored: reducing the speed of the servers of the farm, and reducing their number. Experimental results show that in both cases, performance measures like mean HTTP response time, farm throughput, and farm capacity can be predicted quite accurately by the physically scaled farm.

The web farm experimental results have been obtained from testbeds consisting of up to eight machines. More experimentation with larger scale testbeds, or directly with real farms, might address the limitations of this testbed and further solidify our findings.

SHRiNK is a technique that could be used in many other interesting areas such as wireless networks. Further investigations into these regimes might reveal more useful insights.

This thesis introduces novel probabilistic methods to deal with the increasing size and speed of operation of today's networks. The dissertation makes contributions in the areas of web caching and modeling, and performance prediction of IP networks and web farms. The hope is that the proposed methods are going to be used in practice to improve the performance of the Internet, and that their analysis will serve as a useful foundation for the future research in these exciting fields.

Bibliography

[1] M. Abrams, C.R. Standbridge, G. Abdulla, S. Williams, and E.A. Fox. Caching proxies: Limitations and potentials. In *Proceedings of WWW-4*, Boston, December 1995.

[2] The Apache web-server. `http://httpd.apache.org`, accessed January 2002.

[3] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating content management techniques for web proxy caches. In *Proceedings of the 2nd Workshop on Internet Server Performance*, May 1999.

[4] A.Tenenbaum and R. Nemes. Two spectra of self-organizing sequential search algorithms. *SIAM Journal of Computing*, 11(3), 1982.

[5] G. Banga, F. Douglis, and M. Rabinovich. Optimistic deltas for www latency reduction. In *Proceedings of USENIX Technical Conference*, Anaheim, CA, January 1997.

[6] P. Barford. *Modeling, Measurement and Performance of World Wide Web Transactions.* PhD thesis, Boston University, 2001.

[7] P. Barford and M. Crovella. Critical path analysis of TCP transactions. In *Proceedings of ACM SIGCOMM*, 2000.

[8] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the ACM SIGMETRICS Conference*, June 1998.

[9] E. Berlekamp. Factoring polynomials over large finite fields. *Mathematics of Computation*, 24:713–735, 1970.

[10] D. Bertsekas and R. Gallager. *Data Networks.* Prentice Hall, 1992.

[11] D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, 1996.

[12] B. Bollobas. *Randomized Graphs.* Academic Press, 1985.

[13] T. Bonalds, A. Prutiere, G. Gegnie, and J. Roberts. Insensitivity results in statistical bandwidth sharing. In *Proceedings of ITC17*, September, 2001.

[14] L. Breslau, P. Cao, L. Fan, G. Philips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of IEEE INFOCOM*, New York, 1999.

[15] M. Busari and C. Williamson. On the sensitivity of web proxy cache performance to workload characteristics. In *Proceedings of IEEE INFOCOM*, Anchorage, Alaska, April 2001.

[16] Web caching white paper. `http://www.cacheflow.com/technology/whitepapers/web.cfm`, accessed January 2001.

[17] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[18] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware*, September 1998.

[19] I. Caragiannis, C. Kaklamanis, and E. Papaioannou. Efficient on-line frequency allocation and call control in cellular networks. *Theory of Computing Systems*, 35:521–543, 2002.

[20] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, 1999.

[21] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the 19th ACM Symposium on Principles of Database Systems (PODS)*, 2000.

[22] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. *ACM Wireless Networks Journal*, 8(5), September 2002.

[23] Cisco. NetFlow services and applications. White paper, 2000. `http://cisco.com/warp/public/cc/pd/iosw/ioft/neflct/tech/napps_wp.htm`, accessed January 2002.

[24] Kimberly Claffy, George Polyzos, and Hans-Werner Braun. Applications of sampling methodologies to network traffic characterization. In *Proceedings of SIGCOMM*, 1993.

[25] E. Coffman and P. Denning. *Operating Systems Theory.* Prentice-Hall, 1973.

[26] C. Courcoubetis, G. Kesidis, A. Ridder, J. Walrand, and R. Weber. Admission control and routing in ATM networks using inferences from measured buffer occupancy. *IEEE Transactions on Communications*, pages 1778–1784, 1995.

[27] C.R. Cunba, A. Bestavros, and M.E. Crovella. Characteristics of www client-based traces. Technical Report BU-CS-96-010, Boston University, 1996.

