THE DESIGN AND EVALUATION OF WEB PREFETCHING AND CACHING TECHNIQUES

BY BRIAN DOUGLAS DAVISON

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Haym Hirsh

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2002

ABSTRACT OF THE DISSERTATION

The Design and Evaluation of Web Prefetching and Caching Techniques

by Brian Douglas Davison

Dissertation Director: Haym Hirsh

User-perceived retrieval latencies in the World Wide Web can be improved by preloading a local cache with resources likely to be accessed. A user requesting content that can be served by the cache is able to avoid the delays inherent in the Web, such as congested networks and slow servers. The difficulty, then, is to determine what content to prefetch into the cache.

This work explores machine learning algorithms for user sequence prediction, both in general and specifically for sequences of Web requests. We also consider information retrieval techniques to allow the use of the content of Web pages to help predict future requests. Although history-based mechanisms can provide strong performance in predicting future requests, performance can be improved by including predictions from additional sources.

While past researchers have used a variety of techniques for evaluating caching algorithms and systems, most of those methods were not applicable to the evaluation of prefetching algorithms or systems. Therefore, two new mechanisms for evaluation are introduced. The first is a detailed trace-based simulator, built from scratch, that estimates client-side response times in a simulated network of clients, caches, and Web servers with various connectivity. This simulator is then used to evaluate various

prefetching approaches. The second evaluation method presented is a novel architecture to simultaneously evaluate multiple proxy caches in a live network, which we introduce, implement, and demonstrate through experiments. The simulator is appropriate for evaluation of algorithms and research ideas, while simultaneous proxy evaluation is ideally suited to implemented systems.

We also consider the present and the future of Web prefetching, finding that changes to the HTTP standard will be required in order for Web prefetching to become commonplace.

Acknowledgements

Here I acknowledge the time, money, and support provided to me. Officially, I have been financially supported by a number of organizations while preparing this dissertation. The most recent and significant support has been through the National Science Foundation under NSF grant ANI 9903052. In addition, I was supported earlier by a Rutgers University special allocation to strategic initiatives in the Information Sciences, and by DARPA under Order Number F425 (via ONR contract N6600197C8534).

I should point out that much of the content of this dissertation has been published separately. Parts of Chapter 1 were included in a WWW workshop position statement [Dav99a]. A version of Chapter 2 has been published in *IEEE Internet Computing* [Dav01c]. Part of Chapter 3 (written with Haym Hirsh) was presented in a AAAI/ICML workshop paper [DH98]. Much of Chapter 5 was presented at a SIGIR conference [Dav00c]. A version of Chapter 6 was presented in a Hypertext conference [Dav02a]. Part of Chapter 7 was presented at WCW [Dav99e]. Part of Chapter 8 was presented at MASCOTS [Dav01b]. A version of Chapter 10 was presented at WCW [Dav99d]. A portion of Chapter 11 (written with Chandrasekar Krishnan and Baoning Wu) was presented at WCW [DKW02]. Much of Chapter 12 was presented at WCW [Dav01a].

Much of this dissertation is based on experiments using real-world data sets. I thank those who have made such Web traces available, including Martin Arlitt, Laura Bottomley, Owen C. Davison, Jim Dumoulin, Steven D. Gribble, Mike Perkowitz, and Carey Williamson. This work would not have been possible without their generosity.

I'd like to thank my advisor Haym Hirsh (who encouraged me to choose my own path, and said it was time to interview, and thus to finish the dissertation), and committee members Ricardo Bianchini, Thu Nguyen, and Craig Wills.

I've also appreciated the advice, support, and contributions of collaborators and

reviewers, including, but not limited to: Fred Douglis, Apostolos Gerasoulis, Paul Kantor, Kiran Komaravolu, Chandrasekar Krishnan, Vincenzo Liberatore, Craig Nevill-Manning, Mark Nottingham, Weisong Shi, Brett Vickers, Gary Weiss, and Baoning Wu.

The years at Rutgers would not have been nearly as much fun without the other current and former student members of the Rutgers Machine Learning Research Group: Arunava Banerjee, Chumki Basu, Daniel Kudenko, David Loewenstern, Sofus Macskassy, Chris Mesterharm, Khaled Rasheed, Gary Weiss, and Sarah Zelikovitz. Thank you for listening to me, and letting me listen to you.

Finally, my love and gratitude are extended to my wife, Karen, and our family, both immediate and extended by blood or marriage. Without their love, understanding, and support, I would have had to get a real job years ago and this work would never have been finished...

Dedication

This work is dedicated to all those who use the Web daily.

May it only get better and better.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACK ..... Acknowledgement

CDF ..... Cumulative Distribution Function

CDN ..... Content Delivery Network

CPU ..... Central Processing Unit

DNS ..... Domain Name System

HTML ..... HyperText Markup Language

HTTP ..... HyperText Transport Protocol

IOLA ..... Ideal Online Learning Algorithm

IP ..... Internet Protocol

IPAM ..... Incremental Probabilistic Action Prediction

ISP ..... Internet Service Provider

NCS ..... Network and Cache Simulator

PDF ..... Portable Document Format

PPM ..... Prediction by Partial Match

RAM ..... Random Access Memory

RFC ..... Request For Comments

RTT ..... Round Trip Time

SPE ..... Simultaneous Proxy Evaluation

TCP ..... Transport Control Protocol

UDP ..... User Datagram Protocol

URL ..... Uniform Resource Locator

WWW ..... World-Wide Web

Chapter 1

Introduction

1.1 Motivation

Introduced a little over a decade ago, the World-Wide Web has transformed much of the world's economy and will continue to do so. It can be compared to the telephone, television, and automobile as a paradigm shift in the way people work, communicate, and play. As the popular interface to what has been called the "information superhighway", the Web purports to provide instantaneous access to vast quantities of information. However, the perception of time is always relative — access to information on today's Web is rarely instantaneous. While performance continues to improve over time from improvements in bandwidth and device latencies, users continue to desire yet additional speed [KPS+99]. Likewise, content providers continue to make greater demands on bandwidth as it increases.

Good interactive response-time has long been known to be essential for user satisfaction and productivity [DT82, Bra86, Roa98]. This is also true for the Web [RBP98, BBK00]. A widely-cited study from Zona Research [Zon99] provides evidence for the "eight second rule" — in electronic commerce, if a Web site takes more than eight seconds to load, the user is much more likely to become frustrated and leave the site. Thus there is also significant economic incentive for many content providers to provide a responsive Web experience.

1.1.1 Methods to improve Web response times

Many factors contribute to a less-than-speedy Web experience, including heterogeneous network connectivity, real-world distances, and congestion in networks or servers due

to unexpected demand. As a result, many researchers (often as entrepreneurs) have considered the problem of improving Web response times. Some want to improve performance by achieving bandwidth increases and response time improvement through bigger "pipes" or alternative communication technologies. Others want to use the existing infrastructure more efficiently. Web caching, along with other forms of data dissemination, has been proposed as a technology that helps reduce network usage and server loads and improve typical response times experienced by the user. When successful, pre-loading Web objects into local caches can be used to further reduce web response times [PM96], and even to shift network loads from peak to non-peak periods [MRGM99]. Our interest is in pre-loading interactively, so that by dynamically pre-loading Web objects likely to be of interest into local caches, we may invisibly improve the user experience by improving the response time.

To be more precise, let us start with a definition:

> *Pre-loading* is the speculative installation of data in a cache in the anticipation that it will be needed in the future. (1)

On the Web, pre-loading typically implies the transmission of data over a network, and can be performed anywhere that a cache is present, including client browsers and caching proxies, but the definition equally applies to cache pre-loading in disk caches, CPU caches, or back-end Web server caches where the source of the data is some other system (such as a database). We use *prefetching* to distinguish the more specific form of pre-loading in which the system holding the cache initiates the retrieval of its own volition. Thus:

> *Prefetching* is the (cache-initiated) speculative retrieval of a resource into a cache in the anticipation that it can be served from cache in the future. (2)

Pre-loading, then, is broader, encompassing both prefetching and *prepushing*, in which content is pushed from server to cache. While this dissertation will focus on prefetching, it will utilize models in which the server sends *hints* specifying the server's recommendation on what should be prefetched.

1.1.2 User action prediction for the Web

Most requests on the Web are made on behalf of human users, and like other human-computer interactions, the actions of the user can be characterized as having identifiable regularities. Much of these patterns of activity, both within a user, and between users, can be identified and exploited by intelligent action prediction mechanisms. Prediction here is different from what data mining approaches do with Web logs. Our user modeling attempts to build a (relatively) concise model of the user so as to be able to dynamically predict the next action(s) that the user will take. Data mining, in contrast, is typically concerned with characterizing the user, finding common attributes of classes of users, and predicting future actions (such as purchases) without the concern for interactivity or immediate benefit (e.g., see the KDD-Cup 2000 competition [BK00]).

Therefore one research focus is to apply machine learning techniques to the problem of user action prediction on the Web. In particular, we wish to be able to predict the next Web page that a user will select. If one were able to build such a user model, a system using it could anticipate each page retrieval and fetch that page ahead of time into a local cache so that the user experiences short response times. In this dissertation, we will demonstrate the use of machine learning models on real-world traces with predictive accuracies of 12-50% or better, depending on the trace.

Naturally, pre-loading objects into a cache is not a new concept, and has already been incorporated into a few proxy caches and into a number of browser extensions (see our Web site on Web caching [Dav02b] for pointers to caching products and browser extensions). Thus we concentrate on the incorporation of a variety of sources of information for prediction, and the principled evaluation and comparison of such systems. We believe that multiple sources are necessary in order to incorporate desired characteristics into the system. Such sources of information would certainly include client history so that an individual's pattern of usage would serve as a strong guide. But we would also want to include community usage patterns (from proxy and origin servers) so that typical usage patterns may be used as intelligent defaults for points in which there is no individual history. Context is also important when history is not relevant

— we use the textual contents of recent pages as a guide to the current interests of the user and the link contents of those pages as significant influences to what may be chosen next. Finally, we recognize that the contents of related applications (such as Usenet news and electronic mail) also present URLs that can be chosen as pages to be retrieved, but leave that as future work to others (e.g., [HT01]). Our conjecture is that the appropriate combination of information from sources such as these will make more accurate predictions possible via a better user model, and thus reduce the amount of extra bandwidth required to generate adequate improvements in response times.

However, the evaluation of such models in terms of response time improvements requires the incorporation of real-world considerations such as network characteristics and content caching. Through simulation, we will model network connections with latencies and bandwidth limitations, thus limiting the transmission speed of content, or delaying its transmission when the simulated network is already at capacity. Embedding prediction mechanisms to perform prefetching within this environment will allow for more accurate evaluation of the efficacy of the prediction techniques. Simulation experiments in this dissertation will show the potential for prefetching on server traces to cut median response times in half or better, and to reduce the sum of all response times by 15-20%, by transmitting an additional 80-85% bytes over what would be transmitted by the equivalent caching-only configuration.

1.1.3 Questions and answers

In general, the quest to pre-load Web content has generated a number of questions:

- How can future Web activity be predicted?
- What methods can be used for pre-loading Web content?
- Can better predictions be made by combining the results of separate prediction systems?

- Does the accuracy of a prediction algorithm reflect its utility in a cache pre-loading environment?
- How can the performance improvements of caching and prefetching techniques be estimated?
- How are implemented caching systems evaluated?
- How can Web cache prefetching systems be evaluated?
- What safety and integrity concerns are there for prefetching systems?
- Why are pre-loading systems relatively uncommon on today's Web?

The answers to these questions, and others, to various degrees, can be found in this dissertation. The search for answers has led to investigation into a variety of areas, including machine learning, simulation, networking, and information retrieval.

What we have found is that while many researchers have proposed various methods for prediction and pre-loading, few are able to accurately evaluate such systems in terms of client-side response times. Generally one may choose from three general approaches to studying system performance: analytic modeling, simulation, and direct measurement. Each provides unique insights into the problem. Analytic approaches provide tools to model systems and scenarios to find trends and limits. Simulation allows for the rapid testing of a variety of algorithms without causing undue harm on the real world. Direct measurements provide grounding in reality with "existence proofs" and challenges for explanations. To realistically consider response times in the Web, however, strict analytic models become unmanageably complex, and thus we have concentrated our efforts on the latter two.

The primary focus of this work has been the design and development of two tools for evaluation of such techniques: a simulator for testing caching and prefetching algorithms, and an evaluation architecture for the testing of implemented proxy caches. Thus, in addition to explorations and surveys of prediction techniques, the majority of the dissertation will propose, implement, validate, and give examples of the use of each tool.

1.2 Contributions

Although there has been significant attention paid to pre-loading from the research community in the time since this thesis was conceived, this dissertation makes a number of contributions around the two themes of this thesis:

- Prediction and Prefetching Methods for the Web

    1. We identify and enumerate key aspects of idealized online learning algorithms.
    2. We propose and demonstrate the utility of a simple approach to user action prediction (realizing an accuracy of approximately 40%) that allows the user's profile to change over time.
    3. We implement a parameterized history-based prediction method, and with it evaluate a space of Markov-like prediction schemes (achieving accuracies of 12-50% or better, depending on the dataset).
    4. We explore the potential for content-based prediction by studying aspects of the Web, and evaluate proposed content-based methods on a full-content Web usage trace. We find a content-based approach to be 29% better than random link selection for prediction, and 40% better than not prefetching in a system with an infinite cache.
    5. We consider the performance improvements possible when combining predictions from multiple sources, and find that combining disparate sources can produce significantly better accuracy than either source alone.
    6. We identify deficiencies in HTTP for prefetching and propose corrective extensions to HTTP.

- Accurate Evaluation

    1. We collect a small, full-content user workload for off-line analysis, since typical usage logs are insufficient.

2. We survey evaluation mechanisms for Web cache pre-loading and reveal problems in them.
3. We identify the need to include network effects in simulations to accurately estimate client-side response times.
4. We implement and validate a new proxy caching and network effects simulator to estimate Web response times at the client. Using it, we find that client prefetching based on Web server predictions can significantly reduce typical Web response times, without excessive bandwidth overhead.
5. We propose and implement a novel black-box proxy evaluation architecture that is specifically capable of evaluating prefetching proxies.
6. We perform experiments using the evaluation architecture to validate the implementation and to test multiple proxies with both artificial and real-world workloads.

1.3 Dissertation Outline

Immediately following this chapter is a primer on Web caching to provide a minimal background and to motivate the rest of the dissertation. The remaining chapters are divided into two parts.

Initially we consider the task of action prediction. The approaches taken for action prediction can vary considerably, and often have complex implementations. In many cases, however, simple methods can do well, and so in Chapter 3 we first consider a relatively unsophisticated approach to the similar problem of using past history to predict user actions within the application domain of UNIX shell commands. We then examine in Chapter 4 the particular problem of prediction of requests for Web objects. Various Markov-like prediction models are detailed, evaluated, and compared under multiple definitions of performance.

These initial chapters also deal with the complementary idea of content-based prediction. Here we consider the contents of Web pages to provide hints and recommendations for future requests. We first explore the potential of content-based mechanisms

by measuring characteristics of the Web, including topical locality and descriptive quality of page proxies in Chapter 5. Having established the potential for content-based prediction, we then consider in Chapter 6 one approach to using the pages that a user visits as a guide to the next page the user will request, and evaluate the approach on a small full-content Web trace.

In later chapters of the dissertation, we deal head-on with more real-world issues, including careful evaluation of caching and prefetching systems. Thus in Chapter 7 we introduce these concerns, and survey proxy cache evaluation techniques. In Chapter 8 we develop and describe an advanced network and cache simulator, and describe efforts to validate this simulator by comparing it to real-world measurements of Web traffic. Chapter 9 then uses the simulator in experiments to incorporate predictions from multiple sources for prefetching. We additionally systematically explore various parameter settings for simulations using two Web server traces to ascertain better performing prefetching configurations.

In contrast to simulation, Chapter 10 introduces a novel architecture for the simultaneous evaluation of multiple *real-world* black-box proxy caches. This architecture is then implemented, tested, and used to evaluate proxy caches in Chapter 11. Chapter 12 discusses the need to change HTTP because of problems with the current HTTP specification and its interpretation. Finally, Chapter 13 wraps up the dissertation by looking ahead to the future of Web caching, and the steps needed to make prefetching widespread.

Chapter 2

A Web Caching Primer

2.1 Introduction

At this time educated readers are familiar with the spectacular growth of the Internet and with the World Wide Web. More content provides incentive for more people to go online, which provides incentive to produce more content. This growth, both in usage and content, has introduced significant challenges to the scalability of the Web as an effective means to access popular content.

When the Web was new, a single entity could (and did) list and index all of the Web pages available, and searching was just an application of the Unix egrep command over an index of 110,000 documents [McB94]. Today, even though the larger search engines index billions of documents, any one engine is likely to see only a fraction of the content available to users [LG99].

Likewise, when the Web was new, page retrieval was not a significant issue — the important thing was that it worked! Now with the widespread commercialization of the Web, exceeding the 'eight-second rule' for download times may mean the loss of significant revenue [Zon99] as increasing numbers of users will go on to a different site if they are unsatisfied with the performance of the current one.

Finally, as increasing use of the Web has necessitated faster and more expensive connections to the Internet, the concern for efficient use of those connections has similarly increased. Why should an organization (either a net consumer or producer of content) pay for more bandwidth than is necessary?

This chapter provides a primer on one technology used to make the Web scalable: Web resource caching. Web caching can reduce bandwidth usage, decrease user-perceived latencies, and reduce Web server loads. Moreover, this is generally accomplished in a manner that is transparent to the end user. As a result, caching has become a significant part of the infrastructure of the Web, and thus represents a strong commercial market (more than $2B in caching sales expected in 2003 [Int99]). Caching has even spawned a new industry: *content delivery networks*, which are likewise growing at a fantastic rate (expecting over $3B in sales and services by 2006 [CK02]). In subsequent sections we will introduce caching, discuss how it differs from traditional caching applications, describe where Web caching is used, and when and why it is useful.

Those readers familiar with relatively advanced Web caching topics such as the Internet Cache Protocol (ICP) [WC97b], invalidation, or interception proxies are not likely to learn much here. Instead, this chapter is designed for a more general audience that is familiar only as a user of the Web. Likewise, this is not a how-to guide to deploy caching technologies, but rather argues at a higher level for the value of caching on the Web to both consumers and producers of content. Furthermore, while this chapter will provide references to many other published papers, it is not intended as a survey. Instead, the reader may wish to examine some surveys [Wan99, BO00, Mog00] for a more detailed treatment of current Web caching trends and techniques.

2.2 Caching

A simple example of caching is found in the use of an address book that one keeps close at hand. By keeping oft-used information nearby, one can save time and effort. While the desired phone number may be listed in a phone book, it takes a longer and may first require the retrieval of the phone book from across the room. This is the general principle used by caching systems in many contexts.

2.2.1 Caching in memory systems

A long-standing use of caching is in memory architectures [Smi82, HP87, Man82, MH99]. The central processing unit (CPU) of modern computers operate at very high speeds. Memory systems, typically are unable to work at the same speeds, and so when the CPU needs to access information in memory, it must wait for the slower memory to provide the required data. If a CPU always had to slow down for memory access, it would operate at the effective speed of a much slower CPU. To help minimize such slowdowns, the CPU designer provides one or more levels of cache — a small amount of (expensive) memory that operates at, or close to, the speed of the CPU. Thus, when the information that the CPU needs is in the cache, it doesn't have to slow down. When the cache contains the requested object, this is called a *hit*. When the requested object cannot be found in the cache, there is a cache *miss*, and so the object must be fetched directly (incurring any requisite costs for doing so).

Typically, when a cache miss occurs, the object retrieved is then placed in the cache so that the next time the object is needed, it can be accessed quickly. An assumption of *temporal locality* — the idea that a recently requested object is more likely than others to be needed in the future — is at work here. A second kind of locality, *spatial locality*, in which nearby objects are likely to be needed, is also typically assumed. For this reason (and to take advantage of other efficiencies), memory systems retrieve multiple consecutive memory addresses and place them in the cache in a single operation, rather than just one object at a time.

Regardless of how objects get placed in a cache, at some point, the cache will become full. Some algorithm will be used to remove items from the cache to make room for new objects. Various methods have been proposed and implemented, including variations of simple ideas like first-in/first-out (FIFO), least recently used (LRU), or least frequently used (LFU). When selecting a *replacement algorithm* such as one of these, the goal is to optimize performance of the cache (e.g., maximizing the likelihood of a cache hit for typical memory architectures).

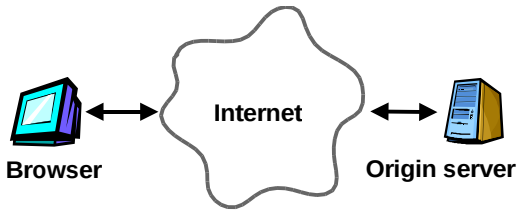


Figure 2.1: A simplistic view of the Web in which the Web consists of a set of Web servers and clients and the Internet that connects them.

2.2.2 Mechanics of a Web request

Before getting into caching on the Web, we first need to address how the World-Wide Web works. In its simplest form, the Web can be thought of as a set of Web clients (such as Web browsers, or any other software used to make a request of a Web server) and Web servers. In order to retrieve a particular Web resource, the client attempts to communicate over the Internet to the *origin Web server* (as depicted in Figure 2.1). Given a URL such as `http://www.web-caching.com/`, a client can retrieve the content (the home page) by establishing a connection with the server and making a request. However, in order to connect to the server, the client needs the numerical identifier for the host. It queries the domain name system (DNS) to translate the hostname (`www.web-caching.com`) to its Internet Protocol (IP) address (`209.182.1.122`). Once it has the IP address, the client can establish a connection to the server. Once the Web server has received and examined the client's request, it can generate and transmit the response. Each request and response include headers (such as shown in Figure 2.2) and possibly content (e.g., the HTML of the page requested, or the data for a requested image). Each step in this process takes some time (see Figure 2.3 for estimates of the time cost to establish a connection, request content, and receive the content).

The HyperText Transfer Protocol (HTTP) specifies the interaction between Web clients, servers, and intermediaries. The current version is HTTP/1.1 [FGM+99]. Requests and responses are encoded in headers that precede an optional body containing content. Figure 2.2 shows one set of request and response headers. The initial line of the request shows the method used (GET), the resource requested ("/"), and the version

Request Headers:

```
GET / HTTP/1.1
Host: www.web-caching.com
Referer: http://vancouver-webpages.com/CacheNow/
User-Agent: Mozilla/4.04 [en] (X11; I; SunOS 5.5.1 sun4u)
Accept: */*
Connection: close
```

Response Headers:

```
HTTP/1.1 200 OK
Date: Mon, 18 Dec 2000 21:25:23 GMT
Server: Apache/1.3.12 (Unix) mod_perl/1.18 PHP/4.0B2
Cache-Control: max-age=86400
Expires: Tue, 19 Dec 2000 21:25:23 GMT
Last-Modified: Mon, 18 Dec 2000 14:54:21 GMT
ETag: "838b2-4147-3a3e251d"
Accept-Ranges: bytes
Content-Length: 16711
Connection: close
Content-Type: text/html
```

Figure 2.2: Sample HTTP request and response headers. Headers identify client and server capabilities as well as describe the response content.

of HTTP supported (1.1). GET is just one of the methods defined for HTTP; another commonly used method is POST, which allows for content to be sent with the request (e.g., to carry variables from an HTML form). Likewise, the first line of the response headers shows the HTTP version supported, and a response code with standard values (e.g., 200 is OK, 404 is Not Found, etc.).

The headers of an HTTP transaction also specify aspects relevant to the cacheability of an object. The headers from the example in Figure 2.2 that are meaningful for caching include `Date`, `Last-Modified`, `ETag`, `Cache-Control`, and `Expires`. For example, in HTTP GET requests that include an `If-Modified-Since` header, Web servers use the `Last-Modified` date on the current content to return the object only if the object had changed after the date of the cached copy. An accurate clock is essential for the origin server to be able to calculate and present modification times as well as expiration times in the other tags. An `ETag` (entity tag) represents a signature for the object and allows for a stronger test than `If-Modified-Since` — if the signature of the current object at this URL matches the cached one, the objects are considered equivalent. The `Expires`

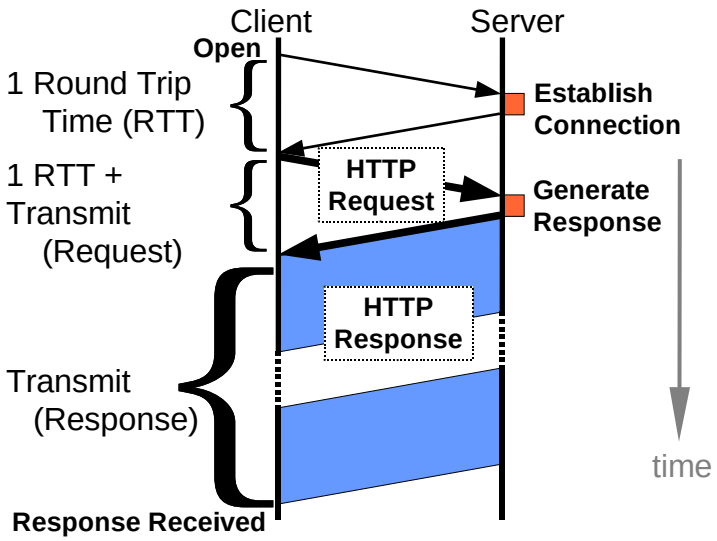


Figure 2.3: HTTP transfer timing cost for a new connection. The amount of time to retrieve a resource when a new connection is required can be approximated by two round-trip times plus the time to transmit the response (plus DNS resolution delays, if necessary).

and `Cache-Control: max-age` headers specify how long the object can be considered valid. For slowly or never changing resources, setting an explicit expiration date is the way to tell caches how long they can keep the object (without requiring the cache to contact the origin server to validate it).

2.2.3 Web caching

Now that we have discussed the basics of caching (from memory systems) and seen some of the details of an individual Web request and response, we can introduce some of the characteristics of caching on the Web and then discuss why caching is successful. Caching in the Web works in a similar manner to that of caching in memory systems — a Web cache stores Web resources that are expected to be requested again. However, it also differs in a number of significant aspects. Unlike the caching performed in many other domains, the objects stored by Web caches vary in size. As a result, cache operators and designers track both the overall *object hit rate* (percentage of requests

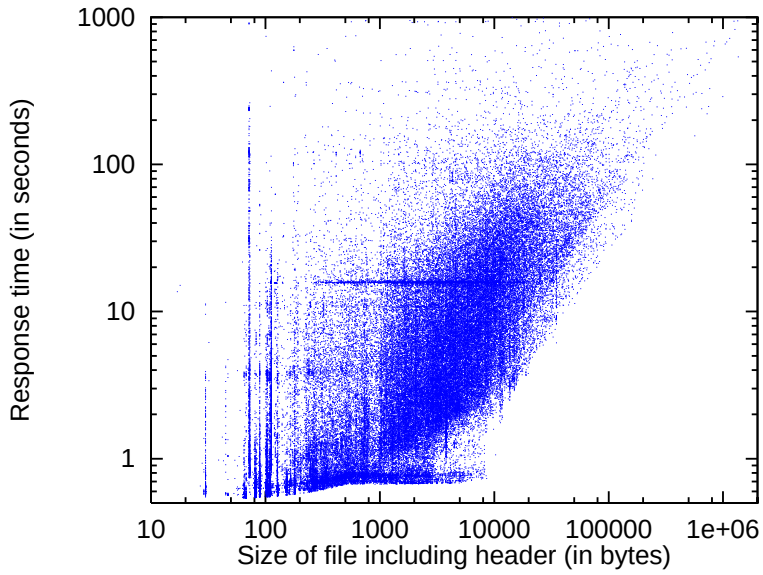


Figure 2.4: In this scatter-plot of real-world responses (the first 100,000 from the UC Berkeley Home IP HTTP Trace of dialup and wireless modem users [Gri97]), the response time varies significantly, even for resources of a single size.

served from cache) as well as the overall *byte hit rate* (percentage of bytes served from cache). Traditional replacement algorithms often assume the use of a fixed object size, so variable sizes may affect their performance. Additionally, non-uniform object size is one contributor to a second difference, that of non-uniform retrieval costs. Web resources that are larger are likely to have a higher cost (that is, higher response times) for retrieval, but even for equally-sized resources, the retrieval cost may vary as a result of distance traveled, network congestion, or server load (see Figure 2.4 for one example of real usage in which retrieval times for the same size object vary over multiple orders of magnitude). Finally, some Web resources cannot or should not be cached, e.g., because the resource is personalized to a particular client or is constantly updated (as in *dynamic pages*, discussed further in Section 2.2.8).

2.2.4 Where is Web caching used?

Caching is performed in various locations throughout the Web, including the two endpoints about which a typical user is aware. Figure 2.5 shows one possible chain of caches through which a request and response may flow. The internal nodes of this chain are

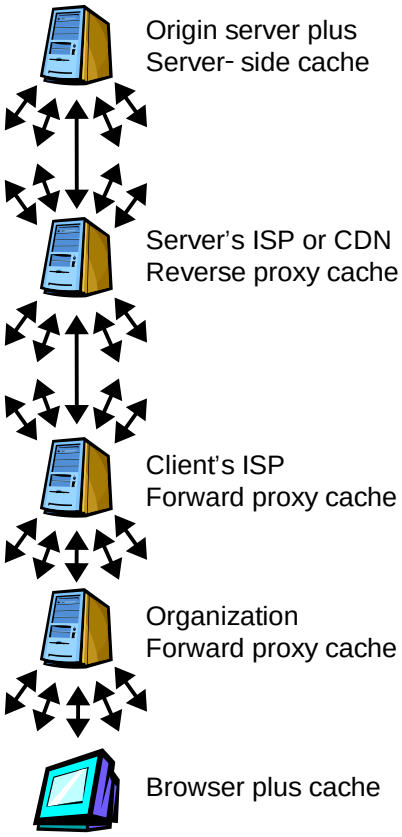


Figure 2.5: Caches in the World Wide Web. Starting in the browser, a Web request may potentially travel through multiple caching systems on its way to the origin server. At any point in the sequence a response may be served if the request matches a valid response in the cache.

termed *proxies*, since they will generate a new request on behalf of the user if the proxy cannot satisfy the request itself. A user request may be satisfied by a response captured in a browser cache, but if not, it may be passed to a department– or organization–wide proxy cache. If a valid response is not present there, the request may be received by a proxy cache operated by the client's ISP. If the ISP cache does not contain the requested response, it will likely attempt to contact the origin server. However, *reverse proxy caches* operated by the content provider's ISP or content delivery network (CDN) may instead respond to the request. If they do not have the requested information, the request may ultimately arrive at the origin server. Even at the origin server, content,

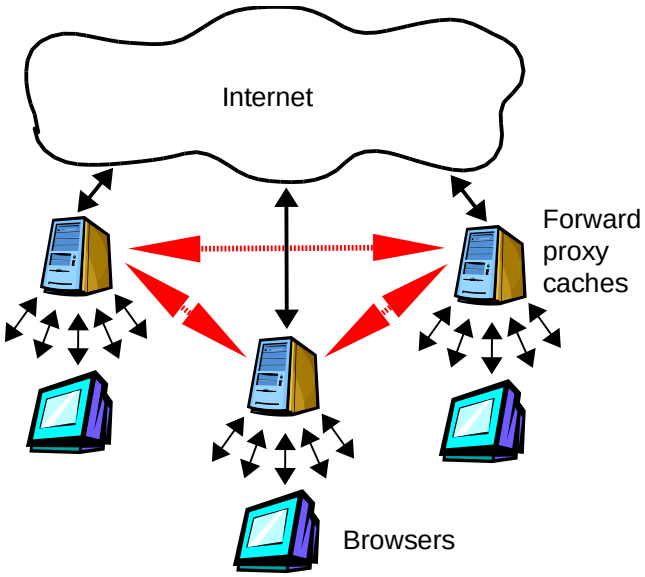


Figure 2.6: Cooperative caching in which caches communicate with peers before making a request over the Web.

or portions of content can be stored in a *server-side cache* so that the server load can be reduced (e.g., by reducing the need for redundant computations or database retrievals).

Wherever the response is generated, it will likewise flow through the reverse path back to the client. Each step in Figure 2.5 has multiple arrows, signifying relationships with multiple entities at that level. For example, the reverse proxy (sometimes called an *http accelerator*) operated by the content provider's ISP or CDN can serve as a proxy cache for the content from multiple origin servers, and can receive requests from multiple downstream clients (including forward caches operated by others, as shown in the diagram).

In general, a cache need not talk only to the clients below it and the server above it. In fact, in order to scale to large numbers of clients it may be necessary to consider using multiple caches. A hierarchical caching structure [CDN+96] is common, and is partially illustrated in Figure 2.5. In this form, each cache serves many clients, which may be users or other caches. When the request is not served by a local cache, the request is passed on to a higher level in the hierarchy, until reaching a root cache, which, having no parent, would request the object from the origin server.

An alternative approach is to use a cooperative caching architecture (see Figure 2.6), in which caches communicate with each other as peers (e.g., using an inter-cache protocol such as ICP [WC97b]). In this form, on a miss, a cache would ask a predetermined set of peers whether they already have the missing object. If so, the request would be routed to the first responding cache. Otherwise, the cache attempts to retrieve the object directly from the origin server. This approach can prevent storage of multiple copies and reduce origin server retrievals, but pays a penalty in the form of increased inter-cache communication. Instead of asking peers directly about each request, another variation periodically gives peers a summary of the contents of the cache [FCAB98, RW98], which can significantly reduce communication overhead.

Variations and combinations of these approaches have been proposed; current thinking [WVS+99] limits cooperative caching to smaller client populations, but both cooperative caching and combinations are in use in real-world sites, particularly where network access is very expensive.

Technically, a client may or may not know about intermediate proxy caches. If the client is configured directly to use a proxy cache, it will send to the proxy all requests not satisfied by a built-in cache. Otherwise, the client has to look up the IP address of the origin host. If the content provider is using a CDN, the DNS servers may be customized to return the IP address of the server (or proxy cache) closest to the client (where closest likely means not only network distance, but may also take into consideration additional information to avoid overloaded servers or networks). In this way, a reverse proxy server can operate as if it were the origin server, answering immediately any cached requests, and forwarding the rest.

Even when the client has the IP address of the server that it should contact, it might not end up doing so. Along the network path between the client and the server, there may be a network switch or router that will 'transparently' direct all Web requests to a proxy cache. This approach may be utilized at any of the proxy cache locations shown in Figure 2.5. Under this scenario, the client believes it has contacted the origin server, but instead has reached an *interception proxy* which will serve the content either from cache, or by first fetching it from the origin server.

2.2.5 Utility of Web caching

Web caching has three primary benefits. The first is that it can reduce network bandwidth usage, which can save money for both clients (consumers of content) and servers (content creators) through more efficient usage of network resources. The second is that it can reduce user-perceived delays. This is important as the longer it takes to retrieve content, the lower the value as perceived by the user [RBP98, BBK00]. Finally, caching can also reduce loads on the origin servers. While this effect may not seem significant to end users, it can allow significant savings in hardware and support costs for content providers as well as a shorter response time for non-cached resources.

The combination of these features makes caching technology attractive to all Web participants, including end-users, network managers, and content creators. There are, of course, some potential drawbacks, which we will explore shortly. First, we need to discuss how caching achieves the benefits we have described.

2.2.6 How caching saves time and bandwidth

When a request is satisfied by a cache (whether it is a browser cache, or a proxy cache run by your organization or ISP, or an edge proxy operated by a CDN), the content no longer has to travel across the Internet from the origin Web server to the cache, saving bandwidth for the cache owner as well as the origin server.

TCP, the network protocol used by HTTP, has a fairly high overhead for connection establishment and sends data slowly at first. This, combined with the knowledge that most requests on the Web are for relatively small resources, means that reducing the number of connections that are necessary and holding them open (making them persistent) so that future requests can use them, has a significant positive effect on client performance [PM94, NGBS+97]. In particular, for clients of forward proxies, time can be saved; the client can retain a persistent connection to the proxy, instead of taking the time to establish new connections with each origin server that a user might visit during a session. Use of persistent connections is particularly beneficial to clients suffering from high-latency network service such as via dial-up modems [CDF+98, FCD+99].

Furthermore, busy proxies may be able to use persistent connections to send requests for multiple clients to the same server, reducing connection establishment times to servers as well. Therefore, a proxy cache supporting persistent connections can cache connections on both client and server sides (avoiding the initial round trip time for a new HTTP connection shown in Figure 2.3).

Caching on the Web works because of various forms of temporal and spatial locality — people request popular objects from popular servers. In one study spanning more than a month, out of all the objects requested by individual users, close to 60% of those objects on average were requested more than once by the same user [TG97]. So, individual users request distinct Web objects repeatedly. Likewise, there is plenty of content that is of value to more than one user. In fact, of the hits provided in another caching study, up to 85% were the result of multiple users requesting the same objects [DMF97].

In summary, Web caching works because of popularity — the more popular a resource is, the more likely it is to be requested in the future — either by the same client, or by another. However, caching is not a panacea; it can introduce problems as well.

2.2.7 Potential problems

There are a number of potential problems associated with Web caching. Most significant from the views of both the content consumer and the content provider is the possibility of the end user seeing *stale* (e.g., old or out-of-date) content, compared to what is available on the origin server (i.e., *fresh* content). HTTP does not ensure *strong consistency* and thus the potential for data to be cached too long is real. The likelihood of this occurrence is a trade-off that the content provider can explicitly manage (as we discuss in the next section).

Caching tends to improve the response times only for cached responses that are subsequently requested (i.e., hits). Misses that are processed by a cache generally have decreased speed, as each system through which the transaction occurs will increase the latency experienced (by a small amount). Thus, a cache only benefits requests for content already stored in it. Caching is also limited by a high rate of change for

popular Web resources, and importantly, that many resources will be requested only once [DFKM97]. Finally, some responses cannot or should not be cached, as described below.

2.2.8 Content cacheability

Not every resource on the Web is cacheable. Of those that are, some will be cacheable for long periods by any cache, and others may have restrictions such as caching for short periods or to certain kinds of caches (e.g., non-proxy caches). This flexibility maximizes the opportunity for caching of individual resources. The cacheability of a Web site affects both its user-perceived performance, and the scalability of a particular hosting solution. Instead of taking seconds or minutes to load, a cached object can appear almost instantaneously. Regardless of how much the hosting costs, a cache-friendly design will allow more pages to be served before needing to upgrade to a more expensive solution. Fortunately, the cacheability of a resource is determined by the content provider. In particular, it is the Web server software that will set and send the HTTP headers that determine cacheability,[1] and the mechanisms for doing so vary for each Web server. To maximize the cacheability of a Web site, typically all static content (buttons, graphics, audio and video files, and pages that rarely change) are given an expiration date far in the future so that they can be cached for weeks or months at a time.

By setting an expiration date far into the future, the content provider is trading off the potential of caching stale data with reduced bandwidth usage and improved response time for the user. A shorter expiration date reduces the chance of the user seeing stale content, but increases the number of times that caches will need to validate the resource. Currently, most caches use a client polling approach that favors re-validation for objects that have changed recently [GS96]. Since this can generate significant overhead (especially for popular content), a better approach might be for

[1]Note that HTML META tags are not valid ways to specify caching properties and are ignored by most proxy caching products since they do not examine the contents of an object (i.e., they do not see the HTML source of a Web page).

the origin server to invalidate the cached objects [LC98], but only recently has this approach been considered for non-proprietary networks. Work is underway to develop the Web Cache Invalidation Protocol (WCIP) [LCD01], in which Web caches subscribe to invalidation channel(s) corresponding to content in which they are interested. This protocol is intended to allow the caching and distribution of large numbers of frequently changing Web objects with freshness guarantees.

Dynamically generated objects are typically considered uncacheable, although they are not necessarily so. Examples of dynamically generated objects include fast-changing content like stock quotes, personalized pages, real-time photos, results of queries to databases (e.g., search engines), page access counters, and e-commerce shopping carts. Rarely would it be desirable for any of these to be cacheable at an intermediate proxy, although some may be cacheable in the client browser cache (such as personalized resources that don't change often).

However, dynamically generated objects constitute an increasing fraction of the Web. One proposed approach to allow for the caching of dynamic content is to cache programs that generate or modify the content, such as an applet [CZB99]. Another is to focus on the server, and to enable caching of portions of documents so that server-side operations are optimized (e.g., server-side products from companies like SpiderCache [Spi02], Persistence [Per02] and XCache [XCa02]). Since most dynamic pages are mostly static, it might make sense to just send the differences between pages or between versions of a page [HL96, MDFK97] or to break those documents into separately cacheable pieces and reassemble them at the client (e.g., [DHR97]). The latter is also essentially what is proposed in Edge Side Includes [OA01] (e.g., Akamai's EdgeSuite service [Aka02]), except that the Web pages are assembled at edge servers rather than at the client.

2.2.9 Improving caching latencies

Caching can provide only a limited benefit (typically reaching object hit rates of 40-50% with sufficient traffic), as a cache can only provide objects that have been previously requested. If future requests can be anticipated, those objects can be obtained in advance. Once available in a local cache, those objects can be retrieved with minimal

delay, enhancing the user experience. *Prefetching* is a well-known approach to decrease access times in the memory hierarchy of modern computer architectures (e.g., [Smi82, Mow94, KCK+96, VL97, KW98a, LM99, JG99, SSS99]), and has been proposed by many as a mechanism for the same in the World Wide Web (e.g., [PM96, KLM97]).

While the idea is promising, prefetching is not straightforward to evaluate (as we point out later in this thesis and then propose an alternative approach in Chapter 10) and has not been widely implemented in commercial systems. It can be found in some browser add-ons and workgroup proxy caches that will prefetch the links of the current page, or periodically prefetch the pages in a user's bookmarks. One significant difficulty is to accurately predict which resources will be needed next so that mistakes which result in wasted bandwidth and increased server loads are minimized. Another concern is the need to determine what content may be prefetched safely, as some Web requests have (potentially undesirable) side-effects, such as adding items to an online shopping cart (which we discuss further in Chapter 12).

An alternative to true prefetching has been suggested by Cohen and Kaplan when they proposed techniques that do everything but prefetch [CK00]. That is, they demonstrated the usefulness of 1) pre-resolving (performing DNS lookup in advance); 2) pre-connecting (opening a TCP connection in advance); and 3) pre-warming (sending a dummy HTTP request to the origin server). These techniques can be implemented in both proxies and browsers, and could significantly reduce the response times experienced by users without large increases in bandwidth usage nor the retrieval of content with side effects.

2.3 Why do research in Web Caching?

There are a number of reasons to do research in Web caching.[2] Motivations in Web caching research can be distinguished by their outlook. In the short term, there is the desire to improve upon Web caching systems as they can affect the performance of the

[2]Many of these issues were brought up in a discussion on the `ircache` mailing list (`http://www.ircache.net/`). For the original posts, see the thread on 'free bandwidth' in the ircache archives at `http://www.roads.lut.ac.uk/lists/ircache/` for September 1998.

Web today. This includes:

- **Reducing the cost of connecting to the Internet.** Traffic on the Web consumes more bandwidth than any other Internet service, and so any method that can reduce the bandwidth requirements is particularly welcome, especially in parts of the world in which telecommunication services are much more expensive than in the U.S., and in which service is often provided on a cost per bit basis. Even when telecommunication costs are reasonable, a large percentage of traffic is destined to or from the U.S. and so must use expensive trans-oceanic network links.

- **Reducing the latency of today's WWW.** One of the most common end-user desires is for more speed. Web caching can help reduce the "World Wide Wait", and so research (such as in Web prefetching) that improves upon user-perceived response times is quite welcome. Latency improvements are most noticeable, again, in areas of the world in which data must travel long distances, accumulating significant latency as a result of speed-of-light constraints, or by accumulating processing time by each system over many network hops, and likewise increasing the chance of experiencing congestion as more networks are traversed to cover such distances. High latency as a result of speed-of-light constraints is particularly taxing in satellite communications.

In the long term one might argue that research in Web caching is not necessary, as the cost of bandwidth continues to drop. However, research in Web caching will continue to reap benefits as:

- **Bandwidth will always have some cost.** The cost of bandwidth will never reach zero, even though end-user costs are currently going down as competition increases, the market grows, and economies of scale contribute. At the same time, the cost of bandwidth at the core has stayed relatively stable, requiring ISPs to implement methods such as caching to stay competitive and reduce core bandwidth usage so that edge bandwidth costs can be low. Regardless of the

cost of bandwidth, one will always want to maximize the return on investment, and caching will often help. If applied indiscriminately, prefetching can make bandwidth needs larger, but when bandwidth is purchased at a flat rate (i.e., a T1 without per-bit charges), intelligent prefetching can be applied during idle periods to maximize the utility of the fixed cost.

- **Non-uniform bandwidth and latencies.** Because of physical limitations such as environment and location (including speed-of-light constraints mentioned above), as well as financial limitations, there will always be variations in bandwidth and latencies. Caching can help to smooth these effects.

- **Network distances are increasing.** More users are sending requests through firewalls and other proxies for security and privacy, which slows Web response times. Likewise, the use of virtual private networks for telecommuters and remote offices means that Web requests must first be routed through the VPN and suffer the associated latency costs. Both of these increase the number of hops through which content must travel, directly increasing the cost of a Web retrieval. While access to some objects served by CDNs are getting better, these still represent only a relatively small fraction of a typical user's content [KV01, BV02].

- **Bandwidth demands continue to increase.** New users are still getting connected to the Internet in large numbers. Even as growth in the user base slows, demand for increased bandwidth will continue as high-bandwidth media such as audio and video increase in popularity. If the price is low enough, demand will always outstrip supply. Additionally, as the availability of bandwidth increases, user expectations of faster performance correspondingly increase.

- **Hot spots in the Web will continue.** While some user demand can be predicted (such as for the latest version of a free Web browser), and thus have the potential for intelligent load balancing by distributing content among many systems and networks, other popular Web destinations come as a surprise, sometimes as a result of current events, but also potentially just as a result of desirable content and word of mouth. These 'hot spots' resulting from flash traffic loads will

continue to affect availability and response time and can be alleviated through distributed Web caching.

- **Communication costs exceed computational costs.** Communication is likely to always be more expensive (to some extent) than computation. We can build CPUs that are much faster than main memory, and so memory caches are utilized. Likewise, caches will continue to be used as computer systems and network connectivity both get faster.

There are a number of parallels to the persistence of caching in the Web — one is that of main memory in computer systems. The cost of RAM has decreased tremendously over the past decades. Yet relatively few people would claim to have enough, and in fact, demand for additional memory to handle larger applications continues unabated. Therefore, virtual memory (caching) continues to be used strategically instead of purchasing additional physical memory.

2.4 Summary

The benefits of caching are apparent. Given the choices of caching products on the market today, how can one be selected? The process involves determining requirements like features (such as manageability, failure tolerance, and scalability), and performance (such as requests per second, bandwidth saved, and mean response times — often measured in cache benchmarks [RW00, RWW01]). Likewise, as more Web traffic is served by caches and content delivery services instead of origin servers, it is prudent for content developers to consider how to maximize the usefulness of this technology.

By reducing network latencies and bandwidth demands in a way that is functionally transparent to the end points, caching has become a significant part of the infrastructure of the Web. This chapter has provided an overview of Web caching technology, including where Web caching is utilized, and discussed how and why it is beneficial to end users and content providers. To examine this topic in more depth, take a look at books from Addison-Wesley [KR01, RS02] and O'Reilly [Wes01], or visit `www.web-caching.com`.

Chapter 3

Incremental Probabilistic Action Prediction

3.1 Introduction

How predictable are people? Each of us displays patterns of actions throughout whatever we do. Most occur without conscious thought. Some patterns are widespread among large communities, and are taught, as rules, such as reading from left to right, or driving on the correct side of the road. Other patterns are a function of our lifestyle, such as picking up pizza on the way home from work every Friday, or programming the VCR to record our favorite comedy each week. Many are a result of the way interfaces are designed, like the pattern of movement of your finger on a phone dialing a number you call often, or how you might log into your computer, check mail, read news, and visit your favorite website for the latest sports scores.

As computers pervade more and more aspects of our lives, the need for a system to be able to adapt to the user, perhaps in ways not programmed explicitly by the system's designer, become ever more apparent. A car that can offer advice on driving routes is useful; one that can also guess your destination (such as a pizza parlor because it is Friday and you are leaving work) is likely to be found even more useful, particularly if you didn't have to program it explicitly with that knowledge. The ability to predict the user's next action allows the system to anticipate the user's needs (perhaps through speculative execution or intelligent defaults) and to adapt to and improve upon the user's work habits (such as automating repetitive tasks). Additionally, adaptive interfaces have also been shown to help those with disabilities [GDMW95, DM92].

This chapter will provide an overview of prediction models, and discuss some aspects of the approaches possible as well as enumerate characteristics that an idealized prediction approach might have. However, we will leave experimental examination of

most of these methods until Chapter 4, and spend the rest of this chapter providing an introduction to just one approach — the use of a simple mechanism for the prediction of user actions. The domain of UNIX command prediction will serve as a testbed for the development of IPAM, an algorithm based on first-order Markov-models with an emphasis on recency for prediction of the next action, which we describe below in Section 3.4.

3.2 Background

The problem of modeling a user's behavior has received much attention from researchers in recent years. Given an accurate model, the user's interface can be customized — perhaps to improve ease of comprehension or navigation by providing hints or suggestions, or to increase the likelihood of generating a sale on an e-commerce site, or to invisibly work to improve the user's perceptions of performance by anticipating the needs of the user in advance.

Regardless of the ultimate application, the task is the same — to build a model that can predict future user actions. Building such a model can take various forms, including user interviews and explicit feedback, or by passively watching the user [MMS85, Cha99]. This chapter focuses on perhaps the most common source of information about the user — the user's past actions.

However, we are also concerned with building and using those predictions within an interactive system — typically one that uses each action as a chance to learn and an opportunity to make predictions to assist the user. This means the learning algorithm needs to be usable in an online setting, unlike batch learning approaches that are typically too slow to be retrained using a large dataset whenever they need to be updated.

"Machine learning is concerned with the question of how to construct computer programs that automatically improve with experience." [Mit97]. However, while learning to predict a user's action is certainly within the domain of machine learning, the problem domain is specific enough that the approaches commonly employed are not

necessarily typical machine learning algorithms.

The approaches taken for action prediction vary considerably, from subsequence matching (as in [DH97b, SKS98, PP99b]) in which past sequences of actions can provide a possible next action for the current sequence, to Markov and Markov-like models including both single and multi-step (e.g., [Bes95, PM96, JG99, DK01, PP99a, SYLZ00]) in which the possible next actions are encoded as states with probabilities from the current state (labeled with the most recent action). Other approaches are possible, such as something in between like Prediction by Partial Match (e.g., [KL96, Pal98, PM99, FJCL99]) and TDAG [Lai92, LS94] which combine Markov models of varying order, to inducing a grammar (as in [NM96]). Many of these methods have connections to data compression (in particular, TDAG, PPM, and grammar induction), and often have complex implementations. We will discuss a number of them in Chapter 4. In many cases, however, simple methods can do well, as we show in this chapter by introducing a simple Markov-like algorithm for action prediction and test it in the domain of UNIX commands.

3.3 UNIX Command Prediction

3.3.1 Motivation

We consider here user actions within a command line shell. We have concentrated initially on UNIX command prediction[1] because of its continued widespread use; the UNIX shell provides an excellent testbed for experimentation and automatic data collection. However, our interest is in more general action prediction. This chapter, therefore, reflects our focus on the underlying technology for action prediction, rather than on how prediction can be effectively used within an interface.

We wish to address the task of predicting the next element in a sequence, where the sequence is made up of nominal (unordered as well as non-numeric) elements. This type of problem (prediction of the next step in a series) is not studied often by machine

[1] In this work we ignore command arguments and switches, but others [KG00] have replicated our approach on full command lines and similarly found it to be quite successful.

learning researchers; concept recognition (i.e., a boolean classification task such as sequence recognition) is more common, as is the use of independent samples from a distribution of examples. UNIX commands, and user actions in general, however, are not independent, and being nominal, don't fall into the domain of traditional statistical time-series analysis techniques.

Below we use the user histories from two studies to suggest that relatively naive methods can predict a particular user's next command surprisingly well. Finally, we will present and analyze a novel algorithm that satisfies these characteristics and additionally improves on the performance of our previous work [DH97a, DH97b].

3.3.1.1 Evaluation Criteria

In most machine learning experiments that have a single dataset of independent examples, cross-validation [Sto74, Sto78, BFOS84] is the standard method of evaluating the performance of an algorithm. When cross-validation is inappropriate, partitioning the data into separate training and test sets is common. For sequential datasets, then, the obvious split would have the training set contain the first portion of the sequence, and the test set contain the latter portion (so that the algorithm is not trained on data occurring after the test data). However, since we are proposing an adaptive method, we will be evaluating performance online — each algorithm is tested on the current command using the preceding commands for training. This maximizes the number of evaluations of the algorithms on unseen data and reflects the expected application of such an algorithm.

When considering performance across multiple users with differing amounts of data, we use two methods to compute averages. *Macroaveraged* results compute statistics separately for each user (e.g., the mean accuracy), and then average these statistics over all users. Alternatively, *microaveraged* results compute an average over all data, determining the number of correct predictions made across all users divided by the total number of commands for all users combined. The former provides equal weight to all users, since it averages across the average performance of each user; the latter gives equal weight to all predictions, emphasizing users with large amounts of data.

```
...
96102513:34:49 cd
96102513:34:49 ls
96102513:34:49 emacs
96102513:34:49 exit
96102513:35:32 BLANK
96102513:35:32 cd
96102513:35:32 cd
96102513:35:32 rlogin
96102513:35:32 exit
96102514:25:46 BLANK
96102514:25:46 cd
96102514:25:46 telnet
96102514:25:46 ps
96102514:25:46 kill
96102514:25:46 emasc
96102514:25:46 emacs
96102514:25:46 cp
96102514:25:46 emacs
...
```

Figure 3.1: A portion of one user's history, showing the timestamp of the start of the session and the command typed. (The token BLANK marks the start of a new session.)

3.3.1.2 People Tend To Repeat Themselves

In order to determine how much repetition and other recognizable regularities were present in the average user's command line work habits, we examined two datasets of many users' activities in a UNIX shell. In the first (Rutgers) dataset, we collected command histories of 77 users, totaling over 168,000 commands executed during a period of 2-6 months [DH97a, DH97b] (see Figure 3.1 for an example of the data that was collected). The bulk of these users (70) were undergraduate computer science students in an Internet programming course and the rest were graduate students or faculty. All users had the option to disable logging and had access to systems on which logging was not being performed.

The average user had over 2000 command instances in his or her history, using 77 distinct commands during that time. On average over all users (macroaverage), 8.4% of the commands were new and had not been logged previously. The microaverage of new commands, however, was only 3.6%, reflecting the fact that smaller samples had larger numbers of unique commands. Approximately one out of five commands were the same

as the previous command executed (that is, the user repeated the last command 20% of the time).

The second (Greenberg) dataset is a larger collection of user activities with the `csh` UNIX shell [Gre88, Gre93]. It contains more than twice as many users and approximately twice as many commands overall. This dataset was acquired after completing the tests on the first, but we will present the results of both datasets when appropriate.

3.3.1.3 Earlier Results

In previous work [DH97a, DH97b], we considered a number of simple and well-studied algorithms. In each of these, the learning problem was to examine the commands executed previously, and to predict the command to be executed next. We found that, without explicit domain knowledge, a naive method based on C4.5 [Qui93] was able to predict each command with a macroaverage accuracy of 38.5% (microaverage was 37.2%). Given a sequence of commands, C4.5 was trained to use the preceding two commands to predict the next. Thus for each prediction, C4.5 was trained on the series of examples of the form $(\text{Command}_{i-2}, \text{Command}_{i-1}) \Rightarrow \text{Command}_i$; for $1 \leq i \leq k$, where k is the number of examples seen so far. Command_0 and Command_{-1} are both defined to have the special value BLANK to allow prediction of the first and second commands where otherwise there would be no previous two commands.

While the prediction method was a relatively straightforward application of a standard machine learning algorithm, it has a number of drawbacks, including that it returned only the single most likely command. C4.5 also has significant computational overhead. It can only generate new decision-trees; it does not incrementally update or improve the decision tree upon receiving new information. (While there are other decision-tree systems that can perform incremental updates [Utg89], they have not achieved the same levels of performance as C4.5.) Therefore, C4.5 decision tree generation must be performed outside of the command prediction loop.

Additionally, since C4.5 (like many other machine learning algorithms) is not incremental, it must revisit each past command situation, causing the decision-tree generation to require more time and computational resources as the number of commands

in the history grows. Finally, it treats each command instance equally; commands at the beginning of the history are just as important as commands that were recently executed. Note that C4.5 was selected as a common, well-studied decision-tree learner with excellent performance over a variety of problems, but not with any claim of superiority over other algorithms applicable to this domain.

These initial experiments dealt with some of these issues by only allowing the learning algorithm to consider the command history within some fixed window. This prevented the model generation time from growing without bound and from exceeding all available system memory. This workaround, however, caused the learning algorithms to forget relatively rare but consistently predictable situations (such as typographical errors) and restricted consideration only to recent commands.

3.3.2 An Ideal Online Learning Algorithm (IOLA)

By focusing on the application of adaptive interfaces, we can propose characteristics of an "ideal" learning algorithm for such problems. An Ideal Online Learning Algorithm (IOLA) and its realization would:

1. have predictive accuracy as good as the best known resource-unlimited methods;
2. operate incrementally (modifying an existing model rather than building a new one as new data are obtained);
3. be affected by all events (remembering uncommon, but useful, events regardless of how much time has passed);
4. not need to retain a copy of the user's full history of actions;
5. output a list of predictions, sorted by confidence;
6. adapt to changes to the target concept;
7. be fast enough for interactive use;
8. learn by passively watching the user; and

	A1	A2	A3	A4	A5
A1	1/10	4/10	2/10	1/10	2/10

(a) Before update.

	A1	A2	A3	A4	A5
A1	1/11	4/11	3/11	1/11	2/11

(b) After update.

Figure 3.2: Standard (Bayesian) incremental update: one row of a 5x5 state transition table, before and after updating the table as a result of seeing action A3 follow A1.

9. apply even in the absence of domain knowledge.

Such an algorithm would be ideally suited for incorporation into many systems that interact with users.

3.4 Incremental Probabilistic Action Modeling (IPAM)

3.4.1 The algorithm

In our previous work, we implicitly assumed that the patterns of use would form multi-command chains of actions, and accordingly built algorithms to recognize such patterns. If, however, we make the simpler Markov assumption that each command depends only on the previous command (i.e., patterns of length two, so that the previous command is the state), we can use the history data collected to count the number of times each command followed each other command and thus calculate the probability of a future command. This could be implemented by the simple structure of an n by n table showing the likelihood of going from one command to the next, as shown in Figure 3.2a.

For the anticipated use of action prediction in an adaptive interface, however, an incremental method is desirable. Instead of just using a static table of probabilities calculated on some training data, the table can be updated after every new instance (as shown by the portions of a state transition table before and after an update in Figure 3.2). From the recorded counts, the fractions can be interpreted as probabilities for the various state transitions. However, as mentioned earlier, we believe it is useful to weigh recent events more highly when building an adaptive model. This can be accomplished

	A1	A2	A3	A4	A5
A1	.1	.4	.2	.1	.2

(a) Before IPAM update.

	A1	A2	A3	A4	A5
A1	.08	.32	.36	.08	.16

(b) After IPAM update.

Figure 3.3: IPAM incremental update: one row of a 5x5 state transition table, before and after updating the table using IPAM as a result of seeing action A3 follow A1. The value of *alpha* used was .8.

in this probabilistic model by the use of an update function with an exponential decay (in which the most recent occurrence has full impact; older occurrences have ever-decreasing contributions). Given the previous table of probabilities and another table containing probabilities from new data points, a combined new table may be computed by the weighted average of the two, where the weights sum to 1. So, for example, if the weights were both .5, the new probabilities would have equal contributions from the old table and from the new. Assuming that the table updates were performed periodically, the data points making up the first table would be contributing only $\frac{1}{2^n}$ percent of the final weights (where n is the number of table updates so far).

We can extend this model further, to an algorithm that starts with an empty table and updates after every command. An empty table is one in which all commands are equally likely (initially a uniform probability distribution). After seeing a command, c_i for the first time, a new row is added for that command, and has a uniform distribution. When the second command, c_{i+1}, is seen, it too gets a new row with a uniform distribution, but we update the first row (since we saw c_i followed by c_{i+1}) by multiplying all elements of that row by a constant $0 \leq alpha \leq 1$, and the probability of seeing c_{i+1} is increased by adding $(1 - alpha)$. In this way, we emphasize more recent commands, at the expense of older actions, but the sum of the probabilities in each row is always 1. An example can be found in Figure 3.3.

Note that an *alpha* of 0 equates to a learner that always predicts what it most recently saw for that command, and an *alpha* of 1 corresponds to an algorithm that never changes its probabilities (in this case, preserving a uniform distribution).

	A1	A2	A3
Default	.712	.128	.16
A1	.8	.2	0

(a) Before IPAM update.

	A1	A2	A3	A4
Default	.5696	.1024	.128	.2
A1	.64	.16	0	.2
A4	.5696	.1024	.128	.2

(b) After IPAM update.

Figure 3.4: Selected rows of a 5x5 state transition table, before and after updating the table as a result of seeing action A4 for the first time.

```
Update(PreviousCommand, CurrentCmd):
  - Call UpdateRow(DefaultRow, CurrentCmd)
  - Call UpdateRow(PreviousCommand's row, CurrentCmd)

UpdateRow(ThisRow, CurrentCmd):
  - If initial update for ThisRow, copy distribution from default row
  - Multiply probability in each column by alpha and add (1-alpha) to
    column that corresponds to CurrentCmd
```

(a) The update function.

```
Predict(NumCmds,PreviousCmd):
  - Call SelectTopN(NumCmds, PreviousCmd's row, {})
  - Let P be the number of commands returned
  - If P < NumCmds, call SelectTopN again, but ask for the top
    (P - NumCmds) commands from the default row and to exclude those
    commands already returned
  - Return the first set followed by the default set of predicted commands

SelectTopN(NumCmds,Row,ExcludeCmds):
  - Sort the probabilities in Row
  - Eliminate commands in ExcludeCmds
  - Return the top NumCmds from sorted list
```

(b) The predict function.

Figure 3.5: Pseudo-code for IPAM.

For prediction, the command with highest probability for the given state would be output as the most likely next command. Instead of making no prediction for a command with an empty row, we can track probabilities in an additional *default* row, which would use the same mechanism for updating but would apply to all commands

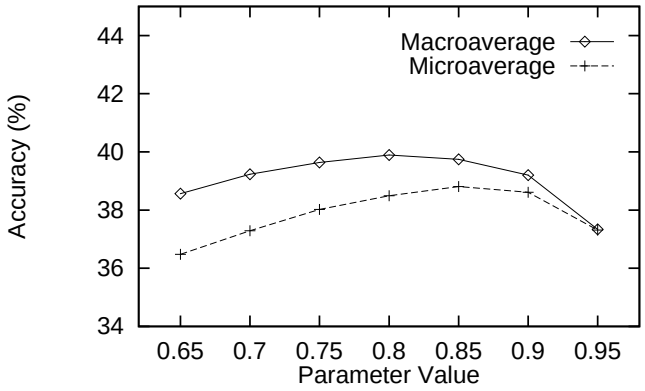


Figure 3.6: For a range of *alpha* values, the predictive accuracy of the Incremental Probabilistic Action Modeling algorithm on the Rutgers data is shown.

seen so far (without consideration of the preceding command). Finally, since we are keeping track of overall likelihoods in this default row, we can use it to initialize rows for new commands (making the assumption that these default statistics are better than a uniform distribution). Figure 3.4 illustrates the process of adding a new row when an action is first seen.

See Figure 3.5 for pseudo-code for the `Update` and `Predict` functions that implement this Incremental Probabilistic Action Modeling (IPAM) algorithm.

3.4.2 Determining *alpha*

We empirically determined the best average *alpha* by computing the performance for this algorithm on the Rutgers dataset with each of seven values of *alpha* (from .65 to .95 in increments of .05). While the best value of *alpha* varied, depending on how performance was calculated over the dataset, our subjective choice for the best overall was .80. (See Figure 3.6 for a graph of the parameter study of *alpha* showing the average user's performance as well as the average performance over all commands.) We will use this value of *alpha* for the rest of the experiments in this chapter. Since *alpha* controls the amount of influence recent commands have over earlier commands, we expect that this value will vary by problem domain. To ensure fair evaluation of our approach with this parameter, we do not recalculate a best *alpha* for the Greenberg data, but instead

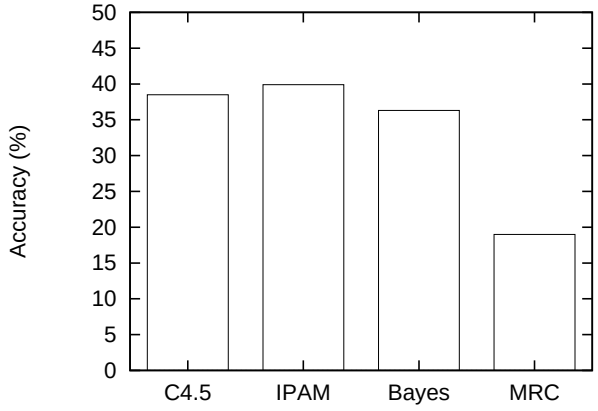
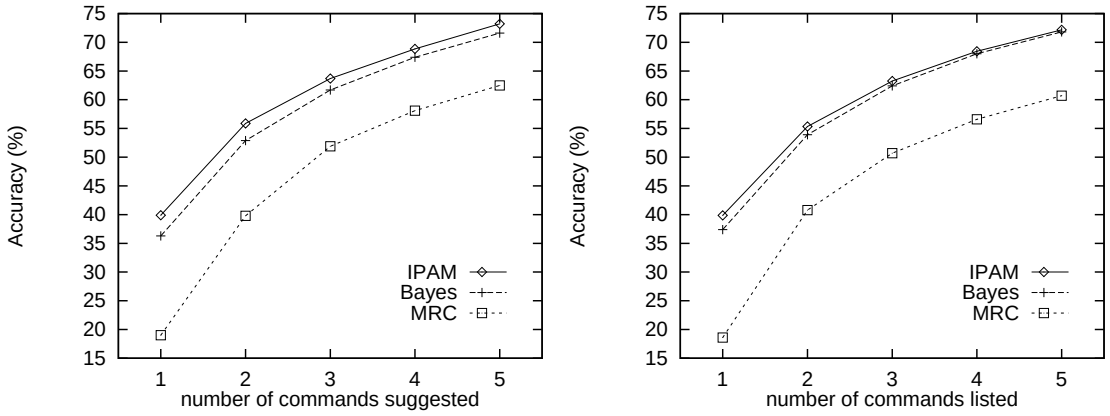


Figure 3.7: Macroaverage (per user) predictive accuracy for a variety of algorithms on the Rutgers dataset.



(a) Rutgers dataset.

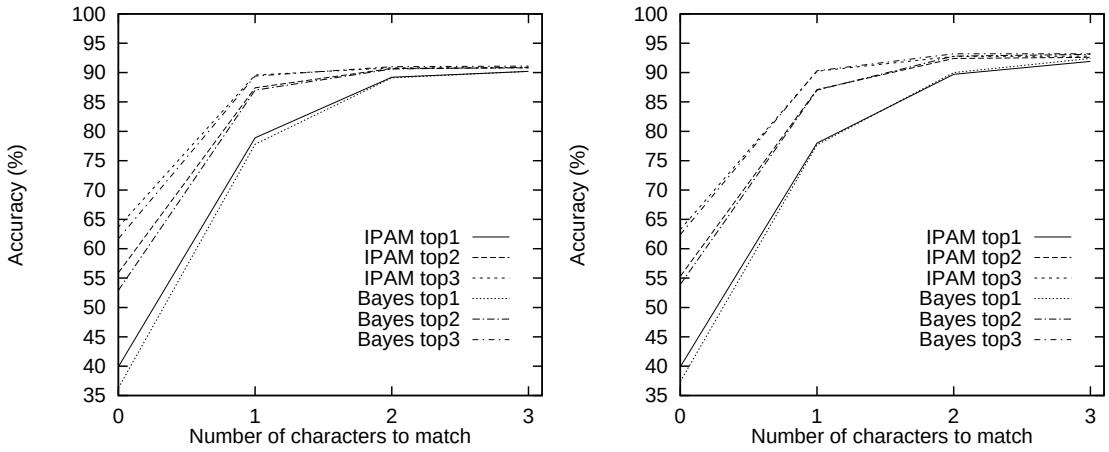
(b) Greenberg dataset.

Figure 3.8: Average per user accuracies of the top-n predictions. The likelihood of including the correct command grows as the number of suggested commands increases.

re-use the same value.

3.5 Evaluation

This algorithm, when applied to the Rutgers data set, performs better than C4.5 (given an *alpha* of .80). It achieves a predictive accuracy of 39.9% (macroaverage) and 38.5%



(a) Rutgers dataset.

(b) Greenberg dataset.

Figure 3.9: Command completion accuracies. The likelihood of predicting the correct command grows as the number of initial characters given increases.

(microaverage) versus C4.5's 38.5% and 37.2% (macroaverage and microaverage, respectively) for best guess predictions (see the bars labeled C4.5 and IPAM in Figure 3.7). For comparison, we also show our method without the specialized update, which corresponds to naive Bayes (that is, a predictor in which the conditional probabilities to select the most likely next command are based strictly on the frequency of pairs of commands), as well as a straightforward most recent command predictor (labeled MRC) that always suggests the command that was just executed.

To be precise, over the 77 Rutgers users, IPAM beat the C4.5-based system sixty times, tied once, and lost sixteen times on the task of predicting the next command. At the 99% confidence level, the average difference between their scores was 1.42 ± 1.08 percentage points, showing that the improvement in predictive accuracy for IPAM over C4.5 is statistically significant, given this ideal value for *alpha*. While in practice the improvement in accuracy might not be noticeable, but it is important to note the simplicity (as well as other IOLA characteristics) of IPAM in comparison to C4.5.

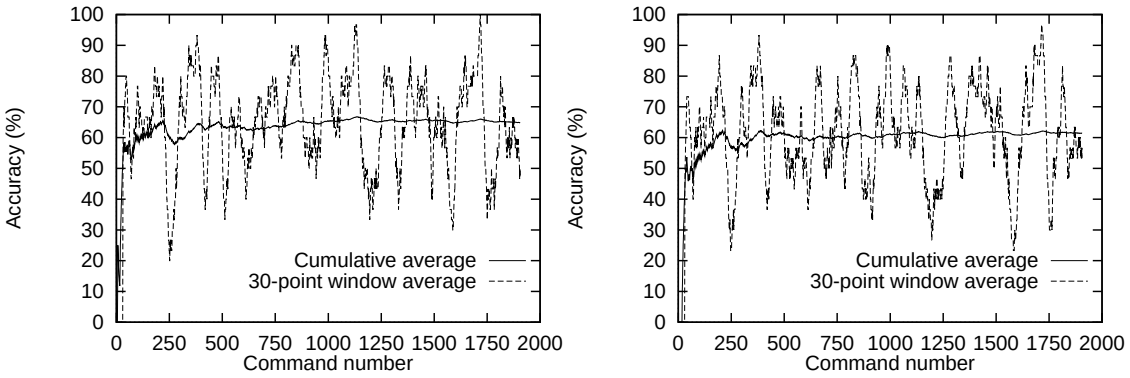
IPAM keeps a table in memory of size $O(k^2)$, where k is the number of distinct commands. Predictions can be performed in constant time (when a list of next command is kept sorted by probability), with updates requiring $O(k)$ time.

Some applications of this method may be able to utilize multiple predictions for a given situation. For example, an adaptive interface might present for easy selection the top five most likely choices that a user would make. Thus, such an interface would want the five predictions that are most likely to be correct. Systems that can generate top-n predictions are also valuable when the predictions are to be combined with other information with predictive value. Since IPAM generates a list of commands with associated probabilities for prediction, we can also compute the average accuracy of the top-n commands for varying values of n (as compared to only the single most likely command). Figures 3.8a and b show that we do get increased performance and that for $n = 5$, the correct command will be listed almost 75% of the time. This makes it possible for an interface designer to consider the trade-off of increased likelihood of listing the correct command versus the increased cognitive load of an interface showing multiple suggestions.

In UNIX command prediction, it is also helpful to be able to perform command completion (that is, taking the first k characters typed and produce the most likely command that is prefixed by those characters). Such a mechanism would enable shells that perform completion when there is a unique command with that prefix (such as `tcsh`) to also be able to perform completion when there are multiple possibilities. Figures 3.9a and b measure the predictive accuracy when given 0-3 initial characters to match when applied to all of the data. (Note that command completion when given 0 initial characters is just command prediction.) As expected, accuracy improves as additional characters of the command are provided to the system. For this task, examining just the first character almost doubles predictive performance (from 40% to close to 80% when predicting the single most likely command).

3.6 Discussion

While not shown, the results described apply to both macroaverage performance (shown in most figures) and microaverage performance, although the former is almost always slightly higher. While the results on the first dataset (collected from users of `tcsh`) can be argued as showing the *potential* for this method (since the selection of *alpha*



(a) IPAM.

(b) Bayes.

Figure 3.10: Cumulative performance (for n=3) over time for a typical user.

was based on the same set), the performance on the larger, and arguably more representative, Greenberg dataset (collected almost ten years earlier from users of *csh*) demonstrates a more believable performance.

Although learning may be performed throughout the history of a user's actions, the cumulative accuracy of a user does not vary much after an initial training period. Figures 3.10*a* and *b* show the performance of the IPAM and Bayes algorithms, respectively over the history of one user. The solid line depicts the current overall average predictive accuracy (from the first command to the current command), while the dashed line shows the variations in predictive accuracy when measured over a window containing only the most recent 30 commands.

We might consider initializing the table in IPAM with useful values rather than starting from scratch. For example, is the performance improved if we start with a table averaged over all other users? This lets us examine cross-user training to leverage the experience of others. Unfortunately, preliminary experiments indicate that, at least for this dataset, starting with the average of all other users' command prediction tables does not improve predictive accuracy. This result matches with those of Greenberg [Gre93] and Lee [Lee92], who found that individual users were not as well served by systems tuned for best average performance over a group of users.

The goal of our work has been to discover the performance possible without domain knowledge. This can then be used as a benchmark for comparison against 'strong methods', or as a base upon which a system with domain-specific knowledge might be built. IPAM's implementation, in addition, was guided by the characteristics of an IOLA, and thus has other benefits in using limited resources in addition to performance.

3.7 Related Work

The problem of learning to predict a user's next action is related to work in a number of areas. There are many similarities in the problems studied in plan and goal recognition in the user modeling community (e.g., [Bau96, LE95, Les97]). Such work attempts to model users in terms of plans and goals specific to the given task domain. Efforts in this area typically require that the modeling system know the set of goals and plans in advance. This usually requires a significant human investment in acquiring and representing this domain knowledge. In contrast, our goal is to explore the potential for action prediction in a knowledge-sparse environment (i.e., where user plans are not known or cannot easily be developed).

