Optimization in Web Caching: Cache Management, Capacity Planning, and Content Naming

by

Terence P. Kelly

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2002

Doctoral Committee:

Professor Peter Honeyman, Chair
Professor Jeffrey K. MacKie-Mason
Professor Brian Noble
Professor Michael P. Wellman
Dr. Jim Gray, Microsoft Research

ABSTRACT

Optimization in Web Caching:
Cache Management, Capacity Planning,
and Content Naming

by

Terence P. Kelly

Chair: Peter Honeyman

Caching is fundamental to performance in distributed information retrieval systems such as the World Wide Web. This thesis introduces novel techniques for optimizing performance and cost-effectiveness in Web cache hierarchies.

When requests are served by nearby caches rather than distant servers, server loads and network traffic decrease and transactions are faster. Cache system design and management, however, face extraordinary challenges in loosely-organized environments like the Web, where the many components involved in content creation, transport, and consumption are owned and administered by different entities. Such environments call for *decentralized* algorithms in which stakeholders act on local information and private preferences.

In this thesis I consider problems of optimally designing new Web cache hierarchies and optimizing existing ones. The methods I introduce span the Web from point of content creation to point of consumption: I quantify the impact of content-naming practices on cache performance; present techniques for variable-quality-of-service cache management; describe how a decentralized algorithm can compute economically-optimal cache sizes in a branching two-level cache hierarchy; and introduce a new protocol extension that eliminates redundant data transfers and allows "dynamic" content to be cached consistently.

To evaluate several of my new methods, I conducted trace-driven simulations on an unprecedented scale. This in turn required novel workload measurement methods and effi-

cient new characterization and simulation techniques. The performance benefits of my proposed protocol extension are evaluated using two extraordinarily large and detailed workload traces collected in a traditional corporate network environment and an unconventional thin-client system.

My empirical research follows a simple but powerful paradigm: measure on a large scale an important production environment's exogenous workload; identify performance bounds inherent in the workload, independent of the system currently serving it; identify gaps between actual and potential performance in the environment under study; and finally devise ways to close these gaps through component modifications or through improved inter-component integration. This approach may be applicable to a wide range of Web services as they mature.

© Terence P. Kelly 2002
All Rights Reserved

To all my teachers . . .

from school, from my family, and amongst my friends.

ACKNOWLEDGMENTS

This dissertation includes results from joint work with Jeff MacKie-Mason, Sugih Jamin, Yee Man Chan, Jonathan Womer, Daniel Reeves, and Jeff Mogul [85–89]. Chapter 3 is based on a collaboration with MacKie-Mason, Jamin, Chan, and Womer supervised by Michael Wellman under the auspices of the Michigan Adaptive Resource Exchange (MARX) project. The algorithm described in Section 4.5 is joint work with Reeves and the analysis of content naming in Chapter 7 is joint work with Mogul. Professors Michael Wellman and Peter Honeyman contributed substantially to my work by providing thoughtful, patient guidance; consistent, generous funding; and formidable computational resources.

Most of my early research was supported by DARPA/ITO grant F30602-97-1-0228, from the Information Survivability Program. RAND Corporation, Microsoft Research, and WebTV also provided substantial support; I respectively thank Bob Anderson, Eric Horvitz, and Stuart Ozer for stimulating internships at these organizations. The University of Michigan (U-M) Center for Information Technology Integration (CITI) funded much of my later work and a U-M Rackham Predoctoral Fellowship supported my final round of work; I am grateful to Peter Honeyman of CITI and the Rackham Graduate School for making it possible for me to finish. Finally, the Council on Library and Information Resources provided additional funding through an A.R. Zipf Fellowship.

My empirical work required vast and breathtakingly expensive computational resources. Mike Wellman repeatedly purchased hardware upgrades to support my early work with NLANR data. The far larger WebTV trace that supported my most original investigations wouldn't have left Mountain View without the "half" machine designed, assembled and configured by Peter Honeyman, Jim Rees, Chuck Lever and Brad Quinn of CITI. A marvel of cost-effective storage technology, the half machine earned the awe and respect of all who saw it; I took particular joy in showing it off to a manager who had recently paid eight times its cost for a storage device of comparable capacity. Tom Hacker and Abhijit Bose

of the University of Michigan Center for Parallel Computing (CPC) repeatedly went above and beyond the call of duty to support my computational work, and more than once winked at my excessive use of CPC resources. Jeff Kopmanis of the U-M Advanced Technology Lab (ATL) patiently performed numerous hardware upgrades and software reconfigurations to support my research over the years, and Professor John Laird generously provided the ATL community with the very capable server that Jeff upgraded. Geoff Voelker of U.C. San Diego provided crucial additional firepower when local resources proved insufficient. Jeff Mogul of Compaq arranged to loan U-M a $200,000 server for our joint work, along with a donation of $30,000 in supporting hardware; these resources made Chapter 7 possible. Glenn Cooper of Compaq customized the loaner machine's configuration to my needs. Finally, over the years I have relied heavily on the U-M Computer Aided Engineering Network (CAEN) for nearly all of my software development and typesetting. CAEN provides its users with an extraordinarily uniform and well-administered computing environment and is remarkably responsive to user concerns. I'm particularly grateful to CAEN Assistant Director Amadi Nwankpa for his outstandingly thoughtful replies to numerous questions large and small. Thanks to folks like Amadi, CAEN is one of the few computing environments where mail to the help desk can yield replies that correct one's mathematical errors.

The trace data I used in my early simulations were collected by the National Laboratory for Applied Network Research under National Science Foundation grants NCR-9616602 and NCR-9521745. James Pitkow of Xerox PARC gave me a Georgia Tech Web client trace that he and Lara Catledge collected, and Boston University's CS Department provided the client trace used in Section 6.1.2. Jeff Mogul of Compaq Corporation supplied a proxy trace nearly as large and detailed as the WebTV data for our joint work. Jeff Ogden of Merit Networks provided bandwidth pricing data and JoElla Coles of ITD supplied the LAN cost data of Section 4.3.

The WebTV trace that made possible my most original empirical work deserves special mention. Literally dozens of WebTV personnel helped to collect the anonymized trace described in Chapter 6. My manager, Stuart Ozer, lent resources, expertise, and most importantly his endorsement to the project. Arnold de Leon provided a large computer for trace collection and with Jeff Allen worked out a thousand operational details. Jay Logue modified WebTV's proxy software to log additional data and serve documents pre-expired.

Todd Stansell and Jonathan Tourtellot assembled a 1.2-TB storage array from spare parts to support my project and Brad Whisler managed day-to-day trace logging. Paul Roy allowed us to run our measurements on a large fraction of WebTV's production system after Andrea Chien conducted successful preliminary tests on smaller client populations. Jake Brutlag helped me to understand WebTV's complex service architecture and to document WebTV's proxy log format. WebTV client engineers explained the various client devices and the rationale behind their design; David Surovell, Scott Sanders, Monique Barbanson, David Conroy and Wiltse Carpenter were particularly helpful. None of my work at WebTV would have happened without Eric Horvitz of Microsoft Research, who introduced me to the extraordinary research potential of WebTV.

I could not have developed the ideas in this thesis without numerous intellectual contributions from U-M and beyond. Daniel Reeves first demonstrated that efficient single-pass simulation of stack algorithms is possible, thereby inspiring much of my work on the topic and making possible the results of Section 4.5. Jeff Mogul's ground-breaking work on the HTTP namespace and duplicate suppression laid the foundation for the collaboration that yielded Chapter 7. Mike Wellman, Peter Honeyman and Jeff MacKie-Mason helped me to bring together ideas from both Economics and Computer Science in a new way, following precedents established many years ago by Jim Gray. My research into the economics of distributed storage has furthermore profited from lengthy discussions with Jonathan Womer, Bill Walsh and Yee Man Chan. Chan, Reeves, Mogul and Martin Arlitt participated in *N*-version testing of my cache simulators and stack distance implementations, thereby helping to validate the correctness of my results.

Mikhail Mikhailov explained several puzzling phenomena that I observed in trace data, reassuring me more than once that the problem was on the Web rather than in my head, my data, or my analysis code. Mikhailov furthermore performed several tabulations of his own data sets on my behalf and read drafts of one of my publications *twice*, offering valuable feedback each time. Over the years many others have greatly improved the research included in this thesis through their thoughtful reviews, including Mike Wellman, Jeff MacKie-Mason, Peter Honeyman, Jim Gray, John Dilley, Martin Arlitt, Jake Brutlag, Flavia Peligrinelli Ribeiro, Bill Walsh, and Daniel Reeves.

I'd like to extend special thanks to the advisors who have guided my development as a researcher over the past six years. Early on, Ken Steiglitz and David Dobkin encouraged

me along the path that ultimately led to grad school. Kai Nagel passed along to me not only the science of automotive traffic modeling but also the mentality of statistical physics; his influence is evident in the results of Section 4.2.2. Kai furthermore drilled into my head the notion that "computational science means working at the limits of available resources while expanding those limits," a maxim that has cost my institution and my industrial partners a small fortune over the years. Mike Wellman educated me in topics ranging from game theory to graph search algorithms, and through the example of his own papers showed me the art of scientific writing. Finally, Peter Honeyman told me how things really work in the world of research. Among his many invaluable lessons are several that transcend the narrowly academic, e.g., the importance of promoting brand awareness through logo-bearing adhesives and novelty items. More than an advisor, Peter has been a true mentor and indeed a spiritual guide. When confronted with a vexing decision or moral dilemma I simply ask myself, "what would honey do?" and the right path becomes clear. Thanks, Peter.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF TABLES x

LIST OF FIGURES xi

CHAPTERS

1 Introduction 1
- 1.1 Market-Based Solutions 4
- 1.2 Optimal Capacity Planning 5
- 1.3 Cache Analysis and Simulation 6
- 1.4 Workload Measurement 7
- 1.5 Content-Naming and Performance 8
- 1.6 Summary 9

2 Caching Problems 11
- 2.1 Cache Sizing 14
- 2.2 Cache Installation 16
- 2.3 Removal Policies 17
- 2.4 Redundant Transfers 18
- 2.5 Cache Consistency 18

3 Preference-Sensitive Removal Policies 20
- 3.1 Value-Sensitive Caching 21
  - 3.1.1 Value Model 22
  - 3.1.2 Value-Sensitive Removal Policies 23
- 3.2 Prediction vs. Value Sensitivity 24
- 3.3 Experiments 29
  - 3.3.1 Heterogeneous Valuations 30
  - 3.3.2 Homogeneous Valuations 34
- 3.4 Limits to Biased LFU 36
- 3.5 Incentives 39

4 Optimal Cache Sizing . . . 41
4.1 Monetary Costs and Benefits . . . 42
4.2 A Simple Hierarchical Caching Model . . . 43
4.2.1 Centralized Optimization . . . 44
4.2.2 Decentralized Optimization . . . 46
4.3 Cost Calculations . . . 47
4.4 A Detailed Model of Single Caches . . . 49
4.5 Fast Simultaneous Simulation . . . 52
4.6 Numerical Results . . . 55
4.7 Discussion . . . 58

5 Cache Analysis, Traces, and Simulation . . . 61
5.1 Analytic Modeling . . . 61
5.2 Trace-Collection Methods and Available Traces . . . 63
5.2.1 Requirements . . . 63
5.2.2 Server Logs . . . 64
5.2.3 Proxy Logs and Sniffers . . . 64
5.2.4 Instrumented Clients . . . 65
5.2.5 Publicly-Available Traces . . . 66
5.2.6 Summary . . . 67
5.3 Efficient General Simulation . . . 68

6 Workload Measurement . . . 72
6.1 Trace Collection . . . 73
6.1.1 Trace Characteristics . . . 81
6.1.2 Trace Representativeness . . . 84
6.2 Inherent Performance Bounds . . . 86
6.2.1 System-Wide Miss Rates . . . 86
6.2.2 Browser Caches . . . 88
6.2.3 Filter Effects in Cache Hierarchies . . . 90
6.3 Discussion . . . 92

7 Content Naming and Redundant Transfers . . . 96
7.1 Related Work . . . 98
7.1.1 Resource Modification . . . 98
7.1.2 Mirroring . . . 99
7.1.3 Duplicate Suppression . . . 100
7.1.4 Harmful Practices . . . 100
7.1.5 Summary . . . 101
7.2 Prevalence of Aliasing . . . 101
7.2.1 Aliasing and Response Attributes . . . 105
7.2.2 Causes of Aliasing . . . 107
7.3 Performance Implications of URL-Indexed Caching . . . 108
7.3.1 Abstract Cache Models . . . 108
7.3.2 Simulation Results . . . 111

7.4 Explaining Redundant Transfers . . . 113
7.5 Avoiding Redundant Transfers . . . 114
7.5.1 Latency Analysis . . . 117
7.5.2 Security Issues . . . 118
7.5.3 Similar Proposals . . . 120
7.5.4 Alternatives to DTD . . . 120
7.6 Discussion . . . 121

8 Summary and Future Work . . . 123
8.1 Summary of Contributions . . . 124
8.2 Future Work . . . 126

BIBLIOGRAPHY . . . 129

LIST OF TABLES

Table

3.1 Notation of Chapter 3. 21
3.2 Summary statistics on three request streams after filtering out uncachable documents. 27
3.3 Traces recorded at six NLANR sites, 1–28 March 1999. 31
4.1 Notation of Section 4.2. 44
4.2 Merit Networks Inc. prices of Internet connectivity for commercial and educational customers in U.S. dollars. 48
4.3 LAN bandwidth costs of 10 Mbps shared Ethernet at the University of Michigan. Data courtesy JoElla Coles of ITD. 49
4.4 Notation of Section 4.4. 50
4.5 Traces derived from access logs recorded at six NLANR sites, 1–28 March 1999. Run times shown are wall-clock times to compute given quantities, in seconds. The run times sum to under four hours, ten minutes. 56
6.1 WebTV client devices. 74
6.2 WebTV trace summary statistics. 78
6.3 Traces used in Web and file system research. "Objects" refers to distinct payloads, or to URLs in traces that do not distinguish payloads. 79
6.4 WebTV trace Zipf parameters. 80
6.5 Response sizes in various traces. 84
6.6 MIME type distribution of WebTV trace. 85
6.7 Percent relative improvement (R.I.) over current hit rates. 90
7.1 WebTV reduced trace aliasing statistics. 102
7.2 Prevalence of aliasing by MIME type in WebTV trace. 105
7.3 Causes of aliasing. 107
7.4 URL-indexed and compulsory miss rates and % of URL-indexed payload transfers that are redundant. 112
7.5 Compaq reduced trace summary statistics. 113

LIST OF FIGURES

Figure

2.1 A branching multi-level storage hierarchy. Requests from browsers are filtered through dedicated and shared caches on their way to origin servers. Point A is a candidate location for a shared cache, considered in the text. . . 12
2.2 Cache cost functions. 15
3.1 The swLFU algorithm. 24
3.2 Quality-of-service (byte hit rate) as a function of W_u for SV trace summarized in Table 3.2. 25
3.3 Workload characteristics: CDF of LRU stack distances of hits (left) and Zipf-like popularity distribution (right). 26
3.4 Left: Hot set drift at six NLANR sites, March 1999. Right: windowed hit rates for LRU and LFU at two cache sizes, August 1998 SV trace. 28
3.5 Byte hit rate at cache sizes 1 MB–8 GB for LRU and four LFU variants, August 1998 NLANR SV trace. 30
3.6 VHR as function of cache size for two A-swLFU variants and GD-Size. LRU is also shown at larger cache sizes for comparison. Note that vertical scales do not begin at zero. 32
3.7 Tuned perfect & in-cache A-swLFU, in-cache GDSF, and GD-Size. March 1999 UC trace. The results shown required roughly 60 CPU days to compute using the parallel simulator of Section 5.3. 33
3.8 Cost = size case: byte hit rates as function of cache size for GD-Size/LRU and four LFU variants: perfect vs. in-cache and K=10 aging vs. no aging. March 1999 NLANR traces. Note that vertical scales do not start at zero, and their upper limits vary. 35
3.9 Byte HR as function of aging parameter K for in-cache LFU (solid lines) and Perfect LFU (dashed lines) for various cache sizes. March 1999 SD trace. 36
3.10 Histogram of mean weighted values $\overline{V}_u$ for popular URLs in NLANR's Silicon Valley L3 cache request log of 26 August 1998 (a busy day at a busy site) for a particular random assignment of w_i values to clients. Other assignments of w_i yield qualitatively similar results. 37

3.11 Overlap among top k items in lists sorted on weighted and unweighted criteria. Reference counts n_{iu} are from the NLANR SV log of 17 March 1999. 38
4.1 Two-level caching hierarchy of Section 4.2. 43
4.2 Recovering priority depth. In this example, document 1 has been referenced. We initialize an accumulator to the size of document 1 (in this example, 34) plus the sum of sizes of all documents in its right subtree (in this example, zero). We then walk up to the root. When we move from a right child to its parent (e.g., from document 1 to document 6) we do nothing. However when we move from a left child to its parent (e.g., from document 6 to document 5) we add to the accumulator the size of the parent (75) and the sum of sizes of all documents in the parent's right subtree (124). When we reach the root, the accumulator contains the sum of the sizes of the referenced document and all higher-priority documents, i.e., the priority depth of the referenced document (233). 54
4.3 Frequency distribution (top) and cumulative distribution (bottom) of LRU stack distances in six traces. Compare these data with Table 10 and Figure 8 of Arlitt & Jin [13]; temporal locality is far weaker in our network cache traces than in their very large server workload. 57
4.4 Exact hit rates (top) and byte hit rates (bottom) as function of cache size for six large traces, LRU removal. Fast simultaneous simulation method yields correct results only for cache sizes $\geq$ largest object size in a trace; smaller cache sizes not shown. 59
5.1 RAM requirements of current multi-threaded simulator as function of number of active worker threads (number of processors used) for the six NLANR traces of Table 3.3. 70
6.1 Distribution of NTP parameters during data collection. 76
6.2 Hourly request volume by GMT time, 1-hour time bins. 76
6.3 Percentage of IMS requests from diskful (left) and diskless (right) clients, 1-hour windows. Note that vertical scales differ. 77
6.4 Zipf-like reference counts of URLs and reply bodies. 80
6.5 Left: CDF of Zipf α across client sub-populations. Right: CDF of R^2. . . . 81
6.6 Concentration of references. 82
6.7 Distribution of references/client and bytes/client. 83
6.8 Frequency histograms of $\log_2$-transformed data. *Left:* references/client. *Right:* stack distances in BPS sample. 83
6.9 Distribution of inter-reference intervals in WebTV (left) and Boston University (right) client traces. 85
6.10 Percentage of replies containing new documents. 87
6.11 Ratio of distinct documents to transactions vs. number of transactions examined. 87
6.12 Distribution of maximal browser hit rates (left) and effectively infinite browser cache sizes (right). 88

6.13 Aggregate WebTV client success functions. 90
6.14 Browser success function and proxy success functions for several different browser cache sizes. 92
7.1 Left: Fraction of payloads aliased versus trace length. Right: Fraction of transactions carrying aliased payloads versus trace length. 103
7.2 Left: CDF of payload and URL degrees. Center: CDF of transactions by degree of URL & payload involved. Right: CDF of bytes transferred by degree of payload involved. 103
7.3 CDFs of change and alias ratios. 104
7.4 CDFs by payload size for all payloads (top row) and three popular MIME types. Solid lines indicate aliased payloads, transactions involving aliased payloads, and aliased bytes transferred; dashed lines non-aliased. All horizontal scales are identical and show payload size in bytes. 106
7.5 URL-indexed and DTD caches. 110

CHAPTER 1

Introduction

Caching is essential to distributed information retrieval systems such as the World Wide Web, helping to reduce network traffic, server load, and client latency. In order to scale, systems like the Web must exploit caching to the extent permitted by offered workload. Not surprisingly, caching is widespread on the Web today, but by any measure it is far from optimal. The design and operation of components such as browser and proxy caches, and the protocols that govern their interactions, often serve the Web's exogenous workload inefficiently. The roots of this problem are partly historical. Web technologies evolved into their present form on "Internet time," during a period of intense commercial competition in the 1990s when time-to-market pressures forced hasty deployments of poor designs. Another factor is the decomposition of the Web into independently designed yet interoperable components, e.g., servers, proxies and browsers. Decomposition has permitted rapid component evolution—server software today, for instance, is far more capable than that of the early Web—but it has led to a component-centric view of performance that often ignores system-level performance and interactions among components. Finally, the preferences of system "stakeholders" and the monetary costs of relevant technologies rarely inform cache design decisions or run-time algorithms in principled ways. This dissertation addresses these problems by describing ways of optimizing the design, operation, and inter-operation of Web caches in terms of both conventional performance metrics and novel measures involving monetary costs and user preferences.

Mainstream Web researchers and practitioners have long recognized that bottom-line concerns ultimately motivate caching. Wessels' recent book *Web Caching*, for instance, opens with the question, "Why cache the Web? The short answer is that caching saves

money" [165]. However the widespread vague recognition that "money is important" has not translated into widespread adoption of economically principled design methods or preference-sensitive run-time behavior; Wessels' discussion of proxy cache sizing, for instance, says nothing about the tradeoff between the monetary costs of cache misses and storage. Recognition that economic considerations should predominate in design decisions is growing slowly, driven mainly by electronic commerce:

> Quality of service of e-commerce sites has been usually managed by the allocation of resources such as processors, disks, and network bandwidth, and by tracking conventional performance metrics such as response time, throughput, and availability. However, the metrics that are of utmost importance to the management of a Web store are revenue and profits. Thus, the resource management schemes for e-commerce servers should be geared towards optimizing business metrics as opposed to conventional performance metrics [110].

Similar sentiments are echoed by van Moorsel [157], but this perspective remains the exception rather than the rule. This dissertation validates my thesis: *that economic perspectives can help us to enhance both the performance and cost-effectiveness of Web caching systems*. As we shall see, some of the novel principles and methods that enable us to do so have precedents in the literature on database capacity planning and the literature on processor memory hierarchies. By extending, generalizing, re-interpreting and complementing existing methods we can optimize performance metrics appropriate to the age of electronic commerce.

Divide-and-conquer is an essential strategy in distributed system design and the Web could not exist without it. However excessive focus on Web components can divert attention from the fundamentals of exogenous workload and the question of how best to serve it. This is especially true of intermediate components such as caching proxies, which are shielded from the raw workloads entering the Web at client and server ends. Even within a company like Microsoft, whose product line—FrontPage, IIS, ProxyServer, IE—spans the Web from point of content creation to point of consumption, component product teams often regard system-level performance as someone else's problem. This dissertation demonstrates that researchers, implementors, and administrators must shift from a component-centric perspective to a system-level focus now that Web technologies and

workloads have matured. It describes previously unknown interactions across Web components that can impair the performance of Web cache hierarchies; these subtle, non-local effects call for solutions that transcend the narrow focus of today's component designers and product groups. Furthermore, this dissertation quantifies the substantial degree of waste in existing Web cache hierarchies by comparing their performance with upper bounds inherent in offered workload. More importantly, it describes a simple protocol extension capable of closing the gap between actual and potential performance, a protocol extension that can enable independently-developed components to achieve the same performance as a well-integrated single-vendor product line.

One contribution of this dissertation is to formulate important Web cache design problems as proper *optimization* problems that explicitly incorporate technology costs and system user preferences. My results complement the existing capacity planning literature by identifying cases where capacity expansion beyond minimal system requirements yields lower overall operating costs; similarly, they extend and generalize the existing Web caching literature with techniques for adapting to user preferences. For problems involving large-scale systems, e.g., branching cache hierarchies, I consider *decentralized* techniques involving only local computations on local information, and compare the solutions we obtain from such methods with those of centralized approaches.

Another contribution is a very large scale empirical exploration of Web workloads. I have obtained extraordinarily large and detailed data sets from Compaq Corporation and WebTV Networks; I collected the latter data set using an innovative measurement technique. To analyze these data I have developed scalable methods for trace-driven simulation and workload characterization. These methods allowed me to demonstrate that unnecessary cache misses occur frequently in a production system currently serving over a million paying customers. In an effort to explain this problem, I quantified for the first time the performance penalty that arises from interactions between conventional cache management algorithms and the exogenous inputs entering the Web from opposite ends: content *naming* at the server end, and content *access* patterns at the client end.

The remainder of this section surveys the most important facets of my research, relates them to existing literature, and summarizes my contributions.

1.1 Market-Based Solutions

Market-based solutions are appealing in distributed computing systems because in some models they compute optimal resource allocations in a decentralized (and hence scalable) manner. Not surprisingly, the use of price systems and market-like schemes for computer resource allocation has been proposed sporadically for over three decades [152]. One design method involves building "computational market economies" directly inspired by economic theory. Examples of this approach include the SPAWN distributed computing system [162] and Kurose & Simha's file allocation scheme [98]; Wellman provides a review of "market-oriented programming" and its application to distributed resource allocation [163]. An alternative approach, evident in much of my work, is to generalize well-known resource-allocation algorithms in economically meaningful ways. I have shown, for instance, that biased cache replacement policies can increase aggregate system value by diverting storage space to stakeholders who value cache hits most.

First-generation Web architectures provided only "best effort" service, in the sense that they were insensitive to the service quality preferences of system users (e.g., content providers). A mature, fully commercialized Web will provide variable quality of service (QoS), delivering highest performance to users who value performance most; content delivery networks (CDNs) such as Akamai are early examples of this trend. While preliminary investigations of variable-QoS Web *servers* have appeared [3, 29, 133, 161], little complementary research exists on variable-QoS Web *caching*. This is surprising, because storage space in shared Web caches is a scarce resource that may be diverted to serve some system users at the expense of others, and therefore such caches are obvious loci for variable-QoS mechanisms. My investigations of preference-sensitive caching have yielded removal policies that are tailored to observed regularities in Web cache workloads and that also account for heterogeneous QoS demand. I have shown that biased removal policies deliver higher overall value to system users than conventional replacement policies when used to maximize value to content providers. I have also shown that the problem of maximizing value to clients can be more difficult, and have identified interactions between client preferences and request patterns that can cause the additional difficulty. Chapter 3 presents these results.

1.2 Optimal Capacity Planning

Careful capacity planning and resource allocation within emerging Web caching systems becomes increasingly important as their size grows: Calculating precisely the resource requirements of an isolated proxy might not be worth the bother, but deployments on the scale of Akamai and WebTV will likely reward exact reckoning with substantial savings. Furthermore caching entails resource tradeoffs that must be made wisely as we move toward a future of ubiquitous ultra-thin clients, e.g., wireless palmtop browsers, where no resource is cheap or plentiful: Huge losses will result if millions of devices each waste a dollar. Surprisingly, recent literature on Web caching and Web capacity planning is largely silent on the problem of serving offered workload at minimal cost. For instance, the obvious tradeoff between bandwidth and cache storage costs is rarely mentioned in Web research, despite the fact that data-engineering folklore has provided straightforward approaches to this problem for over a decade [71]. I have extended these well-known rules of thumb to a practical, general, exact method for computing the optimal size of a single cache based on workload and the costs of memory and cache misses. This method relies on a highly efficient, novel single-pass simulation technique that Daniel Reeves and I developed.

A single-cache optimization method is not sufficient for system-level optimization because Web caches are deployed in branching hierarchies: Many browsers share a common proxy, and many proxies may share a common backbone-network cache. Design decisions at one node influence the workload reaching other nodes, and this kind of interaction might require that we consider the entire system simultaneously to compute global optima. Furthermore, the Web differs from other multi-layered storage systems in that nodes are geographically dispersed and administered by separate organizations. A capacity planning method that requires a "central planner" to collect and process information from all nodes may be infeasible for reasons of scalability, reliability, and privacy. Decentralized resource allocation schemes wherein nodes compute local allocations based solely on local information are far more desirable, provided that they compute the same allocations as an optimal central planner. I have shown that under certain conditions optimal cache sizes may be computed in a large two-level branching cache hierarchy via a greedy local algorithm. Chapter 4 describes my optimal capacity planning results.

1.3 Cache Analysis and Simulation

Purely analytic approaches to cache evaluation often yield powerful results in the special case where cache entries are of uniform size and miss penalties are uniform. However Web object sizes and miss costs can be non-uniform, and this both complicates the analysis and diminishes the practical value of analytic results. We must therefore often resort to numerical methods (cache simulation) when evaluating design alternatives.

Simulation is undoubtedly necessary and often straightforward but never easy when done well. Severe scalability challenges confront Web researchers who analyze workloads or evaluate new designs empirically: The Web is growing rapidly, and the research community's expectations for the scale of empirical investigations have risen correspondingly. Therefore we require efficient and scalable algorithms to support trace-driven simulation and analysis of large workloads. However, simulation methodology does not figure prominently in the Web caching literature, despite the fact that research projects are sometimes hampered by inadequate simulators. To cite one well-known example, Cao & Irani's empirical evaluation of their GreedyDual-Size cache removal policy was impaired by a simulator capable of processing only two million requests at a time, whereas their largest trace contained 24 million requests [42, page 196].

I have developed a general-purpose parallel cache simulator capable of fully exploiting available CPUs and RAM on shared-memory architectures. Furthermore, Daniel Reeves and I devised an efficient algorithm that simultaneously computes arbitrarily-weighted hit rates at *all* cache sizes for a class of removal policies that includes LRU; our algorithm is also useful for analyzing temporal locality in request streams. An implementation is freely available and has been used by researchers in three countries. Although generalized for the special needs of the Web, our algorithm is closely related to techniques developed in the processor caching literature between the mid-1970s and early 1980s. This kind of algorithm, however, appears not to be widely known among Web researchers, and anecdotal evidence suggests that it outperforms less efficient methods in current use by a substantial margin. Martin Arlitt of HP Labs reports that my simple unoptimized implementation of the Reeves/Kelly algorithm computes LRU stack distances for a very large trace roughly six times faster than his own highly-optimized implementation of a fundamentally slower algorithm (19 hours vs. roughly 5 days).

Chapter 5 reviews the shortcomings of purely analytic evaluation methods, discusses deficiencies in existing Web traces and trace-collection methods, and describes the design of my parallel cache simulator. The parallel simulator is necessary in cases where the fast Reeves/Kelly single-pass simulation method of Section 4.5 is inapplicable.