[28] F. Douglis, A. Haro, and M. Rabinovich. Hpp: Html macro-preprocessing to support dynamic document caching. In *Proceedings of the 1st USENIX symposium on Internet Technologies & Systems, USITS*, December 1997.

[29] R. Durrett. *Probability: Theory and Examples.* Duxbury Press, 2nd edition, 1996.

[30] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of ACM SIGCOMM*, 2002.

[31] W. Fang and L. Peterson. Inter-as traffic patterns and their implications. In *Proceedings of the 4th Global Internet Symposium*, December 1999.

[32] A. Feldmann. Characteristics of TCP connection arrivals. In *Self-similar network traffic and performance evaluation*, K.Park and W.Willinger, editors, Wiley, 2000.

[33] A. Feldmann, A. C. Gilbert, and W. Willinger. Data networks as cascades: Investigating the multifractal nature of internet wan traffic. In *Proceedings of ACM SIGCOMM*, 1998.

[34] FineGround. Breaking new ground in content acceleration. White Paper. `http://www.fineground.com/prod/whitepaper.shtml`, accessed January 2001.

[35] Fineground networks. `http://www.fineground.com/`, accessed November 2002.

[36] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transaction on Networking*, pages 397–413, 1991.

[37] S. Floyd and E. Kohler. Internet research needs better models. In *Proceedings of ACM HOTNETS*, 2002.

[38] Fluid models for large, heterogeneous networks. `http://www-net.cs.umass.edu/fluid/`, accessed January 2002.

[39] Force 10 networks. `http://www.force10networks.com/`, accessed October 2002.

[40] C. Fraleigh, C. Diot, B. Lyles, S. Moon, P. Owezarski, D. Papagiannaki, and F. Tobagi. Design and deployment of a passive monitoring infrastructure. In *Proceedings of the Workshop on Passive and Active Measurements, PAM*, April 2001.

[41] C. Fraleigh, S. Moon, C. Diot, B. Lyles, and F. Tobagi. Packet-level traffic measurements from a tier-1 IP backbone. Technical Report TR01-ATL-110101, Sprint ATL Technical Report, November 2001.

[42] S. Ben Fredj, T. Bonalds, A. Prutiere, G. Gegnie, and J. Roberts. Statistical bandwidth sharing: a study of congestion at flow level. In *Proceedings of SIGCOMM*, 2001.

[43] P. Giaccone, B. Prabhakar, and D. Shah. Towards simple, high-performance schedulers for high-aggregate bandwidth switches. In *Proceedings of IEEE INFOCOM*, 2002.

[44] R. J. Gibbens and F. P. Kelly. Resource pricing and the evolution of congestion control. *Automatica*, 1999.

[45] S. Glassman. A caching relay for the world wide web. In *Proceedings of the 1th International World Wide Web Conference*, May 1994.

[46] P. Haas and L. Stokes. Estimating the number of classes in a finite population. Technical Report RJ 10025, IBM Almaden Research Center, San Jose, CA, 1996.

[47] M. Harchol-Balter, M. Crovella, and C. Murta. On choosing a task assignment policy for a distributed server system. *Journal of Parallel and Distributed Computing*, 59(2):204–228, November 1999.

[48] C.V. Hollot, V. Misra, D. Towlsey, and W. Gong. On designing improved controllers for AQM routers supporting TCP flow. In *Proceedings of INFOCOM*, 2001.

[49] V. Holmdahl, B. Smith, and T. Yang. Cooperative caching of dynamic content on a distributed web server. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, 1998.

[50] B. Housel and D. Lindquist. Webexpress: A system for optimizing web browsing in a wireless environment. In *Proceedings of the 2nd annual international conference on Mobile computing and networking, MOBICOM*, November 1996.

[51] J. Hunt, K.-P. Vo, and W. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2), April 1998.

[52] G. Iannaccone, C. Diot, C. Graham, and N. McKeown. Monitoring very high speed links. In *Proceedings of ACM Sigcomm Internet Measurement Workshop*, 2001.

[53] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In *Proceedings of the 1st USENIX symposium on Internet Technologies & Systems, USITS*, December 1997.

[54] S. Jin and A. Bestavros. Popularity-aware greedydual-size algorithm for web access. In *Proceedings of the IEEE ICDCS Conference*, April 2000.