A smaller number of researchers have, instead, studied methods that have similar goals of generality. Yoshida and Motoda [MY97, YM96, Yos94] apply specially developed machine learning techniques to perform command prediction. This lets them implement speculative execution, and they report fairly high predictive accuracy (albeit on a small amount of real user data). However, much of the success of their work comes from knowing a fair amount about each action a user takes, by using an extension to the operating system that lets them record the I/O accesses (e.g., reading files) that each command performs.

Greenberg [Gre93] and Lee [Lee92] have studied patterns of usage in the UNIX domain, focusing on simple patterns of repetitions. They found that the recurrence rate (the likelihood of repeating something) was high for command lines as well as for commands themselves, but that individual usage patterns varied. More recently, Tauscher and Greenberg [TG97] extended Greenberg's recurrence analysis to URL revisitation

in World Wide Web browsers. These efforts consider only some simple methods for offering the top-n possibilities for easy selection (such as the most recent n commands).

Stronger methods (including a genetic algorithm-based classifier system) were attempted by Andrews in his master's thesis [And97] to predict user commands, but he had only a small sample of users with fairly short histories (≤ 500) in a batch framework and thus unclear implications. In a conference paper, Debevc, Meyer, and Svecko [DMS97] report on an application called the Adaptive Short List for presenting a list of potential URLs as an addition to a Web browser. This method goes beyond recency-based selection methods, and instead tracks a priority for each URL which is updated after each visitation. The priority for a particular URL is computed essentially as the normalized averages of: a count of document usage, relative frequency of document usage, and the count of highest sequential document usage. This contrasts with our approach most significantly in that it does not consider relationships between documents and thus patterns of usage. Therefore, their Adaptive Short List computes a simplistic 'most likely' set of documents without regard to historical context.

A number of researchers have studied the use of machine learning in developing intelligent interfaces incorporating action prediction. For example, WebWatcher [JFM97] predicts which links on a page on World Wide Web a user will select, and Hirashima *et al.* [HMT98] present a method for context-sensitive filtering, but both systems rely on the precise nature of the artifacts being manipulated (namely decomposable pages of text). Predictive methods in automated form completion [SW96, HS93] are similarly tailored to the specifics on the application. Similarly, the Reactive Keyboard [DWJ90] uses simple history-matching methods for prediction, but at the detailed level of keypresses, allowing for easy word or phrase completion as a document is being typed.

Programming by demonstration [Cyp93b, NM93] also has some similarities to this work. For example, Cypher's Eager [Cyp93a] can automate explicitly marked loops in user actions in a graphical interface. They, too, are concerned with performance when integrated into a user interface. While our approach is not designed to notice arithmetic progressions in loops, we can find and use the patterns in usage that do not recur as explicit loops and do not require special training by the user. Masui [MN94] also learns

repeated user patterns, requiring the user to hit the 'repeat' button when the system should learn or execute a macro.

Sequence prediction is also strongly related to data compression [BCW90] since an algorithm that can predict the next item in a sequence well can also be used to compress the data stream. Indeed, many of the approaches we describe could indeed be used in this fashion, precisely because they apply when only the user's history is available. However, we differ in that success in compression would only be an interesting phenomenon but not one that we explicitly target for our methods. Perhaps even more importantly, we target methods in which additional information sources can be easily injected. Our methods also are designed to be responsive to *concept drift*, since we make no assumptions about the stability of a user's actions over time — something that tends to reduce the usefulness of dictionary or grammar-based compression schemes [NM96]. Laird and Saul [LS94] present the TDAG algorithm for discrete sequence prediction, and apply it to a number of problems, including text compression, dynamic program optimization, and predictive caching. TDAG is based on Markov-trees, but limits the growth of storage by discarding the less likely prediction contexts. It is a fast online algorithm, but it, too, does not explicitly consider the problem of concept drift — each point in a sequence has essentially the same weight as any other point.

While more distantly related, work in anomaly detection for computer systems [Kum95, Lun90] develops ways to quantify a user's normal behavior so that unusual activity can be flagged as a potential intrusion. Most intrusion detection systems can be categorized as using either statistical-anomaly detection, or rule-based detection. While rule-based expert systems monitor for known attack patterns and thus trigger few false alarms, they are also criticized as encouraging the development of ad hoc rules [ESNP96] and require significant human engineering effort to develop. In contrast, statistical systems traditionally build profiles of normal user behavior and then search for the unusual sequences of events for consideration [DS98, FP99, Lan00]. Unlike most systems that perform anomaly detection by audit-trail processing off-line, our method works online, incrementally updating users' profiles as additional data arrives and could be augmented to provide user recognition.

Finally, IPAM's success has fostered work by others. Jacobs and Blockeel [JB01] use Inductive Logic Programming to capture regularities to help build shell scripts. Zukerman *et al.* [ZAN99] considers user models for plan recognition that predict user's immediate activities and their eventual goals in games and the Web. Gorniak and Poole [GP00b, GP00a] consider the problem of automatically building models for action prediction without modifying the application. Ruvini and Dony [RD99] claim near IOLA status for their approach to programming by demonstration. Lastly, Korvemaker and Greiner [KG00] explicitly attempt to improve upon IPAM's results in predicting full UNIX command lines, but find little or no improvement possible.

3.8 Summary

This chapter has introduced a method for action prediction that fulfills the requirements of an Ideal Online Learning Algorithm. Incremental Probabilistic Action Modeling has an average predictive accuracy for UNIX command prediction at least as good as that previously reported with C4.5. It operates incrementally, will remember rare events such as typos, and does not retain a copy of the user's action history. IPAM can generate top-n predictions, and by weighing recent events more heavily than older events it is able to react to 'concept-drift'. Finally, its speed and simplicity make it a strong candidate for incorporation into the next adaptive interface.

By only using the previous command, IPAM provides a straightforward method for history-based prediction. In Chapter 4 we examine more complex prediction models and consider the application of such history-based models for the domain of interest — Web request prediction. Prediction solely from history has some limitations, however, and so in Chapters 5 and 6 we consider an alternative — the exploitation of content to assist in modeling user interests for prediction.

Chapter 4

Web Request Prediction

4.1 Introduction

Modeling user activities on the Web has value for both content providers and consumers. Consumers may appreciate better responsiveness as a result of precalculating and preloading of content in advance of their requests. Additionally, consumers may find adaptive and personalized Web sites that can make suggestions and improve navigation useful. Likewise, the content provider will appreciate the insights that modeling can provide and the potential financial implications of a happier consumer that gets the information desired even faster.

Sequence prediction (that is, predicting the next item in a sequence) is fundamental to prefetching, whether for memory references, or disk accesses, or Web requests. Without accurate prediction, prefetching is not worthwhile. Therefore, it is useful to consider the various methods for prediction, even before embedding them within a prefetching system. For this reason we have introduced the simple IPAM prediction method in Chapter 3, and examine the topic in more depth here. Later, in Chapters 8 and 9 we will embed these prediction mechanisms within a simulator to estimate user-perceived response times, rather than just predictive accuracy.

Sequence prediction in general is a well-studied problem, particularly within the data compression field (e.g., [BCW90, CKV93, WMB99]). Unfortunately, the Web community has re-discovered many of these techniques, leading to islands of similar work with dissimilar vocabulary. Here we will both re-examine these techniques as well as offer modifications motivated by the Web domain. This chapter will describe, implement, and experimentally evaluate a number of methods to model usage and predict Web requests. Our focus will be on Markov-based and Markov-like probabilistic

techniques, both for their predictive power, but also their popularity in Web modeling and other domains.

We are interested in applying prediction to various types of Web workloads – those seen by clients, proxies, and servers. Each location provides a different view of Web activities, and the context in which they occur. As a result, different levels of performance will be possible.

This chapter will provide:

- an enumeration and discussion of the many aspects of how, what, and why to evaluate Web sequence prediction algorithms,
- a consistent description of various algorithms (often independently proposed) as just variations and simplifications of each other,
- the implementation of a prediction program sufficiently generalized to implement a variety of techniques,
- a comparison of multiple algorithms on a single set of traces to facilitate comparison,
- evidence of some "concept drift" within Web traces (similar to what we found in Chapter 3), and
- three Web domain-inspired improvements to existing algorithms.

In the next section, we will detail the many concerns and approaches we will take to the empirical evaluation of various Web request prediction algorithms. In Section 4.3 we describe the various algorithms and their extensions as we have implemented them. In Section 4.4 we describe the workloads used and present experimental results in Section 4.5. Finally, Sections 4.6 and 4.7 discuss and summarize the findings of this chapter.

4.2 Evaluation concerns and approaches

In this section we discuss the various aspects of how and what to evaluate when we compare Web request prediction algorithms. Two high-level concerns that are addressed repeatedly are the questions of 1) whether to modify typical evaluation approaches to better fit the domain, and 2) whether to modify predictions to better fit the domain, or both.

4.2.1 Type of Web logs used

One important aspect of any experimental work is the data sets used in the experiments. While we will introduce the Web workloads that we will use in Section 4.4, the type of workload is an evaluation concern. At a high level, we are simply concerned with methods that learn models of typical Web usage. However, at a lower level, those models are often simply identifying co-occurrences among resources — with the ultimate goal to make accurate predictions for resources that might be requested given a particular context.

However, there are multiple types of relationships between Web resources that might cause recognizable co-occurrences in Web logs [Bes96, CJ97, Cun97]. One possible relationship is that of an embedded object and its referring page — an object, such as an image, audio, or Java applet that is automatically retrieved by the browser when the referring page is rendered. Another relationship is that of traversal — when a user clicks on a link from the referring page to another page. The first (embedding) is solely an aspect of how the content was prepared. The second (traversal), while likewise existing because of a link placed by the content creator, is also a function of how users navigate through the Web hypertext.

Many researchers (see, for example [JK97, JK98, NZA98, Pal98, ZAN99, SYLZ00, SH02, YZL01]) distinguish between such relationships and choose not to make predictions for embedded resources. They are concerned with "click-stream" analysis — just the sequence of requests made by the user and not the set of automatically requested additional resources. Sometimes the data can provide the distinction for us — a Web

server often records the referrer in its access logs, which captures the traversal relationship. But the HTTP referrer header is not a required field, and is thus not always available. In other cases, analysis of the data is required to label the kind of request — for example, requests for embedded links are usually highly concentrated in time near the referring page. Unfortunately, many of the publicly available access logs do not provide sufficient detail to allow us to make such distinctions authoritatively. In addition, methods that use click-stream data exclusively will require a more complex implementation, as they assume that the embedded resources will be prefetched automatically, which requires parsing and may miss resource retrievals that are not easily parsed (such as those that result from JavaScript or Java execution). As a result, in this chapter we have chosen to disregard the type of content and the question of whether it was an embedded resource or not, and instead use all logged requests as data for training and prediction. This approach is also taken by Bestavros *et al.* [Bes96, Bes95], and Fan *et al.* [FJCL99].

4.2.2 Per-user or per-request averaging

As described in Chapter 3, one can calculate average performance on a per-user (macroaverage) or per-request (microaverage) basis. Even though predictions are made on a per-user basis (i.e., on the basis of that user's previous action which is not necessarily the most recent action in the system), we don't always build per-user models. In our experiments, we will build single models (for the typical user) for Web servers and proxies, and only consider per-user models when making predictions at the client. Thus for comparison, we will generally report only per-request averages.

4.2.3 User request sessions

A session is a period of sustained Web activity by a user. In most traces, users have request activities that could be broken into sessions. In fact, during the month of December 1999, the median number of sessions per user was four (where a session boundary is defined by two or more hours without Web access), and has been growing at a rate of 1.4% per month, according to data from 20,000 Internet users collected by

Jupiter Media Metrix [MF01]. In practice, it may be helpful to mark session boundaries to learn when not to prefetch, but alternately, it may be desirable to prefetch the first page that the user will request at the beginning of the next session. We have not analyzed the data sets for sessions — for the purposes of this chapter, each is treated like a set of per-user strings of tokens. Thus, even though a Web log contains interleaved requests by many clients, our algorithms will consider each prediction for a client solely in the context of the requests made by the same client. In addition, it does not matter how much time has passed since the previous request by the same client, nor what the actual request was. If the actual request made matches the predicted request, it is considered a success.

4.2.4 Batch versus online evaluation

The traditional approach to machine learning [Mit97] evaluation is the batch approach, in which data sets are separated into distinct training and test sets. The algorithm attempts to determine the appropriate concept by learning from the training set. The model is then used statically on the test set to evaluate its performance. This approach is used in a number of Web prediction papers (e.g., [ZAN99, AZN99, NZA98, Sar00, SYLZ00, YZL01, ZY01]). While we can certainly do the same (and will do so in one case for comparison), our normal approach will be to apply the predictive algorithms incrementally, in which each action serves to update the current user model and assist in making a prediction for the next action. This matches the approach taken in other Web prediction papers (e.g., [Pad95, PM96, FJCL99, Pal98, PM99]). This model is arguably more realistic in that it matches the expected implementation — a system that learns from the past to improve its predictions in the future. Similarly, under this approach we can test the code against all actions (not just a fraction assigned to be a test set), with the caveat that performance is likely to be poor initially before the user model has acquired much knowledge (although some use an initial warming phase in which evaluation is not performed to alleviate this effect).

When using the prediction system in a simulated or actual system, note that the predictions may cause the user's actual or perceived behavior to change. In prefetching,

the user may request the next document faster if there was no delay in fetching the first (because it was preloaded into a cache). Likewise, a proxy- or server-based model that sends hints about what to prefetch or content to the browser will change the reference stream that comes out of the browser, since the contents of the cache will have changed. Thus, the predictive system may have to adjust itself to the change in activity that was caused by its operation. Alternatively, with appropriate HTTP extensions, the browser could tell the server about requests served by the cache (as suggested in [FJCL99, Duc99]). In future chapters we will incorporate caching effects, and so we will limit ourselves here to models that are built from the same requests as those which are used for evaluation. This model is appropriate for prefetching clients and proxies, as long as they don't depend on server hints (built with additional data).

4.2.5 Selecting evaluation data

Even when using an online per-user predictive model, it will be impossible for a proxy to know when to "predict" the first request, since the client had not connected previously. Note that if the predictions are used for server-side optimization, then a generic prefetch of the most likely first request for any user may be helpful and feasible, and similarly for clients a generic prefetch of the likely first request can be helpful. Likewise, we cannot test predictions made after the user's last request in our sample trace. Thus, the question of which data to use for evaluation comes up. While we will track performance along many of these metrics, we will generally plot performance on the broadest metric — the number of correct predictions out of the total number of actions. This is potentially an underestimate of performance, as the first requests and the unique requests (that is, those requests that are never repeated) are counted, and may become less of a factor over time. If we were to break the data into per-user sessions, this would be a larger factor as there would be more first and last requests that must be handled.

4.2.6 Confidence and support

In most real-world implementation scenarios, there is some cost for each prediction made. For example, the cost can be cognitive if the predictions generate increased

cognitive load in a user interface. Or the cost can be financial, as in prefetching when there is a cost per byte retrieved. Thus, we may wish to consider exactly when we wish to make a prediction — in other words, to not take a guess at every opportunity.

We consider two mechanisms to reduce or limit the likelihood of making a false prediction. They are:

- Thresholds on confidence. Confidence is loosely defined as the probability that the predicted action will occur, and is typically based on the fraction of the number of times that the predicted action occurred in this context in the past. Our methods use probabilistic predictors, and thus each possible prediction has what can be considered an associated probability. By enforcing a minimum threshold, we can restrict the predictions to those that have high expected probability of being correct. Thresholds of this type have been used previously (e.g., [Pad95, PM96, LBO99, DK01]).

- Thresholds on support. Support is strictly the number of times that the predicted action has occurred in this context. Even when a prediction probability (i.e., confidence) is high, that value could be based on only a small number of examples. By providing a minimum support, we can limit predictions to those that have had sufficient experience to warrant a good prediction [JK97, JK98, SKS98, LBO99, FJCL99, PP99b, SYLZ00, DK01, YZL01].[1]

In fielded systems, these two factors may be combined with some function. However, for our studies it is useful to examine them separately.

4.2.7 Calculating precision

Given the ability to place minimum thresholds on confidence and support, the system may choose not to make a prediction at all. Thus, in addition to overall accuracy (correct predictions / all actions), we will also calculate precision — the accuracy of the predictions when predictions are made. However, the exact selection of the denominator

[1] In fact, Su *et al.* [SYLZ00] go further, requiring thresholds not only of page popularity (i.e., support), but also ignoring sequences below some minimum length.

can be uncertain. We note at least two choices: those actions for which a prediction was attempted, and those actions against which a prediction was compared. The former includes predictions for requests that were never received (i.e., made beyond the last request received from a client). Since we cannot judge such predictions, when reporting precision, we will use the latter definition.

4.2.8 Top-n predictions

All of the variations we have described above provide probabilities to determine a prediction. Typically there are multiple predictions possible, with varying confidence and support. Since it may be useful (and feasible) to prefetch more than one object simultaneously, we will explore the costs and benefits of various sets of predictions. As long as additional predictions still exceed minimum confidence and support values, we will generate them, up to some maximum prediction list length of n. The top-n predictions can all be used for prediction and prefetching, and, depending on the evaluation metric, it may not matter which one is successful, as long as one of the n predictions is chosen by the user.

In some systems (e.g., [FJCL99]), there are no direct limits to the number of predictions made. Instead, effective limits are achieved by thresholds on confidence or support, or by available transmission time when embedded in a prefetching system. In this chapter we do not directly limit the number of predictions, but instead consider various threshold values. Limits are typically needed to trade-off resources expended versus benefits gained, and that trade-off depends on the environment within which the predictions are being made. Later, in Chapters 8 and 9 we will embed this prediction system within a simulated environment and will then need to specifically limit the predictions (prefetches) performed because of resource constraints.

4.3 Prediction Techniques

n-grams

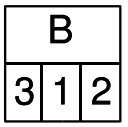
Typically the term *sequence* is used to describe an ordered set of actions (Web requests, in this case). Another name, from statistical natural language processing, for the same ordered set is an n-gram. Thus, an n-gram is a sequence of n items. For example, the ordered pair (A, B) is an example 2-gram (or bigram) in which A appeared first, followed by B.

For prediction, we would match the complete prefix of length $n-1$ (i.e., the current context) to an n-gram, and use the n-gram to predict the last item contained in it. Since there may be multiple n-grams with the same prefix of $n-1$ actions, a mechanism is needed to decide which n-gram should be used for prediction. Markov models provide that means, by tracking the likelihood of each n-gram, but using a slightly different perspective — that of a state space encoding the past. In this approach, we explicitly make the Markov assumption which says that the next action is a function strictly of the current state. In a k-step Markov model, then, each state represents the sequence of k previous actions (the context), and has probabilities on the transitions to each of the next possible states. Each state corresponds to the sequence of $k = n - 1$ actions in a particular n-gram. There are at most $|a|^k$ states in such a system (where $|a|$ is the number of possible actions).

Typically k is fixed and is often small in practice to limit the space cost of representing the states. In our system we allow k to be of arbitrary size, but in practice we will typically use relatively small values (primarily because long sequences are infrequently repeated on the Web).

Markov Trees

To implement various sequence prediction methods, a highly-parameterized prediction system was implemented. We use a data structure that works for the more complex algorithms, and just ignore some aspects of it for the simpler ones.



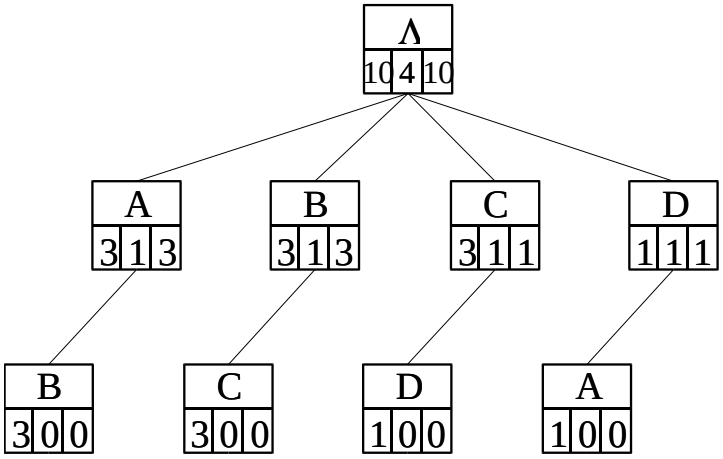
Data recorded is 'B'. The first number (3) is the number of times 'B' has been seen in this context (SelfCount). The second (1) is the count of the number of children of this node (NumChildren). The third (2) is the total number of times its children have been seen (ChildCounts).

Figure 4.1: A sample node in a Markov tree.

In particular, we build a Markov tree [SSM87, LS94, FX00] — in which the transitions from the root node to its children represent the probabilities in a zero-th order Markov model, the transitions to their children correspond to a first order model, and so on. The tree itself thus stores sequences in the form of a trie — a data structure that stores elements in a tree, where the path from the root to the leaf is described by the key (the sequence, in this case). A description of an individual node is shown in Figure 4.1.

Figure 4.2 depicts an example Markov tree of depth two with the information we record after having two visitors with the given request sequences. The root node corresponds to a sequence without context (that is, when nothing has come before). Thus, a zero-th order Markov model interpretation would find the naive probability of an item in the sequence to be .3, .3, .3, and .1, for items A, B, C, and D, respectively. Given a context of B, the probability of a C following in this model is 1. These probabilities are calculated from the node's *SelfCount* divided by the parent's *ChildCounts*.

To make this process clear, we provide pseudocode to build a Markov tree in Figure 4.3, and Figure 4.4 illustrates one step in that process. Given the sequence (A, B, A), we describe the steps taken to update the tree in 4.4a to get the tree 4.4b. All of the suffixes of this sequence will be used, starting with the empty sequence. Given a sequence of length zero, we go to the root and increment *SelfCount*. Given next the sequence of length one, we start at the root and update its *ChildCounts*, and since there is already a child A, update that node's *SelfCount*. Given next the sequence of length two, we start again from the root, travel to child B, and update it's *SelfCount* and find that we need to add a new child. Thus we also update B's *NumChildren*, and its *ChildCounts* as we add the new node. Assuming we are limiting this tree to depth



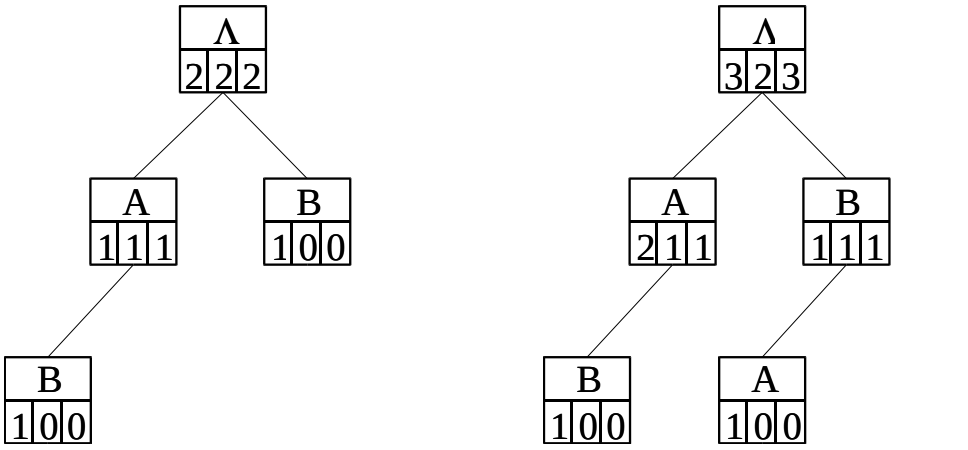
Sequences from two different sessions: (A, B, C, D, A, B, C) followed by (A, B, C).

Figure 4.2: A sample trace and simple Markov tree of depth two built from it.

Given a sequence s, a MaxTreeDepth, and an initial tree (possibly just a root node) t:

```
for i from 0 to min(|s|,MaxTreeDepth)
    let ss be the subsequence containing the last i items from s
    let p be a pointer to t
    if —ss— = 0
      increment p.SelfCount
    else
      for j from first(ss) to last(ss)
        increment p.ChildCounts
        if not-exists-child(p,j)
          increment p.NumChildren
          add a new node for j to the list of p's children
        end if
        let p point to child j
        if j = last(ss)
          increment p.SelfCount
        end if
      end for
    end if
end for
```

Figure 4.3: Pseudocode to build a Markov tree.



(a) Markov tree from sequence (A, B).

(b) Markov tree from sequence (A, B, A).

Figure 4.4: Before and after incrementally updating a simple Markov tree.

two, we have finished the process, and have (b).

In effect, we have built essentially what Laird and Saul [Lai92, LS94] call a TDAG (Transition Directed Acyclic Graph). Like them, we limit the depth of the tree explicitly and limit the number of predictions we make at once. In contrast, they additionally have a mechanism to limit the expansion of a tree by eliminating nodes in the graph that are rarely visited.

Path and point profiles

Thus, after building a Markov tree of sufficient depth, we can use it to match sequences for prediction. Schechter *et al.* [SKS98] describes a predictive approach which is effectively a Markov tree similar to what we have described above. The authors call this approach using *path profiles*, a name borrowed from techniques used in compiler optimization, as contrasted with *point profiles* which are simply bigrams (first order Markov models). The longest path profile matching the current context is used for prediction, with frequency of occurrence used to select from equal-length profiles.

k-th order Markov models

Our Markov tree can in general be used to find k-th order Markov probabilities by traversing the tree from the root in the order of the k items in the current context. Many researchers (e.g., [Pad95, PM96, Bes96, Bes95, JK97, JK98, NZA98, LBO99, Duc99, SR00, FS00, ZY01]) have used models equivalent to first order Markov models (corresponding to trees like that depicted in 4.2) for Web request prediction, but they are also used in many other domains (e.g., the UNIX command prediction from Chapter 3 and hardware-based memory address prefetching [JG99]). Others have found that second order Markov models give better predictive accuracy [SH02, ZAN99], and some, even higher order models (e.g., fourth-order [PP99a]).

During the use of a non-trivial Markov model of a particular order, it is likely that there will be instances in which the current context is not found in the model. Examples of this include a context shorter than the order of the model, or contexts that have introduced a new item into the known alphabet (that is, the set of actions seen so far). Earlier we mentioned the use of the longest matching sequence (path profile) for prediction. The same approach can be taken with Markov models. Given enough data, Markov models of high order typically provide high accuracy, and so using the largest one with a matching context is commonly the approach taken. Both Pitkow and Pirolli [PP99b] and Deshpande and Karypis [DK01] take this route, but also consider variations that prune the tree to reduce space and time needed for prediction (e.g., to implement thresholds on confidence and support, testing on a validation set, and minimum differences in confidence between first and second most likely predictions). Our code allows this approach, and in general provides for a minimum and maximum n-gram length for prediction. Su *et al.* also combine multiple higher-order n-gram models in a similar manner. Interestingly, Li *et al.* [LYW01] argue in contrast that the longest match is not always the best and provide a pessimistic selection method (based on Quinlan's pessimistic error estimate [Qui93]) to choose the context with the highest pessimistic confidence of all applicable contexts, regardless of context length. They show that this approach improves precision as the context gets larger.

PPM

There are, of course, other ways to incorporate context into the prediction mode. PPM, or prediction by partial matching [BCW90, WMB99], is typically used as a powerful method for data compression. It works similarly to the simple Markov model-based approach above, using the largest permissible context to encode the probabilities of the next item. However, it also tracks the probability of the next item to be something that has never been seen before in this context (called the "escape" symbol when used for compression), and thus explicitly says to use the next smaller matching context instead. There are multiple versions of PPM that correspond to different ways of calculating the escape probability: PPM-A, PPM-C, PPM-D, as well as others that we do not consider. Since the next item in the sequence could (and is) given some probability at each of the levels below the longest matching context, we have to examine all matching contexts to sum the probabilities for each candidate prediction (appropriately weighted by the preceding level's escape probability). We do this by merging the current set of predictions with those from the shorter context by multiplying those probabilities from the shorter context by the escape symbol confidence (e) in the longer context, and multiplying those in the longer context by (1 - e). We can calculate various escape probabilities using the counts stored in our Markov tree. A few researchers have used PPM-based models for Web prediction (e.g., [FJCL99, Pal98, PM99]). Actually Fan *et al.* go further, building Markov trees with larger contexts. Instead of using large contexts (e.g., order n) for prediction, they use a context smaller than n (say, m), and use the remaining portion of the tree to make predictions of requests up to $n - m$ steps in advance.

Recency

As we discussed in Chapter 3, the patterns of activity of a user can change over time. The same is true of Web users. As the user's interests change, the activity recorded may also change — there is no need to revisit a site if the user remembers the content, or if the user now has a different concern. Thus, the users themselves provide a source

of change over time in the patterns found in usage logs. There is a second source of change that may be even more significant — changes in content. A user's actions (e.g., visiting a page) depend upon the content of those pages. If the pages change content or links, or are removed, then the request stream generated by the user will also change. Thus we speculate that as long as Web access is primarily an information gathering process, server-side changes will drive changes in user access patterns.

We can attempt to quantify the changes found in some Web logs by using a variant of the IPAM algorithm described in Chapter 3. Recall that it used an *alpha* parameter to emphasize recent activity. However, for processing efficiency, we cannot update all transition probabilities for a given state on each visit. Instead, here we use a simplification: before incrementing the count for a new action, on every $\frac{1}{(1-\text{alpha})}$ visits, we halve the counts of all actions that have been recorded from the previous action. So for example, an alpha of .99 would halve the counts on every 100th visit. Thus we still maintain the property of emphasizing recent events, and in addition, we can bound the maximum count value to $\frac{2}{(1-\text{alpha})}$. This is helpful as it allows us to limit the storage needed to represent the values. Other approaches have been used to emphasize recent activity. Duchamp [Duc99], for example, keeps track of the next page requested by the most recent fifty visitors to a page, thus limiting the effect that a no-longer-popular page sequence can have, but unrealistically requiring cooperation from browsers to self-report link selections.

Additional parameters

In addition to the varieties described above, the prediction system can also vary a number of other parameters that are motivated by the Web domain:

- **Increasing the prediction window** (that is, the window of actions against which the prediction is tested). Typically evaluation is performed by measuring the accuracy of predicting the next request. Instead of only predicting the very next request, we can measure the accuracy when the prediction can match any of the next n requests. This may be useful in matching the utility of preloading a cache as the resource may be useful later.

- **De-emphasizing objects likely to be already cached.** Since the browser has a cache, it is unlikely to make requests for recently requested resources. Thus, the system can reduce the likelihood of predicting recently requested resources.

- **Retaining past predictions.** Likewise, we know that users may go back to pages in their cache, and click on a different item. Our system can retain predictions made in the past (that is, a set of possible next requests and their probabilities) and combine them with current predictions to get (hopefully) a more accurate set of predictions.

- **Considering inexact sequence matching.** Web request sequences are potentially noisy. In particular, users may change the order in which some pages are visited, or skip some resources entirely and still have a successful experience. Thus, our system has been expanded to be able to consider some kinds of inexact sequence matching for building a prediction model.

More detail on these approaches will be provided when we describe experiments using them.

Finally, we should note that while we would like to find or build an IOLA as we described in Chapter 3, for the purposes of these tests we will focus on potential predictive performance (e.g., accuracy) and ignore certain aspects of implementation efficiency. In particular, our codes are designed for generality, and not necessarily for efficiency (especially in terms of space — to make more experiments tractable, we have had to pay some attention to time).

4.4 Prediction Workloads

There are three primary types of Web workloads, corresponding to three viewpoints on the traffic. Here we characterize each of the three and describe the datasets of each type that we will use. A summary of the datasets used can be found in Table 4.1.

Trace name	Described	Requests	Clients	Servers	Time
EPA-HTTP	[Bot95]	47748	2333	1	1 day
Music Machines	[PE98, PE00]	530873	46816	1	2 months
SSDC	[DL00]	187774	9532	1	8 months
UCB-12days	[Gri97, GB97]	5.95M	7726	41836	12 days
UCB-20-clients	Section 4.4.2	163677	20	3755	12 days
UCB-20-servers	Section 4.4.3	746711	6620	20	12 days

Table 4.1: Traces used for prediction and their characteristics.

4.4.1 Proxy

Typically sitting between a set of users and all Web servers, a proxy sees a more limited user base than origin Web servers, but in some cases a proxy may group together users with some overlapping interests (e.g., users of a workgroup or corporate proxy may be more likely to view the same content). It typically records all requests not served by browser caches, but logs may contain overlapping user requests from other proxies or from different users that are assigned the same IP address at different times.

The models built by proxies can vary — from the typical user in a single model to highly personalized models for each individual user. In any case, the predictive model can be used to prefetch directly into its cache, or to provide hints to a client cache for prefetching.

We will use one proxy trace in our experiments. The UC Berkeley Home IP HTTP Traces [Gri97] are a record of Web traffic collected by Steve Gribble as a graduate student in November 1996. Gribble used a snooping proxy to record traffic generated by the UC Berkeley Home IP dialup and wireless users (2.4Kbps, 14.4Kbps, and 28.8Kbps land-line modems, and 20-30Kbps bandwidth for the wireless modems). This is a large trace, from which we have selected the first 12 out of 18 days (for comparison with Fan *et al.* [FJCL99]), for a total of close to six million requests.

4.4.2 Client

The client is one of the two necessary participants in a Web transaction. A few researchers have recorded transactions from within the client browser [CP95, CBC95,

CB97, TG97] making it possible to see exactly what the user does — clicking on links, typing URLs, using navigational aids such as Back and Forward buttons and Bookmarks. It is also typically necessary to log at this level to capture activity that is served by the browser cache, although one potential alternative is to use a cache-busting proxy [Kel01].

Using an individual client history to build a model of the client provides the opportunity to make predictions that are highly personalized, and thus reflect the behavior patterns of the individual user. Unfortunately, logs from augmented browsers (such as those captured by Tauscher and Greenberg [TG97]) are rare. Instead, we will use a subset of requests captured by an upstream proxy (from the UCB dataset) with the understanding that such traces do not reflect all user actions — just those that were not served from the browser cache.

We have extracted individual request histories from the UCB proxy trace. However, not all "clients" identified from proxy traces with unique IP addresses are really individual users. Since proxies can be configured into a hierarchy of proxy caches, we have to be concerned with the possibility that proxy traces could have "clients" which are really proxy caches themselves, with multiple users (or even proxies!) behind them. Likewise, even when a client corresponds to a particular non-proxy system, it may correspond to a mechanized process that repeatedly fetches one resource (or a small set of resources). Since the latter correspond to highly regular request patterns, and the former correspond to overlapping request patterns, we will attempt to avoid the worst of both in the concern for fairness in evaluation. We have ranked the clients by total numbers of requests, and ignored the top twenty, and instead selected the second twenty as the representative set of active users.

The potential performance improvement by prefetching into the client cache is, of course, one of the primary motivations of this dissertation. Building good user models can also help in the development of intelligent interfaces and appropriate history mechanisms.

4.4.3 Server

Servers provide a complementary view on Web usage from that of clients. Instead of seeing all requests made by a user, they see all requests made to one or more servers. Since they only know of requests for particular servers, such logs are unlikely to contain information about transitions to other systems. Technically, the HTTP Referrer header provides information on transitions into a particular server, but these are rarely provided in publicly available logs.

When learning a predictive model, the server could build an individual model for each user. This would be useful to personalize the content on the Web site for the individual user, or to provide hints to the client browser on what resources would be useful to prefetch. One difficulty is the relative scarcity of information unless the user visits repeatedly, providing more data than the typical site visit which commonly contains requests for only a handful of resources. An alternative is to build a single model of the typical user — providing directions that may say that most users request resource B after fetching resource A. When trends are common, this approach finds them. A single model can also provide information that could be used in site re-design for better navigation [PE97, PE00]. In between the two extremes lies the potential for a collaborative filtering approach in which individual models from many users can contribute to suggest actions useful to the current user, as pointed out by Zukerman *et al.* [ZA01]. For the experiments in this dissertation, we generally build a single model based on the traffic seen where the model is stored. Thus, a server would build a single model, which while likely not corresponding to any particular user, would effectively model the typical behavior seen.

In addition to those already described, predictive Web server usage models can also be used to improve server performance through in-memory caching, reduced disk load, and reduced loads on back-end database servers. Similarly, they could be used for prefetching into a separate server-specific reverse proxy cache (with similar benefits).

Server logs are widely available (compared to other kinds of logs), but they have some limitations, as they do not record requests to non-server resources and do not see

responses served by downstream caches (whether browser or proxy).

This chapter will use traces from three servers. The first is the EPA-HTTP server logs [Bot95] which contain close to 48,000 requests corresponding to 24 hours of service at the end of August 1995. The second is two months of usage from the Music Machines website [PE98, PE00], collected in September and October of 1997. Unlike most Web traces, the Music Machines website was specifically configured to prevent caching, so the log represents all requests (not just the browser cache misses). The third (SSDC) is a trace of approximately eight months of non-local usage of a website for a small software development company. This site was hosted behind a dedicated modem, and was collected over 1997 and 1998. Additionally, we have extracted the twenty-most-popular servers from the UCB proxy trace.

4.5 Experimental Results

In this section we will examine the effect of changes to various model parameters on predictive performance. In this way we can determine the sensitivity of the model (or data sets) to small and large changes in parameter values, and to find useful settings of those parameters for the tested data sets.

4.5.1 Increasing number of predictions

In order to help validate our prediction codes, we replicated (to the extent possible) Sarukkai's HTTP server request prediction experiment [Sar00]. This experiment used the EPA-HTTP data set, in which the first 40,000 requests were used as training data, and the remainder for testing.

We set up our tests identically, and configured our prediction codes to use a first order Markov model (i.e., an n-gram size of 2, with no minimum support or confidence needed to predict). Thus unlike the remainder of the experiments presented in this chapter, this experiment builds a model using the initial training data, and freezes it for use on the test data. This is to facilitate comparison with Sarukkai's results. The Static Predictions line in Figure 4.5 corresponds closely to the performance of the

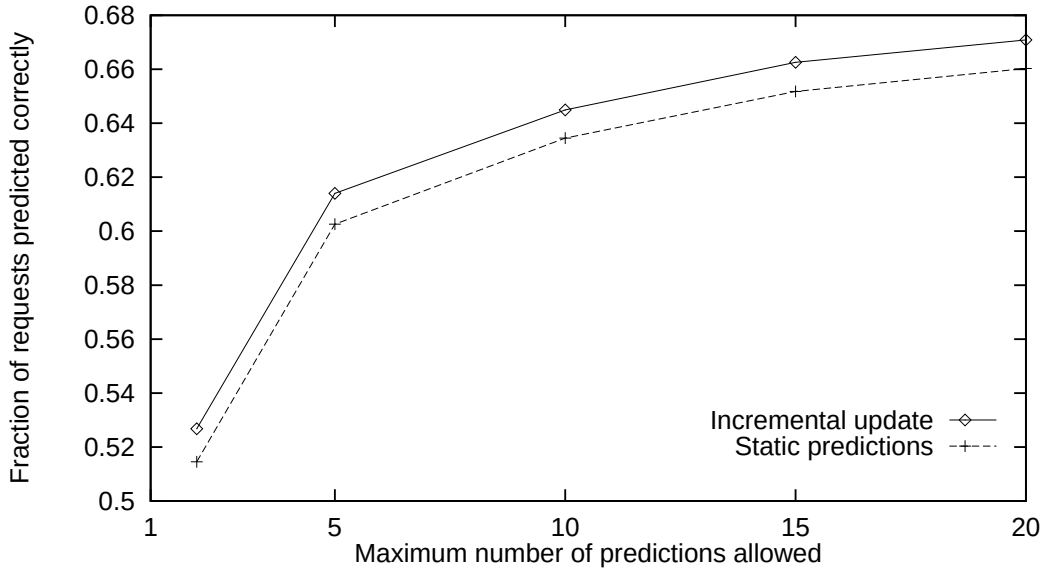


Figure 4.5: Predictive accuracy for the EPA-HTTP data set with varying numbers of allowed predictions.

first test of Markov chains reported in Figure 5 of [Sar00]. Figure 4.5 also shows the approximately 1% absolute increase in predictive accuracy of the same system when it is allowed to incrementally update its model as it moves through the test set.

The EPA-HTTP logs, however, are rather short (especially the test set), and so we consider the performance of prediction for other server logs in subsequent tests and figures. Since they provide a more realistic measure of performance, we will use incremental predictive accuracy throughout the rest of this dissertation.

In Figure 4.6, we examine incremental predictive performance while varying the same parameter (number of predictions permitted) over four other traces (SSDC, UCB servers, Music Machines, and UCB clients). For the initial case of one allowed prediction, we find that performance ranges from slightly over 10% to close to 40% accuracy. As the number of predictions allowed increases to 20, predictive performance increases significantly — a relative improvement of between 45% and 167%. The server-based traces show marked performance increases, demonstrating that the traces do indeed contain sufficient data to include most of the choices that a user might make. The performance of the client-based trace, conversely, remains low, demonstrating that the experience of an individual user is insufficient to include the activities that the user

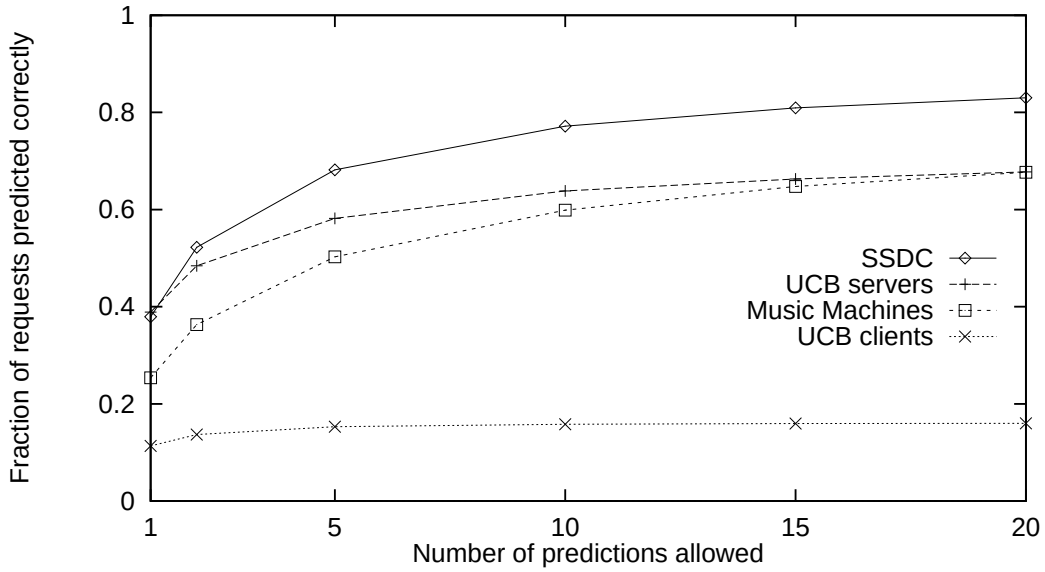


Figure 4.6: Predictive accuracy for various data sets with varying numbers of allowed predictions.

will perform in the future. Finally, note that to achieve such high levels of predictive performance, systems will need to make many predictions, which usually come with some resource cost, such as time, bandwidth, and CPU usage.

4.5.2 Increasing n-gram size

One potential way to improve accuracy is to consider n-grams larger than two. This increase in context allows the model to learn more specific patterns. Figure 4.7 shows the relative improvement (compared to n=2) in incremental predictive accuracy for multiple traces when making just one prediction at each step. Thus, if the accuracy at n=2 were .2, an accuracy of .3 would be a 50% relative improvement, and all traces are shown to have zero improvement at n=2. In each case, the longest n-gram is used for prediction (ties are broken by selecting the higher probability n-gram), and 1-grams are permitted (i.e., corresponding to predictions of overall request popularity) if nothing else matches. The graph shows that adding longer sequences does help predictive accuracy, but improvement peaks and then wanes as longer but rarer (and

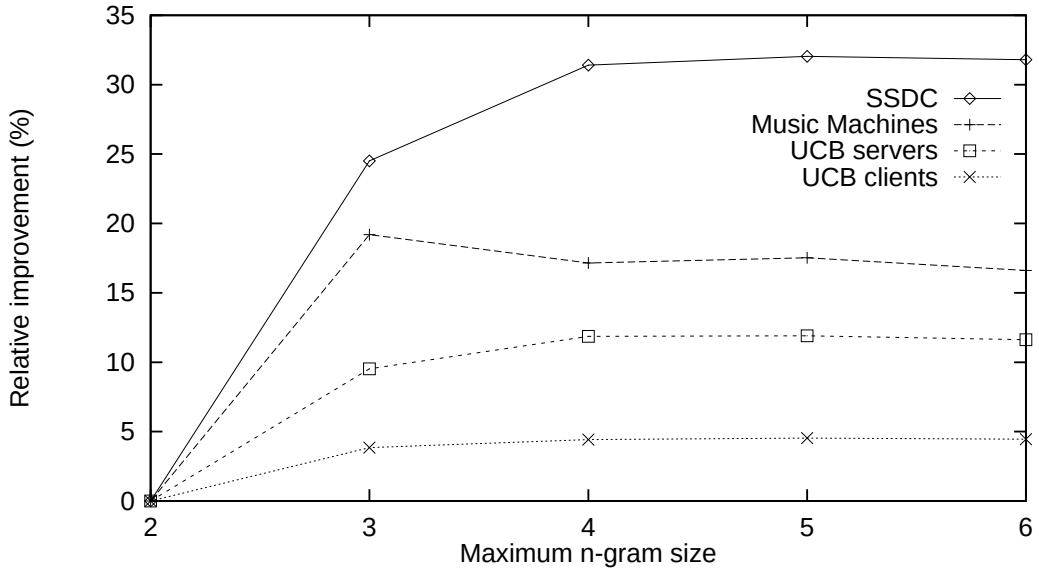


Figure 4.7: Relative improvement in predictive accuracy for multiple data sets as maximum context length grows (as compared to a context length of two).

less accurate) sequences are used for prediction.[2] Thus, the figure shows that larger n-grams themselves can be useful, but Figure 4.8 demonstrates that using the largest n-grams alone for predictions is insufficient. This performance is explained by the fact that longer n-grams match fewer cases than shorter n-grams, and thus able to make fewer correct predictions.

4.5.3 Incorporating shorter contexts

Prediction by partial match provides an automatic way to use contexts shorter than the longest-matching one. In our experiments, we find that the PPM variations are able to perform slightly better than the comparable longest n-gram match. Figure 4.9 plots the performance improvement for three versions of PPM (described in Section 4.3) over the default n-gram approach. As Figure 4.9(a) shows, in the best case, PPM-C gives a close to 3% improvement when only the best prediction is used. When multiple predictions are permitted (as in Figure 4.9(b)), the performance improvement is even less.

[2]As a result, in subsequent tests we will often use an n-gram limit of 4, as the inclusion of larger n-grams typically does not improve performance.

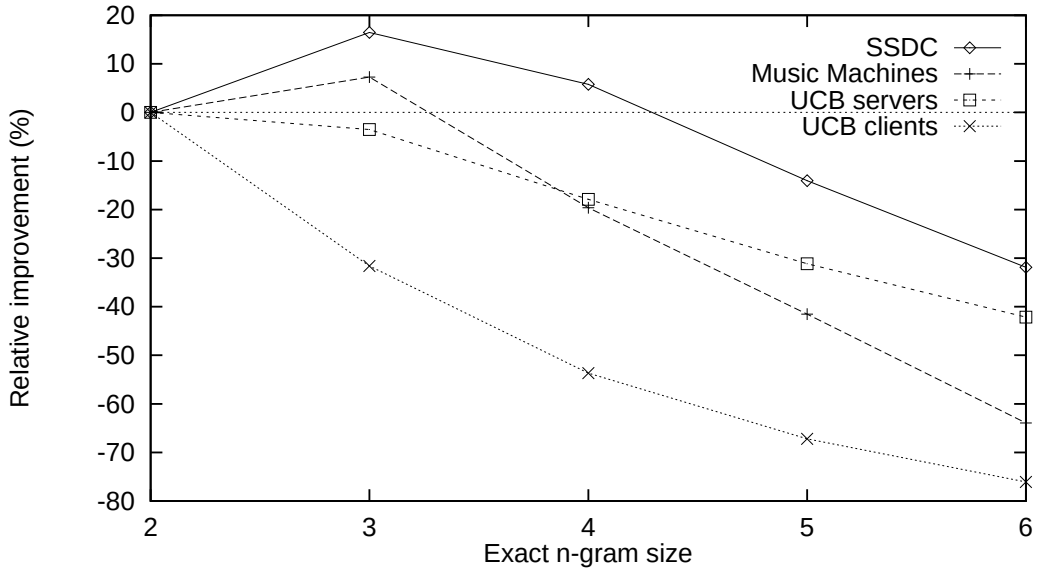


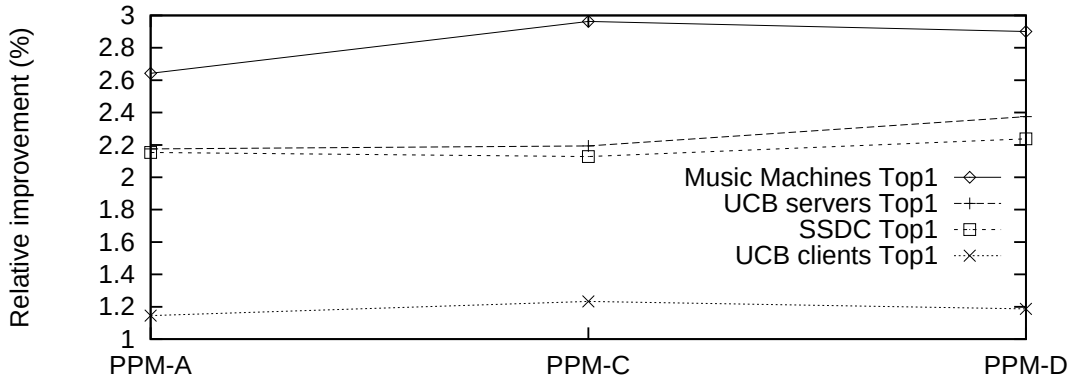
Figure 4.8: Relative improvement in predictive accuracy for multiple data sets using only maximal length sequences (as compared to a context length of two).

However, we can also endow n-grams with the ability to incorporate shorter n-grams. In this version of our system, the n-grams always merge with the results of prediction at the shorter context, with a fixed weight (as opposed to the weight from the dynamically calculated escape probability in the PPM model). Figures 4.10(a) and (b) shows the predictive accuracy for various weightings when using a maximum n-gram of size 6, and minimum of 2. The n-gram model combined with the sample best (out of the five tested) weighted shorter context provides comparable performance (less than .5% absolute difference) to the best PPM model.

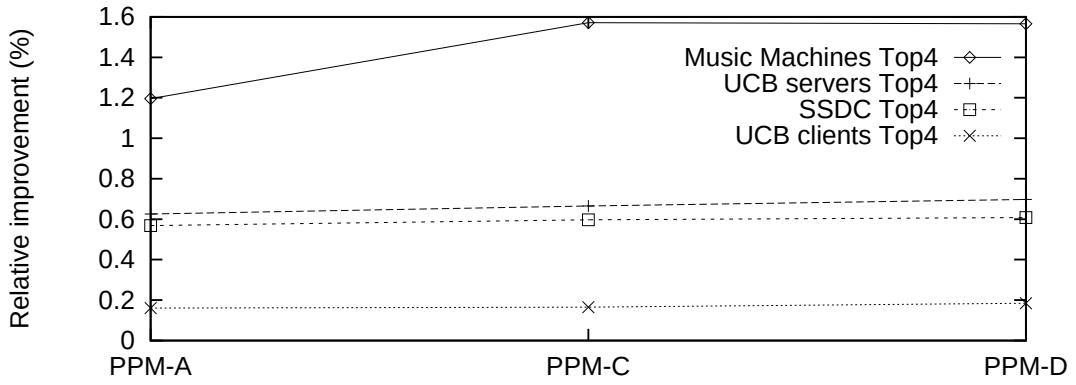
4.5.4 Increasing prediction window

Another way to improve reported accuracy is to allow predictions to match more than just the very next action. In a real Web system, prefetched content would be cached and thus available to satisfy user requests in the future. One can argue that when a prediction does not match the next request, it is incorrect. However, if the prediction instead matches and can be applied to some future request, we should count it as correct. Thus, in this test we apply a Web-specific heuristic for measuring performance.

In Figure 4.11 we graph the relative performance improvement that results when



(a) Top-1 predictions

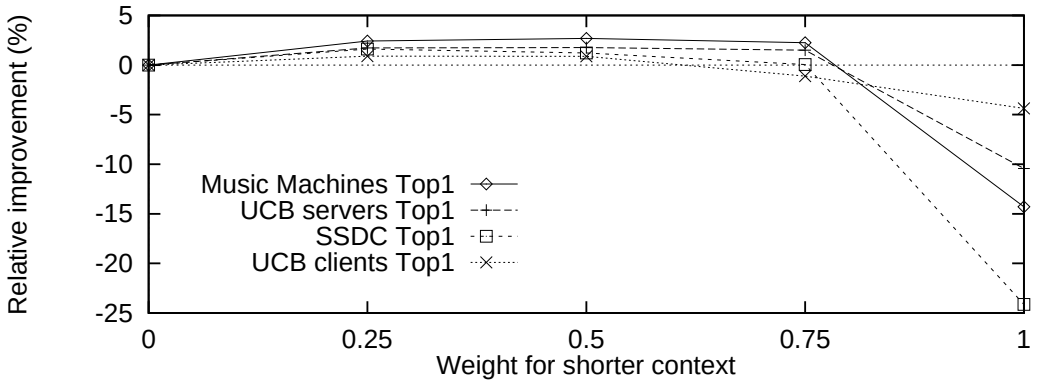


(b) Top-4 predictions

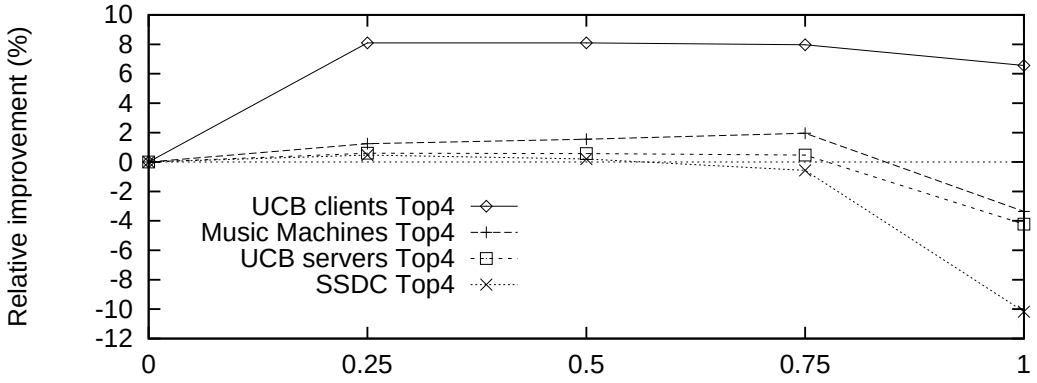
Figure 4.9: Relative improvement in predictive accuracy when using three PPM-based prediction models instead of n-grams.

we allow predictions to match the next 1, 2, 3, 5, 10 and 20 subsequent requests. While at 20 requests performance continues to increase, the rate of growth is tapering. The apparent boost in potential performance suggests that even when the next request is not predicted perfectly, the predicted requests are potentially executed in the near future.

Predictive accuracy, even when using the mechanisms described here, is limited at least by the recurrence rates of the sequences being examined. That is, in this chapter, we only consider predicting from history which means that every unique action cannot be predicted when it is first introduced. Thus, if we were to plot predictive performance on a scale of what is possible, the graphs would be 2-6% absolute percentage points



(a) Top-1 predictions



(b) Top-4 predictions

Figure 4.10: Relative improvement in predictive accuracy when using shorter contexts in n-gram prediction.

higher.

4.5.5 De-emphasizing likely cached objects

One drawback to the above approach is that it breaks the typical *modus operandi* of "predict the next request" with which most machine learning or data compression researchers are familiar. Here we retain the traditional evaluation model, but change instead the prediction based on a Web domain-specific heuristic. Since recently requested objects are likely to be in a user's cache, it is unlikely that the user will request them again, even when such an object has high probability from the model. This will

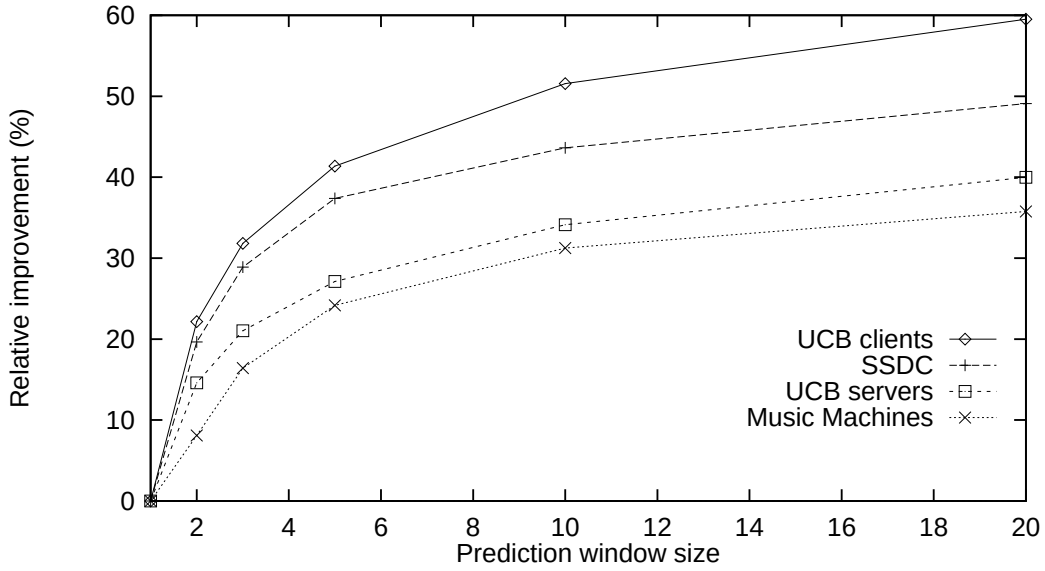


Figure 4.11: Relative improvement in predictive accuracy as the window of actions against which each prediction is tested grows.

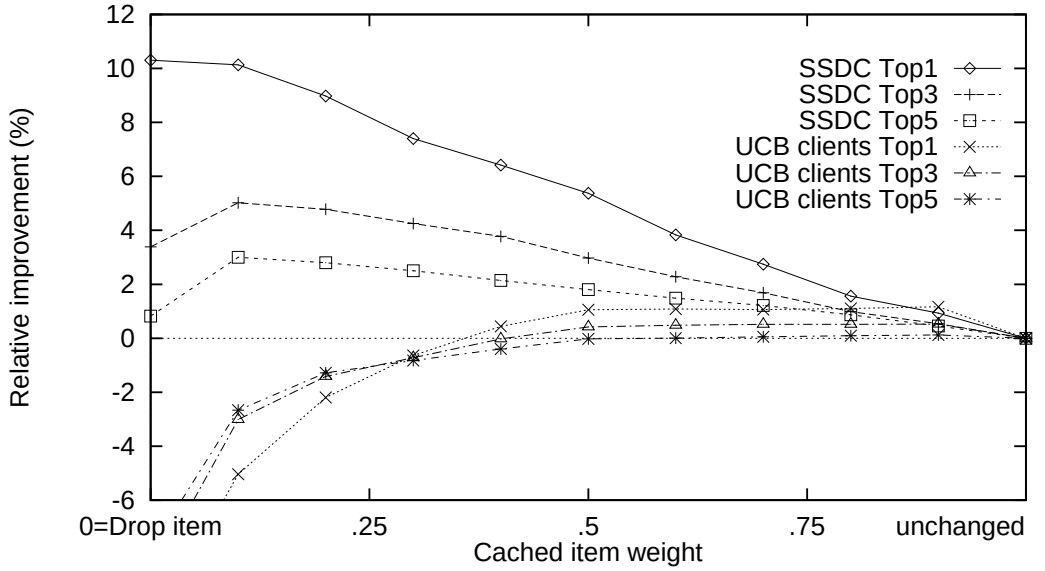


Figure 4.12: Relative changes in predictive accuracy as cached requests are de-emphasized.

be the case for most embedded resources (like images), that are often highly cacheable, but is even possible for HTML pages, when a user takes an atypical route through a Web site. Since it will not be always true (some objects are uncacheable, and so the user will need to retrieve them repeatedly), we are unlikely to want to prevent such

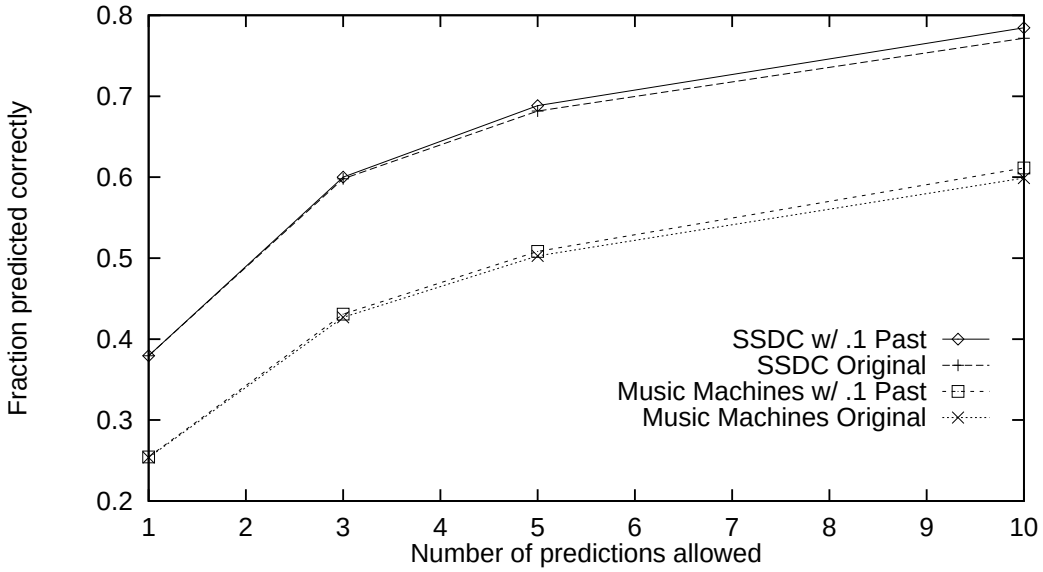


Figure 4.13: Predictive accuracy as past predictions are retained.

near-term repetitions, but we do want to de-emphasize them.

Our system keeps a window of the most recent n requests made per client. We compare a potential prediction to those in that window, and examine the appropriate emphasis given to those predictions by varying the weight given to them. Figure 4.12 shows the performance possible using this technique on two traces and three values for the number of predictions allowed (all within a first order Markov model). It shows that for the SSDC trace, it is beneficial to reduce the likelihood of re-predicting objects that have been recently requested (and thus, likely still in cache). Since the Music Machines trace was generated from a Web site that set all objects to be uncacheable, a test using that Web log would never show improvements (thus, we do not plot it here). The UCB clients trace likely contain references with a smaller likelihood of being cacheable, and thus only show slight improvements for small decreases in weights. The UCB server traces (not shown for clarity) have similar or slightly worse performance than the UCB clients.

4.5.6 Retaining past predictions

Another Web domain-specific improvement can be realized by considering how users often browse the Web. When a particular page (e.g., B) is found to be uninteresting, or has no further value, the user will often use the back button to select a new link (e.g., link C, from page A) of potential value. The act of going back to a previous page is of interest in two ways. First, the revisiting of page A is not recorded in external request logs because the page is typically cached. Second, the links from the earlier page (A) may be selected, even when that page was not the most recently requested one. Thus, since the links of page A may be valuable to the user in later contexts, so might the predictions of next pages from A. We implemented a mechanism to attempt to capture this effect by merging the previous prediction with the current one, subject to a weighting.

In our experiments, we found this to be only minimally useful, and even then only for cases in which multiple predictions are allowed. Figure 4.13 shows that the improvement in predictive performance increases slightly as the number of allowed predictions increases.

4.5.7 Considering inexact sequence matching

In many domains, small variations in the sequences are not significant. HTTP request traces are one of those domains in which some variation is expected — the particular request ordering of a page and its embedded resources is a function of the page HTML source, the browser rendering speed and enabled options (such as Java and JavaScript), and network performance. Additionally, on some visits, a resource may be retrieved, where in other visits, the resource may be cached and thus missing from the request trace.

Therefore, we considered a mechanism that allowed for such variation. In our standard implementation (i.e., for exact sequences) with maximum n-gram size of 4, the arrival of D in the *real* sequence (A, B, C, D) will trigger model updates for all suffixes:

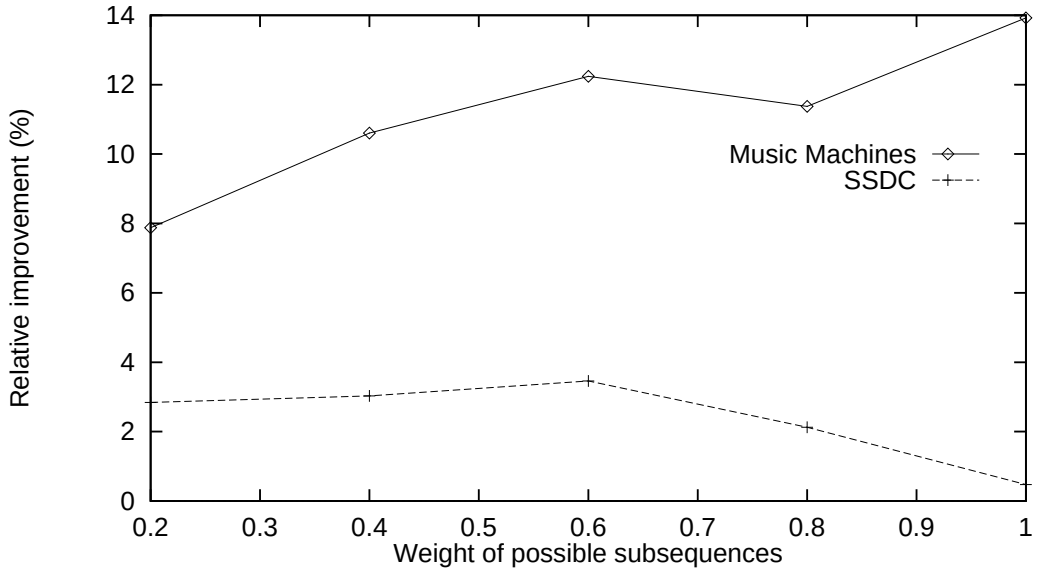


Figure 4.14: Relative improvement in predictive accuracy for n-grams as the weight for non-exact matches is varied.

(D), (C, D), (B, C, D), and (A, B, C, D). In this way, we track how often each subsequence was seen. To allow inexact matching, we generalized the model to incorporate updates corresponding to *unreal* sequences, and continue to use exact matching at the point of prediction.

In our modified implementation, the arrival of D in the real sequence (A, B, C, D) will again trigger model updates for all real suffixes, but also all unreal suffixes that include the last event: (A, D), (B, D), (A, C, D), (A, B, D). The counts for these unreal sequences are updated by adding a weight (typically less than 1).

Figure 4.14 shows relative improvement as the unreal update weight is varied. The scenario plotted is for a fixed-size Markov model (order 4), the top five predictions are used, no confidence or support thresholds, and performance is tested across various prediction window sizes. This approach is particularly beneficial in cases like these, but is not effective (or can even be detrimental) for scenarios that predict from multiple contexts (e.g., PPMs) and are focused on evaluation based strictly on predicting the next action. We should also point out a practical consideration — this approach is slower and requires a much larger prediction model as many of the unreal sequences that are now captured were sequences that did not occur naturally. This factor, combined with

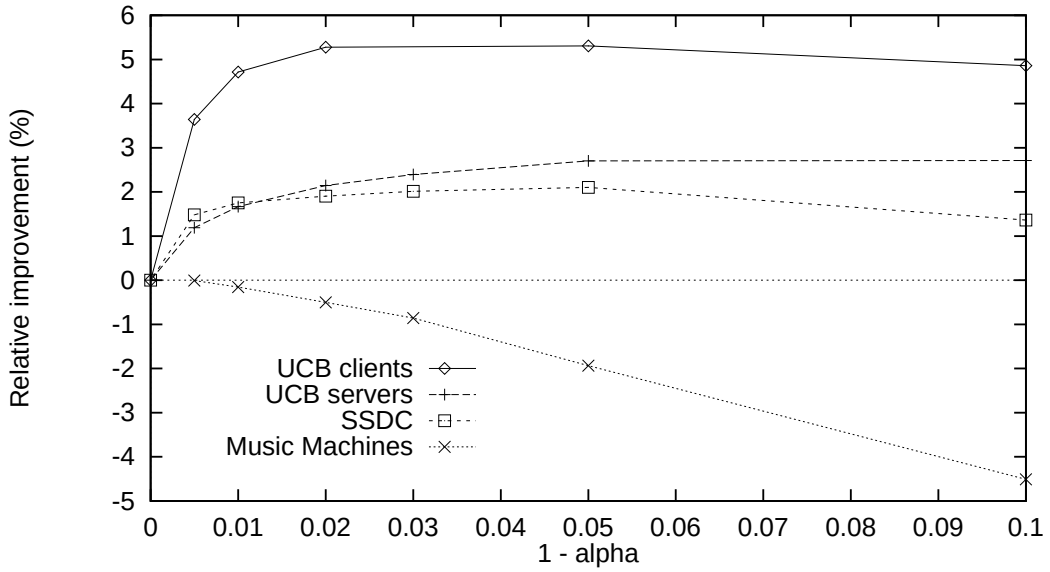


Figure 4.15: Relative change in predictive accuracy as alpha (emphasis on recency) is varied.

improvements of just 3-16%, suggest that this approach is not likely to be useful in practice.

4.5.8 Tracking changing usage patterns

As we mentioned in the discussion of predictive methods in Section 4.3, we can emphasize recent activity by changing the value of alpha. Figure 4.15 shows the effect of alpha on predictive accuracy. In these experiments, we set the maximum and minimum n-gram size to be 4 and 1, respectively. Note that an alpha of 1 means no emphasis on recency, and an alpha of 0 means to use the most recent action as the prediction of the action coming next. Unlike most of the traces in the figure, which show small improvements with an appropriate *alpha*, the Music Machines trace declined slightly in performance as *alpha* decreased. We speculate that concept drift in server logs are predominantly from server changes and not changes in user patterns (from changing user interests). If the Music Machines content did not change over the (two month) time period of the trace, then an emphasis on recent activity is not warranted.

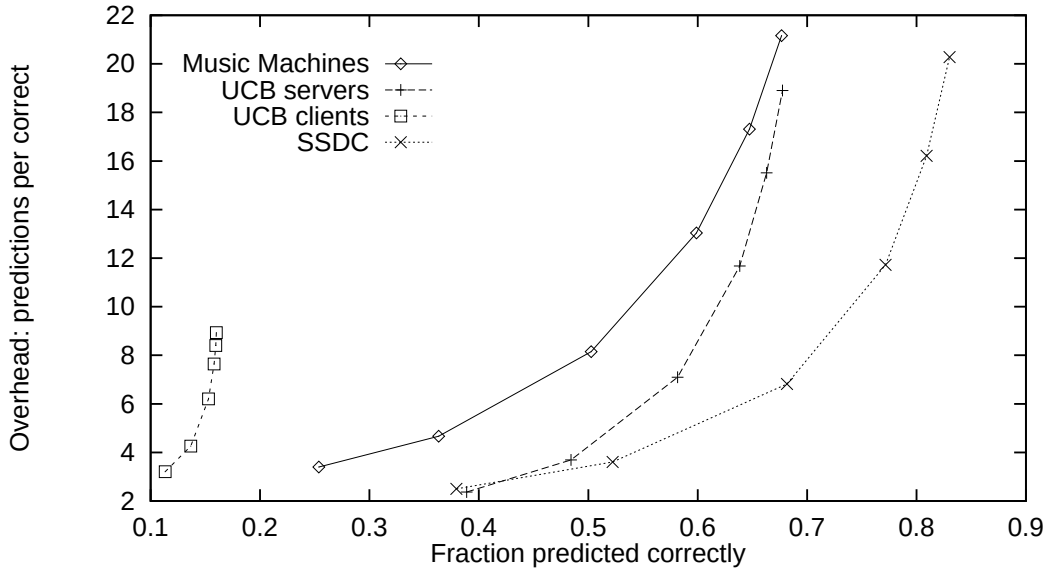


Figure 4.16: Ratio of number of predictions made to those predicted correctly.

4.5.9 Considering mistake costs

Accuracy alone is typically only important to researchers. In the real world, there are costs for mistakes. Figure 4.16 provides one measure of the cost of allowing for additional predictions per action. As the number of allowed predictions per action increases (1, 2, 5, 10, 15, 20) the fraction predicted correctly grows, but the number of predictions needed to get that accuracy also increases significantly. (Note that this example configuration, the same as in Figure 4.6, does not limit predictions to any particular confidence or support, and so likely overestimates the number of predictions made in practice.) In a prefetching scenario in which mistakes are not kept for Music Machines or SSDC, the average per-request bandwidth could be up to twenty times the non-prefetching bandwidth. Note however, that this analysis does not consider the positive and extensive effects of caching, which we will examine in future chapters. The simple point is that there is always a tradeoff — in this case we can see that for increasing coverage, we will require additional bandwidth usage.

Therefore, it is worthwhile to consider how to reduce or limit the likelihood of making a false prediction, as discussed in Section 4.2.6. Figure 4.17 demonstrates the increased precision possible (with lower overall accuracy) as the threshold is increased

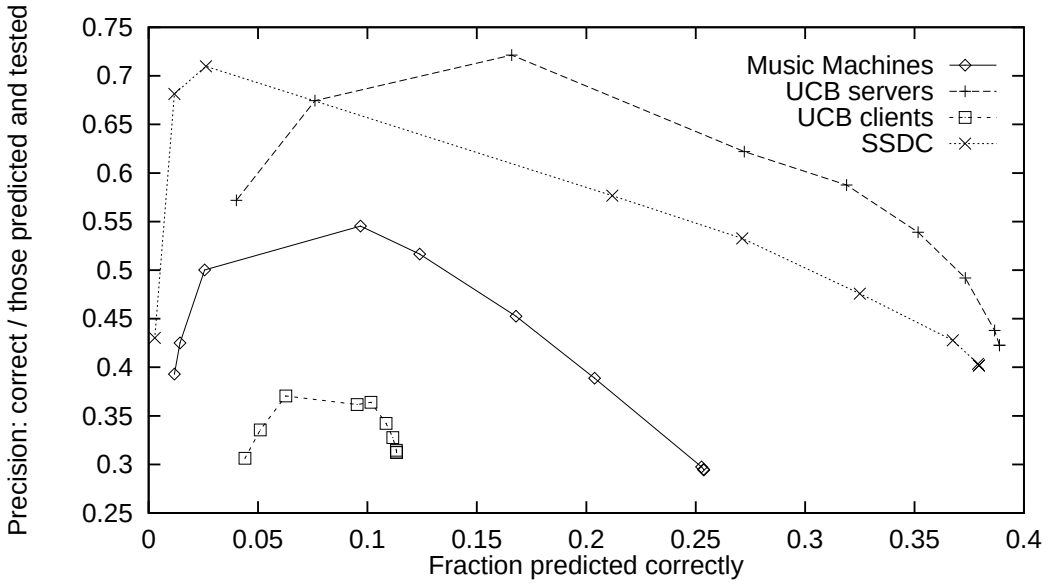


Figure 4.17: Ratio of precision to overall accuracy for varying confidence.