1.4 Workload Measurement

Web caching research suffers from a shortage of satisfactory workload data. The fundamental exogenous workload placed on the Web *as a system* consists of the universe of content available from Web servers, the names (URLs) through which content is "published," and end-user requests for content. Most publicly-available traces, however, reflect only the workload placed on individual *components* of the Web, e.g., servers and proxies. It is impossible to infer the fundamentals of data supply and demand from such sources; server workloads don't reflect documents that are never referenced, and proxy workloads don't reflect browser cache hits. While Web component workloads can help us to design better components, we require *system* workloads when we consider fundamentally new Web architectures and design methods. Furthermore existing Web traces record insufficient information about the content (data payloads) returned by servers and therefore shed no light on the performance impact of content-naming practices. In this dissertation I employ two remarkably detailed and large Web workload traces collected in very different environments: a proxy trace recorded on the Compaq corporate network in early 1999 and a client trace collected at WebTV Networks in late 2000.

Since Netscape Navigator and Microsoft Internet Explorer displaced open-source browsers in the late 1990s, it has been difficult for researchers to instrument browsers to collect true client traces, i.e., transaction records that include *all* client accesses, not merely those that miss in the browser cache. Anecdotal evidence suggests that *commercial* enterprises have logged client activity on a large scale using proprietary methods [2], but neither the methods used nor the data collected have been described in the research literature. Between 1995 and my work at WebTV in 2000, researchers recorded countless proxy and server traces, but no true client traces. Furthermore the client traces collected in academic environments in the mid-1990s were far smaller than proxy and server traces, encompassing hundreds of clients and fewer than a million transactions.

The anonymized trace I collected at WebTV is two orders of magnitude larger than any other client trace described in Web-related literature and more detailed in most respects than existing client traces. It includes data-payload checksums for every transaction and records over 347 million transactions initiated by over 37,000 clients during a period of 16 days. To measure workload on this unprecedented scale I employed method never used before: A "cache-busting" proxy served all replies to clients marked pre-expired, thereby effectively disabling browser caches and allowing the proxy to record *all* client requests, including those that would normally be served silently from the browser cache.

Chapter 6 describes how the WebTV trace was collected and presents a detailed workload characterization. My analysis reveals a large gap between the maximal browser cache hit rates determined by client access patterns and those of actual WebTV browser caches. In other words, I found that redundant proxy-to-browser data-payload transfers are surprisingly common in the WebTV system.

1.5 Content-Naming and Performance

One possible explanation for the redundant transfers identified in Chapter 6 is *aliasing*, which occurs when different URLs "point to" identical data payloads. Aliasing can cause unnecessary cache misses in conventional caches that associate cached reply payloads with URLs, e.g., when the payload required to serve the current request is cached, but not in association with the current request URL. More generally, content naming practices—the complex and changing relationship between URLs and data payloads—can cause conventional URL-indexed caches to needlessly retrieve the same payload more than once. Researchers have investigated aliasing in the graph of hypertext links that connects Web pages [36, 144], but the prevalence of aliasing in user-initiated Web *transactions* and the impact of content-naming practices on the performance of conventional cache hierarchies have not been previously quantified.

Working with Jeff Mogul of Compaq Corporation, I have determined precisely the fraction of conventional cache misses that are due to content-naming practices in the aforementioned Compaq and WebTV workload traces. The problem is surprisingly severe: Roughly 10% of payload transfers to conventional URL-indexed browsers and 23% of transfers to proxies are redundant, and are entirely due to the mismatch between conventional URL-

indexed caching and exogenous Web workload (client access patterns and server content-naming practices). Mogul and I independently developed a simple, backward-compatible HTTP protocol extension that completely eliminates redundant payload transfers, regardless of cause. Our "Duplicate Transfer Detection" (DTD) scheme can withstand even *adversarial* workloads: Scramble and confuse the relationship between URLs and data payloads however you please; you will never cause a DTD cache to retrieve the same payload twice. DTD enables a cache hierarchy to attain the maximal hit rates inherent in its workload, and is flexible with respect to the objective function it optimizes: It can minimize either latency or bandwidth.

Chapter 7 analyzes content-naming practices and their impact on conventional cache performance, and presents the Mogul/Kelly Duplicate Transfer Detection protocol extension.

1.6 Summary

This dissertation broadens our perspective on Web caching in several ways. It generalizes our notion of Web workload to include the preferences of system users and describes how these preferences can guide the allocation of cache storage space. It describes principled ways of incorporating both technology costs and access patterns into optimal cache capacity planning, and it demonstrates that capacity planning need not be centralized to be effective. It shows that an end-to-end system-level perspective yields greater insight than the traditional component-oriented focus by proving that content providers' content-naming practices interact with client access patterns in such a way as to impose a large performance penalty on conventional Web caches. Finally, it describes a simple, general, robust, and backward-compatible solution to the pervasive problem of redundant data transfers on the Web.

The basic paradigm of my empirical work is straightforward and likely applicable in a wide variety of contexts beyond the Web: 1) measure fundamental, exogenous, system-level workload in an important production environment, 2) quantify performance bounds inherent in offered workload, independent of the system currently serving it, 3) identify gaps between the actual and potential performance of the current system, and 4) devise

ways of closing these gaps while doing minimal violence to the existing installed base of components, protocols and standards.

My research is relevant to a wide variety of information systems. Cost-minimizing design methods are clearly needed for large-scale deployments of spartan clients; huge losses will result if millions of devices each waste a few dollars. In the longer term, my research will apply to new problems. As entertainment content shifts from broadcast media to retrieval-on-demand systems, optimal cache sizing and management methods will take on new significance: The video rental outlet of the future is a shared networked cache, and it must be designed well to compete in the marketplace. Technologies evolve and their roles change, but caching will always be fundamental to information systems and optimal cache design methods are therefore of lasting relevance. Furthermore, because time-to-market considerations continue to compel hasty deployments of poorly-integrated Internet systems, opportunities for workload analysis and protocol enhancement along the lines of my content-naming investigation will likely arise repeatedly in the years ahead.

The remainder of this dissertation is structured as follows: Chapter 2 formally describes the problems I consider. Chapters 3 and 4 present my results on biased removal policies and optimal cache sizing, respectively. Chapter 5 motivates the need for trace-driven simulation based on large, detailed workload traces, and Chapter 6 describes how I collected such a trace at WebTV Networks. Chapter 7 investigates content-naming practices and the performance problems that arise from their interactions with hierarchies of conventional URL-indexed Web caches. Chapter 8 summarizes my contributions and outlines future work.

CHAPTER 2

Caching Problems

World Wide Web technologies have evolved rapidly and somewhat haphazardly, driven in many cases by competitive pressures and resulting time-to-market considerations. Consequently it is often difficult to describe succinctly and to reason about the Web as it exists "in the wild." Analytic progress requires that we abstract essentials from a bewildering mass of detail, and the present chapter makes explicit the simplifying assumptions that I use in this thesis. Readers interested in the details of real-world Web technologies in general may refer to Krishnamurthy & Rexford's excellent recent book on the subject [96]; those interested in Web caching in particular may consult recent books by Wessels [165] and Rabinovich & Spatscheck [136].

Throughout this thesis we shall consider branching storage hierarchies such as the one depicted in Figure 2.1. Client-end system workload consists of *references* (or *requests*, or *accesses*) that enter the system exogenously, e.g., from human users interacting with browser software. These requests propagate *upstream* toward *origin servers* until they are satisfied by replies containing *data payloads* (or *documents*, or *objects*)[1] accompanied by *metadata*. The server-end aspect of exogenous system workload is the universe of available data payloads and the names through which data are accessible. When we consider the World Wide Web and measured Web workloads, we must sometimes distinguish carefully

[1]The Web caching literature and the core Web protocol specifications do not use terminology consistently or precisely (to take the most notorious example, the HTTP/1.1 specification defines the central concept of "resource" in a circular fashion [64, 118]). In this thesis the term "payload" denotes the particular byte sequence returned in a reply. "Document" or "resource" connotes a networked resource that may change while retaining the same name. We shall never speak of payloads being modified, but we shall sometimes speak of document or resource modification.

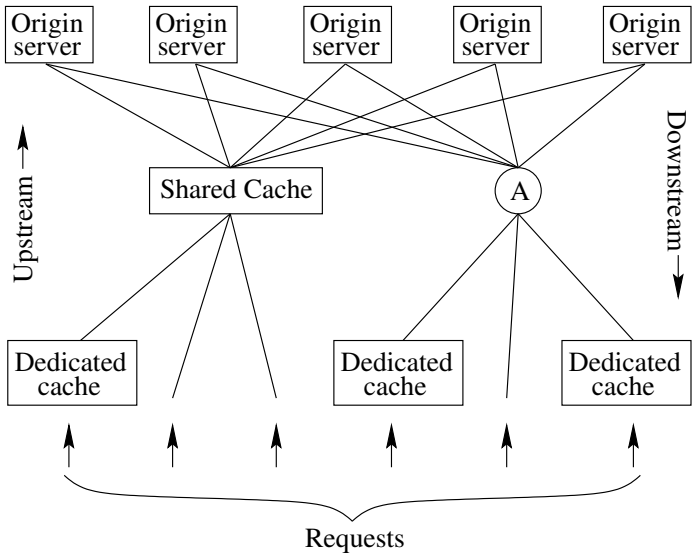

Figure 2.1: A branching multi-level storage hierarchy. Requests from browsers are filtered through dedicated and shared caches on their way to origin servers. Point A is a candidate location for a shared cache, considered in the text.

between the document names or "Uniform Resource Locators" (URLs) [25] contained in requests and the reply payloads they elicit, because some workload traces identify payloads separately from URLs.

Data payloads may be *cached* at intermediate storage nodes as they travel *downstream* toward points of request, and subsequent references may be satisfied by a cached copy stored along the path from point of request to origin server; when this happens we say that a *cache hit* has occurred. Some nodes are *shared caches* that serve requests from several lower-level nodes; others are *dedicated* to a single request stream. We shall consider "Web-like" systems that differ from other layered storage systems, e.g., shared-memory multiprocessors and distributed file systems, in the following ways:

1. Data payloads are not of uniform size.

2. Cache miss penalties are non-uniform.

3. Payloads are atomic; partial payloads are not transmitted or stored (HTTP/1.1 supports partial-payload replies, but this feature is not widely used in practice).

4. Cached payloads are read-only; only origin servers may modify them.

5. Caches are fully associative.

6. All data movement and caching is demand-driven; prefetching does not occur, and payloads are cached and evicted only in response to requests.

7. Servers are typically stateless, and the protocol governing requests and replies assumes stateless servers.

8. System components are physically distributed and may be owned and administered by different organizations whose interests do not coincide.

9. The namespace is unbounded (unlike a CPU address space).

10. Cache consistency mechanisms involve expiration times that origin servers associate with payloads, or that intermediate caches compute heuristically. (Stronger consistency mechanisms, e.g., involving callbacks, would violate the statelessness property.)

Our goal as system designers is to optimize cost and performance metrics that describe how well the system handles offered workload. We are permitted to modify the system serving exogenous workload but not the workload itself, e.g., we may introduce or alter components but we cannot re-arrange client access patterns or modify server content-naming practices. In this thesis I consider interventions involving caching.

One class of problems that I consider is that of serving offered workload at minimal cost. Of particular interest is the tradeoff between the monetary cost of cache storage capacity and that of bandwidth, because this tradeoff is important in practice, because both costs are relatively easy to estimate, and because both can often easily be expressed in comparable units (dollars). In the interest of generality, however, we shall prefer optimization techniques that permit us to assign arbitrary costs to how the system under our control serves offered workload. This allows us to contemplate any costs that can in principle be expressed in monetary terms, e.g., the disutility of latency for interactive users.

Of the many available opportunities for intervention, I focus on the following: We can install caches where none currently exist, e.g., at point "A" in Figure 2.1, if doing so reduces overall costs. Furthermore, we can add storage capacity to caches. After *static* decisions regarding cache placement and size have been made, a crucial *dynamic* intervention remains: Replacement policies can attempt to minimize the aggregate cost of cache misses that occur while serving requests. Finally, we can identify and rectify cases where the protocols

that govern interactions among caches in a hierarchy are ill suited to offered workload. In summary, I address the following questions:

1. When is a cache economically justifiable?
2. What is the optimal size of a cache?
3. How can a cache of fixed, finite capacity best serve offered workload?
4. How can caches better cooperate to serve workload?

I consider these questions in the sequence shown, addressing each assuming that answers to the previous questions have been fixed.

To the extent that it considers monetary cost at all, existing capacity planning literature sometimes regards it as the objective function in a constrained optimization problem:

> The purpose of capacity planning for Internet services is to enable deployment which supports transaction throughput targets while remaining within acceptable response time bounds and minimizing the total dollar cost of ownership of the host platform [129].

The HP Labs MINERVA system automates this process to an extent [8]. Because I focus on *unconstrained* problems it might seem that my approach ignores or contradicts the conventional capacity planning literature, much of which is mature and well developed [112]. However, we shall see that under certain reasonable assumptions performance constraints and the goal of cost minimization may be considered separately, because design prescriptions from conventional capacity planning methods and from my own cost-minimization techniques can be reconciled very easily. The two families of methods are complementary but not tightly coupled, allowing them to develop independently.

The subsections that follow outline the main issues surrounding the caching problems that I consider.

2.1 Cache Sizing

Informally, the cache sizing problem is to determine the optimal size of an existing cache, i.e., a storage capacity that minimizes the total cost of serving requests submitted

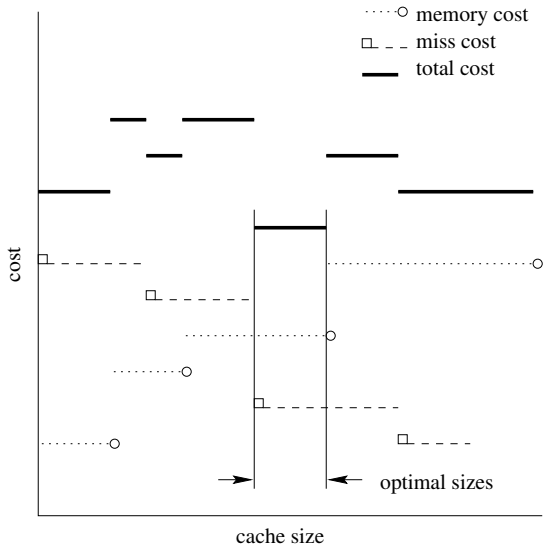

Figure 2.2: Cache cost functions.

to the cache. The tradeoff at issue is the cost of storing payloads locally versus the cost of repeatedly retrieving them; the latter may reflect the cost of upstream bandwidth, server load, end-user latency or other costs. Formally, let $\$_M(s)$ denote the memory cost of cache capacity s, and let $\$_A(s)$ represent the aggregate cost of cache misses incurred when a cache of size s processes the given workload. Our goal is to find an optimal size $s*$ that minimizes total cost $\$_M(s*) + \$_A(s*)$. In general, both $\$_M(s)$ and $\$_A(s)$ are monotonic step functions, as illustrated in Figure 2.2. Note that minimal total cost need not occur at a single cache size, that total cost is a step function but need not be monotonic, and that local minima may exist in the total cost function. Finally, note that total cost increases monotonically for cache sizes greater than the "working set size" (sum of distinct payload sizes) of the offered workload; we may therefore ignore cache sizes larger than this effectively-infinite bound.

As we consider the offline problem of determining optimal cache size, we shall represent workload in one of two ways: as an explicit sequence of references (a trace), or in a probabilistic form that is more amenable to analytic techniques. In an explicit representation we are given a sequence of M references to one of N documents; associated with each request is a nonnegative miss cost.[2] In a probabilistic representation we are given only

[2]Throughout this dissertation miss cost represents the *additional* penalty we face when cache misses occur rather than any absolute measure of disutility; in other words, miss cost is the difference between the utility of a cache hit and that of a miss. It is assumed to be nonnegative.

the relative popularity of each document. A probabilistic representation ignores temporal locality and other workload details, greatly simplifying the problem; Section 4.2 exploits the simplicity of this representation in conjunction with an idealized cost model to compute simultaneously optimal sizes for *many* caches in a branching storage hierarchy. If workload is given as an explicit reference sequence, the difficult part of the cache sizing problem is computing aggregate miss cost as a function of cache size $\$_A(s)$; this is the subject of Section 4.4.

Designers are sometimes given performance constraints (e.g., throughput targets and responsiveness bounds), and cache size is one of many parameters that must be chosen in such a way as to satisfy them. It is possible, for instance, that a certain minimal storage capacity $s_{\min} > s*$ is required to achieve hit rates high enough to satisfy a mean latency target. The correct procedure is therefore to determine the minimal cache size $s_{\min}$ required to satisfy all performance constraints, using the methods of the conventional capacity planning literature [112]; compute $s*$ using methods such as those described in Chapter 4; and finally choose the *larger* of $s_{\min}$ and $s*$. (Here we assume that additional cache hits resulting from choosing an optimal size $s* > s_{\min}$ will not cause performance constraints to be violated. This is a reasonable assumption; cache misses nearly always require more time and computational resources than hits.) In other words, if we are given exogenous performance constraints, our problem is one of optimal cache *expansion* rather than optimal cache *sizing*.

2.2 Cache Installation

The question of whether a cache should be installed at a given location must be answered before we consider optimal cache sizing. However, given an expected workload and a method for computing aggregate miss cost as a function of cache size $\$_A(s)$, it is straightforward to decide whether a cache is economically justifiable: Installation entails some fixed cost $\$_{\text{fixed}}$ in addition to the cost of storage $\$_M(s)$. The cost of *not* installing a cache is $\$_A(0)$, and the cost of installing a cache of optimal size $s*$ is $\$_{\text{fixed}} + \$_A(s*) + \$_M(s*)$. We simply choose the alternative with lower cost. As in the cache sizing problem, the difficult part is computing $\$_A(s)$ based on workload. So far we have ignored interactions between different caches' workloads, e.g., the impact of browser caches on the workload that

reaches shared proxy caches. However in Section 4.2.1 we shall see that when workload is expressed probabilistically the cache installation problem can be solved for a two-level branching cache hierarchy.

Again, conventional performance constraints do not complicate the task of deciding whether or not to install a cache, for the same reason: If the minimum storage capacity required to satisfy performance constraints, $s_{\min}$, is greater than $s*$, we compare $\$_{\text{fixed}} + \$_A(s_{\min}) + \$_M(s_{\min})$ with $\$_A(0)$ when deciding whether a cache is economically justifiable, i.e., we compare an optimally *expanded* cache with no cache.

Of course, if it is not possible for a cache of *any* capacity to provide reasonably responsive service and satisfy other performance constraints, then we ought not install a cache. This possibility cannot be dismissed, because careful studies of intermediate caching servers in distributed file systems have concluded that under some conditions such caches can *degrade* some performance metrics, e.g., latency [121, 122]. As stated previously, throughout this dissertation we assume that cache hits are always preferable to cache misses; miss costs represent the *additional* penalty we incur from cache misses, and are nonnegative.

2.3 Removal Policies

Given that a cache has been installed and its capacity is fixed, the remaining problem is how best to serve its workload. If workload is represented probabilistically, this is a straightforward task due to the assumption of independent references: The cache must solve a classic knapsack problem, storing a subset of data payloads with maximal popularity-weighted miss cost subject to a capacity constraint. Therefore when considering the cache service problem we shall restrict attention to the case where workload is represented as an explicit trace. A cache removal policy should strive to minimize the aggregate cost of processing all requests, i.e., the sum over all cache misses of miss cost.

Alternatively, we might speak of the value of cache hits rather than the cost of cache misses, and say that a cache should maximize value, perhaps by preferentially storing the most valuable documents. The two perspectives—cost minimization and value maximization—are substantively equivalent but differ in connotation, and in some cases we shall adopt the latter view. In particular, the "value" perspective is more natural in situations where miss costs are supplied to a cache by system users, e.g., servers and clients. Chapter 3

considers a scenario in which content providers declare to a cache the value they receive from cache hits, and Section 3.4 describes difficulties that can arise when clients supply hit values that bias a removal policy.

2.4 Redundant Transfers

Web transactions involve requests containing names (URLs) that elicit replies containing data payloads. Content providers define the relationship between URLs and reply payloads, and this relationship is neither simple nor stable: Identical URLs can yield different reply payloads and different URLs can yield identical payloads. We refer to these phenomena as *resource modification* and *aliasing*, respectively.

Traditional Web caches use URLs to organize and locate stored data, i.e., cached reply payloads are associated with, and accessed via, the URL that yielded them. Content-naming practices at the server end can interact with client request patterns in such a way as to cause redundant payload transfers in conventional "URL-indexed" caches. Aliasing, for instance, can cause redundant transfers when a conventional cache already holds the payload needed to satisfy the current request, but not in association with the current request URL. These observations suggest that URL-indexed caching may be poorly suited to Web workloads. Chapter 7 quantifies the prevalence of namespace complexities such as aliasing and resource modification in real Web workloads and the rate of redundant transfers due to the use of URL-indexed caches. It also describes a simple, backward-compatible protocol extension capable of completely eliminating redundant payload transfers, regardless of cause. Jeff Mogul and I devised this protocol extension independently and are evaluating it together; we call it "Duplicate Transfer Detection" (DTD). A happy side effect of DTD is that it ensures perfect cache consistency for all types of data.

2.5 Cache Consistency

To remain *semantically transparent*, caches must serve the same payload as the origin server would at the time they process requests. Consecutive accesses to the same URL sometimes yield different reply payloads, and URL-indexed caches therefore require some mechanism to determine whether a payload cached in association with the current request

URL is *fresh*, i.e., is the same as the origin server would return. The statelessness constraint discussed earlier precludes invalidation-based consistency mechanisms in which servers track cache contents and explicitly instruct caches to discard stale entries. Remaining alternatives include *expiration*, in which reply metadata specifies a time beyond which the reply data should not be considered fresh, and *revalidation*, in which caches verify freshness by contacting the origin server.

In practice, the *freshness policies* of today's Web caches employ a combination of the two, serving requests from cache if a fresh cache entry is available for the current request URL and revalidating if an entry exists but is stale. Reply metadata may specify an absolute expiration time or an age limit for cache entries; if origin servers provide no such metadata, the cache freshness policy will typically compute an estimated time-to-live heuristically. Revalidations may ask whether a resource has changed since it was retrieved from the origin server, or they may compare the *entity tags* of the cached resource with the origin server's current view of the resource. Entity tags ("Etags") are a kind of opaque, unordered version identifier that origin servers associate with payloads; matching Etags imply identical payloads.

Some URLs correspond to simple static files residing on disk at the server. Others, however, invoke scripts, programs, or database queries whose output is often termed "dynamic content." Similarly, replies are sometimes customized for individual users using mechanisms such as "cookies" [97]. Origin servers may explicitly mark customized and dynamic replies "uncachable," or they may instruct caches to revalidate the cached payload each time it is used, thus saving bandwidth while preserving semantic transparency.

As we shall see in Chapter 7, expiration mechanisms sometimes fail to preserve semantic transparency, and existing revalidation mechanisms often fail in surprising ways, causing unnecessary payload transfers. The Mogul/Kelly Duplicate Transfer Detection protocol extension is compatible with and complementary to existing expiration and revalidation mechanisms, guarantees semantic transparency, can be used with "dynamic" and customized content, lacks the subtle failure modes of HTTP's existing consistency mechanisms, and eliminates redundant data-payload transfers entirely.

CHAPTER 3

Preference-Sensitive Removal Policies

Due to differences in server capacity, external bandwidth, and client demand, some Web servers value cache hits more than others. If a shared cache knows the extent to which different servers value hits, it can employ a *preference-sensitive* replacement policy that attempts to deliver higher aggregate value to content providers.[1] Storage space in shared Web caches—proxies serving corporate- or campus-sized LANs and backbone caches embedded in high-speed networks, as opposed to browser caches—can be diverted to serve those who value caching most by the removal policy. Caches are therefore ideal loci for variable-QoS mechanisms. Finally, it is widely observed that cache hit rates are proportional to the *logarithm* of cache size, and that removal policies vary widely in performance by several metrics; therefore a better removal policy can yield benefits equivalent to a *several-fold* increase in cache size.

This section introduces a novel preference-sensitive LFU/LRU hybrid, "Aged server-weighted LFU" (A-swLFU), that is designed to exploit observed regularities in Web request patterns. I compare this algorithm with others from the Web caching literature, discuss the problems associated with obtaining servers' private valuation information, and describe difficulties that arise when a removal policy attempts to accommodate heterogeneous *client* preferences, as opposed to server preferences.

[1] A note on terminology: In this thesis, as in the Web caching literature, the terms "cost-aware," "preference-sensitive" and "value-sensitive" are used interchangeably. All describe cache replacement policies that attempt to minimize the total cost of processing a request stream in which a miss cost is associated with each document or with each request.

Table 3.1: Notation of Chapter 3.

u	Typical URL
W_u	Server-assigned weight on URL u
size_u	Payload size of URL u (bytes)
α	Zipf exponent
K	Aging parameter in A-swLFU
L	Aging term in GD-Size
i	Typical client
s	Typical server
W_s	Weight from server s
w_i	Weight from client i
n_{iu}	Number of references to URL u by client i
N_u	Overall reference count on URL u
V_u	Removal priority of u in cwLFU

Section 3.1 discusses the nature of value-sensitive replacement policies and describes several from the existing Web caching literature. Section 3.2 explains how the traditional caching problem can be decomposed into two problems—value differentiation and prediction—and presents empirical analyses of Web trace data to justify the design decisions underlying the prediction features of my own algorithm. Section 3.3 presents empirical results comparing the value-sensitive performance of several value-sensitive algorithms. Section 3.4 describes circumstances under which biased frequency-sensitive algorithms such as ours do *not* perform well, and Section 3.5 discusses economic incentive issues surrounding value-sensitive caching.

3.1 Value-Sensitive Caching

Actual production caches currently employ LRU-like algorithms or periodic purge policies, often for reasons related to disk performance, but a far wider range of removal policies has been explored in the research literature. Williams et al. present a systematic taxonomy of policies organized by the sort keys that determine the removal order of cached documents [166]. For instance, LRU evicts documents in ascending order of last access time and LFU employs ascending reference count. Bahn et al. [18] provide a comprehensive review of the literature on removal policies, which is too large to be summarized here.

The early literature on Web cache replacement algorithms considered policies intended to maximize performance metrics such as hit rate and byte hit rate; in a sense, the implicit design paradigm is one in which the cache designer "hard wires" into a cache the objective function it will maximize by specifying a rigid replacement policy.

Starting in the late 1990s, several researchers have independently explored more flexible approaches to cache management. Many of these reflect a sophisticated design approach in which a cache attempts to optimize an objective function that is *not* hard-wired into the replacement policy; the objective function is specified by associating a miss penalty with each reference. The need to provide different service levels to different content providers motivates my interest in such algorithms. I begin with the assumption that different servers value cache hits on their objects differently, possibly with quite large differences. Some servers will have clients who are intolerant of delay, and who may be willing to pay for a higher quality of service. Others may be constrained in their external network connections and server equipment, and thus may value off-loading traffic to a network cache, particularly during anomalous heavy-load ("flash crowd") events. Together with complementary research into variable-QoS Web content hosting [3, 29, 133, 161], the growing family of value-sensitive caching policies addresses the needs of a heterogeneous user community.

3.1.1 Value Model

We assume that servers associate with each of their URLs u a number W_u indicating the value they receive per byte when u is served from cache: The value generated by a cache hit equals $W_u \times \text{size}_u$. This information could be transmitted to a shared cache in HTTP reply headers. (We might speak of W_u as per-byte miss cost rather than hit value; the two perspectives are essentially equivalent.) Thus, we can compare all replacement algorithms—value sensitive or insensitive, value or cost based—in terms of *value hit rate* (VHR), defined as

$$\text{VHR} \equiv \frac{\Sigma_{\text{hits}} W_u \times \text{size}_u}{\Sigma_{\text{requests}} W_u \times \text{size}_u}. \tag{3.1}$$

This performance metric is a natural generalization of familiar measures: When $W_u = 1$ for all documents, VHR is equal to byte hit rate; if $W_u = 1/\text{size}_u$ it is equal to hit rate.

3.1.2 Value-Sensitive Removal Policies

Several removal policies designed to maximize VHR have been proposed. Cao & Irani's "GreedyDual-Size" (GD-Size) algorithm attempts to optimize an arbitrary objective function that may be supplied dynamically, at cache run time [42]. In the terminology of our value model, given value weights W_u GD-Size seeks to maximize aggregate value across all requests. Following a request for u, the document's removal priority is set to $W_u + L$. L is an aging term initialized to zero; following a removal it is set to the priority of the evicted document. LRU breaks ties between documents whose removal priority is otherwise identical [41]. GD-Size is a value-sensitive *recentist* algorithm, because when all W_u are equal, it reduces to LRU. At around the same time that GD-Size was first proposed, Wooster & Abrams explored similar removal policies that retain documents that require the longest time to retrieve from origin servers [173].

"Server-weighted LFU" (swLFU) is a simple *frequentist* cache replacement policy [86]. Removal priority is determined by weighted reference count $W_u \times N_u$, where N_u is the number of requests for u since it last entered the cache; last access time breaks ties between documents with identical value-weighted reference counts. When all W_u are equal and positive, swLFU reduces to LFU; when all weights are zero it becomes LRU. Figure 3.1 describes the algorithm in pseudocode.

swLFU retains those URLs that contribute most to aggregate user value per unit of cache space:

$$\frac{\text{contribution of } u \text{ to aggregate value}}{\text{unit size}} = \frac{W_u \times \text{size}_u \times N_u}{\text{size}_u} = W_u \times N_u$$

As expected, swLFU does indeed favor URLs with high weights. A positive correlation between service quality (byte hit rate (BHR)) and declared weight is evident when we experimentally measure BHR as a function of randomly-assigned weight in a trace-driven simulation (Figure 3.2). In our tests a tenfold increase in W_u corresponds to roughly a doubling in BHR. If servers must *pay* the cache for the value they receive (shown as an optional feature in Figure 3.1), we might say that swLFU attempts to *maximize cache revenue*. If W_u are tied to payments, servers will be deterred from reporting inflated weights. Furthermore, provided that servers know they will receive more cache hits if and only if they declare higher weight, they have an incentive to report weights that reasonably approximate their

```
for each requested document u
    if u is in cache
        deliver u to client
        record access time of u
        N_u ← N_u + 1
        [optional] charge (W_u × size(u)) dollars to server of u
    else
        retrieve u and W_u from server
        deliver u to client
        if size(u) ≤ cache size
            while (sum of sizes of cached URLs + size(u) > cache size)
                among cached URLs with minimal N × W, remove LRU item
            place u in cache
            record W_u and access time of u
            N_u ← 1
        end if
end for
```

Figure 3.1: The swLFU algorithm.

true valuations. Economic incentive issues such as this rarely appear in the mainstream Web caching literature. The only example of which I am aware is that Rizzo & Vicisano criticize Wooster & Abrams' removal policy, which keeps in cache documents that take longest to retrieve, on the grounds that it rewards slow origin servers [140, 173].