[55] S. Jin and A. Bestavros. Greedydual* web caching algorithm: Exploiting the two sources of temporal locality in web request streams. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[56] R. Karp. An introduction to randomized algorithms. *Journal of Discrete Applied Mathematics*, 34:165–201, 1991.

[57] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.

[58] F. P. Kelly. *Reversibility and Stochastic Networks.* John Wiley & Sons, 1979.

[59] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice.* Addison Wesley, 2001.

[60] S. Kunniyur and R. Srikant. Analysis and design of an Adaptive Virtual Queue (AVQ) algorithm for active queue management. In *Proceedings of SIGCOMM*, 2001.

[61] D. Lee, J. Choi, S. Noh, S.L. Min, Y. Cho, and C.S. Kim. On the existence of a spectrum of policies that subsumes the lru and lfu policies. In *Proceedings of the ACM SIGMETRICS Conference*, May 1999.

[62] T. Lindvall. *Lectures on the Coupling Method.* Wiley Interscience, 1992.

[63] V. Misra, W. Gong, and D. Towsley. A fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED. In *Proceedings of SIGCOMM*, 2000.

[64] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefit of delta encoding and data compression for http. In *Proceedings of ACM SIGCOMM*, 1997.

[65] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[66] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice-Hall, 1993.

[67] A. Myers, J. Chuang, U. Hengartner, Y. Xie, W. Zhuang, and H. Zhang. A secure, publisher-centric web caching infrastructure. In *Proceedings of IEEE INFOCOM*, 2001.

[68] Myvitalagent. `http://www.qip.lucent.com/`, accessed January 2001.

[69] E. M. Nahum, M.-C. Rosu, S. Seshan, and J. Almeida. The effects of wide-area conditions on www server performance. In *Proceedings of ACM SIGMETRICS*, 2001.

[70] Nlanr cache access logs. `ftp://ircache.nlanr.net/Traces/`, accessed January 2002.

[71] J. Norris. *Markov Chains.* Cambridge University Press, 1997.

[72] Network simulator. `http:www.isi.edu/nsnam/ns`, accessed June 2000.

[73] C. J. Nuzman, I. Saniee, W. Sweldens, and A. Weiss. A compound model for tcp connection arrivals. In *Proceedings of ITC Seminar on IP Traffic Modeling*, Monterey, 2000.

[74] V. Padmanabhan and J. Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, 26(3), July 1996.

[75] V. Padmanabhan and L. Qiu. The content and access dynamics of a busy web site: Findings and implications. In *Proceedings of ACM SIGCOMM*, 2000.

[76] R. Pan. *Randomized Methods for Bandwidth Partitioning and Performance Prediction in the Internet*. PhD thesis, Stanford University, 2002.

[77] R. Pan, B. Prabhakar, and K. Psounis. Choke, a stateless active queue management scheme for approximating fair bandwidth allocation. In *Proceedings of IEEE INFOCOM*, 2000.

[78] R. Pan, B. Prabhakar, K. Psounis, and M. Sharma. A study of the applicability of a scaling hypothesis. In *Proceedings of ASCC*, 2002.

[79] R. Pan, B. Prabhakar, K. Psounis, and D. Wischik. Shrink: A method for scalable performance prediction and efficient network simulation. In *Proceedings of IEEE INFOCOM*, 2003.

[80] V. Paxson and S. Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[81] Peakstone. `http://www.peakstone.com/technology.html`, accessed January 2002.

[82] K. Psounis. Class-based delta encoding: A scalable scheme for caching dynamic web content. In *Proceedings of IEEE ICDCS Workshops*, Vienna, Austria, July 2002.

[83] K. Psounis, R. Pan, and B. Prabhakar. An approximate fair dropping scheme for variable length packets. In *Proceedings of Hot Interconnects 8*, Stanford, California, August 2000. Also appears (by invitation) at *IEEE Micro*, January/February 2001.

[84] K. Psounis, R. Pan, B. Prabhakar, and D. Wischik. The scaling hypothesis: Simplifying the prediction of network performance using scaled-down simulations. In *Proceedings of ACM HOTNETS*, 2002.

[85] K. Psounis and B. Prabhakar. A randomized web-cache replacement scheme. In *Proceedings of IEEE INFOCOM*, Anchorage, Alaska, April 2001.