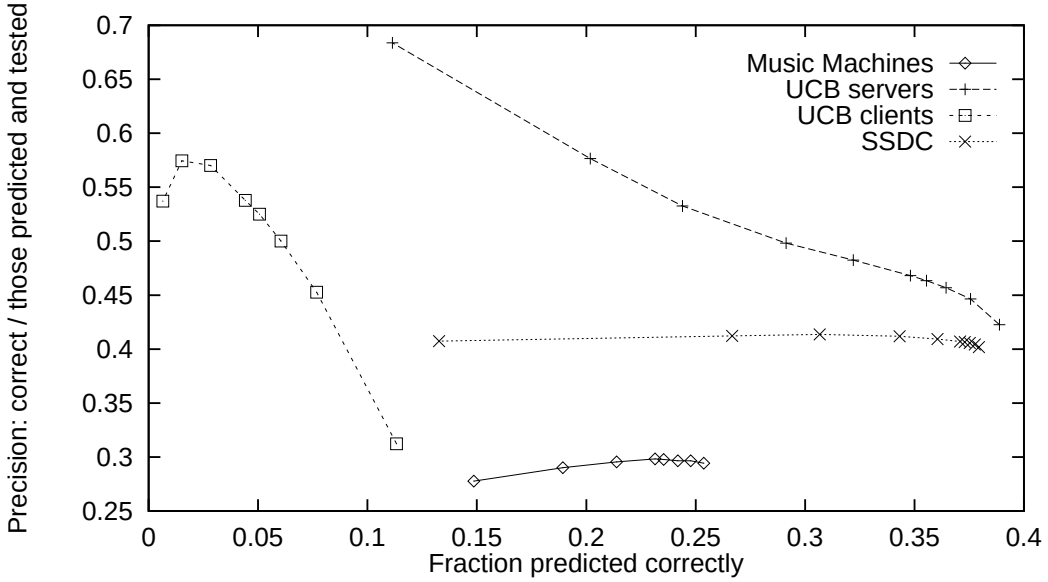


Figure 4.18: Ratio of precision to overall accuracy for varying support.

for an otherwise traditional 1-step Markov model. For each trace, we plot the precision as the minimum confidence threshold varies from .01 to .95. Low threshold values result in higher fractions predicted correctly. The figure demonstrates that while much larger values of precision are possible, they come at a cost of the number of requests predicted correctly. On the other hand, high precision minimizes the rate of mis-predictions,

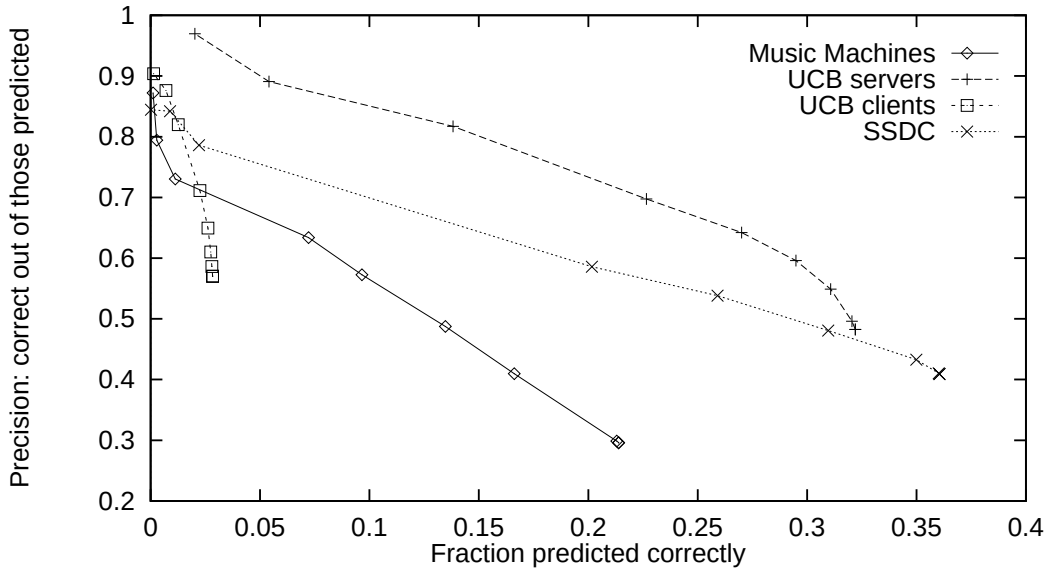


Figure 4.19: Ratio of precision to overall accuracy for a minimum support of 10 with varying confidence.

reducing the costs associated with each correct prediction.

Figure 4.18 shows the same scenario except using a support threshold (ranging from 1 to 50 instances) instead. The effect on increased precision is more variable, as it depends on the number of times a each pattern is found in a trace. Thus, the UCB clients have relatively few repetitions per client, so the curve is shifted further to the left. Neither SSDC nor Music Machines achieve significant growth in precision, but UCB servers with a threshold of 50 instances gets a precision close to .7.

Figure 4.19 provides one combined view — a minimum support of 10 instances combined with a varying confidence threshold. As can be seen, with this combination it is possible to achieve precision that exceeds that possible from either alone. The lesson here is that high accuracy predictions are quite achievable, but are typically applicable to a much smaller fraction of the trace.

4.6 Discussion

We've demonstrated various techniques to improve predictive accuracy on logs of Web requests. Most of these methods can be combined, leading to significant improvements.

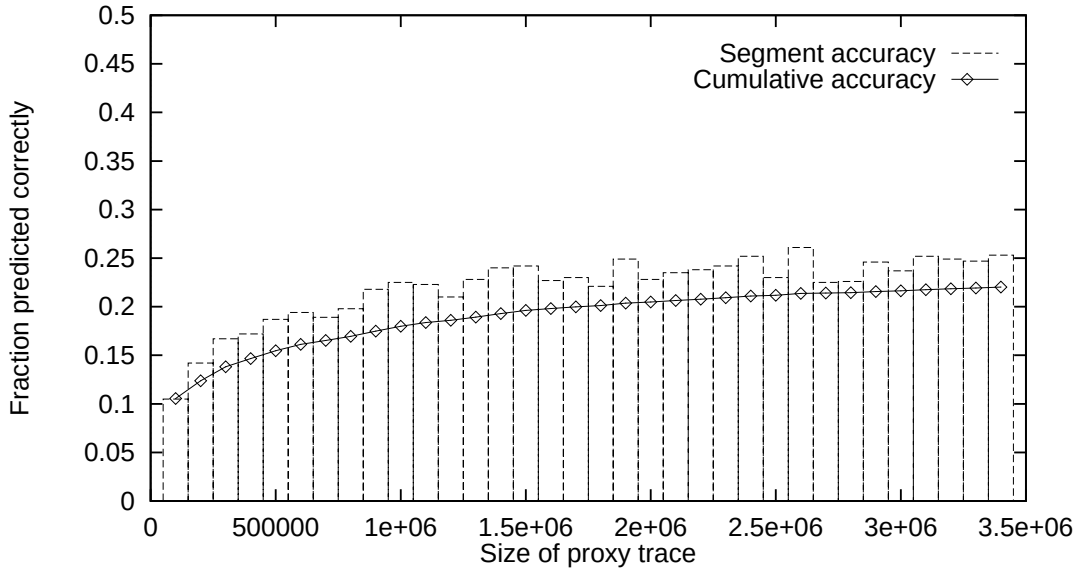


Figure 4.20: Predictive accuracy for a first-order Markov model on UCB proxy data as the data set size varies.

A first-order Markov model is able to get approximately 38% top-1 predictive accuracy on the SSDC trace. A top-5 version can increase that by another 30% to have 68% predictive accuracy. After a few more adjustments (including using PPM, version C, with maximum n-gram 6, reducing predictions likely to be in cache down to 20%, and including past predictions with weight .025), accuracies of 75% are possible. Put another way, this constitutes a 20% reduction in error. And if we consider only those predictions that were actually tested (i.e., ignoring those for which the client made no more requests), we get a prediction accuracy of over 79%. This performance, of course, comes at the cost of a large number of guesses. In this case, we are making five guesses almost all of the time. Alternatively, lower overall accuracy but at higher precision is possible with some modifications (such as the use of confidence and support thresholds).

Some investigators have focused on this issue. Although they use a simplistic model of co-occurrence, Jiang and Kleinrock [JK97, JK98] develop a complex cost-based metric (delay cost, server resource cost) to determine a threshold for prefetching, and use it to consider system load, capacity, and costs. Likewise Zukerman *et al.* [ZA01, AZN99, ZAN99] as well as others (e.g., [SH02, Hor98]) develop decision-theoretic utility models to balance cost and benefit and for use in evaluation.

Finally, the astute reader may have noticed that while we mentioned proxy workloads for completeness back in Section 4.4, we have not included them in the figures shown, as they do not provide significant additional information. While running the UCB client workloads and calculating the client-side predictive accuracy (which is what we report), we also calculated the accuracy that a proxy-based predictor would have achieved. In general, its accuracy was 5-10% higher, relative to the client accuracy (i.e., .1-2% in absolute terms).

For reference, we provide one graph of performance over one large proxy trace. Figure 4.20 shows the short– and long-term trends for predictive accuracy for the first 3.4M requests of the UCB trace. This is for a baseline predictor: a first-order Markov model allowing just one guess without thresholds. It shows that the model continues to learn throughout the trace (as the per 100k segment accuracies are always above the cumulative accuracy). Note that even with millions of requests, the displayed trace only corresponds to about eight days of traffic, and thus is understandable why its performance would not have stabilized yet. However, it is also apparent that the overall mean predictive accuracy will likely converge to an asymptote somewhere less than 25%.

4.7 Summary

In this chapter we have examined in detail the problem of predicting Web requests, focusing on Markov and Markov-like algorithms. We have discussed the problems of evaluation, and have provided a consistent description of algorithms used by many researchers in this domain. We built generalized codes that implement the various algorithms described. The algorithms were tested on the same set of Web traces, facilitating performance comparisons.

We demonstrated the potential for using probabilistic Markov and Markov-like methods for Web sequence prediction. By exploring various parameters, we showed that larger models incorporating multiple contexts could outperform simpler models. We also provided evidence for limited "concept drift" in some Web workloads by testing

a variation that emphasized recent activity. Finally, we demonstrated the performance changes that result when using Web-inspired algorithm changes. Based on these results, we recommend the use of a multi-context predictor (such as PPM, or the multi-context n-gram approach described in this chapter). However, for the Web, it appears that relatively few contexts are needed — n-grams don't need to be much more than 3 or 4. Incorporating some caching-related heuristics may also help, such as de-emphasizing predictions likely to be cached. Finally, the utilization of thresholds on prediction can significantly reduce the likelihood of erroneous predictions.

In this chapter we hinted at some of the effects that client caching can have on performance. This issue will be explored further in later chapters, once we have better mechanisms for evaluating the effects of prefetching techniques.

Chapter 5

The Utility of Web Content

5.1 Introduction

Prediction from history, as described in Chapters 3 and 4, has some limitations. In particular, history cannot be used to predict an action that has never been seen. Thus on the Web, when a user visits a brand new page, no prediction is possible using a model based solely on history. More importantly, a user model only using history ignores a rich set of available content that can guide user actions. Specifically, many of the resources being retrieved contain textual content, which can be analyzed, and Web pages contain hyperlinks to other resources. This link structure can carry significant information implicitly, such as approval, recommendation, similarity, and relevance.

Pitkow and Pirolli [PP97] have observed that "hyperlinks, when employed in a non-random format, provide semantic linkages between objects, much in the same manner that citations link documents to other related documents." This observation can be restated as *most Web pages are linked to others with related content.* This idea, combined with another that says that *text in, and possibly around, HTML anchors describe the pages to which they point*, is the foundation for a usable World-Wide Web. In this chapter we examine to what extent these ideas hold by empirically testing whether topical locality mirrors spatial locality of pages on the Web. In particular, we demonstrate that this semantic linkage, as approximated by textual similarity, is measurably present in the Web. We find that the likelihood of linked pages having similar textual content to be high; the similarity of sibling pages increases when the links from the parent are close together; titles, descriptions, and anchor text represent at least part of the target page; and that anchor text may be a useful discriminator among unseen child pages. These results present the foundations necessary for the success of many

proposed techniques (e.g., [PPR96, Spe97]) and implemented Web systems, including search and meta-search engines (e.g., [Inf02b, SE95, SE97, DH97c, HD97, Inf02a, ZE98, ZE99, LG98b, LG98a, Goo02, Alt02, Ink02, Fas02, McB94, Lyc02b], focused crawlers (e.g., [CGMP98, CvdBD99, BSHJ+99, Men97, MB00, Lie97, RM99], linkage analyzers (e.g., [BFJ96, DH99, KRRT99, FLGC02, CDR+98, CDG+99, BH98b, BP98, DGK+99, IBM00, Kle98], and intelligent Web agents (e.g., [AFJM95, JFM97, Mla96, Lie95, Lie97, MB00, BS97, LaM96, LaM97, Lie97, PP97, Dav99a]), as we will describe below.

While in this dissertation we are concerned with extracting information from Web content to guide user action prediction, the utility of Web content is much broader. Thus in this chapter we consider extensively where and how long-held intuitions about the nature of Web linkages and the content from and to which they point can be used by providing examples from a number of areas. We then describe experimental work to collect and analyze a Web dataset and show how it supports the common assumptions about the nature of the Web. While we provide some intuitions and empirical data to show why content can be helpful for usage modeling in the section below, we postpone our experimental examination of how it can help with prefetching to the following chapter.

5.2 Motivation

It is commonly believed that most links on the Web connect pages with related content. This assumption, along with another that says that text in and around Web links describe the pages to which they point, is the foundation for a usable World-Wide Web. They make browsing possible, since users would not follow links if those links were unlikely to point to relevant and useful content. These ideas have also been noticed by researchers and developers, and are implicit in many of the search, crawling, and navigation services found on the Web today.

Expanding on this, intuition suggests that content matters when characterizing Web traffic because:

- **Web usage is rarely random.** Web users typically have some kind of information-seeking goal, even if it is fleeting (i.e., when surfing from one topic to another). In particular, Web users have an expectation of what information is available behind a link, based at least in part on the *content* of the pointing page.

- **Web users click on links.** More than television where most viewers passively receive information for significant periods of time, Web users actively click on links to direct their experience and decide what content to receive. The links they click on are part of the content they get, and have a tremendous influence on where they go next. Out of the billions of URLs that a user could type into a browser at any point in time, the user will most often choose instead to click a link from the page being viewed.

  To illustrate, we analyzed the Web server logs from a small software development company (188 thousand hits over an 8 month period), and found that approximately 83.4% of the 48,000 hits from a user's choice (i.e., not from spiders or inline images) had a Referer tag that showed the URL of the referring page. These hits represent the Web link selections (rather than, for example, URLs typed in by hand or selected from a set of bookmarks). This result coincides nicely with the statistic reported by Tauscher and Greenberg [TG97] (who measured client actions from a browser's perspective) which said that 82.7% of all open document actions were generated by selections within the page (5% bookmarks, 6.8% typed in, etc.). In an earlier study, Catledge and Pitkow [CP95] found an even higher percentage (over 90% of open document requests were from selecting a hyperlink in a document). In Chapter 6 we will additionally show that a large portion of the clicks will be to links on pages that have been requested in the recent past.

Thus, for evaluation of many Web algorithms, typical Web usage logs are insufficient. As we describe elsewhere [Dav99c], standard logs generated by Web servers or proxies do not provide enough data for accurate analysis of user activity. For example, they do not include many of the headers which specify cacheability, or whether the request is to validate a cached copy. Further, Web resources change rapidly [GS96, Kah97, DFKM97,

CM99] and so URLs cited in the logs are likely to have changed or disappeared entirely in a short period of time. Thus, in order to do offline analysis of content-based prefetching methods, a full-content trace needs to be collected (which we do, and use in experiments we report later in Chapter 6).

Intuitions about Web content locality are so fundamental that in many cases they are not mentioned, even though without them the systems would fail to be useful. When mentioned explicitly (as in [MB00, DH99, GKR98, BS97, Kle98, BP98, CDR+98, Ami98]), their influence is measured implicitly, if at all. This chapter is an attempt to rectify the situation — we wish to measure the extent to which these ideas hold.

This chapter primarily addresses two topics: it examines the textual similarity of pages near one another in the Web, and the related issue of the quality of descriptions of Web pages. The former is most relevant to focused Web crawlers and to search engines using link analysis (as we discuss in Section 5.3.2), while the latter is primarily of use to Web indexers (Section 5.3.1), meta-search tools (Section 5.3.3), and to human browsers of the Web since users expect to find pages that are indeed described by link text (when browsing the Web) and to find pages that are characterized accurately by the descriptive text presented by search engine results. We show empirical evidence of topical locality in the Web, and of the value of descriptive text as representatives of the targeted page. In particular, we find that the likelihood of linked pages having similar textual content to be high; that the similarity of sibling pages increases when the links from the parent are close together; that titles, descriptions, and anchor text represent at least part of the target page; and that anchor text may be a useful discriminator among unseen child pages.

For the experiments described in this chapter, we select a set of pages from the Web and follow a random subset of the links present on those pages. This provides us with a corpus in which we can measure the textual similarity of nearby or remote pages and explore the quality of titles, descriptions, and anchor links with respect to their representation of the document so described. In the next section, we will describe some applications for this work, giving examples from many areas, including Web indexers, search ranking systems, focused crawlers and Web prefetchers. We will then describe

our experimental methodology, present the results found, discuss related work, and conclude with a summary of our findings.

5.3 Applications

The World-Wide Web is not a homogeneous, strictly-organized structure. While small parts of it may be ordered systematically, many pages have links to others that appear almost random at first glance. Fortunately, further inspection generally shows that the typical Web page author does not place random links in her pages (with the possible exception of banner advertising), but instead tends to create links to pages on related topics. This practice is widely believed to be typical, and as such underlies a number of systems and services on the Web, some of which are mentioned below.

Additionally, there is the question of describing the Web pages. While it is common for some applications to just use the contents of the Web pages themselves, there are situations in which one may have only the titles and/or descriptions of a page (as in the results page from a query of a typical search engine), or only the text in and around a link to a page. A number of systems (such as content-based prefetchers, included below in Section 5.3.5) could or do assume that these "page proxies" accurately represent the pages they describe, and we include some of those systems below.

5.3.1 Web indexers

A Web indexer takes pages crawled from the Web and generates an inverted index of those pages for later searching. Companies operating search engine indices including Google [Goo02], AltaVista [Alt02], Inktomi [Ink02], FAST [Fas02], etc., all have indexers of some sort that perform this function. However, many search engines once indexed much less than the full text of each page. An early search engine called the WWW Worm [McB94], for example, indexed titles and anchor text. Lycos [Lyc02b], at one time, only indexed the first 20 lines or 20% of the text [KABL96]. More recently Google started out by indexing just the titles [BP98]. Today it is common for the major engines to index not only all the text, but also the title of each page. Smaller services such as

research projects or intranet search engines may opt for reduced storage and index less. What is less common is the indexing of HTML META tags containing author-supplied keywords and descriptions. Some search engines will index the text of these fields, but others do not [Sul02], citing problems with search engine "spamming" (that is, some authors will place keywords and text that are not relevant to the current page but instead are designed to draw traffic as a result of popular search terms).

Likewise, while indexers typically include anchor text (text within and/or around a hypertext link) as some of the terms that represent the page on which they are found, most do not use them as terms to describe the page referenced. One significant exception is Google, which does index anchor text. By doing so, Google is able to present target pages to the user that have not been crawled, or have no text, or are redirected to another page. One drawback, however, is that this text might not in fact be related to the target page. One publicized example was the query "more evil than satan himself" which, at least for a while, returned Microsoft as the highest ranked answer from Google [Sul99, Spr99].

So, for search engine designers, we want to address the questions of how well anchor text, title text, and META tag description text represent the target page's text. Even when title and descriptions are indexed, they may need to be weighted differently from terms appearing in the text of a page. Our goal (among others) is to provide some evidence that may be used in making decisions about whether to include such text (in addition to or instead of the target text content) in the indexing process.

5.3.2 Search ranking and community discovery systems

Traditionally, search engines have used text analysis to find documents relevant to a query. Today, however, many Web search engines incorporate additional factors of user popularity (based on actual user traffic), link popularity (that is, how many other pages link to the page), and various forms of page status calculations. Both link popularity and status calculations depend, at least in part, on the assumption that page authors do not link to random pages. Presumably, link authors want to direct their readers to pages that will be of interest or are relevant to the topic on the

current page. The link analysis approaches used by Clever [IBM00, Kle98] and others [BH98b, BP98, DGK+99] depend on having a set of interconnected pages that are both relevant to the topic of interest and richly interconnected in order to calculate page status. Additionally, some [CDR+98, CDG+99] use anchor text to help rank relevance of a query to communities discovered from the analysis. The similar link analysis performed in community discovery research (e.g., [KRRT99, FLGC02]) make the same assumptions of the relatedness of linked pages, but use the presence of patterns of such links to reinforce their models.

LASER [BFJ96] demonstrates a different use of linkage information to rank pages. It computes the textual relevance, and then propagates that relevance backward along links that point to the relevant pages. The goal is to enable the engine to find pages that are good starting points for automated crawling, even if those pages don't rank highly based on text alone.

Our analysis may help to explain the utility of anchor text usage, as well as show how likely neighboring pages are to be on the same topic.

5.3.3 Meta-search engines

Meta-search engines (e.g., MetaCrawler [Inf02b, SE95, SE97], SavvySearch [DH97c, HD97] which is now incorporated into Search.com, and DogPile [Inf02a]) are search services that do not search an index of their own, but instead collect and compile the results of searching using other engines. While these services may do nothing more than present the results they obtained for the client, they may want to attempt to rank the results or perform additional processing. Grouper [ZE98, ZE99], for example, performs result clustering. While early versions of MetaCrawler and Inquirus [LG98b, LG98a] fetch all documents for analysis on full-text, simpler versions (perhaps with little available bandwidth) might decide to fetch only the most likely pages for further analysis. In this case, the meta-engine has only the information provided by the original search engines (usually just URL, title, and description), and the quality of these page descriptors is thus quite important to an after-the-fact textual ranking or clustering of the pages.

5.3.4 Focused crawlers

Focused crawlers are Web crawlers that follow links that are expected to be relevant to the client's interest (e.g., [CvdBD99, BSHJ+99, Men97, MB00, Lie97, RM99] and the query similarity crawler in [CGMP98]). They may use the results of a search engine as a starting point, or they may crawl the Web from their own dataset. In either case, they assume that it is possible to find highly relevant pages using local search starting with other relevant pages. While not concerned with crawling, Dean and Henzinger [DH99] demonstrate a similar approach to find related pages for search engine results.

Since focused crawlers may use the content of the current page, or anchor text to determine whether to expand the links on a page, our examination of nearby page relevance and anchor text relevance may be useful.

5.3.5 Intelligent browsing agents

There have been a variety of agents proposed to help people browse the Web. Many of those that are content-based depend on the contents of a page and/or the text contained in or around anchors to help determine what to suggest to the user (e.g., [AFJM95, JFM97, Mla96, Lie95, Lie97, MB00, BS97, LaM96, LaM97]) or to prefetch links for the user (e.g., [Lie97, PP97, Dav99a]).

By comparing the text of neighboring pages, we can build models for future estimates of the relevance of unseen pages neighboring some current one. We also find out how well anchor text describes the targeted page.

5.4 Experimental Method

In this section we describe our data, explain how it was collected, identify the mechanism we use to measure similarity, and detail the experiments performed.

5.4.1 Data set

5.4.1.1 Initial Data Set

Ideally, when characterizing the pages of the WWW, one would choose a random set of pages selected across the Web. Unfortunately, while the Web is currently estimated to contain billions of pages, no one entity has a complete enumeration. Even the major search engines only know of a fraction of the Web, and the pages retained in those datasets are biased samples of the Web [LG98c, LG99]. As a result, the unbiased selection of a random subset of the Web is an open question [BB98], although some progress has been made [HHMN00].

Accordingly, the data set used as the starting points in this chapter were selected at random from a subset of the Web. We randomly selected 100,000 pages out of the approximately 3 million pages that our local research search engine (DiscoWeb [DGK+99]) had crawled and indexed by early December 1999. The pages in the DiscoWeb dataset at that time were generated primarily from the results of inquiries made to the major search engines (such as HotBot [Lyc02a] and AltaVista) plus pages that were in the neighborhood of those results (i.e., direct ancestors or descendants of pages in those results). Thus, selecting pages from this dataset will bias our sample toward pages in the neighborhood of high-ranking English-language pages (that is, pages near other pages that have scored highly on some query to a search engine).

5.4.1.2 Remaining Data Set

From the initial data set, we randomly selected one outgoing link per page and retrieved those pages. We also randomly reselected a different outgoing link per page (when the number of unique links was > 1) and fetched those pages as well. The latter set was used for testing anchor text relevance to sibling pages and to measure similarity between sibling pages.

The		ACM1		Conference	will	take	you	beyond	cyberspace
1	-	0	-	1	2	3	4	5	6
..... (Anchor text only)									
..... Anchor + terms up to distance 1									
..... Anchor + terms up to distance 2									
..... Distance 5									

Table 5.1: Illustration of different amounts of text surrounding an anchor.

5.4.1.3 Retrieval, Parsing, and Textual Extraction

The pages were retrieved using the Perl LWP::UserAgent library, and were parsed with the Perl HTML::TreeBuilder library. Text extraction from the HTML pages was performed using custom code that down-cased all terms and replaced all punctuation with whitespace so that all terms are made strictly of alphanumerics. Content text of the page does not include title or META tag descriptions, but does include alt text for images. URLs were parsed and extracted using the Perl URI::URL library plus custom code to standardize the URL format (down-casing host, dropping #, etc.) to maximize matching of equivalent URLs. The title (when available), description (when available), and non-HTML body text were recorded, along with anchor text and target URLs. The anchor text included the text within the link itself (i.e., between the <a> and), as well as surrounding text (up to 20 terms but never spanning another link). See Table 5.1 for an example. Term vector representation [SM83] was used for each sample of text.

5.4.2 Textual similarity calculations

To perform the textual analysis, we calculated a simple measure. Elsewhere [Dav00c] we report on the similar results with additional measures. The measure has the following useful properties:

- The measure can be applied to any pair of documents,
- The measure produces scores in the range [0..1], and
- Identical documents generate a score of 1 while documents having no terms in common generate a score of 0.

TFIDF [SM83, SB98], which multiplies the term frequency of a term in a document by the term's inverse document frequency, was selected for its widespread use and long history in information retrieval. Intuitively it gives extra weight to terms heavily used by a document, but penalizes terms that are found in most documents. Given term w_i in document X, the specific formulas used were:

$$\text{TFIDF}(w_i, X) = \frac{\text{TF}(w_i, X) * \text{IDF}(w_i)}{\sqrt{\sum_{all\ w}(\text{TF}(w, X) * \text{IDF}(w))^2}}$$

where

$$\text{TF}(w, X) = \lg(\text{number of times } w \text{ appears in } X + 1)$$

and

$$\text{IDF}(w) = \lg(\frac{\text{number of docs} + 1}{\text{number of docs with term } w})$$

Note that the IDF values are calculated from the documents in the combined retrieved sample, not over the entire Web.

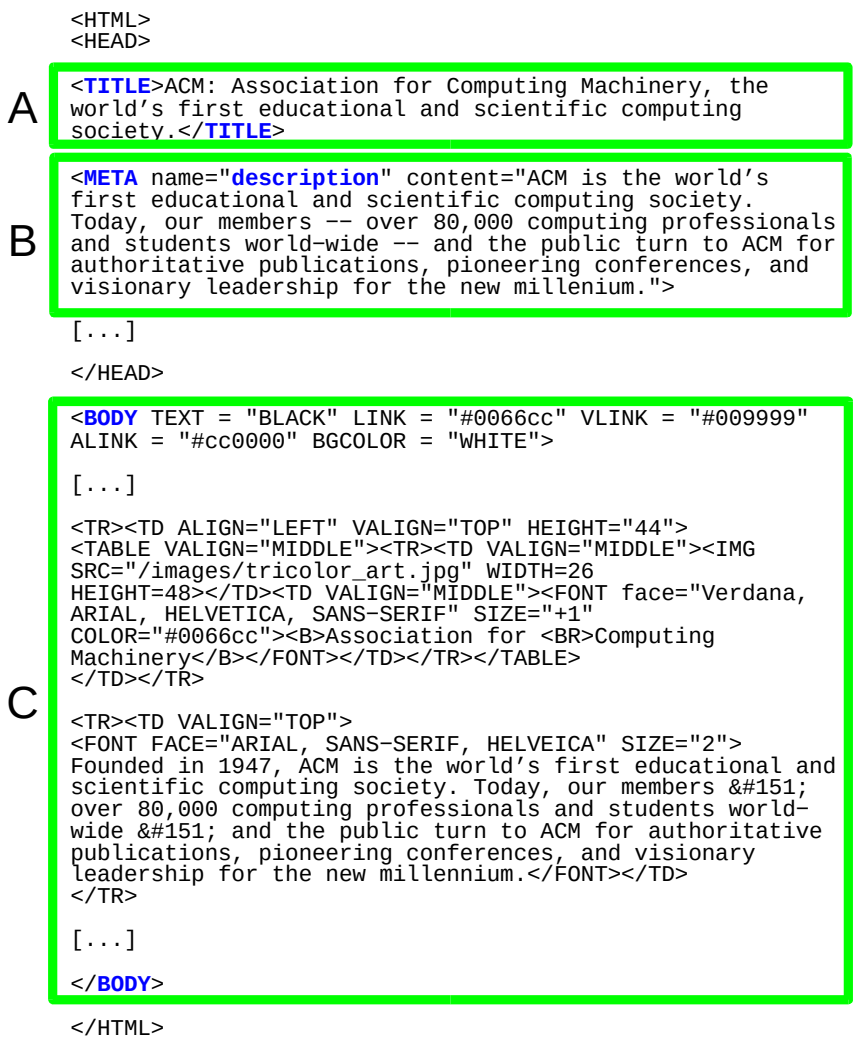
The TFIDF values for each term of the document are then normalized (divided by the sum of the values) so that the values of the terms in a document sum to 1. Each document then is represented as a vector of values corresponding to the TFIDF values for each possible term. To calculate the similarity between document vectors X and Y we use the cosine measure.

$$\text{TFIDF-Cos}(X, Y) = \frac{\sum_{all\ w} \text{TFIDF}(w, X) * \text{TFIDF}(w, Y)}{\sqrt{\sum_{all\ w} \text{TFIDF}(w, X)^2 * \sum_{all\ w} \text{TFIDF}(w, Y)^2}}$$

5.4.3 Experiments performed

The primary experiments performed include measuring the textual similarity:

- of the title to the text in the body of the page, and of the description to the body (see Figure 5.1)
- of a page and one of its children
- of a page and a random page




```
<HTML>
<HEAD>
<TITLE>ACM: Association for Computing Machinery, the
world's first educational and scientific computing
society.</TITLE>
<META name="description" content="ACM is the world's
first educational and scientific computing society.
Today, our members -- over 80,000 computing professionals
and students world-wide -- and the public turn to ACM for
authoritative publications, pioneering conferences, and
visionary leadership for the new millenium.">
[...]
</HEAD>
<BODY TEXT = "BLACK" LINK = "#0066cc" VLINK = "#009999"
ALINK = "#cc0000" BGCOLOR = "WHITE">

[...]

<TR><TD ALIGN="LEFT" VALIGN="TOP" HEIGHT="44">
<TABLE VALIGN="MIDDLE"><TR><TD VALIGN="MIDDLE"><IMG
SRC="/images/tricolor_art.jpg" WIDTH=26
HEIGHT=48></TD><TD VALIGN="MIDDLE"><FONT face="Verdana,
ARIAL, HELVETICA, SANS-SERIF" SIZE="+1"
COLOR="#0066cc"><B>Association for <BR>Computing
Machinery</B></FONT></TD></TR></TABLE>
</TD></TR>

<TR><TD VALIGN="TOP">
<FONT FACE="ARIAL, SANS-SERIF, HELVEICA" SIZE="2">
Founded in 1947, ACM is the world's first educational and
scientific computing society. Today, our members &#151;
over 80,000 computing professionals and students world-
wide &#151; and the public turn to ACM for authoritative
publications, pioneering conferences, and visionary
leadership for the new millennium.</FONT></TD>
</TR>

[...]

</BODY>
</HTML>
```

Figure 5.1: Sample HTML page from `www.acm.org`. We record text found in section A (title), section B (description), and section C (body text).

- of two pages with the same immediate ancestor (i.e., between siblings) and with respect to the distance in the parent document between referring URLs
- of anchor text and the page to which it points
- of anchor text and a random page
- of anchor text and a page different from the one to which it points (but still linked from the parent page)

We additionally measured lengths of titles, descriptions (text provided in the description META tag of the page), anchor texts, and page textual contents. We also examined how often links between pages were in the same domain, and if so, the same host, same directory, etc.

We also performed experiments with stop word elimination and Porter term stemming [Por97], with similar results, and thus they are not included below (but can be found in a technical report [Dav00b]). No other feature selection was used (i.e., all terms were included).

5.5 Experimental Results

5.5.1 General characteristics

For a baseline, we first consider characteristics of the overall dataset. Out of the initial 100,000 URLs selected, 89,891 were retrievable. An additional 111,107 unique URLs were retrievable by randomly fetching two distinct child links from each page of the initial set (whenever possible). The top five represented hosts were: www.geocities.com (561 URLs), www.webring.com (419 URLs), www.amazon.com (303 URLs), members.aol.com (287 URLs), and www.tripod.com (196 URLs). Combined, they represent less than 1% of the URLs used. Figure 5.2 shows the most frequent top-level domains in our data; close to half of the URLs are from .com, and another 26.8% of the URLs came from .edu, .org, and .net. Approximately 18% of the URLs represent top-level home pages (i.e., URLs with a path component of just /). The initial dataset contained a mean of approximately 49 links per page, with the distribution shown in Figure 5.3.

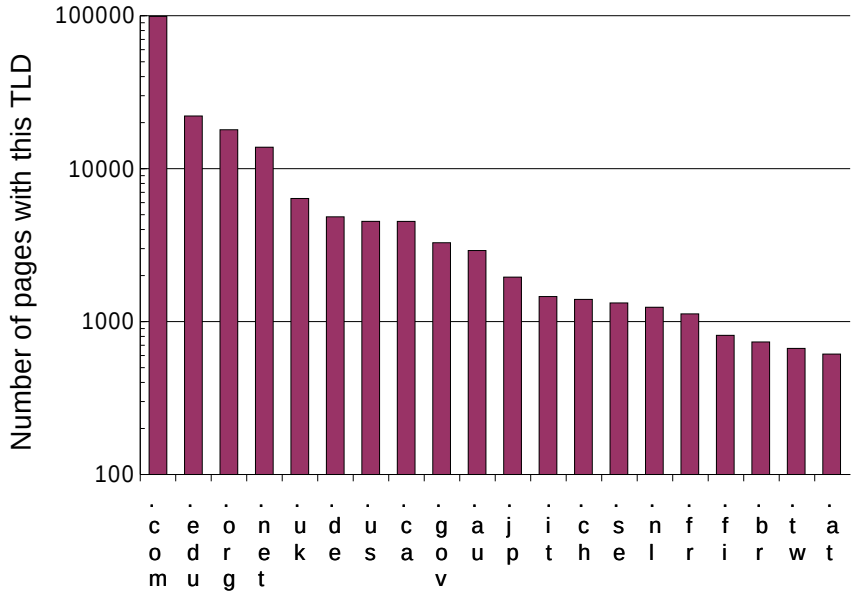


Figure 5.2: Representation of the twenty most common top-level domain names in our combined dataset, sorted by frequency.

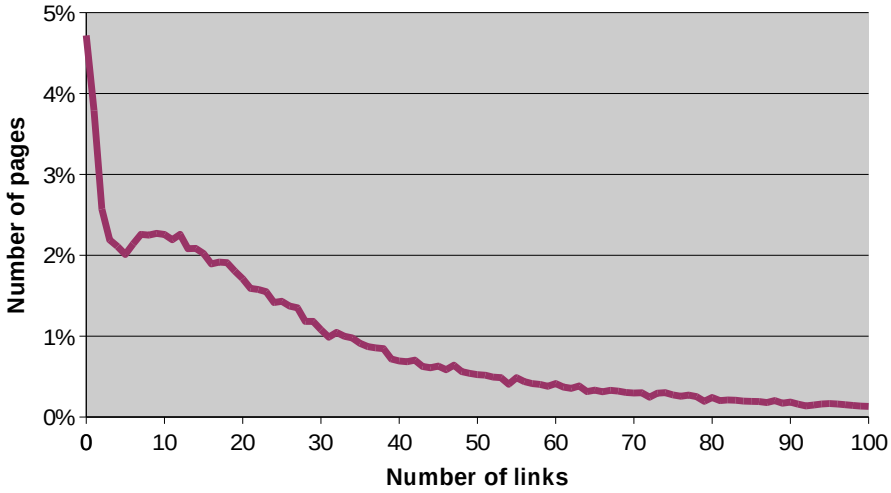


Figure 5.3: The distribution of the number of links per Web page.

With respect to content length, the sample distributions used for source and target pages are similar, so we present one distribution (pages from the initial dataset containing titles), shown in Figure 5.4. Thus it can be seen that almost half of the Web pages contain 250 words or less.

As shown in Figure 5.5, for pairings of pages with links between them, the domain name matched 55.67% of the time. For pairings of siblings, the percentage was 46.32%.

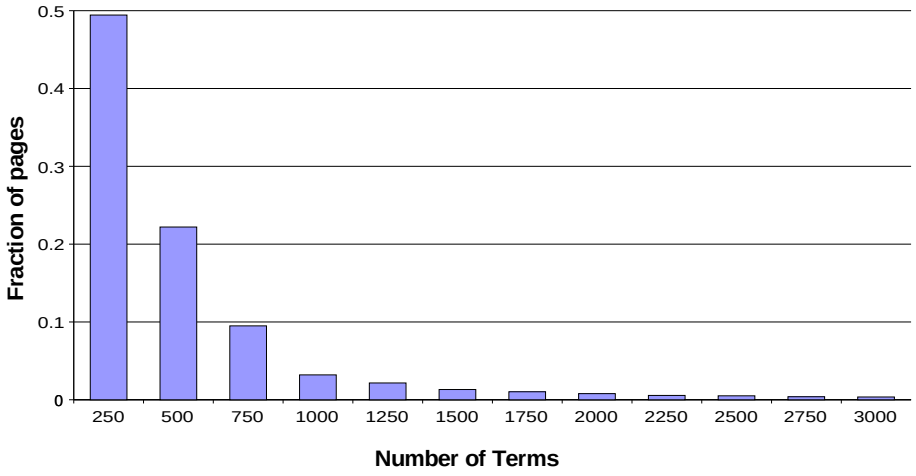


Figure 5.4: Distribution of content lengths of Web pages.

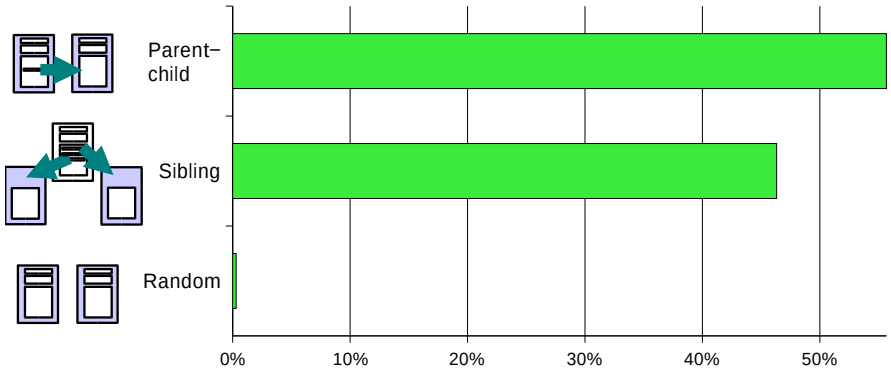


Figure 5.5: Percentages of page pairings in which the domain name matched.

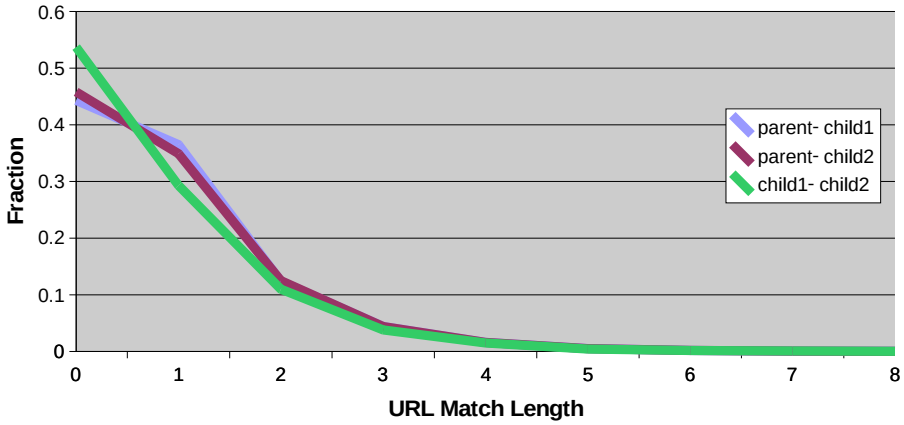


Figure 5.6: Distributions of URL match lengths between the URLs of a parent document and a randomly selected child document are similar to that of a parent and a different child document, as well as between the URLs of the two selected child (sibling) documents.

For random pairings of pages, the domain name matched 0.003% of the time. We also measured the number of segments that matched between URLs. A score of 1 means that the host name and port (more strict than just domain name matching) matched. For each point above 1, an additional path segment matched (i.e., top-level directory match would get 2; an additional subdirectory would get 3, and so on). The distributions of these segment match lengths for connected pages are shown in Figure 5.6.

Figure 5.7 shows TFIDF-cosine similarities for the author-supplied same-page descriptors (titles and description META tag contents). Here we can see that the description is typically a slightly better descriptor for the body of the page than the title.

5.5.2 Page to page characteristics

Figure 5.8 presents the TFIDF similarity scores of the current page to the linked page, to random pages and between sibling pages. This figure demonstrates that random page texts have almost nothing in common, and that linked page texts have more in common than sibling pages.

In Figure 5.9, we plot sibling page similarity scores as a function of distance between

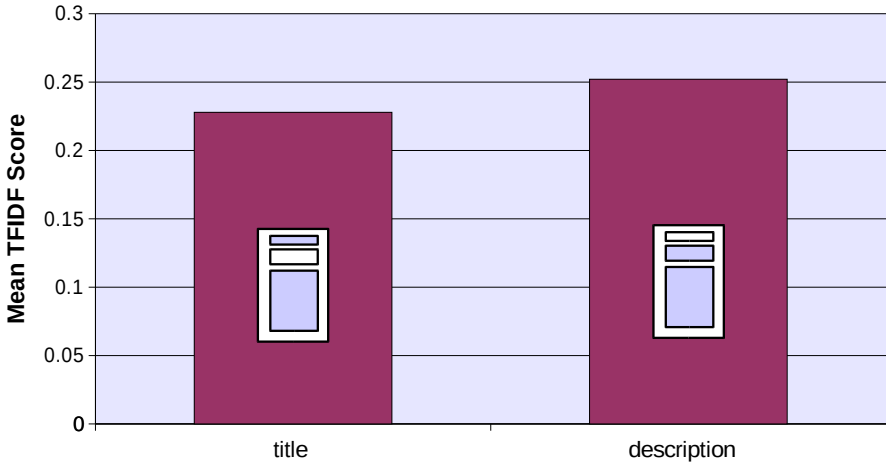


Figure 5.7: Textual similarity scores for page self-descriptors. In this (and following figures), icons are used to help represent the relevant portions of pages (shaded) and the relationship being examined. Here, we consider the similarity between the title and the body text and between the description and body text.

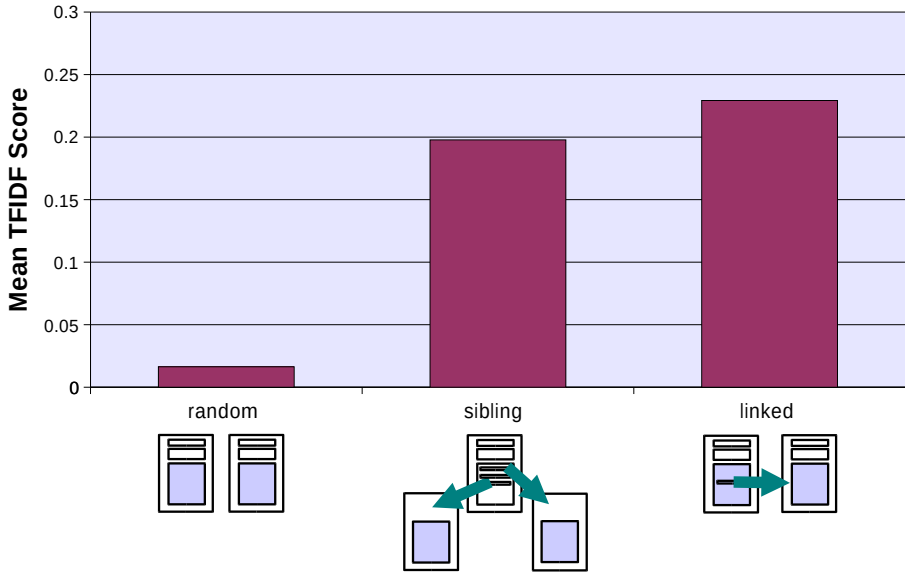


Figure 5.8: TFIDF-cosine similarity for linked pages, random pages, and sibling pages.

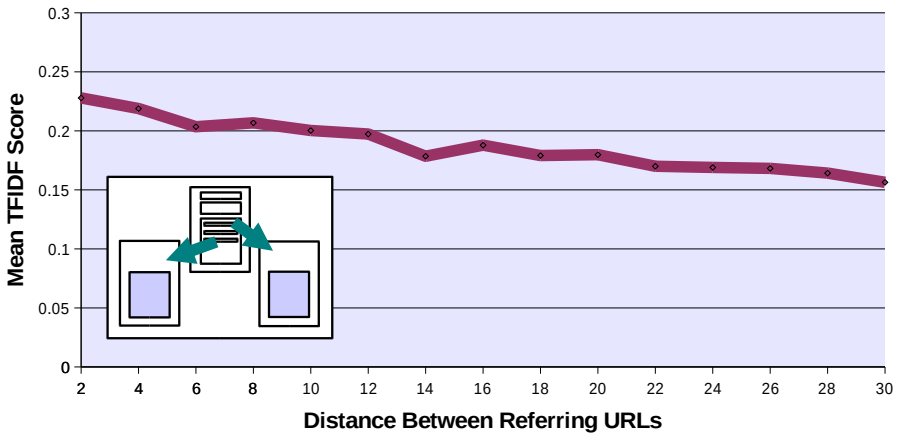


Figure 5.9: Plot of similarity score between sibling pages as a function of distance between referring URLs in parent page for TFIDF.

referring URLs in the parent page (where distance is the count of the number of links away). Thus, if two outgoing links (A,C) from page X are separated by a third outgoing link (B), then C is a distance two away from A. We find that in general, the closer two randomly selected URLs are on their parent page, the more likely they are to share the same terms. This is corroborated by others [DH99, CDI98] who have observed that links to pages on similar topics are often clustered together on the parent page.

5.5.3 Anchor text to page text characteristics

Anchor text, by itself, has a mean length of 2.69 terms, slightly lower than the averages reported in studies of smaller Web page collections by Amitay [Ami97]. In comparison, titles are almost twice as long with a mean length of 5.27 terms (distributions of both are shown in Figure 5.10). However, we can also consider using text before or after the anchor text, and when we consider using up to 20 terms before and 20 terms after, we get a mean of 11.02 terms.

Figure 5.11 shows that anchor text score is a much better descriptor for non-random pages. Even the similarity of anchor text to pages that are siblings of the targeted page get scores at least an order of magnitude better than random.

We also tested the improvement that additional text around the anchor might provide to the similarity between link and target text. However, the mean TFIDF similarity

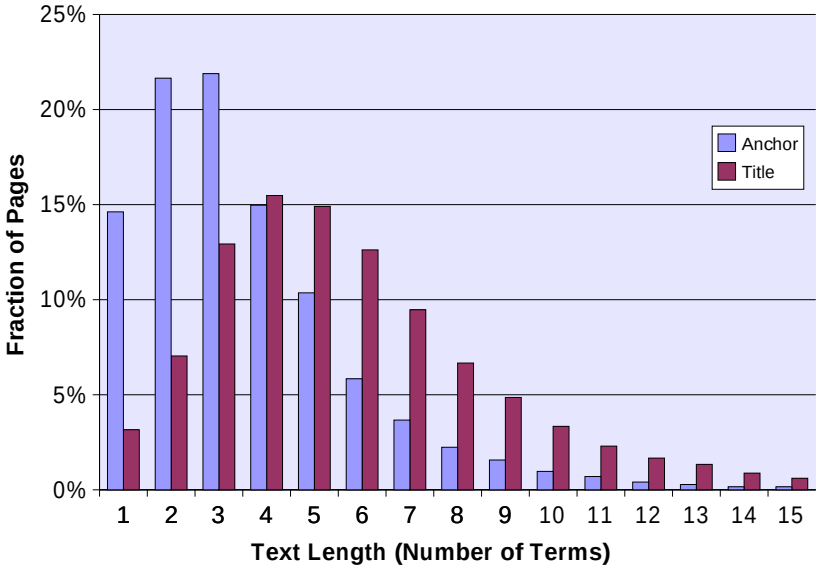


Figure 5.10: Distribution of the number of terms per title and per anchor.

Figure 5.11: Measured similarity of anchor text only to linked text, text of a sibling of the link, and the text of random pages.

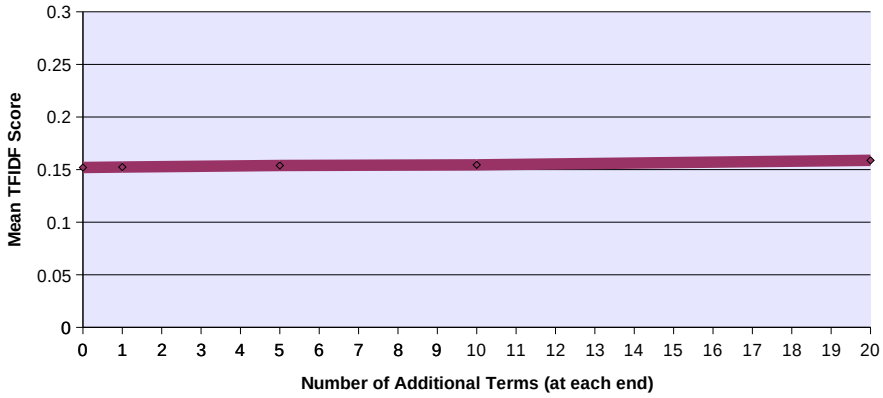


Figure 5.12: Performance of varying amounts of anchor text to linked text.

1) e mail	6) web site	11) contact us	16) Web sites
2) home page	7) you can	12) last updated	17) world wide
3) all rights	8) this site	13) more information	18) copyright 1999
4) rights reserved	9) this page	14) check out	19) Web page
5) click here	10) http www	15) new york	20) your own

Table 5.2: The twenty most common bigrams found after removing bigrams containing articles, prepositions, conjunctions, and various forms of the verb *to be*.

scores (Figure 5.12) between the target page text and anchor text plus varying amounts of surrounding text are almost constant. While there is some improvement as more text is added, it is relatively small.

5.6 Related Work

While at first glance they may look different, the results presented above are compatible to those reported by Chakrabarti *et al.* [CDR+98]. They found that including fifty bytes of text around the anchor would catch most references of the term "Yahoo" for a large dataset of links to the Yahoo home page [Yah02]. Our interpretation is that while additional text does increase the chance of getting the important term(s), it also tends to catch more unimportant terms, lowering the overall term probability scores. While these results may not be particularly encouraging for the use of text surrounding an anchor, such text is occasionally quite useful (especially for link text made of low-content terms like "click here").

Text on the Web is not the same as text off the Web. Amitay [Ami97, Ami99]

examines the linguistic choices that Web authors use in comparison to non-hypertext documents. Without going into the same detailed analysis, we did find some similar characteristics of Web pages. The bigrams "home page" and "click here" were the seventh- and thirteenth-most popular (of all raw bigrams), and certainly not typical bigrams of off-Web text. Interestingly, "all rights" and "rights reserved" were the eleventh- and twelfth-most popular, perhaps reflecting the increasing commercialization of the Web. Table 5.2 contains a list of the most frequent content-bearing bigrams (i.e., adjacent terms not containing articles, prepositions, conjunctions, or forms of the verb *to be*).

Amitay [Ami00, AP00, Ami01] has built a system call InCommonSense to automatically extract the descriptions that people write about other pages to use as summaries (e.g., for page descriptions in search engine results). This approach (implicitly using paragraph writing conventions) is likely to be more robust that the simplistic method described in this chapter, and so might provide the basis for improvement in the utility of surrounding text.

In many cases, it may be desirable to differentiate between different types of Web links (e.g., [HG98]) and to have a better model for determining internal versus external site links. Elsewhere [Dav00a], we have examined the problem of nepotistic links, in which links exist for reasons other than merit (e.g., advertising or navigational links rather than links created as recommendations). Such links are often (but not exclusively) created to influence search engine rankings that use some forms of link analysis. We demonstrated the potential of some simple methods for recognizing such links.

5.7 Summary

This chapter provides empirical evidence of topical locality of pages mirroring spatial locality in the Web — that is, WWW pages are typically linked to other pages with similar textual content. We found that pages are significantly more likely to be related topically to pages to which they are linked, as opposed to other pages selected at random, or other nearby pages. Furthermore, we found evidence of topical locality

within pages, in that sibling pages are more similar when the links from the parent are closer together.

We also found that anchor text is most similar to the page it references, followed by siblings of that page, and least similar to random pages, and that the differences in scores are easily statistically significant because of the large number of samples and often large (an order of magnitude or more). This suggests that anchor text may be useful in discriminating among unseen child pages. We note that anchor text terms can be found in the target page nearly as often as the title terms on that target page, but that the titles also have better similarity scores. We have pointed out that on average the inclusion of text around the anchor does not particularly improve similarity measures (but neither does it hurt). Finally, we have shown that titles, descriptions, and anchor text all have relatively high mean similarities to the corresponding document, implying that these page proxies represent at least part of the target page well.

Web links provide some measure of semantic linkage between pages as well as the anchor text used in the links. We have demonstrated that this semantic linkage, as approximated by textual similarity, is measurably present in the Web, thus providing the underpinnings for various Web systems, including search engines, focused crawlers, linkage analyzers, intelligent Web agents, and content-based usage predictors.

Chapter 6

Content-Based Prediction

6.1 Introduction

The World Wide Web has become the ultimate hypertext interface to billions of individually accessible documents. Unfortunately, for most people, retrieval times for many of those documents are well above the threshold for perception, leading to impressions of the "World Wide Wait." From Chapter 2 we saw that Web caching is one approach that can effectively eliminate the retrieval time for many recently accessed documents, but it cannot help with documents that have never been visited in the past.

If future requests can be correctly predicted, those documents could potentially be pre-loaded into a local cache to provide fast access when requested by a user, even when they have not been accessed recently. This chapter proposes and evaluates such a predictive approach. While conventional predictive techniques look at past actions as a basis to predict the future, we consider an approach based on the hypertext characteristics of the Web.

In Chapter 5, we showed that Web pages are typically linked to pages with related textual content, and more specifically, that the anchor text was a reasonable descriptor for the page to which it pointed. Many applications already take advantage of this Web characteristic (e.g., for indexing terms not on a target page [Goo02, BP98] or to extract high-quality descriptions of a page [Ami98, AP00]). In this work we will examine in detail another application of this result.

Our motivating goal is to accurately predict the next request that an individual user is likely to make on the WWW. Therefore, this chapter examines the value of

using Web page content to make predictions for what will be requested next. Many researchers have considered complex models for history-based predictions for pre-loading (e.g., [PM96, Sar00]), but relatively few have considered using anything other than simplistic approaches to the use of Web page content. Unlike many techniques that do examine content, our approach does not noticeably interfere with the user experience at all — it does not ask for a statement of interest, nor does it modify the pages presented to the user. Instead, it can be used invisibly to improve user-perceived performance.

One naive content-based approach is to pre-load all links on a page. A slightly more intelligent approach is to pre-load as time permits the links of a page, in HTML source order from first to last (corresponding generally to links visible from top to bottom). In this work we compare those approaches with an information retrieval-based one that ranks the list of links using a measure of textual similarity to the set of pages recently accessed by the user. In summary, we find that textual similarity-based predictions outperform the simpler approaches.

6.2 Background

As we described in Chapter 1, by transparently reducing network latencies and bandwidth demands, Web resource caching [Dav01c] has become a significant part of the infrastructure of the Web. Unfortunately, caching can only help when the objects requested are already present in the cache. Typically, caches contain objects that have been accessed in the past. Prefetching [PM96, KLM97], however, can be used speculatively to put content into the cache in advance of an actual request. One difficulty, however, is in knowing what to prefetch. Typical approaches have used Markovian techniques (e.g., [Duc99, Sar00]) on the history of Web page references to recognize patterns of activity. Others prefetch bookmarked pages and often-requested objects (e.g., [Pea02]). Still others prefetch links from the currently requested page [Lie97, CY97, Cac02a, Web02, Ims02, Kle99, IX00, PP97, Dee02].

However, prefetching all of the links of the current page is not a viable option, given that the number of links per page can be quite large, and that a prefetching

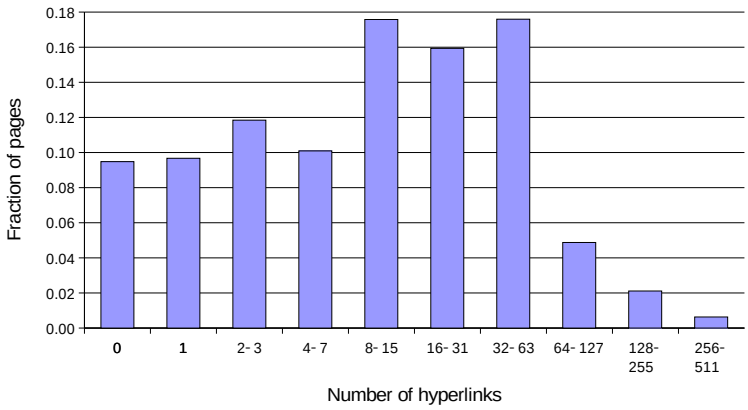


Figure 6.1: The distribution of the number of hypertext (non-embedded resource) links per retrieved HTML page.

system typically has only a limited amount of time to prefetch before the user makes a new selection. While likely heavy-tailed [CP95, CBC95], this "thinking-time" is typically less than one minute [KR01]. Likewise, prefetching content that is never used exacts other costs as additional resources such as bandwidth are consumed. We do not consider the thinking time or prefetching costs further in this chapter (see [Bes96, JK98, AZN99, ZAN99] for various utility theoretic approaches to balancing cost and benefits).

The difficulty of selecting a choice in content-based analysis is illustrated in Figure 6.1. It shows the distribution of the number of unique hypertext (non-embedded resource) links per retrieved HTML page for the dataset used in this chapter. The median number of links per page is 11, while the mean is 26, indicating the presence of some pages with a large number of links. (We describe this dataset further in Section 6.5.1.) Note that this histogram reflects the distribution of pages requested by users (which includes repeated retrievals), versus the more or less static distribution of pages on the Web (such as described by Bray [Bra96]).

The prefetching of content may cause problems, as not all content is cacheable (so prefetching it only wastes resources), and prefetching even cacheable content can abuse server and network resources, worsening the situation [Dav01a, CB98]. An alternative is to do everything but prefetch [CK00] — that is, to resolve the DNS in advance, connect

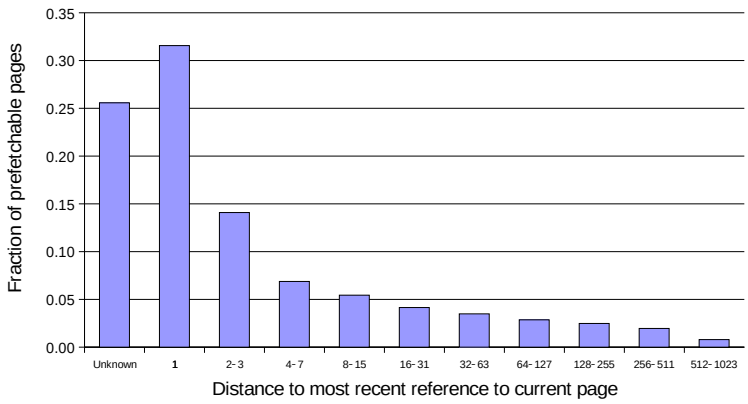


Figure 6.2: The distribution of the minimum distance, in terms of the number of requests back that the current page could have been prefetched.

in advance to the Web server, and even warm up the Web server with a dummy request.

Prefetching systems are difficult to evaluate, especially content-based prefetching ones (although we will consider one approach in Chapters 10 and 11) because even state-of-the-art proxy evaluation techniques (e.g., Polygraph [Rou02] as used in caching competitions [RWW01]) use artificial data sets without actual content. Even when using captured logs as the workload, real content will need to be retrieved, and will no longer be the same as the content seen when the logs were generated.

A more complex analysis is shown in Figure 6.2. It shows (by summing the fractions for columns 1 and 2-3) that approximately 46% of prefetchable pages (those that are considered likely to be cacheable) can be reached by examining the links of the current page and its two immediate predecessors in the current user's request stream. (The bar marked unknown represents those pages that were never linked from any pages seen during our data collection.)

For comparison, the per-client average recurrence rate (that is, the rate at which requests are repeated [Gre93]) for clients making more than 500 prefetchable requests was 49.2%, and for all clients was 22.5% (since some clients made very few requests). This means that if each client utilized an infinite cache that passively recorded cacheable responses, the per client average hit rate on prefetchable content would be less than

25% (corresponding to the average recurrence rate). However, if the clients additionally prefetched all perceived prefetchable links, the hit rates on prefetchable content would increase to almost 75%, because the prefetchable pages that would not be in the cache (i.e., the misses) correspond at most to the pages represented by a distance of unknown in Figure 6.2. Note that these are the results of per-user analysis on `text/html` responses not satisfied by the browser cache. Additionally, a single shared cache (that stores all user requests) would have a recurrence rate of 50.9% for cacheable resources (47.3% for all resources). From this analysis we conclude that there is significant potential for content-based prediction of future Web page requests when caching is considered.

6.3 Related Work

Chan [Cha99] outlines a non-invasive learning approach to construct Web user profiles that incorporates content, linkage, as well as other factors. It uses frequency of visitation, whether bookmarked, time spent on page, and the percentage of child links have been visited to estimate user interest in a page. Predictions of user interest in a page are made by building a classifier with word phrases as features and labeled with the estimation of page interest. Thus, new pages can be classified as potentially being of interest from the examination of their contents. Chan's goal is similar but not the same as ours. He is interested in modeling what the user might like to see, while we are concerned with what the user will do. (See Haffner *et al.* [HRI+00] for a theoretical discussion of the similarities of the two goals.) Unlike the approaches considered here, his method requires that anything that is to be considered for recommendation must already be prefetched (for analysis). Additionally, the methods he proposed were not incremental, nor did they change over time.

Chinen and Yamaguchi [CY97] describe and evaluate the performance of their Wcol prefetching proxy cache. Their system simply prefetches up to the first n embedded images and m linked documents. In a trace with a large number of clients, they found that if all targets were prefetched, a hit rate of 69% was achievable. Prefetching ten targets resulted in approximately 45%, and five targets corresponded to about 20% hit

rate. Here we use a variant of their approach (termed original order) as one of the prediction methods that we will use for evaluation.

More sophisticated link ranking approaches are possible. Klemm's WebCompanion [Kle99] is a prefetching agent that parses pages retrieved, tracks costs to retrieve from each host, and prefetches the pages with the highest estimated round trip time. Average access speedups of over 50% with a network byte overhead (i.e., wasted bandwidth) of 150% are claimed from tests using an automated browser.

Ibrahim and Xu [IX00] describe a neural net approach to prefetching, using clicked links to update weights on anchor text keywords for future predictions. Thus they rank the links of a page by a score computed by an artificial neuron. System performance of approximately 60% page hit ratio is claimed based on an artificial news-reading workload.

A number of others have considered the similar problem of Web page recommendation and Web page similarity. However, they (e.g., [DH99]) tend to use information that can only be acquired by the retrieval of additional pages, which would likely overwhelm the link bandwidth and negate the purpose of intelligent prefetching. Our approach, in contrast, uses only the content that the client has previously requested to base decisions on what to prefetch next.

Recently Yang *et al.* [YSG02] considered various approaches to hypertext classification. Their results are mixed, finding that identification of hypertext regularities and appropriate representations are crucial to categorization performance. They note, however, that "algorithms focusing on automated discovery of the relevant parts of the hypertext neighborhood should have an edge over more naive approaches." While their study examines a related question, unlike our work it treats all links (both incoming and outgoing) equally, and does not consider the use of anchor or surrounding text.

One of the difficulties in comparing work in the area of content-based prediction and prefetching is the inability to share data. Some, like Chinen and Yamaguchi, evaluate their work by using their system on live data. Even if the trace of resources requested were available, such traces deteriorate in value quickly because the references contained within them change or disappear. Others, such as Klemm as well as Ibrahim and Xu

test their systems with relatively contrived methods, which make it difficult to ascertain general applicability. Our initial approach here has been to capture a full-content Web trace so that analysis can be performed offline. However, because of privacy concerns, we (like other researchers) are unable to provide such data to others. In Chapter 10 we will propose a mechanism capable of comparing multiple prefetching proxy caches simultaneously. We will develop a prototype system in Chapter 11, but it is only applicable to fully realized prefetching systems.

6.4 Content-Based Prediction

In many user interfaces, multiple user actions may appear atomic and thus may be inaccessible for analysis. For those environments, prediction methods are often limited to those that can recognize and extrapolate from patterns found in a history log (e.g., UNIX shell commands as we described in Chapter 3, or telecommunication switch alarms [Wei01]). In contrast, the Web provides easy access to the content being provided to the user. Moreover, this content prescribes boundaries within which the user is likely to act. As described in Chapter 5, studies [CP95, TG97, Dav99c] have shown that the user is likely to select a link on the current page between 80 and 90% of the time. Therefore, the set of links shown to the user is a significant guide to what the user is likely to request next.

In addition, the content provides possible next steps that historically-based prediction approaches cannot. A user model based on past experience cannot predict an action previously never taken. Thus, a content-based approach can be a source of predictions for new situations with which history cannot help.

While a user does provide some thinking time between requests in which prefetching may occur, it is limited, and so there may be insufficient time to prefetch all links. Moreover, even if there were sufficient time, the user and network provider may find the extra bandwidth usage to be undesirable. Thus, the approach is typically not to predict all links, but instead to assign some weight to each link and use only the highest scoring links for prediction (and subsequent prefetching).

A prefetching system, whether integrated into a browser, or operating at a proxy, has access to the content of the pages served to the user. Given a particular page, the external links that it contains can be extracted. The anchor and surrounding text can be used as a description of the page (as shown in [Dav00c] and used in [Ami98, AP00]). Given a set of links and their descriptions, the problem is then how to rank them.

Ideally, we would have a model of a user's interest and the current context in which the user is found (e.g., as in [BJ01]). One approach is to explicitly ask the user about their current interest, as done in WebWatcher [AFJM95, JFM97] and in AntWorld [KBM+00] or to rate pages (as in Syskill and Webert [PMB96, PB97] and AntWorld). We could also consider modifying the content seen by the user to give hints or suggestions of recommended or preloaded content (as in Letizia [Lie95, Lie97], WebWatcher, QuIC [EBHDC01], the prefetcher implemented by Jiang and Kleinrock [JK97], and common uses of WBI [BMK97]).

However, our preference is to build a user model as unobtrusively as possible (as in [Cha99]) to perform prefetching behind the scenes. For the purposes of this study, this means we cannot ask the user explicitly about his or her interest, nor change the content, but it does enable us to work offline with logs of standard usage. Therefore, we will not consider obtrusive approaches further here.

Instead we will use the textual contents of recently requested pages as a guide to the current interests of the user. This method allows us to model a user's changing interests without mind-reading or explicit questions, and is intended to combine with other sources of predictions to generate an adaptive Web prefetching approach [Dav99a]. How well it performs is the subject of the next section.

6.5 Experimental Method

As mentioned earlier, experimental evaluation of a content-based prefetching system is difficult. Rather than deploying and testing a complete prefetching system, we have elected to test the content-based prediction methods described above in an offline manner. For comparison, one baseline we will use is the accuracy of a random ranking.

6.5.1 Workload collection

We implemented a custom Java-based non-caching HTTP/1.0-compliant proxy to monitor and record the full HTML content and all headers of user requests and responses. We captured approximately 135,000 requests, of which just over a third generated HTML responses. (The rest correspond to embedded resources such as images and non-data responses such as errors or content moved.) Users were predominantly computer science students and staff from Rutgers University. More than fifty distinct clients used the service from August 1998 to March 1999.

While covering close to seven months, this trace is rather small, and predominantly reflects the traffic of university users. However, with concerns over privacy, the choice was made to recruit explicit volunteers instead of surreptitiously recording the activity of all users of an existing cache or packet snooping on a network link (e.g., as in [FCD+99, Fel98]).

This log does not distinguish users — it only distinguishes clients based on IP address, and so if multiple users of this proxy operated browsers on the same machine (a possibility under UNIX) or behind the same proxy, those users' requests could be interleaved. Likewise, users with dynamic IP addresses are seen as distinct users on each session and multiple users (if assigned the same IP address at different times) may seem like the same user returning over multiple sessions. We believe the likelihood of multiple users per IP (whether by proxy, or the re-use of dynamic addresses) happening is small for this data. More importantly, though, the log does not reflect the requests made by users that are satisfied by browser caches. Users of this proxy were directed to use the proxy, and not told to disable browser caching.

The data was recorded in several segments, generally corresponding to separate runs of the proxy. The first segment, representing about 5% of the log, was used for exploratory analysis and was manually examined. As we describe specific adaptations made to the data, it is from the measurement and analysis of this initial segment. Overall results shown in graphs will reflect measurements over the entire dataset. As a result, any effects of fitting the initial data is limited to just 5% of the dataset.

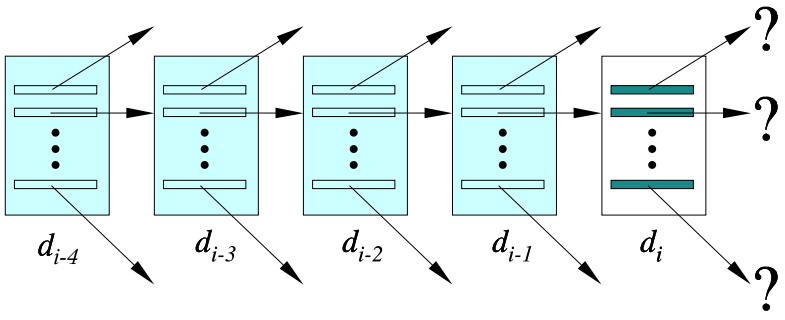


Figure 6.3: Sequence of HTML documents requested. The content of recently requested pages are used as a model of the user's interest to rank the links of the current page to determine what to prefetch.

6.5.2 Data preparation

Our custom proxy attempted to record all HTTP [FGM+99] headers of requests and responses, and the contents of all HTML responses. These were written in chronological order, thus interleaving requests when multiple users were active. Since we are interested in predictions from sequences of requests for individual users, we extracted the HTML records and sorted by client and timestamp. From these sorted access requests, we built sets of up to five HTML responses. (While a larger number of responses might have provided more data, we wanted to model the current user interest which might be diluted with data from further in the past.) Thus, for example, responses d_1, d_2, d_3, d_4, d_5 would be the first set, from which we would attempt to predict d_6. Likewise, d_2, d_3, d_4, d_5, d_6 corresponded to the second set, from which we would attempt to predict d_7. However, occasionally error messages (e.g., HTTP response codes 403, 404), redirections (e.g., 301, 302), and content unchanged (code 304) ended up being included, as Web servers sometimes described these as of type `text/html` as well.

Since we had stored the content, whenever we encountered a 304 content unchanged response for content that we had previously recorded, we used our copy of the earlier content. Requests for URLs generating the remaining 304 responses could be predicted, but not used as content (since that content must have been sent before starting our measurements we couldn't have it) for use in future predictions.

We discarded those responses with HTTP error codes (i.e., response codes ≥ 400), but retained redirections (e.g., response codes 301, 302) since links were made to one

URL, which would then redirect elsewhere. In this case, the original log might have included retrieval events $d_1, d_2, d_3, d_4, d_5, r_6, d_6, d_7$ (where r_6 corresponds to a redirection to d_6) but we would instead generate the two cases d_1, d_2, d_3, d_4, d_5 predicting r_6 and d_2, d_3, d_4, d_5, d_6 predicting d_7. We use r_6 as a target of prediction since any link followed (if present at all) would have been made to r_6, but there is no usable content in r_6 so we don't use it for text analysis. Unfortunately, as mentioned earlier, we only wanted HTML responses, but occasionally we would see redirections for non-HTML content, such as banner advertising that would use one or more redirections. We manually filtered URLs for some of the more popular advertising hosting services to minimize this effect.

The pages were parsed with the Perl HTML::TreeBuilder library. Text extraction from the HTML pages was performed using custom code that down-cased all terms and replaced all punctuation with whitespace so that all terms are made strictly of alphanumerics. Content text of the page includes title but not META tags nor alt text for images. URLs were parsed and extracted using the Perl URI::URL library plus custom code to standardize the URL format (down-casing host, dropping #, etc.) to maximize matching of equivalent URLs. The basic representation of each textual item was bag-of-words with term frequency [SM83].