Arlitt et al. have introduced a frequency-sensitive variant of GD-Size, "GD-Size with Frequency" (GDSF) [10]. In GDSF a document's removal priority is set to $N_u \times W_u + L$ following a reference, where L has the same meaning as in GD-Size. Bahn et al. describe *family* of value-sensitive algorithms, collectively known as "Least Unified Value" (LUV), whose emphasis on frequency and recency can be adjusted [18]. Jin & Bestavros have developed a sophisticated *self-tuning* parameterized generalization of GD-Size called GD* [82].

3.2 Prediction vs. Value Sensitivity

One approach to designing Web caching systems, typical of the earliest literature, is to implement new features on an ad hoc basis and test performance experimentally. A more refined approach, common in the mature Web caching literature, is to identify regularities

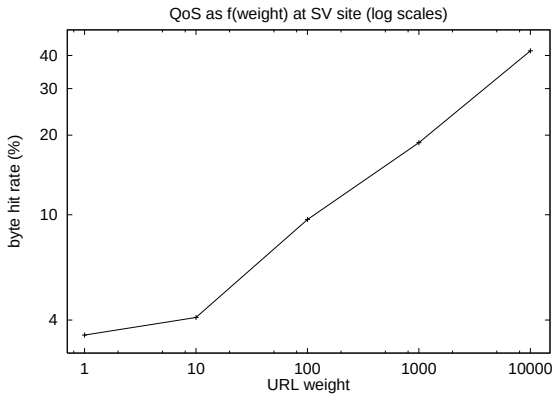

Figure 3.2: Quality-of-service (byte hit rate) as a function of W_u for SV trace summarized in Table 3.2.

in Web cache workloads and to implement features that are well-suited to these regularities; Rizzo et al. provide an elaborate example [140]. This section describes a conceptual framework for characterizing workloads and designing value-sensitive removal policies, then presents empirical workload analysis that guided the design of A-swLFU.

The performance of any value-sensitive caching system depends on how well it solves two distinct problems: *prediction* and *value differentiation*. Any measure of performance will depend on having objects already waiting in the cache before they are requested, hence prediction. Because cache space is scarce it is not possible to store permanently all requested objects (otherwise removal policies would be unnecessary); therefore a cache should identify and store the most valuable documents, i.e., those whose presence in cache is expected to yield the highest value through future cache hits. This value/prediction framework is similar in spirit to an elegant approach developed independently by Bahn et al. [18], though different in emphasis.

Conventional removal policies have largely focused on solving the prediction problem, ranking documents for removal based on estimated likelihood of future requests. Thus, we might expect recentist algorithms like LRU to perform well when there is substantial temporal locality in user requests; frequentist algorithms like LFU are better suited to time-independent requests.

We are primarily interested in the issue of value differentiation. However, an algorithm will not serve users well if it excels at value differentiation but performs poorly at prediction. Therefore I analyzed trace data and studied the prior literature to find regularities important for *prediction*, and used these findings to hard-wire certain features into the

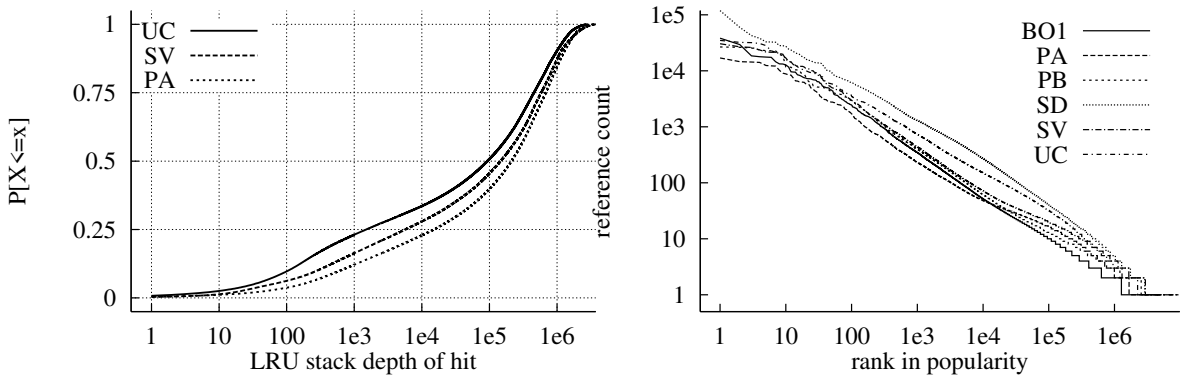

Figure 3.3: Workload characteristics: CDF of LRU stack distances of hits (left) and Zipf-like popularity distribution (right).

new algorithm, while allowing value differentiation to be driven by valuation inputs (W_u). Four Web request stream characteristics relevant to *prediction* are evident in the trace data I analyzed and in the prior literature:

1. Low temporal locality of reference.

2. Zipf-like document popularity distribution.

3. Nonstationary request process.

4. Weak size-popularity correlation.

Temporal locality in a request stream is quantified via LRU stack distance transformation. Requested items in the stream are added to an infinite-capacity stack as follows: If the item is not present in the stack ("miss"), we push it on the top (at depth 1) and output ∞; this increases by 1 the depth of all items already in the stack. If an item *is* present in the stack ("hit"), we output its depth, remove it, and replace it at the top. For example, the symbol stream "ABBACBD" yields "∞ ∞ 1 2 ∞ 3 ∞." Maximal temporal locality occurs when all references to the same symbol are adjacent on the input, in which case all hits occur at depth 1; the string "AABBBCD" has the same relative symbol frequencies as the previous example, but now a stack distance transform yields "∞ 1 ∞ 1 1 ∞ ∞." Barford et al. [19] and Mahanti & Williamson [103] apply stack distance analysis to Web caching. We shall return to LRU stack distance transformation in Chapter 4, which discusses its relationship to cache performance.

	PA site	SV site	UC site
# origin servers	114,381	124,698	105,710
# URLs	3,412,105	3,744,274	2,884,598
# requests	7,011,622	7,897,659	5,568,112
Bytes req'd	131,665,275,664	161,620,444,331	127,346,723,989
Infinite cache			
size (bytes)	60,037,623,775	66,976,225,688	51,825,514,504
hit rate	51.3364	52.5901	48.1943
byte HR	54.4013	58.5596	59.3036
value HR	48.5422	57.4670	56.5745

Table 3.2: Summary statistics on three request streams after filtering out uncachable documents.

Figure 3.3 (left) shows the cumulative distribution of LRU stack hits in 14-day request streams collected during August 1998 at three NLANR caches [66]; Table 3.2 displays summary statistics for these three request streams. The median stack depth of hits ranges from 100,000 to 200,000, indicating weak temporal locality. This conclusion is consistent with other researchers' findings, e.g., Mahanti & Williamson, who report consistently low temporal locality across several shared-cache workloads [103], and Barford et al., who report that temporal locality in *client* traces declined between 1995 and 1998 [19]. The implication for removal policy design is that pure recentist algorithms like LRU are a poor choice.

A second observation is that the frequency of document requests in Web traces is Zipf-like, i.e., the number of references to the ith most popular document is proportional to $1/i^{\alpha}$. This is qualitatively apparent in Figure 3.3 (right), a log-log plot of reference count as a function of popularity rank for six 28-day NLANR traces collected during March 1999; Table 3.3 presents estimates of the α parameter. If we assume that temporal locality is so weak as to be negligible and that document references follow an independent reference model, frequentist prediction is appropriate. Breslau et al. argue that an independent reference model wherein document popularity follows a Zipf-like distribution describes Web workloads well [33].

Even if document references are independent, the distribution that generates them may change over time. This effect is apparent when we examine day-to-day changes in the set

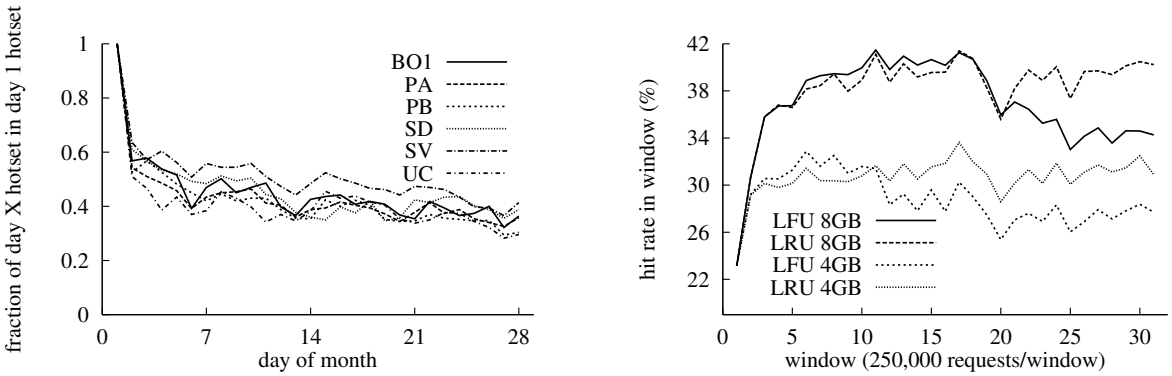

Figure 3.4: Left: Hot set drift at six NLANR sites, March 1999. Right: windowed hit rates for LRU and LFU at two cache sizes, August 1998 SV trace.

of most popular documents in each trace ("hot set drift"). Figure 3.4 (left) shows, for each of the first 28 days in March 1999 at each of six NLANR cache sites, the fraction of that day's 500 most popular documents that were among the 500 most popular on 1 March. The composition of the "hot set" changes gradually over time in our workloads; Mahanti & Williamson report qualitatively similar results for other traces [103]. The implication is that pure frequentist prediction (LFU) will likely suffer a "cache pollution" problem: Formerly-popular documents that are no longer popular will clutter the cache as time goes on. The pollution conjecture is confirmed by a simple experiment: Using an August 1998 NLANR trace, I simulated LRU and LFU caches at 4 GB and 8 GB. Figure 3.4 (right) shows hit rates computed separately within non-overlapping windows of 250,000 requests each. LFU initially outperforms LRU, but over time its performance deteriorates; as we would expect, the pollution problem is more severe at the smaller cache size. Figure 3.4 illustrates the danger of using small traces: If we had used only the first 2.5 million requests for the figure on the right (i.e., the first 10 windows), we would obtain a *qualitatively* misleading result, namely that LFU is simply better than LRU.

Finally, no clear relationship between document size and popularity is evident in the six traces used in our experiments. Table 3.3 provides summary statistics on the six traces used in the main experiments of this section, including size-frequency correlations. In each trace the correlation between document size and popularity does not differ significantly from zero. The design implication is that we should not discriminate against either large or small documents.

The foregoing analysis of request streams suggests that a combination of frequentist prediction and recentist "pollution control" is appropriate. A simple combination of LFU and LRU, together with value sensitivity, yields the aged server-weighted LFU (A-swLFU) removal policy. The default replacement policy is swLFU as described in Section 3.1 (evict objects based on value-weighted frequency count); however, on every Kth eviction we remove the LRU item. A-swLFU reduces to original swLFU and plain LRU as special cases ($K = 0$ and $K = 1$, respectively). A-swLFU is not the first recentist/frequentist hybrid: Lee et al. [100] define a different continuum between LRU and LFU for the *unweighted* case ($W_u = 1$ for all u); Bahn et al. have generalized this algorithm to the *weighted* case of interest to us [18].

Whereas "LRU" unambiguously specifies a single replacement policy, LFU-like algorithms are parameterized by answers to the following questions:

1. What criteria break ties between documents with identical reference counts?
2. Are reference counts maintained on items even after they have been evicted from cache?
3. Is placement following a miss mandatory or optional?

Throughout this section, LRU is the secondary removal criterion in all algorithms; Figure 3.5 shows the impact of the last two parameters on byte hit rate for LFU algorithms that use LRU to break ties. I consider variants of LFU in which reference counts persist across evictions ("Perfect LFU" in the terminology of Breslau et al. [33]), and in which they are defined only for cached items ("in-cache LFU"). While some theoretical investigations consider optional-placement algorithms [79], in my tests it *never* confers a substantial advantage over mandatory placement and often incurs a severe performance penalty, possibly because it ensures that a large fraction of the many twice-requested documents in our traces never result in cache hits. Therefore I consider only mandatory-placement variants of LFU in this section.

3.3 Experiments

This section compares value-sensitive removal policies through trace-driven simulations using Web request streams collected at six NLANR cache sites during 1–28 March

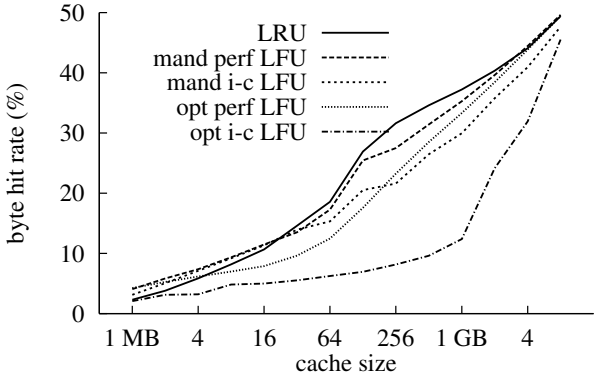

Figure 3.5: Byte hit rate at cache sizes 1 MB–8 GB for LRU and four LFU variants, August 1998 NLANR SV trace.

1999 [66]. Prior to simulation I pre-process raw cache access logs by removing dynamic content and preserving only successful requests for items not present in client caches. The six processed traces are summarized in Table 3.3. The simulations described in this section present severe computational challenges and motivate the design and implementation of an efficient parallel cache simulator, which is described in Section 5.3.

3.3.1 Heterogeneous Valuations

To explore the relative performance of value-sensitive removal policies I conducted experiments of the following form: Randomly assign to each server s a weight W_s drawn uniformly from the set $\{1, 10, 100, 1000, 10000\}$, then set $W_u = W_s$ for all documents u hosted by server s, and finally compute value hit rates for various algorithms at different cache sizes. We use a high-variance weight distribution because, as Section 3.4 explains in greater detail, weighted-LFU algorithms behave very much like ordinary unweighted LFU when weights span a narrow range. Figure 3.6 shows mean VHR over five weight assignments at cache sizes ranging from 64 MB to 16 GB for perfect and in-cache variants of A-swLFU with $K = 100$; no attempt was made to tune K to particular traces or cache sizes. We present LRU at cache sizes from 1–16 GB to illustrate the gap between conventional and value-sensitive algorithms. The results suggest that even without a well-tuned aging parameter A-swLFU yields better aggregate value to servers than GD-Size at smaller cache sizes in most of our traces.

Table 3.3: Traces recorded at six NLANR sites, 1–28 March 1999.

	BO1	PA	PB	SD	SV	UC
requests	11,583,087	13,548,917	19,803,754	37,085,277	23,738,274	26,024,662
documents	5,252,946	4,901,241	9,820,054	8,640,338	9,375,514	7,615,462
servers	193,422	168,082	291,410	247,459	265,305	250,484
unique bytes	104,474,161,664	76,038,927,331	188,308,442,149	204,928,271,675	159,114,665,878	150,119,984,279
bytes requested	236,150,085,697	220,658,618,173	383,130,815,921	620,283,701,022	412,899,064,992	397,548,684,913
max H.R. (%)	54.6	63.8	50.4	76.7	60.5	70.7
max B.H.R. (%)	59.4	67.3	55.2	69.1	63.2	64.4
mean N_u	2.205	2.764	2.017	2.532	4.292	3.417
std. dev. N_u	37.77	25.60	29.71	34.10	74.59	43.80
Zipf α (R^2)	.578 (.88)	.751 (.93)	.560 (.89)	.854 (.93)	.692 (.92)	.784 (.91)
mean size_u	19,888.68	15,514.22	19,175.91	16,971.30	23,717.62	19,712.52
std. dev. size_u	337,424.3	220,990.6	269,819.6	221,649.6	289,507.6	312,418.1
median size_u	3895	3584	3712	3886	4080	3830
$\text{Cov}(\text{size}_u, N_u)$	5158.76	4533.34	4168.46	-25,014.49	3097.10	-11,984.42
$\text{Corr}(\text{size}_u, N_u)$	0.00040476	0.00080136	0.00052007	-0.00115845	0.00040978	-0.00087585

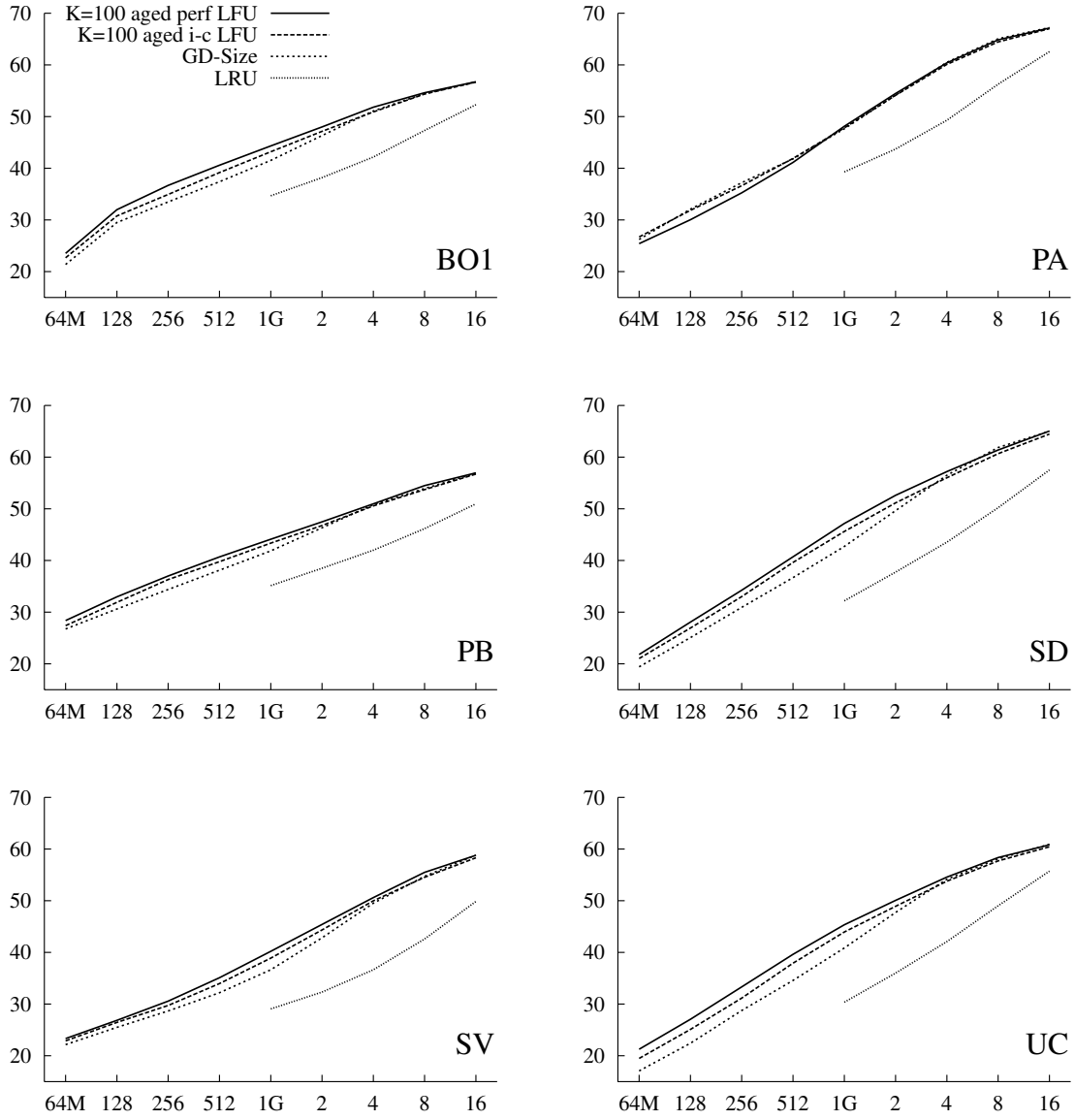

Figure 3.6: VHR as function of cache size for two A-swLFU variants and GD-Size. LRU is also shown at larger cache sizes for comparison. Note that vertical scales do not begin at zero.

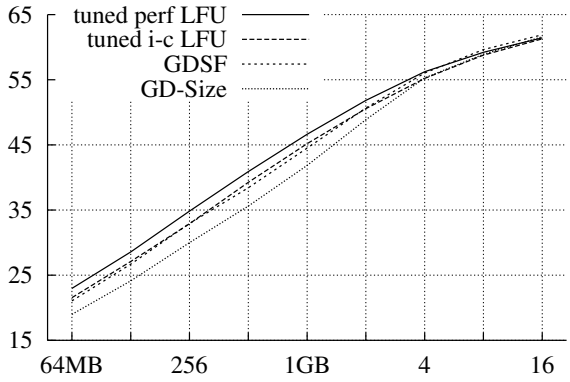

Figure 3.7: Tuned perfect & in-cache A-swLFU, in-cache GDSF, and GD-Size. March 1999 UC trace. The results shown required roughly 60 CPU days to compute using the parallel simulator of Section 5.3.

How does A-swLFU perform with a well-tuned K parameter? Figure 3.7 shows VHR averaged over 20 random assignments of W_u for GD-Size, in-cache GDSF, and perfect and in-cache A-swLFU with K values of $0, 10, 20, \ldots, 150$; at each cache size we present the A-swLFU with the highest VHR. Perfect A-swLFU performs the best for caches that are 4 GB or smaller, consistent with what Breslau et al. report for the *unweighed* case [33]. However the gains over in-cache A-swLFU and GDSF are modest and may not justify the extra cost of retaining frequency tables on evicted documents. Optimally-tuned in-cache A-swLFU and GDSF perform almost identically; both are value-sensitive combinations of recentist and frequentist approaches, so this is not surprising. GD-Size, which does not exploit frequency information, performs noticeably worse except at large cache sizes. Figure 3.9 and the accompanying text in Section 3.3.2 discuss tuning the K parameter in greater detail.

A-swLFU works best when cache space is scarce. This performance advantage is especially important for main-memory caches. Some caching systems are disk I/O constrained [141]. If Web demand and network bandwidth grow so rapidly that disk bandwidth cannot keep pace, RAM-only caches become a favorable design option. Furthermore, Gray & Shenoy predict that as RAM prices drop over the next decade, main memory will fill many of the roles currently played by disks [72]. The absence of disks removes many practical constraints that currently limit cache designers' choice of removal policy. A value-sensitive replacement algorithm enables a diskless cache to provide "premium" service for

those willing to pay for minimal latency. My results show that GDSF and A-swLFU are good replacement policies for such a cache.

3.3.2 Homogeneous Valuations

As a "sanity check" I also consider the degenerate case where all documents have equal weight ($W_u = 1$ for all u). As noted in Section 3.1, GD-Size reduces to ordinary LRU in this case, and the VHR performance metric reduces to byte hit rate. Figure 3.8 presents byte hit rates at cache sizes ranging up to 16 GB generated by GD-Size/LRU and four LFU variants (all combinations of aged ($K = 10$) vs. ordinary ($K = 0$) and perfect vs. in-cache). Our results confirm Breslau et al.'s conclusion that (un-aged) in-cache LFU performs poorly in terms of byte hit rate [33]. However, the addition of aging *without* any attempt to tune the aging parameter improves the performance of in-cache LFU beyond that of un-aged perfect LFU. As expected, aged perfect LFU generally performs best. Finally, in three of six cases (PA, PB, and SD) LRU outperforms un-aged perfect LFU at all cache sizes, contrary to Breslau et al.'s claim that perfect LFU generally performs better than LRU in terms of BHR. We attribute the difference to the size of Breslau et al.'s traces, which are too small for cache pollution effects to occur. More remarkably, aged in-cache LFU outperforms aged perfect LFU on two traces (PA and SD), and performs roughly as well one other (SV).

How much can we gain by tuning K at a particular cache? Figure 3.9 shows byte hit rate as K varies from zero to 25 for in-cache LFU (solid lines) and perfect LFU (dashed lines) at cache sizes ranging from 256 MB (lowermost solid/dashed pair) to 16 GB (top pair). The solid and dashed lines meet at $K = 1$ because both algorithms reduce to LRU at that K value. Remarkably, in-cache LFU with optimal K outperforms perfect LFU with optimal K at *every* cache size. In other words, well-tuned aging appears to eliminate any advantage of maintaining reference counts on evicted documents in the unweighted case. Figure 3.9 furthermore appears to confirm the conjecture that the optimal amount of aging depends on cache size; larger caches require more aggressive aging (lower K).

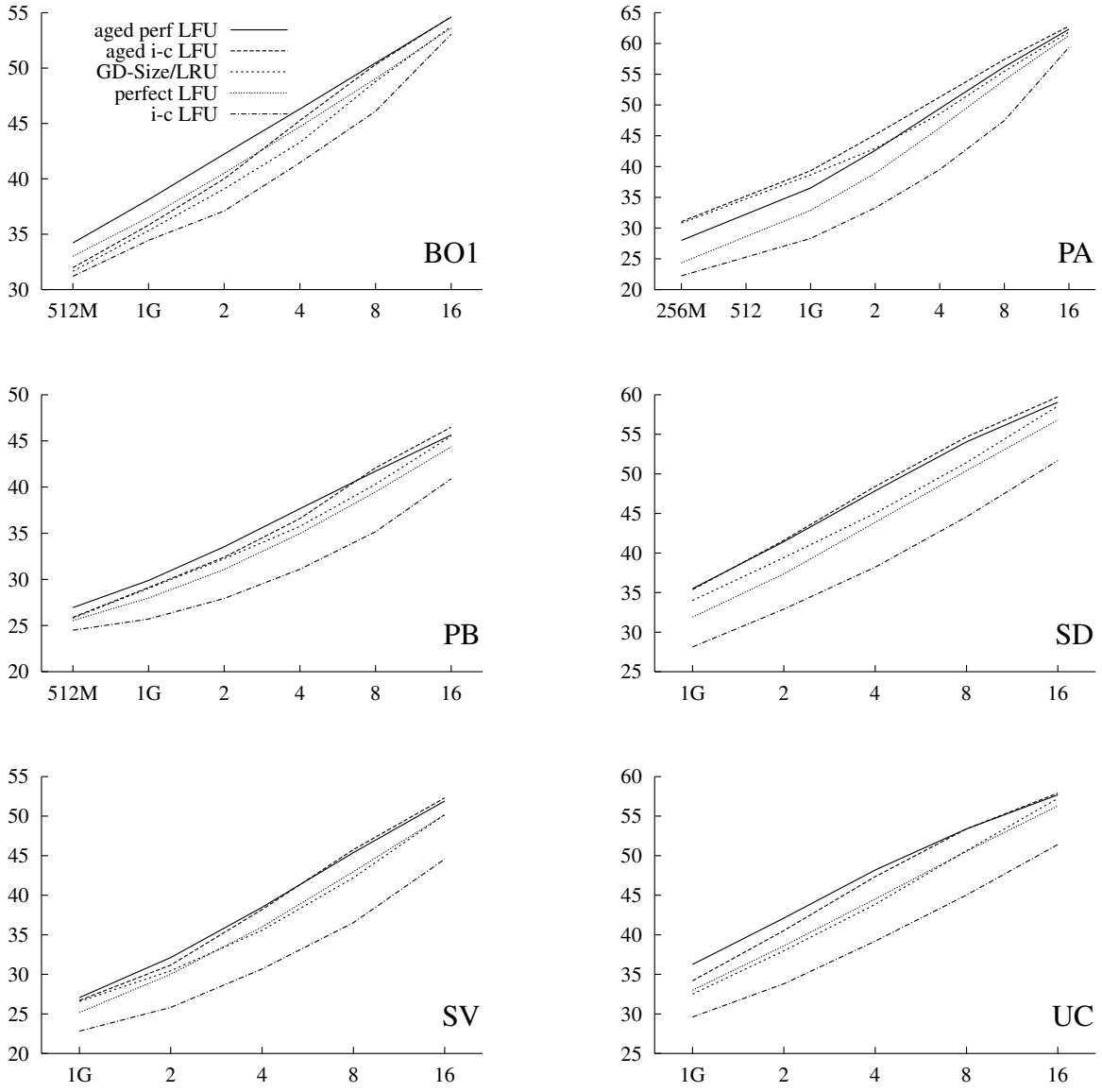

Figure 3.8: Cost = size case: byte hit rates as function of cache size for GD-Size/LRU and four LFU variants: perfect vs. in-cache and K=10 aging vs. no aging. March 1999 NLANR traces. Note that vertical scales do not start at zero, and their upper limits vary.

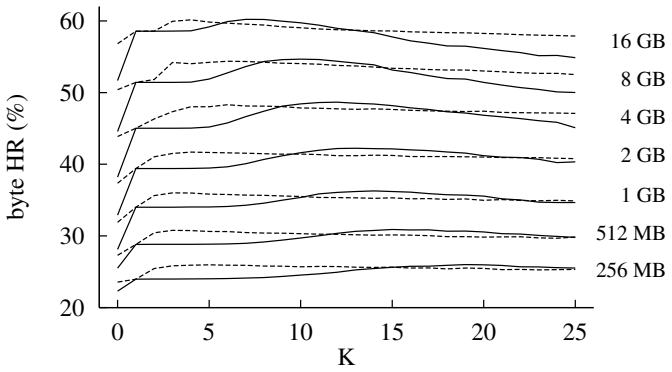

Figure 3.9: Byte HR as function of aging parameter K for in-cache LFU (solid lines) and Perfect LFU (dashed lines) for various cache sizes. March 1999 SD trace.

3.4 Limits to Biased LFU

Weighted-LFU algorithms do not perform much better than their value-insensitive counterparts when access patterns overwhelm or dilute the valuation information contained in weights. I demonstrate this in two situations: when weights are assigned by clients instead of servers, and when weights span a narrow range.