[86] K. Psounis and B. Prabhakar. Efficient randomized web-cache replacement schemes using samples from past eviciton-times. *IEEE/ACM Transactions on Netwrorking*, 10(4), August 2002.

[87] K. Psounis, B. Prabhakar, and D. Engler. A randomized cache replacement scheme approximating lru. In *Proceedings of the 34th Annual Conference on Information Sciences and Systems*, Princeton University, March 2000.

[88] M. Puterman. *Markov Decision Processes, Discrete Stochastic Dynamic Programming.* Wiley Series in Probability and Mathematical Statistics, 1994.

[89] M. Rabing. *Probabilistic Algorithms, in J.F. Traub, editor, Algorithms and Complexity, Recent Results and New Directions, pages 21-39.* Academic Press, New York, 1976.

[90] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proceedings of IEEE INFOCOM*, 2002.

[91] R. Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, 19(2):63–67, February 1976.

[92] L. Rizzo and L. Vicisano. Replacement policies for a proxy cache. *IEEE/ACM Transactions On Networking*, 8(2), April 2000.

[93] S. Ross. *Introduction to Stochastic Dynamic Programming.* Academic Press, 1983.

[94] D. Rubenstein, J. Kurose, and D. Towsley. Detecting shared congestion of flows via end-to-end measurement. In *Proceedings of ACM SIGMETRICS*, 2000.

[95] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *Proceedings of ACM SIGCOMM*, 2000.

[96] D. Shah, P. Giaccone, and B. Prabhakar. An efficient randomized algorithm for input-queued switch scheduling. In *Proceedings of HOT Interconnects 9*, Stanford, CA, August 2001.

[97] D. Shah and B. Prabhakar. The use of memory in randomized load balancing. In *Proceedings of ISIT*, 2002.

[98] G. Shedler and C. Tung. Locality in page reference strings. *SIAM Journal of Computing*, 1(3), September 1976.

[99] A. Silberschatz and P. Galvin. *Operating System Concepts.* Addison Wesley Longman, Fifth Edition, 1997.

[100] B. Smith, A. Acharya, T. Yang, and H. Zhu. Exploiting result equivalence in caching dynamic web content. In *Proceedings of the 2nd USENIX symposium on Internet Technologies & Systems, USITS*, October 1999.

[101] R. Solovay and V. Strassen. A fast monte-carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, 1977.

[102] N. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, 2000.

[103] L. Tassiulas. Linear complexity algorithms for maximum throughput in radio networks and input queued switches. In *Proceedings of IEEE INFOCOM*, 1998.

[104] L. Tassiulas. Linear complexity algorithms for maximum throughput in radio networks and input queued switches. In *Proceedings of IEEE INFOCOM*, San Fransisco, CA, March 1998.

[105] J. Walrand. A transaction-level tool for predicting TCP performance and for network engineering. In *Proceedings of MASCOTS*, 2000.

[106] S. Williams, M. Abrams, C.R. Standbridge, G. Abdulla, and E.A. Fox. Removal policies in network caches for world wide web documents. In *Proceedings of ACM SIGCOMM*, Stanford University, August 1996.

[107] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variability: Statistical analysis of ethernet lan traffic at the source level. *IEEE/ACM Transactions on Networking*, 5(1):71–86, 1997.

[108] R. W. Wolff. *Stochastic Networks and the Theory of Queues.* Prentice Hall, 1989.

[109] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative web proxy caching. In *Proceedings of SOSP*, 1999.

[110] R. Wooster and M. Abrams. Proxy caching that estimates edge load delays. In *Proceedings of the 6th International World Wide Web Conference*, Santa Clara, CA, April 1997.

[111] Z. Xiao and K. Birman. A randomized error recovery algorithm for reliable multicast. In *Proceedings of IEEE INFOCOM*, 2001.

[112] A. Yao. Should tables be sorted. *Journal of the ACM*, 28(3):615–628, 1981.

[113] M. Zhang, R. Wang, L. Peterson, and A. Krishnamurthy. Probabilistic packet scheduling: Achieving proportional share bandwidth allocation for TCP flows. In *Proceedings of IEEE INFOCOM*, 2002.

[114] H. Zhu and T. Yang. Class-based cache management for dynamic web content. In *Proceedings of IEEE INFOCOM*, 2001.