6.5.3 Prediction methods

We present a total of four methods to rank the URLs of the current page for prediction. Two are simple methods: the baseline *random* ordering, and *original rank ordering* (i.e., first-occurrence order in page). The second two rank each link based on the similarity of the link text with the combined non-HTML text of the preceding pages (as illustrated in Figure 6.3). To measure the similarity between two text documents (D_1, D_2), we use a very simple metric[1]:

$$\mathrm{TF}(w, D_i) = \text{number of times term } w \text{ appears in } D_i$$

[1]Other variations of this similarity metric (e.g., to include factors for document length) were tested on the first data segment but they performed similarly or worse. We make no claims as to the optimality of this similarity metric for this task.

$$\text{Text-Sim}(D_1, D_2) = \sum_{all\ w} \text{TF}(w, D_1) * \text{TF}(w, D_2)$$

The first method (termed *similarity* in our tests) uses the highest similarity score (in the case of multiple instances of the same link to a URL) of the anchor and surrounding text to the preceding pages. The second (termed *cumulative*) is identical, except that it sums the similarity scores when there are multiple links to the same target (giving extra weight to such a URL).

Links recognized include the typical `a href`, but also `area href`, and `frame src`. We did not parse embedded JavaScript to find additional links, although in reviewing the initial data segment we did find JavaScript with such links.

Since links are typically presented within some context, in addition to anchor text we tested the addition of terms before and after the anchor. For example, if the HTML content of a page were:

```
The <a href="http://www.acm.org/acm1/">ACM</a> Conference will
take you beyond cyberspace.
```

then 0 additional terms would provide just {ACM}. If we permit 5 additional terms on each side, then we would get the set {The, ACM, Conference, will, take, you, beyond} to be used for comparison against the text of the preceding pages. Note that text around an anchor may be used by more than one link, in the case that the window around an anchor overlapped that of another anchor. In experiments on the first data segment, we tested the use of up to 20, 10, 5, and 0 additional terms and found that in most cases, when more terms were used, performance was slightly better. Thus we use as many as twenty additional terms before and after the anchor text. However, the text around a URL was never permitted to extend into another URL's anchor text.

Furthermore, we noticed that when exploring pages within a site, certain elements of that site were often repeated on every page. Examples would include a copyright notice with a link to a use license page. In this case, such repeated elements would get undue weight over non-repeated text. As a result, a link containing the repeated text (the link to a license, in the case above) would be ranked highly, even if irrelevant to the true content of the pages. To combat this effect, we tested a variation that no

longer used all text. Instead we used a Perl version of htmldiff [DB96] to compare sequentially requested pages and generate only the text that was present on the second (more recent) page. Thus, we used the first page, plus the textual differences to the second page, plus the differences between pages 2 and 3, etc. In this way we hoped to eliminate significant emphasis on repeated structure in the pages.

We used the well-known Porter [Por97] stemming (since it helped slightly), but did not use stop word elimination (since it performed either slightly worse or no change). The non-HTML text (including title, keywords, and meta description) of the preceding pages was combined to serve as a "document" representing the current user interest against which we would measure similarity. However, since parts of the current page are also represented in the anchor texts pointing to the target documents, we tested the use of the current page in the model for user interest, and found that it also decreased performance on the initial segment slightly, and so we do not include it. Thus in summary, we use the non-repeating text of the preceding four pages as our model of current user interest against which we measure similarity, as illustrated in Figure 6.3.

6.5.4 Evaluation

Given the ranking methods described above, we will evaluate their predictive accuracy (the fraction of times that they include the correct next request in their prediction) over the entire dataset. In most cases, however, we will scale the results to show the fraction predicted correctly out of those possible to get correct (defined below). We will consider variations in the number of predictions by evaluating algorithms that use their best 1, 3, or 5 predictions. Additionally, we will consider the case in which the clients have an infinite cache into which predictions are prefetched. With such a cache, success is measured not by whether the immediate prediction was correct, but whether the requested object is present in the cache (a test arguably closer to real world considerations).

The ultimate goal of this effort is to use the best prediction method to provide suggestions for prefetching. It is generally not possible to prefetch the results of form entry, and is potentially undesirable to prefetch dynamically generated pages, so we

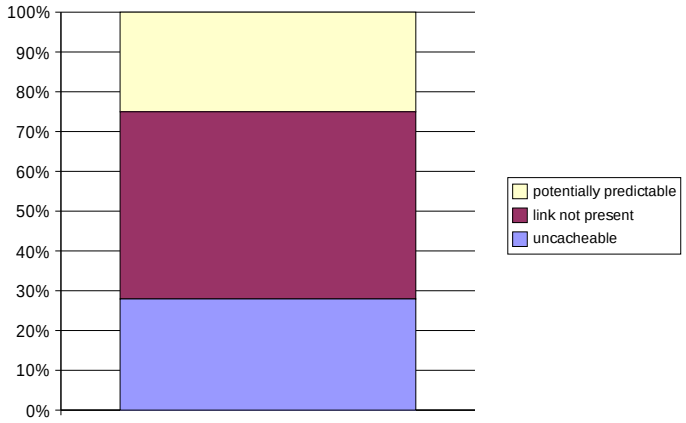


Figure 6.4: Fraction of pages that are uncacheable, found as a link from previously requested page (potentially predictable), or not found as a link from previous page.

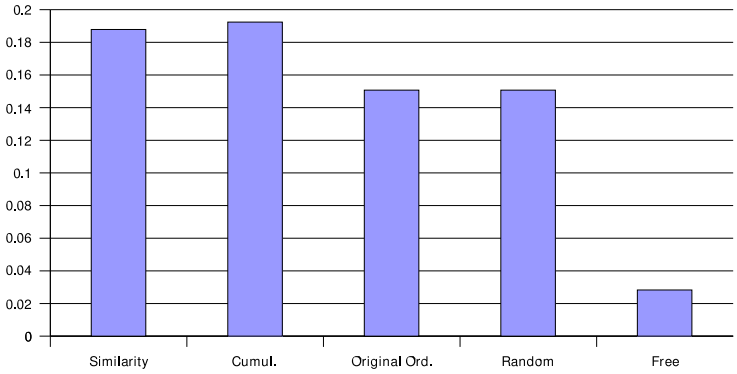
would like to not allow predictions for such pages (i.e., all responses generated by POSTs or with URLs containing "cgi" or "?"). Those responses represent 28% of all HTML collected. Additionally, we can recognize those cases where the next cacheable URL requested was not present in the preceding page, making a correct prediction an impossibility in another 47%. The remaining 25% have the potential to be predicted correctly by choosing from the links of the current page (see Figure 6.4). Approximately 2.8% of those pages (.7% of the total) had only one URL, and so will be predicted correctly under all prediction mechanisms.

Typically a prefetching system has time to prefetch more than one page, and so we consider the case in which we can select the top three and five most-likely URLs and mark the set as "correct" if one of them is the requested page. Under this arrangement, the set of potentially predictable pages stays the same (25%), but the number of default or "free" wins grows since there will be more cases in which the number of distinct links available is within the size of the predicted set.

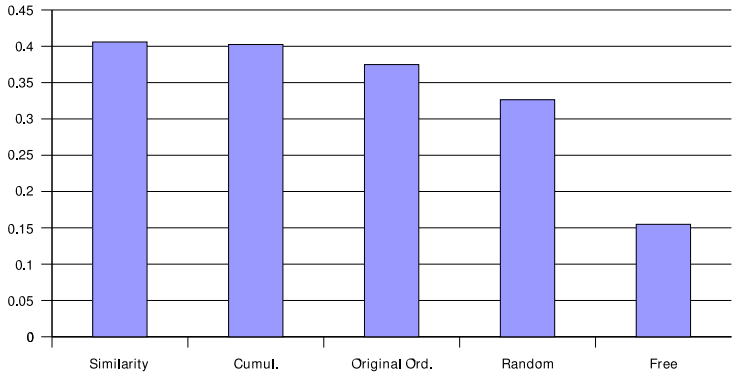
6.6 Experimental Results

6.6.1 Overall results

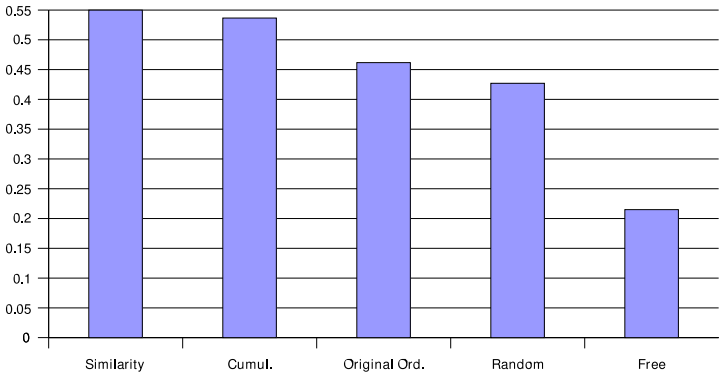
Figure 6.5 shows the overall accuracy for each of the content prediction algorithms, as a fraction of predictions that are possible to get correct from the last page of links.



(a) Top one prediction



(b) Top three predictions



(c) Top five predictions

Figure 6.5: Overall predictive accuracy (within potentially predictable set) of content prediction methods when considering links from the most recently requested page.

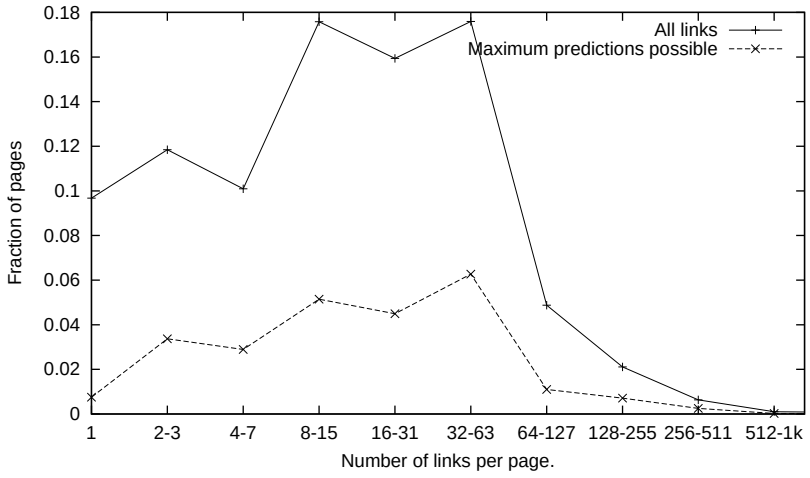


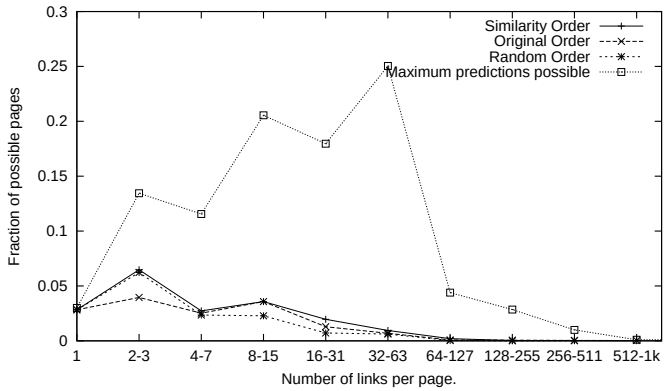
Figure 6.6: Potential for prediction when considering links from the most recently requested page.

It depicts performance for the cases in which only the highest-scoring prediction is evaluated as well as when the top-3- and top-5-highest-scoring predictions are used (note the different y-axis scales). In addition to being included within the performance for each algorithm, the "free" cases are also plotted to show a lower bound (recall that the free cases are pages in which the number of links in the page is no larger than the number of predictions permitted, thus automatically correct).

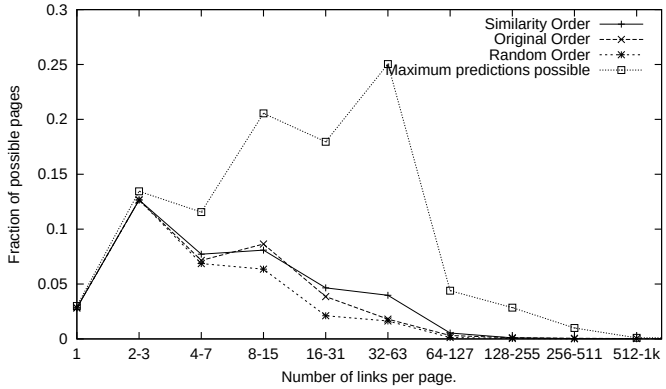
It can be seen that the similarity and cumulative ranking methods outperform the original rank ordering, and all three outperform the baseline random order. This suggests that the similarity-based approaches are beneficial for this link-ranking task. It also suggests that either by custom or by design, users tend to select links earlier in the page. The cumulative approach edges out the similarity approach for top-one, but similarity is best for top-three and top-five. This is likely because top-three and top-five are more likely to include entries that might appear more than once (thus benefiting from the cumulative emphasis).

6.6.2 Distributions of performance

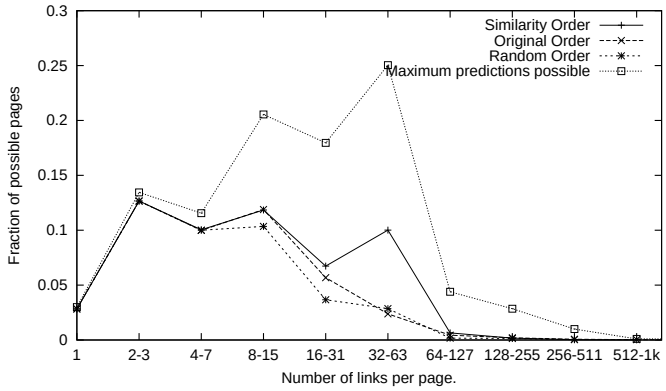
This chapter examines the predictability of Web page requests based on the links of the current page. In this section we consider the distribution of successful predictions with



(a) Top one prediction



(b) Top three predictions



(c) Top five predictions

Figure 6.7: Performance distribution for content prediction algorithms when allowing guesses from the most recently requested page.

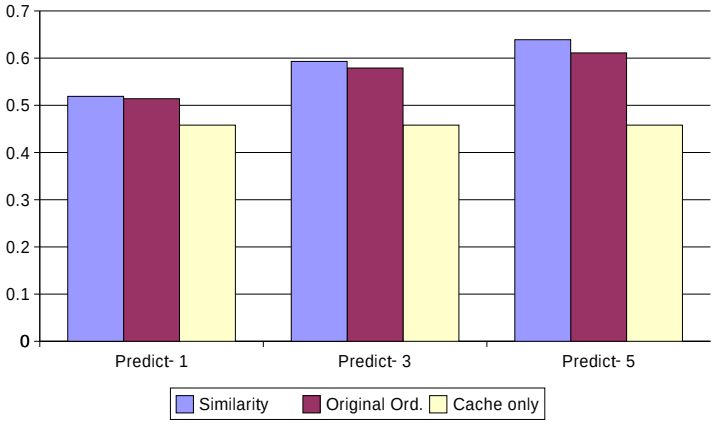


Figure 6.8: Predictive accuracy (as a fraction of all pages) of content prediction methods with an infinite cache compared to no prediction.

respect to the number of links on the current page. Figure 6.6 compares the overall distribution of the number of links per page with the distribution of cases in which the predicted page is found in the links of the last page. This figure demonstrates the limits of strict prediction from the links of the current page — there are many requests that do not appear within that set.

Focusing on those links that do appear within the current page, Figure 6.7 shows the distribution of the number of correct predictions per page size for each method plus the distribution of cases that can possibly be predicted correctly. Note that for clarity we have omitted the cumulative approach from these and further graphs, as its performance is closely tied with that of the similarity method. For most points of Figure 6.7(a), in which a single prediction is permitted, the performances of the various approaches are almost indistinguishable, but with more allowed predictions, more variation becomes visible.

6.6.3 Results with infinite cache

Since the system prefetching Web pages would be putting them into a cache, we consider briefly here the performance of prediction approaches in such a context. Figure 6.8 shows a summary of the predictive accuracy of two ranking methods assuming the use of an infinite cache, and compares them to an infinite cache without prefetching. While

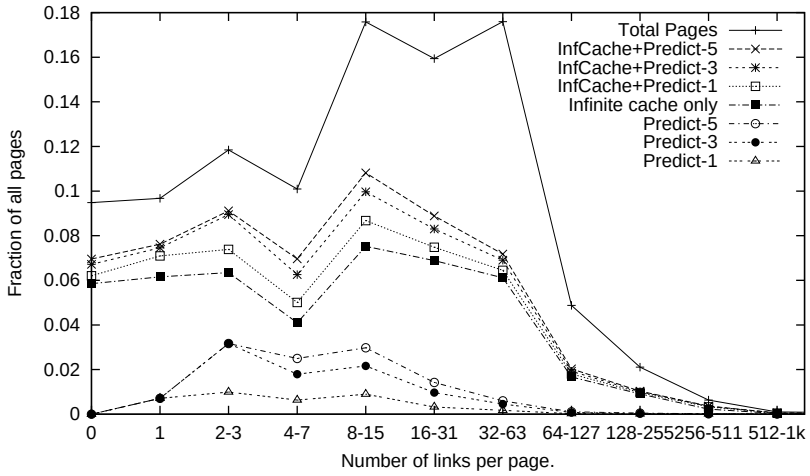


Figure 6.9: Performance distribution for original order with and without an infinite cache.

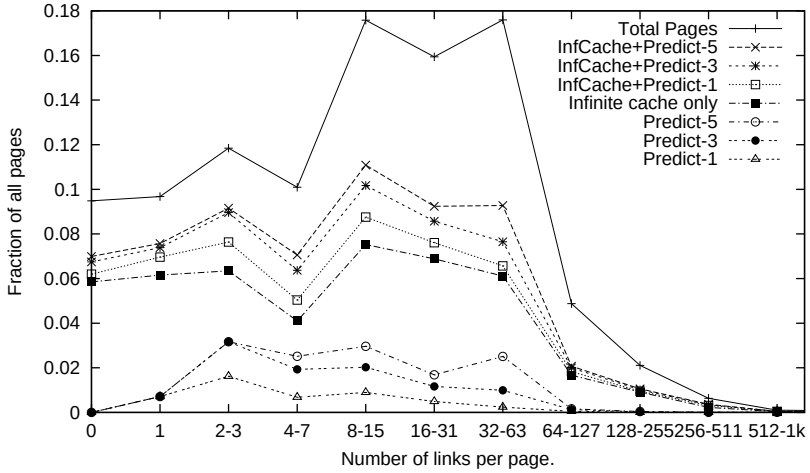


Figure 6.10: Performance distribution for similarity ranking with and without an infinite cache.

caching under these circumstances is obviously a significant contributor to performance, the predictive systems are able to improve on the caching-only performance, particularly when more than one prediction is permitted.

Figures 6.9 and 6.10 provide performance distributions of the original rank orderings and similarity orderings respectively. In these figures, note that the top curve represents the total page distribution, since we are no longer limited to the pages potentially

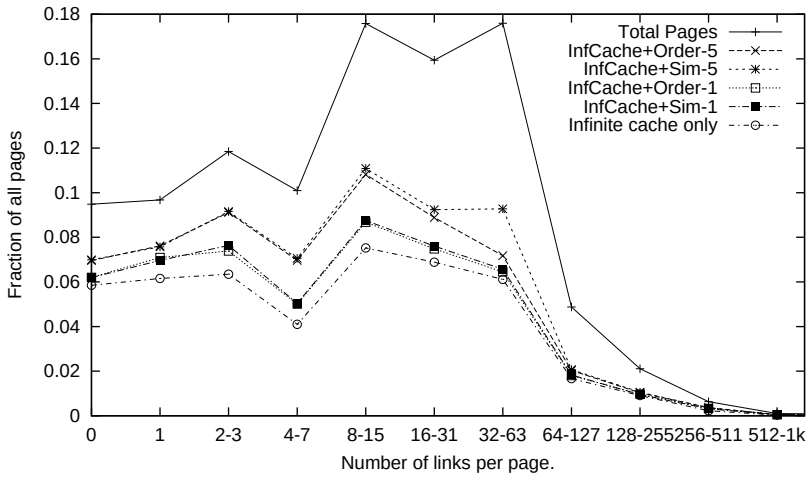


Figure 6.11: Comparison of performance distributions for original order and similarity with an infinite cache.

reached from the last ones retrieved. The original (non-cached) performance is shown (at the bottom), while the performance using an infinite cache is much higher. However, the cache does not do all the work, as the performance of an infinite cache alone is visibly below the cache+prefetching lines.

Figure 6.11 compares the cached performance of both the original rank ordering and similarity based ordering (we omit the top-three curves for clarity). The performance of cached original ranking and similarity ranking are nearly identical in most points; only for one point (at 32-63 links per page) does similarity rise much above the original ordering (when five predictions are allowed).

6.7 Discussion

Recall that the non-cached results presented are in terms of potentially predictable pages. This is, in fact, a relatively small fraction of all requests generated. For example, the best predictive method was shown to be the similarity ranking when predictions of the top 5 links are permitted, achieving an accuracy of approximately 55% (a relative improvement of close to 30% over random link selection). However, this represents success in only 14% of all cases, since only a quarter of all pages are possible to predict

in this manner. In the case of an infinite cache we are able to provide hits for 64% of all requests (a 40% improvement over a system without prefetching). Actual performance is likely to lie in between, as caches are finite and prefetched content can expire before it is needed.

Perhaps more importantly, this analysis is of a relatively small trace, and may not be representative of users in non-academic environments. However, more extensive analysis would require capturing content from a larger, more general population (such as that of an ISP), and is likely to raise significant privacy concerns.

The prefetching examined here is only for HTML resources. However, HTML resources represent only a fraction of all requests made by users. Most are embedded resources (such as images, sounds, or Java applets or the result of JavaScript that are automatically retrieved by the browser). We made the choice to ignore such resources as most of them are easy to predict by examining the HTML of the page in which they are embedded. This choice has a drawback, though, as users do indeed make requests for non-HTML resources, such as PDF and PostScript files (which can often be quite large), plain text resources, and the downloading of programs, albeit much less frequently.

In practice, a prefetching system will have time to fetch a variable number of resources. We have examined only three points on the range — allowing one, three, and five predictions. We believe that the typical number will fall within this range, however, as it will likely be useful to prefetch the embedded resources of prefetched pages, and some resources and pages will be cached from previous retrievals.

While extensive, these tests are not comprehensive — we have not attempted to disprove the possibility of other algorithms (text-based or otherwise) outperforming those presented here. In particular, we believe that systems with a stronger model of the user's interest (e.g., AntWorld or WebWatcher) could provide for better prediction, but alternately may lose when the user's interest shifts (as is often the case when surfing the Web). For comparison, recall that WebWatcher [AFJM95, JFM97] compares the user's stated goal, the given page, and each link within it to other pages requested by

previous users, their stated goals, and the links they selected. Using assessed similarity on this information in a restricted domain, it picked three most likely links, from which the selected link was present 48.9% of the time. In a second, smaller trial, WebWatcher scored just 42.9% (as compared to humans at 47.5% on the same task). Our most comparable experiments have shown a lower accuracy (approximately 40%), but additionally they have been used on a general task in an unlimited domain, with no user goals nor past users for guidance. Thus, these experiments provide evidence of the applicability of content-based methods for predicting future Web pages access.

6.8 Summary

This chapter has examined the performance of Web request prediction on the basis of the content of the recently requested Web pages. We have compared text-similarity-based ranking methods to simple original link ordering and a baseline random ordering and found that similarity-based rankings performed 29% better than random link selection for prediction, and 40% better than no prefetching in a system with an infinite cache. In general, textual similarity-based rankings outperformed the simpler methods examined in terms of accuracy, but in the context of an infinite cache, the predictive performance of similarity and original rankings are fairly similar.

Most proposed Web prefetching techniques make predictions based on historical data (either from an individual's past or from the activity of many users as hints from servers or proxies). History-based prefetching systems can do well, when they have a model for the pages that a user is visiting. However, since users often request unseen (e.g., new) resources (perhaps 40% of the time [TG97]), history may not always be able to provide suggestions, or may have insufficient evidence for a strong prediction. In such cases, content-based approaches, such as those presented here, are ideal alternatives, since they can make predictions of actions that have never been taken by the user and potentially make predictions that are more likely to reflect current user interests.

Chapter 7

Evaluation

In the previous four chapters of this dissertation, we discussed two complementary approaches to Web prediction: history-based prediction and content-based prediction. The next five chapters are primarily concerned with the evaluation of prefetching techniques. While there are various procedures for evaluating prediction methods, they are likely to be insufficient as they perform measurements in isolation from the system in which they will be embedded.

7.1 Introduction

This chapter is particularly concerned with the general task of evaluation. Accurate evaluation of systems and ideas is central to scientific progress. Evaluation, however, is usually more than a binary result – not just success or failure. Typically, an evaluation of performance is really that of a measure suited for comparison with other systems or variations of the same system.

However, in all evaluations there is some measure of uncertainty in the results (especially for comparison), depending on what variables could be controlled. For example, system A might provide a large improvement in performance over system B for some workload. But if that workload is not sufficiently representative of the workloads in which A or B might be applied, then the experiment may have failed to truly determine the better system.

This is a common charge against the use of benchmarks. Conversely, benchmark proponents would cite the need for a standard test so that the results of many systems can be compared. However, even when a standard benchmark, such as SPECweb96 [Sys] is used, the test environment needs to be identical. Performance on a benchmark

can be affected by many variables, as pointed out by Iyengar and Challenger [IC97]. This can include different hardware, software (such as operating system and networking stack), configuration (parameter tuning, logging, disk assignments and partitioning, etc.). Thus, it is often difficult to fairly compare benchmark results when published by different sources.

A related problem arises when a standard benchmark is the primary mechanism against which a system is evaluated. To show performance improvements, system architects will often tune their system specifically to the benchmark in question (e.g., optimizing memory cache design [LW94a]). Unless the benchmark is particularly representative of real-world workloads, such optimizations result in even less confidence in the overall applicability of the evaluation results.

Thus, a valid question is whether a perfect evaluation methodology is possible. In an ideal evaluation methodology, the test would provide:

- accuracy — the evaluation correctly measures the performance aspect of interest.
- repeatability — the same test performed at some other time would provide the same results.
- comparability — evaluation results of one system could be fairly compared to the results of another.
- applicability — the results of evaluation fairly reflect performance of the system in practice.

Finally, most systems do not exist in isolation. Regardless of whether a system or program interacts with people or other systems, those interactions may change when system performance changes. This consideration raises new questions. Sometimes these effects can be managed, as when the other systems are also under experimental control, in which case the performance of the system as a whole is being evaluated as changes within one subsystem are made. However, when they are not under experimental control (such as a system that interacts with others across the Internet, or are used by people), the workload could change as performance changes. For example, if response

time improves, inter-request times may drop and more requests are made. Likewise, if response time gets worse, requests may be canceled or skipped entirely.

In the remainder of this section, we consider some of the issues raised above, but with a focus on the domain of Web and proxy cache evaluation.

7.1.1 Accurate workloads

Historically, the typical workload in Web caching research is that of a log of real usage, either from a Web server, or from a Web proxy, as in Figure 7.1. However, it is also possible to collect logs by packet sniffing (as in [WWB96, MDFK97, GB97]) or by appropriately modifying the browser to perform the logging (e.g., [TG97, CBC95, CP95]).

Those logs are then replayed within a simulation to measure cache performance. Unfortunately, in many cases, the representativeness of the logs is never questioned. Some of the most commonly used traces are logs of Web servers in the mid-1990s. While indeed those logs were once representative of real-world activity, they are not likely to be so today. While Web traffic and the number of deployed Web servers and proxies have exploded, the public availability of real-world traces has apparently decreased because of privacy concerns.

Assuming the availability of a representative trace of activity, that trace will need to be applied to a simulation. If the concerns of side-effects of replaying Web requests (which we will address in Chapter 12) are ignored, the possibility exists to replay the Web requests across a live network. In either case, the chances are that the environment in which the trace is being re-played will not exactly match the environment under which the trace was captured. For example, network properties (such as the bottleneck bandwidth to the rest of the Internet) may have changed (perhaps even intentionally). This suggests, at minimum, that the performance of the re-enacted system cannot be compared to the system under which the trace was captured. It also brings up the potential (as discussed above) that the trace may not be representative of the new environment, or of the new system. If the performance as seen by the user changes, the user may alter the behavior from that which might have otherwise occurred.

```
893252015.307 14 <client-ip> TCP_HIT/200 227 GET
  http://images.go2net.com/metacrawler/images/transparent.gif - NONE/- image/gif
893252015.312 23 <client-ip> TCP_HIT/200 4170 GET
  http://images.go2net.com/metacrawler/images/head.gif - NONE/- image/gif
893252015.318 38 <client-ip> TCP_HIT/200 406 GET
  http://images.go2net.com/metacrawler/images/bg2.gif - NONE/- image/gif
893252015.636 800 <client-ip> TCP_REFRESH_MISS/200 8872 GET
  http://www.metacrawler.com/ - DIRECT/www.metacrawler.com text/html
893252015.728 355 <client-ip> TCP_HIT/200 5691 GET
  http://images.go2net.com/metacrawler/images/market2.gif - NONE/- image/gif
893252016.138 465 <client-ip> TCP_HIT/200 219 GET
  http://images.go2net.com/metacrawler/templates/tips/../../images/pixel.gif -
  NONE/- image/gif
893252016.430 757 <client-ip> TCP_REFRESH_HIT/200 2106 GET
  http://images.go2net.com/metacrawler/templates/tips/../../images/ultimate.jpg -
  DIRECT/images.go2net.com image/jpeg
```

Figure 7.1: Excerpt of a proxy log generated by Squid 1.1 which records the timestamp, elapsed-time, client, code/status, bytes, method, URL, client-username, peerstatus/peerhost and objecttype for each request. It is also an example of how requests can be logged in an order inappropriate for replaying in later experiments.

There are additionally some non-user actions that will change. Requests for embedded resources (images, sounds, animations) are made at the time that the browser has parsed that portion of the encompassing page, and so when page retrieval performance changes, the request time for embedded resources will also change.

Even if the environment and system were identical to the original, the characteristics of playback can cause evaluation concerns. At one level, a Web log represents just a sequence of requests. If the only goal is to simulate replacement policies on a finite cache of unchanging objects, that sequence could be replayed at any rate and still produce identical results. Playback at maximum speed can be found in the research literature. However, it would not be realistic in that Web objects in the real world do change and have particular expiration times. This implies that a slower playback might have a higher probability of requesting objects that have expired, and that a faster playback would generate more cache hits than are warranted. Changes in playback speeds also imply changes in browser performance, connectivity, and user think times between requests, and in general may result in overall performance differences because of higher or lower demands for resources such as bandwidth.

Similarly, request inter-arrival rates need to be maintained. While a fixed request

rate might be simpler, the original request arrival times provide a strong contribution to the variation in network and server loads experienced. In addition, interactive prefetching systems rely on the existence of such "thinking time" — the time between page requests — to provide time in which to prefetch the objects likely to be requested next.

The above discussion of workloads has concentrated on the use of captured workloads. One alternative is the use of artificial workloads. Such workloads are designed to capture the relevant properties of real-world workloads, such as request inter-arrival and recurrence rates. Past research [CP95, CBC95] suggests a heavy tailed distribution of thinking times with a mean of 30 seconds, which can be replicated in carefully created workloads. Artificial workloads provide the advantage (and disadvantage) of being able to generate workloads that do not arise in practice. This is helpful when stress-testing a system for anticipated workloads, but also opens questions for evaluation. For example, even advanced artificial workloads (e.g., those in Web Polygraph [Rou02]) are unable to replicate the Web page relationships that the hypertext links of the Web provide for real datasets.

7.1.2 Accurate evaluation

As mentioned previously, accurate evaluation requires that the evaluation mechanism correctly measures the performance aspect of interest. This is sometimes overlooked in favor of evaluating metrics that are available. For example, early caching research often favored reports of increasing object hit rates. However, hit rates were never the real aspect of interest. Instead, they were a substitute metric for the real interests — reductions in bandwidth usage and improvements in user-perceived response times. In this case, byte hit rates (or better, a measure of all bytes received and generated by a system) would provide a real metric of byte savings. Similarly, user-perceived latencies and response times require significantly more careful evaluation of a more detailed system.

Finally, since Web systems are typically deployed for use by people, a human study

may be required. For example, if the goal is strictly to measure reduced bandwidth usage, the change in user-perceived latency associated with caching usage might prevent accurate measurement. Likewise, measures such as user-perceived response times are still one step removed — user satisfaction might be the ultimate goal, whose measurement would require a user study.

7.1.3 Accurate simulation

A prerequisite of accurate evaluation is accurate simulation. In this case, the simulation must not only be correct (i.e., without errors) but appropriately model various factors that contribute to desired performance. In Web caching research, desired performance includes the three major claimed benefits to caching on the Web: reduced bandwidth utilization, reduced server loads, and reduced client-side response times. Web bandwidth utilization can be modeled effectively with correct simulation, but the latter two may require extra care. To accurately model server loads, it may be necessary to model internal operating system and hardware characteristics (such as connection queue lengths and disk seek times). For accurate client-side response times, it may be necessary to model transport layer characteristics such as TCP slow start effects and user behaviors such as request cancellations (e.g., [FCD+99, CB98]). In both cases, accurate modeling of connection characteristics (multiple parallel connections, persistent connections) will be essential. These effects will need to be integrated into what might otherwise have been caching-only simulators, if accurate simulation of response times are to be achieved. Likewise, for prefetching simulators, the costs of prefetching must be included, such as time and costs for receiving server hints.

7.2 Cache Evaluation Methodologies

Since proxy caches are increasingly used around the world to reduce bandwidth requirements and alleviate delays associated with the World-Wide Web, this section describes a space of proxy cache evaluation methodologies and places a sampling of research within that space. The primary contributions of this section are threefold: 1) definition and

	Workload Source	
Algorithm Implementation	artificial	captured logs
simulated systems/network	A1	A2
real systems/isolated network	B1	B2

Table 7.1: A space of traditional evaluation methodologies for Web systems.

description of the space of evaluation techniques; 2) appraisal of the different methods within that space; and 3) a survey of cache evaluation techniques from the research literature.

It is useful to be able to assess the performance of proxy caches, both for consumers selecting the appropriate system for a particular situation and also for developers working on alternative proxy mechanisms. By evaluating performance across all measures of interest, it is possible to recognize drawbacks with particular implementations. For example, one can imagine a proxy with a very large disk cache that provides a high hit rate, but because of poor disk management it noticeably increases the client-perceived response times. One can also imagine a proxy that achieves a high hit rate by 1) caching images only for a very short time, which frees storage for use by other objects, and 2) prefetching the inline (embedded) images for the page currently requested. This would allow it to report high request hit rates, but not perform well in terms of bandwidth savings. Finally, it is possible to select a proxy based on its performance on a workload of requests generated by dialup users and have it perform unsatisfactorily as a parent proxy for a large corporation with other proxies as clients. These examples illustrate the importance of appropriate evaluation mechanisms for proxy systems.

7.2.1 The space of cache evaluation methods

As mentioned earlier, the most commonly used cache evaluation method is that of simulation on a benchmark log of object requests. In such a system, the byte and page hit rate savings can be calculated as well as estimates of latency improvements. In a few cases, an artificial dataset with the necessary characteristics (appropriate average and median object sizes, or similar long-tail distributions of object sizes and object repetitions) is used. More commonly, actual client request logs are used since they

	Workload Source		
Algorithm Implementation	artificial	captured logs	current requests
simulated systems/network	A1	A2	A3
real systems/isolated network	B1	B2	B3
real systems/real network	C1	C2	C3

Table 7.2: An expanded space of evaluation methodologies for Web systems.

arguably better represent likely request patterns and include exact information about object sizes and retrieval times.

We characterize Web/proxy evaluation architectures along two dimensions: the source of the workload used and form of algorithm implementation. Table 7.1 shows the traditional space with artificial versus captured logs and simulations versus implemented systems. With the introduction of prefetching proxies, the space needs to be expanded to include implemented systems on a real network connection, and in fact, evaluation of an implemented proxy in the field would necessitate another column, that of live requests. Thus, the expanded view of the space of evaluation methodologies can be found in Table 7.2 and shows that there are at least three categories of data sources for testing purposes, and that there are at least three forms of algorithm evaluation. While it may be possible to make finer distinctions within each area, such as the representativeness of the data workload or the fidelity of simulations, the broader definitions used in the expanded table form a simple yet useful characterization of the space of evaluation methodologies.

For the systems represented in each area of the space, we find it worthwhile to consider how the desired qualities of Web caching (reduced bandwidth requirements, server loads and improved response times) are measured. As will be seen below, each of the areas of the space represents a trade-off of desirable and undesirable features. In the rest of this section we point out the characteristic qualities of systems in each area of the space and list work in Web research that can be so categorized.

7.2.2 Methodological appraisal

In general, both realism and complexity increase as one moves diagonally downward and to the right in the evaluation methodology space. Note that only methods in areas A1,

A2, B1 and B2 are truly replicable, since live request stream samples change over time, as do the availability and access characteristics of hosts via a live network connection.

Simulated systems. Simulation is the simplest mechanism for evaluation as it does not require full implementation. However, simulating the caching algorithm requires detailed knowledge of the algorithms which is not always possible (especially for commercial implementations). Even then, typical simulation cannot accurately assess document retrieval delays [WA97].

It is also impossible to accurately simulate caching mechanisms that prefetch on the basis of the contents of the pages being served (termed *content-based prefetching*), such as those in CacheFlow [Cac02b], Wcol [CY97], and WebCompanion [Kle99] since they need at least to have access to the links within Web pages — something that is not available from server logs. Even if page contents were logged (as in [CBC95, MDFK97, Dav02a]), caches that perform prefetching may prefetch objects that are not on the user request logs and thus have unknown characteristics such as size and Web server response times.

Real systems/isolated networks. In order to combat the difficulties associated with a live network connection, measurement techniques often eliminate the variability of the Internet by using local servers and isolated networks, which may generate unrealistic expectations of performance. In addition to not taking into consideration current Web server and network conditions, isolated networks do not allow for retrieval of current content (updated Web pages).

Real systems and networks. Because the state of the Internet changes continuously (i.e., web servers and intermediate network connections may be responsive at one time, and sluggish the next), tests on a live network are generally not replicable. Additionally, to perform such tests the experiment requires reliable hardware, robust software, and a robust network connection to handle the workload applied. Finally, connection to a real network requires compatibility with, and no abuse of, existing systems (e.g., one cannot easily run experiments that require changes to standard httpd servers or experimental extensions to TCP/IP).

Artificial workloads. Synthetic traces are often used to generate workloads that

have characteristics that do not currently exist to help answer questions such as whether the system can handle twice as many requests per second, or whether caching of non-Web objects is feasible. However, artificial workloads often make significant assumptions (e.g., that all objects are cacheable, or that requests follow a particular distribution) which are necessary for testing, but not necessarily known to be true.

Captured logs. Using actual client request logs can be more realistic than artificial workloads, but since on average the lifetime of a Web page is short (less than two months [Wor94, GS96, Kah97, DFKM97]), any captured log loses its value quickly as more references within it are no longer valid, either by becoming inaccessible or by changing content (i.e., looking at a page a short time later may not give the same content). In addition, unless the logs are recorded and processed carefully, it is possible for the logs to reflect an inaccurate request ordering as the sample Squid logs show in Figure 7.1. Note that the request for the main page follows requests for three subitems of that main page. This is because entries in the log are recorded when the request is completed, and the timestamp records the time at which the socket is closed. Finally, proxy cache trace logs are inaccurate when they return stale objects since they may not have the same characteristics as current objects. Note that logs generated from non-caching proxies or from HTTP packet sniffing may not have this drawback. In other work [Dav99c], we further examine the drawbacks of using standard trace logs and investigate what can be learned from more complete logs that include additional information such as page content.

Current request stream workloads. Using a live request stream produces experiments that are not reproducible (especially when paired with live hardware/networks). Additionally, the test hardware and software may have difficulties handling a high real load. On the other hand, if the samples are large enough and have similar characteristics, an argument for comparability might be made.

7.2.3 Sampling of work in methodological space

In this section we place work from the research literature into the evaluation space described above. Note that inclusion/citation in a particular area does not necessarily

indicate the main thrust of the paper mentioned.

Area A1: Simulations using artificial workloads. It can be difficult to characterize a real workload sufficiently to be able to generate a credible artificial workload. This, and the wide availability of a number of proxy traces, means that relatively few researchers attempt this kind of research:

- Jiang and Kleinrock [JK97, JK98] evaluate prefetching mechanisms theoretically (area A1) but also on a limited trace set (area A2).
- Manley, Seltzer and Courage [MSC98] propose a Web benchmark which generates realistic loads to focus on measuring response times. This tool would be used to perform experiments somewhere between areas A1 and A2 as it uses captured logs to build particular loads on request.
- Tewari *et al.* [TVDS98] use synthetic traces for their simulations of caching continuous media traffic.

Area A2: Simulations using captured logs. Simulating proxy performance is much more popular than one might expect from the list of research in area A1. In fact, the most common mechanism for evaluating an algorithm's performance is simulation over one or more captured traces.

- Cunha *et al.* [CBC95], Partl [Par96], Williams *et al.* [WAS+96], Gwertzman and Seltzer [GS96], Cao and Irani [CI97], Bolot and Hoschka [BH96], Gribble and Brewer [GB97], Duska, Marwood and Feeley [DMF97], Caceres *et al.* [CDF+98], Niclausse, Liu and Nain [NLN98], Kurcewicz, Sylwestrzak and Wierzbicki [KSW98], Scheuermann, Shim and Vingralek [SSV97, SSV98, SSV99], and Zhang *et al.* [ZIRO99] all utilize trace-based simulation to evaluate different cache management algorithms.
- Trace-based simulation is also used in evaluating approaches to prefetching, such as Bestavros's server-side speculation [Bes95], Padmanabhan and Mogul's persistent HTTP protocol along with prefetching [PM96], Kroeger, Long and Mogul's

calculations on limits to response time improvement from caching and prefetching [KLM97], Markatos and Chronaki's object popularity-based method [MC98], Fan, Cao and Jacobson's response time reduction to low-bandwidth clients via prefetching [FJCL99] and Crovella and Barford's prefetching with simulated network effects [CB98].

- Markatos [Mar96] simulates performance of a Web server augmented with a main memory cache on a number of public Web server traces.

- Mogul *et al.* [MDFK97] use two large, full content traces to evaluate delta-encoding and compression methods for HTTP via calculated savings (area A2) but also performed some experiments to include computational costs on real hardware with representative request samples (something between areas B1 and B2).

Area A3: Simulation based on current requests. We are aware of no published research that could be categorized in this area. The algorithms examined above in areas A1 and A2 do not need the characteristics of live request streams (such as contents rather than just headers of HTTP requests and replies). Those researchers who use live request streams all use real systems of some sort (as will be seen below).

Area B1: Real systems on an isolated network using an artificial dataset. A number of researchers have built tools to generate workloads for Web systems in a closed environment. Such systems make it possible to generate workloads that are uncommon in practice (or impossible to capture) to illuminate implementation problems. These include:

- Almeida, Almeida and Yates [AAY97a] propose a Web server measurement tool (Webmonitor), and describe experiments driven by an HTTP load generator.

- Banga, Douglis and Rabinovich [BDR97] use an artificial workload with custom client and server software to test the use of transmitting only page changes from a server proxy to a client proxy over a slow link.

- Almeida and Cao's Wisconsin Proxy Benchmark [AC98b, AC98a] uses a combination of Web client and Web server processes on an isolated network to evaluate

proxy performance.

- While originally designed to exercise Web servers, both Barford and Crovella's SURGE [BC98a] and Mosberger and Jin's httperf [MJ98] generate particular workloads useful for server and proxy evaluation.

- The CacheFlow [Cac00] measurement tool was designed specifically for areas C1 and C2, but could also be used on an isolated network with an artificial dataset.

Area B2: Real systems on an isolated network using captured trace logs. Some of the research projects listed in area B1 (using artificial logs) may be capable of using captured trace logs. In addition, we place the following here:

- In evaluating their Crispy Squid, Gadde, Chase and Rabinovich [GCR98] describe the tools and libraries called Proxycizer. These tools provide a trace-driven client and a characterized Web server that surround the proxy under evaluation, much like the Wisconsin Proxy Benchmark.

Area B3: Real systems on an isolated network using current requests. Like area A3 which used simulation, we found no research applicable to this area. Since live requests can attempt to fetch objects from around the globe, it is unlikely to be useful for testing forward proxies within an isolated network. However, we suggest that reverse proxies could be tested internally under this model.

Area C1: Real systems on a live network using an artificial dataset. Some of the research projects in area B1 (e.g., CacheFlow's measurement tool) may be designed for the use of a live network connection. The primary restriction is the use of real, valid URLs that fetch over the Internet rather than particular files on a local server.

Area C2: Real systems on a live network using captured logs. With captured logs, the systems being evaluated are as realistically operated as possible without involving real users as clients. In addition to those listed below, some of the tools from area B1 may be usable in this type of experiment.

- Wooster and Abrams [Woo96, WA97] report on evaluating multiple cache replacement algorithms in parallel within the Harvest cache, both using URL traces and online (grid areas C2 and C3, respectively) but the multiple replacement algorithms are within a single proxy. Wooster also describes experiments in which a client replayed logs to multiple separate proxies running on a single multiprocessor machine.

- Maltzahn and Richardson [MR97] evaluate proxies with the goal of finding enterprise-class systems. They test real systems with a real network connection and used carefully selected high load-generating trace logs.

- Liu *et al.* test the effect of network connection speed and proxy caching on response time using public traces [LAJF98] on what appears to be a live network connection.

- Lee *et al.* [LHC+98] evaluate different cache-to-cache relationships using trace logs on a real network connection.

Area C3: Real systems on a live network using the current request stream. In many respects, this area represents the strongest of all evaluation methodologies. However, most real installations are not used for the comparison of alternate systems or configurations, and so we report on only a few research efforts here:

- Chinen and Yamaguchi [CY97] described and evaluated the performance of the Wcol proxy cache on a live network using live data, but do not compare it to any other caching systems.

- Rousskov and Soloviev [RS98] evaluated performance of seven different proxies in seven different real-world request streams.

- Cormack [Cor98] described performance of different configurations at different times of a live Web cache on a live network.

- Later in Chapter 10 we will describe the Simultaneous Proxy Evaluation (SPE) architecture that compares multiple proxies on the same request stream. While

originally designed for this area with a live request stream and live network connection, it can also be used for replaying captured or artificial logs on isolated or live networks (areas B1, B2, C1, and C2).

7.3 Recommendations for Future Evaluations

While almost every researcher uses some kind of evaluation in their work, there are a number of pitfalls that may be encountered. The better papers will point out the drawbacks of their methodology, but not all do. In this section, we consider many of the typical mistakes that can be found in existing caching research.

Using an inappropriate trace. Some publicly available traces are more than a few years old[1] and thus may no longer be representative of current network traffic. In some cases, however, these are the only ones available with the desired characteristics. Ideally traces would be large, representative of the desired traffic, and be publicly accessible for others to examine and possibly reconstruct tests.

Hiding effects of changes to subsystem. When comparing an artificial or otherwise modified environment to a real-world trace, there are a multitude of factors for which accounting must be made. An example of this is the use of a proxy trace in replaying through a Web browser. Unless specially configured, the browser may use its built-in cache and no longer generate as many external requests as found in the original trace.

Ignoring the effect of connection caching. Connection caching [CKZ99] can be a significant effect of the use of persistent connections in proxies. When a proxy is introduced to an environment, the effects of persistent connections and the caching of connections by the proxy should be considered.

Ignoring the effect of differences between HTTP/1.0 and HTTP/1.1. HTTP/1.1 [FGM+99] introduced a number of improvements over HTTP/1.0 [KMK99], including persistent connections as default behavior and increased support for caching.

[1] Unfortunately, many of the traces used in this dissertation have also gotten old. We encourage ISPs, content providers, and hosting providers to make traces of Web usage publicly available for analysis (after suitable anonymization).

The use of a trace containing one kind of traffic for replaying in another environment should not be made blindly. HTTP traffic measurements today are likely to see both protocols in widespread use.

Ignoring TCP effects on HTTP performance. TCP introduces particular effects for the relatively short-lived HTTP connections, including costs associated with slow start and connection establishment [PK98, SCJO01].

Ignoring canceled requests. While relatively few public traces include information about canceled requests, some studies (e.g., [FCD+99]) have revealed the relatively high presence of such activity. Inappropriate use of such requests in logs (such as fully retrieving them or assuming the sizes reflect a new, smaller object) may distort actual performance of a system.

Ignoring effects of replaying trace. When a trace is replayed, it really does not faithfully replicate what users would do or see in a new environment. For example, the network and destination hosts may have changed content or accessibility. More importantly, the user is likely to have acted differently — when responses are made faster, the user might make requests faster. Or vice versa — if a response was much slower, the user might give up and cancel the request and move on to something else.

Ignoring real timing issues. For those researchers interested in latency and response time effects, it is not sufficient to measure performance in terms of round trip times. As mentioned above, real networking effects may be needed to get accurate estimates of performance.

Obscuring pertinent details of test infrastructure. For example, the use of old proxy cache software (e.g., the CERN proxy [LNBL96]) often implies a lack of support for HTTP/1.1 and possibly persistent connections under HTTP/1.0 as well. Therefore, such details are important to disclose.

Ignoring concerns of freshness in a trace. Since most Web usage traces do not include information containing a response's expiration, a common approach is to test for changes in response size to determine a change in the underlying object. While this is not ideal, it is better than simpler (but wrong) approaches that only check for recurrence of the object request. Likewise, simulations that hold data in a cache forever

are not realistic — many objects should not be cached at all, or only for a short period.

Ignoring other cacheability information. In addition to the concerns mentioned above, there are many factors involved in determining cacheability. For example, under HTTP/1.0, the presence of a cookie can make a response uncacheable. Likewise, many researchers mistakenly assume that uncacheable responses can be determined from the format of the URL (e.g., the inclusion of a '?') when under HTTP/1.1, no such *a priori* relationship exists.

Ignoring DNS issues when building a prototype. The domain name system can be a significant source of slow response times [CK00, CK01a, STA01]. Thus, DNS effects must be considered; for example, prefetching DNS can provide real savings.

Hiding or ignoring bad data. Some researchers choose to drop requests believed to be dynamic from their traces or to eliminate error responses. As long as it is well documented, the practice may be acceptable, but a better, more general approach would be to incorporate all requests into the model.

Vague or missing latency calculations. Since caching and prefetching can provide significant benefits to user-perceived response times, it is imperative that latency and response time calculations be explicitly described.

The concerns presented above are not meant to be exhaustive. Instead, they simply represent many of those raised by the author while reviewing publications cited in this dissertation.

7.4 Summary

Evaluation is a significant concern, both for consumers and for researchers. Objective measurements are essential, as are comparability of measurements from system to system. Furthermore, it is important to eliminate variables that can affect evaluation.

Selection of an appropriate evaluation methodology depends on the technology being tested and the desired level of evaluation. Some technologies, such as protocol extensions, are often impossible to test over a live network because other systems do not support it. Similarly, if the goal is to find bottlenecks in an implementation, one

may want to stress the system with very high synthetic loads since such loads are unlikely to be available in captured request traces. In fact, this reveals another instance in which a live network and request stream should not be used — when the goal of the test is to drive the system into a failure mode to find its limits.

In general, we argue for the increased believability of methodologies that are placed further down and to the right in the evaluation space when objectively testing the successful performance of a Web caching system. If the tested systems make decisions based on content, a method from the bottom row is likely to be required. When looking for failure modes, it will be more useful to consider methodologies near the upper left of the evaluation space.

This survey provides a sampling of some of published Web caching research and presents one of potentially many spaces of evaluation methodologies. In particular, we haven't really considered aspects of testing cache hierarchies and inter-cache communication [CDN+96, Nat02, Dan98, CC95, CC96, ZFJ97, GRC97, GCR97, FCAB98, RCG98, WC97b, WC97a, RW98, VR98, VW00], cache consistency, or low-level protocol issues such as connection caching which are significant in practice [CDF+98].

In this chapter we have described many concerns for accurate, repeatable, comparable, and applicable evaluation methodologies. We have additionally described a space of evaluation methodologies and shown where a sample of research efforts fall within it. By considering the space of appropriate evaluation methodologies, one can select the best trade-off of information provided, implementation costs, comparability, and relevance to the target environment.

In Chapter 8 we will describe a simulator that can be used to develop and evaluate some types of caching and prefetching approaches. Subsequently, in Chapter 9 we will use that simulator to combine predictions from multiple sources and report on the efficacy of the approach. As we described in this chapter, simulation has limitations as well, and so in Chapters 10 and 11, we propose and develop a proxy cache evaluation architecture and test it with real systems. Finally, Chapter 12 concludes the dissertation by looking ahead to the next steps that will be necessary to make wide-spread prefetching possible on the Web.

Chapter 8

A Network and Cache Simulator

8.1 Introduction

Simulation is a common approach for algorithmic research, and is widespread in Web caching research (as we saw in Chapter 7). In general, simulation provides many benefits, including the ability to:

- test scenarios that are dangerous or have not yet occurred in the real world,
- predict performance to aid technology design,
- predict expected behavior of new protocols and designs without implementation costs or disruption of existing systems,
- slow or speed time for effective analysis, and
- quickly survey a range of potential variations.

These features make simulation valuable to Web research, especially early in the research and development cycle. In this chapter, we will explore the use of Web simulation to evaluate caching and prefetching schemes. Unlike much early caching research, we use simulation to estimate response times, rather than just object and byte hit rates. This focus is needed because response time improvement is a common justification for the use of Web caching, and is arguably the initial *raison d'être* for content delivery networks.

A simulator is only useful when it produces *believable* results. Believability, however, is a subjective quality, and typically only comes after the simulator has been validated by exhaustive testing and widespread use. Heidemann *et al.* [HMK00] provide one

definition of validation as the "process to evaluate how accurately a model reflects the real-world phenomenon that it purports to represent."

Thus, in addition to introducing a new caching simulator, this chapter will need to provide some measure of validation so that it may be found believable. It is our intent to demonstrate the validity of the HTTP response times as estimated for the client in a new HTTP simulator by comparison to real-world performance data at both a small scale (individual transactions, carefully measured) and at a large scale (tens of thousands of clients and servers from a log of real-world usage).

The rest of this section will provide an overview of the new simulator as well as the motivation to develop it. Section 8.2 describes the implementation and organization of the simulator. In Section 8.3 we break from the simulator and instead describe a large dataset of real-world traffic which we will use later in part of Section 8.4 in which we validate the simulator against small-scale and large-scale real-world datasets. Section 8.5 will provide additional demonstrations of the utility of the simulator. In Section 8.6 we wrap up the chapter by comparing our simulation work to others and consider future work.

8.1.1 Overview

NCS (Network and Cache Simulator) is an HTTP trace-driven discrete event simulator of network and caching activity. It is highly parameterized for maximal compatibility with previous caching and network simulations. In granularity, it resides between the high-level caching-only simulators prominent in much Web caching research (e.g., [Par96, DMF97, GPB98, BH98a, HWMS98, BBBC99, ZIRO99, BKNM02]), and the detailed simulators of networking protocols and traffic. In an effort to capture estimates of user-perceived response times, it simulates simplistic caching and prefetching functionality at various locations in a network comprising of client browsers, an optional intermediate proxy, and Web servers. Caching is optionally supported at the proxy and clients. Additionally, it simulates many aspects of TCP traffic among these entities on a somewhat idealized network.

The development goals for NCS included:

- Estimate client-side response times and bandwidth usages by
  - Capturing intrinsic network effects (e.g., new connection costs, TCP slow start).
  - Modeling real topologies (including browsers, proxies, and servers, with potentially multiple connections and/or persistent connections).
  - Capturing real-world network effects (including distributions of response times and bandwidths).
- Provide credibility – be able to compare simulation results to real-world numbers.
- Incorporate optional prefetching techniques for testing and evaluation.

While most Web caching simulators measure hit rates and bandwidth used, few consider the detail needed to estimate response times believably. In contrast, NCS is specifically designed to estimate the response times experienced by the user, and so includes network simulation in addition to a caching implementation.

This chapter motivates the design, lists the features, provides an overview of the implementation, and presents some sample experiments to demonstrate the utility of the simulator.

8.1.2 Motivation

As mentioned in Chapter 7, evaluation of Web caching research has varied widely among dimensions of measurement metrics, implementation forms (from logic arguments, to event simulations, to full implementation), and workloads used. Even when restricting one's view to caching-oriented simulators (e.g., those used in [WAS+96, CDF+98, FCD+99, FJCL99, ZIRO99, BH00, DL00]), it is apparent that there is a wide range of simulation detail. However, it has also been noted that some details are of particular importance [CDF+98], such as certain TCP slow-start network effects and HTTP connection caching when trying to capture estimates of response times.

Caching in the Web is known to reduce network bandwidth, server loads, and user-perceived latency. As overall bandwidth in the net improves and becomes less expensive

and more experience is gained in managing loads of popular servers, the final characteristic of latency improvement grows in interest among caching researchers. This, in fact, was the initial motivation for the development of NCS. As we have seen in earlier chapters, prefetching is a well-known approach to reduce response times. However, a significant difficulty in Web prefetching research is the lack of good methods for evaluation. Later in Chapter 10 we will propose a mechanism for evaluation of fully implemented systems that may employ prefetching, but a simulator is more appropriate for the earlier feedback needed in any research effort.

Therefore, NCS has been designed to estimate the client-perceived response times by simulating the network latencies and the effects of caches present in the network. In addition, it contains optional prediction code described in Chapter 4 to provide various levels of prefetching wherever caching is available.

Although inspired by the caching simulators described and used elsewhere [FCD+99, FJCL99, DL00], the coding and development of NCS has proceeded independently. An alternative might have been to use the network simulator *ns* [UCB01] and extend it appropriately for prefetching. However, *ns* is also known to have a steep learning curve [BEF+00], and would require significantly more computational resources because of the more detailed modeling it performs. Instead, initial NCS prototypes were quickly implemented in Perl and appeared quite promising. By using a less-detailed network model, and a current implementation in C, NCS is able to estimate performance for traces using tens of thousands of hosts much faster than real-time. In summary, we wanted more detail (and response-time accuracy) than typical caching-only simulators, but faster simulation times for large experiments than the fine-grained network-oriented simulator *ns*, and to be able to incorporate code that optionally estimates the effects of prefetching.

The code for HTTP over TCP, and especially for slow-start effects, is based significantly on the models shown in [THO96, HOT97, KR00]. Heidemann *et al.* [HOT97] claim their model provides guidance for wide-area and low-bandwidth network conditions, but may be somewhat inaccurate when applied to LANs. Therefore, we anticipate

similar performance, which will be shown in Section 8.4.2. Fortunately, the area of interest for most caching simulations is not that of LANs, and so we expect that the simulation of network effects over wide-area and low-bandwidth conditions will dominate the occasional inaccuracy in the modeling of LAN conditions.

8.2 Implementation

In general, NCS is designed to be flexible so that a large variety of simulated environments can be created. We had particular goals of replicating many of the first order characteristics of existing browsers and proxies (such as caching of data, DNS, and connections). The suggested values were often derived empirically (as in [WC98]) or by direct examination of publicly available source code (for Netscape Communicator [Net99] and Squid [Wes02]).

One perceived drawback of a large parameter set is that all parameters must have a value, even when trying to simulate a fairly simple environment. Conversely, this is better viewed as a feature that makes design choices explicit, rather than implicit as in most simulators.

8.2.1 Features

A quick summary of the features of NCS include:

- The use of a simple form of server and proxy logs as input.
- Optional support of persistent and/or pipelined connections, and more than one connection to the same host.
- Idle connections can be dropped after timeouts, or when a new connection is needed but the max number of connections have been established.
- Provisions for an arbitrary number of clients, an optional intermediate proxy, and an optional number of servers.
- Support for caching at clients and intermediate proxies.

- Parameterized link bandwidths and latencies as input to an emulation of TCP slow-start (with some limitations such as assuming an infinite slow-start threshold).
- Three versions of TCP slow-start: naive slow-start, BSD-style starting with payload of size 2, and a version that uses delayed ACKs.
- Optional inclusion of a simple model for parameterized DNS lookup cost with DNS caching at clients and proxies.
- Optional ability for proxies to buffer received files in their entirety before sending, or can forward data as they are being received.
- Support for proxies with multiple requests for the same item to either open multiple connections, or wait until the first is received and then serve the second request from cache.

With regard to prefetching, it supports prefetching at clients and/or at proxies. It can model the transmission of hints from server to client at various points in the process. Hints can then be integrated into prediction models. It does not prefetch items that are already being retrieved.

Alternatively, NCS does make a number of simplifying assumptions. At the networking level, it always sends (and receives) packets in order, and there is no packet loss. It also ignores TCP byte overheads as well as ACK bandwidth (although ACK timing is used). For some calculations, it also ignores response header bytes and request bandwidth when determining bandwidth resource availability. For most workloads, this is a not a significant issue. Finally, it assumes a simple model of a server — in particular that the server (and its subsystems) are not the bottleneck (e.g., through the use of appropriate file and disk management as in [MRG99, MKPF99]).

As a result of implementing more features than a caching-only simulator but at a lower granularity than *ns*, the runtime of NCS is orders of magnitude faster than *ns*, but slower than a well-written caching simulator, such as that written by Pei Cao while at the University of Wisconsin [Cao97] (and used in [FJCL99]). For a small, 10,000

request subset of the NASA trace, the UW simulator completed in .2 seconds on a 1Ghz Intel Pentium III under Linux. NCS took 1.8 seconds on the same platform, but additionally incorporated network delay characteristics to estimate client response time improvements. In contrast, *ns* took more than 30 minutes ($1.8x10^3$ seconds). To simulate the entire trace with approximately 850,000 requests, the UW simulator needs just seven seconds; NCS requires 165 seconds. Unfortunately, *ns* crashed when building the larger topology needed for the tens of thousands of hosts represented in the trace. When run on a simpler topology of approximately 900 hosts, *ns* required more than three days to complete ($2.8x10^5$ seconds) on the same platform, and was only CPU bound.

8.2.2 NCS parameters

The simulator takes just two parameters at the command line. The first is the name of the primary parameter file (described next), and an optional debugging level (increasing values present more detail).

All parameter files have the following form: each parameter is specified, one per line, in the correct order. Since there is just one value used on each line, anything after a whitespace on the same line is ignored.

The simulator reads an entirely numerical trace of requests, one per line, in a log to generate events. The format of the trace file is as follows: the first 6 white-space delimited values are required (client-id, time, request-id, response-code (e.g., 200, 304, etc.), size, and cacheable (binary value)); the server-id is the only item in the first optional group (if not present, all requests go to the same server); the second optional group includes three more values (elapsed-request-time in ms, the last modification time, and the server-request time in ms).

```
#client-id time req-id code size cacheable [server-id [elapsedms lm serverms]]
```

Most Web traces contain a subset of the values listed above. In the tests we describe, we have converted an existing Web trace (from a proxy or server) into the format described. Here is an excerpt of one trace with all parameters:

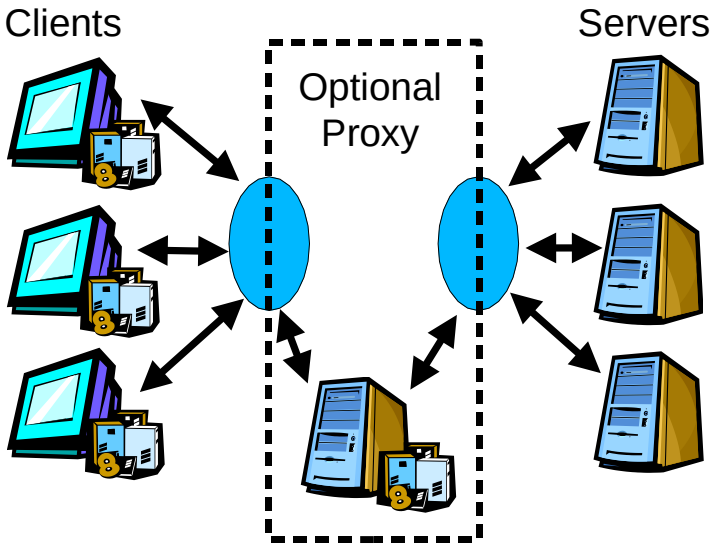


Figure 8.1: NCS topology. NCS dynamically creates clients and servers from templates as they are found in the trace. It can be configured to use caching at clients and at an optional proxy that can be placed logically near clients or servers.

```
46 846890365.49576 142 200 368 1 51 -1 -1 0.796008
21 846890365.508394 143 200 10980 1 23 -1 833504726 3.505611
47 846890365.519689 144 200 12573 1 52 -1 845000914 19.612891
47 846890365.520312 145 200 6844 1 52 -1 846823803 18.945683
48 846890365.555019 146 200 1340 1 53 -1 845410505 14.995183
```

The timings record the number of seconds elapsed since 1 January 1970. Thus these requests were made on 1 November 1996, from four different clients to five different servers. All requests were satisfied (response code 200). The first one does not include a last-modified time. None provide an overall elapsed time, but the servers had response times ranging from approximately .8ms to 19.6ms.

8.2.3 Network topologies and characteristics

As suggested by the parameters described in Section 8.2.2, the simulator is easily configured for a particular set of network topologies. It currently allows for a single set of clients which are each connected at identical bandwidths and latencies. Those clients send their requests either to a single proxy, if used, or to the origin servers. Each server is likewise identically configured with fixed bandwidths and latencies, representing the

typical characteristics of all servers. See Figure 8.1. This particular representation allows for easy management of nodes and parameter files. However, it may be of interest in the future to allow for more complex network topologies and characteristics. For example, it may be useful to have multiple proxies so that different sized sets of clients can be evaluated, or to have some clients use a proxy and others fetch directly. Alternatively, it could be desirable to have multiple hierarchically connected proxies to test performance with proxy chains.

8.2.4 Manual tests

A variety of simple manual tests were performed while debugging the simulator. These included tests of network latencies for request loads with few items, tests of the cache replacement algorithms, and tests for persistent connections, etc. However, since manual tests for debugging and validation purposes require significant human effort, they can only be used for trivial experiments. In addition to the manual tests used for debugging, the experimental performances in Section 8.4.2 were also calculated manually to verify their accuracy, because they were feasibly small. For larger experiments, therefore, manual calculations are unreasonable, and so we depend on alternative methods for validation, as described below in Sections 8.4.3 and 8.4.4.

8.3 The UCB Home-IP Usage Trace

In this section we describe the UC Berkeley Home IP HTTP Traces [Gri97] and some of the effort needed to prepare the logs for our use. This dataset was not selected arbitrarily, but was chosen specifically because of its non-trivial length, recorded timing characteristics, and because it is well-known by researchers. A longer version of this trace has been analyzed in depth by Gribble and Brewer [GB97], but the public version described here has also been used in numerous published papers (e.g., [FCAB98, JBC98, BCF+99, THVK99, FJCL99, BS00, LYBS00, MIB00, PF00, RID00]).

8.3.1 Background

As mentioned previously in Chapter 4, the UC Berkeley Home IP HTTP Traces [Gri97] are a record of Web traffic collected by Steve Gribble as a graduate student in November 1996. Gribble used a snooping proxy to record traffic generated by the UC Berkeley Home IP dialup and wireless users (2.4Kbps, 14.4Kbps, and 28.8Kbps land-line modems, and 20-30Kbps bandwidth for the wireless modems). This is a large trace, covering 8,377 unique clients over 18 days with over nine million requests. His system captured all HTTP packets and recorded, among other items, the time for each request, the first byte of response, and the last byte of response. These timestamps provide a sample of real-world response times that can be used to validate our simulator.

8.3.2 Trace preparation

Most researchers have found that Web traces need to be checked and often cleaned before using them in a simulator or for evaluation [KR98, Dav99c]. The UCB Home-IP Trace is no exception.

Like many proxy logs, the events in this trace are recorded in order of event completion. This can cause anomalies in sequence analysis and when replaying the trace for simulation, and instead must be sorted by request time.

Unfortunately, this trace does not record the HTTP response code associated with each object. Thus, we are unable to distinguish between valid responses (e.g., code 200), error responses (e.g., 404), and file not modified responses (304). For the purpose of simulation, we assume all responses contain a 200 response code.

While attractive for timing simulations, this trace also includes some anomalies. One example of this is impossibly high client bandwidths: for certain responses, the combination of size of reply and timings of request and end of response suggest bandwidths that meet or exceed LAN capabilities, and which certainly do not reflect dialup

or wireless clients of 1996.[1] More importantly, the trace does not directly reflect client-perceived response times which is the timing estimate provided by NCS. Since the trace is captured by a snooping proxy on an Ethernet on the way to the university's Internet connection, two modem (dialup or wireless) round trips are not captured (first round trip initiates the connection, second corresponds to the time to make a request and start receiving the last response), nor is the modem transmission time of the initial packets to open a connection. There is also the transmission cost of the request packet and final data packet (or more, as will be discussed in the next section). Therefore, in order to compare the original trace timings to our simulated client response times, we will add a factor of up to 667ms, reflecting typical timings for a 28.8 modem (which underestimates the higher latencies of wireless modems as well as slower dialup modems).[2] Prior to the adjustment, the trace had a mean response time of 17.4s and a median of 2.9s; afterward it had a mean and median of 18.0s and 3.6s, respectively.

The UCB Home-IP trace also misses the latencies associated with DNS lookups. However, since the simulator can also ignore this, we don't need to be concerned about it for the purposes of validation.[3]

8.3.3 Analysis

We analyzed statistics and distributions of the UCB Home-IP Trace to better understand the activity represented, and to look for anomalies. First we consider the response size distribution. Figure 8.2 shows the cumulative distribution function of response sizes from this usage trace. We found that 25% of all responses were 430 byes or smaller,

[1]Therefore, in our simulations, we drop more than half a million anomalous entries from the trace (primarily those with missing timestamp fields, corresponding to missed packets or canceled requests, but also 13 entries with server-to-proxy bandwidth over 80Mbps, and 7237 entries with size > 100MB).

[2]The value 667ms can be calculated as follows. We assume a modem round-trip time of 200ms, and effective bandwidth of 3420 bytes/sec (to account, in part, for the overhead of TCP/IP header bytes and PPP framing bytes). The transmission time to send and receive the initial packet to set up a new connection is 22.2ms (40 bytes sent and received at the raw 3600 bytes/sec), the time to transmit the request is 95ms (325 bytes at 3420 bytes/sec), and the time to receive the last packet is estimated to be 149.7ms (512 bytes at 3420 bytes/sec). The total delay is then the sum of two round-trip times, the transmit time to send and receive initial packet to set up a new connection, the time to transmit a request, and the time to send a final response packet.

[3]In general, though, DNS effects can contribute significantly to user-perceived retrieval latencies [CK00, CK01a, STA01] and ought to be included in simulation and analysis of Web delays.

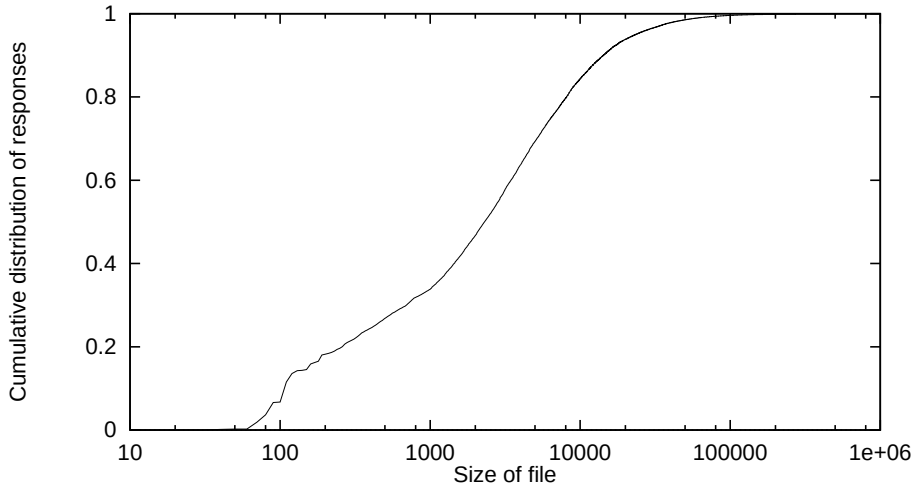


Figure 8.2: Cumulative distribution function of response sizes from the UCB Home-IP request trace.

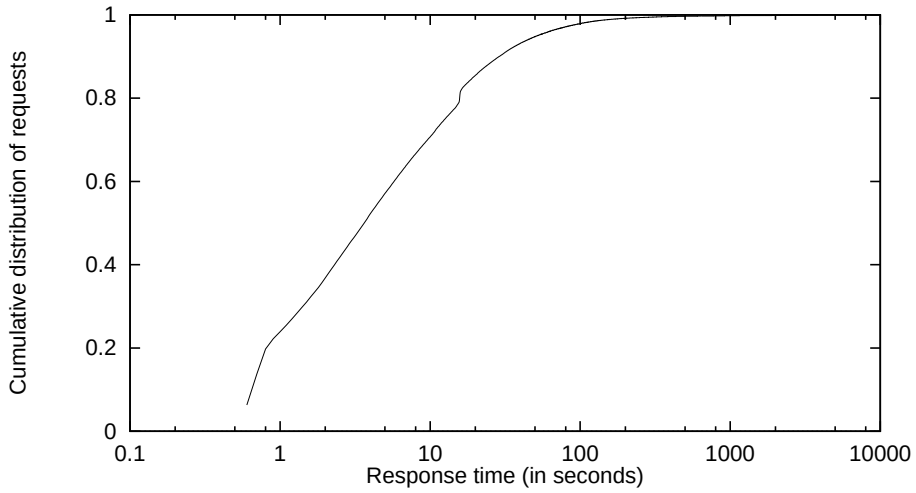


Figure 8.3: Cumulative distribution function of client-viewed response times from the UCB Home-IP request trace.

and that the median response size was approximately 2310 bytes. This compares to a mean response size of approximately 6.7KB.

An obvious measure of user-oriented performance for a Web system is the response time. Figure 8.3 shows the cumulative distribution function of response times. In examining this trace, we found that the first quartile of the response time distribution is at 1.0s, and the median is at 3.6s. The mean response time is much higher at 18.0s.

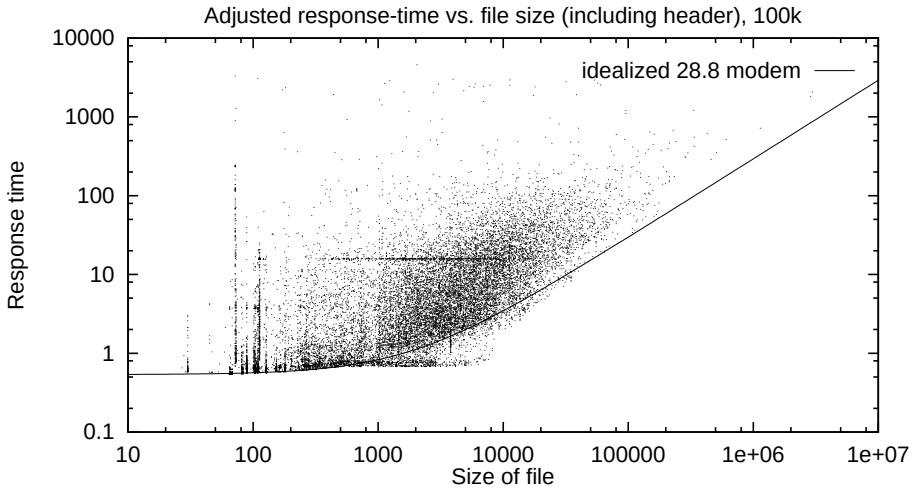


Figure 8.4: Scatter plot of file size vs. estimated actual response time for first 100k requests from UCB trace.