Consider a *client-weighted* "cwLFU" algorithm in which client i supplies weight w_i indicating the utility per byte it receives when its requests are served from cache. Removal priority in cwLFU is determined by

$$V_u \equiv \sum_{\text{clients } i} w_i n_{iu}$$

where n_{iu} is the number of requests for URL u by client i. A problem arises when client weights w_i are uncorrelated with reference counts n_{iu}: The law of large numbers causes the quantity

$$\overline{V}_u \equiv \frac{V_u}{N_u} \quad \text{where} \quad N_u \equiv \sum_i n_{iu}$$

to converge toward the mean of the w_i distribution for URLs with high overall reference counts, because popular documents are referenced by many clients. To illustrate this phenomenon, I obtain n_{iu} data from an NLANR access log, randomly assign to clients integer weights w_i in the range 1–10, and compute $\overline{V}_u$ for URLs with $N_u > 50$. As shown in Figure 3.10, values of $\overline{V}_u$ cluster strongly around 5.5. Ordinary LFU and cwLFU differ only insofar as $\overline{V}_u$ differ substantially across objects, and this does not happen when client

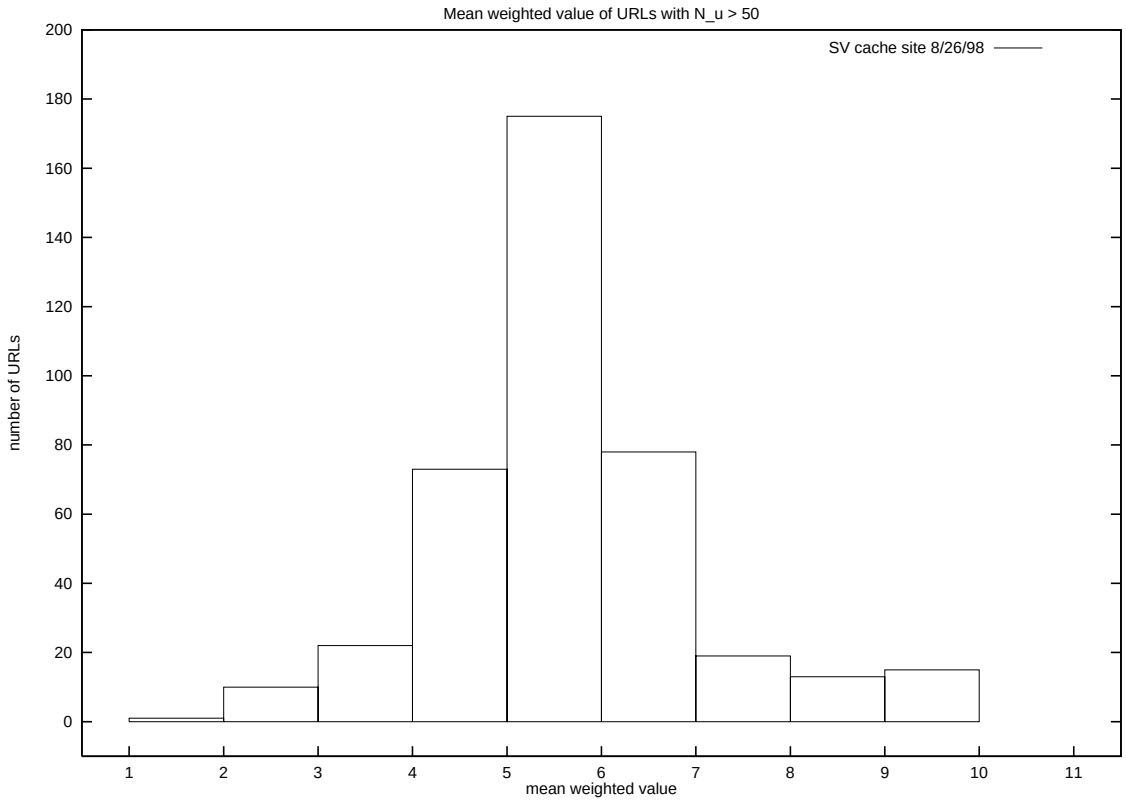

Figure 3.10: Histogram of mean weighted values $\overline{V}_u$ for popular URLs in NLANR's Silicon Valley L3 cache request log of 26 August 1998 (a busy day at a busy site) for a particular random assignment of w_i values to clients. Other assignments of w_i yield qualitatively similar results.

weights are uncorrelated with reference counts. It is conceivable that such correlations do exist in the real world, e.g., we might imagine that impatient clients who value cache hits have similar reading habits. However available data do not allow us to explore such hypothetical correlations, which therefore remain purely speculative. One well-known result is suggestive: Wolman et al. report that the relationship between clients' organizational affiliation (i.e., their department within the University of Washington) and their access patterns is weak. Furthermore even when clients are artificially clustered according to their request patterns, hit rates of shared caches serving these clusters are not substantially higher than for similarly-sized random groups of clients [172].

A-swLFU and swLFU do not perform well with weights drawn from a narrow range, e.g., 1–10. The reason is that document reference counts N_u vary over many orders of magnitude (Figure 3.3). If weights W_u span only one order of magnitude, their influence on the behavior of weighted-LFU variants may be negligible.

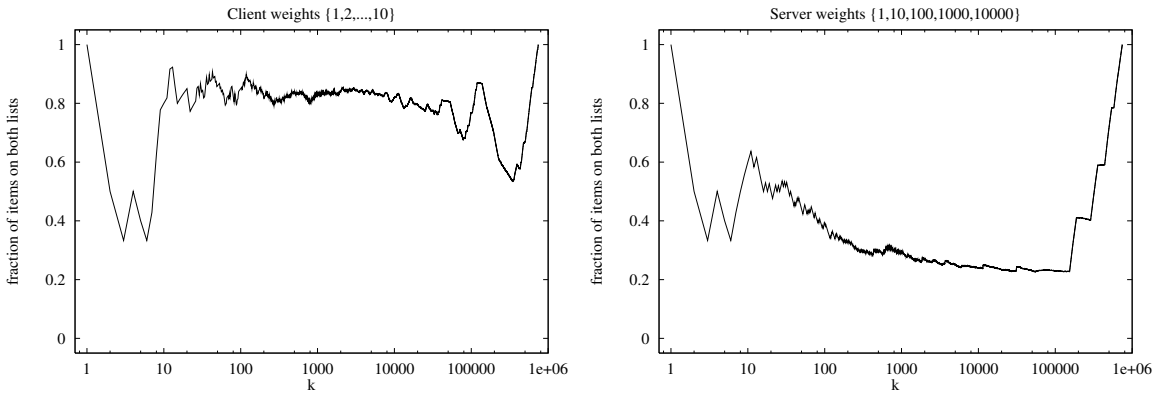

Figure 3.11: Overlap among top k items in lists sorted on weighted and unweighted criteria. Reference counts n_{iu} are from the NLANR SV log of 17 March 1999.

We can illustrate the combined effect of both client weights and weights drawn from a narrow range through a simple experiment: Obtain reference counts n_{iu} from a Web cache access log and assign to the clients in the log weights w_i drawn randomly from $\{1, 2, \ldots, 10\}$. Create two lists of tuples of the form (u, N_u, V_u), one sorted in descending order of reference counts N_u and the other sorted on cwLFU removal priority V_u. Examine the overlap in the top k URLs on both lists as a function of k. If the two lists are very similar, the top k sub-lists will overlap substantially even for small values of k; if the lists are very different, the overlap will be small except for large values of k. This exercise provides a crude comparison of the contents of weighted and unweighted caches: The top k items on our two sorted lists are roughly those that would be contained in cwLFU and unweighted LFU caches of size k after processing the request stream in the access log. This experiment can be performed for swLFU as well as cwLFU; in both cases removal priority is weighted reference count. Figure 3.11 shows list overlap as a function of k in two scenarios: client weights drawn from a narrow range (left), and server weights drawn from our high-variance distribution (right). For a cache capable of holding between 10,000 and 100,000 documents, weighted and unweighted LFU yield very similar cache contents (80% overlap), and therefore similar hit/miss behavior, in the narrow-weight-range cwLFU case. By contrast, the similarity between weighted and unweighted cache contents is far lower (25% overlap) in the wide-weight-range swLFU case. Client weights from a narrow range yield cache contents very similar to ordinary unweighted LFU, whereas server weights from a wide distribution make a substantial difference.

In summary, under certain circumstances weighted-LFU algorithms behave very much like ordinary LFU. Of course, this conclusion depends on the particulars of the weight distributions and other parameters used in the investigations described in this section. However, the interaction between document popularity and weight range and the interaction between client weights and access patterns are generic issues that must be considered in the design of any weighted-LFU removal policy.

3.5 Incentives

By measuring performance (VHR) using server announcements of their values (W_u), we implicitly assume that these announcements are truthful. Unfortunately, when cache replacement is directly affected by the announced values, it will generally be in each server's private interest to systematically misreport its valuations: No matter how low their true values, they would like their objects to get better treatment than another server's objects. The problem of strategic announcements is generic and confronts any value-sensitive replacement policy: A reliable source of user value information is needed to improve on insensitive policies.[2]

A powerful approach to this problem is known as *mechanism design*; Mas-Colell et al. [106] offer a good introduction, McAfee & McMillan [109] review of incentives in auctions, and Varian [159] discusses mechanism design applied to "software agents." The approach provides participants with economic incentives so that it is in their rational self-interest to provide truthful valuation information. The search space of possible incentive schemes is considerably simplified by the Revelation Principle [125], which states that any aggregate user value that can be achieved by some incentive scheme can also be achieved by a scheme in which it is rational for participants to tell the truth. Nonetheless, the design of incentive mechanisms is technically challenging, and is beyond the scope of this discussion. We simply review a few observations on the possible shape of a good scheme.

One important result originally due to Vickrey [160] and generalized to a much richer set of problems in Varian & MacKie-Mason [158] lends some intuition for the problem.

[2]The problem of inducing servers to truthfully reveal private valuation information is distinct from the problem of preventing a *cache* from over-reporting hits in a scheme in which servers pay for cache hits. Economics offers insight into the former problem ("bid shading"), but not the latter (fraud).

Vickrey proposed the second price auction: Charge the winner of a single-good auction the second highest bid. The bidder's announcement affects only *when* she wins, not how much she pays, and it can be shown that the bidder's dominant strategy is to bid her true valuation.

The generalized Vickrey auction suggests that charging a server for each hit the valuation announced for the object that was most recently evicted might be incentive compatible. This works if caching decisions are a one-shot activity. Unfortunately it is not, and in this example, the server's bid affects *future* payments, so it is not optimal to tell the truth. For example, if the current price is less than the server's true value, it will want to overbid to increase its object's duration in the cache, since each hit will produce value greater than its cost.

Chan et al. propose a quite different approach to value-sensitive caching, in which a cache periodically auctions off disk space [45]. In that setting the authors are able to provide an incentive-compatible scheme. However, they report that their cache market yields value lower than swLFU except at extremely small cache sizes (1 and 4 MB [sic]). This is probably because a periodic allocation framework, which seems necessary to achieve incentive compatibility, is not well suited to the natural event-driven dynamic of caching.

CHAPTER 4

Optimal Cache Sizing

This chapter describes two approaches to the problem of determining exact optimal storage capacity for Web caches based on workload and the costs of memory and cache misses. The first approach considers memory/bandwidth tradeoffs in an idealized cost model. It assumes that workload is described probabilistically, i.e., that it consists of independent references drawn from a known distribution, and that caches employ a "Perfect LFU" removal policy. For the cache installation problem, I derive conditions under which a shared higher-level "parent" cache serving several lower-level "child" caches is economically viable. For the cache sizing problem, I characterize circumstances under which globally optimal storage capacities in such a hierarchy can be determined through a *decentralized* computation in which caches individually minimize local expenditures.

The second approach is applicable if the workload at a single cache is represented by an explicit request sequence and the cache employs one of a family of removal policies that includes LRU. Arbitrary miss costs are associated with individual requests, and the cost of cache storage need only be monotonic. Per-request miss costs based on the expense of upstream bandwidth are often readily available in practice. In principle it is also possible to estimate miss costs arising from other sources, e.g., the disutility that human end users incur from latency; econometric research into this topic has begun [7, 29, 78]. I present an efficient single-pass algorithm to compute aggregate miss cost as a function of cache size in $O(M \log M)$ time and $O(M)$ memory, where M is the number of requests in the workload. Because it allows us to compute *complete* stack distance transformations and hit rates at *all* cache sizes with modest computational resources, this algorithm permits analysis of reference locality and cache performance with no loss of precision.

4.1 Monetary Costs and Benefits

Web cache capacity planning must weigh the relative costs of storage and cache misses to determine optimal cache size. While the monetary costs and benefits of caching do not figure prominently in the academic literature, they are foremost in industry analysts' minds:

> CacheFlow is targeting the enterprise, where most network managers will be loath to spend $40,000 to save bandwidth on a $1,200-per-month T1 line. To sell these boxes, CacheFlow must wise up and deliver an entry-level appliance starting at $7,000 [83].

This section considers the problem of determining optimal cache sizes based on economic considerations. I focus exclusively on the storage cost vs. miss cost tradeoff and ignore throughput and response time issues, which are covered extensively elsewhere [112]. As explained in greater detail in Section 2.1, performance constraints and cost minimization may sometimes be considered separately in the cache sizing problem, because in some cases one should simply choose the larger of the two cache sizes they separately require. In other words, under some circumstances, if economic arguments prescribe a larger cache than needed to satisfy throughput and latency targets, an opportunity exists to save money overall by additional spending on storage capacity.

Section 4.2 begins with a simple model that considers only memory and bandwidth costs. The memory/bandwidth tradeoff is the right one to consider in a highly simplified model, because bandwidth savings is the main reason why many institutions deploy Web caches: According to a survey of Fortune1000 network managers who have deployed Web caches, 54% do so to save bandwidth, 32% to improve response time, 25% for security reasons, and 14% to restrict employee access [75]. The analysis of Section 4.2 is similar in spirit to Gray's "five-minute rule" [71] extended to large-scale hierarchical caching systems. I show how the economic viability of a shared high-level cache is related to system size and technology cost ratios. I furthermore demonstrate that under certain conditions, globally-optimal storage capacities in a large branching cache hierarchy can be determined through scalable, decentralized, local computations. Section 4.4 addresses the shortcomings of the simple model's assumptions, describing an efficient method of computing the optimal storage capacity of a single cache for *completely arbitrary* workloads, miss costs, and storage costs. This method allows us to compute *complete* stack distance transfor-

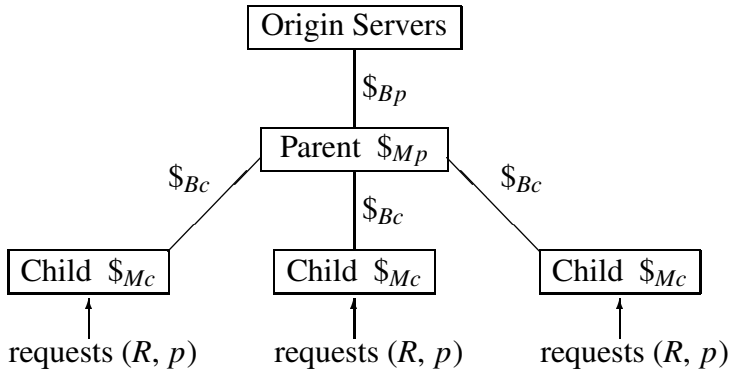

Figure 4.1: Two-level caching hierarchy of Section 4.2.

mations and arbitrarily-weighted hit rates at *all* cache sizes for large traces using modest computational resources. Section 4.7 concludes by discussing the two models' limitations and their relation to other literature.

4.2 A Simple Hierarchical Caching Model

Consider a two-level cache hierarchy as depicted in Figure 4.1 in which C lower-level caches each receive request streams described by the same popularity distribution at the rate of R references per second; child request streams need not be exactly identical, but their aggregate statistical properties (relative popularity of documents and mean request rate) are the same. Requests that cannot be served by one of these "child" caches are forwarded to a single higher-level "parent" cache. A document of size S_i bytes may be stored in a child or parent cache at a cost, respectively, of $\$_{Mc}$ or $\$_{Mp}$ dollars per byte. Bandwidth between origin servers and the parent costs $\$_{Bp}$ dollars per byte per second, and bandwidth between the parent and each child costs $\$_{Bc}$. Our objective is to serve the child request streams at minimal overall cost in the long-term steady state (all caches "warm"). The tradeoff at issue is the cost of storing documents closer to where they are requested versus the cost of repeatedly retrieving them from more distant locations.

Request streams are described by an independent reference model in which document i is requested with relative frequency p_i where $\sum_i p_i = 1$; the rate of request for document i is therefore $p_i R$ requests per second. The model of Breslau et al. [33] (independent references from a Zipf-like popularity distribution) is a special case of the class of reference streams

Table 4.1: Notation of Section 4.2.

M	total number of requests
N	total number of distinct documents
C	number of child caches
R	rate of requests reaching each child cache (requests/second)
i	index of a typical document
p_i	relative popularity of document i, $\sum_i p_i = 1$
S_i	size of document i (bytes)
$\$_{Mc}$	cost of storage at a child cache (\$/byte)
$\$_{Mp}$	cost of storage at parent cache (\$/byte)
$\$_M$	cost of storage when $\$_{Mc} = \$_{Mp}$ (\$/byte)
$\$_{Bc}$	cost of bandwidth between child cache and parent (\$/(byte/sec))
$\$_{Bp}$	cost of bandwidth between parent and origin server (\$/(byte/sec))

considered here. Given independent references drawn from a fixed distribution, the most natural cache removal policy is "Perfect LFU", i.e., LFU with reference counts that persist across evictions [33] (Perfect LFU is *optimal* for such a workload only if documents are of uniform size). Our analysis furthermore requires that caches retain precisely those items with maximal Perfect-LFU reference counts, so we shall therefore assume that all caches use *optional-placement* Perfect LFU: Following a request, the requested item is cached only if its reference count is sufficiently high. Optional-placement variants of removal policies are common in the theoretical caching and paging literature [77, 79].

4.2.1 Centralized Optimization

Because we ignore congestion effects at caches and on transmission links, we may compute optimal cache sizes by determining optimal dispositions for each *document* independently, and then sizing caches accordingly. A document may be cached 1) at the parent, 2) at *all* children, or 3) nowhere. These alternatives are mutually exclusive: By symmetry, if it pays to cache a document at any child, then it ought to be cached at all children; and if a document is cached at the children it is pointless to cache it at the parent. The costs of the three options for document i are

cache at children	cache at parent	don't cache
$CS_i\$_{Mc}$	$S_i\$_{Mp} + Cp_iRS_i\$_{Bc}$	$Cp_iRS_i(\$_{Bp} + \$_{Bc})$

The document should be cached at the children if and only if this option is cheaper than the alternatives (we break ties by caching documents closer to children, rather than farther):

$$CS_i\$_{Mc} \leq S_i\$_{Mp} + Cp_iRS_i\$_{Bc} \quad \Rightarrow \quad p_i \geq \frac{C\$_{Mc} - \$_{Mp}}{CR\$_{Bc}} \tag{4.1}$$

$$CS_i\$_{Mc} \leq Cp_iRS_i(\$_{Bp} + \$_{Bc}) \quad \Rightarrow \quad p_i \geq \frac{\$_{Mc}}{R(\$_{Bp} + \$_{Bc})} \tag{4.2}$$

Each child cache should therefore be exactly large enough to accommodate documents i whose popularity p_i satisfies Equations 4.1 and 4.2; Perfect LFU replacement ensures that, in the long-term steady state, precisely those documents will be cached at the children. By similar reasoning, the parent cache should be just big enough to hold documents for which parent caching is the cheapest option:

$$p_i < \frac{C\$_{Mc} - \$_{Mp}}{CR\$_{Bc}} \tag{4.3}$$

$$S_i\$_{Mp} + Cp_iRS_i\$_{Bc} \leq Cp_iRS_i(\$_{Bp} + \$_{Bc}) \quad \Rightarrow \quad p_i \geq \frac{\$_{Mp}}{CR\$_{Bp}} \tag{4.4}$$

Taken together, the requirements for parent caching (Equations 4.3 and 4.4) imply a necessary condition for the cache installation problem; a parent cache is justifiable only if there are sufficiently many children:

$$\frac{C\$_{Mc} - \$_{Mp}}{CR\$_{Bc}} > p_i \geq \frac{\$_{Mp}}{CR\$_{Bp}} \quad \Rightarrow \quad C > \frac{\$_{Mp}\$_{Bc}/\$_{Bp} + \$_{Mp}}{\$_{Mc}} \tag{4.5}$$

Note that it is straightforward to handle cases where a shared cache entails a fixed cost: We compute the cost of serving the given workload without a shared cache and with optimal child cache sizes (the only options for each document are to cache it at the children, or nowhere). We then compute the cost of serving workload assuming a shared cache and optimal cache sizes everywhere, and then simply select the cheaper alternative.

Of particular interest is the special case where per-byte memory costs at parent and children are equal, and the number of children is large. If $\$_{Mp} = \$_{Mc} = \$_M$ then the criterion that determines whether a parent cache is economically justified by the number of children (Equation 4.5) simplifies to

$$C > \frac{\$_{Bc}}{\$_{Bp}} + 1 \tag{4.6}$$

Equations 4.5 and 4.6 are independent of document sizes, popularity, and request rates. In the simplified model we are considering, the number of child caches required for a shared parent to be economically viable depends entirely on technology costs.

If in addition to uniform memory costs we furthermore assume that C is very large, the criteria for caching at a child (Equations 4.1 and 4.2) simplify to

$$p_i \geq \left(\frac{C-1}{C}\right)\frac{\$_M}{R\$_{Bc}} \approx \frac{\$_M}{R\$_{Bc}} \quad \text{and} \quad p_i \geq \frac{\$_M}{R(\$_{Bc}+\$_{Bp})}$$

If the first of these inequalities is satisfied, then the second must also be satisfied, because R and all costs are strictly positive. Therefore in the case where the number of children is large and memory costs are identical at parent and children, document i should be cached at children iff

$$p_i \geq \frac{\$_M}{R\$_{Bc}} \tag{4.7}$$

4.2.2 Decentralized Optimization

I now describe circumstances under which a *decentralized* computation that uses only local information yields the same result as the centralized computation of Section 4.2.1. Imagine that the parent and children caches are operated by independent entities, each of which seeks to minimize its own operating costs ($\$_{Mp}$ and $\$_{Bp}$ for the parent, $\$_{Mc}$ and $\$_{Bc}$ for the children). Each child's decision whether or not to cache each document is independent of whether the document is cached at the parent, because the transmission and storage costs facing children are unaffected by caching decisions at the parent. The higher-level cache in turn bases its caching decisions solely on the document requests submitted to it and the costs it must pay to satisfy them. A child will cache document i iff

$$S_i\$_{Mc} \leq S_i p_i R\$_{Bc} \quad \Rightarrow \quad p_i \geq \frac{\$_{Mc}}{R\$_{Bc}} \tag{4.8}$$

After the lower-level caches have sized themselves to accommodate exactly those documents whose relative popularity satisfies Equation 4.8, requests for those documents will not reach the parent. The parent will, however, receive requests for all other documents j

at the rate of Cp_jR, and will choose to cache all documents that satisfy

$$S_j\$_{Mp} \leq Cp_jR\$_{Bp} \quad \Rightarrow \quad p_j \geq \frac{\$_{Mp}}{CR\$_{Bp}} \tag{4.9}$$

The condition of Equation 4.9 is identical to that of the previous centralized-optimization result (Equation 4.4). Furthermore, when memory costs are uniform Equation 4.8 becomes the child-caching criterion for large numbers of children (Equation 4.7). Therefore the caching decisions—and hence cache sizes—determined independently by parent and children through (literally) greedy local computations are the same as those that a globally-optimizing "central planner" would compute.

4.3 Cost Calculations

In practice bandwidth costs rarely have the convenient dimensions we have thus far assumed, because they typically involve fixed installation costs as well as periodic maintenance and service fees. However, we can convert periodic costs into a single cost using a standard present-value calculation [32]; in the simplest case

$$\text{present value} \quad = \quad \frac{\text{periodic payment}}{\text{interest rate during period}} \tag{4.10}$$

For example, if the annual interest rate is 5%, the present value of perpetual yearly payments of $37 is $37/.05 = $740. Slightly more sophisticated calculations can account for finite time horizons (depreciation periods) and variable interest rates. A back-of-the-envelope PV calculation sheds light on the industry analysts' negative remark about CacheFlow cited in Section 4.1: If the appliance yields 15% bandwidth savings on a $1,200/month line ($180/month in cost savings) and if the annual interest rate is 5%, then the product's present value exceeds $40,000. However, if we assume a finite product life, we find that PV exceeds purchase price only for lifetimes of roughly 7 years or more assuming 50% bandwidth savings.

To put the model of this section in perspective, we briefly survey the actual costs of bandwidth in Michigan circa early 2000. Note that the general form of these costs does not correspond to the simple model of Section 4.2: Whereas that model assumed bandwidth costs proportional to traffic volume, in practice both external and local bandwidth at U-M

Table 4.2: Merit Networks Inc. prices of Internet connectivity for commercial and educational customers in U.S. dollars.

Technology & Bandwidth	Installation	Annual costs Edu	Annual costs Comm	$\$_B$ ($/(byte/sec)) Edu	$\$_B$ ($/(byte/sec)) Comm
Private Line					
56 Kbps	6,602	8,395	9,520	24.93	28.14
ISDN					
64 Kbps	3,763	7,484	8,609	19.18	21.99
128 Kbps	3,763	8,504	10,609	10.87	13.50
256 Kbps	9,880	10,377	13,217	6.79	8.57
384 Kbps	10,224	11,996	15,326	5.21	6.60
Fractional T1					
128 Kbps	7,307	14,077	16,182	18.05	20.68
256 Kbps	7,307	14,842	17,682	9.50	11.28
384 Kbps	7,307	15,352	18,682	6.55	7.93
768 Kbps	7,307	16,882	20,682	3.59	4.38
Full T1 line(s)					
1.5 Mbps	7,307	19,942	24,682	2.17	2.67
3.0 Mbps	9,962	35,344	40,163	1.91	2.17

are purchased in terms of capacity rather than usage. Our motive for considering the actual monetary costs of Internet and LAN bandwidth is Equation 4.6, which states a threshold condition for the viability of a shared cache in terms of the ratio of internal and external bandwidth costs.

Table 4.2 presents prices charged by Merit Networks, Inc. and corresponding bandwidth costs based on Equation 4.10. All present-value calculations assume a 5% annual interest rate.

The cost of 10 Mbps shared Ethernet installations at the University of Michigan provides a crude estimate of LAN bandwidth costs circa 2000. Table 4.3 presents LAN bandwidth costs based the University's internal prices. Prices shown are determined by the following formula:

$$\text{price} = 1.1 \times (\text{number of hosts} \times \$458 + \$23,000)$$

(By summer 2002 switched Ethernet running at much faster speeds (GbpsE) is becoming common, and at some institutions internal bandwidth prices might be two to four orders

Table 4.3: LAN bandwidth costs of 10 Mbps shared Ethernet at the University of Michigan. Data courtesy JoElla Coles of ITD.

number of clients	installation cost ($)	bandwidth per client (bytes/sec)	bandwidth cost ($/(byte/sec))
1	25803	1250000.0	0.020643
5	27819	250000.0	0.111276
10	30338	125000.0	0.242704
15	32857	83333.3	0.394284
20	35376	62500.0	0.566016
25	37895	50000.0	0.757900
30	40414	41666.7	0.969936
40	45452	31250.0	1.454464
50	50490	25000.0	2.019600
75	63085	16666.7	3.785100
100	75680	12500.0	6.054400

of magnitude lower [69].) Consistent with the assumptions of this section, we compute available bandwidth per LAN client for the idealized case of identical client behavior. Note that if we take any $\$_{Bp}$ from Table 4.2 and any C and $\$_{Bc}$ from Table 4.3, these will satisfy Equation 4.6 for any $C > 1$. (Again, we emphasize that actual internal and external bandwidth costs at U-M do not follow the simple usage-based proportionality assumption of Section 4.2, so this observation is at best suggestive.)

Some readers may object that technology costs fluctuate too rapidly to guide design decisions. While it is true that memory and bandwidth prices change rapidly, engineering principles based on technology price *ratios* have remained remarkably robust for long periods [70]. Because the main results of this section are stated in terms of ratios, it is reasonable to suppose that they are relatively insensitive to short-term technology price fluctuations.

4.4 A Detailed Model of Single Caches

The model assumptions and optimization procedures of Section 4.2 are problematic for several reasons: The workload model assumes an idealized steady state, ignoring such

Table 4.4: Notation of Section 4.4.

M	total number of requests
N	total number of distinct documents requested
x_t	document requested at virtual time t
S_i	size of document i (bytes)
$\$_t$	cost incurred if request at time t misses (\$)
$\$_M(s)$	storage cost of cache capacity s (\$)
D_t	set of documents requested up to time t
$P_t(i)$	priority of document $i \in D_t$
δ_t	priority depth function defined on documents in D_t (bytes)
$\$_A(s)$	total miss cost over entire reference sequence (\$)

features as cold-start effects and temporal locality. The model assumes that caches use Perfect-LFU replacement. Production caches, however, nearly always use variants of LRU; many cache designers reject Perfect LFU because of its higher time and memory overhead. Real-world storage and miss costs are not simple linear functions of capacity.

In this section I describe a method for determining the optimal size of a *single* cache that suffers from none of the above deficiencies. I assume that 1) workload is described by an *explicit sequence* of requests; 2) an *arbitrary* miss cost is associated with each request; 3) the cache uses one of a large family of replacement policies that includes LRU and a variant of Perfect LFU; and 4) the cost of cache storage capacity is an arbitrary nondecreasing function. The first assumption allows us to apply this algorithm to traces, e.g., proxy logs. The second allows us to assess different miss costs for documents of different size, or for requests to the same document during peak vs. off-peak hours. The third assumption means that my method is applicable to the vast majority of production Web caches, and the fourth allows us to consider any reasonable storage cost function.