In order to better understand the distributions involved, we also examined the relationship between the size of a file transfered, and the response time for that file. In Figure 8.4, the actual response time vs. size of file is plotted for the first 100,000 requests in the UCB trace. We also plot the idealized performance of a 28.8 modem. Note the large number of points along the lower-right edge that correspond to transmission bandwidths that are higher than expected (below the 28.8 modem line). For example, there are many points at 8000 bytes with a response time of less than 1s. Assuming 1s, this corresponds to 64000 bits per second, which is clearly above the bandwidth limits for dialup and wireless technologies of 1996. We have two conjectures that provide plausible explanations. One is that these points (contributing to the fuzziness along the lower right edge) is an artifact of packet buffering at the edge device to which the modems were attached (such as a terminal server). The second possibility is that we are seeing the effect of data compression performed by the modems. Note that if we were to plot the original dataset (without filtering obvious anomalies), we would see even more extremal points to the lower right.

In the next section, we will compare simulated timing results with those shown here.

8.4 Validation

Verification and validation [Sar98] are essential if the results of those simulations are to be believed. This is particularly important when the simulated experiments are not easily replicated (e.g., perhaps because of extensive development effort required, or proprietary log data). Lamentably, even limited validation of simulators has been uncommon in the Web caching community.

We can validate either network effects or caching effects, or their combination. To limit variation when making comparisons, we will attempt to use the same data set published previously to the extent possible (including the UCB dataset described above). Unfortunately, the results in many papers use proprietary data that cannot be shared. We describe related work later in Section 8.6.1.

In this section we will use real-world data to validate simulated HTTP network performance at both a small scale (individual transactions, carefully measured) and at a large scale (tens of thousands of clients and servers from a log real-world usage).

First we discuss the validation process necessary for our Web performance simulator. Then we compare simulated end-user response times with real-world values at small and large scales. Finally, we compare caching performance with another respected simulator.

8.4.1 The validation process

Typically it is impossible for a simulator to exactly mirror real-world experience. NCS is no exception. In NCS, we make assumptions and simplifications as a trade-off for faster completion of simulations. For example, NCS ordinarily uses fixed parameters for many aspects of client, network, and server performance. In the real world, these values would be a function of changing world conditions. Different servers would have different connectivity characteristics (including network latency and bandwidth) and service loads at different times. Dial-up clients will likewise differ in bandwidths and latencies depending on the hardware, software, and phone lines used. NCS only models the traffic captured in the input trace, ignoring unlogged traffic (e.g., non-HTTP traffic like HTTPS, streaming media, FTP, or traffic destined for non-logged clients or servers).

Given that the simulation will not generally *replicate* real-world experiences, what can be done? Instead, we can use the simulator to repeat simple real-world experiments, and thus validate the simulator by comparing it to both real-world results and the calculated results of others (which we do in Section 8.4.2). In particular, we can ask whether the same effects are visible, and most importantly, verify that the simulator works as expected (in general, similarly to other results with some deviation as a result of the simplifications made).

8.4.2 Small-scale real-world networking tests

Fortunately, some networking researchers have taken the time to validate their networking models by comparing them to real-world results (e.g., [HOT97]). By publishing such work in detail, others are not only able to validate new simulators, but also to compare the fidelity of various theoretical models to those results.

In this section we do exactly this. We use the real-world measurements reported in published papers [HOT97, NGBS+97] and attempt to reproduce the real-world experiments under simulation. Heidemann *et al.* [HOT97] use two sets of experiments for validation of their model of various implementations of HTTP over TCP. The first measures the overall retrieval time of a small cluster of resources.[4] This workload is tested in two environments (Ethernet and a high-speed Internet connection) using two forms of the HTTP protocol (HTTP/1.0 and HTTP/1.0 with persistent connections). The actual measurements were gathered and used by Heidemann *et al.*

The second set of experiments measures the overall retrieval time of a 42KB Web page with 42 embedded images totaling 125KB. This workload is tested in multiple environments (including Ethernet, high-speed Internet, and modem), but Heidemann *et al.* only considered validating one protocol (HTTP/1.1-style with pipelining). The measurements (along with others) were collected and published by a different group [NGBS+97].

[4] These resources were a single 6651B page with embedded 3883B and 1866B images, corresponding to the Yahoo home page on May 1, 1996

Environment		NCS	[HOT97]	ratio
protocol	network	simulated	measured	m:s
HTTP/1.0	Ethernet	32.0ms	36.8ms (10ms +/12.0ms)	1.15
HTTP/1.0+KA	Ethernet	27.7ms	26.6 (8.8ms +/-1.7)	0.96
HTTP/1.0	Fast-Internet	1729ms	1716 (101ms +/-20.1)	0.99
HTTP/1.0+KA	Fast-Internet	1167ms	1103 (48ms +/-9.5)	0.95

Table 8.1: Validation of simulator response time estimates on the small cluster workload. Examines HTTP/1.0 (serial retrievals on separate connections) and HTTP/1.0+KeepAlive (serial retrievals on a single connection). Measured shows the mean of 100 trials with standard deviation and 95% confidence intervals in parentheses from [HOT97]. Ratio m:s is the ratio of measured time vs. simulated time.

The simulator was configured to be as realistic as possible. Beyond the basic modeling of HTTP over TCP/IP that Heidemann *et al.* describe [HOT97], we incorporated additional costs. We estimated the cost of a establishing a new connection based on transmission of a 40B packet (i.e., just TCP/IP headers) round-trip and a CPU time cost of 1ms.[5] We model the appropriate number of parallel connections. We also modeled a fixed reply header with an estimated size of 200B. Finally, we reduced bandwidth estimates in the modem cases to account for TCP/IP and PPP error correction framing bytes.

In Table 8.1 we show the results of our simulator and compare it to the measurements made by Heidemann *et al.* It examines performance corresponding to HTTP/1.0 (in which serial retrievals on separate connections were made) and HTTP/1.0+KeepAlive (serial retrievals on a single persistent connection, saving the time for subsequent connection establishments). As can be seen from the ratio column, the simulated results are quite close to the real measurements. They also improve upon the adjusted modeled values predicted in Table 7 of [HOT97] in three cases, and equal the fourth (reducing the average error from 11.5% to 6.4%). From these results it appears that we may be underestimating the cost of establishing a new connection.[6]

[5]This is a revised value of CPU time cost – in an earlier version of this work [Dav01b] we mistakenly used a smaller value of .1ms.

[6]A more appropriate estimate of the cost of a new connection may be on the order of multiple ms to account for slower machines and relatively unoptimized code back in 1996. For example, Feldman *et al.* [FCD+99] report the mean total connection setup time in their modem data as 1.3s, which suggests the presence of significant server delays.

Environment			NCS	[NGBS+97]	ratio	ratio
protocol	network	client/server	simulated	measured	m:s	m:a
HTTP/1.1+P	Ethernet	libwww/Apache	164.3ms	490ms	2.98	1.33
HTTP/1.1+P	F-Internet	libwww/Apache	1623ms	2230ms	1.37	1.22
HTTP/1.1+P	Modem	libwww/Apache	53073ms	53400ms	1.01	1.00

Table 8.2: Validation of simulator on large cluster workload. Examines performance for HTTP/1.1+Pipelining on a single connection. Measured shows the mean of 5 trials from [NGBS+97]. m:s ratio is the ratio of measured time to simulated time. m:a ratio is the ratio of measured time to adjusted simulated time.

Table 8.2 compares the results of NCS to the measurements from [NGBS+97]. In this test of pipelining, the simulator performs poorly for Ethernet, but respectably on the Fast-Internet and Modem cases. We suspect this is the result of a number of factors:

1. We model CPU time costs as a delay (which may occur in parallel with transmission of other data) rather than a use of resources (on the server side) during which no transmission likely takes place.

2. We do not know precisely the sizes nor ordering of the images, and instead use equal-sized values.

3. We do not know where within the first HTML retrieval that the embedded image references are placed, and so assume that the client can attempt to retrieve all objects at start.

4. We do not model the pipeline buffering effects described in [NGBS+97].

These effects are more pronounced for the Ethernet case because the scale of measurements is much smaller. The column marked ratio to adjusted shows what the ratio would be if we were to account for factors 1, 3 (assuming a delay of two packets), and 4 (adding a single 50ms delay as described by Nielsen *et al.*). The adjusted values show improvement compared to the adjusted modeled values predicted in Table 8 of [HOT97] in all cases (reducing the average error from 39.3% to 18.3%, although [HOT97] was validating a slightly different dataset from an earlier version of the paper by Nielsen *et al.*).

Environment			NCS	[NGBS+97]	ratio
protocol	network	client/server	simulated	measured	m:s
HTTP/1.0	Ethernet	libwww/Apache	194ms	720ms	3.71
HTTP/1.1	Ethernet	libwww/Apache	358ms	810ms	2.27
HTTP/1.0	Fast-Internet	libwww/Apache	5402ms	4090ms	0.76
HTTP/1.1	Fast-Internet	libwww/Apache	5628ms	6140ms	1.09
HTTP/1.1	Modem	libwww/Apache	62672ms	65600ms	1.05
HTTP/1.0+KA	Modem	Netscape4/Apache	53523ms	58700ms	1.10
HTTP/1.0+KA	Modem	IE4/Apache	53823ms	60600ms	1.13

Table 8.3: Validation of NCS on large cluster workload. Examines performance for HTTP/1.0 (with up to 4 parallel connections), HTTP/1.1 (with a single persistent connection), and HTTP/1.0+KeepAlive (with up to either 4 or 6 parallel connections for Netscape or MSIE respectively). Measured shows the mean of 5 trials, except for the last two rows where it is the mean of 3 trials, from [NGBS+97]. m:s ratio is the ratio of measured time to simulated time.

Since [NGBS+97] include many more measurements than those used by Heidemann *et al.*, we provide additional comparisons in Table 8.3, again using Heidemann *et al.*'s estimates of network RTT and bandwidth. Again, the simulated values are not far from the measured values, except for the case of Ethernet (for the same reasons as described earlier).

Over these datasets, NCS performs similarly to the model used by Heidemann *et al.* and on average is slightly closer to the actual measurements reported. Thus we can conclude that at least in the small, NCS is likely to provide a reasonable estimate of real-world delays attributable to TCP.

8.4.3 Large-scale real-world networking tests

By making reasonable estimates of network and host parameters (shown in Table 8.4), we can replay the large UCB Home-IP trace (as described above in Section 8.3) within the simulator, and compare the resulting response times. Figure 8.5 shows the distribution and Figures 8.6 and 8.7 show the cumulative distributions of response times for the original trace and the results of two simulations (a *deterministic* run and a *stochastic* run that used heavy-tailed distributions of latencies to randomly determine latencies on a per-host and per-connection basis). Likewise, we show results from the

Parameter	Value
packet size	512 bytes
request size	325 bytes
client network latency (one-way)	100 ms
client bandwidth	3420 Bps
server network latency (one-way)	5 ms
server bandwidth	16384 Bps
server per request overhead	30 ms
cost of new connection	22 ms
max conns. from client to host	4
latency distribution for hosts	Pareto
latency distribution for connections	Pareto

Table 8.4: Some of the simulation parameters used for replication of UCB workload.

same experiments in Figures 8.8-8.10 which compare the CDFs and distributions of effective bandwidth. The closeness of all of these graphs helps to validate the simulator. However, the results will not be identical for a number of reasons:

- The simulator does not estimate the same timings that the snooping proxy measured. The 667ms adjustment mentioned above in Section 8.3.2 is only a gross modification, and is not likely to adequately address the timings missing from the variety of client access devices, nor account for the differences in timings that result from significant buffering performed by terminal servers nor modem

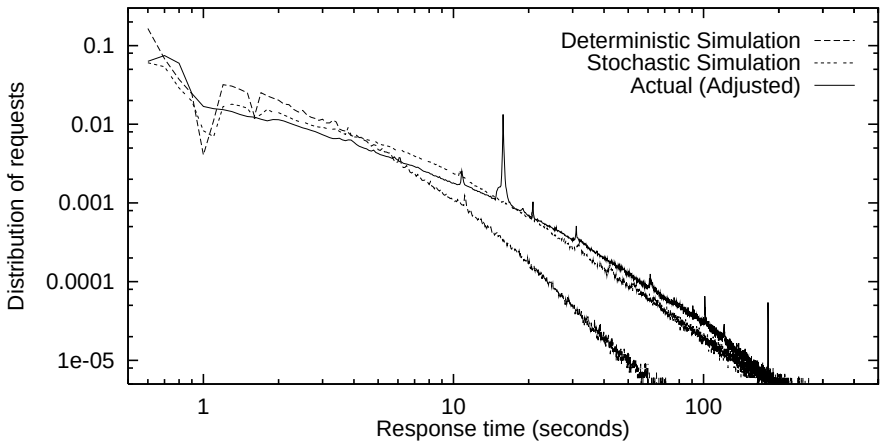


Figure 8.5: Comparison of the distributions of response times from the UCB Home-IP request trace.

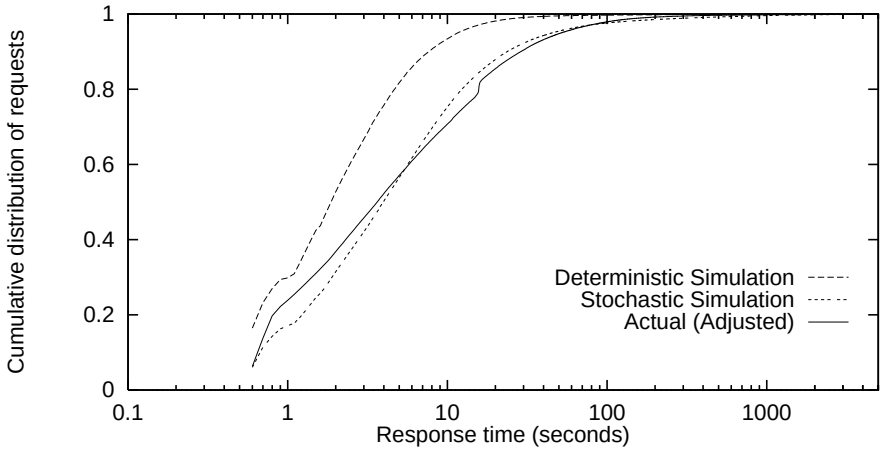


Figure 8.6: Comparison of the CDF of response times from the UCB Home-IP request trace.

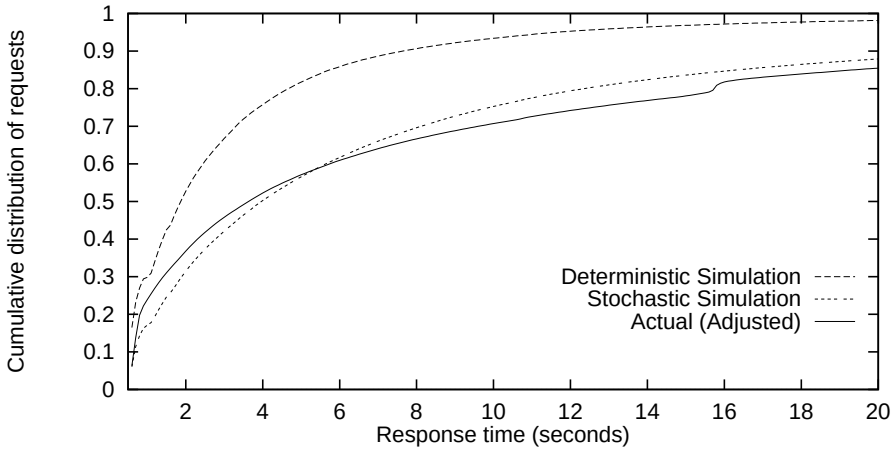


Figure 8.7: Comparison of CDF of response times from the UCB Home-IP request trace (magnification of first 20s).

compression.

- NCS does not model all aspects of the Internet (e.g., packet loss, TCP congestion control) nor does it model all traffic on relevant network links.

- The simulator does not match the original host and network characteristics. Under the deterministic version of the simulator, each node is given the same network and host characteristics, which does not model the variations in connectivity (or

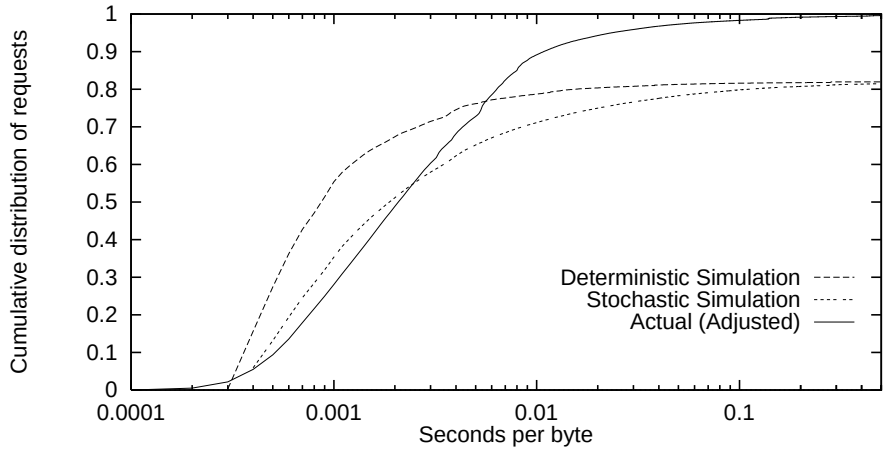


Figure 8.8: Cumulative distribution function of simulated and actual effective bandwidth (throughput) from the UCB Home-IP request trace.

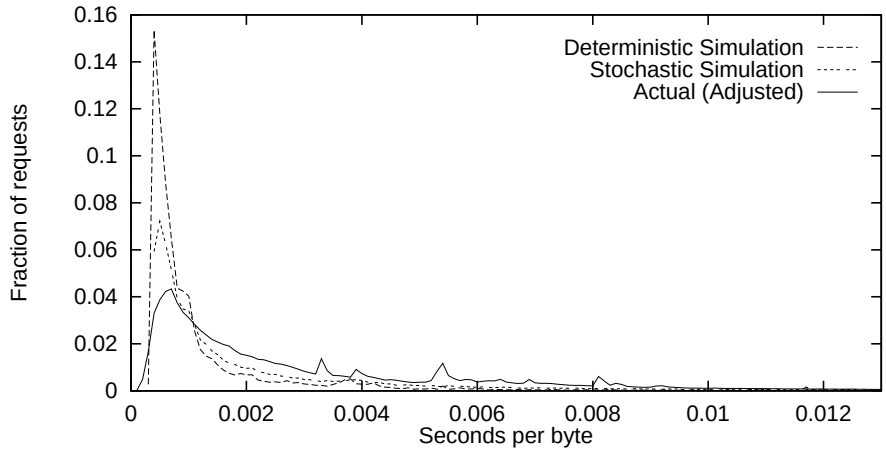


Figure 8.9: Distribution of simulated and actual effective bandwidth (throughput) from the UCB Home-IP request trace.

responsivity of Web hosts). Under the stochastic version, while these characteristics are determined in part by sampling from particular distributions, the activity of a client may actually depend on the connectivity which is being set independently.

- The parameters for the simulations have not been exhaustively explored, suggesting that there may be parameter values that would produce results even closer to the trace results.

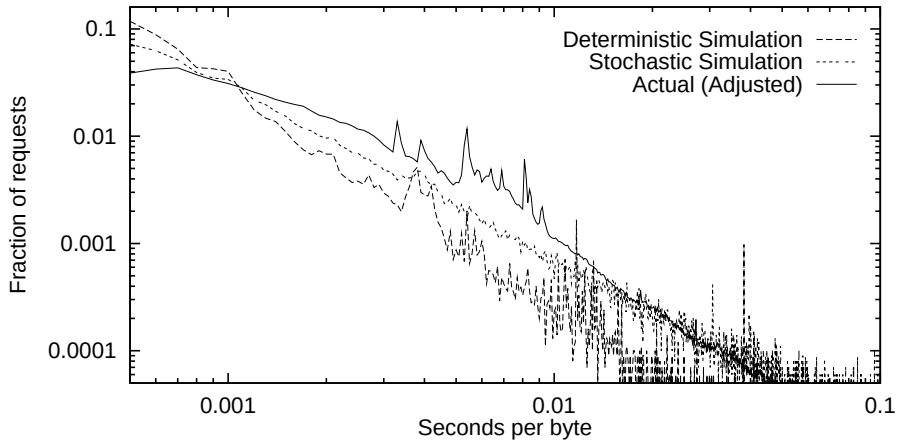


Figure 8.10: Log-log distribution of effective bandwidth (throughput) from the UCB Home-IP request trace.

The graphs also show that the stochastic run of the simulator is appreciably better than the deterministic run. In the stochastic simulation, we used Pareto distributions for upstream latency for clients, and downstream latency for servers. To test the use of the stochastic aspects of the simulator, thirty trials of the stochastic version were run, over traces using 500k requests. In these tests the average of the average response times was 27.75s with a standard deviation of 13.14s. The average median was 4.24s, with a standard deviation of .36s. Thus, as expected with heavy-tailed distributions, we see variation in the means, but are able to find stable results in the median.

As in Figure 8.4, we have also displayed file size vs. response time from a simulated run, using static values for bandwidths and latencies, shown in Figure 8.11. In this scatter-plot, a well-defined lower right edge is visible, corresponding to the best response-times possible with the given bandwidths and latencies. The notch visible at approximately 1024 bytes corresponds to the point at which three packets are needed rather than two to transmit the file (since under typical TCP implementations, the server will have to wait for an acknowledgment from the client before sending the third packet).

The obvious differences between Figures 8.4 and 8.11 suggest that a single static specification of bandwidths and latencies for clients and servers is insufficient to generate

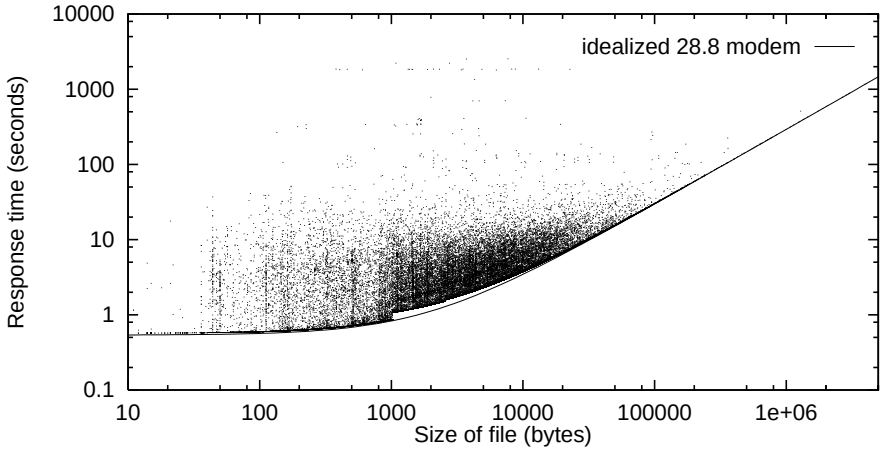


Figure 8.11: Scatter plot of file size vs. simulated (static) response time for first 100k requests from UCB trace.

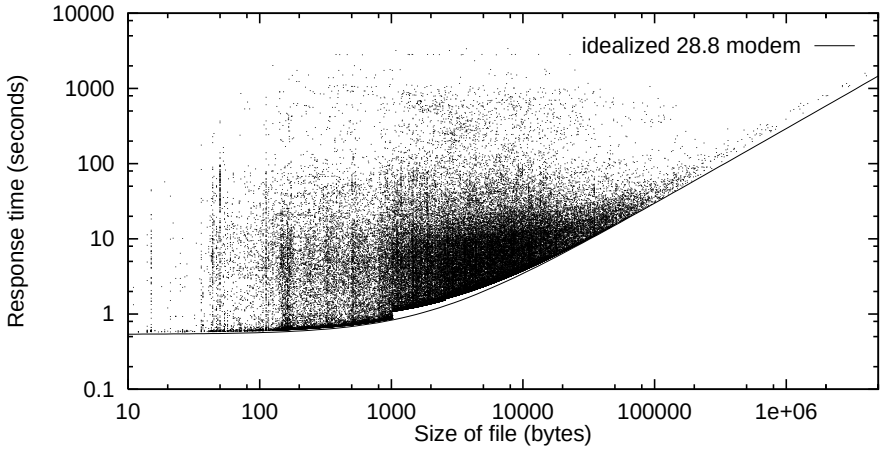


Figure 8.12: Scatter plot of file size vs. simulated (stochastic) response time for first 100k requests from UCB trace.

a sufficiently wide distribution of response times. Figure 8.12, on the other hand, does show more variance in the distribution of response times, qualitatively closer to the actual distributions shown in Figure 8.4, albeit with a defined lower right edge.

Thus Figure 8.12 may be considered "closer" to real-world performance than Figure 8.11, and as a result, demonstrate the trade-off of fidelity vs. complexity. While using heavy-tailed stochastic models may provide increased fidelity (in some sense), it increases the complexity of experiments that use the simulator (because of the need to

Trace	Median	Mean
Adjusted actual	3.6s	18.0s
Deterministic sim.	1.9s	5.0s
Stochastic simul.	4.0s	25.7s

Table 8.5: Basic summary statistics of the response times provided by the actual and simulated trace results.

perform repeated experiments to eliminate the stochasticity as the cause of the effect under consideration). The summary statistics are also closer, as shown in Table 8.5. The median response time from the stochastic run was only 11% larger than the actual median response time, as compared to the deterministic run which was 47% smaller. Means also improved — 42% larger vs. 72% smaller. In any case, we maintain that NCS is capable of generating aggregate response time results that are comparable to real-world results, even in large-scale experiments.

8.4.4 LRU caching tests

To validate the caching code used in the simulator, we used 12 days from the UCB modem workload [Gri97] under two additional sets of code. The first is a simple Perl script that calculates the recurrence rate [Gre93] of the requests and caching statistics given the assumption of an infinite cache. Such a script tells us, among other things, that:

- There are a total of 1921788 unique cacheable objects out of 5953843 served objects. Total bytes delivered is just over 39GB.
- The overall recurrence rate is: 67.23%.
- Taking into account typically uncacheable requests, and requiring that size and last-modification date (when available) must match, the maximum hit rate of an infinite passive cache is 33.45%.

The second code is the University of Wisconsin caching simulator [Cao97].It simulates various sized caches under a few different replacement algorithms. It calculates an object hit rate of 34.49%.

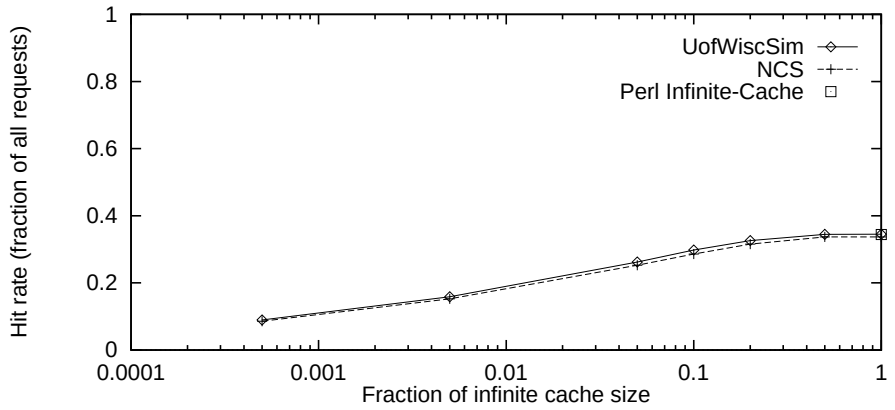


Figure 8.13: Performance of the cache replacement algorithms for NCS and the University of Wisconsin simulator show similar results.

Note that while the UofW simulator uses last-modified times if available, it does not take into consideration the response code nor URLs that are typically considered uncacheable, which the Perl script does. If the Perl script is set to consider all responses as cacheable, it finds an overall infinite cache hit rate of 34.41%.

The UofW simulator also calculates hit rates for various fractions of the infinite cache size. These fixed cache sizes are then simulatable by NCS. In Figure 8.13 the hit rates are reported for various cache sizes for NCS and the UofW simulator. There are slight variations in hit rates, but these can be attributed to small differences in implementation codes.

8.5 Sample Experiments

In this section we provide some sample experiments to demonstrate the utility of the simulator and the various environments that it can model. Unlike the experiments in Section 8.4, we are not intending to replicate the original scenario under which the log might have been captured. Instead, we use the capabilities of the simulator to test new environments and variations to determine the effects of such changes (such as the introduction of a caching proxy, which we describe in Section 8.5.3).

8.5.1 Demonstration of proxy utility

In some situations, proxies can provide performance benefits even without caching any content. Here we will simulate such a situation to demonstrate a sample of what benefits are possible and as an example use of the simulator. While one could calculate these results analytically, the simulator is capable of performing the calculations easily when configured properly and given an appropriate artificial trace on which to operate.

Consider the following scenario: a client is attached to the Internet with a bandwidth of 10000B/s, and a network latency of 1s; a server is likewise attached with the same bandwidth and latency. Thus the overall (uncontested) bandwidth between them is still 10000B/s, but the one-way latency is 2s (thus RTT is 4s). We will assume a segment size of 1000, that request and response headers are negligible in size, and that the server takes 0s to calculate a response. If the client makes a new connection to the server, and requests a file that is 10000B long, it will take (according to the simulator) 16.9s to get the entire file.

However, if a proxy is introduced exactly half-way between the two, with infinite bandwidth, no internal latencies and zero-latency connections to the Internet, the response time will improve. In this situation, the bandwidth is still 10000B/s between client and proxy, and between proxy and server, but the delay between client and proxy, and between proxy and server, is now just 1s (RTT of 2s). Even though the RTT from client to server is still conceptually 4s, TCP's slow start is driven by the RTT in the local connection, and effectively forms a pipeline such that activity (or delays) are occurring simultaneously on both sides of the proxy, resulting in a response time of just 13s. For naive slow-start (instead of the more sophisticated one used here) the response times would be 20.6s vs. 14.7s. In either case, the improvement is a significant fraction of the cost of the retrieval.

If we use a more realistic environment in which both client and server are connected via a DSL line (1Mb/s, 10ms latency), we get 232ms response time instead of 200ms with a proxy in the middle. In reality, the introduction of a proxy does introduce some latency for the computation time in the proxy (plus disk, if it is a proxy cache). This,

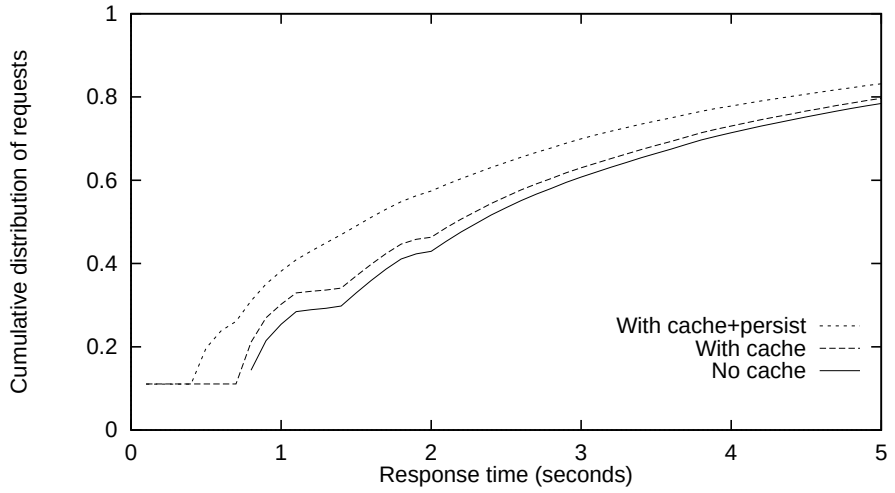


Figure 8.14: Comparison of client caching on the CDF of response times from the UCB Home-IP request trace.

too, could be modeled by the simulator for greater accuracy. In certain environments, however, if this latency is small enough, the overall effect can be positive, even without caching. In any case, the simulator provides an environment in which different scenarios can be tested.

8.5.2 Additional client caching

NCS provides the ability to model Web caching at clients and/or proxies. Here we examine the effect of adding a 1MB cache (in addition to any cache already present) at the client. This new space is called an extension cache [FJCL99]. We further assume that the content of this cache will expire in at most one day from the last request for that object. This simulation uses no persistent connections, nor pipelining, as in the original UCB Home-IP trace.[7]

To examine the UCB Home-IP trace, we ran variations of the statically-configured simulator, on the first 12 days of requests. We used the versions of the traces in which sizes did not include headers, and added a fixed header according to the average response header size from the overall trace (158 bytes). The servers are assumed to

[7]Actually, [GB97] says persistent connections were present, but very uncommon.

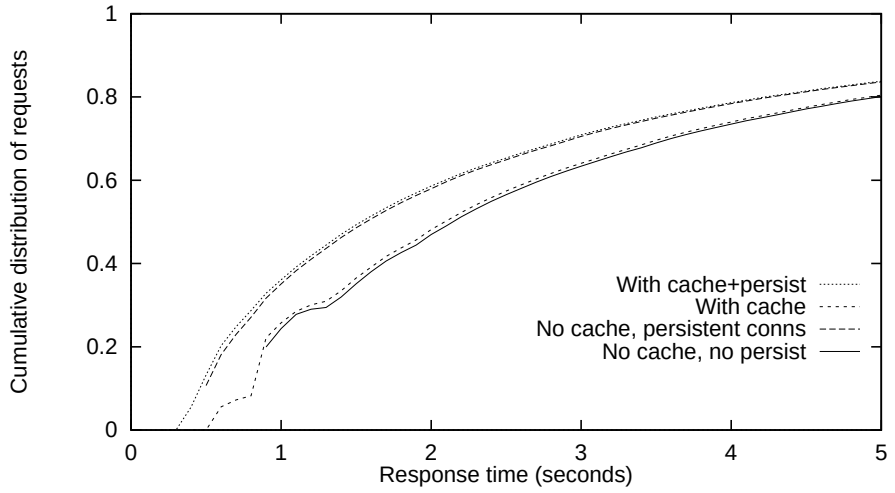


Figure 8.15: Comparison of proxy caching on the CDF of response times from the UCB Home-IP request trace.

be connected with T1 (1.544Mbit) bandwidth, 50ms one-way latencies, and 30ms per-request overhead. Clients are connected at an effective bandwidth of 27.4kbps with 100ms one-way latencies.

From this test, it can be seen that a small amount of additional client caching can improve performance. Just over 11% of requests are served from the client extension caches. The mean response time drops from close to 5.5s to 5.1s, and efficiency improves 17%. Adding persistent connections improves performance even further, dropping the mean to 4.6s (and improving efficiency over uncached by 26%). The median results are visible in Figure 8.14, in which the uncached median stands at 2.4s, caching improves that to 2.2s, and persistent connections brings the median down to 1.6s.

8.5.3 The addition of a caching proxy

In lieu of client caching, we can use NCS to examine the effect of caching at the proxy level. In these experiments we assume the use of a single proxy cache that provides all content to the clients in the trace. While the proxy will cache objects for all clients (assuming a relatively small, fixed 512MB cache size), the clients will not use an extension cache.

In this test, the clients and servers are configured identically to those in Section 8.5.2, with the exception of no caching at the client. Here we add instead a proxy, with 45Mbit connectivity upstream and downstream. The modeled proxy also adds 2ms latency in both directions, and a per-request CPU overhead of 10ms.

In this system, a scenario with a non-caching proxy without persistent connections achieves a mean client response time of 5.10s. A caching proxy saves little more than 1.4% in mean response time, but does provide a reasonable hit rate (19.7%) and reduced network bandwidth usage (a savings of 9%). These minor improvements are expected in such a scenario, as the predominant bottleneck (the poor client connectivity) has not been addressed in this scenario (either by caching at the client, or by persistent connections between client and proxy).

Support for persistent connections provides a good boost in response time. A non-caching proxy with persistent connections shaves 562ms from the average response time, and .6s off of the median (a savings of 10.7% and 27.3%, respectively). Add caching, and the mean response time drops to 4.65s (11.5% response-time savings over the non-caching, non-persistent scenario). See Figure 8.15 to compare all four configurations.

8.5.4 Modeling DNS effects

NCS offers a simple mechanism to model DNS effects. Each node can provide a fixed cache of DNS lookups for a parameterized amount of time. While the cost of DNS lookups is currently fixed (or possibly selected randomly from some distribution), a more complex simulation (not currently implemented) could be possible, utilizing organization-wide DNS caches to supplement the client cache.

8.5.4.1 Client caching of DNS entries

Here we model characteristics of the Netscape 4 browser (10 entries in the DNS cache, expiration after 900s), and assume a fixed cost of 600ms to retrieve a new entry for the UCB Home-IP trace. We are not simulating any additional Web caching so that only the caching of DNS responses contributes to changes in performance. Otherwise, this simulation is identical to the caching client scenario in Section 8.5.2. We can show

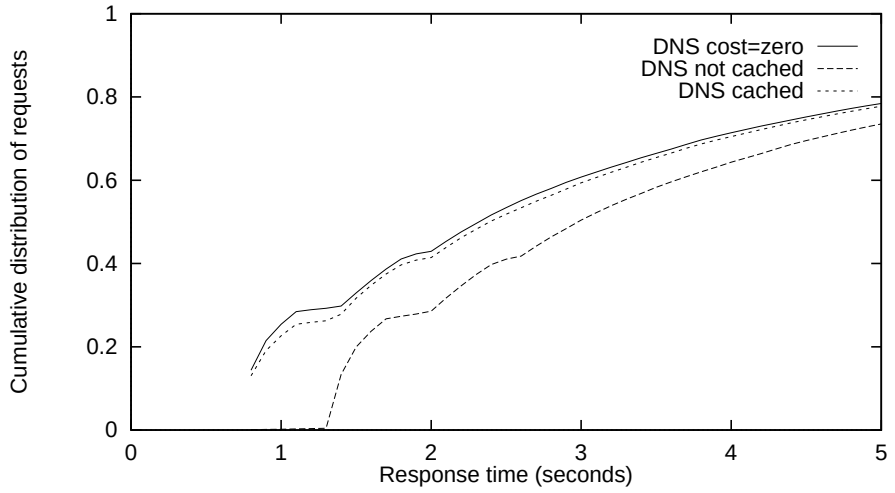


Figure 8.16: Comparison of DNS caching at the client on the CDF of response times from the UCB Home-IP request trace.

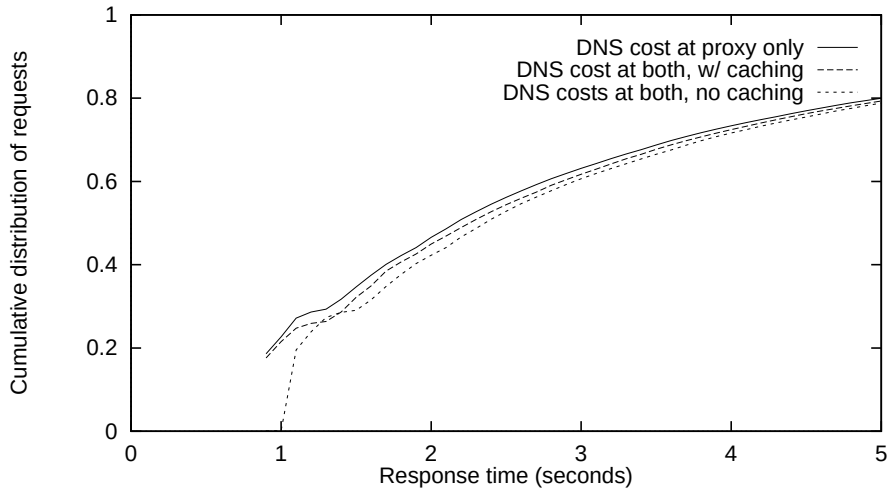


Figure 8.17: Effect of DNS caching at a proxy on the CDF of response times from the UCB Home-IP request trace.

improvements in all measures: mean per-request response times decreased by 11% (from 6.25s to 5.54s); median per-request response times decreased by 20% (from 3.0s to 2.4s); and mean per-request throughput also improved by 19%. See Figure 8.16.

8.5.4.2 Proxy caching of DNS entries

DNS caching can also be performed by shared proxies, and is in fact a benefit of using a (non-transparent) proxy. When a client is configured to use a parent proxy, the only DNS lookup that is necessary is the IP address of the proxy, since all requests will be sent to it. The proxy will have to do the DNS lookup, and can operate a larger and better-managed DNS cache than most browsers. In the case in which requests are "transparently" intercepted by a proxy, both the client and the proxy have to ask for DNS resolution. In Figure 8.17, we again assume modem-based clients using the UCB trace with a cost of 600ms for DNS results from clients, and a cost of 200ms for DNS results for the proxy, and show a measurable improvement in typical response times when using an explicit proxy.

8.5.5 Incorporating prefetching

NCS incorporates the ability to perform prefetching — that is, to speculatively issue retrieval requests in advance of an actual request in the trace. Prefetching can be performed by clients or by proxies. The model to decide what to prefetch can be built by the client, proxy, or server. In the case of a server or proxy model, the predictions can be included as hints that are sent to the downstream prefetcher that can use them or integrate them with its own predictions. However, the model built is reflective of the requests seen by the system building the model — i.e., a server model sending hints will be built from the requests (prefetching or demand) generated by its clients.

To demonstrate this feature, we will use the SSDC Web server trace and prediction code introduced in Chapter 4, and configure the simulated network topology to consist of non-caching DSL clients (128kbps bandwidth, 80ms round trip delay), a prefetching proxy with 5MB of cache placed at the ISP to which the Web server is connected via 33.6kbps modem. The proxy cache is additionally configured such that it will only retain objects for an hour at most. The prediction module makes only one prediction using the largest Markov model with sufficient support after each request. The maximum order used was three; the minimum order was one. Ten requests of an object were

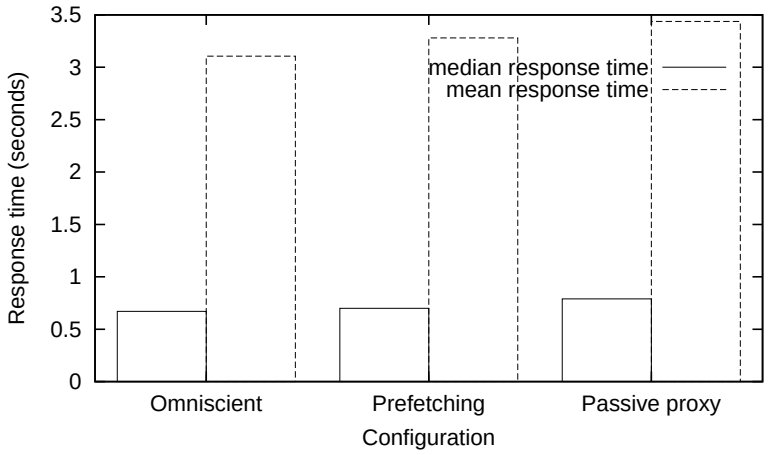


Figure 8.18: Effect of prefetching at a proxy on the response times from the SSDC Web server request trace.

considered sufficient to be used for prediction.

If the proxy is configured to passively cache only, the simulation estimates a mean end-user response time of 3.437s and median of .790s. It had an object hit rate of 51%, and used only 72.2% of the demand bandwidth when communicating with the origin server. Alternatively, the prefetching version described above reduced the median client delay to .70s and the mean to 3.280s, at the cost of another 4.7% in demand bandwidth.

Given a suitably augmented trace file (that is, one with the next future request listed alongside the current request), we can simulate a single-step omniscient predictor. Thus, for comparison, if we had an omniscient predictor that would give the next request made by a client, the same prefetching proxy would have achieved a median client delay of .670s and mean of 3.106s, and is shown in Figure 8.18. Thus, the prefetching version came within 4.5% median response time and 6% mean response time of what was possible.

8.6 Discussion

8.6.1 Related work

Many researchers use simulation to estimate Web caching performance as mentioned in Chapter 7. Often they measure object and byte hit rates and ignore response times. Response time improvement, however, is a common justification for the use of Web caching, and is arguably the initial *raison d'être* for content delivery networks such as Akamai [Aka02]. Few, however, report on the verification and validation efforts they performed, if any. Typically the most provided is a statement that hit rates (or other measures) are similar to other published results (e.g., [KLM97]). Here we describe simulators that either estimate response time or provide some measure of validation.

PROXIM [CDF+98, FCD+99] is a caching and network effects simulator developed by researchers at AT&T Labs. Like NCS, PROXIM accounts for TCP slow start and does not consider packet losses and their resulting effects. It does, however, consider canceled requests and their effect on bandwidth usage, unlike NCS or the other simulators mentioned here. PROXIM does not include prefetching and only one of their papers [FCD+99] provides for even minimal validation — showing similar curves for the simulated response time distribution as the original response time distribution, and comparing median total response times. The traces used by Feldmann *et al.* were captured by a snooping proxy, much like those described in Section 8.3, and have likely been cleaned and adjusted similarly (although this isn't described explicitly). One unusual aspect of these experiments is that the traces provide extraordinary timing details (including timestamps of TCP as well as HTTP events), allowing PROXIM to use RTT estimates from measurements of SYN to SYN-ACK and HTTP request and response timestamps on a per-connection basis.

Like NCS, the simulator described in Fan *et al.* [FJCL99] uses the timing information in a portion (12 days) of the UCB trace [Gri97] to estimate response times seen by each modem client. However, it also uses a much simpler model of response time that

- estimates client-seen response time to be the sum of 1) time between seeing the request and the first byte of the response, and 2) the time to transfer the response

over the modem link,

- appears to ignore connection setup costs,
- appears to ignore TCP slow start effects, and
- groups overlapping responses together.

However, Fan *et al.* claim that by grouping overlapping responses together they are able to measure the time spent by the user waiting for the whole document. This simulator does incorporate some prefetching — it uses one kind of prediction by partial match (PPM) for proxy initiated prefetching (in which the only objects prefetchable are those cached at the proxy, resulting in no new external bandwidth usage).

Kroeger *et al.* [KLM97] examine the limits of latency reduction from caching and prefetching. They use proxy traces from a large company and find that 23% of the latency experienced by a user is internal, 20% is external but cannot be fixed by caching or prefetching, and the remaining 57% correspond to external latencies that can be improved. Prediction is performed by an optimal predictor, with some limitations on when prediction is permitted. The latencies used are not calculated *per se*, but are extracted directly from the logs.

The *ns* simulator [UCB01] is likely the best-known networking simulator, but is not typically used for caching performance measurements, possibly because of slow simulation speeds. It uses detailed models of networking protocols to calculate performance metrics (see [BEF+00] for an overview). This simulator has been validated by wide-spread user acceptance and extensive verification tests [Flo99].

8.6.2 Future directions

Much of NCS is simplistic, and could be improved (both in fidelity and in simulation efficiency) with additional time and effort. Suggested areas of improvement include:

- Range-based requests and caching. HTTP/1.1 traces often include partial content responses, implying that range-requests were used.

- Better models for packet loss. The simulator does include a simple uniform probabilistic model of loss, but one based on congestion, along with recovery mechanisms, would be more appropriate.

- A better model for consistency. The current simulator does not sufficiently model changes at the source, and as a result it does not generate simulated GET If-Modified-Since requests.

- Model terminated requests. Some traces provide information that specifies if a response was terminated early (as a result of user cancellation via the stop button, or by clicking ahead before the document fully loaded). Similarly, it may be important that prefetch requests using GET be terminable when demand requests are present.

- Prefetching other than most-likely next request(s). Large objects need extra time to be retrieved, and so prefetching further in advance (i.e., more than one step ahead) may be needed.

Such improvements would increase simulation fidelity. Their impact on simulation results, however, is unknown.

8.6.3 Summary

In this chapter we have motivated the development of NCS and described its features and capabilities. We additionally provided a number of sample experiments showing the simulator's utility in a variety of contexts.

While we expect that true believability will come with hands-on experience, we have also used this chapter to argue for the validity and believability of NCS for simulating retrieval latencies. In the process, we have provided a case study for network simulator validation by comparison to real-world data and discussed some of the preparation required for using the UCB Home-IP HTTP Trace.

Chapter 9

Multi-Source Prefetching

9.1 Introduction

In this thesis we have examined Web predictions from two general sources — prediction from content, and prediction from history. These can be broken down into many independent sources of prediction information:

- Links from the current Web page.
- Links from other browser sources, such as bookmarks.
- Links from content in user's context, such as Usenet news and e-mail.
- URLs based on the user's historical access patterns.
- URLs based on the average user's access patterns as seen by the server.
- URLs based on a workgroup's access patterns as seen by a proxy.

Some of these we have explored in depth in previous chapters. However, each of these sources of predictions have limitations. Each provides a different viewpoint and each has relevant knowledge at different times.

This chapter will explore the results of combining predictions from multiple sources. In particular, we will examine the performance improvements possible when:

- combining content-based prediction and history-based prediction.
- combining multiple history-based predictions from various sources.

Each combination is capable of providing performance improvements in some situations.

While not common, other researchers have proposed the idea of combining information from various sources for prefetching. Hine *et al.* [HWMS98] builds models of resources likely to be accessed in the same session and combines that knowledge with a categorization of the client's interest in resources at the site. In contrast, our approach will build models of user behavior from the perspective of the client as well as the server, but does not attempt to classify the user interest.

Jiang and Kleinrock [JK97, JK98], on the other hand, explicitly suggest the use of both client and server-based predictions. In their approach, if the client has visited this page less than five times and the server has a prediction, then use the server's prediction. Otherwise, use the client's prediction. Unfortunately, the experimental tests are too limited to determine realistic performance measures.

9.2 Combining Multiple Predictions

Prediction is often quite similar to classification, and combining multiple classifiers has a history of success in machine learning tasks. For example, boosting is a method for finding accurate classifiers by combining many weaker classifiers [FS96, SFBL97]. The weighted majority [LW94b] and Winnow [Lit88] algorithms can be seen as methods for combining the classifications of many simplistic "experts". However, combining ranked predictions in machine learning is not so common.

In a sense, the PPM-style predictors introduced in Chapter 4 already combine multiple predictions. PPM takes the predictions of multiple contexts (i.e., the predictions of Markov models of different orders) and combines them to generate an overall probability distribution of the possible next actions.

Our approach will be similar. After every action in sequence, each prediction source provides a (possibly empty) list of predictions, each associated with a weight. We will then merge those predictions using a fixed weight, as we did in Section 4.5.3 with performance similar to PPM. More complex merging functions are certainly possible, and may offer improved performance (as they have in other areas such as search engines [CSS99, DKNS01]).

One issue relevant (but not specific) to our application is the concern for trust of predictions. If we were to implement a prefetching system that incorporated the use of server hints as a source of predictions, we would need a trust model for those predictions. Since the predictions are likely to influence the prefetching activity of our system, unscrupulous server operators could send hints with high weights to cause, for example, increased traffic that might be reported to advertisers, or if given sufficient traffic, to overload some unsuspecting target server. While we do not explore this issue further, we note that some mechanism will be needed to model the trustworthiness of the servers sending such hints.

9.3 Combining Content and History

In this section we consider the task of combining content-based and history-based predictions. We will use the same log as described in Chapter 6, apply the history-based mechanisms from Chapter 4, and compare the predictive performance of each separately and in combination. Unfortunately, because we will be using content-based predictions, we will be unable to use NCS to simulate response times, as the content-based predictions include objects that were never requested on the original trace, and thus we do not have sizes or retrieval times for those objects. We will instead focus on predictive performance by incorporating the content-based predictions into a modified version of the history-based prediction codes used in Chapter 4.

Recall the performance of the content-based predictor from Chapter 6. For comparison, we'll use the top-1 and top-5 similarity-based orderings using up to 20 terms around the anchor, and we'll measure the predictive performance over all points (not just the possible ones). In this environment, the content-based predictor gets 5.3% and 15.8% correct for top-1 and top-5, respectively. A PPM-based predictor (using bigrams and trigrams) that makes a prediction whenever the probability of success is estimated to be .1 or above gets 6.8% and 8.4%, respectively. However, if we combine the two predictors — by using the content-based predictor whenever the history-based predictor did not use all of its allowed predictions, we get much better performance, as shown in Figure 9.1. For this dataset, at least, the two predictors make predictions in mostly

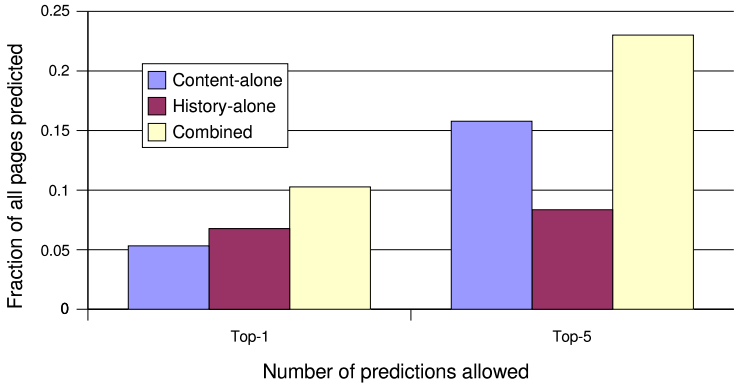


Figure 9.1: Predictive accuracy on the full-content Rutgers trace using prediction from history, from content, and their combination.

different scenarios, resulting in performance that is close (between 85% and 95%) to the sum of the performance of the two predictors alone.

9.4 Combining History-Based Predictions from Multiple Sources

In this section we examine the performance potential for combining multiple history-based predictions. We consider three logs — the SSDC Web server log, a Web server log from NASA, and the UCB Web proxy log.

For each trace we will present the results of an arbitrary combination of simulation parameters. For the Web server logs we will also example a wider selection of possible prediction parameter settings. In particular, we will examine the effect of the following on response times:

- the number of predictions permitted (from 1 to 5),
- the maximum n-gram size used (from 2 to 4),
- the minimum support for prediction (from 1 to 20), and
- the minimum confidence for prediction (from .01 to .2).

9.4.1 Web server log: SSDC

9.4.1.1 Initial experiments

As described in Section 4.4.3, the SSDC Web server log records the requests made by users of a modem-connected small software development company's Web server. We will use NCS to simulate a sample of potential prefetching configurations and evaluate the client-perceived response times that result from those configurations. In our simulations we will assume the use of a modem with a slightly less than 32kbps bandwidth and 100ms one-way latency. A mix of clients connections are likely (i.e., both modem and broadband users) so the simulated client is given approximately 120kbps bandwidth and 40ms one-way latency. Thus clearly the origin server bandwidth is the bottleneck, as it was for the system in reality.

When caching or prefetching at the client is used, the client will have an additional 2MB cache. Prediction models at either the client or server will be built with the same parameters: n-grams must be between 1 and 4 requests long; a scenario must be seen at least 10 times before a prediction is made; and the probability in a prediction must be at least .1 to be used. The latter two provide thresholds to require some confidence in the prediction before spending the resources to prefetch.

Table 9.1 summarizes the performance of various configurations. It provides the mean and median response time for each approach, as well as the bandwidth required, and the fraction of requests served by prefetched content, plus the fraction of requests that prefetched content that was unusable.

From the same experiments, Figure 9.2 depicts the fraction of requests served with

	Response time		**Fraction of**	**% reqs pref'd**	**% reqs pref'd**
Approach	**mean**	**median**	**original bw**	**successfully**	**unsuccessfully**
Prefetch perfectly	4.23s	0.95s	97.5%	30.88%	1.26%
Prefetch both	4.64s	1.17s	133.7%	34.06%	59.94%
Prefetch server	4.62s	1.18s	132.3%	33.89%	58.98%
Prefetch client	5.16s	1.68s	94.7%	0.08%	0.11%
No prefetching	5.16s	1.68s	94.7%	0%	0%
No cache	5.91s	1.91s	100%	0%	0%

Table 9.1: Summary statistics for prefetching performance when one prediction is made with the SSDC log.

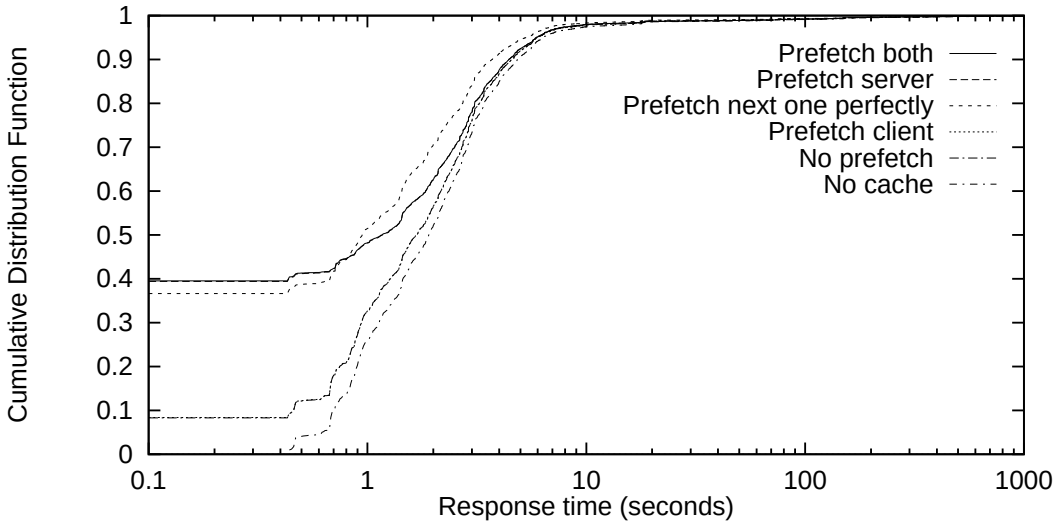


Figure 9.2: Cumulative distribution of client-side response times when only one prediction is used with the SSDC log.

at most a given response time, by showing the cumulative distribution of client-side response times. The bottom-most line corresponds to the performance seen without any additional caching or prefetching. As a result, it has the worst of all performances with a mean and median response times of 5.91s and 1.91s (recorded in Table 9.1 while using the originally recorded bandwidth (defined for comparison as 100%). The next two lines overlap almost entirely — the configuration in which an additional cache at the client is used, and when those clients can prefetch their single best prediction. This is because the clients are rarely able to make predictions (because of the minimum thresholds established). With an object hit rate of over 8%, the mean and median response times improve, to 5.16s and 1.68s, and bandwidth needs are reduced by 5%.

The server, however, has the benefit of seeing all requests and thus having lots of data to provide for a better model. When the server provides the predictions for the client to prefetch, it achieves client object hit rates of 39.4% and mean/median response times of 4.62s/1.18s. Even with that performance, the per-client model can still provide some benefit, as when the two are combined, hit rates increase slightly to 39.6%, reducing the median slightly to 1.17s, but increasing the mean response time to 4.64s. Bandwidth usage in both cases increases by close to a third.

At the top we show performance provided by the perfect predictor — that is, if after

Approach	Response time mean	Response time median	Fraction of original bw	% reqs pref'd successfully	% reqs pref'd unsuccessfully
Prefetch perfectly	4.23s	0.95s	97.5%	30.88%	1.26%
Prefetch both	4.41s	0.75s	179.1%	41.94%	122.8%
Prefetch server	4.55s	0.94s	170.4%	39.42%	106.8%
Prefetch client	5.16s	1.68s	94.8%	0.10%	0.21%
No prefetching	5.16s	1.68s	94.7%	0%	0%
No cache	5.91s	1.91s	100%	0%	0%

Table 9.2: Summary statistics for prefetching performance when five predictions are made with the SSDC log.

each request the client knew the next request needed by the user, it could attempt to prefetch at that time. This scenario provides a median response time of just .95s and reduces mean response time to 4.23s while using only 97.5% of original bandwidth. The perfect predictor has bandwidth usage less than 100% because of the use of caching, but has greater usage than the caching-only configuration because it has some requests that are unsuccessful (not retrieved in time to be useful) and may additionally push some otherwise useful objects out of cache.

If we allow up to five predictions, we can improve performance further. Table 9.2 provides statistics over the same set of configurations (and depicted again as a CDF in Figure 9.3), but with the use of five predictions for prefetching. Five choices

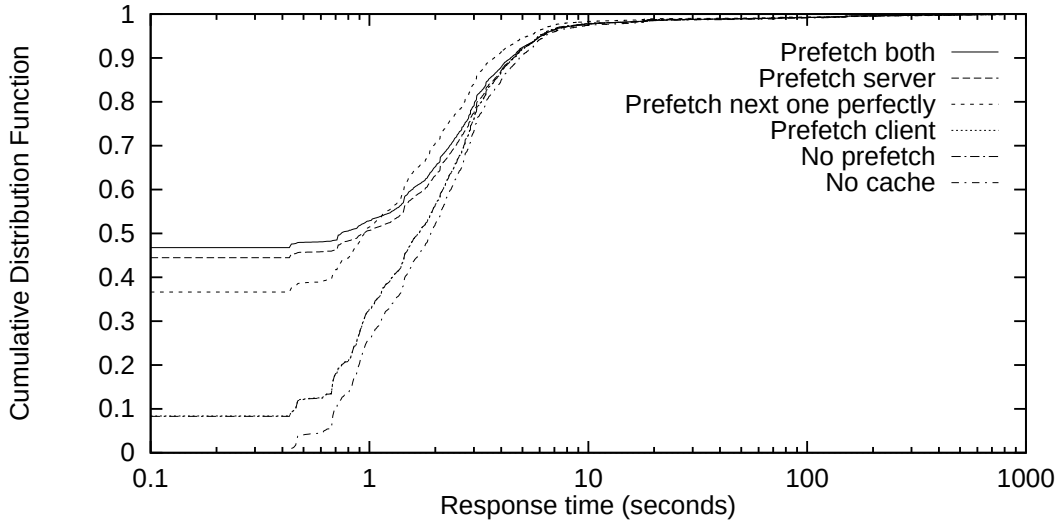


Figure 9.3: Cumulative distribution of client-side response time when five predictions are used with the SSDC log.

does not help client prediction in this dataset, but the server's model can increase hit rates further, to 44.5% and decrease mean and median response times to 4.55 and .94s respectively. The cost for this increased performance is increased bandwidth usage — 70% more than the original. When combined with the client predictor, hit rates reach 46.8% and client response times drop to a mean of 4.41s and median of .75s, at a cost of 75% more than the original bandwidth.

At first glance, such performance improvements seem impossible, since hit rates and median response times are considerably better than the perfect predictor. However, there is a rational explanation. Although the perfect predictor knows exactly what the next request will be, that knowledge is only available when the current request is made. In many cases, it is not possible to retrieve the object before it is requested. The five guess configurations, on the other hand, prefetch many more objects, but do so further in advance, allowing the object to arrive perhaps multiple requests before it is needed and thus providing even higher hit rates and shorter average response times.

9.4.1.2 Parameter studies

We now systematically examine how performance changes as the prediction parameters are modified. We first investigate the relationship between median response times to that of mean response times. We show the asymmetry of the two metrics here to provide a reference point for comparison as we will focus on median measurements in future graphs.

Figure 9.4 plots the median versus mean response times for the various configurations of our prefetching system that uses server hints for the SSDC trace. One point represents the caching-only configuration; the other points signify the other tested configurations. Mean response times range from slightly over 4 seconds to just over 18 seconds. Median response times vary from under .5 seconds to close to 2.1 seconds. This figure shows that while many configurations can significantly improve median response times (since there are many points well under the caching-only configuration), none significantly improve mean response times (since better points are not much below the caching-only configuration), although many significantly increase mean response

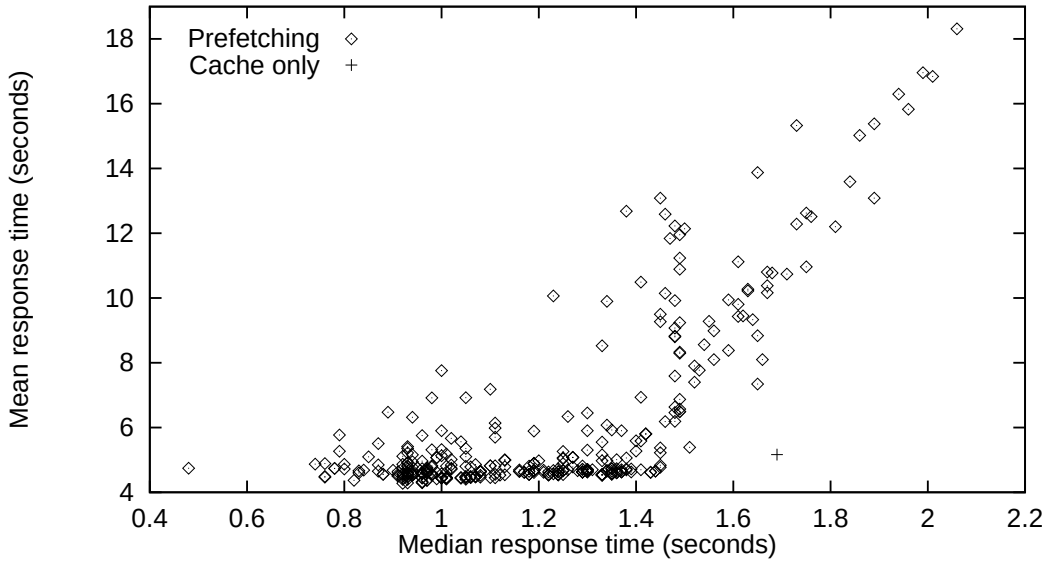


Figure 9.4: Scatterplot comparing median versus mean response times for various configurations using the SSDC trace.

times. It also suggests that the very large or slow to retrieve objects are less likely to be prefetchable (since they are a significant contributor to the mean, which does not improve much).

Our next concern is for the cost of prefetching as compared to its benefits, and how parameters might be used to optimize the benefits gained. Therefore, in Figure 9.5, we plot the median response time versus bandwidth usage. We also distinguish among the number of predictions permitted (labelled Prefetching 1, Prefetching 2, etc. in the figure). All of the prefetching approaches use more bandwidth than a caching-only configuration, but it is apparent that many prefetching approaches are capable of reducing the median response time while using no more than twice the bandwidth of the original trace. The figure also demonstrates that many configurations are able to cut the caching-only median response time in half (or better).

We can also make some observations based on the number of predictions permitted. Using just a single best prediction is typically insufficient for large improvements in response times. However, configurations using two predictions are sufficient to provide a range of points along the lower left edge, demonstrating various tradeoffs between bandwidth and response time. The use of three or four predictions result in curves

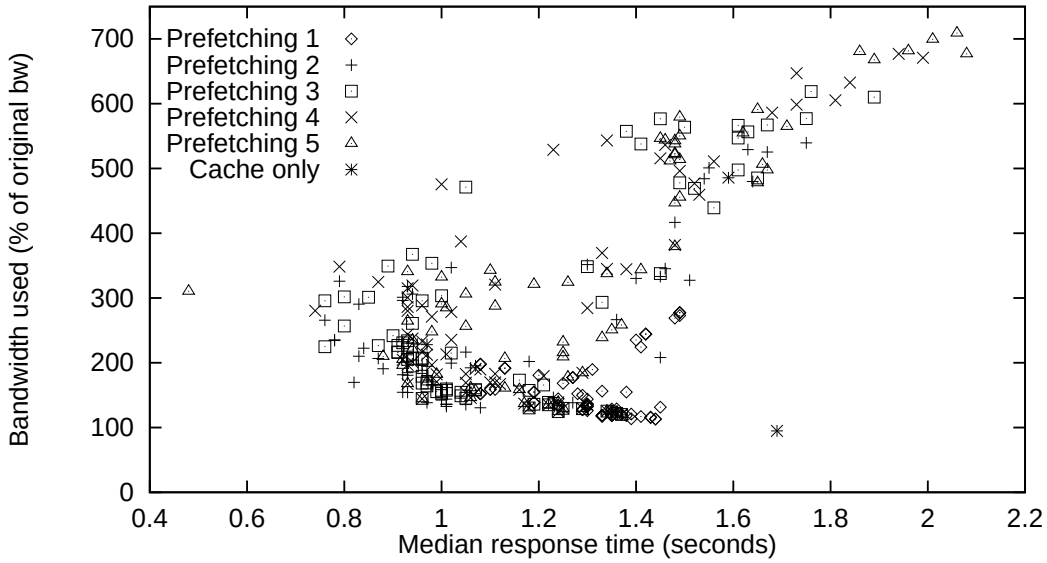


Figure 9.5: Scatterplot comparing median response time versus bandwidth consumed using the SSDC trace, highlighting the number of predictions permitted.

slightly further back. Five predictions are likewise further back, but are also more spread out through the field, achieving both the best and worst median response times.

Finally, four points deserve highlighting. First is the caching only configuration, using 94.7% bandwidth to provide a 1.69 seconds median response time. Second, a five prediction point provides an outlier with much better median response time than any others, at 0.48 seconds. That configuration allowed n-grams up to length four (the maximum tested) and required a minimum of two repetitions and minimum probability of 0.1 (both in the middle of ranges tested). Third, while not one of the smallest medians, a two prediction point along the edge provides an excellent tradeoff of 170% bandwidth usage to achieve a median response time of 0.82 seconds. Later, in Figure 9.8 we will use this configuration with the label "Second best server". It used a fixed n-gram length of two, required ten repetitions and a minimum confidence of 0.05 to make a prediction. Fourth, in the top-right corner lies the point with the largest median response time (our "Worst server"). It also used a bigram length of two and minimum confidence of 0.05, but required only 1 repetition. Thus, many predictions satisfied the requirements and were used, resulting in less successful prefetching, leading to congestion and increasing delays.

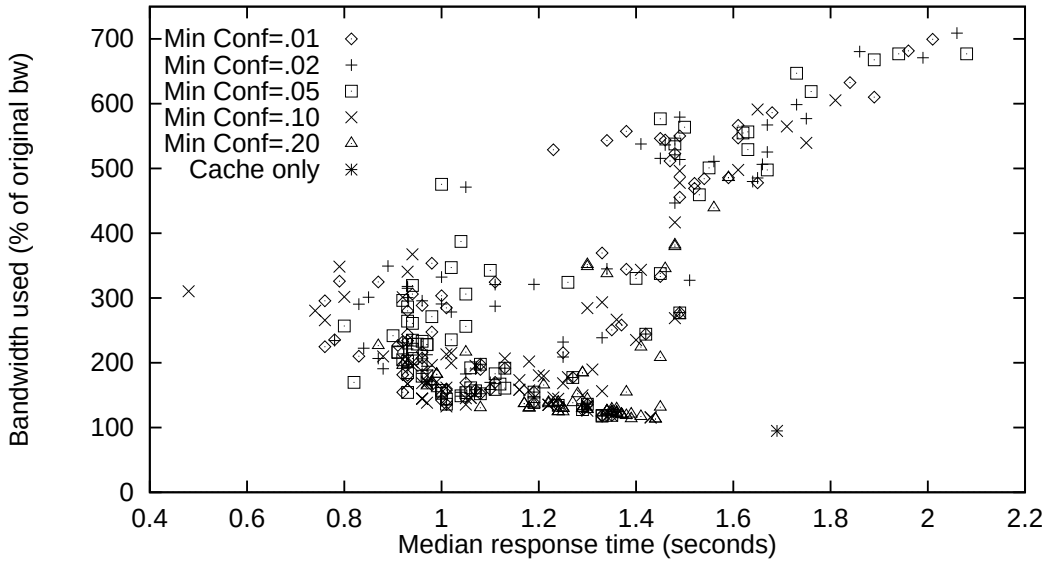


Figure 9.6: Scatterplot comparing median response time versus bandwidth consumed using the SSDC trace, highlighting the minimum confidence required.

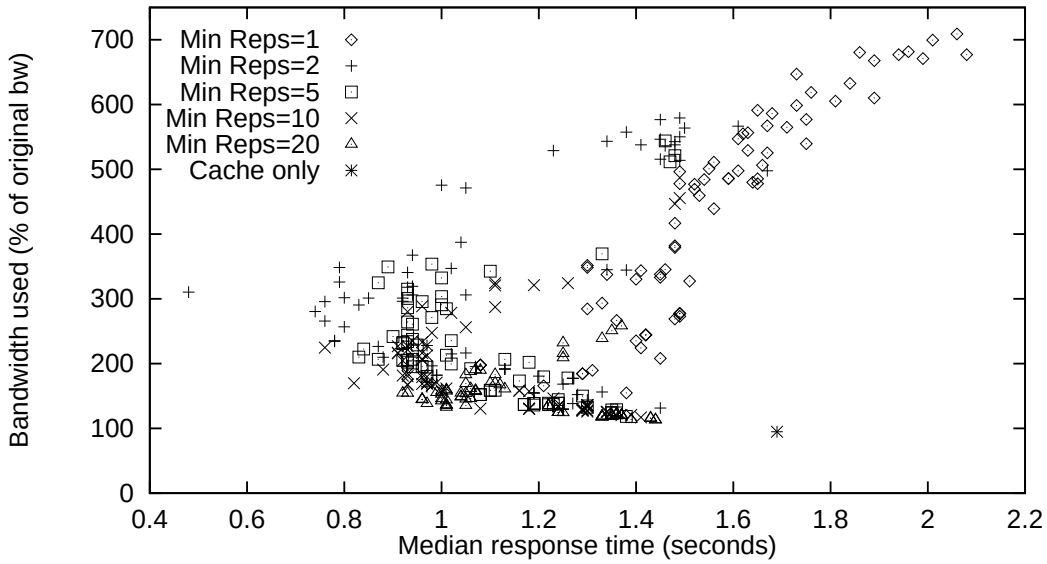


Figure 9.7: Scatterplot comparing median response time versus bandwidth consumed using the SSDC trace, highlighting the minimum repetitions required.

In Figures 9.6 and 9.7 we re-examine the median response time and bandwidth used, but in the context of the parameter settings for the minimum confidence and repetitions required, respectively. While the various confidence values are scattered throughout Figure 9.6, there are definite clusterings in Figure 9.7. The latter shows that the minimum repetition setting should certainly be larger, rather than smaller. The

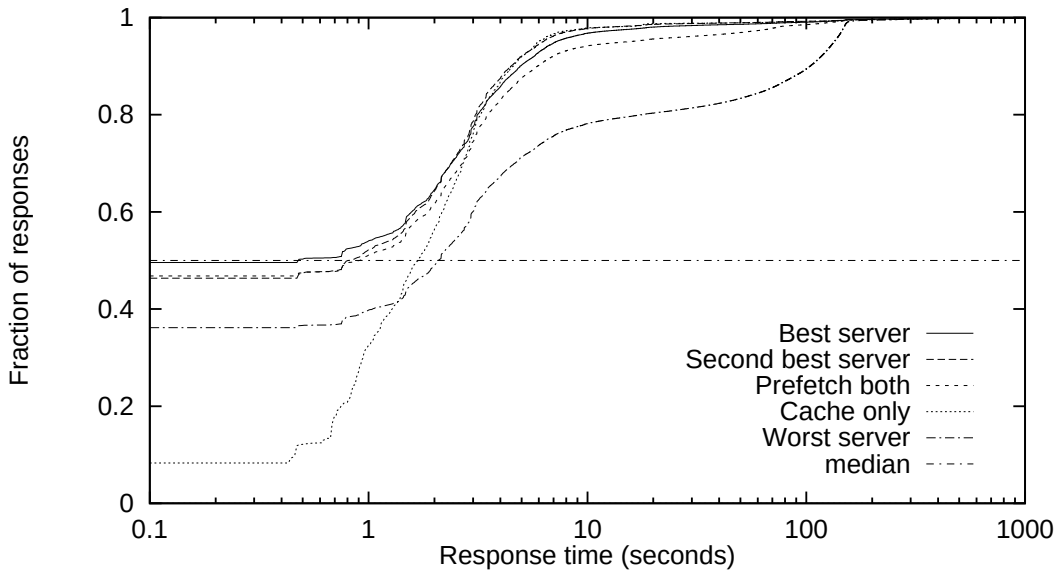


Figure 9.8: CDF of simulated response times for various configurations using the SSDC trace.

best points (along the lower left edge) are primarily those with minimum repetitions of 10 or 20. Thus, it is obvious that better predictions are made when the model has seen the pattern many times.

In Figure 9.8, we plot the cumulative distribution function of the response times for various configurations on the SSDC trace. At the bottom we show the caching-only configuration. The top curve is occupied primarily by the configuration getting the very lowest median response time. However, while it does have a better hit rate and median, its curve ends up being slightly worse than our "Second best server" for response times greater than two seconds. For comparison, we note that the really poor performing configurations, such as our "Worst server" shown here, can perform much worse than the caching-only configuration when they overwhelm limited resources, causing congestion and delays.

While not plotted graphically, we can compare the summation of all response times simulated for each configuration (e.g., the total waiting time experienced by the simulated users). We find that in comparison to the caching-only configuration, the worst configuration cost users an additional 344% in delays, while the best saved 8.15% in waiting time. Surprisingly, the second best saved users more than 15% in delays, while

transmitting only 80% additional bytes compared to the caching-only configuration. This is explained by the "best" server's poorer response times for documents that were large, causing suboptimal performance after about 80% of responses in Figure 9.8.

9.4.2 Web server log: NASA

9.4.2.1 Initial experiments

In addition to the SSDC trace above, we provide prefetching analysis from a sample (12 days, 858123 requests) of a second Web server trace, from NASA [Dum95]. For this simulation using NCS, we assume a server connected at T1 speeds (1.5Mbps, 10ms one-way latency) and clients identical to that of the SSDC simulations (approximately 120kbps and 40ms one-way latency).

Again, the client will have an additional 2MB cache. All prediction models will use n-grams between 1 and 3 requests long, and the scenario must be seen at least 10 times before a prediction is made with a probability of at least .1. Each predictor will be permitted to make just one prediction.

Table 9.3 summarizes the performance of the various configurations. It provides the mean and median response time for each approach, as well as the bandwidth required, and the fraction of requests served by prefetched content, plus the fraction of requests that prefetched content that was unusable. Using the same experiments, Figure 9.9 shows the distribution of client-side response times. The bottom-most line corresponds to the performance seen without any prefetching (just caching). As a result, it has the worst of all performances with a mean and median response times of 1.62s and .47s while using 90.8% of the original bandwidth used without a cache. Client-based prefetching

Approach	Response time		Fraction of original bw	% reqs pref'd successfully	% reqs pref'd unsuccessfully
	mean	median			
Prefetch both	1.37s	0.19s	131.7%	38.4%	46.6%
Prefetch server	1.38s	0.20s	129.0%	38.0%	48.7%
Prefetch client	1.61s	0.46s	91.0%	1.0%	0.2%
Cache only	1.62s	0.47s	90.8%	0%	0%

Table 9.3: Summary statistics for prefetching performance when one prediction is made with the NASA log.

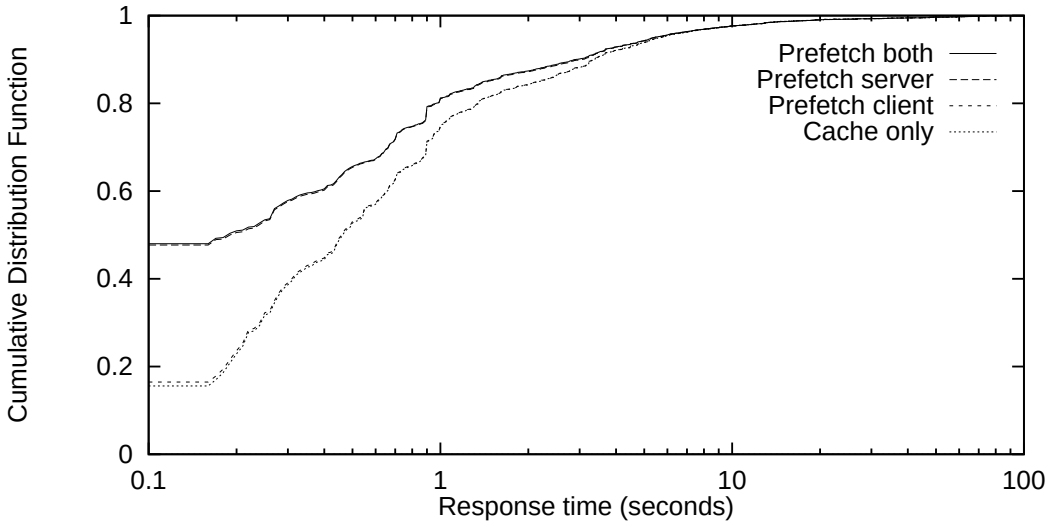


Figure 9.9: Cumulative distribution of client-side response time when only one prediction is used in the NASA trace.

helps slightly, reducing mean and median response times to 1.61s and .46s, at a cost of an additional 1% of bandwidth. Server-based prefetching helps tremendously; median response time drops by more than half to .2s, and mean by 15% to 1.38s, but at a cost of 29% more than the original bandwidth. The combination of client and server predictions results in only slightly better performance — mean response time stays the same, and median drops to .19s. Bandwidth usage increased slightly to 131.7%.

9.4.2.2 Parameter studies

We now examine whether performance changes for simulations based on the NASA trace in the same way that performance changed for SSDC-based simulations.

The initial NASA configurations tested above are limited. In fact, with ideal parameter settings, it is possible to serve more than half of the requests from client cache. This makes the median response time metric useless for comparative purposes (but potentially great for marketing — "reduced median response time by 100%!"). Instead, in Figure 9.10, we compare the 75^{th} percentile response time to the mean, again for the various configurations of our prefetching system that uses server hints for the NASA trace. Unlike the SSDC data, it shows that many configurations are able to improve both 75th percentile response times and mean response times. However, we note again

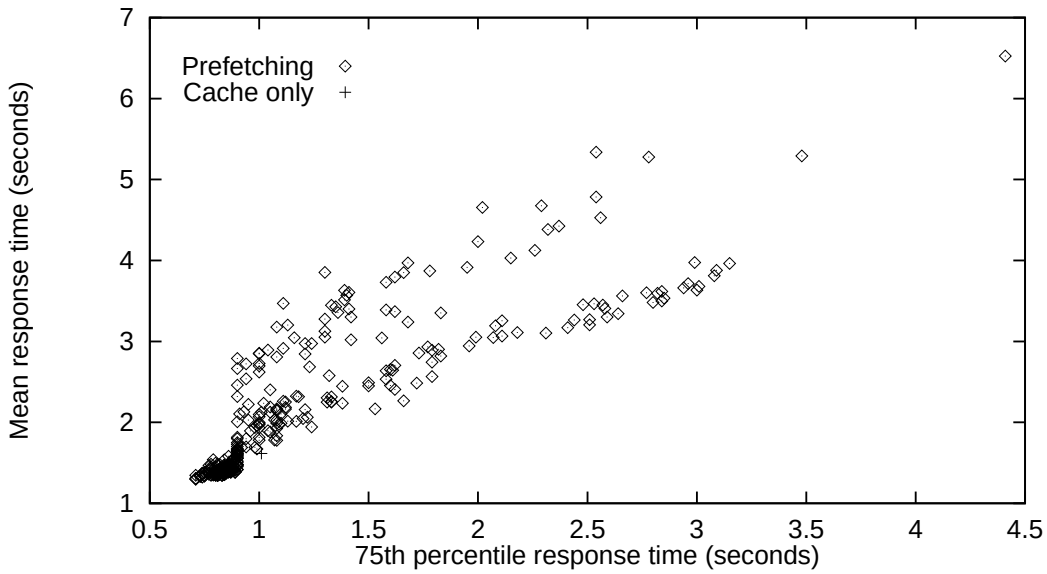


Figure 9.10: Scatterplot comparing 75th percentile versus mean response times for various configurations using the NASA trace.

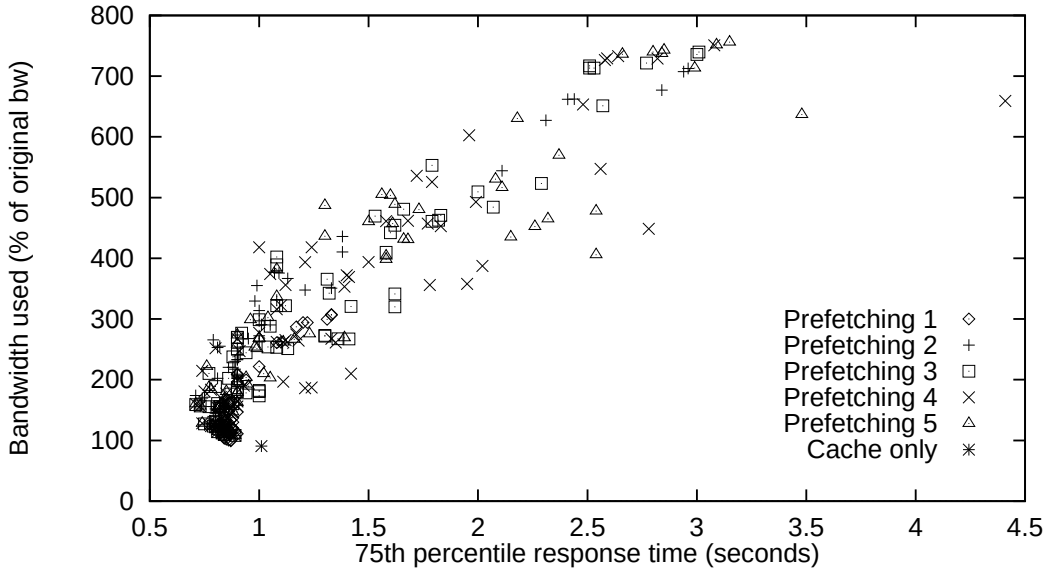


Figure 9.11: Scatterplot comparing median response time versus bandwidth consumed using the NASA trace, highlighting the number of predictions permitted.

that poor configurations can hurt in both dimensions.

We compare the cost of prefetching to its benefits on the 75^{th} percentile response times in Figure 9.11. We also distinguish among the number of predictions permitted. Like the SSDC experiments, it is apparent that many prefetching approaches are capable of reducing the median response time while using no more than twice the bandwidth of

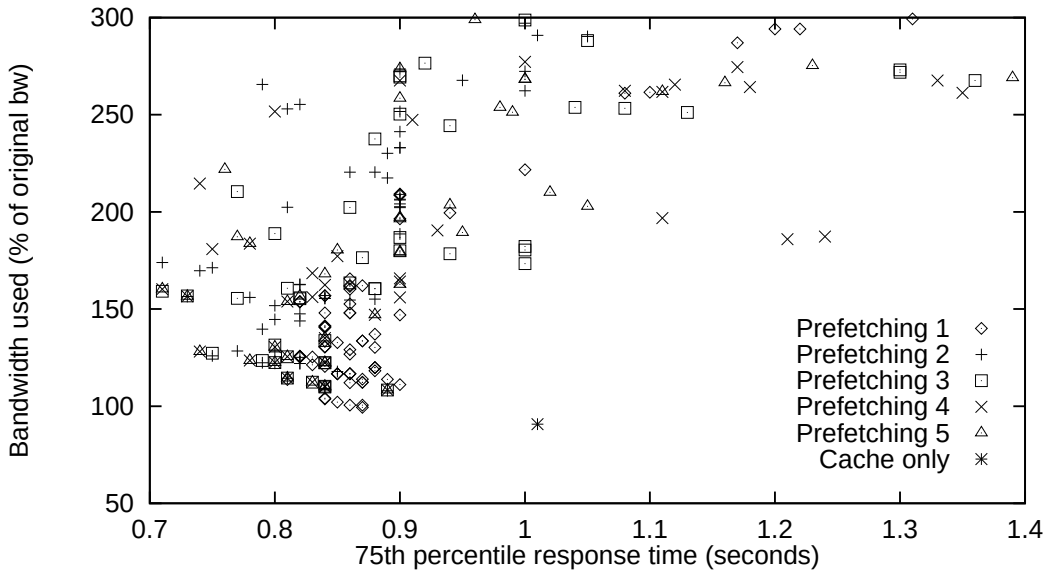


Figure 9.12: A magnified view of the best points in a scatterplot comparing median response time versus bandwidth consumed using the NASA trace, highlighting the number of predictions permitted.

the original trace. The figure also demonstrates that many configurations are able to cut more than 25% off the caching only response times. While not shown, the caching-only configuration had a median response time of 1.615 seconds, and more than half of the configurations tested achieved hit rates of greather than 50%, making their median response-times zero.

Observations can also be made based on the number of predictions permitted. While a single best prediction does improve response time, they do not perform as well as successful configurations using a larger number of predictions. In constrast, configurations using two, three, four, or five predictions all succeed in getting significant improvements in latency without bandwidth requirements getting too large (seen better in Figure 9.12), but are also well-represented throughout the scatterplot.

The caching only configuration uses 90.8% bandwidth to provide a 1.01 seconds 75^{th} percentile response time. The best configuration achieved a response time of .71 seconds at a cost of 168.4% of the original bandwidth. It made up to five guesses using n-grams of length two, with a minimum of 5 repetitions and minimum probability level of .1.

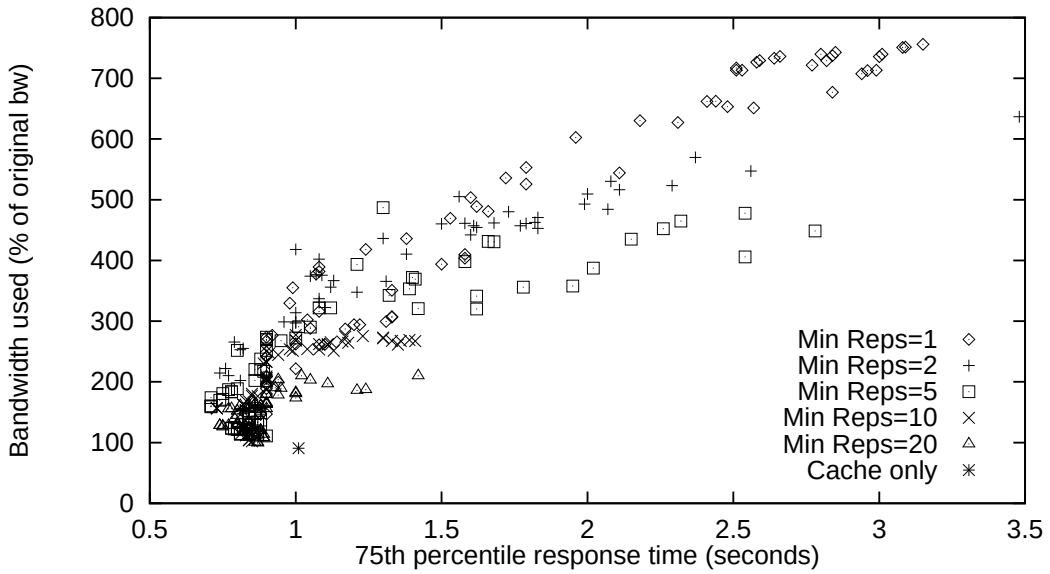


Figure 9.13: Scatterplot comparing median response time versus bandwidth consumed using the NASA trace, highlighting the minimum repetitions required.

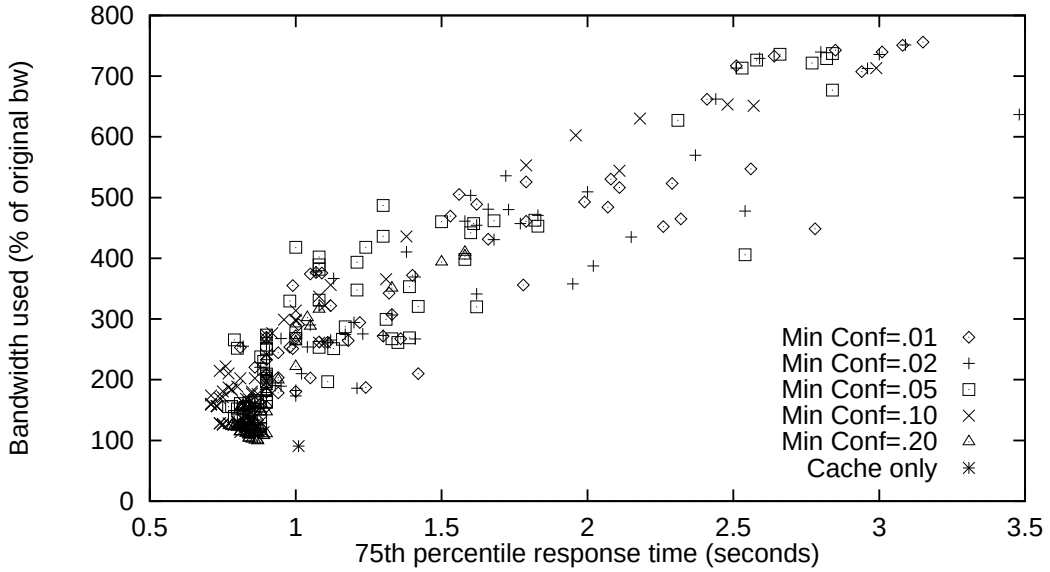


Figure 9.14: Scatterplot comparing median response time versus bandwidth consumed using the NASA trace, highlighting the minimum confidence required.

While the SSDC traces showed significance only for the number of repetitions, Figures 9.13 and 9.14 both show visible clustering. Typically the best performing points have minimums of at least five repetitions, and .1 probability.

In Figure 9.15 we plot the cumulative distribution function of the response times for extreme configurations on the NASA trace. Unlike the SSDC trace, we can see

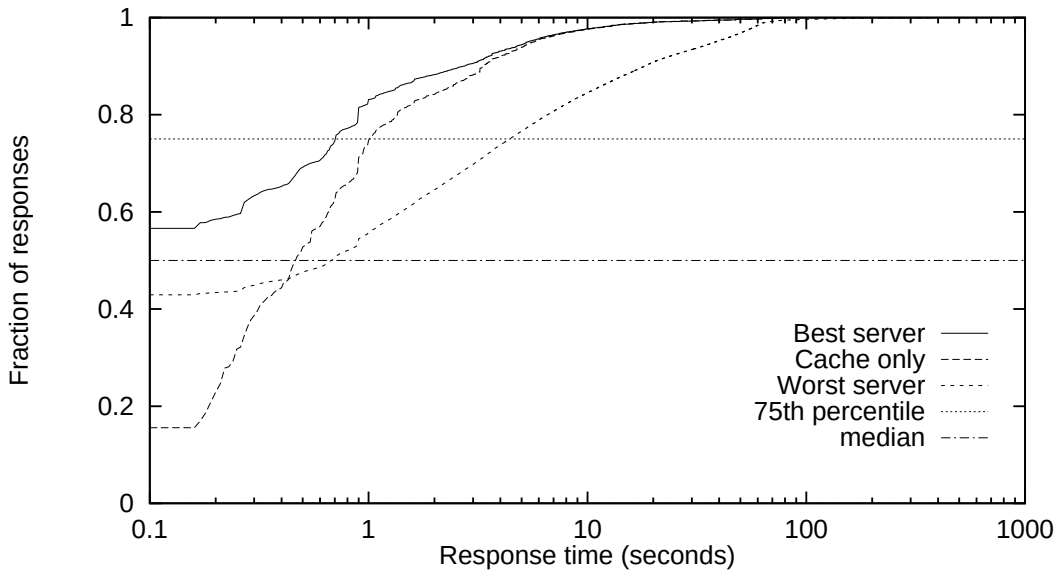


Figure 9.15: CDF of simulated response times for various configurations using the NASA trace.

improvements in response times for close to 95% of all responses. As in the SSDC case, the worst-performing configuration performs significantly worse for essentially all non-cached responses (which account for more than 40%).

Calculation of the sums of all response times simulated for each configuration demonstrates that in the worst case, users will wait an additional 304%, while in the best case, users will wait 19.7% fewer seconds while transmitting 85% additional bytes compared to the caching-only configuration.

9.4.3 Proxy server log: UCB

As described in Section 4.4.1, the UCB Home-IP Trace records the requests made by relatively low-bandwidth dialup and wireless users of the UC Berkeley Home IP service.

Unfortunately, straightforward simulation of history-based prefetching using the UCB trace finds essentially no improvement in client-side response times (in which the median is 1.91s). Plotting the CDF of the prefetching and non-prefetching implementations shows just a single overlapping curve. This result is initially perplexing, and so we consider it further below.

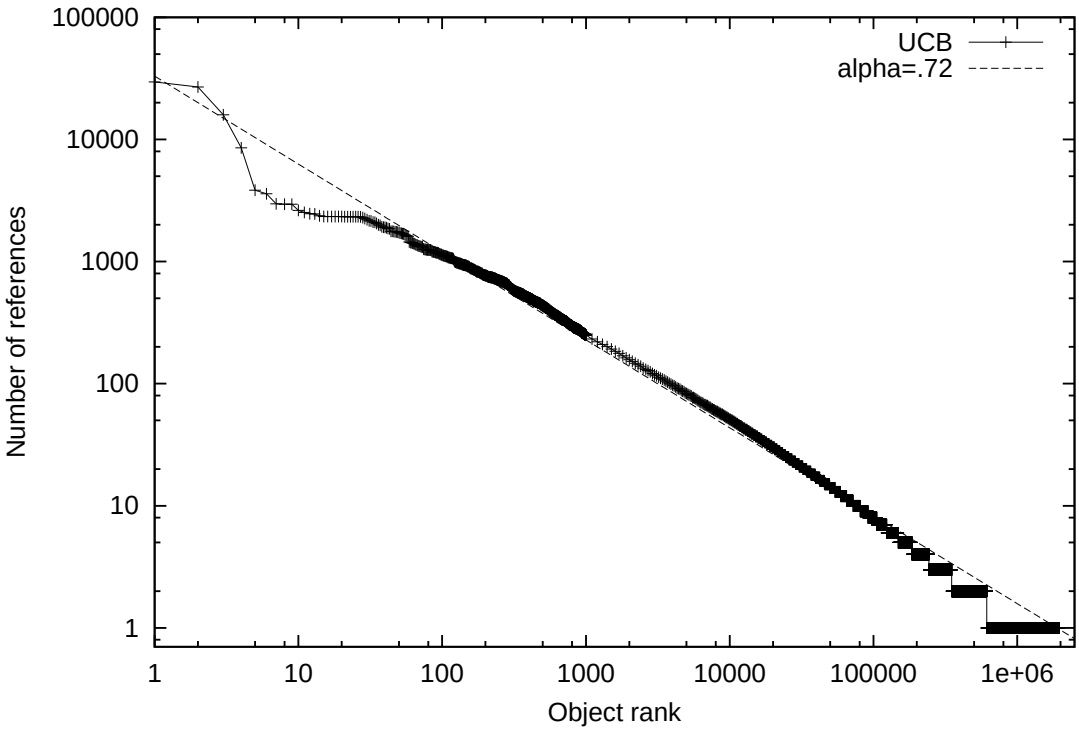


Figure 9.16: Frequency of request versus request rank for the UCB Home-IP trace.

9.5 Proxy Log Analysis

This chapter has explored two combinations of predictions — combining content and history, and combining multiple views of history. In both cases we demonstrated by simulation situations in which combinations can provide better performance than either prediction source alone.

However, the combination of multiple views of history do not appear to be effective when tested on the UCB Home-IP usage trace. While the Home-IP trace is large, containing millions of requests, it also has a large number of clients and a large number of servers (more than 40,000) represented. As a result, the number of requests per server is relatively low — much fewer than the server-specific traces we have discussed so far. Highly referenced servers in this trace contain between 20,000-100,000 references, but there are few such servers. Thus the primary difficulty is a lack of data. Without enough references to individual pages, they will not be predicted in the future.

A comparison of some of the characteristics of the UCB trace with others may be

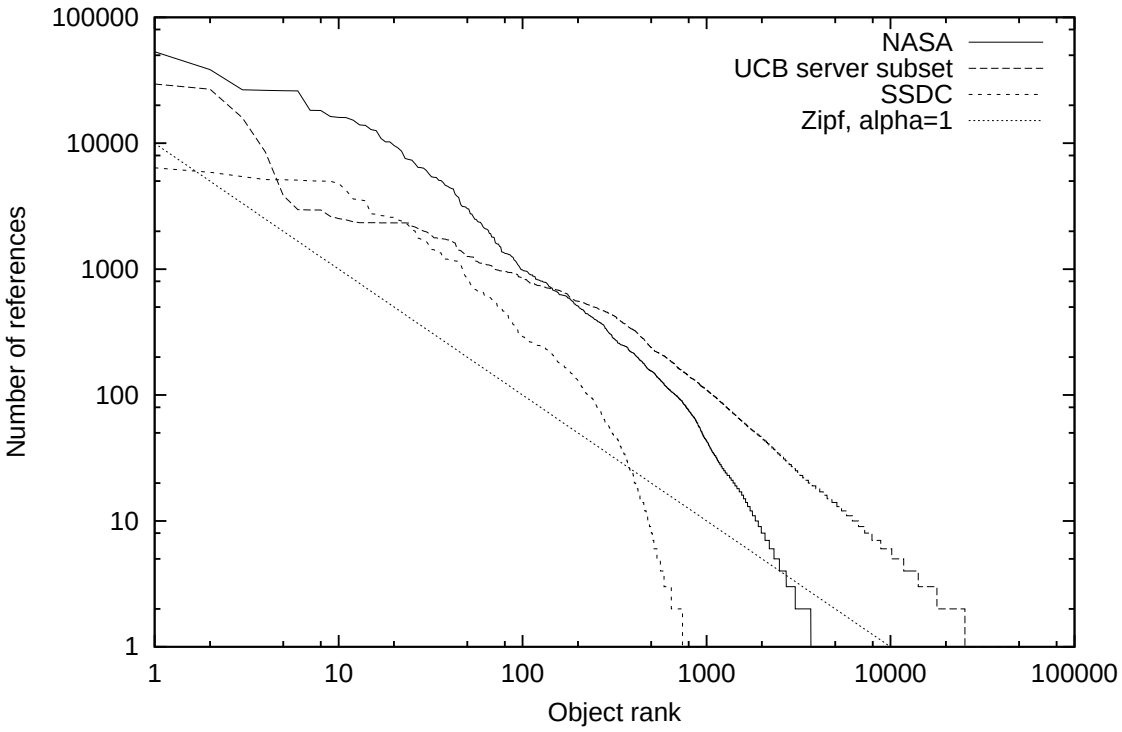


Figure 9.17: Frequency of request versus request rank for various server traces.

helpful. Figure 9.16 shows the number of times a particular request was made versus the rank of that request (where a rank of 1 signifies the most popular) for the UCB proxy trace. Figure 9.17 shows the same for a number of server traces as well as subsets of the UCB trace. A CDF view of all traces can be found in Figure 9.18. The UCB trace is a good fit to the Zipf-like model, in which the relative probability of a request for the ith most popular page is inversely proportional to $1/i^{\alpha}$. However, the server traces are less good of a match. If a Zipf curve were matched to them, α would be higher than typical,[1] and in general they do not as well match the log-log line typical of Zipfian distributions. In fact, other work has found that not all Web traffic has a Zipf-like request popularity with α less than 1 [PQ00]. While not identified as such, the effect is also visible in plots in earlier work [AW97].

One aspect that is apparent is that the Web server traces appear to have fewer infrequent requests. The least frequent request (a reference that appears once, often

[1]For example, [MWE00] shows proxy cache logs with values for α between .74 and .84 with good fits to rank vs. frequency of reference plots, and [BCF+99] examines a different set of proxy traces with α values between .64 and .83.

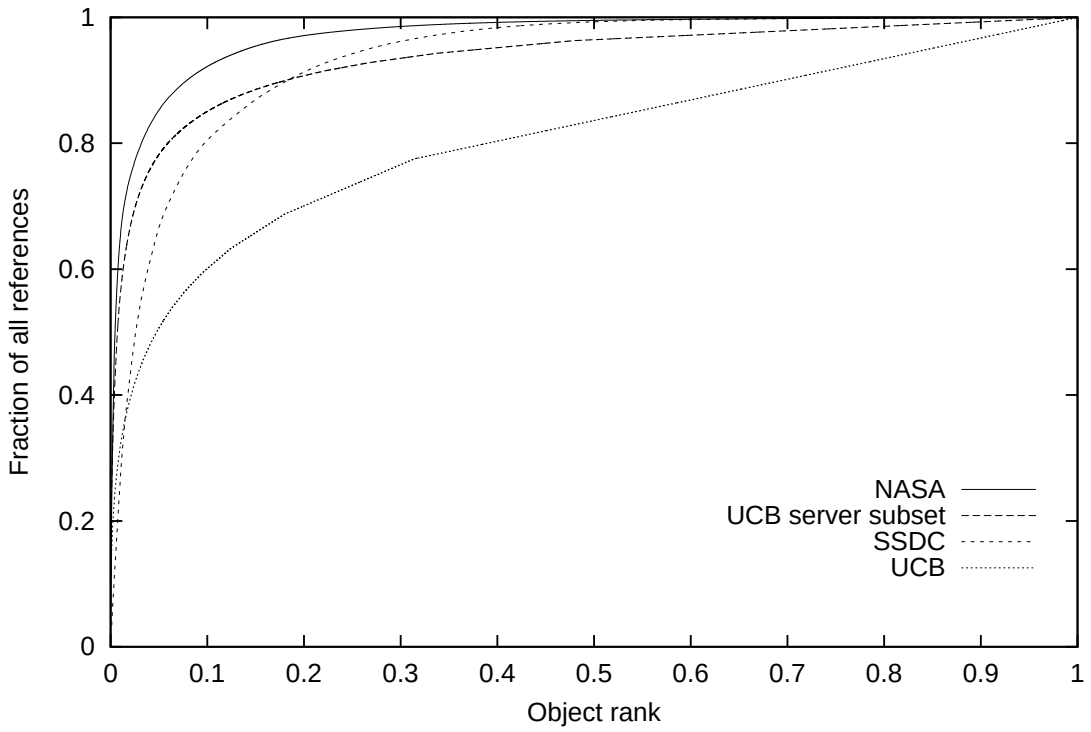


Figure 9.18: Cumulative distribution function of frequency of request versus request rank for various Web traces.

called a one-timer) is much more prevalent in the UCB trace than the others. One timers account for 22.5% of all references. In fact, pages referenced less than 10 times account for more than half of all references (51.8%). In contrast, pages referenced less than 10 times in the NASA trace account for just 1% of all references, and in the SSDC trace account for even less — .6% of all references. In the UCB servers subset, this accounts for 12.9% of all references. For a set of proxy traces, Mahanti *et al.* [MWE00] reports one-timers consisting of between 17.8% and 47.5% of all requests.

Figure 9.19 depicts the cumulative distribution functions of the frequency of page references (that is, the ordered list of the number of distinct times some object was referenced versus the total number of requests for pages in that category). It shows, for example, that all three UCB-based traces (the original UCB trace, a 100,000 request UCB subset, and the UCB server subset) have a larger fraction of requests for rare objects than the standard server-based traces (NASA and SSDC).

The more typical analysis is concerned primarily with the number of objects that

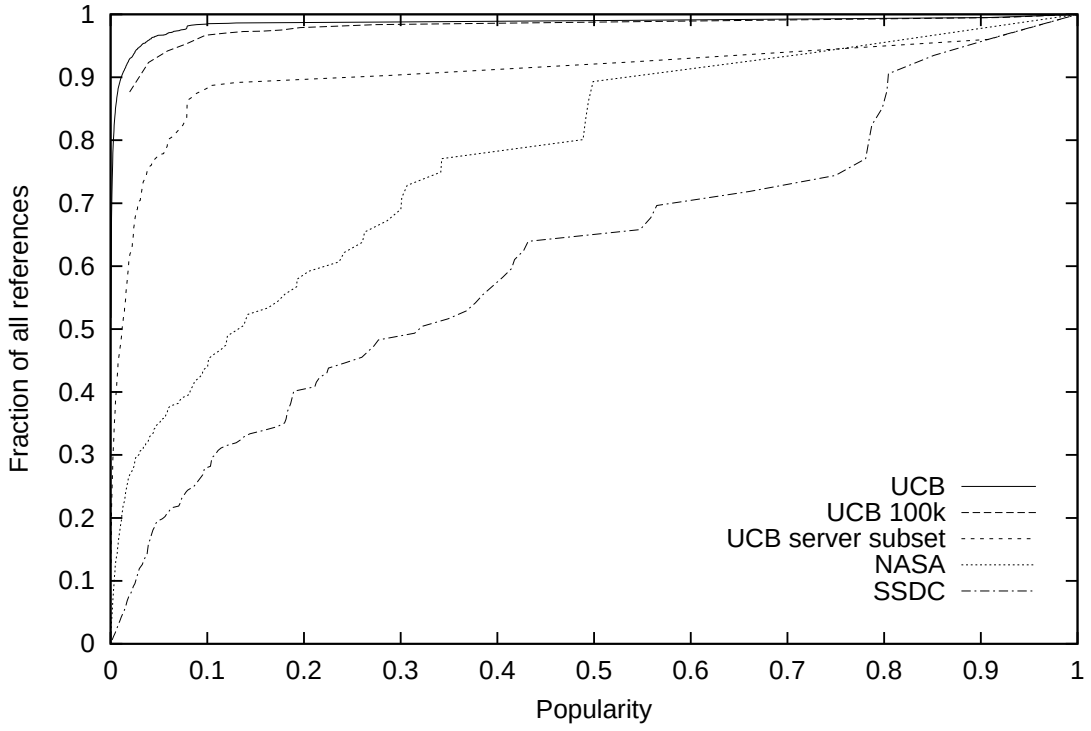


Figure 9.19: The CDF of frequency of page references, ordered by increasing frequency.

account for a large percentage of requests (thus ideal for caching). In the NASA trace, the top-ranked 1% of references account for 65% of the trace. For the SSDC trace, it is 80.5%. Similarly, Doyle *et al.* [DCGV01] describes very large traces from IBM's web site in which a small number of popular objects account for an unusually large percentages of requests. In a 1998 trace, the top 1% of objects generate 77% of all requests. In a 2001 IBM trace, the top 1% accounted for 93% of all requests. In contrast, the top-ranked 1% of UCB references account for only 31.8% of all references.

Therefore, it appears that the popularity of documents in proxy traces are quite different from that of Web server traces. The large fraction of one-timers and other rare pages make an informed prediction difficult, if not impossible.

While the UCB trace as a whole makes effective prefetching difficult, there are still some servers represented within it that have a reasonable number of requests made to them. Thus we selected the top-twenty servers (that is, those that served the most requests). These twenty servers (out of more than 40,000) handled a total of 12.5% of

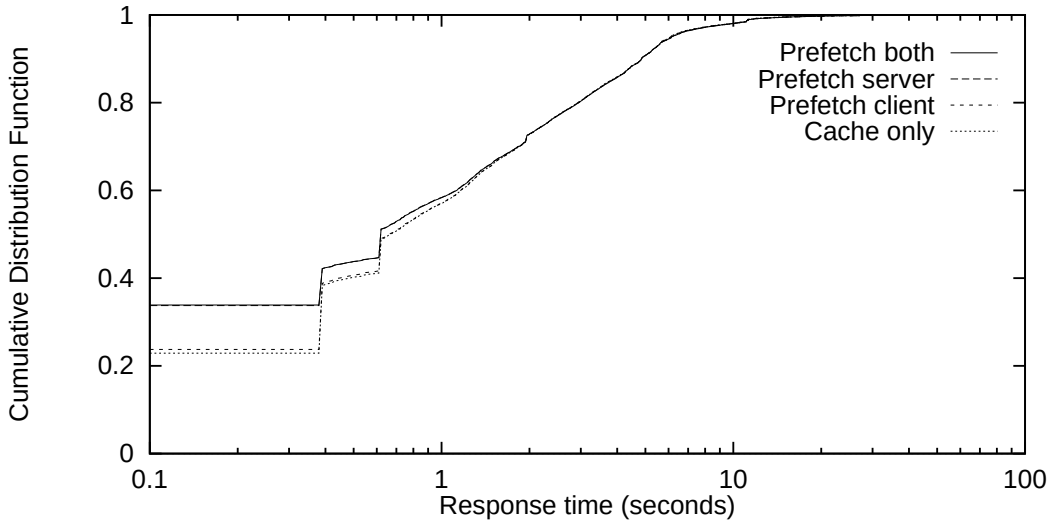


Figure 9.20: CDF of response times for various prefetching configurations using the UCB server subtrace.

all requests in the trace. However, the trace examined still only represents 12 days of traffic, which is relatively short time period for clients to learn and benefit from past experience.

NCS was used to simulate client response times under a variety of prefetching scenarios, much as we did with the server logs earlier in this chapter. The results are shown in Figure 9.20. Using the server's prediction model (either for server alone, or in combination with the client's model), we get a improvement in median response time of 50ms. While not reflected in the response times, the client model does provide increased request hits — without prefetching, 170941 requests were satisfied by the 2MB client cache. Client prefetching increases this to 177441 hits. When added to server prefetching, the benefit is much smaller — an increase from 251954 to 252807.

As a result of relatively strict limits on when predictions are made, the number of requests did not affect bandwidth utilization significantly. Even when prefetching based on client and server predictions, bandwidth utilization was just 16.4% higher than the original trace, and just 24.2% higher than just passive caching with the 2MB client cache.

9.6 Summary

This chapter has explored two combinations of predictions — combining content and history, and combining multiple views of history. In both cases we demonstrated that the combination of predictions may provide better performance than either alone.

We also explored performance as prediction parameters were varied. Simulation experiments with NCS in this chapter demonstrated that prefetching on server traces could cut median response times in half or better, and reduce overall client-perceived response times by 15-20%, by transmitting an additional 80-85% bytes over what would be transmitted by the equivalent caching-only configuration.

From this we found that the minimum repetitions threshold was an important value for getting high quality predictions, while the minimum confidence threshold was much less useful. The number of predictions and the size of the n-grams used for matching was less important — arguably useful points were found with various values of those parameters, although a single prediction did appear to be less helpful than two or more.

However, we've also used this chapter to point out some characteristics of when prefetching will not be particularly effective. In particular, when rare objects comprise a large fraction of all requests (as they do in the UCB proxy trace), history-based predictions are at a severe disadvantage. While intuitions and some data suggest that longer traces (i.e., building a model over a longer period of time) will help, the real answer will be to use multiple prediction sources for prefetching. Content-based prediction is one answer, as is the use of server hints when they are available.

Chapter 10

Simultaneous Proxy Evaluation (spe.tex)

10.1 Introduction

While Chapters 8 and 9 considered the value of simulation for estimating performance, they are primarily of value to researchers trying to test ideas. However, simulators are generally unable to evaluate the performance of *implemented* systems. Simulation often abstracts away many of the implementation-specific details — such as memory and disk speeds, network interrupts, processor characteristics, etc. — which become important when they vary between systems. Indeed, this is one reason why benchmarks are common — they provide a mechanism to test and compare performance on something close to a real-world workload, hopefully exercising those details in the process.

This chapter presents a new architecture for the evaluation of proxy caches. Initially, it grew out of research in techniques for prefetching in Web caches. In particular, we found that existing mechanisms for the evaluation of proxy caches were not well suited to prefetching systems. Objective evaluation is paramount to all research, whether applied or academic. Since this is certainly relevant when exploring various approaches to prefetching, we considered the range of existing proxy evaluation methods, and found that they each had distinct disadvantages. This led to the design of the more general architecture described below, and to its implementation which will be described in Chapter 11.

Our evaluation method, the Simultaneous Proxy Evaluation (SPE) architecture, reduces problems of unrealistic test environments, dated and/or inappropriate workloads, and allows for the evaluation of a larger class of proxies. It objectively measures proxy response times, hit rates, and staleness by forming a wrapper around the tested proxies to monitor all input and output. In the next section we motivate the need for the

SPE architecture by describing the space of proxy evaluation methodologies. We then specify our new evaluation architecture, and comment on implementation issues. We conclude the chapter by describing a set of sample evaluation tasks that would utilize it, and relate SPE to other work.

10.2 Evaluating Proxy Cache Performance

A variety of proxy caches exist in the market. (Web sites such as `web-caching.com` [Dav02b] provide a detailed list of products and vendors.) The features and performance provided sometimes vary more than the marketing literature would suggest. As more and more Internet users access the Internet through proxy servers everyday, these factors play a significant role in determining the perceived quality of the Internet.

Therefore, evaluating the various aspects of proxy performance is essential. There are, in fact, many characteristics of proxy caches that could be evaluated:

- The peak and average number of requests that can be serviced by the proxy cache per unit time.
- The object hit ratio — the number of objects that are serviced from the proxy's cache against the total number of objects served by the proxy.
- The byte hit ratio — the sum of the bytes served from the proxy's cache against the total bytes served by the proxy.
- The average response time as experienced by the user.
- Bandwidth used by the proxy. In addition to being affected by hit rates, a prefetching proxy is likely to use more bandwidth than a non-prefetching proxy.
- Robustness of a proxy.
- Consistency of the data served by a proxy.
- Conformance to HTTP standards.

Many of these are implementation characteristics, and thus can only be evaluated by techniques that work with complete systems. The SPE architecture (described in the next section) is designed to work at this level and is concerned with many of the characteristics of proxy cache performance: bandwidth usage, retrieval latencies, robustness and consistency. As a result, it provides benefits complementary to existing techniques described in Chapter 7. It is, however, the first mechanism to allow for the simultaneous black-box comparison of competing mechanisms on live, real-world data in a real-world environment.

10.3 The SPE Architecture

The SPE architecture can use actual client requests and the existing network to evaluate multiple proxies. It records timing measurements to calculate proxy response times and can compute page and byte hit rates. In summary, it forms a wrapper around competing proxies and produces logs that measure external network usage as well as performance and consistency as seen by a client. By simultaneously evaluating multiple proxies, we can utilize a single, possibly live, source of Web requests and a single network connection to provide objective measurements under a real load. Importantly, it also allows for the evaluation of content-based prefetching proxies, and can test for cache consistency.

10.3.1 Architecture details

In the SPE architecture, clients are configured to connect to a non-caching proxy, the *Multiplier*. Requests received by the Multiplier are sent to each of the proxy caches which are being evaluated. Each attempts to satisfy the request. To do so, each either returns a cached copy of the requested object, or forwards the request to a parent proxy cache, the *Collector*. The Collector then sends a single copy of the request to the origin server for fulfillment when necessary (or potentially asks yet another upstream proxy cache for the document, as shown back in Figure 2.5). By forcing the proxies to use the Collector, we can prevent two types of problems that might otherwise arise: 1) an increase in traffic loads on the network or at the origin server, and 2) side-effects from

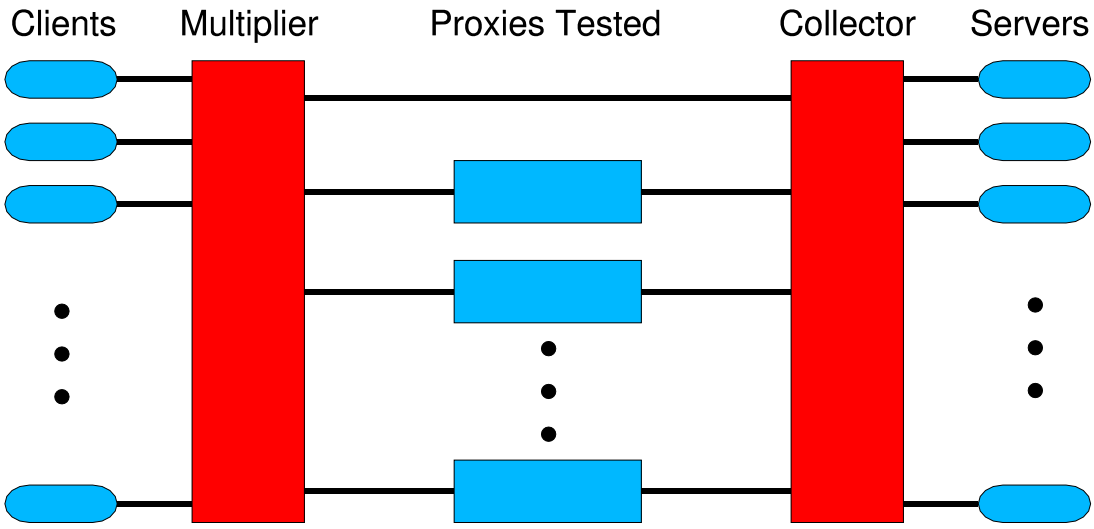


Figure 10.1: The SPE architecture for online evaluation of competing proxy caches.

multiple copies of non-idempotent requests [FGM+99, Dav01a]. The Multiplier also sends the requests directly to the Collector to use the responses to service the client requests and validate the responses from the test proxies. See Figure 10.1 for a diagram of this architecture.

Thus the Multiplier does not perform any caching or content transformation. It forwards copies of each request to the Collector and the test proxies, receives responses, performs validation and logs some of the characteristics of the test proxies. The clients view the Multiplier as a standard HTTP proxy.

Only minimal changes are required in clients, the proxies, or the origin servers involved in the evaluation. The clients need to be configured to connect to the Multiplier instead of the origin server and with popular browsers this is straightforward. The test proxies need to be configured with the Collector as the parent proxy. No changes are required to the Web servers which ultimately service the requests.

When a user of a proxy-enabled browser clicks on a link, the browser sends the request for that URL to the proxy. The proxy, in this case the Multiplier, takes the request and sends a duplicate request to each of the proxies under evaluation and directly to the Collector. Some proxies may have the object in the cache, in which case they return the object quickly. Others may experience a *cache miss*, or need to verify the contents of cache, and so have to send a request through the Collector.

Without a proxy Multiplier, client browsers would need modifications to send requests to each proxy under evaluation. Instead, one can configure off-the-shelf browsers to use the Multiplier as their proxy. In addition to request duplication, the Multiplier calculates request fulfillment times for each of the proxies being evaluated. The Multiplier also determines which response is passed back to the client — either arbitrarily, or by an algorithm such as by first response.[1]

Each of the proxy caches being evaluated can be treated as a black-box — we do not need to know how they work, as long as they function as a valid HTTP/1.0 [BLFF96] or 1.1 [FGM+99] proxy cache. This is helpful in particular for commercial proxies, but in general eliminates the requirement for either detailed knowledge of the algorithms used or source code which is needed for simulation and specialized code additions for logging [MR97] respectively. Typically the proxy caches would run on different machines, but for simple tests that may not be necessary. Alternatively, the same software with different hardware configurations (or vice versa) can be tested in this architecture.

In order to prevent each proxy cache from issuing its own request to the destination Web server, we utilize a proxy Collector which functions as a cache to eliminate extra traffic that would otherwise be generated. It cannot be a standard cache, as it must eliminate duplicate requests that are normally not cacheable (such as those resulting from POSTs, generated by cgi-bin scripts, designated as private, etc.). By caching even uncacheable requests, the Collector will be able to prevent multiple requests from being issued for requests that are likely to have side effects such as adding something to an electronic shopping cart. However, the Collector must then be able to distinguish between requests for the same URL with differing arguments (i.e., differing fields from otherwise identical POST or GET requests).

The Collector returns all documents with the same transmission characteristics as when fetched from the destination server. This includes the amount of time for initial connection establishment as well as transmission rate. By accurately simulating a

[1]Since the first response is likely to be a cached response, it will affect (positively) the users' perceived performance during the test. On the other hand, the response might be stale or erroneous, so a conservative approach would be to send back the response obtained from the Collector directly.

"miss", the Collector 1) enables the black-box proxy to potentially use the cost of retrieval in its calculations, and 2) allows the Multiplier to record response times that more accurately reflect real-world performance. Without it, any requests for objects that are in the Collector's cache will inappropriately get a performance boost.

Each proxy tested thus sees a single proxy (the Multiplier) as client, and a single proxy (the Collector) as parent. If the tested proxies only cache passively, then the overall bandwidth used by the system should be exactly that which is needed to serve the client requests. If one or more proxies perform active revalidation or prefetching, then the overall bandwidth used will reflect the additional traffic.

The architecture attempts to tolerate problems with the test proxies. Problems like non-adherence to protocol standards, improper management of connections, and the serving of stale responses can be caught and logged by the Multiplier. Note that while the SPE architecture is designed to preserve safety for passive proxy caches (allowing just one copy of a request to go to the origin server), it cannot provide guarantees of safety for non-demand requests that use the standard HTTP/1.1 GET mechanism [Dav01a].

To summarize, traditional evaluation of caching algorithms has been via trace-based simulations. We propose an online method, useful because it:

- takes existing proxy caches and uses them as black-boxes for evaluation purposes.
- can test the proxy caches on real-world data.
- eliminates variation in measured performance due to changes over time in network and server loads.
- eliminates variation in measured performance due to "dated" request logs.
- allows for the evaluation of content-based prefetching systems without having to locally mirror all pages referenced in a trace or to generate an artificial workload complete with internal links.
- allows for the evaluation of prefetching systems that might fetch pages never seen by the user (and thus not included in a traditional trace).

- allows for the measurement of consistency by comparing objects returned from proxies with that fetched directly.
- simultaneously compares different proxies on the same request stream.
- can evaluate cache freshness.
- can be used in the environment where the proxy will be used.

We realize that any method has drawbacks. We have identified the following disadvantages of the SPE architecture:

- Evaluating multiple prefetching schemes may place a higher demand for bandwidth on the network. If bandwidth usage is already high, the increased demand may cause congestion.
- Multiple copies of comparable hardware may be needed. At least $n + 1$ systems are needed to compare n different proxy caches.
- The Multiplier and Collector will introduce additional latencies experienced by users. However, our expectation is that this will be less than that of a two-level cache hierarchy (since the Multiplier does not do any caching).
- The proxy Multiplier effectively hides the client IP address. It may be useful for a proxy to perform tasks differently depending on the client, such as predicting future accesses.
- Caches are evaluated independently. Hierarchies are not easily included in the SPE architecture, even though better performance is often claimed when part of a hierarchy.
- SPE tests may not be replicable. This drawback applies to any method using a live workload or network connection.
- The Multiplier and Collector must manage large numbers of connections (discussed further below).

10.3.2 Implementation issues

The SPE architecture described has characteristics that may not be entirely feasible in some circumstances. For example, to eliminate any requirements for user intervention, there should be an existing proxy that can be replaced by the Multiplier (i.e., at the same IP and port address), or the Multiplier and Collector combination could be configured to transparently handle and route HTTP requests. In the earlier description of the SPE architecture, all HTTP requests go through all proxies when not filled by the previous proxy's cache. In particular, all black-box proxy misses go through the Collector. To ensure its use, the Collector should be placed on a firewall so that there is no chance of the proxies being tested getting around it.

To preserve client performance during the test, the Multiplier contacts the Collector directly. It also uses the object returned from the Collector as the standard against which the results of the proxies can be compared for consistency. When a proxy returns a stale object (in comparison) or an error, that can be recorded. By sending a request directly to the Collector, the Multiplier also eliminates the difficulty of determining which response to return to the client, and provides some degree of robustness in case of failure of the proxy caches being tested.

In general, any implementation of the SPE architecture should be as invisible to the participating software as possible. For example, caching connections has been shown to be a significant factor in cache performance [CDF+98], so the Multiplier should support HTTP/1.0 + Keep-Alive and HTTP/1.1 persistent connections to clients for performance, but should only attempt to contact the proxies at the same level at which the client contacted the Multiplier. In this way the Multiplier will attempt to generate the same ratio of HTTP/1.0 and 1.1 requests as it received. Likewise, the Collector may receive a range of request types from the proxies under evaluation depending on their capabilities. For highest fidelity, the Collector would record when it connects to destination Web servers that support HTTP/1.1 and use that information to selectively provide HTTP/1.1 support for requests from the tested proxies.

Unfortunately, the requirements for an accurate Collector are high. If one is unavailable, a traditional proxy cache may need to be used as a Collector, which means it would not attempt to cache the "uncacheable" objects, nor return objects with accurate timing characteristics. This allows for the calculation of hit-rates but not comparative average response time values. Additionally, some proxies may not be able to handle firewalls, so they may attempt to fetch some objects directly. Therefore, instead of passing all requests to the proxies being tested, the Multiplier may need to be configured to fetch non-cachable requests (cgi, forms, URLs containing "?", etc.) directly to ensure that only one copy gets delivered to the origin server.

Other difficulties include implementing a sufficiently robust Multiplier and Collector — for high-load measurements (such as that needed for enterprise-level proxies [MR97]), they need to be able to handle a higher load than the best black-box proxy being tested, and support $k + 1$ times as many connections (where k is the number of black-box proxies). In general, this difficulty is significant — under the SPE architecture, the Proxy and Collector have to manage a large number of connections. As a result, this architecture is not well suited for stress-testing proxies, but instead provides insight to their performance on (perhaps more typical) non-peak workloads.

10.4 Representative Applications

To demonstrate the potential of the SPE architecture, we present the following scenarios. Each exhibits how some aspect of the SPE architecture can be valuable in evaluating alternative proxies.

10.4.1 Measuring hit-rates of black-box proxy caches

Consider first the task of comparing bandwidth reduction across multiple black-box proxy caches, including some that perform prefetching on a believable client load. Since the internal workings may not be known, simulation is not viable, and in order to evaluate the prefetching caches, a realistic workload with full content is desirable. This is a typical evaluation task for corporate purchasing but is also similar to that needed

by researchers evaluating new prototypes.

Properly implemented, the SPE architecture can be used to channel all or part (with load balancing hardware) of the existing, live client request load through the multiple proxies under evaluation. Since both incoming and outgoing bytes to each proxy can be measured, the calculation of actual bandwidth savings on the current workload and network connection can be performed. Additionally, the SPE architecture is sufficient to measure the staleness of data returned by each proxy. Finally, note also that with some loss of generality, a minimal test of this sort can be made without implementing the Collector but instead test the proxies on cachable requests only (with the Multiplier fetching uncacheable requests directly).

10.4.2 Evaluating a cache hierarchy or delivery network

In an early version of this work [Dav99b] we proposed the use of the SPE Multiplier for evaluating the latency benefits of using a cache hierarchy, such as NLANR [Nat02]. In this model, the Collector is not used, and the Multiplier replicates requests, sending them to different places. To test a cache hierarchy, we would send one request to the hierarchy, and the other directly to the origin server. Likewise, to test a content delivery network, one would send the request to the CDN and to the origin server.

There are many concerns with this kind of evaluation, however. Most worrisome is the replication of non-idempotent requests (those causing side-effects) that eventually get back to the origin server. We discuss this issue further in Chapter 12. If that were not an issue (say, for static, highly cacheable objects), then the remaining concerns are in methodology. One must take care in measuring external services as they cannot be restarted and the evaluation itself can affect their performance. For example, by sending the request to an origin server, that request will warm the server, potentially placing the object in a server-side cache, or at least loading the relevant files in a file buffer. Then, in the case that the cache hierarchy or CDN does not have a fresh copy of the content requested, they can get it from the origin at a faster rate than would otherwise be the case. Likewise, servers that employ CDNs to handle delivery of static content may not often serve such content or be tuned to do so, and thus be at a disadvantage

for a comparison which ideally would test equally best cases or average cases on both sides.

10.4.3 Measuring effects of varying OS and hardware

In this task, a single proxy caching solution is used, such as Squid [Wes02]. The question being asked is that of which version of UNIX is best performing, or what hardware is sufficient to get the best performance for a particular organization's workload. If the various installations of Squid are identical, the application-level operations of the cache will be almost identical. Thus, the difference in performance from using an IDE versus SCSI drives, or performance from a stock OS installation versus one that is custom-tuned can be determined. Since SPE allows for each to receive the same request load and generate a single network load, the relative performance of each configuration can be calculated under the current loads and network capabilities of the particular organization.

10.4.4 Evaluating a transparent evaluation architecture

Finally, consider the problem of evaluating the overhead of a transparent evaluation architecture. An implementation of SPE, for example, imposes two additional proxies between client and server (the Multiplier and Collector). Each of these is likely to add some latency to handle connection requests and to copy data. In addition, the Collector portion of SPE provides caching services, and like other proxy caches, has the potential to serve stale data.

Given a SPE implementation and an appropriate (artificial) workload, the SPE implementation could be used to evaluate itself, to find out how much latency it introduces and to attempt to verify the consistency of the Collector implementation.

10.5 Related Work

A number of researchers have proposed proxy cache (or more generally just Web) benchmark architectures (e.g., the Wisconsin Proxy Benchmark [AC98b, AC98a], WebMonitor [AAY97a, AAY97b], hbench:Web [MSC98], and Surge [BC98b]). Some use artificial traces; some base their workloads on data from real-world traces. They are not, however, principally designed to use a live workload or live network connection, and are generally incapable of handling prefetching proxies.

Web Polygraph [Rou02] is an open-source benchmarking tool for performance measurement of caching proxies and other content networking equipment. It includes high-performance HTTP clients and servers to generate artificial Web workloads with realistic characteristics. Web Polygraph has been used to benchmark proxy cache performances in multiple Web cache-offs [RW00, RWW01].

Koletsou and Voelker [KV01] built the Medusa Proxy, which is designed to measure user-perceived Web performance. It operates similarly to our Multiplier in that it duplicates requests to different Web delivery systems and compares results. It also can transform requests, e.g., from Akamaized URLs to URLs to the customer's origin server. The primary use of this system was to capture the usage of a single user and to evaluate (separately) the impact of using either: 1) the NLANR proxy cache hierarchy, or 2) the Akamai content delivery network. The first test is quite similar in concept to that made by Davison [Dav99b] as a partial application of the SPE architecture, which also explored the potential for use of the NLANR hierarchy to reduce retrieval times, but which assumed the use a small local proxy cache in between the browser and the hierarchy. While acknowledging the limitations of a small single-user study, their paper also uses a static, sequential ordering of requests – first to the origin server, and then to the NLANR cache. The effects of such an ordering (such as warming the origin server) are not measured. Other limitations of the study include support only for non-persistent HTTP/1.0 requests and fixed request inter-arrival time of 500ms when replaying logs.

Liston and Zegura [LZ01] also report on a personal proxy to measure client-perceived

performance. Based on the Internet Junkbuster Proxy [Jun00], it measures response times and groups most requests for embedded resources with the outside page. Limitations include support for only non-persistent HTTP/1.0 requests, and a random request load.

Liu *et al.* [LAJF98] describe experiments measuring connect time and elapsed time for a number of workloads by replaying traces using Webjamma [Joh98]. The Webjamma tool plays back HTTP accesses read from a log file using the GET method. It maintains a configurable number of parallel requests, and keeps them busy continuously. While Webjamma is capable of sending the same request to multiple servers so that the server behaviors can be compared, it is designed to push the servers as hard as possible. It does not compare results for differences in content.

While all of the work cited above is concerned with performance, and may indeed be focused on user perceived latencies (as we are), there are some significant differences. For example, our approach is designed carefully to minimize the possibility of unpleasant side effects — we explicitly attempt to prevent multiple copies of a request instance to be issued to the general Internet (unlike Koletsou and Voelker). Similarly, our approach minimizes any additional bandwidth resource usage (since only one response is needed). Finally, while the SPE Multiplier can certainly be used for measuring client-side response times if placed adjacent to clients, it has had a slightly different target: the comparative performance evaluation of proxy caches.

10.6 Summary

In this chapter we presented an online evaluation method, called the SPE (Simultaneous Proxy Evaluation) architecture, which simultaneously evaluates different caching strategies on the same request stream. This is in contrast to existing evaluation methodologies that use less realistic test environments and potentially out-of-date or irrelevant workloads. Importantly, it also allows for the evaluation of content-based prefetching proxies, and can test for cache consistency. To help motivate our unique approach to cache evaluation, we have provided some sample applications to demonstrate the

potential of the proposed architecture.

The SPE architecture has been designed in particular for use in an online environment with real traffic. However, it can still be useful as an evaluation technique with artificial workloads or trace logs as in the experiments reported here. In fact, artificial workloads are often useful for evaluating systems under workloads that cannot easily be captured or are in excess of current loads. In this way one can capture the statistical properties of specialized workloads and still retain the benefits of simultaneous evaluations on a live network.

The SPE architecture is particularly useful for evaluation of black-box proxies and prefetching systems. It can be used to help validate manufacturer claims and provide comparisons between systems using live networks. Evaluation systems that implement the SPE architecture will be welcome additions to the set of measurement tools available to cache designers and network engineers alike. In the next chapter we describe such an implementation.

Chapter 11

SPE Implementation and Validation

11.1 Introduction

There are many possible methods to evaluate caching algorithms and implemented proxy caches, as we describe in Chapter 7. Of those that examine real systems, the typical approach is to apply benchmark tests (e.g. [AC98b, AC98a, BC98a, MJ98, GCR98, BD99, Rou02]) to either measure performance with a standard load, or to determine the maximum load attainable. The current *de facto* process is to compare many proxies using the same Polygraph [Rou02] workload (e.g., in a caching competition [RW00, RWW01]). Unfortunately, these approaches are unusable for evaluating proxy caches that preload content (such as [CY97, Cac02a]), and do not necessarily test on real-world workloads.

A preloading proxy cache typically chooses what to preload based on either user retrieval history or the content of recently retrieved pages. Neither are modeled well (if at all) in artificial workloads and datasets. In the case where the workload is a captured trace from real users, the content is typically unavailable (at least not in the form that the users saw). Additionally, since a preloading proxy cache may choose to preload content that is never actually used, a captured trace will not be able to provide resource characteristics such as size or retrieval time.

For these reasons, we have proposed in Chapter 10 an approach using simultaneous proxy evaluation (SPE), potentially using a live network and client workload. In this chapter we describe an initial implementation of SPE, and our experiences with it. In the next sections we describe our application-layer implementation, and discuss issues that arose from the implementation and how we resolved them. We provide some baseline experiments in Section 11.4 to help validate its performance. In Section 11.5,

we demonstrate the use of our implementation by detailing two sample experiments that evaluate multiple proxies simultaneously. We wrap up with a discussion of future work and conclusions reached from this effort.

11.2 Implementation

Accompanied by a few simple log analysis tools, the Multiplier and Collector are the two major pieces of software that make up any SPE implementation. In our system the Multiplier is a stand-alone multi-threaded program which is compatible with HTTP/1.1 [FGM+99] and HTTP/1.0 [BLFF96] protocols and was developed from scratch. The publicly available Squid proxy server [Wes02] version 2.5PRE7 was modified to develop the Collector.

11.2.1 The Multiplier

The Multiplier listens on a configurable port for TCP connections from clients and assigns a thread from a pre-generated pool for every client connection opened. Connections to the tested proxies are handled by this process using `select(2)`. When a complete request is read, the process opens a TCP connection with the Collector and test proxies, sends the request to them (the HTTP version of this request is the same as the one sent by the client) and waits for more requests (in the case of persistent connections) and responses. Persistent connections are allowed if the client requests it (the version of Squid used to build the Collector supports persistent connections). However, the test proxies are not required to support persistent connections. If they do not, a new connection is opened every time a client sends a request over a persistent connection in the same session. An alternative approach is to not support persistent connections at all, which is clearly not desirable when some proxies do support persistent connections. In any case, the cost of establishing a connection will be recorded each time for a proxy which does not support client-side persistent connections.

Figure 11.1 shows the headers of a sample request and response. The Multiplier

Sample HTTP/1.0 request:

```
GET http://news.bbc.co.uk/fn.gif HTTP/1.0
Referer:  http://news.bbc.co.uk/
Proxy-Connection:  Keep-Alive
User-Agent:  Mozilla/4.76 [en] (WinNT; U)
Host:  news.bbc.co.uk
Accept: image/gif, image/x-xbitmap, image/jpeg
```

Sample HTTP/1.0 response:

```
HTTP/1.0 200 OK
Date:  Mon, 26 Mar 2001 05:42:00 GMT
Server:  Apache/1.3.12 (Unix)
Last-Modified:  Fri, 10 Dec 1999 15:35:39 GMT
ETag:  "2c137-65f-38511dcb"
Content-Length:  1631
Content-Type:  image/gif
Proxy-Connection:  keep-alive
```

Figure 11.1: Sample HTTP/1.0 request from Netscape and response via a Squid proxy.

parses the status line of the request and response to record the request method, response code and HTTP version. The only other headers that are manipulated during interaction with client and proxies are the connection header ("Connection" and the non-standard "Proxy-Connection") to determine when the connection is to be closed and the content length header ("Content-Length") to determine when a request or a response is complete. The Multiplier adds a "Via" header to the requests it sends in conformance with HTTP/1.1 standards.

The Multiplier measures various features like the time to establish connection, the size of object retrieved, the initial latency, the total time to transfer the entire object and the MD5 checksum [Riv92] of the object. A data structure with this information is maintained and once all the connections are closed, they are logged, along with the results of validation of responses. Validation of responses are performed by comparing the status codes, the header fields and the MD5 checksums of the responses sent by collector and each test proxy. Mismatches, if any, are recorded. Figure 11.2 shows some sample lines from the Multiplier log.

11.2.2 The Collector

In this work, we have made changes only to the source code of Squid to implement our partial SPE tool. We have not modified the underlying operating system.

```
Wed Jun 20 02:10:11 2001:(pid=4630):RESULT: Collector:  0:34195 0:34090 5083 {DUMMY,}
192.168.0.2 non-validation HTTP/1.0 GET http://www.cs.rutgers.edu/

Wed Jun 20 02:10:11 2001:(pid=4630):RESULT: Proxy-1:  0:108105 0:50553 5083
{200,200,OK,} 192.168.0.2 non-validation HTTP/1.0 GET http://www.cs.rutgers.edu/

Wed Jun 20 02:10:11 2001:(pid=4630):RESULT: Proxy-2:  1:180157 0:3868 5083
{200,200,OK,} 192.168.0.2 non-validation HTTP/1.0 GET http://www.cs.rutgers.edu/

Wed Jun 20 02:10:11 2001:(pid=4630):RESULT: Proxy-3:  0:113010 0:34160 5083
{200,200,OK,} 192.168.0.2 non-validation HTTP/1.0 GET http://www.cs.rutgers.edu/
```

Figure 11.2: Sample Multiplier log showing the timestamp, the initial latency and transfer time in seconds and microseconds, the size of the object in bytes, the result of response validation, the client IP, the HTTP version, the method, and the URL requested. The possible results are OK (if validation succeeded), STATUS_CODE_MISMATCH (if status codes differ), HEADER_MISMATCH (if headers differ, along with the particular header) and BODY_MISMATCH (if the response body differs).

Recall from Chapter 10 that the Collector is a proxy cache with additional functionality:

- *It will replicate the time cost of a miss for each hit.* By doing so we make it look like each proxy has to pay the same time cost of retrieving an object.
- *It must cache normally uncacheable responses.* This prevents multiple copies of uncacheable responses from being generated when each tested proxy makes the same request.

The basic approach used in our system is as follows: if the object requested by a client is not present in the cache, we fetch it from the origin server. We record the connection setup time, response time and transfer time from the server for this object. In subsequent requests from clients for the same object in which the object is in the cache, we delay some time before sending the first chunk of the cached object, and we also delay some time before sending subsequent chunks, so that even though this request is a cache hit, the total response time experienced is the same as a cache miss.

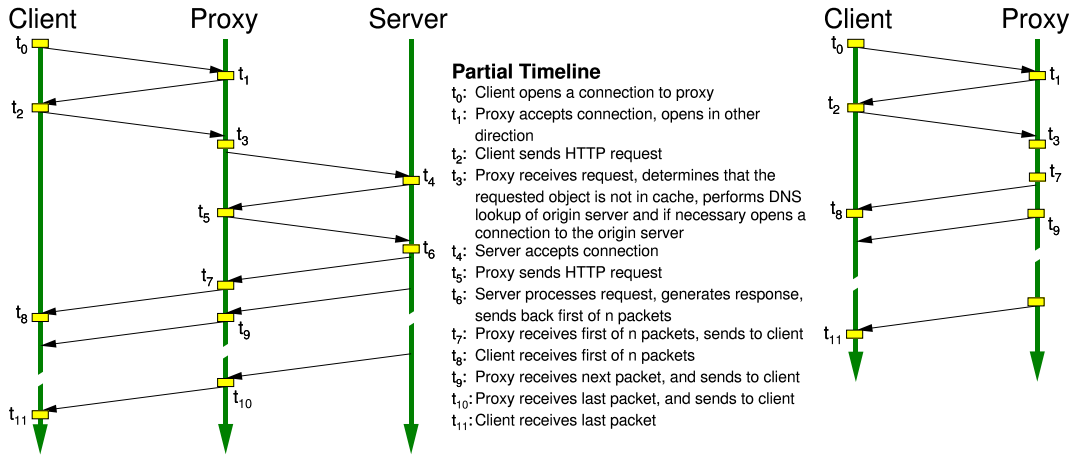
The function *clientSendMoreData* in *client_side.c* takes care of sending a chunk of data to the client. We have modified it so that for cached objects, the appropriate delays are introduced by using Squid's built-in event scheduling apparatus. Instead of sending the chunk right away, we schedule for the desired time the execution of a new function to send the chunk instead.

```
986185568.949 677 128.6.60.20 TCP_MISS/200 13661 GET http://news.bbc.co.uk/cele0.jpg -
DIRECT/news.bbc.co.uk image/jpeg

986185568.965 2 128.6.60.79 TCP_HIT/200 13660 GET http://news.bbc.co.uk/cele0.jpg -
NONE/- image/jpeg

986185569.404 1914 128.6.60.20 TCP_MISS/200 34543 GET http://news.bbc.co.uk/1254610.stm
- DIRECT/news.bbc.co.uk text/html

986185570.593 926 128.6.60.79 TCP_HIT/200 34550 GET http://news.bbc.co.uk/1254610.stm -
NONE/- text/html
```

Figure 11.3: Sample Collector (Squid) log showing the timestamp, elapsed time in ms, client IP address, cache operation/status code, object size, request method, URL, user ident (always '-'), upstream source, and filetype.

To cache uncacheable objects in Squid, two modifications are required. The algorithm which determines the cacheability of responses has to be modified to cache these special objects. In addition, when a subsequent request is received for such an object, the cached object should be used instead of retrieving it from the origin server, since these objects would normally be treated as uncacheable. Squid's *httpMakePrivate* function in *http.c* marks an object as uncacheable and the object is released as soon as it is transferred to the client. Instead of marking it private, we modified the code so that the object is valid in the cache for a configurable amount of time (a minimum *freshness time*). Subsequent requests for this (otherwise uncacheable) object would be serviced from the cache if they are received before the object expires in the cache. The Collector also keeps track of the clients which have seen a particular uncacheable object with a bit mask. If a client has already retrieved this object and makes a request for it again before its expiration, the object will be fetched from the origin server (since such a client would want a fresh copy).

If an object is validated with the origin server when an If-Modified-Since (IMS) request is received and a subsequent non-IMS request is received for the same object, then to fake a miss, the cost for the entire retrieval of the object is used and not the cost for the IMS response. While handling POST requests, an MD5 checksum of the body of the request is computed and stored when it is received for the first time. Thus in subsequent POSTs for the same URL, the entire request has to be read before it could be determined whether the cached response can be used to respond to the request.



(a) Sequence for proxy cache miss.

(b) Sequence for proxy cache hit.

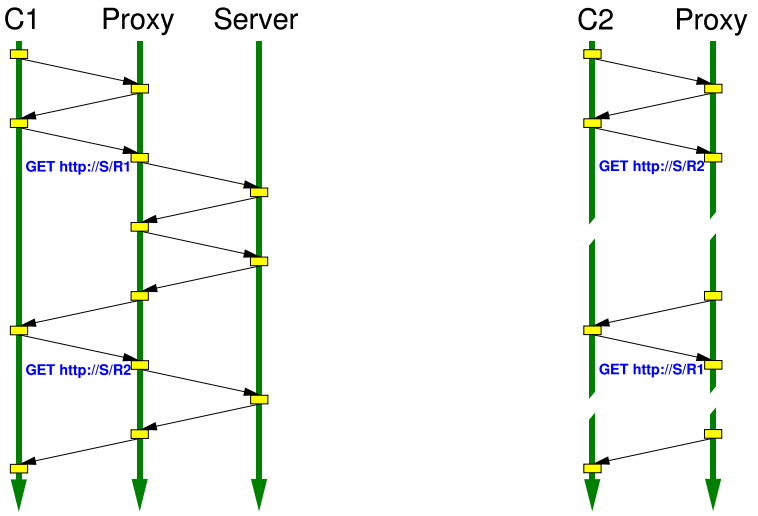
Figure 11.4: Transaction timelines showing the sequence of events to satisfy a client need. The diagrams do not show TCP acknowledgments, nor the packet exchange to close the TCP connection.

Similarly, requests with different cookies are sent to the origin server, even though the URLs are the same. Figure 11.3 shows some lines from a sample Collector log (identical to a standard Squid log).

11.3 Issues and Implementation

In order to reproduce miss timings as hits, we need to be careful about some of the details. In Figure 11.4a, we show a transaction timeline depicting the sequence of events when using a non-caching proxy (or equivalently, when the proxy does not have a valid copy of the content requested). In contrast, the transaction timeline in Figure 11.4b illustrates the typical sequence of events when a caching proxy has a valid copy of the requested content and returns it to the client.

In the case that the client has an idle persistent connection to the proxy, the transaction would start with t_2. If the proxy had an idle connection to the desired origin server, a new connection would not be necessary (merging the actions at t_3 and t_5 and eliminating the activity between). From the client's perspective, the time between t_2 and t_0 constitutes the *connection setup time*. The time between t_8 and t_2 constitutes the *initial response time*. The time between t_{11} and t_8 is the *transfer time*. The complete



(a) Client re-uses persistent connection.

(b) Second client finds cached data.

Figure 11.5: Transaction timelines showing how persistent connections complicate timing replication.

response time would be the time elapsed between t_{11} and t_0.

In the rest of this section, we discuss a number of the most significant issues encountered during implementation and how they have been addressed.

11.3.1 Persistent connections

Issue: Most Web clients, proxies and servers support persistent connections (which are optional in HTTP/1.0 and the default in HTTP/1.1). Persistent connections allow subsequent requests to the same server to re-use an existing connection to that server, obviating the TCP connection establishment delay that would otherwise occur. Squid supports persistent connections between client and proxy and between proxy and server. This process is sometimes called *connection caching* [CKZ99], and is a source of difficulty for our task.