Cache workload consists of a sequence of M references $x_1, x_2, \ldots, x_M$ where subscripts indicate the "virtual time" of each request: If the request at time t is for document i, then $x_t = i$. Associated with each reference is a nonnegative miss cost $\$_t$. Whereas document sizes are constant, the miss costs associated with different requests for the same document need not be equal: If $x_t = x_{t'} = i$ for $t \neq t'$ we require $S_{x_t} = S_{x_{t'}} = S_i$, but we permit $\$_t \neq \$_{t'}$ (e.g., miss costs may be assessed higher during peak usage periods). Finally, the cost of cache storage $\$_M(s)$ is an arbitrary nondecreasing function of cache capacity s; this permits us to consider, e.g., fixed costs.

The set of documents requested up to time t is denoted $D_t \equiv \{i : x_{t'} = i \text{ for some } t' \leq t\}$. A scalar *priority* P_t is defined over documents in D_t; two documents never have equal priority: $P_t(i) = P_t(j)$ iff $i = j$. Informally, the *priority depth* δ_t of a document $i \in D_t$ is the smallest cache size at which a reference to the document will result in a cache hit. Formally,

$$\delta_t(i) \equiv S_i + \sum_{h \in H_t} S_h \quad \text{where} \quad H_t \equiv \{h \in D_t : P_t(h) > P_t(i)\} \tag{4.11}$$

The priority depth of documents not in D_t is defined to be infinity. Priority depth generalizes the familiar notion of LRU stack distance [108] to the case of non-uniform document sizes and general priority functions (the use of stack distances to measure temporal locality is discussed in Section 3.2). Let

$$\$_A(s) \equiv \sum_{t=1}^{M} \$_t I_t(s) \quad \text{where} \quad I_t(s) \equiv \begin{cases} 0 & \text{if } s \geq \delta_t(x_t) \\ 1 & \text{otherwise} \end{cases} \tag{4.12}$$

denote aggregate miss cost over the entire reference sequence as a function of "size" parameter s (note that this is simply a kind of cumulative distribution). For every input sequence, $\$_A(s)$ is equal to the total miss cost incurred by a cache of size s whose eviction order is defined by P provided that $s \geq \max_i S_i$, and that the cache removal policy satisfies the *inclusion property*, meaning that a cache of size s will always contain any smaller cache's contents. The second requirement is familiar from the literature on stack distance transformations of reference streams [23,108,128,155]; replacement policies with this property are sometimes known as "stack policies".[1] The first requirement is necessary because aggregate miss cost is monotonic only for cache sizes capable of holding any document. Mattson et al. describe the relationship between the cumulative distribution of stack distances and cache hit rate [108]; Equation 4.12 simply generalizes this to the case of non-uniform document sizes and non-uniform miss costs.

Given $\$_A(s)$ we can efficiently determine a cache size $s*$ that minimizes total cost $\$_A(s*) + \$_M(s*)$. Because storage cost is nondecreasing in cache capacity, we need not

[1] LRU and the variant of Perfect LFU that caches a requested document only if it has sufficiently high priority ("optional-placement Perfect LFU") are stack policies; FIFO and mandatory-placement LFUs are not [108]. The most interesting recent Web cache removal policies—GD-Size [42], GDSF [10], swLFU [86,87], LUV [18] and GD* [82]—do not satisfy the inclusion property, and therefore the fast single-pass simulation methods described in Section 4.5 cannot be applied to them.

consider total cost at all cache sizes: $\$_A(s)$ is a "step function" that is nonincreasing in s, with at most M "steps," and minimal overall cost must occur at one of them. We may therefore determine a (not necessarily unique) cache size that minimizes total cost in $O(M)$ time.

In summary, my method for computing the optimal size of a single cache from a trace is as follows: Given document sizes, a suitable priority function, and a reference stream, compute the priority depth of each reference using Equation 4.11. Compute aggregate miss cost as a function of cache size using Equation 4.12. Finally, inspect the "steps" in this function's domain; $s*$ is guaranteed to occur at one of them.

At first glance, it might appear that the bottleneck in this approach is the computation of priority depth (Equation 4.11). A straightforward implementation of a priority list, e.g., as a linked list, would require $O(N)$ memory and $O(N)$ time per reference for a total of $O(MN)$ time to process the entire sequence of M requests. For reasonable removal policies, however, it is possible to perform this computation in $O(M \log N)$ time and $O(N)$ memory using an algorithm reminiscent of those developed for efficient processor-memory simulation [23, 128, 155]; I describe my priority-depth algorithm in Section 4.5. Given a pair $(\delta_t(x_t), \$_t)$ for each of M requests, we can compute $\$_A(s)$ after sorting these pairs on δ in $O(M \log M)$ time and $O(M)$ memory. This "post-processing" sorting step is therefore the computational bottleneck for any trace workload, in which $M \geq N$. By contrast, a simulation of a *single cache size* would require $O(M \log N)$ time for practical removal policies.

4.5 Fast Simultaneous Simulation

I now describe an algorithm that computes δ_t for each of M references in $O(M \log N)$ time and $O(N)$ memory by making a single pass over a reference sequence. Daniel Reeves and I developed this algorithm together. The crucial insight that stack distances can be computed in logarithmic time is due to Reeves, who rediscovered a cleaner and simpler version of Bennett & Kruskal's scheme [23]. Because it computes $\$_A(s)$ at the additional cost of sorting the output, in effect this algorithm simultaneously simulates *all* cache sizes of possible interest. An efficient method is necessary to compute stack distances for real traces, in which M and N can both exceed 10 million [87]. To make the issue concrete, whereas

a naïve $O(MN)$ priority depth algorithm required over five days to process 11.6 million requests for 5.25 million documents, my $O(M \log N)$ algorithm completed the job in roughly three minutes on the same computer.

For this method to work, we require that the priority function P corresponding to the cache's removal policy satisfy an additional constraint: The relative priority of two documents may change only when one of them is referenced. This is not an overly restrictive assumption; indeed, some researchers regard it as a requirement for a practical replacement policy, because it permits requests to be processed in logarithmic time [18].

We represent documents in the set D_t as nodes of a binary tree, where an inorder traversal visits document records in ascending priority. Each distinct document requires one node, hence the $O(N)$ memory requirement. Each node stores the aggregate size of all documents in its right (higher-priority) subtree; we can therefore recover $\delta_t(i)$ by traversing the path from document i's node to the root (see Figure 4.2). To process a request, we output the referenced document's priority depth, remove the corresponding node from the tree, adjust its priority, and re-insert it. Tree nodes are allocated in an N-long array indexed by document ID, so locating a node takes $O(1)$ time. All of the other operations use $O(\log N)$ time, for a total of $O(M \log N)$ time to process the entire input sequence. Cormen et al. describe similar ways of augmenting data structures; Exercise 14.2-4 on page 311 of their algorithms text is strongly reminiscent of the method used here [48] (this appears in the first edition of the text as Exercise 15.2-4 on page 289 [47]).

For all removal policies of practical interest, a document's priority only *increases* when it is accessed. A simple binary tree would therefore quickly degenerate into a linked list, so I use a splay tree to ensure (amortized) logarithmic time per operation [93, 146, 154]. It is possible to maintain the invariant that each tree node stores the total size of all documents represented in its right subtree during insertions, deletions, and "splay" operations without altering the overall asymptotic time or memory complexity of the standard splay tree algorithm. A simple ANSI C implementation of our priority depth algorithm is available [84]. Martin Arlitt of Hewlett-Packard Labs reports that my simple, unoptimized implementation of the Reeves-Kelly priority depth algorithm computes stack distances for a very large trace roughly six times faster than his own highly-optimized implementation of a slower algorithm (19 hours vs. roughly 5 days).

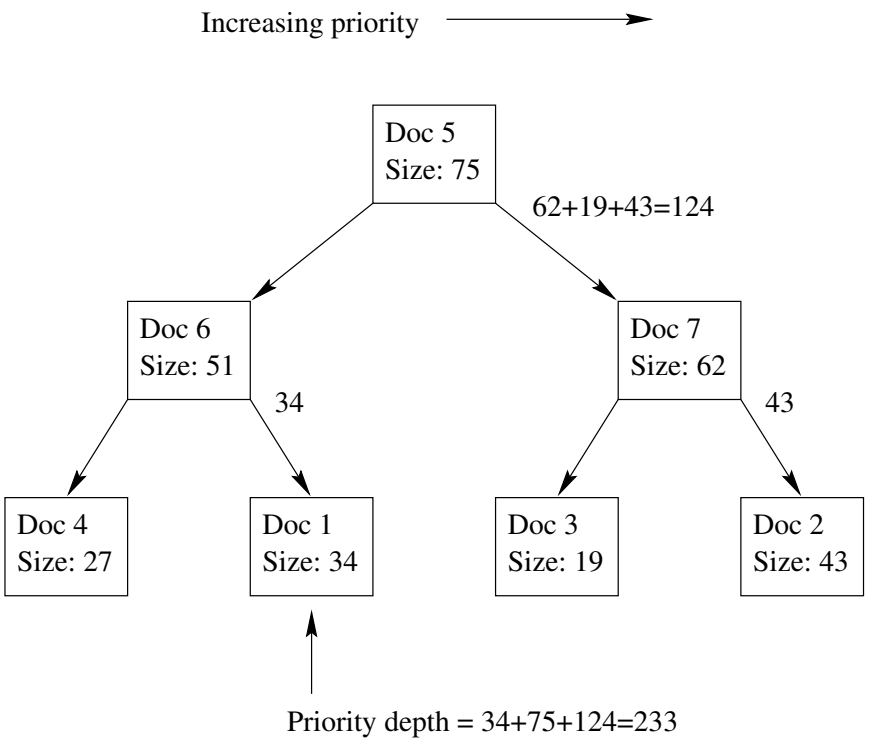

Figure 4.2: Recovering priority depth. In this example, document 1 has been referenced. We initialize an accumulator to the size of document 1 (in this example, 34) plus the sum of sizes of all documents in its right subtree (in this example, zero). We then walk up to the root. When we move from a right child to its parent (e.g., from document 1 to document 6) we do nothing. However when we move from a left child to its parent (e.g., from document 6 to document 5) we add to the accumulator the size of the parent (75) and the sum of sizes of all documents in the parent's right subtree (124). When we reach the root, the accumulator contains the sum of the sizes of the referenced document and all higher-priority documents, i.e., the priority depth of the referenced document (233).

Reeves and I devised our efficient priority depth algorithm before we became aware of similar (though less general) techniques dating back to the mid-1970s [23, 128, 155], which appear not to be widely used in Web-related literature. To the best of our knowledge, no recent papers containing stack depth analyses [4, 15, 19, 20, 103] cite the most important papers on efficient stack distance computation [23, 128, 155]). The idea of using splay trees is suggested by Thompson, who used AVL trees in his own work and reports that AVL-based implementations are complex and error-prone [155]. The Reeves/Kelly priority depth algorithm is simpler than those described in the processor-memory-caching literature because it ignores associativity considerations and assumes that cached data is read-only. It is better suited to Web caching because it handles variable document sizes and arbitrary miss costs.

4.6 Numerical Results

To illustrate the flexibility and efficiency of the Reeves/Kelly priority depth algorithm, I use it to compute *complete* stack distance transformations and LRU hit rates at *all* cache sizes for six four-week NLANR [66] Web cache traces summarized in Table 4.5 and described more fully in Table 3.3. Similarly detailed results rarely appear in the Web caching literature. Almeida et al. present complete stack distance traces for four Web server workloads ranging in size from 28,000–80,000 requests [4]. They furthermore note that the marginal distribution of a stack distance trace is related to cache miss rate, but their discussion assumes uniform document sizes. Arlitt et al. present the only stack depth analysis of large traces (up to 1.35 billion references) of which I am aware [12, 13]. Complete and exact calculations may have been viewed as computationally infeasible. All of the results presented here, however, were computed in a total of under five hours on an unspectacular machine—far less time than was required to download our raw trace data from NLANR.[2]

LRU stack distance, a standard measure of temporal locality in symbolic reference streams, is a special-case output of our priority depth algorithm when all document sizes and miss costs are 1. Section 3.2 explains the relationship between LRU stack distances and

[2]We used a Dell Poweredge 6300 server with four 450-MHz Intel Pentium II Xeon processors and 512 MB of RAM running Linux kernel 2.2.12-20smp.

Table 4.5: Traces derived from access logs recorded at six NLANR sites, 1–28 March 1999. Run times shown are wall-clock times to compute given quantities, in seconds. The run times sum to under four hours, ten minutes.

	BO1	PA	PB	SD	SV	UC
# documents (millions)	5.25	4.90	9.82	8.64	9.38	7.62
# requests (millions)	11.58	13.55	19.80	37.09	23.74	26.02
max S_i (MB)	218.6	104.9	218.7	175.0	107.4	175.0
unique bytes (billions)	104.5	76.0	188.3	204.9	159.1	150.1
bytes requested (billions)	236.2	220.7	383.1	620.3	412.9	397.5
run times (sec):						
priority depths	230	288	399	872	547	587
stack distances	249	341	403	1117	581	716
HR(size)	309	414	497	1439	712	903
BHR(size)	314	423	522	1461	740	913

temporal locality. Intuitively, the LRU stack distance of a given reference is the number of distinct documents that were accessed between the given reference and the previous reference to the same document. If most hits occur at a shallow depth in the LRU stack, this indicates high temporal locality, and suggests that even a small LRU cache will yield a high hit rate. Mattson et al. is the classic reference on stack distance analysis [108]; Almeida et al. [4] and Arlitt & Williamson [15] apply the technique to Web traces.

The frequency distribution of stack distances from our six traces is shown in Figure 4.3 (top). Frequency distributions visually exaggerate temporal locality, particularly when (as is common in the literature) the horizontal axis is truncated at a shallow depth. The situation does not improve if we aggregate the observed stack distances into constant-width bins, because as Arlitt & Williamson have noted, the visual impression of temporal locality created depends on the granularity of the bin sizes we choose [15]. The clearest and least ambiguous way to present these data is with a cumulative distribution, as on the bottom of Figure 4.3, from which order statistics such as the median and quartile stack distances are directly apparent. For all six of our NLANR traces the median stack distance is 100,000 or greater, indicating weak temporal locality; this is not surprising, because L1 (browser) and L2 (proxy) caches filter most of the temporal locality from client reference streams before they reach NLANR's L3 (network backbone) caches.

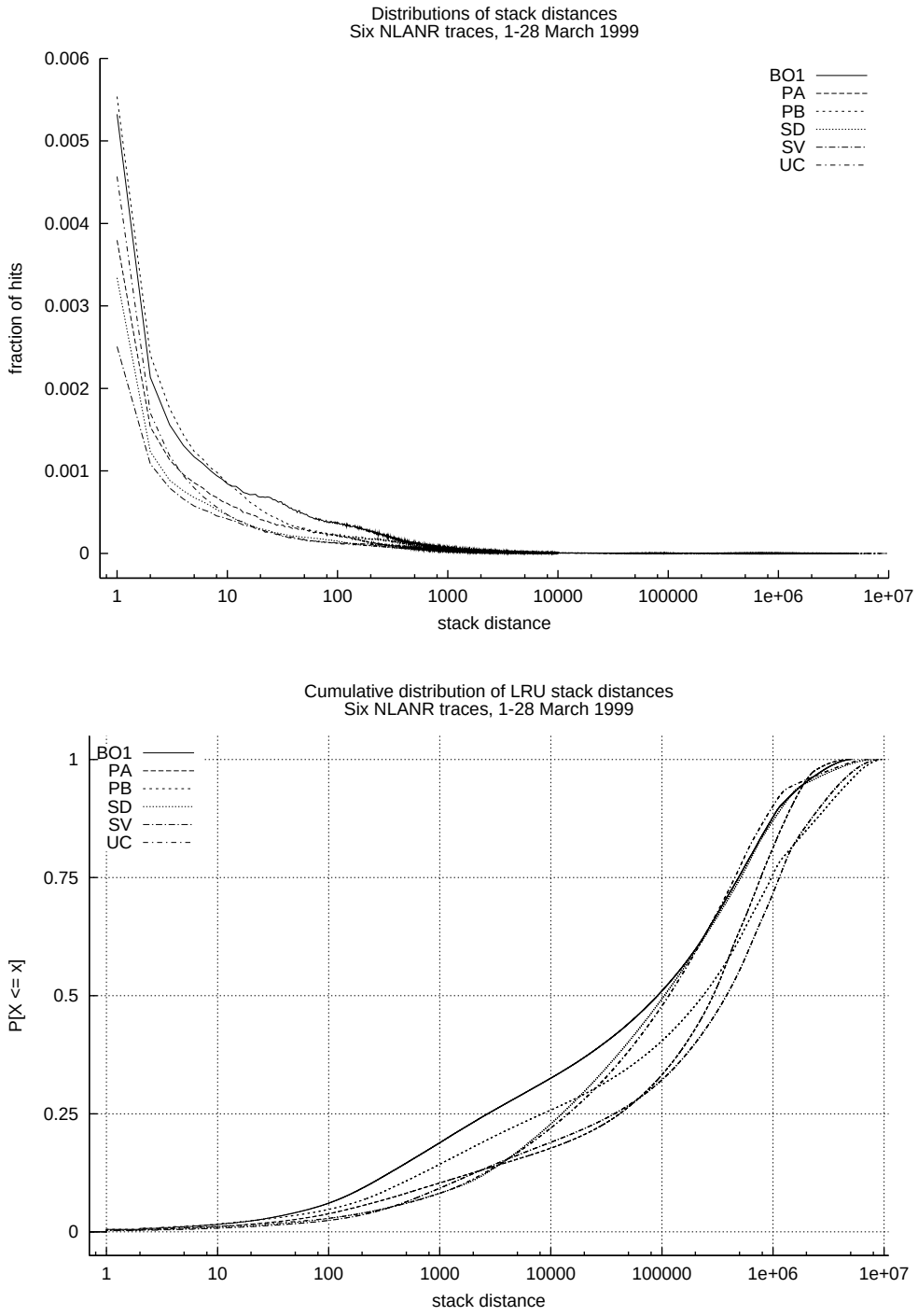

Figure 4.3: Frequency distribution (top) and cumulative distribution (bottom) of LRU stack distances in six traces. Compare these data with Table 10 and Figure 8 of Arlitt & Jin [13]; temporal locality is far weaker in our network cache traces than in their very large server workload.

Figure 4.4 shows LRU hit rates and byte hit rates at all cache sizes for our six Web traces. For the workloads considered, exact performance measurements at all cache sizes appear to offer little *visual* advantage over the customary technique of interpolating measurements taken at regular intervals (e.g., 1 GB, 2 GB, 4 GB, etc.) via single-cache-size simulation. However, since exact hit rate functions may be obtained at very modest computational cost, it is not clear that a less precise approach offers any advantage, either.

4.7 Discussion

The idealized model of Section 4.2 is useful for computing optimal cache sizes only to the extent that its underlying workload and cost assumptions are valid. Breslau et al. argue that the independent reference model is approximately accurate for many purposes [33], but Almeida et al. have describe several shortcomings of this model and propose more accurate alternatives [4]. The model of Section 4.2 assumes a homogeneous population of lower-level caches; Wolman et al. explore in detail the implications of sharing among *heterogeneous* client aggregates, and furthermore consider document modification rates, which I ignore [171, 172]. The primary formal weakness of my model of hierarchical caching is its simple linear cost model. In many cases of practical interest, memory and bandwidth costs are step functions that do not admit accurate linear approximations. Finally, I ignore the low-level aspects of Web operation. Feldmann et al. report that details such as bandwidth heterogeneity and aborted transfers can negate the bandwidth savings that proxy caching would otherwise yield [62].

The single-cache optimization method of Section 4.4 does not model cache consistency mechanisms and therefore does not distinguish between "fast hits," in which the required payload is obtained from cache without revalidation, and "slow hits," in which successful revalidation entails a round-trip to the origin server but no payload data is transferred. In other words, miss costs can be assessed only for payload transfers; successful revalidations entail no payload transfer and therefore are assigned zero cost. This is problematic in cases where latency drives the cost model and where round-trip time is large compared to payload transfer time. However the method is well suited to bandwidth-driven cost models, to latency-based costs in low-RTT, low-bandwidth communications media, and to workloads with large payloads (e.g., entertainment-on-demand workloads).

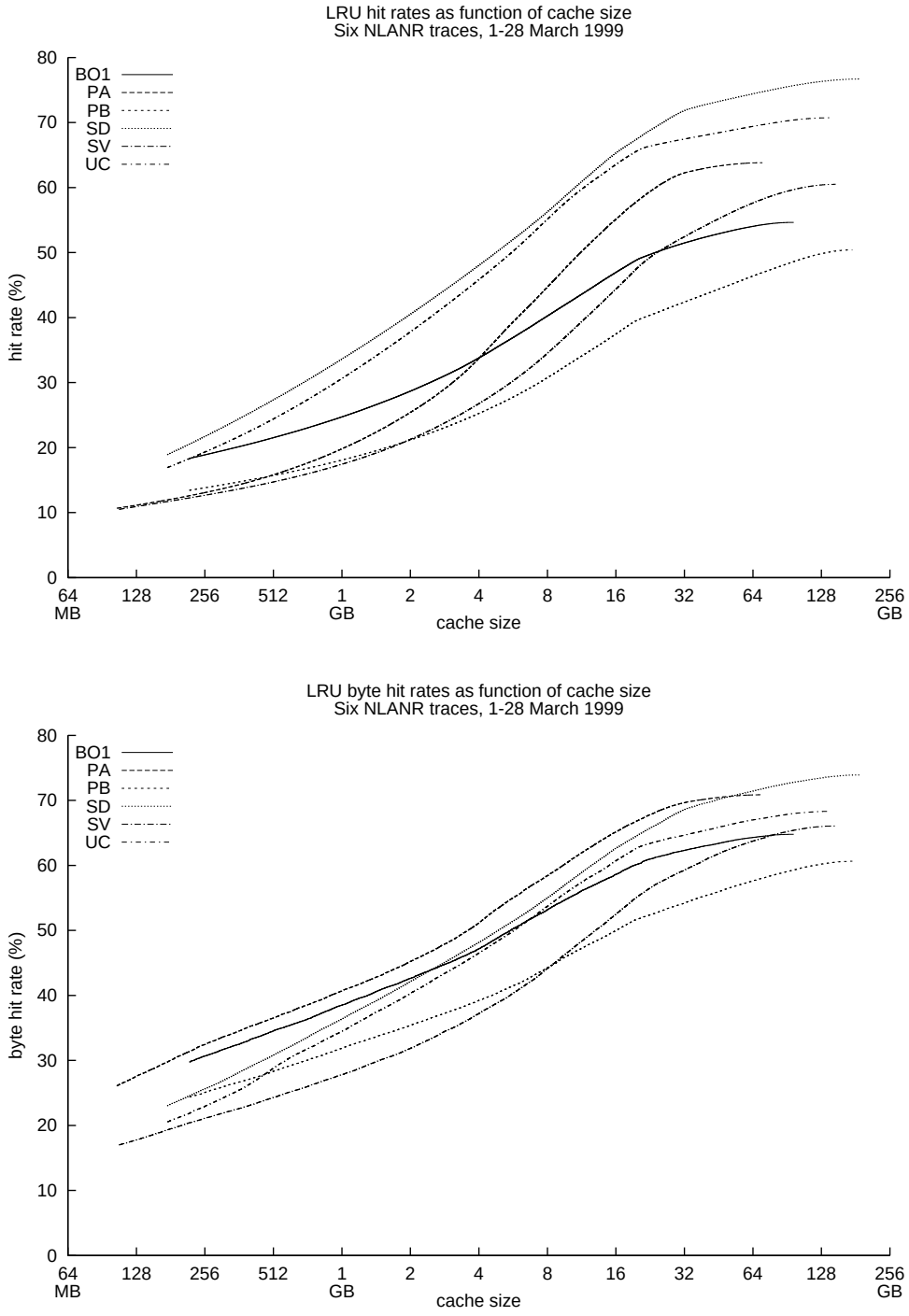

Figure 4.4: Exact hit rates (top) and byte hit rates (bottom) as function of cache size for six large traces, LRU removal. Fast simultaneous simulation method yields correct results only for cache sizes $\geq$ largest object size in a trace; smaller cache sizes not shown.

Another problem with the method is that it does not account for uncertainty in expected workload; it implicitly assumes that a trace recorded in the past represents future reference patterns. Ideally we would like to incorporate uncertainty into the capacity planning process directly, to support risk-averse design in a principled way. One step in this direction would be to explore the relative importance of different aggregate workload characteristics, e.g., the distributions of document popularity and size, on optimal cache size. If simple relationships are found, e.g., between mean popularity-weighted document size and optimal cache size, then it may be possible to account for risk aversion straightforwardly.

Aside from these issues, my workload-driven single-cache optimal sizing method is usable in its present form. One obvious application is the determination of optimal browser cache sizes. Douceur & Bolosky's study of disk usage on a large corporate network indicates that roughly half of all PC disk space is unused [57], so there's no shortage of potential browser cache space in rich-client environments. Resource-constrained thin clients such as wireless palmtop browsers and diskless set-top boxes provide a more compelling context in which to apply my optimization methods, because neither storage nor bandwidth are cheap or plentiful in such environments.

The single-cache optimization method of Section 4.4 is fully general in the sense that per-reference miss costs may reflect any criteria whatsoever. In particular, they may reflect the preferences of system stakeholders, and in this case they permit more flexibility than the value model I defined in my investigation of preference-sensitive removal policies. Whereas the value model of Section 3.1.1 associates miss penalties with *documents*, here we associate them with *references*. This flexibility allows us to assess different miss costs on different references to the same document using a wide variety of criteria, e.g., time of day, server load, network load, and the client who issues each request. This in turn allows us to choose cache sizes well suited not merely to the order in which accesses are made and the sizes of accessed items but also to the *importance* of accesses, which we may define as we please.

CHAPTER 5

Cache Analysis, Traces, and Simulation

While simplified analytic workload models and publicly-available trace data are sufficient for the investigations we have considered so far, they cannot support the full range of research questions considered in this thesis. This chapter explains why it was necessary to develop and employ the novel workload measurement technique described in Chapter 6 and describes the computational challenges of large data sets. Purely analytic investigation of removal policies yields results too weak to guide cache design, and therefore we must often resort to empirical and numerical methods. The sections that follow explain the shortcomings of analytic alternatives to cache simulation, discuss problems with publicly-available Web trace data, review existing trace-collection methodologies, and sketch the design of a parallel cache simulator capable of handling large traces.

5.1 Analytic Modeling

An *offline algorithm* receives all of its input at once. An *online algorithm* receives its input in installments. Cache replacement policies are instances of the latter, because a cache must dispose of its current request before receiving the next. When we speak of "offline removal policies" we refer to policies that exploit clairvoyant knowledge of future accesses in making eviction decisions; such policies, of course, are not realizable in practice, but they can provide upper bounds on the performance of any removal policy governing a finite cache. Belady describes an optimal offline removal policy for the special case of uniform page sizes and uniform miss costs [22] and Hosseini-Khayat considers optimal offline removal in the general case of non-uniform page sizes and miss penalties [77].

The standard framework for analyzing caching and paging policies and other online algorithms is Sleator & Tarjan's *competitive analysis* [145]. We say that an online algorithm is c-competitive if the cost it incurs on any input is not more than c times that of the optimal *offline* algorithm plus a constant; c is called the algorithm's *competitive ratio*. Competitive analysis assumes an adversarial workload model and provides worst-case performance bounds that often underestimate performance under real workloads.

If page sizes may vary, the best competitive ratio achievable by any deterministic online replacement policy is $k+1$, where k is equal to cache size divided by smallest document size [79]; Greedy-Dual Size attains this bound and is therefore said to be *online optimal* [42]. For some minimal-cost caching problems, randomized algorithms with a competitive ratio of $O(\log^2 k)$ are available [79]. Kimbrel extends competitive analysis to caching systems with weak (expiration-based) consistency mechanisms [90]. Because k is typically on the order of 10 million or more, the competitive analysis properties of an algorithm are unlikely to sway a cache designer's choice of removal policy. Furthermore, online-optimal algorithms like LRU and GD-Size are in practice observed to perform far better than competitive analysis suggests.

In addition to weakening the performance bounds we obtain from competitive analysis of paging systems, non-uniform page size and miss cost complicate analysis enormously. Whereas the optimal offline removal policy for the special case of uniform page size and page fault penalty ("longest forward distance") has been known for decades and is both straightforward and computationally tractable [22], the optimal offline policy for non-uniform size and cost has only recently been described, and the computational problem is NP-complete [77]. In other words, even if we could somehow supply a Web cache with clairvoyant knowledge of future access patterns, it is computationally infeasible for the cache to exploit this knowledge to full advantage.

A different analytic approach to understanding reference streams and the performance of paging policies that process them is to develop workload models and derive performance results for various cache management strategies directly from these models. Examples of workload models include the independent reference model, the LRU stack model, and the working set model (see Rau [137] and the references therein for an excellent review of these models from the processor-memory literature). Knuth analyzes optimal offline removal

assuming *random* page references, and devotes some attention to the LRU stack distance model [91].

While superior in most respects to the competitive-analysis approach, analytically tractable workload models often poorly predict the performance of Web removal policies, and therefore trace-driven simulation plays an essential role in replacement policy evaluation. Existing synthetic workload generators and benchmarks such as SURGE [20], WebPolygraph [134], and SPECweb [149] cannot provide acceptable inputs for the kinds of trace-driven simulations I require because they make no attempt to mimic a phenomenon crucial to my investigations: aliasing. Synthetic generators assume a one-to-one relationship between content names (URLs) and content (reply payloads), and therefore cannot shed light on the performance implications of the more complex URL/payload relationship that exists in the wild. We therefore require traces of real workloads collected *in situ*.

5.2 Trace-Collection Methods and Available Traces

This section explains my trace data requirements in terms of my research questions. After explicitly stating my requirements I review existing trace-collection methods and publicly-available data sets.

5.2.1 Requirements

My primary goal is to develop efficient and cost-effective ways to serve the workload submitted to the World Wide Web by content providers and content consumers. Researchers have investigated in detail the workload placed on *components* of the World Wide Web, e.g., servers, proxies, and networks [10–15,58,60,61,171,172]. Little is known, however, about the fundamental exogenous workload placed on the Web *as a system*. At the server end, exogenous workload consists of the universe of available data and the names (URLs) through which it is published. Padmanabhan & Qiu investigate content creation and modification dynamics at a large, busy Web site [130]; this is the only systematic study of available content of which I am aware. At the client end, patterns of client accesses constitute the exogenous workload. A handful of studies, reviewed in Section 5.2.4, have measured and analyzed client workload directly, but many questions remain open.

In particular, interactions between dedicated (browser) and shared intermediate (proxy) caches in storage/retrieval systems like the Web are not well understood. Access patterns in distributed file systems exhibit so little sharing across client reference streams that even small client caches dramatically reduce the maximal hit rates of shared intermediate caches [121]. This observation may not be true of the Web, where sharing might be much stronger. To understand the impact of browser cache size on both browser and proxy cache performance we require *complete* client reference streams, unfiltered by browser caches. More generally, we want traces that record the system's exogenous workload unaltered by the system *currently* serving it, because such traces permit the bottom-up simulation of *any* system that might serve the workload. They allow us to explore as many points in the space of possible designs as our computational resources permit.

The range of questions I wish to address requires a detailed record of (request, reply) transactions for all requests issued by a large population of clients. To model conventional URL-indexed caches, it is necessary to know the (possibly anonymized) URL for each request. To determine upper bounds on cache hit rates and understand the impact of content naming practices on cache performance, it is necessary to identify cases where the reply data payloads in different transactions are the same; an anonymized payload digest is sufficient for this purpose. To model a variety of cache freshness heuristics and revalidation policies, it is necessary to record metadata returned by origin servers in replies.

5.2.2 Server Logs

It is straightforward for researchers to obtain origin server logs from a variety of different sources [14, 104]. Furthermore server logs can be extraordinarily large [13], and since some popular server software is available in source form it is relatively easy to instrument servers to collect very detailed traces. A server, however, sees only a fraction of all the transactions involving the clients that visit it, so server logs are unsuitable for investigation of browser/proxy cache hierarchies.

5.2.3 Proxy Logs and Sniffers

To record transactions involving large numbers of users and servers, researchers sometimes employ packet sniffers [61, 62, 148] or proxy logs [56]; both typically record trans-

actions that pass between a pool of clients and the Internet. Widespread use of caching proxies can complicate the sniffer approach because a sniffer located between a caching proxy and the Internet does not record requests served from the proxy cache. The logs of a caching proxy do not suffer from this problem, but such logs do not necessarily reflect the payloads that origin servers would provide: Proxies might serve stale content unless they revalidate payloads with the origin server with every cache hit. Moreover, proxy and sniffer traces do not record client requests served from browser caches.

Implementors and administrators regard proxy logs primarily as security features; consequently the logging capabilities of most proxies are not well suited to research. Logs rarely record all of the data available to the proxy and typically omit information crucial to accurate trace-driven simulation. In particular, they fail to record cache-related HTTP metadata in reply headers and "`META http-equiv`" tags within HTML files, reply payloads or hashes thereof, and accurate, high-resolution timestamps. Davison and Cáceres et al. have documented the shortcomings of conventional proxy log formats [39, 53].

When a single logical cache consists of multiple host machines, e.g., when Microsoft's Cache Array Routing Protocol (CARP) [50] is used, new problems can arise. The timestamps in MS Proxy Server access logs have one-second resolution, and the system clocks in a CARP array are seldom carefully synchronized. Furthermore, a single client's requests are load-balanced across the array. It is therefore impossible to determine the true order in which references arrive at the proxy array, making the logs useless for removal policy evaluation, which is sensitive to the exact arrival order of references.

5.2.4 Instrumented Clients

In a few cases, researchers have instrumented Web browsers to collect true client traces unfiltered by browser caches. Catledge & Pitkow recorded a client trace at the Georgia Tech Computer Science department in 1994, and researchers at Boston University's CS department recorded a similar trace in 1995 [43, 52]. Both traces are remarkably rich, recording a wide variety of user-interface events unavailable outside the browser. Together, these two traces have supported a number of interesting studies [26, 43, 51, 52]. In principle, client traces can support realistic bottom-up explorations of cache hierarchies and shed light on user interactions invisible outside the client. Researchers cannot easily instrument popular

browsers today because source code is unavailable, but a client proxy such as Medusa [92] can collect much of the same data. However, it remains difficult to deploy an instrumented browser among a large and representative sample of Web users. Furthermore, if such a feat were possible it would still be difficult to synchronize large numbers of client clocks, especially on resource-constrained thin clients, and accurate simulation of a cache hierarchy is impossible without precise event timestamps. Finally, elaborate browser instrumentation may not be an option in memory-constrained thin clients; Adya et al.'s recent study of mobile client browse patterns relies on server logs [1]. To the best of my knowledge, no instrumented-client traces have been collected for research purposes since 1995. (A 1999 sequel to the original Boston University study used a trace that did not reflect browser cache hits [19].) Alexa (now a subsidiary of Amazon.com) has instrumented large numbers of Web browsers through a downloadable toolbar that reports surfing activity to a central logging site, but the traces collected through this proprietary method are not used for research purposes and neither the collection method nor the data logged are described in detail [2].

5.2.5 Publicly-Available Traces

Some of the most detailed and interesting Web workload traces have not been published, to protect proprietary corporate information and end-user privacy [12, 85, 115, 130, 171]. Published traces are often anonymized to conceal the identities of clients, the resources (URLs) accessed, or both. In most cases anonymization does not diminish the scientific value of traces, but it can destroy useful information if performed too aggressively: The NLANR access logs used in Chapters 3 and 4, for instance, anonymize client identities differently *each day*, making it impossible to extract individual client reference streams more than one day long [66].

Another problem with the widely-used NLANR traces is that the total number of human users whose requests pass through *all* of NLANR's "worldwide backbone cache system" is now known to be remarkably small. In 1999 Duke University researchers estimated the total NLANR end-user population at "over 13,000" based on an analytic model, and at 25,000–43,000 based on analysis of Squid's "Via" and "X-forwarded-for" headers [67]. At my request Duane Wessels of NLANR counted 98,144 unique "leaf" IP addresses in

these logs for the first 26 days of October 2000 [164]. Considering the prevalence of dynamically-assigned IP addresses, this probably represents an over-estimate of the true end-user population.

Brian Davison has compiled a comprehensive index of available Web traces [54], many of which are stored at the Web Characterization Repository [74]. Roughly half are server logs. With the exception of the NLANR traces, which are published daily, the most recent trace dates to November 1999. Most of the available proxy traces are small in terms of the number of transactions recorded, the number of clients involved, or the duration of data-collection. In several cases the "infinite cache size" (sum of distinct document sizes) is so small that a cache of reasonable capacity would never need to evict a document; such traces are useless for comparing removal policies. *None* of the traces contain reply-payload checksums, and most lack request and reply metadata. Finally, many publicly-available traces were collected in idiosyncratic environments, e.g., academic computer science departments and computer corporations, and *all* were generated by heavyweight clients (full-featured browsers running on desktop PCs or engineering workstations). Such traces may not be representative of workloads generated by typical end-users on the memory- and bandwidth-constrained browsers that are proliferating as the Web expands onto set-top boxes and wireless handheld devices.

5.2.6 Summary

To summarize, the few existing Web client traces are several years old, reflect the requests of computer science students, and are small in comparison with server and proxy traces. By contrast server and proxy traces are often large and sometimes describe more representative user populations but typically omit reply metadata and references served from browser caches. Section 6.1 describes a collection methodology that combines some of the advantages of client and proxy traces and explains how this technique was used to measure Web client workload on an unprecedented scale.

5.3 Efficient General Simulation

Efficient single-pass simultaneous simulation to determine aggregate miss cost as an exact function of cache size for an explicit workload (sequence of requests) is possible only for a subset of interesting removal policies, and for cache sizes no smaller than the largest referenced data object (see Sections 4.4 and 4.5 for details). To simulate small caches and the full range of interesting removal policies, we must resort to conventional methods, i.e., we must simulate one cache size at a time.

Surprisingly, in contrast to the large literature on processor-memory cache simulation, there seems to be little literature on Web cache simulator design (i.e., simulation of fully associative caches with read-only documents of variable sizes and weak consistency mechanisms). I have identified only one paper that describes in detail the workings of a Web cache simulator [174]. Two other cache simulators are publicly available, but their accompanying documentation does not describe the rationale behind their designs [21, 40]. None of these simulators exploit parallel hardware architectures, and none seem explicitly designed to efficiently process very large traces. On the contrary, the WebCASE simulator described by Zhang et al. is implemented in Perl and emphasizes a graphical interface [174], and the author of the Wisconsin Web Cache Simulator reports that one of her simulators can handle only up to 2 million requests at a time [42, page 196].

In this section I briefly describe a general-purpose Web cache simulator that generated many of the results presented in Chapter 3. My design decisions were guided by the experience of several previous implementations, discussions with Yee Man Chan, and folklore passed along informally at workshops and conferences; like every other Web caching researcher I've met, I learned the art of cache simulation on the street. I do not claim that the design presented here is optimal. Yet, it is exemplary in that it is both relatively simple and able to exploit the computational resources of modern shared-memory computers.

Main memory is often the limiting resource in Web cache simulation. CPU utilization drops below 5% whenever a simulator begins to swap, dramatically extending the run times of large simulations. Fast simulation is essential for evaluating cost-weighted removal policies because we must sample many random cost functions to evaluate such policies. (The comparison of GD-Size, GDSF, and A-swLFU presented in Figure 3.7 required *sixty CPU-days* to compute.) Therefore all of the data required for a simulation run must fit

into main memory. Not many machines at the University of Michigan have enough main memory to support large-trace simulations: An early-2000 survey of 78 UltraSPARCs on the CAEN network revealed that 43 have 64 MB of RAM, 33 have 128 MB, and two heavily-used cycle servers have 256 MB each. Under reasonable assumptions, four of the six NLANR traces described in Table 3.3 require over 256 MB for a single simulation run. We must therefore restrict ourselves to the relatively few computers on campus with large main memories. In nearly every case these machines also happen to have multiple processors, so ideally we would like to parallelize the simulation over multiple CPUs.

Multiple cache simulations are trivially parallelizable, in the sense that simulations of different removal policies, cache sizes, traces, or cost functions can be run independently. However, naïve simulations of the *same trace* run as independent processes waste precious RAM if each process has an identical copy of certain data, e.g., a large lookup table of document sizes. Furthermore this redundant data is *read-only*, suggesting a design in which multiple processes or threads simultaneously simulate the same trace on a multiprocessor sharing access to a single copy of read-only data.

In my design, a simulator process first reads trace data and constructs two large read-only lookup tables: one of document sizes, and one containing the sequence of document references to be simulated. The main thread then spawns several worker threads that will independently simulate different cache sizes, removal policies, or cost functions. Each worker thread requires private read/write memory for maintaining simulated cache contents. The number of workers is chosen to be as high as possible subject to the constraint that the total size of all workers' private memory plus the shared tables not exceed available physical memory. Modern operating systems such as Solaris, Linux, and Irix automatically assign the worker threads to different processors. This approach ensures that all CPUs are utilized provided that sufficient memory is available for an extra simulator thread; similarly, physical memory will be exploited so long as a free processor is on hand.

My simulator's memory requirements vary with trace characteristics and also removal policy; a complex policy like A-swLFU, for instance, requires more memory than LRU. The memory requirements for LFU, GD-Size, swLFU and GDSF in my current implementation are given by the following expression (assuming 32-bit machine words):

$$\#\text{ bytes} \quad = \quad 8N + 4M + T(4S + 16N)$$

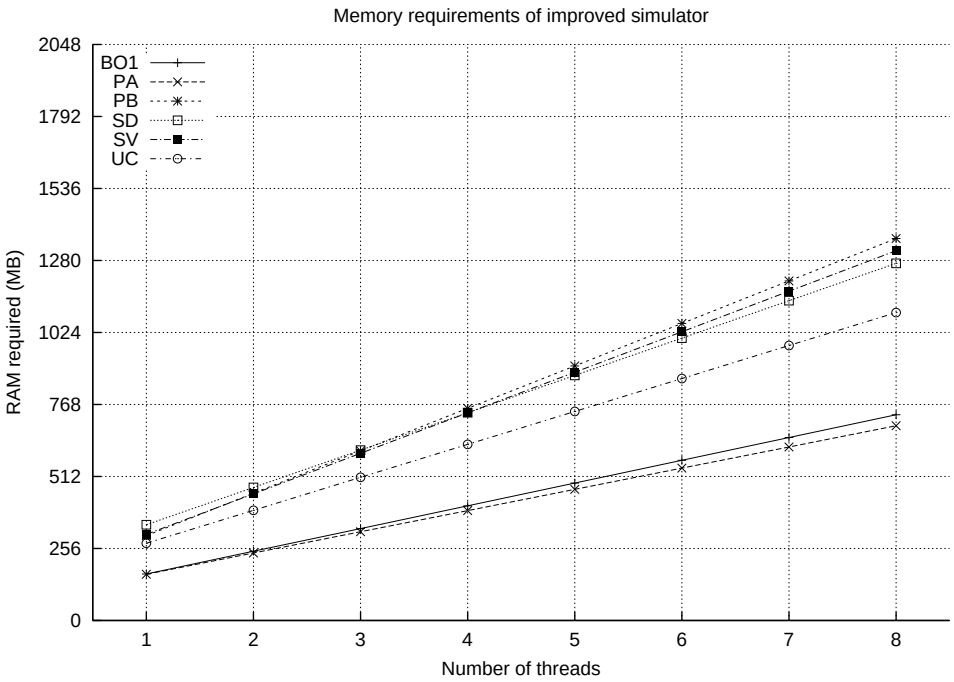

Figure 5.1: RAM requirements of current multi-threaded simulator as function of number of active worker threads (number of processors used) for the six NLANR traces of Table 3.3.

where N is the number of documents in the trace, M is the number of references, T is the number of worker threads, and S is the number of servers. Figure 5.1 shows memory requirements as a function of number of worker threads for the traces I have used. In my experiments with cost-biased removal policies I associate per-byte miss costs with servers rather than with documents; see Section 3.1.1 for details. A happy side effect is that the simulator's memory requirements are substantially reduced. A more general simulator that associated miss costs with *requests* rather than servers or documents might require $4N + 8M + 16TN$ bytes of memory.

The major shortcoming of my parallel simulator is that it does not model cache freshness policies and therefore does not distinguish between "slow hits" (successful revalidations) and "fast hits" (no contact with origin server required because cache entry is fresh); this feature would not have helped my preliminary investigations, because the NLANR traces do not include document metadata such as expiration dates. Furthermore the HTTP/1.1 specification defines (often only implicitly or vaguely) a large parameterized space of compliant cache freshness policies [64], and to the best of my knowledge the freshness policies used in actual production caches are not well documented in the re-

search literature or elsewhere. Anecdotal evidence suggests that the freshness policies of several important production proxy and browser caches stray from the HTTP/1.1 caching recommendations. It is therefore not clear which of the many reasonable freshness policies a general-purpose simulator ought to implement. Finally, the benefits of a freshness policy are limited: It merely allows us to distinguish between fast and slow hits and to model violations of semantic transparency.

CHAPTER 6

Workload Measurement

This chapter discusses a new technique for measuring Web client request streams and describes how it was used to collect a large and detailed client trace at WebTV Networks. It also presents a thorough workload analysis and simulation results describing the aggregate LRU hit rate of the entire client population as a function of browser cache size. These simulation results, made possible by the efficient single-pass algorithm of Section 4.5, represent an upper bound on LRU cache hierarchy performance that is inherent in the offered workload, independent of the system currently serving it. We shall see that the *actual* performance of the WebTV system falls short of the potential revealed by my simulations: Redundant data-payload transfers that cannot be explained as compulsory or capacity misses occur frequently in the WebTV system. I briefly describe how a simple HTTP protocol extension can close this gap; Chapter 7 motivates the protocol extension and Section 7.5 describes it in greater detail.

As noted in Section 5.2, the trace data used in most empirical Web caching research cannot support large-scale bottom-up simulations of browser/proxy cache hierarchies: Existing client traces are too small, and traces based on proxy logs and network sniffers lack crucial detail. This section describes a technique that combines the relative ease of proxy logging with most of the advantages of client instrumentation. In this method a "cache-busting proxy" intercepts requests from unmodified clients and labels all replies uncachable, thereby disabling browser caches and allowing the proxy to log requests that would otherwise be served silently from browser caches. An informal survey of Web researchers reveals that this technique has been proposed before; it was discussed by a group at Boston University in late 1999 [30] and is described in a recent book by Krishnamurthy

& Rexford [96]. Very recently, Adam Bradley of Boston University has implemented a cache-busting proxy [30, 31]. To the best of my knowledge, however, the idea was never used before my work at WebTV.

In September 2000 I collected a large anonymized trace of client accesses at WebTV Networks using a cache-busting proxy. The proxy itself ran in non-caching mode; the trace therefore reflects activity in a cacheless system. The proxy furthermore recorded a checksum of every entity-body (data payload) received from origin servers, as well as a checksum of the (possibly different) entity-body served to the client after transcoding by the proxy. All events in this trace are timestamped at microsecond resolution by well-synchronized proxy clocks. The proxy recorded all cache-related HTTP metadata in client requests, server reply headers, and "`META http-equiv`" tags in HTML files. WebTV's trace spans 16 days and records over 347 million requests to over 36 million documents by over 37,000 clients; this is two orders of magnitude larger than any client trace described in the Web caching literature.

6.1 Trace Collection

With over a million active subscribers WebTV Networks is among the largest Internet service providers (ISPs), and its customer base is arguably more representative of the general public than the traditional subjects of Web traces (computer science students and computer industry employees). Furthermore the WebTV system is extraordinarily well integrated, providing essentially everything but the origin server: client hardware, browser software, proxies, and Internet connectivity. WebTV staff constantly monitor and tune the system to improve its performance, frequently adding new instrumentation as new questions arise. WebTV is an important production environment controlled by a single organization; performance enhancements suggested by workload analysis are far easier to implement in such environments than in the overall Web. For these reasons WebTV is an ideal environment for Web-related research.

WebTV clients represent an interesting intermediate point in design space, midway between the resource-rich PC-based browsers of the early Web and the ultra-thin clients of tomorrow. WebTV employs a relatively inexpensive (often diskless) set-top box to enable Web surfing on a conventional television. The five types of client devices described in

Type	Description	cache size RAM	disk	# in trace	H.R. (%)
FCS	"Classic"	420 KB		8,790	37.64
BPS	No-frills	1240 KB		11,253	41.94
LC2.5	"Plus"	3200 KB		7,370	44.41
LC2	Diskful Plus	1 MB	20 MB	8,535	44.64
ST1	Satellite	3 MB	20 MB	1,221	44.81

Table 6.1: WebTV client devices.

Table 6.1 were in use during September 2000; the last column shows estimates of client cache hit rates based on request volumes reaching the proxy before and after browser caches were disabled. Clients connect to the WebTV service via modem; according to WebTV's measurements, bandwidth to clients varies but is typically roughly 33.6 Kbps.

My original goal was to collect a client trace using instrumented browsers. WebTV frequently downloads software updates to its client devices, so at first this seemed a straightforward approach. However a combination of logistical difficulties and schedule constraints forced an alternative approach. Compared with client software modifications, proxy patches are much easier to implement and deploy and are far more frequent in practice. We therefore decided to record unfiltered client requests by disabling the browser cache with a modified proxy.

A sophisticated centralized service infrastructure compensates for WebTV client limitations by transcoding images, re-writing HTML, and maintaining persistent state (e.g., cookies). Sixteen modified proxies collected WebTV's client trace, with the following non-standard features enabled during data collection:

- events were timestamped at microsecond resolution;
- checksums were logged of all entity bodies received from origin servers and all entity bodies served to clients after transcoding;
- all metadata relevant to caching in client requests, server reply headers, and embedded HTML tags were logged;
- all documents were served to clients with an "`Expires: 0`" HTTP header; and
- the proxy itself was run in non-caching mode.

All trace fields related to user identity and requested content are anonymized to protect user privacy. For instance, URLs and payload digests are irreversibly mapped to unique but otherwise meaningless integers.

During normal operations WebTV proxies process every byte of every data payload, so the additional performance penalty of computing checksums was relatively minor; similarly, the proxies normally parse META tags in HTML. The number of proxy hosts that collected data was twice as large as would normally be used for our sample client population, so the proxies were not overloaded as they recorded the trace. Extensive tests on large human-user test populations within WebTV prior to data collection revealed that the impact of our collection methods on user-perceived latency was not significant. We are confident that any additional latency due to the disabling of proxy and browser caches was minor and did not affect user behavior.

Payload checksums are crucial to the namespace investigations of Chapter 7, for they illuminate the relationship between a URL and the data payload returned as a result of a specific access to that URL at a given instant. The fact that the WebTV trace reflects activity in a cacheless system ensures that the payloads recorded in every transaction are those returned directly from the origin server; there is no chance that the WebTV proxy served (and logged) a stale payload from its cache. Another benefit of the WebTV trace is that it reflects no requests initiated by "robots," which can be identified only with complex and unreliable heuristics [6].

META tags are necessary for correct browser cache simulations and are furthermore interesting because they illustrate discrepancies between HTML-embedded metadata and HTTP headers. For instance, META expiration dates often disagree with HTTP expirations. Of over 149 million requests issued during 16–22 September, 95,558 have expiration dates in both headers and META tags. Among these requests the META expiration is earlier than the response header expiration 21.8% of the time; ignoring the HTML-embedded expiration could cause consistency violations in some of these cases. In 141,159 cases an expiration is specified only in a META tag; unnecessary revalidations would result from ignoring the embedded metadata in some of these cases.

WebTV proxy host clocks are carefully synchronized via Network Time Protocol [114]. A script queried the proxies with `ntpq` at ten-minute intervals throughout the data collection period to check their clock synchronization; Figure 6.1 shows the distribution of

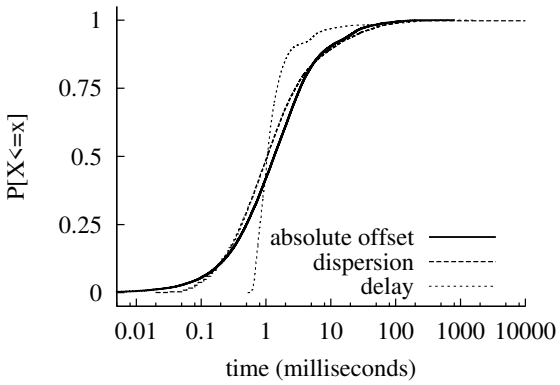

Figure 6.1: Distribution of NTP parameters during data collection.

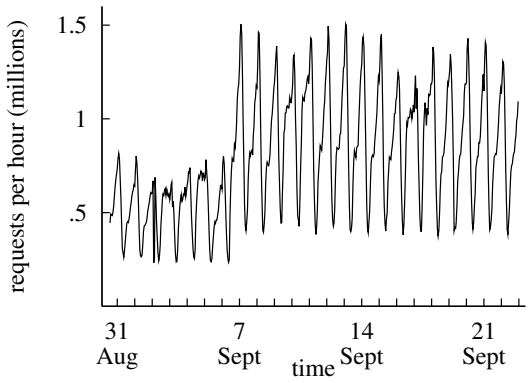

Figure 6.2: Hourly request volume by GMT time, 1-hour time bins.

observed NTP parameters. The absolute offset of the proxies with respect to an accurate reference is nearly always under 10 milliseconds. An individual client's references are nearly always separated by a longer interval. The WebTV trace therefore reflects the true order in which accesses are made, which is crucial for accurate trace-driven simulation.

A sample of client devices was "attached" to our modified proxy bank throughout the data collection period, i.e., all Web sessions initiated by these devices were handled by the special proxies. WebTV clients communicate directly with origin servers for secure transactions, which are therefore not reflected in our trace. Furthermore the trace contains only HTTP requests; we did not record the small volume of FTP and Gopher traffic handled by the proxies. We estimate that well under 5% of traffic escaped detection.

WebTV served documents pre-expired for over sixteen days beginning on Wednesday 6 September 2000. The request rate reaching our proxies promptly doubled, as illustrated in Figure 6.2. By comparing request volumes from each client device type before and

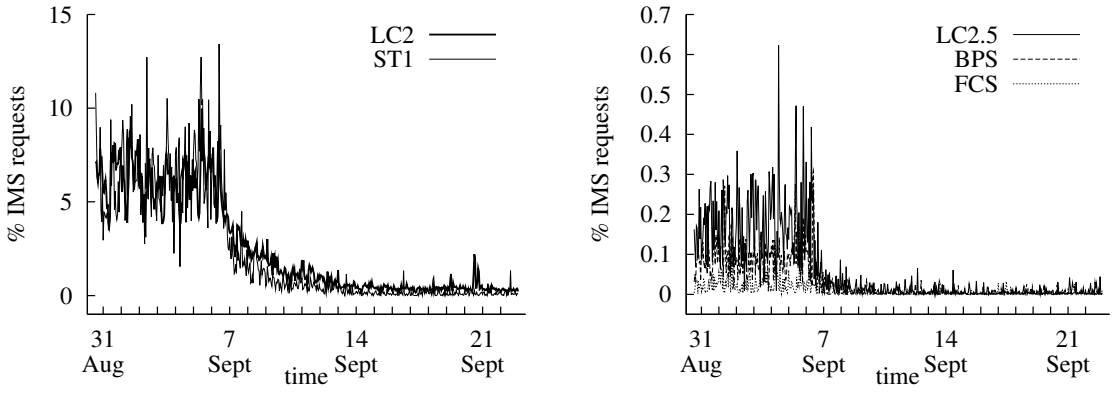

Figure 6.3: Percentage of IMS requests from diskful (left) and diskless (right) clients, 1-hour windows. Note that vertical scales differ.

after 6 September we obtain crude estimates of browser cache hit rates; estimates based on requests of 4–5 vs. 11–12 September are shown in Table 6.1. This technique must be used with care, because browser caches did not cease to operate on 6 September, they merely ceased to cache incoming documents. It is reasonable to suppose that many browser caches remained "warm" even after cache busting began, serving some fraction of requests from cache. We gain insight into browser cache "cool down" from the percentage of "`If-Modified-Since`" (IMS) revalidation requests reaching our proxies from each device type over time, as shown in Figure 6.3.

Rich-client browsers typically use separate regions of memory to hold the currently-viewed document and to cache previously-viewed objects; the contents of the cache therefore change only when new items are inserted into the cache. In the memory-constrained WebTV client, however, the same region of memory is used as both a "staging area" for the current item and as cache for previously-requested items. Large incoming documents therefore cause cache evictions *even if they are not cached themselves*. This accounts for the rapid decrease in IMS requests in Figure 6.3. Collectively, the browser caches in our sample are never flushed completely; even after two weeks IMS requests reach our proxies, possibly from clients with low activity levels. However the fraction of IMS requests quickly falls to negligible levels, particularly for diskless clients. Our estimates of browser cache hit rate are based on proxy request volumes several days after cache busting began, by which time most browser caches have cooled substantially. However it is possible that our estimated hit rates for diskful clients are slightly low. In retrospect, we might have obtained better results from a more thorough cache-busting proxy that attempted to forcibly

	full trace	reduced trace
Clients	37,201	37,165
Server IP addresses	267,595	252,835
Server hostnames	536,451	412,509
URLs	40,756,045	32,541,361
Unique payloads	38,754,890	36,573,310
(URL, payload) pairs	54,910,572	44,785,808
Transactions	347,460,865	326,060,677
Bytes transferred		
Total		1,973,999,619,772
Unique payloads		639,563,546,204

Table 6.2: WebTV trace summary statistics.

flush browser caches, e.g., by serving an HTML file with a large number of newline characters appended to it.

Table 6.2 summarizes the WebTV trace. The trace is roughly as large in most respects as recent proxy traces and substantially larger than mid-1990s Web client traces. Furthermore it reflects a client sample comparable to the entire end-user population served by NLANR's cache hierarchy, which is thought to be under 100,000 (see Section 5.2.5).

My simulations use only successful (HTTP status code 200) transactions. I furthermore exclude transactions involving seventeen payloads for which accurate sizes are not available; these account for slightly over 100,000 transactions. The reduced trace is summarized in the right-hand column of Table 6.2. I associate with each reply payload a single size that includes protocol overhead (HTTP headers) by adding to each payload's Content-Length a median header size of 247 bytes.

Table 6.3 summarizes several of the largest and most important Web workload traces used in recent literature: the two mid-1990s Web client traces discussed in Section 5.2.4, more recent proxy and server traces [12, 13, 115, 172], an AFS client trace [123], and finally the WebTV trace. The most striking feature of Table 6.3 is the large size difference between the early client traces and the more recent proxy and server traces; by nearly every measure the latter are orders of magnitude larger. The difficulty of deploying an instrumented browser on large numbers of clients is largely responsible for the difference. The WebTV trace, while nearly as detailed as the early Web client traces, records roughly as many transactions as the three largest proxy traces *combined*.

Trace	Type	Begin	End	Clients	Objects	Requests	Requests per Client per Day
CITI AFS	client	20 Oct 93	20 Dec 93	37	N/A	12,192,933	5,402
Georgia Tech.	client	3 Aug 94	24 Aug 94	107	9,452	43,060	19
Boston U.	client	21 Nov 94	17 Jan 95	600	46,830	575,775	17
Cable Modem	proxy	3 Jan 97	31 May 97	$\sim$ thousands	16,110,126	117,652,652	
World Cup	server	1 May 98	23 Jul 98	2,770,108	20,728	1,352,804,107	6
Compaq WRL	proxy	1 Jan 99	31 Mar 99	$\sim$ 25,000	N/A	125,259,641	54
U. Washington	proxy	7 May 99	14 May 99	22,984	$\sim$ 18,400,000	$\sim$ 82,800,000	515
Microsoft	proxy	7 May 99	14 May 99	60,233	$\sim$ 15,300,000	$\sim$ 107,700,000	286
WebTV	client	7 Sep 00	22 Sep 00	37,165	36,573,310	347,460,865	425

Table 6.3: Traces used in Web and file system research. "Objects" refers to distinct payloads, or to URLs in traces that do not distinguish payloads.

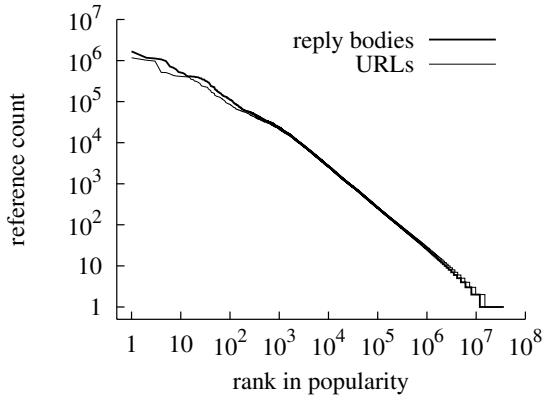

Figure 6.4: Zipf-like reference counts of URLs and reply bodies.