Consider the case in which client $C1$ requests resource $R1$ from origin server S via a proxy cache (see the transaction timeline in Figure 11.5a). Assuming that the proxy did not have a valid copy of $R1$ in cache, it would establish a connection to S and request the object. When the request is complete, the proxy and server maintain the

connection between them. A short time later, $C1$ requests resource $R2$ from the same server. Again assuming the proxy does not have the resource, the connection would be re-used to deliver the request to and response from S. This arrangement has saved the time it would take to establish a new connection between the proxy and server from the total response time for $R2$.

Assume, however, a short time later, client $C2$ also requests resource $R2$, which is now cached (Figure 11.5b). Under our goals, we would want the proxy to delay serving $R2$ as if it were a miss. But a miss under what circumstances? If it were served as the miss had been served to $C1$, then the total response time would be the sum of the server's initial response time and transfer time when retrieving $R2$. But if $C1$ had never made these requests, then a persistent connection would not exist, and so the proxy would have indeed experienced the connection establishment delay. Our most recent measurement of that time was from $R1$, so it could be re-used. On the other hand, if $C2$ had earlier made a request to S, a persistent connection might be warranted. Similarly, if $C2$ were then to request $R1$, we would not want to replicate the delay that was experienced the first time $R1$ was retrieved, as it included the connection establishment time.

In general it will be impossible for the Collector to determine whether a new request from a tested proxy would have traveled on an existing, idle connection to the origin server. The existence of a persistent connection is a function of the policies and resources on either end. The Collector does not know the idle timeouts of either the tested proxy nor the origin server. It also does not know what restrictions might be in place for the number of idle or, more generally, simultaneous connections permitted.

Implementation: While an ideal SPE implementation would record connection establishment time separately from initial response times and transfer times, and apply connection establishment time when persistent connections would be unavailable, such an approach is not possible (as explained above). Two simplifications were possible — to simulate some policies and resources on the proxy and a fixed maximum idle connection time for the server side, or to serve every response as if it were a connection

miss. For this implementation, we chose the latter, as the former would require the maintenance of new data structures on a per-proxy-and-origin-server basis, as well as the design and simulation of policies and resources.

The remaining decision is whether to use persistent connections internally to the origin servers from the Collector. While a Collector built on Squid (as ours is) has the mechanisms to use persistent connections, idle connections to the origin server consume resources (at both ends), and persistent connections may skew the transfer performance of subsequent responses as they will benefit from an established transfer rate and reduced effects of TCP slow-start.

Therefore, in our implementation, we modified Squid to never re-use a connection to an origin server. This effectively allows us to serve every response as a connection miss, since the timings we log correspond to connection misses, and accurately represent data transfer times as only one transfer occurs per connection.

11.3.2 Request pipelining

Issue: Pipelining of requests (having more than one outstanding request at a time) is allowed by HTTP/1.1, but it requires some additional complexity in the client side, since if a transfer fails, it needs to keep track of the failed requests and reissue them.

Implementation: In our SPE implementation, the Multiplier sends pipelined requests to the Collector only when the client generates them. Thus, the client is responsible for reissuing the failed requests. Even though we accept them, we do not send pipelined requests to the test proxies and make sure that there is only one outstanding request at any point of time (for a single connection — if a browser opens multiple connections to the Multiplier, each is treated independently). The impact of this choice is that we do not know whether the tested proxies support pipelined requests, but also means that the Multiplier does not need to be responsible for regenerating failed requests.

11.3.3 DNS resolution

Issue: The first request to an origin server requires a DNS lookup to determine the

server's IP address. In some cases, this process can take a noticeable amount of time [CK00]. For subsequent HTTP requests to the same server, no lookup will be required, as the IP address will have been saved in cache. It is important that even though one proxy may have issued a request that encountered a delay because of DNS resolution times, requests by other proxies should still experience the same delay.

Implementation: In order to provide no advantage to slower proxies that make a request later than a faster one, we incorporate the DNS resolution actions as part of the server connection establishment time. This helps to more accurately replicate the delay seen by the first request.

11.3.4 System scheduling granularity

Issue: Most operating systems have a limit on the granularity of task scheduling that is possible. Unless it is modified within the OS, this limit will provide a lower-bound on the ability to match previously recorded timings.

Implementation: In Linux as well as other UNIX-like systems on the Intel architecture, the `select(2)` system call has granularity of 10ms. Thus, if we tell `select` to wait at most 15ms, and no other monitored events occur, it will likely return after approximately 20ms. Thus, in our initial implementation a large difference would accumulate between what we wanted to delay and the actual amount of time delayed. In order to solve this problem, we use the following method: given some granularity, we must decide to round up or down. That is, for our development platform, if we need to delay 15ms, do we tune it to 10ms or to 20ms? In our implementation we decide this probabilistically. We first extract the largest multiple of the system-specific granularity from our desired delay. The fraction remaining is then used as a probability to determine whether to delay an additional timeslice, and thus the end result is a stochastically chosen delay value that is at most one timeslice larger or smaller than our original need.

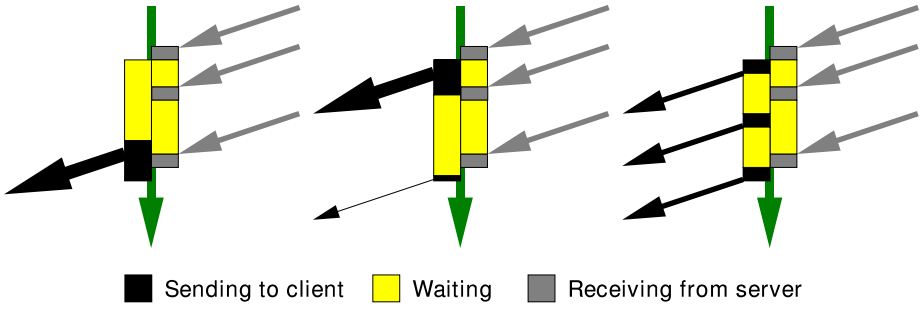
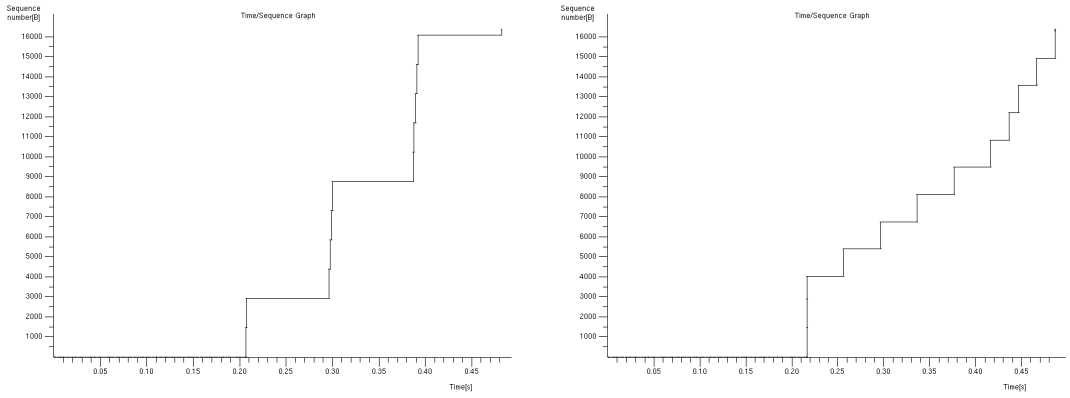


Figure 11.6: Possible inter-chunk delay schedules for sending to client.

11.3.5 Inter-chunk delay times

Issue: When Squid receives a non-trivial response from an origin server, it groups the data received in a single call into an object it calls a chunk, which it then stores into cache. Chunks then are the content of at minimum one packet, and possibly many packets, depending on the rate at which data is being received. In a cache miss, those chunks are sent on to the client as they are received. Since our goal is to replicate the miss experience as much as possible, it would be undesirable to wait and then send all data at the end, even though that might be a simplistic approach to replicate overall response time. Likewise, sending all data but the last byte until the desired time had passed would also be unreasonable, as it does not come close to replicating reality. Ideally, our system would send data at the same rate as it was received, and with the same inter-chunk delays. These three scenarios are depicted graphically in Figure 11.6. In each portion, the timings of when chunks were received are re-drawn on the right, while the various schedules for inter-chunk delays in the case of a cache miss are drawn on the left of the timeline.

The differences between these conditions are real — modern browsers will parse the HTML page as it is received (i.e., in chunks) to find embedded resources such as images to fetch, and when found, issue new requests for them. Prefetching proxies may in fact do something similar (e.g., [CY97, Cac02a]). Likewise, images with a progressive encoding may be rendered by a browser with low-resolution when the initial data is received and then be refined as the remaining data arrive.



(a) SPE Collector miss

(b) SPE Collector hit

Figure 11.7: Time/sequence graphs for one object.

Implementation: As described above, it may be insufficient to merely replicate the total response time — when data arrives within the response time period may be important. The overhead of recording the per-chunk delay was deemed too high for implementation. Instead, we replicate the average delay between chunks, as this only requires a fixed per-object storage overhead.

In order to accurately replicate the overall response time, we record the time difference between the time at which we receive the client request (t_3) and the time at which we begin to send the response (t_7) thus summing the server connection establishment time and the initial response time. In addition, we record the time difference between receiving the first chunk of response from the origin server (t_7) and the time when we receive the last chunk of response from the origin server (t_{10}) as the transfer time. As shown in Figure 11.4, this also corresponds to the time difference between just before sending the first chunk of response to the client and just before sending the last chunk of response to the client. So for subsequent requests for the same object, we intentionally add delays (in particular, before each chunk) so that the total time experienced between events t_3 and t_{10} are identical to the miss case.

We insert delays in two situations: before the first chunk (i.e., before event t_7) we delay the amount of time it took for the connection establishment and initial response

time from the origin server. Before sending subsequent chunks, we delay a portion of the transfer time. In that situation the delay portion is calculated by the following formula:

$$delay = \frac{chunk\ size}{totalsize} * transfertime$$

In our experiments we found that the above formula is not accurate enough as there is a non-negligible chance that the randomly chosen errors (because of select call granularity) will accumulate. Therefore, we use a global optimization approach and change the formula to be:

$$delay = \frac{chunk\ size}{bytesremaining} * timeremaining$$

where *bytesremaining* is defined as the total remaining bytes of the cached object that has yet to be sent, and *timeremaining* is defined as the delay time remaining that is to be distributed to the remaining chunks of the object.

Figures 11.7a and b demonstrate the approach. In 11.7a, the Collector passes packets to the client as they are received from the origin, in effectively four blocks. In 11.7b, the Collector closely replicates the start and end of the transmission, but uses ten (smaller) blocks to do so.

11.3.6 Persistent state in an event-driven system

Issue: Our implementation requires the recording of various time values, some of which are object-based, and some of which are request-based. For example, we record which proxies have retrieved a particular object. Request-based information includes the *timeremaining* delay left to be applied to the current request.

Implementation: Since Squid uses a single process, we cannot easily use global storage for persistent state. Thus, we have extended the structures of `_connStateData`, `_StoreEntry`, and `_clientHttpRequest` in order to record the information for later access.

11.3.7 HTTP If-Modified-Since request

Issue: Not all HTTP requests return content. Instead, some respond only with header information along with response codes — telling the client that the object has moved (e.g., response code 302), or cannot be found (404), or that there is no newer version (304). The latter is of particular interest because it forces the consideration of many situations. Previously we have considered only the case in which a client makes a request and either the proxy has a valid copy of the requested object or it does not. In truth, it can be more complicated. If the client requests object R, and the cache has an old version of R, it must check to see if a modified version is available from the origin server. If a newer one exists, then we have the same case as a standard cache miss. If the cached version is still valid, then the proxy has spent time communicating with the origin server that must be accounted for in the delays seen by the client. Likewise, the client may have a copy and need to verify its freshness with the proxy. If the proxy has a valid copy, it may wish to respond directly to the client, in which case the proxy needs to add delays as if the proxy had to confer with the origin server.

Implementation: We have considered IMS requests, and the general principle we use is to reproduce any communication delays that would have occurred when communicating with the origin server. Thus, if the proxy does communicate with the origin server, then no additional connection establishment and initial response time delay needs to be added. If the proxy is serving content that was cached, then as always, it needs to add transfer time delays (as experienced when the content was received). If the response served to the client does not contain content, then no transfer time is involved.

11.3.8 Network stability

Issue: An object may be cacheable for an arbitrary period of time. Since there exists significant variability in end-to-end network performance on the Internet [Pax99], network characteristics may have changed since the object was first retrieved.

Implementation: We use the times from the most recent origin server access of an object to be our gold standard. In reality, the origin server may be more or less busy and network routes to it may be better or worse. Thus, we have chosen consistent performance reproduction over technically accurate performance reproduction (which would preclude caching). Note that to minimize any adverse effects, we can limit the amount of time that we allow objects to be cached (since a new retrieval will get a new measurement of communication performance).

11.3.9 Caching of uncacheable responses

Issue: As described earlier, a SPE Collector must cache all responses, even those marked as uncacheable, to minimize the potential for duplicate requests to be sent to the origin server.

Implementation: In our Collector, the freshness time for uncacheable objects is set to a small amount of time (arbitrarily chosen to be 180 seconds) since we expect requests for such objects to appear from all interested proxies within a short time period and we don't really want to cache them for a long time — they are not supposed to be cached at all. In some cases, the object could be requested by a test proxy after its expiration in the cache (e.g., if one of the proxies prefetched an object that turned out to be uncacheable and the user makes a request for it later). In this case, it is retrieved again from the origin server. We cannot avoid this by increasing the freshness time since the Collector would be more likely to return stale data.

11.3.10 Request scheduling

Issue: If the Multiplier forwards requests to the Collector and the test proxies immediately after receiving a request from a client, there is a good chance that a test proxy will request an object from the Collector, whose transfer from the origin server has not started at all or whose response headers are still being read. In both these cases, a standard Squid proxy opens a parallel connection with the origin server to service the second request, since it does not have information about the cacheability of the object

and assumes it is safer to retrieve it separately. This is clearly not desirable, since we only want only one request to be sent to the origin server for every request sent by the client.

Implementation: Our Multiplier addresses this problem by sending the request only to the Collector first and only after receiving the complete response from the Collector, sends the request to each of the test proxies. Thus we ensure that when a test proxy passes a client request to the Collector, it will already be in the Collector's cache because the transfer of the object from the origin server is complete. Since the object is completely in cache, the Collector can serve subsequent requests from cache and avoid sending duplicate requests. However, this approach also implies that the inter-request timings seen by the tested proxies are not the same as those generated by the client, since the time that the proxy receives the request is dependent upon when the request is satisfied by the Collector.

11.3.11 Additional client latency

Issue: From Figure 10.1 which showed the SPE architecture and the communication paths between entities, it is clear that the object retrieved must traverse at least two more connections than if it is retrieved by the client directly from the origin server — one between the client and the Multiplier and the other between the Multiplier and the Collector.

Implementation: Though the SPE architecture introduces additional latency on the client's side, we expect it to be small since the SPE architecture typically runs in the same local network as the client. We will evaluate the overhead introduced by our implementation by running one instance within another in Section 11.4.4.

11.4 Validation

In this section, we describe our system configuration and three experiments that test and provide limited validation of our implementation.

11.4.1 Configuration and workloads

Software configuration

In the validation tests and in the experiments, we will use up to five proxy cache configurations. All are free, publicly available with source code. They are: Squid 2.4.STABLE6 [Wes02], Apache 1.3.23 (using mod_proxy) [Apa02], WcolD 961127_143211 [CY97], Oops 1.5.22 [Kha02], and DeleGate 7.9.9 [Sat02]. Each used default settings where possible. We selected a uniform 200MB disk cache for each (except for DeleGate which does not appear to support a maximum disk cache size). The source for each was unmodified except when needed to allow compilation on our development platforms and to fix a few WcolD bugs. In addition, for many tests we will also run a "Dummy" proxy, which is just our Multiplier configured to do nothing but pass requests on to a parent proxy. WcolD is the only one of the five proxy caches to support link and embedded resource prefetching.

To generate workloads, we used two tools. The first is `httperf` [MJ98], which we use whenever generating an artificial workload. The httperf tool is highly configurable, supports HTTP/1.1 and persistent connections. In some experiments, we will collect the per-request timing results captured by httperf to calculate statistics on the overall response times as seen by the client.

The other is the `simclient` tool from the Proxicizer [Gad01, GCR98] toolset[1] since it is able to replay captured Web logs in close to real time, preserving the inter-request delays that were originally present.

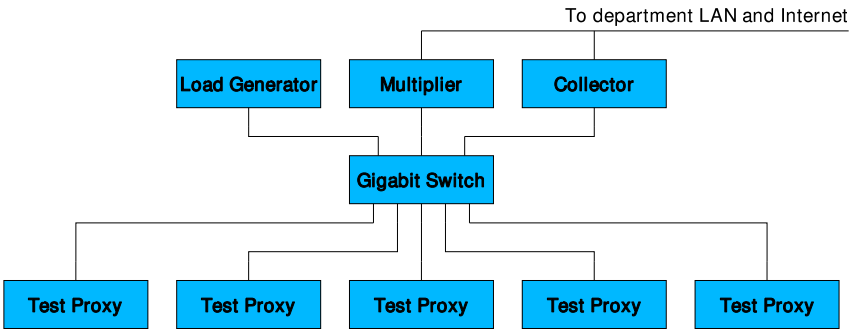


Figure 11.8: Experimental network configuration.

Network configuration

For each test we run a Multiplier and a Collector as described above. As shown in Figure 11.8, each proxy runs on a separate system, as do each of the monitoring and load-generating processes. Each machine running a proxy under test has a single 1Ghz Pentium III with 512MB RAM and a single 40GB IDE drive. All machines are connected via full-duplex 100Mbit/sec Fast Ethernet to a gigabit switch using private IP space. A mostly stock RedHat 7.3 version of Linux was installed on each machine. The Multiplier and Collector ran on similar systems but with dual 1Ghz Pentium III processors, 2GB RAM, and gigabit Ethernet cards, and were dual-homed on the private network and on a LAN for external access.

Data sets

In our experiments we use three representative data sets. To generate the first (entitled *Popular Queries*), we used the top fifty queries as reported by Lycos [Lyc02b] on 24 April 2002. Those queries were fed automatically to Google [Goo02], resulting in 10 URLs per query, for a total of 500 URLs that will be replayed by httperf. We eliminated three that caused problems for httperf in initial tests.

The second dataset (*IRCache*) was extracted from an NLANR IRCache proxy log (sv.ircache.net) on 18 April 2002. The intentions for this log of about half a million

[1]Proxycizer is a suite of applications and C++ classes that can be used in simulating and/or driving Web proxies.

entries were to focus on cacheable responses,[2] so we removed all log entries that had non-200 HTTP response codes or Squid status codes other than `TCP_HIT`, `TCP_MEM_HIT`, `TCP_IMS_HIT`, `TCP_REFRESH_HIT`, and `TCP_REFRESH_MISS`. This eliminated more than 75% of the log entries, leaving only those entries corresponding to objects that had been cached by the proxy or a client of the proxy. We then arbitrarily selected the URLs that ended in `.html` and eliminated any duplicate URLs. We additionally removed those that generated HTTP errors in initial tests, resulting in a set of 1054 URLs.

The third dataset (*Reverse proxy*) was selected to be a Web server workload, to be replayed in real time with simclient. We started with the access logs from the Apache [Apa02] Web server that runs a small departmental Web server, `www.cse.lehigh.edu`, for the month of May 2002. We filtered the original logs for malformed requests as well as requests for dynamic resources (using the heuristic that looks for URLs containing "cgi-bin", "?", or ".cgi", or use of the POST method). The remaining GET and HEAD requests to resources believed to be static numbered over 94,000.

Since the simclient tool from the Proxycizer suite required logs in some proxy format, we converted the Apache logs to Squid logs. Unfortunately, Apache logs have only a one second timestamp resolution, so even though Squid logs typically have millisecond resolution, these logs would not. These logs did, however, have multiple requests at exactly the same time, which is extremely unlikely with a single processor under relatively light load. Thus, we arbitrarily spaced requests that had identical timestamps 10ms apart in the converted trace.

From this log we selected a subset of three days (May 21-23) to be replayed through our SPE implementation. This represented just over 16,000 requests from 977 distinct client IP addresses.

A total of 196 robot IPs could be identified by looking for requests to `/robots.txt` in the full trace. We did not filter these clients in the three day test set, as they represent a portion of what a real reverse proxy would encounter. While we did not analyze the effects that these robots had on the caching performance, Almeida *at al.* [AMR+01]

[2]Proxy logs from NLANR's IRCache proxies [Nat02] cannot be replayed in general because some entries are incomplete. To preserve privacy, all query terms (parts of the URL after a "?") are obscured.

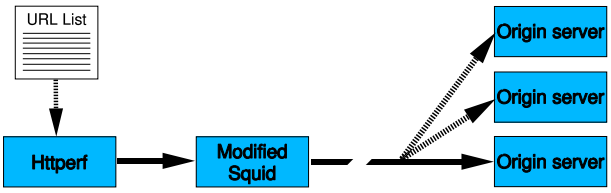


Figure 11.9: Experiment structure. httperf is used to send requests to our modified Squid and measure response times.

have found robot activity to have a significant impact.

11.4.2 Miss timing replication

In this section we test the ability of our Collector, a modified Squid proxy, to hide the use of its cache. To claim success, we must argue that proxies connecting to our Collector would be unlikely to tell whether the it utilizes a cache. To provide evidence supporting this argument, we experimentally measure response times of requests for data on real origin servers. For each run of our experiments, we iterate through a list of URLs (the IRCache workload) and give one at a time to httperf to send the request through our modified proxy (as shown in Figure 11.9.

We repeat the run for each of the four combinations of: original Squid versus modified Squid, and Squid with empty cache (i.e., generating all cache misses) versus Squid with cached results of previous run (making cache hits possible). Each run was repeated once per hour.

Results

After running the tests hourly for most of a day, we calculated relevant statistical properties using `strat` [MB01]. Examining first the IRCache data, we show the hit and miss response time distributions in Figure 11.10. Our primary concern, however, is in the paired differences in response times. That is, what is the typical difference in response time for a miss versus a replicated miss (i.e., a hit)? Since we are not concerned with whether the difference is positive or negative, we plot the distribution of the absolute value of the differences in Figure 11.11.

In Table 11.1 we show various summary statistics for the two datasets. Tests on

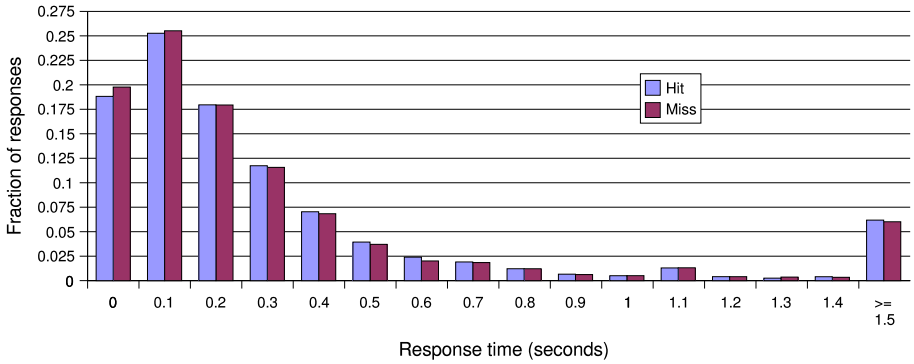


Figure 11.10: Distribution of cache hit and miss response times with NLANR IRCache data using our Collector.

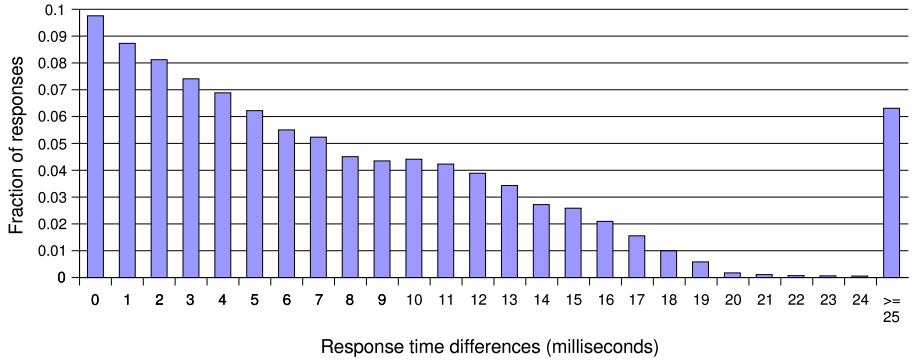


Figure 11.11: Distribution of absolute differences in paired response times between hits and misses with NLANR IRCache data using our Collector.

each dataset were repeated for some number of runs, and from each run we calculated the mean and median difference between response times of a cache miss versus a cache hit, and a cache miss versus a cache miss one hour later. In each row, we show the mean and standard error for the run means and the run medians. All values are in milliseconds.

Data Set	# runs	Cache Miss vs. Hit (ms) Mean ± StdErr of run means	Cache Miss vs. Hit (ms) Mean ± StdErr of run medians	Cache Miss vs. Miss (ms) Mean ± StdErr of run means	Cache Miss vs. Miss (ms) Mean ± StdErr of run medians
Orig. IRCache	13	12.85 ± 4.41	5.440 ± 0.08	-18.03 ± 12.96	0.08 ± 0.16
Abs. Val. IRCache	13	54.20 ± 3.90	6.45 ± 0.07	347.95 ± 10.09	10.53 ± 0.04
Rev. Abs. IRCache	13	7.44 ± 0.82	6.34 ± 0.07	337.84 ± 11.23	10.31 ± 0.46
Popular Queries	17	366.12 ± 15.27	13.01 ± 0.19	670.71 ± 22.92	33.34 ± 1.39

Table 11.1: Mean and standard error of run means and medians using our Collector.

Data Set	# runs	Cache Miss vs. Hit (ms) Mean ± StdErr of run means	Cache Miss vs. Hit (ms) Mean ± StdErr of run medians	Cache Miss vs. Miss (ms) Mean ± StdErr of run means	Cache Miss vs. Miss (ms) Mean ± StdErr of run medians
Abs. Val. IRCache	17	503.3 ± 7.5	245.6 ± 1.0	279.6 ± 8.3	10.3 ± 0.5
Popular Queries	17	600.2 ± 17.6	109.1 ± 2.5	578.1 ± 17.2	40.2 ± 1.6

Table 11.2: Mean and standard error of run means and medians using unmodified Squid.

For reference, the first row of Table 11.1 shows the results when calculating the real (not absolute value) difference between response times. The remaining rows calculate results using the absolute value of the differences. The data for the second row is otherwise identical, but now shows a significant difference in response times between the Miss-Hit and Miss-Miss cases. As expected, the typical difference for the Miss-Hit case is quite small (since our intention was to get this close to zero).

Recall from Section 11.4.1 that the IRCache data set was selected specifically to be cacheable. In fact, when we retrieved the URLs, in some instances the responses were not cacheable. If we remove these points, we get even better results, shown in the third row.

We now examine the data set generated from popular search engine queries. Unlike the IRCache data, this dataset contains many references (approximately two-thirds) to uncacheable objects. Under these conditions, we find the absolute difference in response times between miss and hit pairs of the same requests are much closer to the miss pairs separated by one hour, as shown in the last row.

For comparison, one might also ask what the typical difference is when an unmodified Squid is used (i.e., when hits are served as fast as possible). Results from testing on both datasets are recorded in Table 11.2. In both cases, mean differences in Miss-Hit response times are quite high (similar to Miss-Miss), and the medians are in fact much higher. This confirms the fact that unmodified hits are served much faster than misses, generating as much if not more variation in response times than misses an hour apart.

The above tests compared hit performance with that of a miss. It is also important to be able to generate multiple hits to the same object with consistent response times. The modified Squid is indeed able to achieve very similar response times for repeated

Connection type	mean	median
dummy direct	1057ms	526ms
dummy proxy	1108ms	650ms

Table 11.3: Comparison of response times between direct and proxy connections.

hits. The means of the run means and medians of the absolute difference between two hits for the same object is 6.60 ± 0.64ms and 2.03 ± 0.29, respectively.

From the above analysis, we can state that the typical difference in response times is significantly reduced for subsequent requests for the same object when our modified Squid Collector is utilized. In fact, when only tested on fully cacheable data, we find a useful mean miss-hit difference of just 7.4ms, and mean hit-hit difference of 6.6ms.

11.4.3 Proxy penalty

In this section we are concerned with the internal architecture of our SPE implementation. We want to know whether the implementation imposes extra costs for the proxies being tested, as compared to the direct connection between Multiplier and Collector. The experiment we chose is to force both the direct connection and tested proxy to go through identical processes. Since both proxies are running identical software on essentially identical machines, we can determine the difference in how the architecture handles the first (normally direct) connection and tested connections. We used the IRCache dataset and configured httperf to establish 1000 connections, with up to two requests per connection. Intervals between connection creation was randomly selected from a Poisson distribution with a mean delay of one second.

In this experiment we found that the direct connection was reported to be 50-125ms faster on average (shown in Table 11.3. From this result (of an admittedly small test), we might consider subtracting a factor of 50-100ms from the measured times of tested proxies, if we want to try to calculate exact timings. However, our primary concern is for comparative timings, which we address in Section 11.4.5.

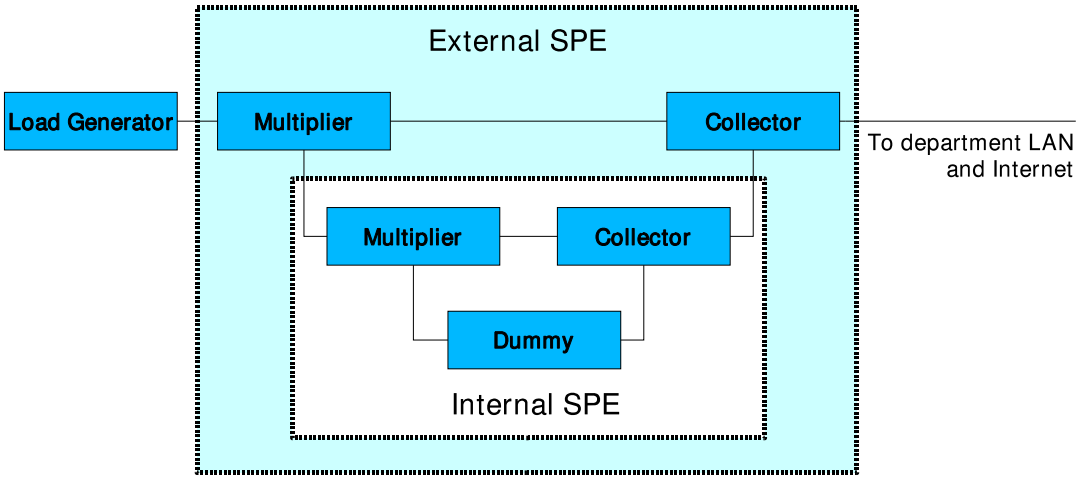


Figure 11.12: Configuration to test one SPE implementation within another.

11.4.4 Implementation overhead

In this experiment we are concerned with the general overhead that a SPE implementation introduces to the request/response data stream. We want to know how much worse performance will be for the users behind the implementation, given that each request will have to go through at least two additional processes (the Multiplier and the Collector).

The idea is to test a SPE implementation using another SPE implementation, as shown in Figure 11.12. We again used the same IRCache artificial workload driven by httperf as input to the external SPE. The inner SPE (being measured) had just one Dummy proxy to drive. The outer one measured just the inner one.

The results of our test are presented in Table 11.4. We found that the inner SPE implementation generated a mean response time of 836ms, as compared to the direct connection response time of 721ms. Medians were smaller, at 444ms and 320ms, respectively. Thus, we estimate the overhead (for this configuration) to be approximately 120ms.

Connection type	mean	median
direct	721ms	320ms
inner SPE	836ms	444ms

Table 11.4: Evaluation of SPE implementation overhead.

Connection type	mean	median
Apache proxy 1	1039ms	703ms
Apache proxy 2	1040ms	701ms
Apache proxy 3	1040ms	701ms
Apache proxy 4	1038ms	703ms

Table 11.5: Performance comparison of identical proxies.

11.4.5 Proxy ordering effects

As described earlier, the Multiplier in our SPE implementation waits until it gets a response from the Collector, and then proceeds to make the same request to the tested proxies, in the order as defined by the configuration. In reality, our implementation is more careful. On a new request that has just received the response from the Collector, the Multiplier will sequentially go through the list of tested proxies. If a persistent connection to that proxy has already been established in this process, the Multiplier uses it and sends the request, and moves on to the next proxy. Otherwise, the Multiplier issues a non-blocking connect to the proxy, and moves on. When the sequence is complete, the Multiplier waits for activity on each connection and responds appropriately (i.e., receiving and checking the next buffer's-worth of data, or sending the request if a new connection has just been established).

In this test we are concerned about the potential effects of the strict ordering of tested proxies. We want to make certain that there is no advantage to any particular position for a proxy being tested. Again we generated an artificial workload using httperf and the IRCache data to a set of identically configured Apache proxy caches.

In this experiment we found that all four proxies had very similar performance, as shown in Table 11.5. Mean response times were within two milliseconds, as were the median response times. Thus, we find that the results of identical systems with identical software are correctly reported to have essentially identical performance results.

11.5 Experiments

For the experiments using our SPE implementation to test proxies, we employed the same configurations as described in Section 11.4.1, except where noted.

Proxy name	Bytes sent	Bytes received	Responses delivered	Mean time for response	Median time for response
(direct)	100.00%	100.00%	100.00%	529ms	285ms
WcolD	99.38%	58.18%	99.78%	360ms	61ms
Oops	98.95%	66.02%	99.73%	401ms	180ms
DeleGate	98.75%	58.97%	99.40%	1045ms	1039ms
Squid	98.96%	58.87%	99.73%	360ms	106ms
Apache	98.99%	59.02%	99.73%	374ms	199ms

Table 11.6: Artificial workload results.

11.5.1 Proxies driven by synthetic workload

Here we wish to test the performance of various proxies using the httperf workload generator. For this experiment we tested Squid, DeleGate, Apache, WcolD, and an Oops proxy.

As in the previous validation tests, we use httperf to generate an artificial workload using the IRCache dataset. Since this dataset does not represent any real sequences of requests, we have disabled WcolD's prefetching mechanisms.

The results of this experiment are shown in Table 11.6 and graphically displayed in Figure 11.13. The timings shown are unadjusted results. **Bytes sent** measures the bytes delivered by the proxy to the Multiplier, relative to those sent by the Collector. Similarly, **Bytes received** measures the bytes received by the proxy from the Collector, relative to the bytes received by the Multiplier from the Collector which has no cache. Thus, a smaller fraction represents a higher byte caching ratio.

WcolD performs best under all measures. It ties with Squid for best mean response time at 360ms, but easily bests Squid's median of 106ms with a median of just 61ms. We record similar results for Apache's caching ability, but its mean response time is slightly higher at 374ms, and median response time is much higher at 199ms. While Oops provides the worst caching ability of the group, and a slightly higher mean response time at 401ms, its median response time is slightly better than Apache at 180ms. Finally, DeleGate is seen to provide similar bandwidth reductions, but has strikingly higher mean and median response times at just over one second.

DeleGate also shows a significant discontinuity in the cumulative response time

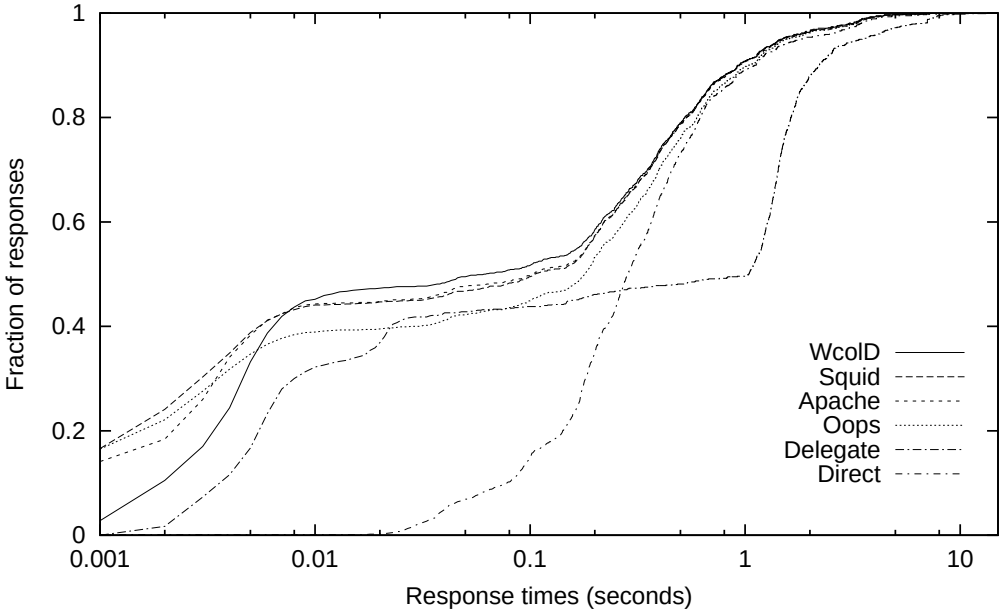


Figure 11.13: The cumulative distribution of response times for each proxy under the artificial workload.

distribution around 1 second in Figure 11.13. Intuition suggested that some kind of timeout was likely involved, which we were able to confirm upon examination of DeleGate's source code where we found the use of `poll(2)` with a timeout of 1000ms.

11.5.2 A reverse proxy workload

We also wish to be able to test content-based prefetching proxies. To avoid the need to interfere with an existing user base and systems, we decided to replay a Web server trace. Since fetching real content can have side effects [Dav01a], we specifically chose to use a log from a Web server that predominantly generated static content.

The simclient tool was used to generate the workload, by replaying the requests in the Reverse proxy dataset. This allows us to preserve the inter-request delays (e.g., including user "thinking times") that were originally present, and matching more closely what a proxy would see in a real-world situation. In order to increase the load presented to the tested proxies, we will use three simclients simultaneously, each replaying requests from the first fifteen hours of a different day. We tested all five proxies, but in this case

permitted WcolD to prefetch embedded resources, but not links to new pages (since those pages could be on servers other than the one represented by this log).

Note that even though the requests in the workload included many HTTP/1.1 requests, the Proxycizer simclient only generates HTTP/1.0 requests. Additionally, unlike the original trace, the simclient does not generate IMS queries, since it does not have a cache of its own, nor does it know which of the original requests carried IMS (although we know some of them did, generating 304 responses).

Likewise, some clients (such as Adobe Acrobat [Ado02]) generate HTTP/1.1 ranged requests which elicit HTTP/1.1 206 partial content responses. We are unable to model this well with the Proxycizer simclient, which only generates HTTP/1.0 requests, and so it instead generates multiple full requests when a real HTTP/1.0 client would have only made one such request (since the whole response would have been returned on the first request). Of the original three day log, 2.3% of the responses were for partial content (HTTP response code 206).

The tests we have performed are relatively short and contain a small working set. As a result, the proxies do not significantly exercise their storage infrastructure or replacement policies. Thus, additional experiments are necessary before making significant conclusions about the proxies tested.

Instead, these tests have demonstrated some of the abilities of our SPE implementation to measure the simultaneous performance of implemented proxies. Unfortunately, because of limitations in the types of workloads used, the sample experiments reported here were unable to demonstrate improved performance from the use of prefetching. Note that we replayed the request log of a LAN-attached Web server, and thus requests and responses did not have to traverse the Internet. With the higher response times typical of the real Internet, caching and prefetching may be able to show greater effects. As experience with and confidence in our SPE implementation grows, we anticipate larger-scale tests with a live user load, rather than replaying a trace, with content retrieved from remote servers.

Results from this experiment can be found in Table 11.7 and are graphically displayed in Figure 11.14. Since this workload was originally generated by a Web server

Proxy name	Bytes sent	Bytes received	Resp. delivered	% resp. match	% not match	Mean lat.	Median lat.
(direct)	100.00%	100.00%	100.00%	100.00%	0.00%	433ms	235ms
WcolD	94.73%	25.78%	99.61%	99.91%	0.09%	140ms	9ms
Oops	89.74%	30.01%	99.52%	99.76%	0.24%	151ms	9ms
DeleGate	95.47%	15.78%	99.75%	100.00%	0.00%	292ms	16ms
Squid	89.79%	14.67%	99.61%	100.00%	0.00%	92ms	7ms
Apache	89.44%	15.43%	99.51%	100.00%	0.00%	73ms	7ms

Table 11.7: Reverse proxy workload results.

with relatively few objects that were statically generated, it is highly cacheable, and so all of the caches perform significantly better than direct retrieval. Thus we see a significant benefit in placing any of these proxies in front of the Web server.

Squid does very well in this test, using the least external bandwidth, matching all responses to that sent by the Collector, and achieving the best median response time of 7ms, and an excellent mean response time of 92ms. However, Apache matches the median response time, and shortens Squid's mean response time by 19ms. This occurs in spite of higher external bandwidth usage. Examination of the experiment logs show

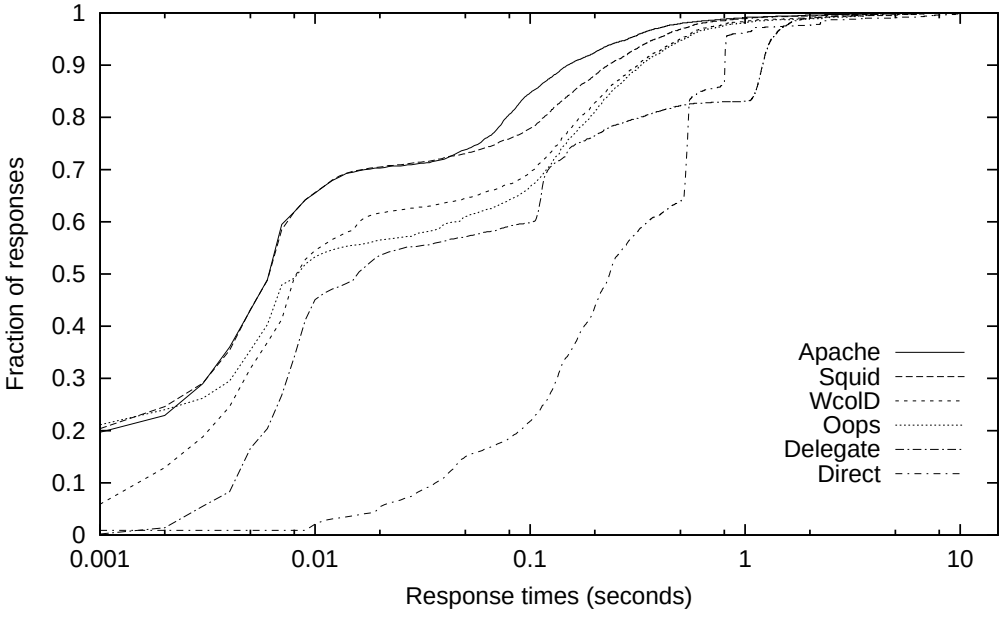


Figure 11.14: The cumulative distribution of response times for each proxy under the full-content reverse proxy workload.

an explanation. For requests with a URL containing a tilde (~), Apache apparently encodes the symbol as `%7E` when passing it on to the Collector. While equivalent from a Web server's point of view, Squid (and thus our Collector) does not recognize the equivalence of the two, and serves it as a real cache miss. Since this cache miss typically follows shortly after completion of the direct request, the response time from the Web server for this object is often better (since some or all of the relevant data may still be in the Web server's memory). The effect can also be seen in Figure 11.14 where Apache and Squid perform almost equivalently for more than 70% of the responses. The slower responses correspond to proxy cache misses, and Apache is serving misses more quickly than Squid.

WcolD and Oops come close with median response times of 9ms, but mean response times are worse (140ms and 151ms, respectively) and not all responses end up matching those sent by the Collector (.09% and .24%, respectively). They also use significantly more bandwidth than any of the others. DeleGate, while caching a similar number of bytes as Apache, has a higher median response time of 16ms and the worst mean response time of 292ms.

11.6 Summary

This chapter has described our implementation of SPE, and many of the issues related to it. We have described validation tests that we have performed on our implementation, as well as demonstrated the use of the current system in two experiments to simultaneously evaluate the performance of five publicly available proxy caches.

The primary contribution of this chapter has been the implementation, description, and evaluation of a SPE implementation. We have shown that the Collector can return cached results as if they were not cached. We have shown that the Multiplier measures performance of equivalent proxies equivalently, and that the overhead of running our system is relatively small. While we believe that even tighter results may be possible (particularly when the implementation includes operating system modifications), this chapter has demonstrated a credible complete SPE implementation.

Chapter 12

Prefetching with HTTP

12.1 Introduction

Previous chapters have examined possible mechanisms to predict user actions on the Web for prefetching, and to evaluate proposed algorithms and implemented prefetching systems. However, even when we find mechanisms that work, challenges remain to get them fielded, notably limitations of the HTTP mechanism itself.

This chapter will provide some answers to the question of why prefetching is not presently widespread as well as why it should not be deployed within the current infrastructure, and what changes are needed to make prefetching possible on the commercial World-Wide Web.

Prefetching into a local cache is a well-known technique for reducing latency in distributed systems. However, prefetching is not widespread in the Web today. There are a number of possible explanations for this, including the following perceived problems:

- the bandwidth overhead of prefetching data that is never utilized,
- the potential increase in queue lengths and delays from naive prefetching approaches [CB98], and
- the effect on the Internet (e.g., added traffic and server loads) if all clients were to implement prefetching.

Fortunately, the first two objections can be satisfied by more careful approaches — prefetchers can be limited to high-confidence predictions, which can limit the bandwidth overhead, and prefetching systems can be used to moderate network loads when requests are spaced appropriately. The third is still up for debate, but when prefetching occurs

over a private connection for objects stored on an intermediate proxy (as in proxy initiated prefetching [FJCL99]), this is no longer an issue.

We will first define prefetchability and then review how Web prefetching has been suggested by many researchers and implemented in a number of commercial systems. In section 12.3 we will discuss a number of less-well-investigated problems with prefetching, and with the use of GET for prefetching in particular. In effect we argue that 1) GET is unsafe because in practice, GET requests may have (not insignificant) side-effects, and 2) GET can unfairly use server resources. As a result, we believe that prefetching with GET under the current incarnation of HTTP is not appropriate. We then propose extending HTTP/1.1 [FGM+99] to incorporate methods and headers that will support prefetching and variable Web server quality of service.

12.2 Background

While some researchers have considered prepushing or presending, this chapter is only concerned with client-activated preloading of content. We are less concerned here about how the client chooses what to prefetch (e.g., Markov model of client history, bookmarks on startup, user-specified pages, the use of proxy or server hints), with the understanding that a hints-based model (as suggested in [PM96, Mog96, Duc99] and others) has the potential to avoid some of the concerns discussed here but may also introduce other problems and likewise lacks specification as a standard.

Therefore, in this chapter we make the not-unreasonable assumption that a prefetching system (at the client or intermediate proxy) has a list of one or more URLs that it desires to prefetch.

12.2.1 Prefetchability

Can prefetchability be defined? Let us start with prefetching. A relatively early paper [DP96] defines prefetching as a cache that "periodically refreshes cached pages so that they will be less stale when requested by the user." While this definition does have a Web-specific perspective, it does not reflect current usage of the term since it is specific

to cache-refreshment. The more general definition from Chapter 1) is:

> *Prefetching* is the (cache-initiated) speculative retrieval of a resource into a cache in the anticipation that it can be served from cache in the future. (2)

Typically the resource has not yet been requested by the client (whether user or machine), but the definition applies equally to cache refreshes (when the content has changed). With this definition we can see that:

- Prefetching requires the use of a cache.
- Prefetchability requires cacheability.

Note that not all prefetched resources will be used before they expire or are removed from the cache. Some will never be requested by the client. This implies that:

- Prefetching requires at least idempotence, since resources may end up being fetched multiple times.
- Speculative prefetchability requires safety. Unless we can guarantee that the client will actually use this resource (before removal or expiration), a prefetching system should not perform any action that has unwanted (from either the client or server's perspective) side-effects at the server.

Therefore, now we have a definition for prefetchable:

> A Web resource is *prefetchable* if and only if it is cacheable and its retrieval is safe. (3)

Typical concerns for safety (e.g., those raised in RFC 2310 [Hol98]) are really concerns for idempotency, but idempotency is insufficient for prefetching, as the prefetched response might never be seen or used. Here we are referring to the definition of safe as having no significant side-effects (see section 9.1.1 of the HTTP/1.1 [FGM+99]).

12.2.2 Prefetching in the Web today

Prefetching for the Web is not a new concept. Discussions of mechanisms and their merits date back at least to 1994-1995.[1] Many papers have been published on the use of prefetching as a mechanism to improve the latencies provided by caching systems (e.g., [PM96, Mog96, WC96, CJ97, CY97, JK98, MC98, Pal98, SKS98, AZN99, CKR99, Kle99, Duc99, FJCL99, PP99b, SR00]).

A number of commercial systems used today implement some form of prefetching. CacheFlow [Cac02a] implements prefetching for inline resources like images, and can proactively check for expired objects. There are also a number of browser extensions for Netscape and Microsoft Internet Explorer as well as some personal proxies that perform prefetching of links of the current page (and sometimes bookmarks, popular sites, etc.). Examples include NetSonic [Web02], PeakJet 2000 [Pea02], and Webcelerator [eAc01b]. In addition, older versions of Netscape Navigator have some prefetching code present, but disabled [Net99].

12.3 Problems with Prefetching

Even with a proliferation of prefetching papers in the research community, and examples of its use in commercial products, there are a number of well-known difficulties with prefetching as introduced in the beginning of this section.

Some researchers even suggest pre-executing the steps leading up to retrieval [CK00] as one way to avoid some of the difficulties of content retrieval and still achieve significant response time savings. However, as suggested above, researchers and developers are still interested in the performance enhancements possible with speculative data transfer. When prefetching (and not prepushing/presending), the standard approach for system builders is to use the HTTP GET mechanism. Advantages of GET and standard HTTP request headers include:

- GET is well-understood and universally used.

[1] For one interesting discussion still accessible, see the threads "two ideas..." and "prefetching attribute" from the www-talk mailing list in Nov-Dec 1995 at `http://lists.w3.org/Archives/Public/www-talk/1995NovDec/thread.html`.

- There are supporting libraries to ease implementation (e.g., for C there is W3C's Libwww [Wor01] and for Perl there is Libwww-perl [Aas02]).
- It requires no changes on the server's side.

However, the use of the current version of GET has some obvious and non-obvious drawbacks, which we discuss in this section. Even from the prefetching client's perspective, there may be some drawbacks. In particular, just like users, a naive prefetcher may not know that the object requested is very large and will take minutes or hours to arrive.

In many cases, however, these drawbacks can be characterized as server abuses, because the inefficiencies and deleterious effects described could be avoided if the content provider were able to decide what and when to prefetch. In the rest of this section we detail the problems with prefetching, with emphasis on the consequences of using the standard HTTP GET mechanism in the current Web.

12.3.1 Unknown cacheability

One of the differences between Web caching and traditional caching in file and memory systems is that the cacheability of Web objects varies. One cannot look at the URL of an object and *a priori* know that the response generated for that URL will be cacheable or uncacheable. Some simple heuristics are known, including examining the URL for substrings (e.g., "?" and "cgi-bin") that are commonly associated with fast-changing or individualized content, for which caching does not provide much use. However, while a URL with an embedded "?" has special meaning for HTTP/1.0 caching (see the HTTP/1.1 specification [FGM$^+$99], section 13.9), a significant fraction of content that might be labeled dynamic (e.g., e-commerce site queries) does not change from day to day [WM00] and should be treated as cacheable unless marked otherwise (as stated by the HTTP/1.1 specification). Only the content provider will know for certain, which is perhaps why the cacheability of a response is determined by one or more HTTP headers that travel with the response.

Some researchers (e.g., [Duc99]) have proposed prefetching and caching data that is otherwise uncacheable, on the grounds that it will be viewed once and then purged

from the cache. However, the HTTP specification [FGM+99] states that such non-transparent operations require an explicit warning to the end user, which may be undesirable. Since many objects are marked uncacheable just for hit metering purposes, Mogul and Leach suggest something similar in their "Simple Hit-Metering and Usage-Limiting for HTTP" RFC [ML97] which states in part (*emphasis added*):

> Note that the limit on "uses" set by the max-uses directive does not include the use of the response to satisfy the end-client request that caused the proxy's request to the server. *This counting rule supports the notion of a cache-initiated prefetch: a cache may issue a prefetch request, receive a max-uses=0 response, store that response, and then return that response (without revalidation) when a client makes an actual request for the resource.* However, each such response may be used at most once in this way, so the origin server maintains precise control over the number of actual uses.

Under this arrangement, the response no longer has to be marked uncacheable for the server to keep track of views and can even direct the number of times that the object may be served.

In summary, though, when a client or proxy prefetches an object that it is not permitted to cache, it has wasted both client and server resources to no avail.

12.3.2 Server overhead

Even when the object fetched is cacheable, the server may have non-negligible costs in order to provide it. The retrieval will consume some portion of the server's resources including bandwidth and cpu, and logging of unviewed requests. Likewise, the execution of some content retrievals may not be desirable from a content-provider's point of view, such as the instigation of a large database search. In fact, the use of overly aggressive prefetchers has been banned at some sites [Pel96].

As more attention is paid to end-user quality of service for the Web (e.g., [BBK00]), the content provider may want to be able to serve prefetched items at a lower quality

of service [Pad95, PM96] so that those requests do not adversely impact the demand-fetched requests.

12.3.3 Side effects of retrieval

According to the HTTP/1.1 specification [FGM+99], GET and HEAD requests are not supposed to have a significance other than retrieval and should be considered safe. In fact, the GET and HEAD methods are supposed to be idempotent methods. In reality, many requests do have side-effects (and are thus not idempotent).

Often these side effects are the reason for some objects to be made uncacheable (that is, the side effects are desirable by the content producers, such as for logging usage). More importantly, some requests, when executed speculatively, may generate undesirable effects for either the content owner, or the intended content recipient. Placing an item into an electronic shopping basket is an obvious example — a prefetching system, when speculatively retrieving links, does not want to end up placing the items corresponding to those links into a shopping cart without the knowledge and consent of the user.

This reality is confirmed in the help pages of some prefetching products. For example, one vendor [eAc01a] states:

> On sites where clicking a link does more than just fetch a page (such as on-line shopping), Webcelerator's Prefetching may trigger some unexpected actions. If you are having problems with interactive sites (such as e-mail sites, online ordering, chat rooms, and sites with complex forms) you may wish to disable Prefetching.

Idempotency is a significant issue and is also mentioned in other RFCs (see for example [BLMM94, BLC95, BLFM98]). It is additionally an issue for pipelined connections in HTTP/1.1 (prefetching or otherwise) as non-idempotent requests should not be retried if a pipelined connection were to fail before completion.[2] The experimental RFC

[2]HTTP idempotency continues to be significant. For a discussion of HTTP idempotency with emphasis on idempotent sequences, see the discussion thread "On pipelining" from the IETF

2310 [Hol98] proposes the use of a new response header called Safe which would tell the client that the request may be repeated without harm, but this is only useful after the client already has header information about the resource. As suggested by Chapter 10, the lack of guaranteed safe prefetching also complicates live proxy evaluation.

The HTTP/1.1 specification [FGM+99] does note that "it is not possible to ensure that the server does not generate side-effects as a result of performing a GET request." A conscientious content developer would not only need to use POST to access all potentially unsafe resources but to prevent GET access to those resources, as linking to content cannot be controlled on the WWW. There may also be some confusion as to the interpretation of the idempotence of GET as a MUST rather than a SHOULD, and what constitutes safety and its significance (as a SHOULD requirement). In any case, enough content providers have built websites (perhaps without thought to prefetching) that do generate side-effects from GET requests that careful developers are unlikely to develop widespread prefetching functionality, for fear of causing problems.

12.3.4 Related non-prefetching applications

There are other situations in which server resources are potentially abused. These include price-comparison systems that repeatedly retrieve pages from e-commerce sites to get the latest product pricing and availability, and extensive mechanical crawling of a site at too fast a rate [Luh01]. Both of these can cause undue loads at a Web site and affect performance of the demand-fetched responses. While the need for an improved interface between crawlers and servers has been recognized (e.g., [BCGMS00]), the need for distinguishing between demand– and non-demand-requests has not been suggested. Interestingly, there has been work [RGM00] to build crawlers that attempt to access the "hidden Web" — those pages behind search forms [Ber00], further blurring the semantic distinction between GET and POST methods.

A server may also benefit from special handling of other non-demand requests, such as non-demand-based cache refreshment traffic [CK01b], in which additional validation

---

HTTP working group mailing list archives at `http://www.ics.uci.edu/pub/ietf/http/hypermail/2000/thread.html`.

requests are made to ensure freshness of cached content. From the content provider's perspective, it would be helpful in general to serve non-demand requests of any kind at a different quality of service than those requests that come from a client with a human waiting for the result in real-time.

12.3.5 User activity conflation

Non-interactive retrieval also has the potential for generating anomalous server statistics, in the sense that a page retrieval is assumed to be represent a page view. Almost certainly some fraction of prefetched requests will not be displayed to the user, and so the logs generated by the server may overestimate and likely skew pageview statistics. This is also the case for other kinds of non-interactive retrievals, such as those generated by Web crawlers. Today, when analyzing server logs for interesting patterns of usage, researchers must first separate (if possible) the retrievals from automated crawlers [KR98, Dav99c]. As Pitkow and Pirolli have noted [PP99a]:

> "... There is no standardized manner to determine if requests are made by autonomous agents (*e.g.*, robots), semi-autonomous agents acting on behalf of users (*e.g.*, copying a set of pages for off-line reading), or humans following hyperlinks in real time. Clearly it is important to be able to identify these classes of requests to construct accurate models of surfing behaviors."

Thus, if the server were maintaining a user model based on history, undistinguished prefetching (or other non-demand) requests could influence that model, which could generate incorrect hints that would cause additional requests for useless resources which would likewise (incorrectly) influence the server's model of the user, and so on. A server that is able to distinguish such requests from demand traffic would also be able to distinguish them in logs and in user model generation. Therefore, a mechanism is needed to allow a server to distinguish demand requests from non-demand requests, as describe below.

12.4 Proposed Extension to HTTP

The preceding section has discussed the problems with prefetching on the Web using current methods. In this section, we describe the need for an extension (as opposed to just new headers) for HTTP, mention briefly some previous proposals for HTTP prefetching extensions, and then provide a starting point for recommended changes to the protocol.

12.4.1 Need for an extension

HTTP is a generic protocol that can be extended through new request methods, error codes, and headers. An extension is needed to allow the server to define the conditions under which prefetching (or any other non-demand request activity) is allowed or served.

The alternative to an extension is more appropriate use of the current protocol. One approach, say, to eliminate some wasted effort would be to use range-requests (much like Adobe's Acrobat Reader [Ado02] does) to first request a portion of the document. If the document was small, then there is a chance the whole object will be retrieved. In any case, the headers would be available. A slight variation would be to use HEAD to get meta-information about the resource. That would allow the client to use some characteristics of the response (such as size or cacheability) to determine whether the retrieval is useful. These suggestions do not address the problems of request side-effects and high back-end overhead which would not be visible from headers, and in fact may be exacerbated by HEAD requests (e.g., if the server has to generate a dynamic response to calculate values such as Content-length). Neither do they allow for a variable quality of service at the Web server.

A second alternative would be to classify systems performing prefetching activity as robots, and require compliance with the so-called "robot exclusion standard" [Kos94]. However, conformity with this non-standard is voluntary, and is ignored by many robots. Further, compliance does not ensure a prevention of server abuse, as there is no restriction on request rates, allowing robots to overload a server at whim.

A third approach would be to incorporate a mechanism to support mandatory extensions to HTTP without protocol revisions. Such a mechanism has been proposed in Experimental RFC 2774 [NLL00], which allows for the definition of mandatory headers. It modifies existing methods by prefixing them with "M-" to signify that a response to such a request must fulfill the requirements of the mandatory headers, or report an error. If RFC 2774 were widely supported, this proposal might only need to suggest the definition of new mandatory headers that allow for prefetching and other non-interactive requests to be identified.

12.4.2 Previous proposals

Other researchers have proposed extensions to HTTP to support prefetching in one form or another.

- In his masters thesis, Lee [Lee96] prepared a draft proposal for additional HTTP/1.1 headers and methods. PREDICT-GET is the same as GET, but allows for separate statistical logging (so that logs are not skewed with prefetching requests). PREDICT-GOT is a method equivalent to HEAD, but is to be used as a way for the client to tell the server that the prefetched item was eventually requested by the client. Lee also lists new headers that would specify what kinds of predictions are desired by the client, and the kinds of predictions returned by the server.

- Padmanabhan and Mogul [PM96] suggest adding a new response header to carry predictions (termed Prediction) that would provide a list of URLs, their probabilities, and possibly their size. Elsewhere [Mog96], Mogul provides examples of other possible response headers that would be useful, including ones to describe mean interarrival times.

- Duchamp [Duc99] defines a new HTTP header (termed Prefetch) along with a number of directives. The new header is included in both requests and responses to pass information on resource usage by the client and suggested resources for prefetching by the server. The actual retrieval is performed using GET.

Unlike these proposals, we would like to see changes that are independent of a particular prediction technique.

12.4.3 Recommending changes

Any realistic set of changes will need to be agreed upon by a large part of the Web community, and so it makes sense to develop a proposal for these changes in consultation with many members of that community. However, there are certain aspects to the changes that appear to be sensible with respect to the type of prefetching described in this document.

We argue that a new method is needed. It should be treated similarly to the optional methods in HTTP/1.1. The capability of the server could be tested by attempting to use the new method, or by using the OPTIONS method to list supported methods. A new method provides safety — if a client uses a new method, older servers that do not yet support prefetching would return the 501 Not Implemented status code, rather than automatically serving the request, which would happen with the use of a new header accompanying a GET request. If the server does support these prefetching extensions, it would have the option to serve the request as if it were a normal GET, or at a lower-priority, or to deny the request with a 405 Method Not Allowed status code. When a client receives some kind of error code using the new method, it may choose to re-try the request with a traditional GET, but with the understanding that such action may have undesirable side-effects.

Therefore, we suggest a second version of GET (arbitrarily called `GET2`). A new header (possibly called `GET2-Constraint`) may also be useful to allow the client to specify constraints (e.g., `safe` to guarantee safety of the processing of this request, `max-size` to prevent fetching even a portion of an object that is too large, `min-fresh` to prevent fetching an object that will expire too quickly, `max-resp-time` to not bother if the response will take too long, etc.). Note that some of the `Cache-control` header values may be relevant here (like `min-fresh`), even though `GET2` requests may be made to servers as well as caches. Another header might be useful to allow client-type identification (e.g., interactive for browsers and other clients needing immediate response,

non-interactive for prefetching systems that will eventually send this content to users, robot for non-interactive systems that will parse and use content directly, etc.), but it may also be sufficient to assume that `GET2` requests will only be generated by non-interactive systems.

In summary, we recommend the development of a protocol extension that, when implemented by non-interactive clients that would otherwise use GET, allows for absolutely safe operation (no side-effects). When implemented by servers, such a protocol gives them the ability to distinguish between interactive and non-interactive clients and thus potentially provide different levels of service.

12.5 Summary

The benefits of Web cache prefetching are well understood. Prefetching could be a standard feature of today's browsers and proxy caches. This chapter has argued that the current support for prefetching in HTTP/1.1 is insufficient. Existing implementations can cause problems with undesirable side-effects and server abuse, and the potential for these problems may thwart additional development. We have made some initial suggestions of extensions to HTTP that would allow for safe prefetching, reduced server abuse, and differentiated Web server quality of service.

Overall, the conclusions we reach about the current state of affairs are the following:

- Prefetching with GET can be applied safely and efficiently between client and proxy under HTTP/1.1 using the `only-if-cached` and `min-fresh` options to the `Cache-control` header. Proxy resources can be abused, however, by a client that prefetches content that will never be used, and prefetching effectiveness is limited to the contents of the cache.

- Neither the safety nor efficiency of prefetching from an origin server can be guaranteed, as the origin server cannot recognize non-demand requests and so must serve every request, including requests for content that is uncacheable or has side effects.

- Proxies and origin servers cannot provide different qualities of service to demand and non-demand requests, again because they cannot distinguish between demand requests and non-demand requests (generated by prefetchers and Web robots).
- Much of the uncacheable content of the Web can be prefetched by clients in the sense that the content can be cached if semantic transparency is allowed to be compromised (although user warnings are still required under HTTP/1.1).

An extension to HTTP could rectify the second and third problems and allow for safe, semantically transparent prefetching for those systems which support it, and "gracefully" degrade to the existing situation if necessary when the extension is not supported by either the client or the server. It is our hope that this chapter will restart a dialog on these issues that will move in time into a standards development process.

Chapter 13

Looking Ahead

13.1 Introduction

To conclude this work, we first ask what have we learned over the preceding chapters. We summarize the key contributions below. We then consider the next steps — how the major systems in this thesis can be extended in the future, and what is needed to make prefetching commonplace in the Web. Finally, we will offer some thoughts about the future of the broader task of improving the performance of content and service delivery.

13.1.1 User action prediction

In the first part of the dissertation, we focused on the task of predicting the next action that a user will take. *We introduced a simple approach called IPAM for predicting the UNIX shell commands issued by a user, and realized a predictive accuracy of approximately 40% over two independent datasets.* We also specified the characteristics of an idealized prediction algorithm.

Later we considered specifically the challenges involved in predicting Web requests from history. We presented many domain-specific choice-points, as well as the details of various Markov-like prediction models. We implemented generalized prediction codes to model the various algorithms, and compared their performance on a consistent set of Web request traces (realizing predictive accuracies from 12-50% or better, depending on the dataset). *The identification, implementation, and evaluation of Markov-based predictions based on the history of actions taken by one or more users* is one of our key contributions. The best performing approaches included predictions from multiple

contexts (as in PPM – prediction by partial match).

In addition to prediction methods using history-based techniques, we also considered the complementary idea of content-based prediction. First *we experimentally validated widespread assumptions of textual similarity in nearby pages and descriptions on the Web*, and then used those assumptions to implement a content-based prediction approach and tested it on a full-content Web usage log. Another key contribution is *the evaluation of the use of content-based prediction methods* in which we found a content-based approach to be 29% better than random link selection for prediction, and 40% better than not prefetching in a system with an infinite cache.

13.1.2 Evaluation of Web cache prefetching techniques

While the initial chapters dealt with the concerns of designing and evaluating prediction techniques in isolation, later chapters concentrated on evaluating those techniques within the environment of a prefetching system. We first introduced the topics of concern to evaluators, and considered specifically the task and common techniques for proxy cache evaluation.

In order to estimate client-side response times, we implemented and *validated a new Web caching and network simulator.* In contrast to testing predictive accuracy as we did earlier, the use of the simulator allowed us to evaluate prediction mechanisms embedded within a prefetching environment. Using NCS, *we demonstrated the potential performance improvements resulting from the use of multiple prediction sources for prefetching.* We showed some scenarios in which a combination of prediction sources can provide better performance than either alone, and found that client prefetching based on Web server predictions can significantly reduce typical Web response times, without excessive bandwidth overhead. We also presented some analysis of when history-based prefetching will not be particularly effective. In general, NCS provides a useful framework to test other network and caching configurations with real-world usage traces.

Since simulation is not helpful to evaluate proxy cache implementation, *we proposed a novel architecture called Simultaneous Proxy Evaluation that can optionally utilize live traffic and real data to evaluate black-box Web proxies.* Unlike existing evaluation

methods, *SPE is able to objectively evaluate prefetching proxies* because it provides access to real content and live data for speculative retrieval. *We then implemented the SPE architecture, discussed various issues and their ramifications to our system, and demonstrated the use of SPE in validation tests and experiments using SPE to evaluate five publicly available proxy caches.*

Finally, *we identified the need to change HTTP because of fairness and safety concerns with prefetching using the current standard.*

13.2 The Next Steps

The past chapters have dealt with the mechanisms to predict user actions on the Web for prefetching, and to evaluate proposed algorithms and implemented prefetching systems.

In fact, multiple approaches to predict Web activity were proposed, implemented, and evaluated. Multiple approaches to evaluate Web prefetching algorithms were proposed and implemented. We showed that higher predictive accuracy and lower client-side response times can be accomplished by combining the results of separate prediction systems.

However, there are some aspects that were not able to be addressed by this dissertation, and so are directions for future work. Such issues include:

- Congestion modeling and packet loss in NCS. The simulator currently provides for the equivalent of infinite queues, which are unrealistic. However, modeling the network at this level would likely also require modeling routers and additional TCP protocol detail, rendering it much slower.

- Proxy evaluation utilizing SPE on a live request stream. While designed to be able to handle live user requests, the experiments using SPE presented in this dissertation used artificial and captured traces. We look forward to the opportunity to use SPE to conclusively evaluate multiple proxies on a live request stream.

- Evaluation of caching and prefetching implemented in browsers. While caching and prefetching in browsers is considered within NCS, the SPE architecture is not

directly suited to the evaluation of client-side systems. One potential alternative is to break out whatever system is being added to a browser (such as the caching and prefetching) and place it into a separate proxy that can then be tested using SPE.

- The cooperation of proxies, whether in a mesh, a hierarchy or as part of a content delivery network. Such systems of proxy caches may be able to provide significant benefits, but would not, for example, be amenable to evaluation using the SPE architecture since SPE would be unable to wrap itself around a truly distributed CDN. However, it may be possible to locate the Multiplier with the load generator, and the Collector at the origin server. If the CDN can be configured to access the Collector instead of the origin server, then SPE could indeed be applied to the constrained task of evaluating response times for requests for a particular server.

- Evaluation of larger scenarios in NCS. Presently NCS builds all of its data structures in memory, limiting possible simulation sizes to the amount of memory supported by the OS to be assigned to an application. While likely to be considerably slower, a disk-based simulator would have the ability to simulate longer and larger scenarios. Larger scenarios might enable simulation of CDN-style sets of proxy caches or test theories of what would happen "if everyone were to prefetch" with some level of aggressiveness.

- The use of a SPE implementation in which clients are not explicitly configured to access the Multiplier. Many proxy caches are deployed to intercept Web requests transparently and thus serve responses without the knowledge of the client. With additional equipment and small changes, SPE could be implemented in this fashion as well, and to be able to evaluate proxies so configured.

There are two reasons why the multi-source prefetching explored in this dissertation cannot be put on the Web today. The first is that some requests on the Web have side effects, as discussed in Chapter 12. The second is that prediction hints (sent, in general from server to client) need to be transported under some standardized mechanism. As demonstrated in this thesis and elsewhere, a Web server or

proxy that has seen many requests for an object will have a better model of what is likely to be requested next. Therefore, in a client-based prefetching system, the predictions need to be sent to the client in some form. A number of researchers have proposed the use of hints of some kind for caching, invalidation, or prefetching [Pad95, PM96, Mog96, CKR98, KW97, KW98b, Duc99]. While the hints themselves could be delivered through HTTP using a particular unique URL, the client still needs to know the identity of the appropriate URL that would contain hints of actions likely to follow a particular object retrieval. One possibility is to calculate the hints URL by the use of a predetermined URL transformation function. Such an approach might take the original URL and add a prefix or suffix, or generate an opaque identifier such as by an MD5 checksum. However, those approaches limit capability with objects that may already be served using such URLs that would result from the transformation. Another, more reasonable and commonly proposed approach is to include the hints or a pointer to them, in the headers of an HTTP response. While not presently standardized, making such a standard does not pose any significant technological hurdles. Both of these difficulties would be obviated with appropriate modifications to HTTP.

13.3 The Future of Content Delivery

In order to provide useful services to massive numbers of clients, the popular network services of today and tomorrow require large-scale yet economical solutions. For aspiring companies, it sometimes does more harm than good to be mentioned on a very high traffic site (e.g., the Slashdot effect [Adl99]). As a result, even smaller-scale services need the ability to scale to large numbers of clients. Instead of the obvious (but expensive and often complex) approach of placing Web servers or caches in distributed hosting centers, content distribution networks are typically the recommended solution.

CDNs have become well-established parts of the network infrastructure. Some are presently independent companies, such as Speedera [Spe02] and Akamai [Aka02], which operates tens of thousands of systems located in thousands of ISP and university facilities. Others operate as part of a network or Web hosting provider, such as AT&T's

services [ATT02], or Digital Island [Cab02] which is presently part of Cable & Wireless's Exodus services. Finally, some use alternative communication methods to deliver content to caches, such as Cidera [Cid02] which uses satellite delivery. CDN services have become valuable mechanisms by which content providers can ensure better Web site performance for their customers.

In effect, CDNs allow customers to share the infrastructure with many other sites and leverage the low probability of a simultaneous surge in traffic to the shared sites. However, CDNs may not be tomorrow's solution, as dynamic content and other services gain in popularity. CDNs have already begun the process of change — they are no longer limited to static Web content, but now can serve various streaming media and some dynamic content (e.g., through the use of Edge Side Includes [OA01]). Increasing support for Sun's J2EE and Microsoft's .NET will augment the ability to provide dynamic Web services from CDN edge platforms (e.g., [VM02]). Thus, the generation of some dynamic content at the edge is just part of a progression, leading to not just caching at the edge, but computation at the edge.

However, while distributing content and some computation to the edge of the well-connected Internet can significantly improve user-perceived performance, it does not directly address the last-mile problem. The user's connection to the Internet is typically the bottleneck, generating the bulk of latencies and transmission delays. To improve this aspect of Web performance, caching, prefetching, and computation must move to the client. In some ways, that is already happening — browsers already cache content and connections, and typically implement JavaScript and a Java environment for applets, thus moving some computation to the client. However, with the cooperation of the major browser manufacturers, much more could be done with prefetching (as shown in this dissertation) and with client-side computation.

Caching and computation at the edge (wherever you define it) are important parts of the solution to improve performance of some applications. Others include:

- Caching and pre-loading of non-Web traffic. DNS is naturally cached by clients

and servers, and while we have mentioned its effects, it is just one of many non-Web protocols that benefit from caching. For example, streaming media such as audio or video can be and is distributed using specialized proxy caches.

- Caching of all network traffic. A few researchers and companies have proposed mechanisms for the caching and compression of network traffic at the packet or byte level [SW98, SW00, Exp02]. Such approaches have the potential to reduce the benefit of passive caches, but prefetching will still be valuable.

- Other techniques to reduce response times. Many other research areas affect user-perceived latencies on the Web, including routing, quality of service (QoS) efforts, and improvements in last-mile networking.

In summary, researchers and developers concerned with optimizing content and service delivery will need to consider a larger view — and not just optimize one portion of the delivery process.

13.4 Summary

Prefetching has been shown (both in this thesis and elsewhere) to be an effective mechanism to reduce client-side response times in the Web. Why is this? In Chapter 2, we mentioned a number of factors supporting the need for caching research. Prefetching can successfully combat the same sources of latencies (such as network congestion, speed-of-light delays, etc.) because it takes advantage of:

- User thinking time which can be overlapped with retrieval. The idle time between requests provide resources that can be speculatively exploited.

- Multiple sources that can provide content speculatively. By retrieving from multiple sources, we can increase the utilization of otherwise idle resources, making the system more efficient.

- Repeated activity by people or systems. Both people and systems repeat sequences of actions, and such patterns of activity can be recognized and exploited by capable systems for prefetching.

Given these arguments, why is Web prefetching not prevalent? While there may be many reasons, one important aspect is the lack of standard mechanisms to recognize non-demand HTTP requests (as we described in Chapter 12) and the transport of prediction hints (as described above). As a result, Web prefetching is more likely to be found within closed systems (e.g., a CDN) in which specialized headers or mechanisms can be deployed without concern for standards.

This final chapter has summarized the contributions of the previous chapters and has speculated on the future of content and service delivery. Given appropriate mechanisms for measurement such as NCS or SPE, prefetching systems can demonstrate significant contributions in the quest to improve user-perceived performance. While prefetching is just one tool to help improve performance, it is one that should not be ignored.

References

[Aas02] Gisle Aas. libwww-perl home page. `http://www.linpro.no/lwp/`, 2002.

[AAY97a] Jussara Almeida, Virgílio A. F. Almeida, and David J. Yates. Measuring the behavior of a World Wide Web server. In *Proceedings of the Seventh Conference on High Performance Networking (HPN)*, pages 57–72. IFIP, April 1997.

[AAY97b] Jussara Almeida, Virgílio A. F. Almeida, and David J. Yates. WebMonitor: A tool for measuring World Wide Web server performance. *first monday*, 2(7), July 1997.

[AC98a] Jussara Almeida and Pei Cao. Measuring proxy performance with the Wisconsin Proxy Benchmark. *Computer Networks And ISDN Systems*, 30(22-23):2179–2192, November 1998.

[AC98b] Jussara Almeida and Pei Cao. Wisconsin Proxy Benchmark 1.0. Available from `http://www.cs.wisc.edu/~cao/wpb1.0.html`, 1998.

[Adl99] Stephen Adler. The Slashdot effect: An analysis of three Internet publications. Available at `http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html`, January 1999.

[Ado02] Adobe Systems Incorporated. Adobe Acrobat Reader. `http://www.adobe.com/products/acrobat/readermain.html`, 2002.

[AFJM95] Robert Armstrong, Dayne Freitag, Thorsten Joachims, and Tom Mitchell. WebWatcher: A learning apprentice for the World Wide Web. In *Proceedings of the AAAI Spring Symposium on Information Gathering from Distributed, Heterogeneous Environments*, Stanford University, March 1995. AAAI Press.

[Aka02] Akamai Technologies, Inc. Akamai home page. `http://www.akamai.com/`, 2002.

[Alt02] AltaVista Company. AltaVista — The search company. `http://www.altavista.com/`, 2002.

[Ami97] Einat Amitay. Hypertext — The importance of being different. Master's thesis, Edinburgh University, Scotland, 1997. Also Technical Report No. HCRC/RP-94.

[Ami98] Einat Amitay. Using common hypertext links to identify the best phrasal description of target web documents. In *Proceedings of the SIGIR'98 Post-Conference Workshop on Hypertext Information Retrieval for the Web*, Melbourne, Australia, 1998.

[Ami99] Einat Amitay. Anchors in context: A corpus analysis of web pages authoring conventions. In Lynn Pemberton and Simon Shurville, editors, *Words on the Web — Computer Mediated Communication*. Intellect Books, UK, October 1999.

[Ami00] Einat Amitay. InCommonSense — Rethinking Web search results. In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME 2000)*, New York, 2000.

[Ami01] Einat Amitay. Trends, fashions, patterns, norms, conventions... and hypertext too. *Journal of the American Society for Information Science (JASIS)*, 52(1), 2001. Special Issue on Information Science at the Millennium. Also available as CSIRO Technical Report 66-2000.

[AMR+01] Virgílio Almeida, Daniel Menascé, Rudolf Riedi, Flávia Peligrinelli, Rodrigo C. Fonseca, and Wagner Meira, Jr. Analyzing the impact of robots on performance of Web caching systems. In *Web Caching and Content Delivery: Proceedings of the Sixth International Web Content Caching and Distribution Workshop (WCW'01)*, Boston, MA, June 2001.

[And97] Tom Andrews. Computer command prediction. Master's thesis, University of Nevada, Reno, May 1997.

[AP00] Einat Amitay and Cecile Paris. Automatically summarising Web sites — Is there a way around it?. In *Proceedings of the Ninth ACM International Conference on Information and Knowledge Management (CIKM 2000)*, Washington, DC, November 2000.

[Apa02] Apache Group. Apache HTTP server documentation. Available at `http://httpd.apache.org/docs/`, 2002.

[ATT02] AT&T Intelligent Content Distribution Service. `http://www.ipservices.att.com/icds`, 2002.

[AW97] Martin F. Arlitt and Carey L. Williamson. Internet Web servers: workload characterization and performance implications. *IEEE/ACM Transactions on Networking (TON)*, 5(5):631–645, October 1997.

[AZN99] David W. Albrecht, Ingrid Zukerman, and Ann E. Nicholson. Pre-sending documents on the WWW: A comparative study. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, volume 2, pages 1274–1279, Stockholm, Sweden, 1999. Morgan Kaufmann.

[Bau96] Mathias Bauer. Acquisition of user preferences for plan recognition. In David Chin, editor, *Proceedings of the Fifth International Conference on User Modeling (UM96)*, 1996.

[BB98] Krishna Bharat and Andrei Broder. A technique for measuring the relative size and overlap of public Web search engines. In *Proceedings of the Seventh International World Wide Web Conference*, Brisbane, Australia, April 1998.

[BBBC99] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in Web client access patterns: Characteristics and caching implications. *World Wide Web*, 2(1):15–28, January 1999.

[BBK00] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into Web server design. In *Proceedings of the Ninth International World Wide Web Conference*, Amsterdam, May 2000.

[BC98a] Paul Barford and Mark Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98/PERFORMANCE '98)*, pages 151–160, Madison, WI, June 1998.

[BC98b] Paul Barford and Mark Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[BCF+99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE INFOCOM*, New York, March 1999.

[BCGMS00] Onn Brandman, Junghoo Cho, Hector Garcia-Molina, and Shiva Shivakumar. Crawler-friendly Web servers. *Performance Evaluation Review*, 28(2), September 2000. Presented at the Performance and Architecture of Web Servers (PAWS) Workshop, June 2000.

[BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[BD99] Gaurav Banga and Peter Druschel. Measuring the capacity of a Web server under realistic loads. *World Wide Web Journal*, 2(1-2):69–83, 1999. Special Issue on World Wide Web Characterization and Performance Evaluation.

[BDR97] Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic deltas for WWW latency reduction. In *Proceedings of the USENIX Technical Conference*, 1997.

[BEF+00] Lee Breslau, Deborah Estron, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[Ber00] Michael K. Bergman. The deep Web: Surfacing hidden value. White paper, BrightPlanet.com, July 2000. Available from `http://www.completeplanet.com /tutorials/deepweb/index.asp`.

[Bes95] Azer Bestavros. Using speculation to reduce server load and service time on the WWW. In *Proceedings of CIKM'95: The Fourth ACM International Conference on Information and Knowledge Management*, Baltimore, MD, November 1995.

[Bes96] Azer Bestavros. Speculative data dissemination and service to reduce server load, network traffic and service time for distributed information systems. In *Proceedings of the International Conference on Data Engineering (ICDE'96)*, New Orleans, LA, March 1996.

[BFJ96] Justin Boyan, Dayne Freitag, and Thorsten Joachims. A machine learning architecture for optimizing Web search engines. In *AAAI Workshop on Internet-Based Information Systems*, Portland, OR, August 1996.

[BFOS84] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, 1984.

[BH96] Jean-Chrysostome Bolot and Philipp Hoschka. Performance engineering of the World Wide Web: Application to dimensioning and cache design. In *Proceedings of the Fifth International World Wide Web Conference*, Paris, France, May 1996.

[BH98a] Adam Belloum and L. O. Hertzberger. Replacement strategies in Web caching. In *Proceedings of ISIC/CIRA/ISAS'98*, Gaithersburg, MD, September 1998.

[BH98b] Krishna Bharat and Monika R. Henzinger. Improved algorithms for topic distillation in hyperlinked environments. In *Proceedings of the 21st International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 104–111, August 1998.

[BH00] Adam Belloum and Bob Hertzberger. Maintaining Web cache coherency. *Information Research*, 6(1), October 2000.

[BJ01] Peter J. Brown and Gareth J. F. Jones. Context-aware retrieval: Exploring a new environment for information retrieval and information filtering. *Personal and Ubiquitous Computing*, 5(4):253–263, 2001.

[BK00] Carla Brodley and Ronny Kohavi. KDD Cup 2000. Online at `http://www.ecn.purdue.edu/KDDCUP/`, 2000.

[BKNM02] Hyokyung Bahn, Kern Koh, Sam H. Noh, and Sang Lyul Min. Efficient replacement of nonuniform objects in web caches. *IEEE Computer*, 35(6):65–73, June 2002.

[BLC95] Tim Berners-Lee and D. Connolly. Hypertext markup language - 2.0. RFC 1866, `http://ftp.isi.edu/ in-notes/rfc1866.txt`, November 1995.

[BLFF96] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielson. Hypertext Transfer Protocol — HTTP/1.0. RFC 1945, `http://ftp.isi.edu/in-notes/rfc1945.txt`, May 1996.

[BLFM98] Tim Berners-Lee, Roy T. Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. RFC 2396, `http://ftp.isi.edu/in-notes /rfc2396.txt`, August 1998.

[BLMM94] Tim Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL). RFC 1738, `http://ftp.isi.edu/in-notes /rfc1738.txt`, December 1994.

[BMK97] Rob Barrett, Paul P. Maglio, and Daniel C. Kellem. How to personalize the Web. In *Proceedings of the ACM SIGCHI'97 Conference on Human Factors in Computing Systems*, pages 75–82, Atlanta, GA, March 1997. ACM Press.

[BO00] Greg Barish and Katia Obraczka. World Wide Web caching: Trends and techniques. *IEEE Communications Magazine Internet Technology Series*, 38(5):178–184, May 2000.

[Bot95] EPA-HTTP server logs. Laura Bottomley of Duke University, 1995. Available from `http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html`.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, Brisbane, Australia, April 1998.

[Bra86] James T. Brady. A theory of productivity in the creative process. *IEEE Computer Graphics and Applications*, 6(5):25–34, May 1986.

[Bra96] Tim Bray. Measuring the Web. In *Proceedings of the Fifth International World Wide Web Conference*, Paris, France, May 1996.

[BS97] Marko Balabanovic and Yoav Shoham. Fab: Content-based, collaborative recommendation. *Communications of the ACM*, 40(3), March 1997.

[BS00] B. R. Badrinath and Pradeep Sudame. Gathercast: The design and implementation of a programmable aggregation mechanism for the Internet. In *Proceedings of IEEE International Conference on Computer Communications and Networks (ICCCN)*, October 2000.

[BSHJ+99] Israel Ben-Shaul, Michael Herscovici, Michal Jacovi, Yoelle S. Maarek, Dan Pelleg, Menachem Shtalhaim, Vladimir Soroka, and Sigalit Ur. Adding support for dynamic and focused search with Fetuccino. In *Proceedings of the Eighth International World Wide Web Conference*, Toronto, Canada, May 1999.

[BV02] Leeann Bent and Geoffrey M. Voelker. Whole page performance. In *Proceedings of the Seventh International Workshop on Web Content Caching and Distribution (WCW'02)*, Boulder, CO, August 2002.

[Cab02] Cable & Wireless Internet Services, Inc. Exodus, a Cable & Wireless service. `http://www.exodus.com/`, 2002.

[Cac00] CacheFlow Inc. CacheFlow performance testing tool. `http://www.cacheflow.com/technology/tools/`, 2000.

[Cac02a] CacheFlow Inc. Active caching technology. `http://www.cacheflow.com /technology/whitepapers/active.cfm`, 2002.

[Cac02b] CacheFlow Inc. CacheFlow products web page. `http://www.cacheflow.com/products/`, 2002.

[Cao97] Pai Cao. Web cache simulator. Available from Pei Cao's Univ. of Wisconsin home page: `http://www.cs.wisc.edu/~cao/`, 1997.

[CB97] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[CB98] Mark Crovella and Paul Barford. The network effects of prefetching. In *Proceedings of IEEE INFOCOM*, San Francisco, CA, 1998.

[CBC95] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW client-based traces. Technical Report TR-95-010, Computer Science Department, Boston University, July 1995.

[CC95] Mark E. Crovella and Robert L. Carter. Dynamic server selection in the Internet. In *Proceedings of the Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS'95)*, August 1995.

[CC96] Robert L. Carter and Mark E. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. Technical Report TR-96-007, Computer Science Department, Boston University, March 1996.

[CDF+98] Ramón Cáceres, Fred Douglis, Anja Feldmann, Gideon Glass, and Michael Rabinovich. Web proxy caching: The Devil is in the details. *Performance Evaluation Review*, 26(3):11–15, December 1998. Proceedings of the Workshop on Internet Server Performance.

[CDG+99] Soumen Chakrabarti, Byron E. Dom, David Gibson, Jon M. Kleinberg, S. Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Mining the Web's link structure. *IEEE Computer*, pages 60–67, August 1999.

[CDI98] Soumen Chakrabarti, Byron E. Dom, and Piotr Indyk. Enhanced hypertext categorization using hyperlinks. In *Proceedings of ACM SIGMOD*, Seattle, WA, 1998.

[CDN+96] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical Internet object cache. In *Proceedings of the USENIX Technical Conference*, pages 153–163, San Diego, CA, January 1996.

[CDR+98] Soumen Chakrabarti, Byron E. Dom, Prabhakar Raghavan, Sridhar Rajagopalan, David Gibson, and Jon M. Kleinberg. Automatic resource compilation by analyzing hyperlink structure and associated text. In *Proceedings of the Seventh International World Wide Web Conference*, Brisbane, Australia, April 1998.

[CGMP98] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through URL ordering. In *Proceedings of the Seventh International World Wide Web Conference*, Brisbane, Australia, April 1998.

[Cha99] Philip K. Chan. A non-invasive learning approach to building Web user profiles. In *Proceedings of WebKDD'99*, pages 7–12, San Diego, August 1999. Revised version in Springer LNCS Vol. 1836.

[CI97] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technology and Systems*, pages 193–206, December 1997.

[Cid02] Cidera, Inc. Cidera home page. http://www.cidera.com/, 2002.

[CJ97] Carlos R. Cunha and Carlos F. B. Jaccoud. Determining WWW user's next access and its application to prefetching. In *Proceedings of Second IEEE Symposium on Computers and Communications (ISCC'97)*, Alexandria, Egypt, July 1997.

[CK00] Edith Cohen and Haim Kaplan. Prefetching the means for document transfer: A new approach for reducing Web latency. In *Proceedings of IEEE INFOCOM*, Tel Aviv, Israel, March 2000.

[CK01a] Edith Cohen and Haim Kaplan. Proactive caching of DNS records: Addressing a performance bottleneck. In *Proceedings of The 2001 Symposium on Applications and the Internet (SAINT-2001)*, San Diego, January 2001.

[CK01b] Edith Cohen and Haim Kaplan. Refreshment policies for Web content caches. In *Proceedings of IEEE INFOCOM*, Anchorage, AK, April 2001.

[CK02] Peter Christy and John Katsaros. The 2002 content distribution at the edge report. Report, NetsEdge Research Group, January 2002.

[CKR98] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Evaluating server-assisted cache replacement in the Web. In *Proceedings of the 6th European Symposium on Algorithms*, August 1998. LLNCS vol. 1461.

[CKR99] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Efficient algorithms for predicting requests to Web servers. In *Proceedings of IEEE INFOCOM*, New York, March 1999.

[CKV93] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *Proceedings of the ACM-SIGMOD Conference on Management of Data*, pages 257–266, May 1993.

[CKZ99] Edith Cohen, Haim Kaplan, and Uri Zwick. Connection caching. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*. ACM, 1999.

[CM99] Xiangping Chen and Prasant Mohapatra. Lifetime behavior and its impact on Web caching. In *Proceedings of the IEEE Workshop on Internet Applications (WIAPP' 99)*, San Jose, CA, July 1999.

[Cor98] Andrew Cormack. Experiences with installing and running an institutional cache. In *Proceedings of the Third International WWW Caching Workshop*, Manchester, England, June 1998.

[CP95] Lara D. Catledge and James E. Pitkow. Characterizing Browsing Strategies in the World Wide Web. *Computer Networks and ISDN Systems*, 26(6):1065–1073, 1995.

[CSS99] William W. Cohen, Robert E. Schapire, and Yoram Singer. Learning to order things. *Journal of Artificial Intelligence Research*, 10:243–270, 1999.

[Cun97] Carlos R. Cunha. *Trace analysis and its applications to performance enhancements of distributed information systems*. PhD thesis, Computer Science Department, Boston University, 1997.

[CvdBD99] Soumen Chakrabarti, Martin van den Berg, and Byron E. Dom. Focused crawling: A new approach to topic-specific Web resource discovery. In *Proceedings of the Eighth International World Wide Web Conference*, Toronto, Canada, May 1999.

[CY97] Ken-ichi Chinen and Suguru Yamaguchi. An interactive prefetching proxy server for improvement of WWW latency. In *Proceedings of the Seventh Annual Conference of the Internet Society (INET'97)*, Kuala Lumpur, June 1997.

[Cyp93a] Allen Cypher. Eager: Programming repetitive tasks by demonstration. In Cypher [Cyp93b], pages 204–217.

[Cyp93b] Allen Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993.

[CZB99] Pei Cao, Jin Zhang, and Kevin Beach. Active Cache: Caching dynamic contents on the Web. *Distributed Systems Engineering*, 6(1):43–50, 1999.

[Dan98] Peter B. Danzig. NetCache architecture and deployment. In *Proceedings of the Third International WWW Caching Workshop*, Manchester, England, June 1998.

[Dav99a] Brian D. Davison. Adaptive Web prefetching. In *Proceedings of the 2nd Workshop on Adaptive Systems and User Modeling on the WWW*, pages 105–106, Toronto, May 1999. Position paper. Proceedings published as Computing Science Report 99-07, Dept. of Mathematics and Computing Science, Eindhoven University of Technology.

[Dav99b] Brian D. Davison. Measuring the performance of prefetching proxy caches. Poster presented at the ACM International Student Research Competition, New Orleans, March 24-28, 1999 and at the AT&T Student Research Symposium (a regional ACM Student Research Competition), November 13, 1998. Awarded third place and first place respectively., 1999.

[Dav99c] Brian D. Davison. Web traffic logs: An imperfect resource for evaluation. In *Proceedings of the Ninth Annual Conference of the Internet Society (INET'99)*, June 1999.

[Dav99d] Brian D. Davison. Simultaneous proxy evaluation. In *Proceedings of the Fourth International Web Caching Workshop (WCW99)*, pages 170–178, San Diego, CA, March 1999.

[Dav99e] Brian D. Davison. A survey of proxy cache evaluation techniques. In *Proceedings of the Fourth International Web Caching Workshop (WCW99)*, pages 67–77, San Diego, CA, March 1999.

[Dav00a] Brian D. Davison. Recognizing nepotistic links on the Web. In *Artificial Intelligence for Web Search*, pages 23–28. AAAI Press, July 2000. Presented at the AAAI-2000 workshop on Artificial Intelligence for Web Search, Technical Report WS-00-01.

[Dav00b] Brian D. Davison. Topical locality in the Web: Experiments and observations. Technical Report DCS-TR-414, Department of Computer Science, Rutgers University, 2000.

[Dav00c] Brian D. Davison. Topical locality in the Web. In *Proceedings of the 23rd Annual ACM International Conference on Research and Development in Information Retrieval (SIGIR 2000)*, Athens, Greece, July 2000.

[Dav01a] Brian D. Davison. Assertion: Prefetching with GET is not good. In A. Bestavros and M. Rabinovich, editors, *Web Caching and Content Delivery: Proceedings of the Sixth International Web Content Caching and Distribution Workshop (WCW'01)*, pages 203–215, Boston, MA, June 2001. Elsevier.

[Dav01b] Brian D. Davison. HTTP simulator validation using real measurements: A case study. In *Ninth International Symposium on Modeling, Analysis, and Simulation on Computer and Telecommunication Systems (MASCOTS 2001)*, Cincinatti, OH, August 2001.

[Dav01c] Brian D. Davison. A Web caching primer. *IEEE Internet Computing*, 5(4):38–45, July/August 2001.

[Dav02a] Brian D. Davison. Predicting Web actions from HTML content. In *Proceedings of the The Thirteenth ACM Conference on Hypertext and Hypermedia (HT'02)*, pages 159–168, College Park, MD, June 2002.

[Dav02b] Brian D. Davison. web-caching.com: Web caching and content delivery resources. `http://www.web-caching.com/`, 2002.

[DB96] Fred Douglis and Thomas Ball. Tracking and viewing changes on the Web. In *Proceedings of the USENIX Technical Conference*, pages 165–176, San Diego, January 1996.

[DCGV01] Ronald P. Doyle, Jeffrey S. Chase, Syam Gadde, and Amin M. Vahdat. The trickle-down effect: Web caching and server request distribution. In *Web Caching and Content Delivery: Proceedings of the Sixth International Web Content Caching and Distribution Workshop (WCW'01)*, Boston, MA, June 2001.

[Dee02] Deerfield Corporation. InterQuick for WinGate. `http://interquick.deerfield.com/`, 2002.

[DFKM97] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey C. Mogul. Rate of change and other metrics: A live study of the World Wide Web. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)*, December 1997.

[DGK+99] Brian D. Davison, Apostolos Gerasoulis, Konstantinos Kleisouris, Yingfang Lu, Hyunju Seo, Wei Wang, and Baohua Wu. DiscoWeb: Applying link analysis to Web search. In *Poster proceedings of the Eighth International World Wide Web Conference*, pages 148–149, Toronto, Canada, May 1999.

[DH97a] Brian D. Davison and Haym Hirsh. Experiments in UNIX command prediction. Technical Report ML-TR-41, Department of Computer Science, Rutgers University, 1997.

[DH97b] Brian D. Davison and Haym Hirsh. Toward an adaptive command line interface. In *Advances in Human Factors/Ergonomics: Design of Computing Systems: Social and Ergonomic Considerations*, pages 505–508, San Francisco, CA, August 1997. Elsevier Science Publishers. Proceedings of the Seventh International Conference on Human-Computer Interaction.

[DH97c] Daniel Dreilinger and Adele E. Howe. Experiences with selected search engines using metasearch. *ACM Transactions on Information Systems*, 15(3):195–222, July 1997.

[DH98] Brian D. Davison and Haym Hirsh. Predicting sequences of user actions. In *Predicting the Future: AI Approaches to Time-Series Problems*, pages 5–12, Madison, WI, July 1998. AAAI Press. Proceedings of AAAI-98/ICML-98 Workshop, published as Technical Report WS-98-07.

[DH99] Jeffrey Dean and Monika R. Henzinger. Finding related pages in the World Wide Web. In *Proceedings of the Eighth International World Wide Web Conference*, pages 389–401, Toronto, Canada, May 1999.

[DHR97] Fred Douglis, Antonio Haro, and Michael Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)*, pages 83–94, Monterey, CA, December 1997. USENIX Association.

[DK01] Mukund Deshpande and George Karypis. Selective Markov models for predicting Web-page accesses. In *Proceedings of the First SIAM International Conference on Data Mining (SDM'2001)*, Chicago, April 2001.

[DKNS01] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the Web. In *Proceedings of the Tenth International World Wide Web Conference*, Hong Kong, May 2001.

[DKW02] Brian D. Davison, Chandrasekar Krishnan, and Baoning Wu. When does a hit = a miss? In *Proceedings of the Seventh International Workshop on Web Content Caching and Distribution (WCW'02)*, Boulder, CO, August 2002. Accepted for publication.

[DL00] Brian D. Davison and Vincenzo Liberatore. Pushing politely: Improving Web responsiveness one packet at a time (Extended abstract). *Performance Evaluation Review*, 28(2):43–49, September 2000. Presented at the Performance and Architecture of Web Servers (PAWS) Workshop, June 2000.

[DM92] Patrick W. Demasco and Kathleen F. McCoy. Generating text from compressed input: An intelligent interface for people with severe motor impairments. *Communications of the ACM*, 35(5):68–78, May 1992.

[DMF97] Bradley M. Duska, David Marwood, and Michael J. Feely. The measured access characteristics of World-Wide-Web client proxy caches. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)*, pages 23–36, Monterey, CA, December 1997. USENIX Association.

[DMS97] Matjaz Debevc, Beth Meyer, and Rajko Svecko. An adaptive short list for documents on the World Wide Web. In *Proceedings of the 1997 International Conference on Intelligent User Interfaces*, pages 209–211, Orlando, FL, 1997. ACM Press.

[DP96] Adam Dingle and Tomas Partl. Web cache coherence. *Computer Networks and ISDN Systems*, 28(7-11):907–920, May 1996.

[DS98] William DuMouchel and Matt Schonlau. A fast computer intrusion detection algorithm based on hypothesis testing of command transition probabilities. In Agrawal, Stolorz, and Piatetsky-Shapiro, editors, *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, pages 189–193, New York, 1998. AAAI Press.

[DT82] Walter J. Doherty and Ahrvind J. Thadani. The economic value of rapid response time. Technical Report GE20-0752-0, IBM, White Plains, NY, November 1982.

[Duc99] Dan Duchamp. Prefetching hyperlinks. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS '99)*, Boulder, CO, October 1999.

[Dum95] NASA-HTTP server logs. Jim Dumoulin of the NASA Kennedy Space Center, 1995. Available from `http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html`.

[DWJ90] John J. Darragh, Ian H. Witten, and M. L. James. The reactive keyboard: A predictive typing aid. *IEEE Computer*, 23(11):41–49, November 1990.

[eAc01a] eAcceleration Corporation. Webcelerator help page on prefetching. `http://www.webcelerator.com/webcelerator/prefetch.htm`, 2001.

[eAc01b] eAcceleration Corporation. Webcelerator home page. `http://www.webcelerator.com/products/webcelerator/`, 2001.

[EBHDC01] Samhaa R. El-Beltagy, Wendy Hall, David De Roure, and Leslie Carr. Linking in context. In *Proceedings of the Twelfth ACM Conference on Hypertext and Hypermedia (HT'01)*, Aarhus, Denmark, August 2001.

[ESNP96] Mansour Esmaili, Reihaneh Safavi-Naini, and Joseph Pieprzyk. Evidential reasoning in network intrusion detection systems. In Josef Pieprzyk and Jennifer Seberry, editors, *Information Security and Privacy: First Australasian Conference, ACISP'96*, pages 253–265, Wollongong, NSW, Australia, June 1996. Springer-Verlag. Lecture Notes in Computer Science 1172.

[Exp02] Expand Networks, Inc. Expand Networks homepage. `http://www.expand.com/`, 2002.

[Fas02] Fast Search & Transfer ASA. Fast home page. `http://www.fastsearch.com/`, 2002.

[FCAB98] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. *Computer Communication Review*, 28(4), October 1998. Proceedings of ACM SIGCOMM.

[FCD+99] Anja Feldmann, Ramón Cáceres, Fred Douglis, Gideon Glass, and Michael Rabinovich. Performance of Web proxy caching in heterogeneous bandwidth environments. In *Proceedings of IEEE INFOCOM*, pages 106–116, New York, March 1999.

[Fel98] Anja Feldmann. Continuous online extraction of HTTP traces from packet traces. In *World Wide Web Consortium Workshop on Web Characterization*, Cambridge, MA, November 1998. Position paper.

[FGM+99] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk, L. Masinter, P. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2616, `http://ftp.isi.edu/in-notes/rfc2616.txt`, June 1999.

[FJCL99] Li Fan, Quinn Jacobson, Pei Cao, and Wei Lin. Web prefetching between low-bandwidth clients and proxies: Potential and performance. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99)*, Atlanta, GA, May 1999.

[FLGC02] Gary William Flake, Steve Lawrence, C. Lee Giles, and Frans M. Coetzee. Self-organization and identification of web communities. *IEEE Computer*, 35(3):66–71, 2002.

[Flo99] Sally Floyd. Validation experiences with the ns simulator. In *Proceedings of the DARPA/NIST Network Simulation Validation Workshop*, Fairfax, VA, May 1999.

[FP99] Tom Fawcett and Foster Provost. Activity monitoring: Noticing interesting changes in behavior. In *Proceedings of the fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 53–62, San Diego, CA, August 1999.

[FS96] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *Proc. 13th International Conference on Machine Learning*, pages 148–156, 1996.

[FS00] Dan Foygel and Dennis Strelow. Reducing Web latency with hierarchical cache-based prefetching. In *Proceedings of the International Workshop on Scalable Web Services (in conjunction with ICPP'00)*, Toronto, August 2000.

[FX00] Guoliang Fan and Xiang-Gen Xia. Maximum likelihood texture analysis and classification using wavelet-domain hidden markov models. In *Proceedings of the 34th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, 2000.

[Gad01] Syam Gadde. The Proxycizer Web proxy tool suite. Available at `http://www.cs.duke.edu/ari/cisi/Proxycizer/`, 2001.

[GB97] Steven D. Gribble and Eric A. Brewer. System design issues for Internet middleware services: Deductions from a large client trace. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)*, December 1997.

[GCR97] Syam Gadde, Jeff Chase, and Michael Rabinovich. Directory structures for scalable Internet caches. Technical Report CS-1997-18, Department of Computer Science, Duke University, November 1997.

[GCR98] Syam Gadde, Jeff Chase, and Michael Rabinovich. A taste of crispy squid. In *Workshop on Internet Server Performance (WISP'98)*, Madison, WI, June 1998.

[GDMW95] Saul Greenberg, John J. Darragh, David Maulsby, and Ian H. Witten. Predictive interfaces: What will they think of next? In Alistair D. N. Edwards, editor, *Extra-ordinary human–computer interaction: interfaces for users with disabilities*, chapter 6, pages 103–140. Cambridge University Press, 1995.

[GKR98] David Gibson, Jon M. Kleinberg, and Prabhakar Raghavan. Inferring Web communities from link topology. In *Proceedings of the 9th ACM Conference on Hypertext and Hypermedia (Hypertext'98)*, 1998. Expanded version at `http://www.cs.cornell.edu/home/kleinber/`.

[Goo02] Google Inc. Google home page. `http://www.google.com/`, 2002.

[GP00a] Peter Gorniak and David Poole. Building a stochastic dynamic model of application use. In *Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI-2000)*, July 2000.

[GP00b] Peter Gorniak and David Poole. Predicting future user actions by observing unmodified applications. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 217–222, Austin, TX, July 2000. AAAI Press.

[GPB98] Arthur Goldberg, Ilya Pevzner, and Robert Buff. Caching characteristics of Internet and intranet Web proxy traces. In *Computer Measurement Group Conference CMG98*, Anaheim, CA, December 1998.

[GRC97] Syam Gadde, Michael Rabinovich, and Jeff Chase. Reduce, reuse, recycle: An approach to building large Internet caches. In *The Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 93–98, May 1997.

[Gre88] Saul Greenberg. Using Unix: collected traces of 168 users. Research Report 88/333/45, Department of Computer Science, University of Calgary, Alberta, 1988. Includes tar-format cartridge tape.

[Gre93] Saul Greenberg. *The Computer User as Toolsmith: The Use, Reuse, and Organization of Computer-based Tools*. Cambridge University Press, New York, NY, 1993.

[Gri97] Steven D. Gribble. UC Berkely Home IP HTTP traces. Online: `http://www.acm.org/sigcomm/ITA/`, July 1997.

[GS96] James Gwertzman and Margo Seltzer. World-Wide Web cache consistency. In *Proceedings of the USENIX Technical Conference*, pages 141–151, San Diego, CA, January 1996. USENIX Association.

[HD97] Adele Howe and Daniel Dreilinger. SavvySearch: A metasearch engine that learns which search engines to query. *AI Magazine*, 18(2), 1997.

[HG98] Stephanie W. Haas and Erika S. Grams. Page and link classifications: Connecting diverse resources. In *Proceedings of the third ACM Conference on Digital libraries*, May 1998.

[HHMN00] Monika R. Henzinger, Allan Heydon, Michael Mitzenmacher, and Marc Najork. On near-uniform URL sampling. In *Proceedings of the Ninth International World Wide Web Conference*, Amsterdam, May 2000.

[HL96] Barron C. Housel and David B. Lindquist. WebExpress: A system for optimizing Web browsing in a wireless environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking (MobiCom '96)*, pages 108–116, Rye, New York, November 1996. ACM/IEEE.

[HMK00] John Heidemann, Kevin Mills, and Sri Kumar. Expanding confidence in network simulation. Research Report 00-522, USC/Information Sciences Institute, April 2000.

[HMT98] Tsukasa Hirashima, Noriyuki Matsuda, and Jun'ici Toyoda. Context-sensitive filtering for hypertext browsing. In *Proceedings of the 1998 International Conference on Intelligent User Interfaces*. ACM Press, 1998.

[Hol98] K. Holtman. The safe response header field. RFC 2310, `http://ftp.isi.edu/in-notes /rfc2310.txt`, April 1998.

[Hor98] Eric Horvitz. Continual computation policies for utility-directed prefetching. In *Proceedings of the Seventh ACM Conference on Information and Knowledge Management*, pages 175–184, Bethesda, MD, November 1998. ACM Press: New York.

[HOT97] John Heidemann, Katia Obraczka, and Joe Touch. Modeling the performance of HTTP over several transport protocols. *IEEE/ACM Transactions on Networking*, 5(5):616–630, October 1997.

[HP87] Fredrick J. Hill and Gerald R. Peterson. *Digital Systems: Hardware Organization and Design*. John Wiley & Sons, New York, 1987. Third Edition.

[HRI+00] Ernst Georg Haffner, Uwe Roth, Andreas Heuer II, Thomas Engel, and Christoph Meinel. What do hyperlink-proposals and request-prediction have in common? In *Proceedings of the International Conference on Advances in Information Systems (ADVIS)*, pages 275–282, 2000. Springer LNCS 1909.

[HS93] Leonard A. Hermens and Jeffrey C. Schlimmer. A machine-learning apprentice for the completion of repetitive forms. In *Proceedings of the Ninth IEEE Conference on Artificial Intelligence Applications*, pages 164–170, Los Alamitos, CA, March 1993. IEEE Computer Society Press.

[HT01] Amy S. Hughes and Joseph D. Touch. Expanding the scope of prefetching through inter-application cooperation. In *Web Caching and Content Delivery: Proceedings of the Sixth International Web Content Caching and Distribution Workshop (WCW'01)*, Boston, MA, June 2001. Synopsis of work in progress.

[HWMS98] John H. Hine, Craig E. Wills, Anja Martel, and Joel Sommers. Combining client knowledge and resource dependencies for improved World Wide Web performance. In *Proceedings of the Eighth Annual Conference of the Internet Society (INET'98)*, Geneva, Switzerland, July 1998.

[IBM00] IBM Almaden Research Center. The CLEVER Project. Home page: `http://www.almaden.ibm.com/cs/k53/clever.html`, 2000.

[IC97] Arun Iyengar and Jim Challenger. Improving web server performance by caching dynamic data. In *USENIX Symposium on Internet Technologies and Systems (USITS'97)*, 1997.

[Ims02] Imsisoft. Imsisoft home page. http://www.imsisoft.com/, 2002.

[Inf02a] InfoSpace, Inc. Dogpile home page. `http://www.dogpile.com/`, 2002.

[Inf02b] InfoSpace, Inc. MetaCrawler home page. `http://www.metacrawler.com/`, 2002.

[Ink02] Inktomi Corporation. Inktomi home page. `http://www.inktomi.com/`, 2002.

[Int99] Internet Research Group. Worldwide Internet caching market forecast. *The Standard*, August 1999. Chart available from `http://www.thestandard.com/research/metrics/display/0,2799,9823,00.html`.

[IX00] Tamer I. Ibrahim and Cheng-Zhong Xu. Neural net based pre-fetching to tolerate WWW latency. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS2000)*, April 2000.

[JB01] Nico Jacobs and Hendrik Blockeel. From shell logs to shell scripts. In *International Workshop on Inductive Logic Programming*, pages 80–90, 2001.

[JBC98] John Judge, H.W.P. Beadle, and J. Chicharo. Sampling HTTP response packets for prediction of Web traffic volume statistics. In *Proceedings of IEEE Globecom*, November 1998.

[JFM97] Thorsten Joachims, Dayne Freitag, and Tom Mitchell. WebWatcher: A tour guide for the World Wide Web. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 770–775. Morgan Kaufmann, August 1997.

[JG99] Doug Joseph and Dirk Grunwald. Prefetching using Markov predictors. *Transactions on Computers*, 48(2), February 1999.

[JK97] Zhimei Jiang and Leonard Kleinrock. Prefetching links on the WWW. In *ICC '97*, pages 483–489, Montreal, Canada, June 1997.

[JK98] Zhimei Jiang and Leonard Kleinrock. An adaptive network prefetch scheme. *IEEE Journal on Selected Areas in Communications*, 16(3):358–368, April 1998.

[Joh98] Tommy Johnson. Webjamma. Available from `http://www.cs.vt.edu/~chitra/webjamma.html`, 1998.

[Jun00] Junkbusters Corporation. The Internet Junkbuster proxy. Available from `http://www.junkbuster.com/ijb.html`, 2000.

[KABL96] T. Koch, A. Ardo, A. Brummer, and S. Lundberg. The building and maintenance of robot based Internet search services: A review of current indexing and data collection methods. Prepared for Work Package 3 of EU Telematics for Research, project DESIRE; Available from `http://www.ub2.lu.se/desire/radar/reports/D3.11/`, September 1996.

[Kah97] Brewster Kahle. Preserving the Internet. *Scientific American*, 276(3):82–83, March 1997.

[KBM+00] Paul B. Kantor, Endre Boros, Benjamin Melamed, Vladimir Menkov, Bracha Shapira, and David J. Neu. Capturing human intelligence in the net. *Communications of the ACM*, 43(8):112–115, August 2000.

[KCK+96] Tracy Kimbrel, Pei Cao, Anna Karlin, Ed Felten, and Kai Li. Integrated parallel prefetching and caching. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '96)*, Philadelphia, PA, May 1996. Longer version available as Princeton Computer Science Department Technical Report TR-502-95, November 1995.

[Kel01] Terence Kelly. Thin-client Web access patterns: Measurements from a cache-busting proxy. In *Web Caching and Content Delivery: Proceedings of the Sixth International Web Content Caching and Distribution Workshop (WCW'01)*, Boston, MA, June 2001.

[KG00] Benjamin Korvemaker and Russell Greiner. Predicting UNIX command lines: Adjusting to user patterns. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 230–235, Austin, TX, July 2000. AAAI Press.

[Kha02] Igor Khasilev. Oops proxy server homesite. Available from `http://www.oops-cache.org/`, 2002.

[KL96] Thomas M. Kroeger and Darrell D. E. Long. Predicting file system actions from prior events. In *Proceedings of the USENIX Technical Conference*, pages 482–487, San Diego, CA, January 1996.

[Kle98] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA-98)*, pages 668–677, San Francisco, CA, January 1998. Expanded version at `http://www.cs.cornell.edu/home/kleinber/`.

[Kle99] Reinhard P. Klemm. WebCompanion: A friendly client-side Web prefetching agent. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):577–594, July/August 1999.

[KLM97] Thomas M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the bounds of Web latency reduction from caching and prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)*, December 1997.

[KMK99] Balachander Krishnamurthy, Jeffrey C. Mogul, and David M. Kristol. Key differences between HTTP/1.0 and HTTP/1.1. In *Proceedings of the Eighth International World Wide Web Conference*, pages 659–673, Toronto, Canada, May 1999.

[Kos94] Martijn Koster. A standard for robot exclusion. Available from: `http://www.robotstxt.org/wc/norobots.html`, 1994.

[KPS+99] Colleen Kehoe, Jim Pitkow, Kate Sutton, Gaurav Aggarwal, and Juan D. Rogers. Results of GVU's tenth World Wide Web user survey. Graphic, Visualization, & Usability Center, Georgia Institute of Technology, Atlanta, GA: `http://www.gvu.gatech.edu/user_surveys/survey-1998-10/tenthreport.html`, May 1999.

[KR98] Balachander Krishnamurthy and Jennifer Rexford. Software issues in characterizing Web server logs. In *World Wide Web Consortium Workshop on Web Characterization*, Cambridge, MA, November 1998. Position paper.

[KR00] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley, 2000.

[KR01] Balachander Krishnamurthy and Jennifer Rexford. *Web Protocols and Practice: HTTP 1.1, Networking Protocols, Caching, and Traffic Measurement*. Addison-Wesley, 2001.

[KRRT99] S. Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Trawling the Web for emerging cyber-communities. In *Proceedings of the Eighth International World Wide Web Conference*, Toronto, Canada, May 1999.

[KSW98] Michal Kurcewicz, Wojtek Sylwestrzak, and Adam Wierzbicki. A filtering algorithm for proxy caches. In *Proceedings of the Third International WWW Caching Workshop*, Manchester, England, June 1998.

[Kum95] Sandeep Kumar. *Classification and detection of computer intrusions*. PhD thesis, Department of Computer Science, Purdue University, West Lafayette, IN, 1995.

[KV01] Mimika Koletsou and Geoffrey M. Voelker. The Medusa proxy: A tool for exploring user-perceived Web performance. In *Web Caching and Content Delivery: Proceedings of the Sixth International Web Content Caching and Distribution Workshop (WCW'01)*, Boston, MA, June 2001.

[KW97] Balachander Krishnamurthy and Craig E. Wills. Study of piggyback cache validation for proxy caches in the world wide web. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)*. USENIX Association, December 1997.

[KW98a] Achim Kraiss and Gerhard Weikum. Integrated document caching and prefetching in storage hierarchies based on Markov-chain predictions. *VLDB Journal*, 7:141–162, 1998.

[KW98b] Balachander Krishnamurthy and Craig E. Wills. Piggyback server invalidation for proxy cache coherency. *Computer Networks and ISDN Systems*, 30, 1998. Also in *Proceedings of the Seventh International World Wide Web Conference*, pages 185-193, Brisbane, Australia, April 1998.

[Lai92] Philip Laird. Discrete sequence prediction and its applications. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, Menlo Park, CA, 1992. AAAI Press.

[LAJF98] Binzhang Lui, Ghaleb Abdulla, Tommy Johnson, and Edward A. Fox. Web response time and proxy caching. In *Proceedings of WebNet98*, Orlando, FL, November 1998.

[LaM96] Brian A. LaMacchia. *Internet Fish*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, 1996. Also available as AI Technical Report 1579, MIT Artificial Intelligence Laboratory.

[LaM97] Brian A. LaMacchia. The Internet Fish construction kit. In *Proceedings of the Sixth International World Wide Web Conference*, Santa Clara, CA, April 1997.

[Lan00] Terran Lane. *Machine learning techniques for the computer security domain of anomaly detection*. PhD thesis, Purdue University, August 2000.

[LBO99] Bin Lan, Stephane Bressan, and Beng Chin Ooi. Making Web servers pushier. In *Proceedings of the Workshop on Web Usage Analysis and User Profiling (WEBKDD'99)*, San Diego, CA, August 1999.

[LC98] Chengjie Liu and Pei Cao. Maintaining strong cache consistency for the World-Wide Web. *IEEE Transactions on Computers*, 47(4):445–457, April 1998.

[LCD01] Dan Li, Pei Cao, and M. Dahlin. WCIP: Web cache invalidation protocol. Internet-Draft draft-danli-wrec-wcip-01.txt, IETF, March 2001. Work in progress.

[LE95] Neal Lesh and Oren Etzioni. A sound and fast goal recognizer. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1704–1710. Morgan Kaufmann, 1995.

[Lee92] Alison Lee. *Investigations into History Tools for User Support*. PhD thesis, Computer Systems Research Institute, University of Toronto, April 1992. Available as Technical Report CSRI–271.

[Lee96] David C. Lee. Pre-fetch document caching to improve World-Wide Web user response time. Master's thesis, Virginia Tech., Blacksburg, VA, March 1996.

[Les97] Neal Lesh. Adaptive goal recognition. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, August 1997.

[LG98a] Steve Lawrence and C. Lee Giles. Context and page analysis for improved Web search. *IEEE Internet Computing*, 2(4):38–46, 1998.

[LG98b] Steve Lawrence and C. Lee Giles. Inquirus, the NECI meta search engine. In *Proceedings of the Seventh International World Wide Web Conference*, Brisbane, Australia, April 1998.

[LG98c] Steve Lawrence and C. Lee Giles. Searching the World Wide Web. *Science*, 280(5360):98–100, April 1998.

[LG99] Steve Lawrence and C. Lee Giles. Accessibility of information on the Web. *Nature*, 400:107–109, July 1999.

[LHC+98] Jieun Lee, Hyojung Hwang, Youngmin Chin, Hyunchul Kim, and Kilnam Chon. Report on the costs and benefits of cache hierarchy in Korea. In *Proceedings of the Third International WWW Caching Workshop*, Manchester, England, June 1998.

[Lie95] Henry Lieberman. Letizia: An agent that assists Web browsing. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 924–929, Montreal, August 1995.

[Lie97] Henry Lieberman. Autonomous interface agents. In *Proceedings of the ACM SIGCHI'97 Conference on Human Factors in Computing Systems*, Atlanta, GA, March 1997.

[Lit88] Nicholas Littlestone. Learning when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.

[LM99] Chi-Keung Luk and Todd C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers*, 48(2), 1999. Special Issue on Cache Memory and Related Problems.

[LNBL96] Ari Luotonen, Henrik Frystyk Nielsen, and Tim Berners-Lee. CERN/W3C httpd status page. `http://www.w3.org/Daemon/Status.html`, 1996.

[LS94] Philip Laird and Ronald Saul. Discrete sequence prediction and its applications. *Machine Learning*, 15(1):43–68, 1994.

[Luh01] James C. Luh. No bots allowed! *Interactive Week*, April 2001. Online at `http://www.zdnet.com/intweek/stories/news/0,4164,2707542,00.html`.

[Lun90] T. F. Lunt. IDES: An intelligent system for detecting intruders. In *Proceedings of the Symposium: Computer Security, Threat and Countermeasures*, Rome, Italy, 1990.

[LW94a] Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.

[LW94b] Nicholas Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108:212–261, 1994.

[LYBS00] Jeremy Lilley, Jason Yang, Hari Balakrishnan, and Srinivasan Seshan. A unified header compression framework for low-bandwidth links. In *Proceedings of MobiCom: Sixth Annual International Conference on Mobile Computing and Networking*, Boston, August 2000.

[Lyc02a] Lycos, Inc. HotBot home page. `http://hotbot.lycos.com/`, 2002.

[Lyc02b] Lycos, Inc. Lycos home page. `http://www.lycos.com/`, 2002.

[LYW01] Ian Tianyi Li, Qiang Yang, and Ke Wang. Classification pruning for Web-request prediction. In *Poster Proceedings of the 10th World Wide Web Conference (WWW10)*, Hong Kong, May 2001.

[LZ01] Richard Liston and Ellen Zegura. Using a proxy to measure client-side Web performance. In *Web Caching and Content Delivery: Proceedings of the Sixth International Web Content Caching and Distribution Workshop (WCW'01)*, Boston, MA, June 2001.

[Man82] M. Morris Mano. *Computer System Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1982. Second Edition.

[Mar96] Evangelos P. Markatos. Main memory caching of Web documents. *Computer Networks and ISDN Systems*, 28(7-11):893–905, May 1996.

[MB00] Filippo Menczer and Richard K. Belew. Adaptive retrieval agents: Internalizing local context and scaling up to the Web. *Machine Learning*, 39(2/3):203–242, 2000. Longer version available as Technical Report CS98-579, University of California, San Diego.

[MB01] Ross J. Micheals and Terrance E. Boult. Efficient evaluation of classification and recognition systems. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition*, December 2001.

[MC98] Evangelos P. Markatos and Catherine E. Chronaki. A top–10 approach for prefetching the Web. In *Proceedings of the Eighth Annual Conference of the Internet Society (INET'98)*, Geneva, Switzerland, July 1998.

[McB94] Oliver A. McBryan. GENVL and WWWW: Tools for taming the Web. In *Proceedings of the First International World Wide Web Conference*, Geneva, Switzerland, May 1994.

[MDFK97] Jeffrey Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of ACM SIGCOMM*, pages 181–194, Cannes, France, September 1997. An extended and corrected version appears as Research Report 97/4a, Digital Equipment Corporation Western Research Laboratory, December, 1997.

[Men97] Filippo Menczer. ARACHNID: Adaptive Retrieval Agents Choosing Heuristic Neighborhoods for Information Discovery http://dollar.biz.uiowa.edu/~fil/papers.html. In *Proceedings of the 14th International Conference on Machine Learning (ICML97)*, 1997.

[MF01] Alan L. Montgomery and Christos Faloutsos. Identifying Web browsing trends and patterns. *Computer*, 34(7):94–95, July 2001.

[MH99] Miles J. Murdocca and Vincent P. Heuring. *Principles of Computer Architecture*. Prentice Hall, 1999.

[MIB00] Jean-Marc Menaud, Valérie Issarny, and Michel Banâtre. Improving effectiveness of Web caching. In *Recent Advances in Distributed Systems*, volume 1752 of *LNCS*. Springer Verlag, 2000.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[MJ98] David Mosberger and Tai Jin. httperf—A tool for measuring Web server performance. *Performance Evaluation Review*, 26(3):31–37, December 1998.

[MKPF99] Evangelos P. Markatos, Manolis G.H. Katevenis, Dionisis Pnevmatikatos, and Michail Flouris. Secondary storage management for Web proxies. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS99)*, pages 93–104, 1999.

[ML97] Jeffrey C. Mogul and P. Leach. Simple hit-metering and usage-limiting for HTTP. RFC 2227, `http://ftp.isi.edu/in-notes/rfc2227.txt`, October 1997.

[Mla96] Dunja Mladenic. Personal WebWatcher: Implementation and design. Technical Report IJS-DP-7472, Department of Intelligent Systems, J. Stefan Institute, Univ. of of Ljubljana, Slovenia, October 1996.

[MMS85] Tom M. Mitchell, Sridhar Mahadevan, and Louis I. Steinberg. LEAP: A learning apprentice for VLSI design. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, August 1985.

[MN94] T. Masui and K. Nakayama. Repeat and predict — two keys to efficient text editing. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 118–123, New York, April 1994. ACM Press.

[Mog96] Jeffrey C. Mogul. Hinted caching in the Web. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[Mog00] Jeffrey C. Mogul. Squeezing more bits out of HTTP caches. *IEEE Network*, 14(3):6–14, May/June 2000.

[Mow94] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Computer Systems Laboratory, Stanford University, March 1994.

[MR97] Carlos Maltzahn and Kathy J. Richardson. Performance issues of enterprise level Web proxies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13–23, Seattle, WA, June 1997. ACM Press.

[MRG99] Carlos Maltzahn, Kathy J. Richardson, and Dirk Grunwald. Reducing the disk I/O of Web proxy server caches. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, California, June 1999.

[MRGM99] Carlos Maltzahn, Kathy J. Richardson, Dirk Grunwald, and James H. Martin. On bandwidth smoothing. In *Proceedings of the Fourth International Web Caching Workshop (WCW99)*, San Diego, CA, March 1999.

[MSC98] Stephen Manley, Margo Seltzer, and Michael Courage. A self-scaling and self-configuring benchmark for Web servers. In *Proceedings of the Joint*

International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98/PERFORMANCE '98), pages 270–271, Madison, WI, June 1998.

[MWE00] Anirban Mahanti, Carey Williamson, and Derek Eager. Traffic analysis of a Web proxy caching hierarchy. *IEEE Network*, 14(3):16–23, May/June 2000.

[MY97] Hiroshi Motoda and Kenichi Yoshida. Machine learning techniques to make computers easier to use. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1622–1631. Morgan Kaufmann, August 1997.

[Nat02] National Laboratory for Applied Network Research. A distributed testbed for national information provisioning. Home page: `http://www.ircache.net/`, 2002.

[Net99] Netscape, Inc. Mozilla source code. Available at `http://lxr.mozilla.org/classic/`, 1999.

[NGBS+97] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hakon Wium Lie, and Chris Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. *Computer Communications Review*, 27(4), October 1997. Proceedings of SIGCOMM '97. Also available as W3C NOTE-pipelining-970624.

[NLL00] H. Nielsen, P. Leach, and S. Lawrence. An HTTP extension framework. RFC 2774, `http://ftp.isi.edu/in-notes /rfc2774.txt`, February 2000.

[NLN98] Nicolas Niclausse, Zhen Liu, and Philippe Nain. A new and efficient caching policy for the world wide web. In *Workshop on Internet Server Performance (WISP'98)*, Madison, WI, June 1998.

[NM93] Craig G. Nevill-Manning. Programming by demonstration. *New Zealand Journal of Computing*, 4(2):15–24, May 1993.

[NM96] Craig G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, Department of Computer Science, University of Waikato, New Zealand, May 1996.

[NZA98] Ann E. Nicholson, Ingrid Zukerman, and David W. Albrecht. A decision-theoretic approach for pre-sending information on the WWW. In *Proceedings of the 5th Pacific Rim International Conference on Artificial Intelligence (PRICAI'98)*, pages 575–586, Singapore, 1998.

[OA01] Oracle Corporation and Akamai Technologies. Edge side includes home page. Online at `http://www.edge-delivery.org/`, 2001.

[Pad95] Venkata N. Padmanabhan. Improving World Wide Web latency. Technical Report UCB/CSD-95-875, UC Berkeley, May 1995.

[Pal98] Themistoklis Palpanas. Web prefetching using partial match prediction. Master's thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, CA, March 1998. Available as Technical Report CSRG-376.

[Par96] Tomas Partl. A comparison of WWW caching algorithm efficiency. In *ICM Workshop on Web Caching*, Warsaw, Poland, 1996.

[Pax99] Vern Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.

[PB97] Michael J. Pazzani and Daniel Billsus. Learning and revising user profiles: The identification of interesting web sites. *Machine Learning*, 27:313–331, 1997.

[PE97] Mike Perkowitz and Oren Etzioni. Adaptive Web sites: Automatically learning from user access patterns. In *Proceedings of the Sixth International World Wide Web Conference*, Santa Clara, CA, April 1997. Poster.

[PE98] Mike Perkowitz and Oren Etzioni. Adaptive Web sites: Automatically synthesizing Web pages. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, WI, July 1998. AAAI Press.

[PE00] Mike Perkowitz and Oren Etzioni. Adaptive Web sites. *Communications of the ACM*, 43(8):152–158, August 2000.

[Pea02] PeakSoft Corporation. PeakJet 2000 web page. `http://www.peaksoft.com/peakjet2.html`, 2002.

[Pel96] Jeff Pelline. Accelerators cause headaches. *c|net News.com*, December 1996. `http://news.cnet.com/news/0,10000,0-1003-200-314937,00.html`.

[Per02] Persistence Software, Inc. Persistence home page. `http://www.persistence.com/`, 2002.

[PF00] Sanjoy Paul and Zongming Fei. Distributed caching with centralized control. In *Proceedings of the Fifth International Web Caching and Content Delivery Workshop (WCW'00)*, Lisbon, Portugal, May 2000.

[PK98] Venkata N. Padmanabhan and Randy H. Katz. TCP fast start: A technique for speeding up Web transfers. In *Proceedings of IEEE Globecom*, pages 41–46, Sydney, Australia, November 1998.

[PM94] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP latency. In *Proceedings of the Second International World Wide Web Conference: Mosaic and the Web*, pages 995–1005, Chicago, IL, October 1994.

[PM96] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve World Wide Web latency. *Computer Communication Review*, 26(3):22–36, July 1996. Proceedings of SIGCOMM '96.

[PM99] Themistoklis Palpanas and Alberto Mendelzon. Web Prefetching Using Partial Match Prediction. In *Proceedings of the Fourth International Web Caching Workshop (WCW99)*, San Diego, CA, March 1999. Work in progress.

[PMB96] Michael Pazzani, Jack Muramatsu, and Daniel Billsus. Syskill & Webert: Identifying interesting Web sites. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, 1996.

[Por97] Martin F. Porter. An algorithm for suffix stripping. In Karen Sparck Jones and Peter Willet, editors, *Readings in Information Retrieval*. Morgan Kaufmann, San Francisco, 1997. Originally published in *Program*, 14(3):130-137 (1980).

[PP97] James E. Pitkow and Peter L. Pirolli. Life, death, and lawfulness on the electronic frontier. In *ACM Conference on Human Factors in Computing Systems*, Atlanta, GA, March 1997.

[PP99a] Peter L. Pirolli and James E. Pitkow. Distributions of surfers' paths through the World Wide Web: Empirical characterization. *World Wide Web*, 2:29–45, 1999.

[PP99b] James E. Pitkow and Peter L. Pirolli. Mining longest repeated subsequences to predict World Wide Web surfing. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[PPR96] Peter Pirolli, James E. Pitkow, and Ramana Rao. Silk from a sow's ear: Extracting usable structures from the Web. In *Proceedings of CHI '96: Human factors in computing systems*, Vancouver, B.C., Canada, April 1996. ACM Press.

[PQ00] Venkata N. Padmanabhan and Lili Qui. The content and access dynamics of a busy Web site: Findings and implications. In *Proceedings of ACM SIGCOMM*, pages 111–123, Stockholm, Sweden, August 2000.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[RBP98] Judith Ramsay, Alessandro Barbesi, and Jenny Preece. A psychological investigation of long retrieval times on the World Wide Web. *Interacting with Computers*, 10:77–86, 1998.

[RCG98] Michael Rabinovich, Jeff Chase, and Syan Gadde. Not all hits are created equal: Cooperative proxy caching over a wide area network. In *Proceedings of the Third International WWW Caching Workshop*, Manchester, England, June 1998.

[RD99] Jean-David Ruvini and Christophe Dony. Learning users habits: The APE project. In *Proceedings of the IJCAI'99 Workshop on Learning About Users*, pages 65–73, Stockholm, Sweden, July 1999.

[RGM00] Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden Web. Technical Report 2000-36, Computer Science Dept., Stanford University, December 2000.

[RID00] Daniela Rosu, Arun Iyengar, and Daniel Dias. Hint-based acceleration of Web proxy caches. In *Proceedings of the 19th IEEE International Performance, Computing, and Communications Conference (IPCCC 2000)*, Phoenix, AZ, February 2000.

[Riv92] Ronald Rivest. The MD5 message-digest algorithm. RFC 1321, `http://ftp.isi.edu/in-notes/rfc1321.txt`, April 1992.

[RM99] Jason Rennie and Andrew McCallum. Efficient Web spidering with reinforcement learning. In *In Proceedings of the Sixteenth International Conference on Machine Learning (ICML-99)*, 1999.

[Roa98] Chris Roast. Designing for delay in interactive information retrieval. *Interacting with Computers*, 10:87–104, 1998.

[Rou02] Alex Rousskov. Web Polygraph: Proxy performance benchmark. Available at `http://www.web-polygraph.org/`, 2002.

[RS98] Alex Rousskov and Valery Soloviev. On performance of caching proxies. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98/PERFORMANCE '98)*, pages 272–273, Madison, WI, June 1998.

[RS02] Michael Rabinovich and Oliver Spatscheck. *Web Caching and Replication*. Addison-Wesley, New York, 2002.

[RW98] Alex Rousskov and Duane Wessels. Cache digests. *Computer Networks and ISDN Systems*, 30(22–23):2155–2168, November 1998.

[RW00] Alex Rousskov and Duane Wessels. The third cache-off. Raw data and independent analysis at `http://www.measurement-factory.com/results/`, October 2000.

[RWW01] Alex Rousskov, Matthew Weaver, and Duane Wessels. The fourth cache-off. Raw data and independent analysis at `http://www.measurement-factory.com/results/`, December 2001.

[Sar98] Robert G. Sargent. Verification and validation of simulation models. In D. J. Medeiros, E. F. Watson, J. S. Carson, and M. S. Manivannan, editors, *Proceedings of the Winter Simulation Conference*, 1998.

[Sar00] Ramesh R. Sarukkai. Link prediction and path analysis using Markov chains. In *Proceedings of the Ninth International World Wide Web Conference*, Amsterdam, May 2000.

[Sat02] Yutaka Sato. DeleGate home page. Online at `http://www.delegate.org/`, 2002.

[SB98] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523, 1998.

[SCJO01] F. Donelson Smith, Felix Hernandez Campos, Kevin Jeffay, and David Ott. What TCP/IP protocol headers can tell us about the Web. In *Proceedings of the ACM SIGMETRICS/Performance Conference on Measurement and Modeling of Computer Systems*, pages 245–256, 2001.

[SE95] Erik Selberg and Oren Etzioni. Multi-service search and comparison using the MetaCrawler. In *Proceedings of the Fourth International World Wide Web Conference*, Boston, MA, December 1995.

[SE97] Erik Selberg and Oren Etzioni. The MetaCrawler architecture for resource aggregation on the Web. *IEEE Expert*, 12(1):8–14, Jan/Feb 1997.

[SFBL97] Robert E. Schapire, Yoav Freund, Peter Bartlett, and Wee Sun Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. In *Proc. 14th International Conference on Machine Learning*, pages 322–330. Morgan Kaufmann, 1997.

[SH02] Rituparna Sen and Mark H. Hansen. Predicting a Web user's next request based on log data. To appear in the *Journal of Computational and Graphical Statistics*. Available from `http://cm.bell-labs.com/who/cocteau/papers/`, 2002.

[SKS98] Stuart Schechter, Murali Krishnan, and Michael D. Smith. Using path profiles to predict HTTP requests. *Computer Networks and ISDN Systems*, 30:457–467, 1998. Proceedings of the Seventh International World Wide Web Conference.

[SM83] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw Hill, New York, 1983.

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Spe97] Ellen Spertus. Parasite: Mining structural information on the Web. In *Proceedings of the Sixth International World Wide Web Conference*, Santa Clara, CA, April 1997.

[Spe02] Speedera Networks, Inc. Speedera home page. `http://www.speedera.com/`, 2002.

[Spi02] SpiderSoftware, Inc. Spidersoftware home page. `http://www.spidersoftware.com/`, 2002.

[Spr99] Tom Spring. Search engines gang up on Microsoft. *PC World*, November 1999. Available online at `http://www.pcworld.com/pcwtoday/article/0,1510,13760,00.html`.

[SR00] N. Swaminathan and S. V. Raghavan. Intelligent prefetching in WWW using client behavior characterization. In *Proceedings of the Eighth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2000.

[SSM87] Glenn Shafer, Prakash P. Shenoy, and Khaled Mellouli. Propagating belief functions in qualitative Markov trees. *International Journal of Approximate Reasoning*, 1(4):349–400, 1987.

[SSS99] Elizabeth Shriver, Christopher Small, and Keith Smith. Why does file system prefetching work?. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 71–83, Monterey, CA, June 1999.

[SSV97] Peter Scheuermann, Junho Shim, and Radek Vingralek. A case for delay-conscious caching of Web documents. In *Proceedings of the Sixth International World Wide Web Conference*, Santa Clara, CA, April 1997.

[SSV98] Junho Shim, Peter Scheuermann, and Radek Vingralek. A unified algorithm for cache replacement and consistency in Web proxy servers. In *Proceedings of the Workshop on the Web and Data Bases (WebDB98)*, 1998. Extended version will appear in Lecture Notes in Computer Science, Springer Verlag, 1998.

[SSV99] Junho Shim, Peter Scheuermann, and Radek Vingralek. Proxy cache design: Algorithms, implementation and performance. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):549–562, July/August 1999.

[STA01] Anees Shaikh, Renu Tewari, and Mukesh Agrawal. On the effectiveness of DNS-based server selection. In *Proceedings of IEEE INFOCOM*, Anchorage, AK, April 2001.

[Sto74] M. Stone. Cross-validation choices and assessment of statistical predictions. *Journal of the Royal Statistical Society, Series B*, 36:111–147, 1974.

[Sto78] M. Stone. Cross validation: A review. *Mathematische Operationsforschung und Statistik. Series Statistics*, 9(1):127–139, 1978.

[Sul99] Danny Sullivan. More evil than Dr. Evil? From the Search Engine Report, at `http://www.searchenginewatch.com/sereport/99/11-google.html`, November 1999.

[Sul02] Danny Sullivan. Search engine features for webmasters. From Search Engine Watch, at `http://www.searchenginewatch.com/webmasters/features.html`, March 2002.

[SW96] Jeffrey C. Schlimmer and Patricia Crane Wells. Quantitative results comparing three intelligent interfaces for information capture: A case study adding name information into an electronic personal organizer. *Journal of Artificial Intelligence Research*, 5:329–349, 1996.

[SW98] J. Santos and David Wetherall. Increasing effective link bandwidth by suppressing replicated data. In *Proceedings of the USENIX Annual Technical Conference*, pages 213–224, New Orleans, June 1998.

[SW00] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, August 2000.

[SYLZ00] Zhong Su, Qiang Yang, Ye Lu, and Hong-Jiang Zhang. WhatNext: A prediction system for Web request using n-gram sequence models. In *First International Conference on Web Information Systems and Engineering Conference*, Hong Kong, June 2000.

[Sys] System Performance Evaluation Cooperative (SPEC). Specweb96 benchmark. `http://www.specbench.org/osg/web96/`.

[TG97] Linda Tauscher and Saul Greenberg. How people revisit Web pages: Empirical findings and implications for the design of history systems. *International Journal of Human Computer Studies*, 47(1):97–138, 1997.

[THO96] Joe Touch, John Heidemann, and Katia Obraczka. Analysis of HTTP performance. Available at `http://www.isi.edu/lsam/publications/http-perf/`, August 1996.

[THVK99] Renu Tewari, Michael Hahlin, Harrick M. Vin, and Jonathan S. Kay. Design considerations for distributed caching on the Internet. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS)*, Austin, May 1999.

[TVDS98] Renu Tewari, Harrick M. Vin, Asit Dan, and Dinkar Sitaram. Resource-based caching for Web servers. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 1998.

[UCB01] UCB/LBNL/VINT. Network simulator ns. `http://www.isi.edu/nsnam/ns/`, 2001.

[Utg89] Paul E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4(2):161–186, 1989.

[VL97] Steven P. Vander Wiel and David J. Lilja. When caches aren't enough: Data prefetching techniques. *Computer*, 30(7), July 1997.

[VM02] Michael Vizard and Cathleen Moore. Akamai content to deliver. *InfoWorld*, March 22 2002. Online at `http://www.infoworld.com/articles/se/xml/02/03/25/020325seakamai.xml`.

[VR98] Vinod Valloppillil and Keith W. Ross. Cache Array Routing Protocol v1.0. Internet Draft draft-vinod-carp-v1-03.txt, February 1998.

[VW00] Paul Vixie and Duane Wessels. Hyper Text Caching Protocol (HTCP/0.0). RFC 2756, `http://ftp.isi.edu/in-notes/rfc2756.txt`, January 2000.

[WA97] Roland P. Wooster and Marc Abrams. Proxy caching that estimates page load delays. In *Proceedings of the Sixth International World Wide Web Conference*, pages 325–334, Santa Clara, CA, April 1997.

[Wan99] Jia Wang. A survey of Web caching schemes for the Internet. *ACM Computer Communication Review*, 29(5):36–46, October 1999.

[WAS+96] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal policies in network caches for World-Wide Web documents. In *Proceedings of ACM SIGCOMM*, pages 293–305, Stanford, CA, 1996. Revised March 1997.

[WC96] Zheng Wang and Jon Crowcroft. Prefetching in World-Wide Web. In *Proceedings of IEEE Globecom*, London, December 1996.

[WC97a] Duane Wessels and Kimberly Claffy. Application of Internet Cache Protocol (ICP), version 2. RFC 2187, `http://ftp.isi.edu/in-notes/rfc2187.txt`, September 1997.

[WC97b] Duane Wessels and Kimberly Claffy. Internet Cache Protocol (ICP), version 2. RFC 2186, `http://ftp.isi.edu/in-notes/rfc2186.txt`, September 1997.

[WC98] Zhe Wang and Pei Cao. Persistent connection behavior of popular browsers. `http://www.cs.wisc.edu/~cao/papers/persistent-connection.html`, 1998.

[Web02] Web 3000 Inc. NetSonic Internet Accelerator web page. `http://www.web3000.com/`, 2002.

[Wei01] Gary M. Weiss. Predicting telecommunication equipment failures from sequences of network alarms. In Willi Kloesgen and Jan Zytkow, editors, *Handbook of Data Mining and Knowledge Discovery*. Oxford University Press, 2001.

[Wes01] Duane Wessels. *Web Caching*. O'Reilly & Associates, 2001.

[Wes02] Duane Wessels. Squid Web proxy cache. Available at `http://www.squid-cache.org/`, 2002.

[WM00] Craig E. Wills and Mikhail Mikhailov. Studying the impact of more complete server information on Web caching. In *Proceedings of the Fifth International Web Caching and Content Delivery Workshop (WCW'00)*, Lisbon, Portugal, May 2000.

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, 1999. Second edition.

[Woo96] Roland Peter Wooster. Optimizing response time, rather than hit rates, of WWW proxy caches. Master's thesis, Virginia Polytechnic Institute, December 1996. `http://scholar.lib.vt.edu/theses/materials/public/etd-34131420119653540/etd-title.html`.

[Wor94] Kurt Worrell. Invalidation in large scale network object caches. Master's thesis, University of Colorado, Boulder, CO, 1994.

[Wor01] World Wide Web Consortium. Libwww — the w3c sample code library. `http://www.w3.org/Library/`, 2001.

[WVS+99] Alec Wolman, Geoffrey M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M. Levy. On the scale and performance of cooperative Web proxy caching. *Operating Systems Review*, 34(5):16–31, December 1999. Proceedings of the 17th Symposium on Operating Systems Principles.

[WWB96] Roland O. Wooster, Stephen Williams, and Patrick Brooks. HTTPDUMP: Network HTTP packet snooper. Working paper available at `http://www.cs.vt.edu/~chitra/work.html`, April 1996.

[XCa02] XCache Technologies. XCache home page. `http://www.xcache.com/`, 2002.

[Yah02] Yahoo!, Inc. Yahoo! `http://www.yahoo.com/`, 2002.

[YM96] Kenichi Yoshida and Hiroshi Motoda. Automated user modeling for intelligent interface. *International Journal of Human-Computer Interaction*, 8(3):237–258, 1996.

[Yos94] Kenichi Yoshida. User command prediction by graph-based induction. In *Proceedings of the Sixth International Conference on Tools with Artificial Intelligence*, pages 732–735, Los Alamitos, CA, November 1994. IEEE Computer Society Press.

[YSG02] Yiming Yang, Sean Slattery, and Rayid Ghani. A study of approaches to hypertext categorization. *Journal of Intelligent Information Systems*, 18(2/3):219–241, 2002.

[YZL01] Qiang Yang, Haining Henry Zhang, and Ian Tianyi Li. Mining Web logs for prediction models in WWW caching and prefetching. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'01)*, San Francisco, August 2001. Industry Applications Track.

[ZA01] Ingrid Zukerman and David W. Albrecht. Predictive statistical models for user modeling. *User Modeling and User-Adapted Interaction*, 11(1/2):5–18, 2001.

[ZAN99] Ingrid Zukerman, David W. Albrecht, and Ann E. Nicholson. Predicting users' requests on the WWW. In *Proceedings of the Seventh International Conference on User Modeling (UM-99)*, pages 275–284, Banff, Canada, June 1999.

[ZE98] Oren Zamir and Oren Etzioni. Web document clustering: A feasibility demonstration. In *Proceedings of the ACM/SIGIR Conference on Research and Development in Information Retrieval*, 1998.

[ZE99] Oren Zamir and Oren Etzioni. Grouper: a dynamic clustering interface to Web search results. In *Proceedings of the Eighth International World Wide Web Conference*, Toronto, Canada, May 1999.

[ZFJ97] Lixia Zhang, Sally Floyd, and Van Jacobson. Adaptive Web caching. In *Proceedings of the NLANR Web Cache Workshop*, Boulder, CO, June 1997.

[ZIRO99] Junbiao Zhang, Rauf Izmailov, Daniel Reininger, and Maximilian Ott. WebCASE: A simulation environment for web caching study. In *Proceedings of the Fourth International Web Caching Workshop (WCW99)*, San Diego, CA, March 1999.

[Zon99] The economic impacts of unacceptable Web site download speeds. White paper, Zona Research, 1999. Available from `http://www.zonaresearch.com/deliverables/white_papers/wp17/index.htm`.

[ZY01] Michael Zhen Zhang and Qiang Yang. Model-based predictive prefetching. In *Proceedings of the 2nd International Workshop on Management of Information on the Web — Web Data and Text Mining (MIW'01)*, Munich, Germany, September 2001.

Vita

Brian D. Davison

1969 Born in Paoli, Pennsylvania.

1987 Graduated from Twin Valley High School, Elverson, Pennsylvania.

1987-91 Attended Bucknell University, Lewisburg, Pennsylvania.

1991 B.S. in Computer Engineering, Bucknell University.

1991-2002 Graduate work in Computer Science, Rutgers, The State University of New Jersey, New Brunswick, New Jersey.

1991-1993 Teaching Assistant, Dept. of Computer Science, Rutgers University.

1994-2001 Research Assistant, Dept. of Computer Science, Rutgers University.

1995 M.S. in Computer Science, Rutgers, The State University of New Jersey.

1998 S.A. Macskassy, A. Banerjee, B.D. Davison, and H. Hirsh. Human performance on clustering Web pages: A preliminary study. *Proc. of the 4th Int'l Conf. on Knowledge Discovery and Data Mining*, 264-268.

1999 B.D. Davison. Simultaneous proxy evaluation. *Proc. of the 4th Int'l Web Caching Workshop*, 170-178.

2000 H. Hirsh, C. Basu, and B.D. Davison. Learning to personalize. *Communications of the ACM*, 43(8):102-106.

2000 B.D. Davison. Topical locality in the Web. *Proc. of the 23rd Annual Int'l Conf. on Research and Development in Information Retrieval*, 272-279.

2000 B.D. Davison and V. Liberatore. Pushing politely: Improving Web responsiveness one packet at a time. *Performance Evaluation Review*, 28(2):43-49.

2001 B.D. Davison. A Web caching primer. *IEEE Internet Computing*, 5(4):38-45.

2001 B.D. Davison. Assertion: Prefetching with GET is not good. *Proc. of the 6th Int'l Web Content Caching and Distribution Workshop*, 203-215.

2001-2002 Instructor, Dept. of Computer Science & Engineering, Lehigh University, Bethlehem, Pennsylvania.

2002 B.D. Davison. Predicting Web actions from HTML content. *Proc. of the 13th ACM Conf. on Hypertext and Hypermedia*, 159-168.

2002 B.D. Davison, C. Krishnan, and B. Wu. When does a hit = a miss? *Proc. of the 7th Int'l Workshop on Web Content Caching and Distribution.*