	URLs	Unique payloads
Count	32,541,361	36,573,310
Zipf α	1.0341 ± 0.000032	0.9376 ± 0.000043
Zipf β	7.6313 ± 0.000227	6.9112 ± 0.000308
R^2	0.969766	0.928005

Table 6.4: WebTV trace Zipf parameters.

The WebTV trace is large not merely in terms of number of clients and transactions but also in terms of its "working set." The sum of distinct payload sizes in the WebTV trace is roughly 600 GB. At the time the WebTV trace was collected, thirty-one production-grade cache products competed in a "Cache-Off" benchmark exercise [142]. The mean capacity of these caches was 83.5 GB and the median size was 42 GB. The WebTV workload could fill the largest entrant's cache (315 GB) nearly twice. However the trace's working set is not impossibly large by the standards of September 2000; the bank of modified WebTV proxies that collected the trace had a total capacity of roughly 600 GB. Similarly, the sum of distinct payload sizes requested by typical clients in the trace is moderately large for a set-top device, but not excessively so. The median client receives under 20 MB of distinct payloads, and over 26% of the client devices that generated the WebTV trace had larger browser caches [85]. In Section 7.3 I consider effectively-infinite browser and proxy caches, i.e., caches sufficiently large that they suffer no capacity misses.

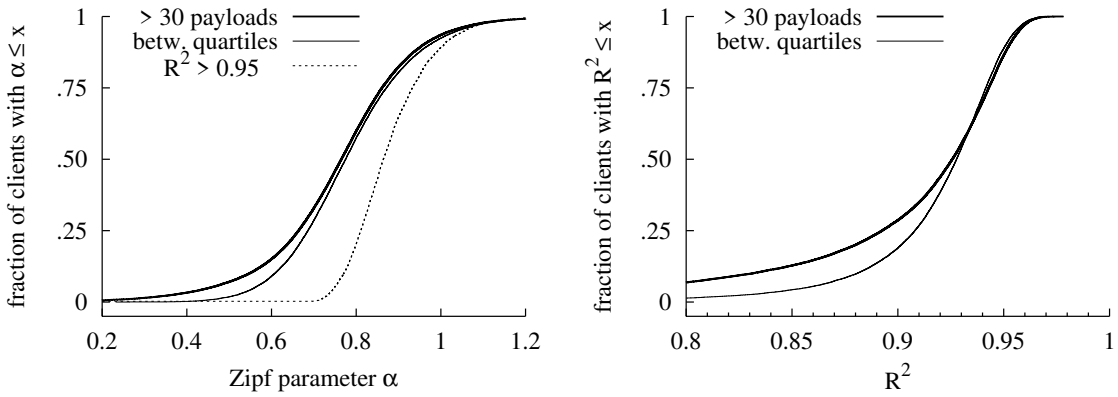

Figure 6.5: Left: CDF of Zipf α across client sub-populations. Right: CDF of R^2.

6.1.1 Trace Characteristics

The popularity distributions of URLs and reply bodies, shown in Figure 6.4, affect cache performance. As in most Web workloads studied to date the popularity distribution of URLs is Zipf-like, as is that of distinct data payloads. Table 6.4 reports Zipf parameters for the popularity distribution of both URLs and reply bodies, obtained by fitting to the WebTV data linear least-squares models of the form

$$\log_{10}(\text{reference count}) \;=\; -\alpha \log_{10}(\text{popularity rank}) + \beta$$

using standard algorithms [81, 135]. The Zipf α parameters in the WebTV client trace are remarkably close to unity. Cunha et al. report similar findings on the Boston University client trace [52]. By contrast, the Zipf parameter is often higher at servers [130] and lower at proxies [33]. Padmanabhan & Qiu give an interesting analysis of why the Zipf α varies at different levels in a cache hierarchy [130]. Breslau et al. discuss the implications of Zipf-like popularity distributions for caching [33]. The coefficient of determination R^2 describes how well a linear regression model explains the data; R^2 is unity when the model perfectly explains the data and is zero when the model explains none of the variation in the data. The high R^2 values in Table 6.4 confirm the visual impression of Figure 6.4 that a Zipf model is appropriate to the data.

We furthermore compute the Zipf α parameter of the payload popularity distribution for *each client* in the WebTV trace. Figure 6.5 shows the distributions of α across three subsets of the client population: 1) all clients that request more than thirty distinct payloads (this

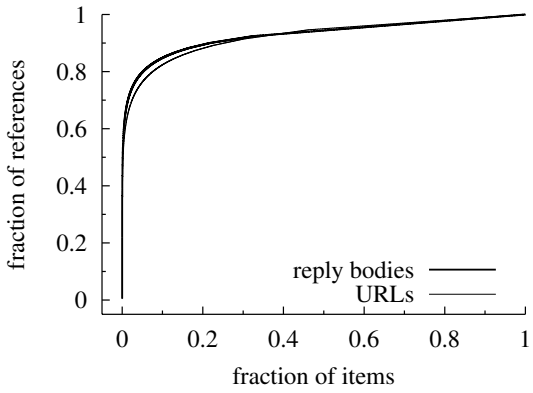

Figure 6.6: Concentration of references.

excludes only a handful of clients), 2) clients whose number of distinct referenced payloads is between the 25th and 75th percentiles, and 3) clients whose model fit is particularly good ($R^2 > 0.95$). The figure displays separately the distributions of R^2 for the first two groups. Three remarkable features are apparent in Figure 6.5: For most clients a Zipf model describes the popularity of accessed payloads reasonably well ($R^2 > 0.9$). Furthermore, whereas α for the overall trace (shared proxy serving cacheless clients) is roughly 0.938, it is *lower* in over 87% of individual client reference streams; at present I have no theoretical explanation for why α should be higher at a shared proxy than at clients. Finally, α is noticeably higher in clients for which the Zipf model fit is close.

Figure 6.6 shows the cumulative concentration of references across URLs and payloads sorted in descending order of popularity. The top one percent of payloads accounts for over two thirds of all transactions and the top ten percent account for nearly 85%; for URLs the figures are 62.2% and 82.4%, respectively. Concentration of references in the WebTV client trace is much stronger than in proxy traces (e.g., Figure 5a of Arlitt et al. [12]). Browser caches filter reference concentration as well as reference locality from the original client reference streams, so that both locality and concentration are markedly lower in the reference stream that reaches proxies. Figure 10 of Padmanabhan & Qiu [130] shows *server* vs. proxy reference concentration.

Client activity levels span a wide range. Figure 6.7 shows the distribution of the number of requests and total bytes transferred to clients in the WebTV trace. Visual inspection suggests that these distributions are roughly lognormal, i.e., the logarithm of requests-per-client and of bytes-per-client appears to be roughly Gaussian, as illustrated in Figure 6.8.

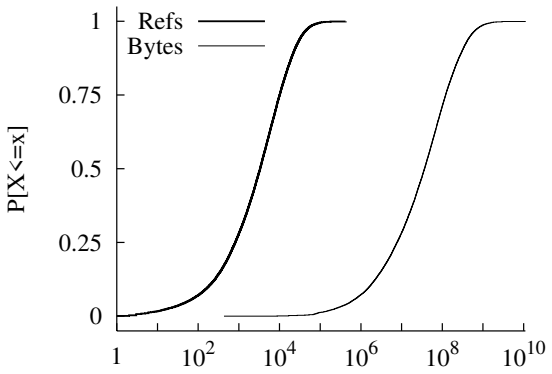

Figure 6.7: Distribution of references/client and bytes/client.

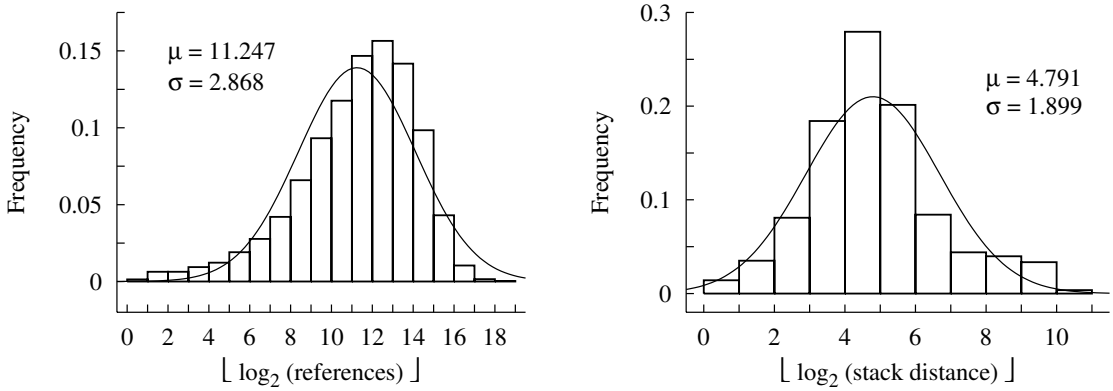

Figure 6.8: Frequency histograms of $\log_2$-transformed data. *Left:* references/client. *Right:* stack distances in BPS sample.

The figure shows histograms of log-transformed data superimposed over a normal curve defined by sample mean and sample variance.

Similarly, graphical evidence suggests that the distribution of LRU stack distances at the client is roughly lognormal (Figure 6.8, right; the data shown are from a sample of requests from a subset of BPS-type client devices, described below). As explained in Section 4.4, this distribution is closely related to the success function of an LRU cache [108] and is often used to measure temporal locality in reference streams [4, 15]. Almeida et al. report that references reaching *servers* appear to have lognormal stack distance distributions, and that lognormal stack distance models predict cache success functions well [4].

Trace set	Response content-length mean	median
WebTV (includes HTTP headers)	6,054	1,821
Compaq	11,192	2,894
Berkeley Home IP [73]	7,964	2,239
ATT-delta [119]	7,881	3,210
HP cable modem [12]	21,568	4,346
Washington [172]	7.7KB	not avail.

Table 6.5: Response sizes in various traces.

6.1.2 Trace Representativeness

Any study that generalizes from one or two traces must consider whether they are representative of Web use in general. I compared the WebTV trace with a Compaq data set (described in Section 7.3.2) and with traces used in prior literature in terms of Zipf parameters (Section 6.1.1), response sizes, MIME type distributions, and "surfing tempo."

Table 6.5 shows mean and median response body sizes from several relatively large, recent trace sets. The WebTV and Compaq traces roughly span the range of means and medians, except for those of the HP cable modem trace. The WebTV sizes are similar to (but slightly smaller than) the Berkeley sizes, consistent with the use of slow final hops in both environments. The Compaq sizes are not inconsistent with those from the AT&T and Washington broadband environments. The mean reply size in the cable modem trace is skewed because users took advantage of increased bandwidth to download extraordinarily large files; Martin Arlitt verified this by manually retrieving the files. Arlitt furthermore speculates that the large median reply size in the cable modem trace may be due to the same phenomenon [9].

Table 6.6 shows the fraction of transactions, bytes transferred, payloads, and working set associated with popular MIME types; all types that account for 1% or more in any category are shown. Roughly 4.3% of payloads are served (at different times) with more than one MIME type; in such cases I define the payload's type to be the most common type. In terms of transactions and bytes transferred, the WebTV trace is roughly similar to other workloads reported in the literature, e.g., the AT&T trace described in Table 1 of Douglis et al. [58]. JPEG files are more prominent in WebTV's client trace, probably because client caches served many JPEG accesses in the AT&T trace. In terms of distinct

MIME type	Transactions		Payloads	
	% by count	% by bytes	% by count	% by bytes
image/gif	68.389	34.391	17.247	5.727
image/jp[e]g	18.627	25.843	24.854	18.441
text/html	10.255	22.391	54.212	43.902
app'n/[x-]javascript	1.169	0.625	1.142	0.088
audio/(midi,x-midi,mid)	0.253	1.171	0.089	0.159
video/mpeg	0.077	9.808	0.291	20.843
app'n/octet-stream	0.034	0.716	0.038	1.390
video/quicktime	0.002	0.400	0.010	1.143
video/x-msvideo	0.001	0.547	0.009	1.440
all other	1.192	4.106	2.107	6.867

Table 6.6: MIME type distribution of WebTV trace.

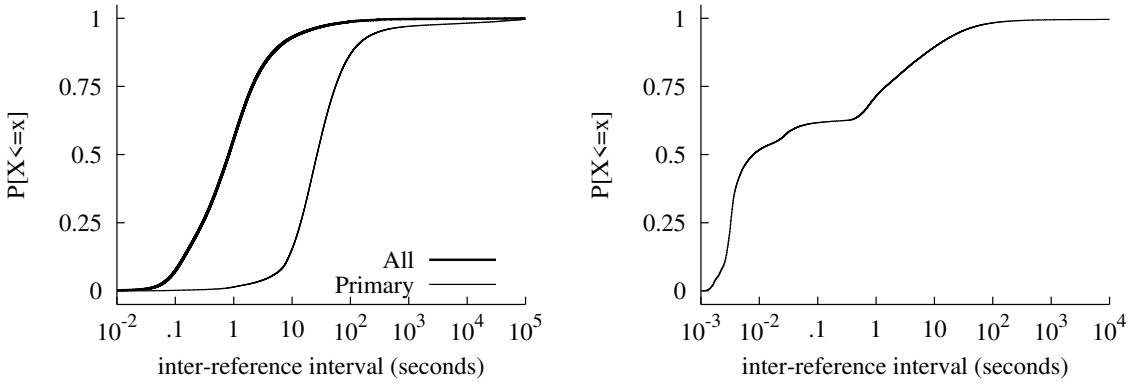

Figure 6.9: Distribution of inter-reference intervals in WebTV (left) and Boston University (right) client traces.

payloads, HTML is far more prevalent in the WebTV trace (54% vs. 24%). The practice of decomposing logical pages into multiple HTML frames, more common in September 2000 than in November 1996, might partly explain the difference.

Wolman et al. collected a large Web trace at the University of Washington using a packet sniffer in May 1999 [171]. Their Figure 1 reports the distribution of MIME types in this trace. Image files account for more transactions and more bytes transferred in the WebTV trace, probably due to client caching on the University of Washington campus.

We might expect bandwidth-constrained thin clients to "surf" at different rates than conventional rich-client browsers in academic or corporate environments. Figure 6.9 shows the distribution of inter-reference intervals for the last seven days of the WebTV trace and for

the Boston University client trace. WebTV requests directly initiated by user actions (e.g., the fetch that results from following a hyperlink) are marked as "primary" in the trace, and the distribution of intervals between primary references is plotted separately for WebTV. The Boston distribution is bi-modal due to browser cache hits (compound objects, e.g., HTML pages with embedded images, are *not* responsible; such objects are present in both traces). The WebTV data reflect a cacheless low-bandwidth environment, and therefore it is somewhat surprising that WebTV browsers appear to be operating roughly as fast as Xmosaic: 89.5% of BU intervals are 10 seconds or less; for WebTV the figure is 93%.

In summary, the WebTV trace is roughly consistent with other data used in Web-related research in terms of a variety of characteristics. The differences are largely attributable to the fact that the WebTV trace was recorded in an entirely cacheless environment.

6.2 Inherent Performance Bounds

This section considers bounds on the performance of *any* cache system serving the WebTV workload, bounds that are inherent in the workload itself. We shall consider several inherent performance bounds, describe how they evolve over time, explore the effect of multi-level cache hierarchies on these bounds, and investigate whether the WebTV system's performance approaches the bounds inherent in its workload.

6.2.1 System-Wide Miss Rates

The most obvious example of an inherent bound is the compulsory miss rate. In a given cache reference stream, the first time a reply payload appears in a transaction it *must* be fetched from afar; the request cannot be satisfied by the cache. The compulsory miss rate of the WebTV workload can be obtained directly from Table 6.2 as the ratio of distinct payloads to transactions: Approximately 11% of transactions require that a payload be retrieved into the WebTV system. We can compute a minimal byte miss rate in analogous fashion using total bytes transferred and sum of distinct payload sizes; it is roughly 32.4%. In other words, for the WebTV workload the difference between perfect caching and no caching is a factor of three in bandwidth consumption and an order of magnitude difference in the number of payload retrievals.

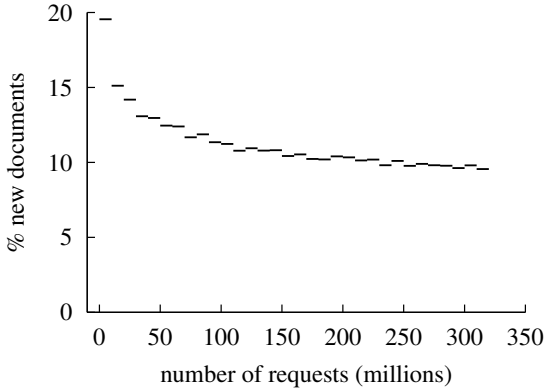

Figure 6.10: Percentage of replies containing new documents.

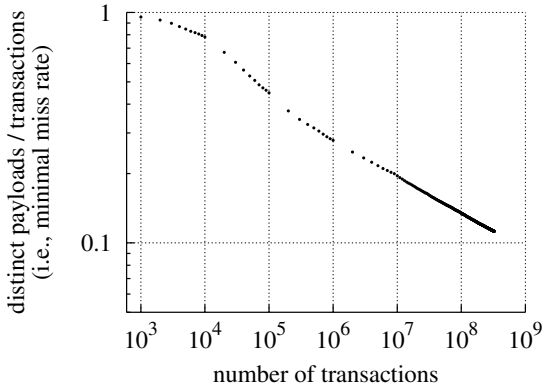

Figure 6.11: Ratio of distinct documents to transactions vs. number of transactions examined.

Figure 6.10 shows the percentage of replies containing never-before-seen payloads in non-overlapping windows of 10,000,000 requests for the first 320 million references in the WebTV trace. This is identical to the minimal (compulsory) miss rate of the overall WebTV system, assuming caches so large that capacity misses never occur. The figure shows that in the absence of redundant payload transfers the steady-state hit rate of an infinitely large WebTV proxy serving cacheless clients exceeds 90%; even a cold proxy cache would enjoy an 80% hit rate. In practice, imperfect cache consistency mechanisms and namespace complexities (e.g., aliasing) cause unnecessary cache misses and redundant payload transfers. Section 7.5 describes a simple and practical way to eliminate these problems entirely and raise hit rates to the full potential suggested by Figure 6.10.

Another way to view the evolution of compulsory miss rate over time is to compute the ratio of distinct documents to transactions using truncated traces of varying length. In

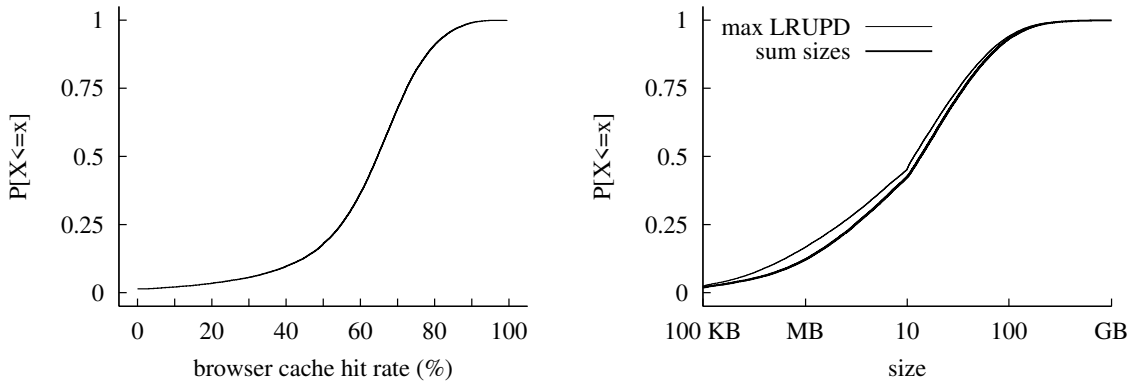

Figure 6.12: Distribution of maximal browser hit rates (left) and effectively infinite browser cache sizes (right).

other words, for different values of K, compute compulsory miss ratio using only the first K transactions in the overall trace, ignoring the remainder of the trace. Figure 6.11 shows the results of this exercise on a log-log scale. While Figure 6.10 gives the impression that compulsory miss rates level off at around ten percent after a week or so, Figure 6.11 reveals a more subtle pattern: The overall compulsory miss rate declines according to a *power law*, and this pattern persists even after hundreds of millions of transactions have been processed.

6.2.2 Browser Caches

Figure 6.12 shows the distribution of maximal *browser* hit rates under ideal conditions for individual client request sequences, and the distribution of browser cache sizes required to achieve maximal hit rates. "Ideal conditions" means that the first request that yields a given payload is a miss, but all subsequent requests that would return the same payload are hits. In other words, no redundant transfers occur, and only compulsory misses occur. This is similar to Mogul's "perfect coherency" cache [115–117], but it assumes no misses due to the namespace. In Mogul's terminology, I simulate a "perfect duplicate suppression" cache large enough to store all requested documents. We see from the left-hand subfigure that the median of maximal individual browser hit rates is roughly 65%.

A browser cache attains maximal hit rate if it can store all requested documents; the sum of distinct payload sizes is therefore termed the "infinite cache size" of a request sequence. However if we assume LRU replacement we can compute the maximal *priority depth* across

references in a workload [89]; this is the smallest LRU cache size that experiences no capacity misses. The distribution of infinite cache sizes and maximal LRU priority depths is shown on the right of Figure 6.12. For the workloads studied an 11.6 MB LRU cache is effectively infinite for half of clients.

We obtain a complete picture of the relationship between browser cache size and potential hit rate by computing each client's success function (hit rate as a function of cache size) separately, assuming LRU replacement. We now permit capacity misses, but as before no redundant transfers occur. Efficient single-pass simultaneous simulation algorithms for this computation have long been available for the special case where document sizes and miss penalties are uniform [23, 128, 155]; Daniel Reeves and I generalized them to non-uniform sizes and miss costs as described in Sections 4.4 and 4.5. Using the Reeves-Kelly algorithm I first compute browser cache hit rates for each client at every cache size. I then aggregate the results into a single success function for the entire client population.[1]

To avoid the confounding effects of cache cool-down (Figure 6.3) and cold-start, I also perform the same exercise for a sample of 1,959 modern diskless (BPS) clients with moderately heavy request volumes (between median and 75th percentiles) and moderate locality (maximal browser cache hit rates between the 25th and 75th percentiles). We use each client's first 2,000 references to warm the browser cache and tabulate hit rates based only on its next 1,000 requests. Results for both the BPS sample and the entire client population are shown in Figure 6.13; estimates of actual WebTV browser hit rates based on proxy request volumes before and after browser caches were disabled (Table 6.1) are included for comparison. These results are similar to the success function presented in Figure 5 of Bestavros et al., which assumes LFU replacement [26].

Aggregate browser cache success functions are essential to informed tradeoffs between browser functionality and cache hit rates in thin-client systems such as WebTV. New versions of browser software support new features and therefore require more resources, e.g., physical memory, but capacity expansion is not possible in the installed base of client de-

[1] As noted in Section 4.4, fast simultaneous simulation yields correct results only for cache sizes as least as large as the largest document in a trace when used to model rich-client browsers such as Netscape and IE, in which replies larger than the cache do not alter its contents. The memory-constrained WebTV browser, however, uses the same region of memory as both a cache and a staging area for the document currently being viewed. A reply larger than the cache will therefore flush the browser cache's contents, even though such an oversized reply cannot be cached. Stack methods can be used to model WebTV-like browser caches at *all* cache sizes.

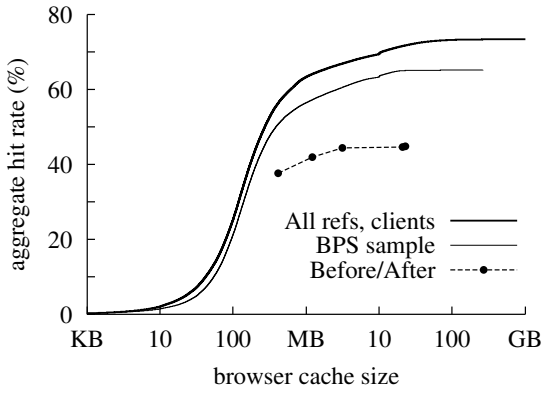

Figure 6.13: Aggregate WebTV client success functions.

	Cache	Measured	BPS sample		All clients	
Type	Size	H.R. (%)	H.R.	R.I.	H.R.	R.I.
FCS	420 KB	37.6	50.8	35.1	56.4	50.0
BPS	1240 KB	41.9	57.3	36.8	64.1	53.0
LC2.5	3200 KB	44.4	60.6	36.5	66.9	50.7
LC2	21 MB	44.6	65.1	46.0	71.7	60.8
ST1	23 MB	44.8	65.1	45.3	71.8	60.3

Table 6.7: Percent relative improvement (R.I.) over current hit rates.

vices. One option for browser designers is to steal application memory from the cache; this has been done several times at WebTV. However it is impossible to know the performance implications of such a decision without browser cache success functions obtained through measurement (Table 6.1) or simulation (Figure 6.13).

Table 6.7 summarizes the relative improvement in aggregate browser cache hit rates that would result from eliminating redundant payload transfers, i.e., it summarizes the gap between the actual and potential hit rates of WebTV caches shown in Figure 6.13. The large gap between the simulated and measured success functions in the WebTV data indicates that redundant payload transfers occur frequently; Chapter 7 discusses the causes of redundant transfers and presents a fully general solution.

6.2.3 Filter Effects in Cache Hierarchies

Cache hierarchies complicate our consideration of inherent performance bounds because the actions of lower-level caches affect the workload reaching higher-level caches.

The reference sequence reaching a shared Web proxy cache, for instance, consists of the merged miss sequences of the browser caches it serves. We must understand this coupling to tailor higher-level caches to the design of lower-level caches and their workloads.

Literature from the 1970s considers locality filtering in *linear* CPU memory hierarchies. The classic paper on stack distances by Mattson et al. extends stack processing techniques to the analysis of a rather peculiar abstract multi-level storage system [108]. Unfortunately the model considered appears to have been chosen for its analytic tractability rather than relevance to real systems, and it does not seem applicable to the semantics of actual processor, database, file system, or Web cache hierarchies. Gecsei, who describes ways of extending stack processing to more realistic types of multi-level linear storage hierarchies [68], assumes a combination of write-back semantics and cache-inclusion properties that make them inapplicable to branching Web cache hierarchies. Lam & Madnick consider storage hierarchies with slightly different semantics and discuss conditions that lead to undesirable inclusion violations [99]; Baer & Wang extend this work [16]. In both cases write-back semantics are integral to the analysis, which therefore cannot be extended to reasonable models of Web cache hierarchies.

In contrast to the mathematical rigor of the early literature, recent literature on locality filtering in database and Web caches relies exclusively on empirical methods (trace-driven simulation, often using *synthetic* inputs). Doyle et al. consider the implications of locality filtering for Web server cluster architecture using both synthetic and trace workloads [59]. Williamson describes a similar investigation with emphasis on proxy caches [167]. Zhou et al. evaluate a formidably complex second-level database buffer cache removal policy via trace-driven simulation using real traces and synthetic benchmarks [176]. These recent papers do not build upon the earlier formal literature on locality filtering, nor do I.

A crucial issue in locality filtering is whether enough locality remains in the miss sequence of lower-level caches to make higher-level caches worthwhile. The reduction in locality due to client caching might be offset by the merging of many dedicated-cache miss streams into a single shared-cache reference sequence, but this depends on sharing across client reference streams. Muntz & Honeyman report that hit rates of shared intermediate caches in network file systems do not exceed 20% if reference streams are filtered even by small client caches [121]. The WebTV data, however, point to a dramatically different conclusion: Assuming *infinite* browser caches, up to 73.4% of requests can be served from

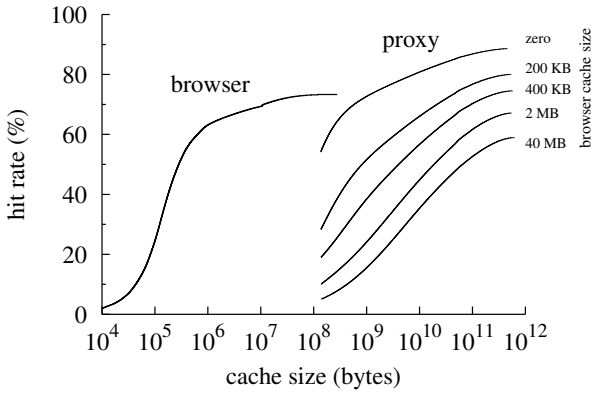

Figure 6.14: Browser success function and proxy success functions for several different browser cache sizes.

browser caches. Of the remaining requests, 57.7% can be served from a sufficiently large shared proxy cache. Sharing across client reference streams is far stronger in Web workloads than in distributed file systems, and shared caches are therefore potentially far more effective. Client caches do not "skim all of the cream" in Web workloads.

Furthermore when we consider the effect of *finite* browser cache sizes on proxy cache performance, a remarkable pattern emerges. Figure 6.14 shows on the right LRU proxy cache success functions for workloads filtered by each of five browser cache sizes; the browser cache success function of Figure 6.13 is included on the left for comparison. Not surprisingly, proxy hit rates at any cache size are lower for larger browser cache sizes. The remarkable feature of the proxy success functions, however, is that over a range of interesting sizes (1–100 GB) they are *steeper* for larger browser caches. In other words, the *marginal* benefits of proxy cache expansion *increase* with larger browser caches. Note that this result is not general across the entire domain of the proxy success functions; it is not true, for instance, between 10^8 and 10^9 bytes. However it does hold across a wide range of proxy cache sizes that were reasonable by the standards of September 2000; recall from Section 6.1 that the capacity of typical commercial cache products at the time ranged from 40 to 80 GB.

6.3 Discussion

The cache-busting proxy technique requires no client modifications and relatively minor proxy modifications, and is more feasible than conventional client instrumentation,

especially for collecting large and representative data sets. Yet, the method is not without limitations and disadvantages. It fails to record user-interface events and therefore cannot support the same range of investigations as an instrumented client, which remains the "gold standard" for trace collection. Furthermore the new method provides event timestamps recorded at the proxy, and these do not necessarily reflect the latency experienced by clients. If documents are served pre-expired as in the WebTV procedure, browsers may continue to serve some requests from cache even after cache-busting begins (Figure 6.3), so the technique does not provide a perfect record of client requests. Finally, it is unclear whether cache busting alters user behavior by increasing latency. Nonetheless a cache-busting proxy trace sheds far more light on client access patterns than ordinary proxy logs. It gives us (most of) the detail we want at a price we can afford.

My investigation of performance bounds inherent in the workload studied demonstrates that hierarchical caching yields substantial benefits. A well-functioning cache system can reduce payload transfers into the aggregate WebTV system by an order of magnitude and network traffic by a factor of three. Furthermore the role of a shared proxy is not insignificant: Merging 37,000 browser cache miss streams yields an access sequence with sufficient re-referencing to support high proxy hit rates. Finally, when we consider finite caches we find that larger browser caches yield lower absolute proxy hit rates but higher marginal returns to proxy capacity expansion across a wide range of reasonable proxy cache sizes.

From a practical standpoint the most interesting result of my workload analysis is the large gap between actual and potential browser cache hit rates for the client population as a whole (Figure 6.13). One way to think about the simulated success functions is that they describe performance in a "trivial namespace" Web, in which a simple one-to-one correspondence exists between URLs and data payloads. To an extent such a namespace is achievable in practice, e.g., by embedding payload checksums in URLs. Content delivery networks do this already: Akamai URLs contain partial MD5 checksums of data payloads. If URLs contain payload checksums, replies never expire and stale content is never served from cache; a simple namespace improves both the performance and the correctness of CDNs by making cache consistency issues disappear entirely. However, unchanging and mnemonic document names are necessary in systems like the Web, which precludes an entirely flat namespace. In practice, content naming on the Web is both complex and unstable: The same payload is sometimes available through different URLs, and the same URL

sometimes yields different payloads. Chapter 7 considers the extent to which content-naming practices contribute to redundant payload transfers in hierarchies of conventional Web caches. For the present we simply note that redundant transfers are very common in the WebTV system.

Several practical techniques have been proposed to avoid redundant payload transfers, but published "duplicate suppression" schemes for the Web do not completely eliminate the problem and require that modified servers supply hints to clients. Mogul reports that one such proposal would increase an infinite proxy's hit rate and byte hit rate by at most 5.4% and 6.2% respectively, assuming full participation by all origin servers [116, 117]. While the benefits of imperfect duplicate suppression for infinite proxy caches may be modest, WebTV's data show that the benefits of *perfect* duplicate suppression for *finite browser* caches are substantial. My most conservative simulation data suggest that browser cache hit rates would increase by 35% to 45% over their estimated current levels (Table 6.7). Fortunately a simple and practical hop-by-hop protocol extension can completely eliminate redundant proxy-to-browser transfers in systems such as WebTV; Section 7.5 describes this protocol extension, which Jeff Mogul and I conceived independently. Briefly, it works as follows:

- The browser caches every data payload it receives, without exception.
- The browser issues ordinary requests to the proxy.
- Before the proxy returns a reply payload it first sends the payload's checksum.
- The browser compares the checksum to those of items in its cache. A match ends the transaction; otherwise the proxy transmits the full payload.

Several variants of this overall approach are possible: The proxy could transmit a full reply preceded by a payload checksum and abort the transmission at the client's request. Alternately the proxy might wait for an explicit "proceed" from the client. The former entails no user-latency penalty and does not throttle the proxy or prolong transactions. However it may have little impact in high-bandwidth, high-round-trip-time environments if documents are small (the full payload reaches the client before its "abort" message reaches the proxy). The latter variant introduces an additional RTT into the transaction but completely eliminates redundant transfers; it may be attractive in low-bandwidth, low-RTT environments.

For purposes of duplicate suppression the proposed approach entirely ignores the namespace and consistency mechanisms (URLs and expiration metadata). Unnecessary misses due to aliasing and inappropriate metadata therefore disappear completely; only compulsory and capacity misses occur. The proposed approach could be used as a hop-by-hop mechanism between any two levels in a cache hierarchy. The semantics of existing protocols, e.g., HTTP, are largely unchanged if cached payloads and their digests are used to avoid redundant transfers. The scheme is similar in spirit to the Santos/Spring/Wetherall method of eliminating redundant network traffic between routers [143, 150].

Re-writing the rules of browser/proxy interaction is difficult in the most general case because different vendors' products must interoperate during migration and backward compatibility with legacy software must be maintained. However in more tightly integrated environments where a single organization controls both browser and upstream service infrastructure, e.g., WebTV and AOL, such changes are feasible. These also happen to be bandwidth-constrained environments, where the prospect of transferring compact message digests rather than far larger entity-bodies is particularly attractive.

CHAPTER 7

Content Naming and Redundant Transfers

We have seen in Section 6.2 that redundant payload transfers to WebTV browsers are surprisingly common. This chapter considers the extent to which this problem arises from the interaction among 1) standard cache management strategies, 2) content-naming practices, i.e., the relationship between request URLs and reply payloads defined by origin servers, and 3) client access patterns. In particular, we shall explore the prevalence of aliasing and resource modification and quantify the extent to which namespace phenomena such as these account for redundant transfers to conventional Web caches in a browser-proxy hierarchy.

Aliasing occurs in Web transactions when requests containing different URLs yield replies containing identical data payloads. Existing browsers and proxies perform cache lookups using URLs, and aliasing can cause redundant payload transfers when the reply payload that satisfies the current request has been accessed and cached with a URL other than the current one. Awareness of this problem is slowly growing in the commercial world: A major cache appliance vendor now encourages site designers to make Web pages cache-friendly by avoiding aliasing [49, page 9]. Within well-administered sites, aliasing might decrease when such advice is heeded. Other trends, however, are moving the Web in the opposite direction and are beyond the control of individual sites. For example, commercial Web authoring tools typically include many small "clip art" images, and a particular image bundled with such a tool is available through a different URL at each site that uses it.

Given the proliferation of technologies that create aliases and the potential for aliasing to cause redundant transfers, it is surprising that relatively little is known about the scale and consequences of this phenomenon. Only a few previous studies have considered the

prevalence of aliasing in Web transactions, the performance penalty of conventional URL-indexed cache management in large multi-level cache hierarchies, or ways to eliminate redundant payload transfers. Few of the Web workload traces that researchers have collected can illuminate the relationship between request URLs and reply payloads, because they do not describe payloads in sufficient detail.

This section quantifies aliasing and the impact of URL-indexed cache management on browser and proxy cache miss rates by examining large anonymized client and proxy traces collected, respectively, at WebTV Networks in September 2000 and at Compaq Corporation in early 1999. We investigate whether conventional URL-indexed caching is primarily responsible for the high rate of redundant proxy-to-browser payload transfers previously reported in the WebTV system, quantify the prevalence of redundant server-to-proxy payload transfers, consider other causes of redundant payload transfers, and discuss a simple way to eliminate all redundant transfers, regardless of cause.

Unlike the NLANR data used for the investigations of removal policies in Chapter 3, the traces used in this section record anonymized versions of both request URLs and reply data payloads; together, these illuminate namespace phenomena such as resource modification and aliasing. The richness of our data sets requires that we introduce new terminology not needed when discussing the simpler NLANR traces. In the context of a given trace of Web transactions, a reply payload is *aliased* if it is accessed via more than one distinct URL in the entire trace. Similarly a URL is *modified* if it yields more than one distinct reply payload. Certain terms related to aliasing, such as "duplication," lack clear, widely-accepted definitions in the literature, and I shall avoid them when discussing my own work.

When discussing cache performance, we are concerned with whether the payload required to serve a request is obtained from cache or must be fetched from elsewhere. Therefore throughout this section the term "hit" refers to accesses for which the required payload is obtained from cache, regardless of whether messages are exchanged with the origin server. A successful revalidation is a hit, because it averts the payload transfer. Similarly "miss" denotes a transaction in which the reply payload is not obtained from cache. The trace-driven simulations of this section do not consider caches that violate semantic transparency in the sense of RFC 2616 [64], i.e., they model caches that either miss or return exactly the same payload that the origin server would return at the moment of access.

7.1 Related Work

While few studies have directly addressed our central topic, many have investigated aspects of the HTTP namespace and their impact on cache performance. This section reviews literature on the relationship between URLs and reply payloads.

7.1.1 Resource Modification

"Resource modification" is the complement of aliasing: Requests containing identical URLs yield different reply bodies. Because it has direct implications for cache consistency and object "cachability," resource modification has been extensively studied. Section 7.2 compares the prevalence of aliasing and resource modification and reports that more transactions are affected by the former.

Douglis et al. report that rates of resource modification in a trace from a corporate environment are high enough to substantially reduce the hit rates of conventional URL-indexed caches [58]. More recently, Brewington & Cybenko consider the burden that modification rates place on search engines [35]. After fitting a combination of exponential and Weibull models to their data, they report that roughly 10% of inter-modification intervals are 10 days or less and roughly 72% are 100 days or less. Brewington's doctoral thesis considers the problem of monitoring changing information resources in greater theoretical and empirical depth [34]. This research is based on polling URLs obtained from users who have requested notification when specified resources change, and might therefore reflect a sample of resources with atypical rates of change. Padmanabhan & Qiu analyze the dynamics of content creation, modification and deletion at the MSNBC Web site [130]. They report a median inter-modification interval of approximately 10,000 seconds and note that most alterations to files are relatively minor.

Resources expected to change frequently are often called "dynamic," although this poorly-defined term blurs the distinction between the process by which a response is generated, and whether it is "cachable." In practice, cache implementors and researchers employ heuristics to identify uncachable responses by looking either for signs of dynamic generation (such as "cgi" in a URL) or for metadata, such as cookies, implying that a resource gives a different response to every request. Wolman et al. report that Squid deems uncachable 40% of replies in a large trace collected at the University of Washington in

May 1999; Zhang reports that customized and dynamic content together render roughly 7.1% of the objects in his trace uncachable [175].

Work by Wills & Mikhailov, however, casts doubt on the assumption that it is pointless to cache seemingly "dynamic" or "customized" content. They report that even if a previous access to a URL had returned a "Set-Cookie" header, in most cases the absence of a request cookie, or the presence of a different cookie, does not affect the reply payload returned for a subsequent access [168]. Repeated accesses to query resources at E-commerce sites sometimes return identical payloads [170]. Iyengar & Challenger exploited the cachability of dynamic replies at a large, busy Web server and report impressive performance gains [80]. Smith et al. report dynamic reply cache hit rates of 13.6% and 38.6% for two workloads [147].

Wolman et al. incorporate observed resource popularity and modification rates into an analytic model of hierarchical caching [172]. Their model describes the impact of resource modification rates on cache hit rates and suggests that cooperative caching schemes yield diminishing returns as client populations increase.

7.1.2 Mirroring

"Mirroring" typically refers to a special case of aliasing in which replicas of pages or entire sites are deliberately made available through different URLs. Shivakumar & Garcia-Molina investigate mirroring in a large crawler data set [144]. They report far more aliasing than appears in the WebTV client trace: 36% of reply bodies are accessible through more than one URL. Bharat et al. survey techniques for identifying mirrors on the Internet [28]. Bharat & Broder investigate mirroring in a large crawler data set and report that roughly 10% of popular hosts are mirrored to some extent [27].

Broder et al. consider approximate mirroring or "syntactic similarity" [36]. Although they introduce sophisticated measures of document similarity, they report that most "clusters" of similar documents in a large crawler data set contain only *identical* documents. In other words, simple aliasing is the dominant form of similarity in their workload. These authors report that 18–41% of reachable payloads are aliased.

7.1.3 Duplicate Suppression

Douglis et al. report that 18% of the full-body responses recorded at a corporate firewall that resulted in a new instance of a particular resource were identical to at least one other instance of a different resource [58].

Several "duplicate suppression" proposals address performance problems caused by duplication. The HTTP Distribution and Replication Protocol (DRP) employs payload digests to avoid unnecessary data transmission in deliberate replication over HTTP [156]. A DRP client obtains "index files" containing digests indicating the current state of resources, and the client can then request precisely those resources for which its copies are obsolete.

Mogul reviewed several end-to-end duplicate-suppression schemes involving "hints" supplied by origin servers to clients, and by clients to caches. These proposals do not entirely eliminate the problem of redundant payload transfers, and a trace-driven simulation demonstrates that one such scheme yields 5.4% and 6.2% improvements in hit rates and byte hit rates, respectively. Even these modest gains are *upper bounds*, because they assume the full participation of all origin servers [116, 117].

Santos & Wetherall [143] and Spring & Wetherall [150] describe a general protocol-independent network-layer technique for eliminating redundant traffic by caching *packet* payloads and transmitting digests thereof to avoid redundant transfers. Muthitacharoen et al. designed a network file system for low-bandwidth environments that performs similar operations on chunks of files [124].

Inktomi's Traffic Server proxy cache product has included a technique called "content fingerprinting," which uses payload digests to avoid storing multiple copies of identical payloads [107]. Content fingerprinting suppresses duplicates in storage, but not on the network. Bahn et al. describe a similar scheme [17].

7.1.4 Harmful Practices

The HTTP/1.1 specification is long and complex [64]. Not all servers are fully compliant, and the compliance of products does not always improve over time [94]. Non-compliance can clearly cause redundant payload transfers and other kinds of waste. However, redundant transfers can also occur if mechanisms introduced into HTTP/1.1 to improve cache correctness are used in strange but *compliant* ways. For instance, identical

payloads served by a single site are sometimes accompanied by *different* entity tags [169], causing new-style "`If-None-Match`" revalidation attempts to fail where old-fashioned "`If-Modified-Since`" requests might succeed. In this case, the server is compliant with the specification, but not with the most efficient possible implementation.

Furthermore, several common practices that do not violate the protocol complicate the HTTP namespace in harmful ways. Mikhailov & Wills report, for instance, that content providers sometimes embed session identifiers in dynamically-written URLs rather than cookies [113]. Ad rotation often creates many minor variants of the HTML for a Web page, inflating resource modification rates. Padmanabhan & Qiu document other types of minor changes that occur frequently at the MSNBC site [130].

7.1.5 Summary

Existing literature touches on a number of issues surrounding aliasing on the Web, but the prevalence of this phenomenon across user-initiated transactions and the impact of URL-indexed cache organization on miss rates in multi-level cache hierarchies is poorly understood. Most proxy traces employed in empirical Web caching research shed no light on aliasing because they do not record data payloads or digests thereof. Data sets collected by Web crawlers often include payload digests but cannot support trace-driven simulations of cache hierarchies; they illuminate aliasing across *available* resources rather than *accessed* resources. The WebTV trace described in Section 6.1.1 is well suited to my investigation because it records all client requests and corresponding server replies in a large, cacheless production environment.

7.2 Prevalence of Aliasing

For the purposes of this section, a transaction record is simply a pair (U, P) where U is a request URL and P is a reply data payload. We say that a reply payload P is *aliased* if there exist two or more records $(U, P), (U', P)$ containing the same reply payload P but different URLs U and U'. Similarly, we say that a URL U is *modified* if there exist two or more transactions containing U as the URL and different reply payloads P and P'. The *degree* of a payload is the number of distinct URLs that appear with it in transaction records, and the

URLs	32,541,361
Modified URLs	1,859,929
Unique payloads	36,573,310
Aliased payloads	1,821,182
(URL, payload) pairs	44,785,808
Transactions	326,060,677
w/ modified URLs	32,277,753
w/ aliased payloads	176,595,754
Payload sizes	
Range (min–max)	40–91,397,479
Median	5,487
Mean	17,487
Sum	639,563,546,204
Sum of aliased	19,726,808,472
Transfer sizes	
Median	1,821
Mean	6,054
Sum	1,973,999,619,772
Sum of aliased	711,717,843,218

Table 7.1: WebTV reduced trace aliasing statistics.

degree of a URL is the number of distinct reply payloads that appear with it in the trace. Aliased payloads and modified URLs each have degree two or greater.

Table 7.1 summarizes the prevalence of aliasing and resource modification in the reduced WebTV trace. The table shows that aliased payloads account for over 54% of transactions and 36% of bytes transferred in the WebTV trace, suggesting that conventional URL-indexed caches might suffer many redundant transfers and receive much redundant network traffic when processing the WebTV workload. Section 7.3 addresses these issues. Note that whereas over half of transactions involve aliased payloads, only 10% involve modified URLs; *aliasing affects far more transactions than resource modification*.

The figures cited above regarding the prevalence of aliasing are of limited scientific interest if they are merely artifacts of trace length. The assertion that "X% of payloads are aliased" is misleading if X varies with trace length. As in the discussion of compulsory miss rates surrounding Figure 6.11 in Section 6.2.1, we gain insight into this issue by computing quantities of interest using truncated prefixes of the overall WebTV trace and plotting these quantities against prefix length. Figure 7.1 shows fraction of payloads aliased and fraction of transactions involving aliased payloads as functions of trace length. We see

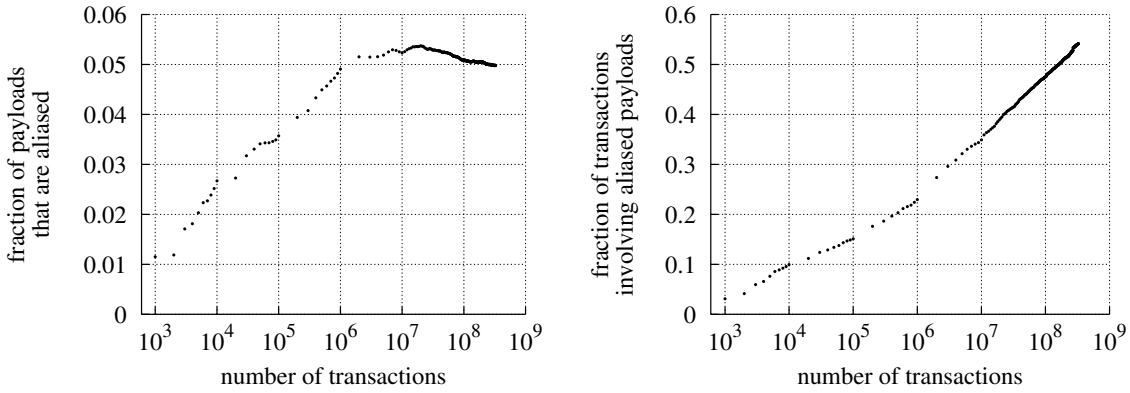

Figure 7.1: Left: Fraction of payloads aliased versus trace length. Right: Fraction of transactions carrying aliased payloads versus trace length.

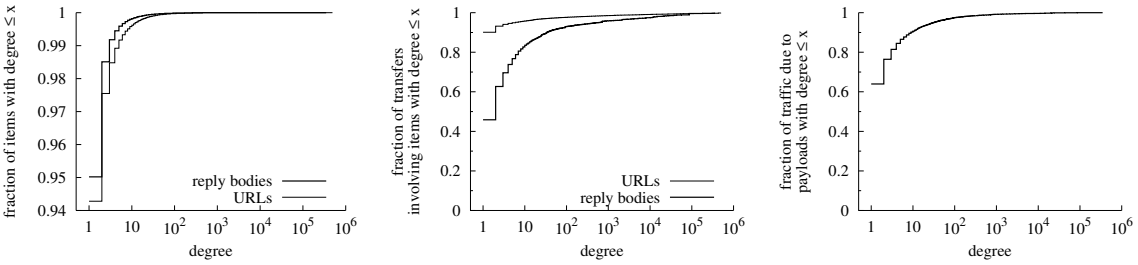

Figure 7.2: Left: CDF of payload and URL degrees. Center: CDF of transactions by degree of URL & payload involved. Right: CDF of bytes transferred by degree of payload involved.

from the right-hand plot that the latter quantity is indeed an artifact of trace length; it grows with the logarithm of trace length. Somewhat surprisingly, however, the left-hand plot shows that the fraction of payloads that are aliased is *not* an artifact of trace length. After a few days (roughly 100 million transactions) this quantity levels off at roughly 5% and remains constant. Whereas the crawler studies cited in Section 7.1.2 report that 20–40% of *available* payloads are aliased in the static sense that they are *reachable* via multiple URLs, the WebTV trace suggests that in the long term only around 5% of payloads are are actually *accessed* via different URLs by a large client population. Taken together, these facts suggest that *user-initiated transactions discover far less aliasing than is actually present in the hyperlink structure of the Web.*

The distributions of the degrees of payloads and URLs in the WebTV trace are shown on the left in Figure 7.2. Fewer than 5% of payloads are aliased, but one is accessed via 348,491 different URLs. Similarly only 5.7% of URLs are modified, but one yields 491,322

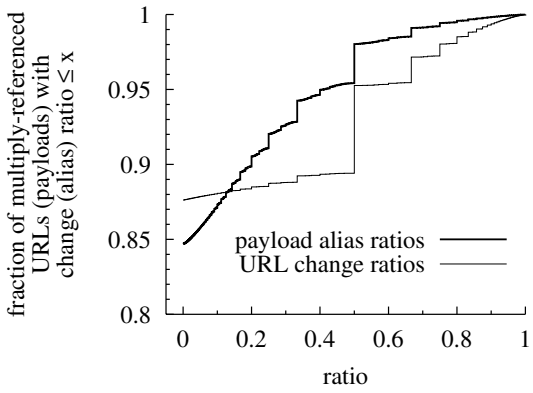

Figure 7.3: CDFs of change and alias ratios.

distinct payloads. This analysis downplays the prevalence of aliasing and modification because it does not consider the number of times that different (URL, payload) pairs occur in the trace. The plot in the center shows the distributions of payload and URL degrees weighted by reference count. Finally, the plot on the right shows the distribution of bytes transferred by the degree of the payload involved. The figure shows that roughly 10% of traffic is due to payloads accessed via 10 or more distinct URLs.

In over 41 million successful transactions (12.72%) a payload is accessed through a different URL than in the previous access to the same payload. By contrast, under 14.3 million transactions (4.37%) involve a different payload than the previous transaction with the same URL. Here again the prevalence of aliasing exceeds that of resource modification. (Note that this does not imply that aliasing causes more cache misses than resource modification; in fact, the reverse might be true.)

Following Douglis et al. [58] I compute for each multiply-referenced URL its "change ratio," the fraction of its accesses that return a different data payload than its previous access. We furthermore compute for each multiply-referenced *payload* an analogous metric, the "alias ratio," defined as the fraction of its accesses made through a different URL than its previous access. The distributions of change ratios and alias ratios across multiply-referenced URLs and payloads, respectively, are shown in Figure 7.3. The figure shows that 15.3% of multiply-referenced payloads are aliased and 12.4% of multiply-referenced URLs are modified. However the figure also shows that alias ratios are generally lower than change ratios. For example, only 2% of multiply-referenced payloads have alias ratios above 0.5 whereas 4.7% of multiply-referenced URLs have change ratios over 0.5.

MIME type	Transactions w/ Aliased Payloads		Aliased Payloads	
	% by count	% by bytes	% by count	% by bytes
image/gif	66.113	62.608	13.016	11.901
image/jp[e]g	30.655	30.748	6.976	6.472
text/html	15.993	12.729	1.577	1.172
app'n/[x-]javascript	66.564	67.410	1.863	4.370
audio/(midi,x-midi,mid)	82.854	81.833	35.157	32.052
video/mpeg	32.472	13.284	6.422	1.932
app'n/octet-stream	63.557	19.263	10.563	3.886
video/quicktime	7.583	1.205	1.515	0.488
video/x-msvideo	8.882	5.574	2.125	1.472
all other	47.089	21.200	3.324	2.275

Table 7.2: Prevalence of aliasing by MIME type in WebTV trace.

7.2.1 Aliasing and Response Attributes

Techniques meant to eliminate redundant transfers usually impose some costs. If we could impose those costs only on those subsets of responses that are most likely to benefit from an alias elimination technique, we could (in principle) reduce overall costs without similarly reducing overall benefits.

Table 7.2 shows the prevalence of aliasing among popular MIME types in the WebTV trace. The table uses the same sort order as Table 6.6. Aliasing is most common among MIDI payloads: 35% of MIDI payloads are accessed via two or more different URLs, and over 80% of MIDI transactions involve aliased payloads. However Table 6.6 shows that MIDI accounts for under 2% of all traffic and under 1% of all transactions.

GIF files account for over two thirds of transactions and over one third of bytes transferred in the WebTV trace (Table 6.6), and roughly two thirds of GIF transactions involve aliased payloads (Table 7.2). Taken together, these facts imply that *nearly half of all transactions involve aliased GIF payloads* ($0.66113 \times 0.68389 = 0.45214$). By contrast, aliasing is far less prevalent among HTML and JPEG payloads, which together account for roughly 29% of transactions and 48% of bytes transferred; fewer than 7.5% of transactions involve aliased HTML or JPEG payloads. These findings are consistent with the hypothesis that Web authoring tools account for much of the aliasing in Web transactions; unfortunately the traces I use are anonymized in such a way as to prevent more detailed investigation of

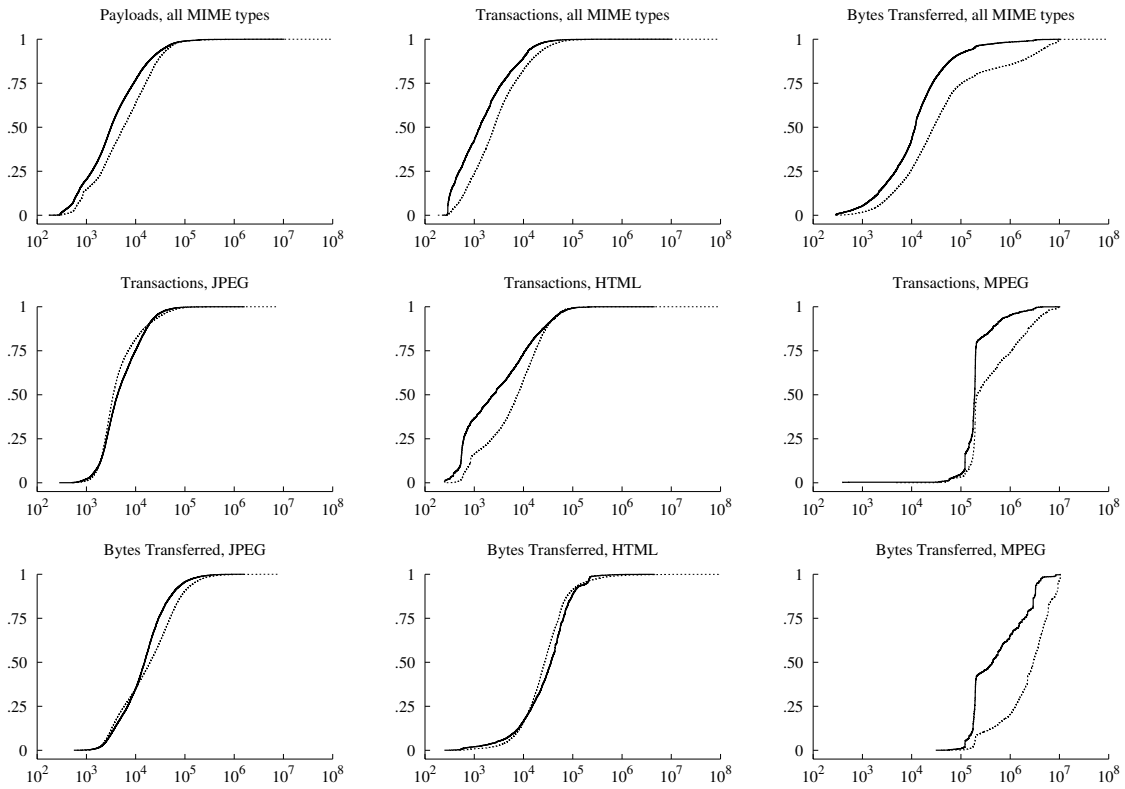

Figure 7.4: CDFs by payload size for all payloads (top row) and three popular MIME types. Solid lines indicate aliased payloads, transactions involving aliased payloads, and aliased bytes transferred; dashed lines non-aliased. All horizontal scales are identical and show payload size in bytes.

the issue. Section 7.5.4 discusses means of eliminating aliasing caused by Web authoring tools.

Figure 7.4 shows several distributions involving the sizes of payloads in the WebTV trace. The top row of distributions shows that aliased payloads, and the transactions and bytes transferred due to them, tend to be smaller than their non-aliased counterparts. However when we examine particular MIME types this generalization does not always hold. For example, aliasing is associated with slightly larger payload sizes in JPEG transactions and HTML traffic. Techniques that attempt to eliminate redundant payload transfers should add a minimal number of header bytes, since the bias toward aliasing of small payloads implies that potential benefits can easily be squandered.

1	Mirrored content	`http://mir1.bar.com/img.gif` `http://mir2.bar.com/img.gif`
2	Within-site	`http://bar.com/image.gif` `http://bar.com/i.gif`
3	Different sites, same `abs_path`	`http://bar.com/img.gif` `http://foo.com/img.gif`
4	Everything different	`http://bar.com/image.gif` `http://foo.com/i.gif`

Table 7.3: Causes of aliasing.

7.2.2 Causes of Aliasing

Aliasing can arise in several different ways, e.g., deliberate mirroring, aliasing within a single site, and identical content available at different sites. We can further decompose the last cause into cases where the `abs_path` component of the URL is the same, or different. Table 7.3 provides examples of the possibilities.

Knowing the cause of aliasing can help us decide where to focus efforts at remediation. A site can replace an ad-hoc mirroring strategy with a CDN, which does not introduce aliasing into the HTTP namespace [55]. Site administrators can avoid type 2 aliasing, following the advice of a CacheFlow white paper on cache-friendly site design [49]. Unfortunately the widespread use of Web authoring tools can cause type 3 aliasing, and this is beyond the control of individual sites. Furthermore aliasing occurs even *within* such tools: DreamWeaver [102], for instance, contains 632 unique image files under *642* different filenames.

The raw WebTV trace is anonymized in such a way that aliasing of types 1 and 3 cannot be distinguished. Furthermore the reduced trace used in my empirical work omits anonymized `abs_path` fields, preventing us from distinguishing between types 3 and 4. We can, however, identify cases where different URLs contain identical versus different host components.

Payloads of degree 2, i.e., payloads accessed via exactly two different URLs, fall into exactly one of the categories in Table 7.3. Degree-2 payloads account for 70% of aliased payloads, 31% of transactions involving aliased payloads, and 34.6% of aliased payload bytes transferred. 80.56% of degree-2 payloads are accessed via URLs with different host components; the remainder are cases of within-site aliasing. Though far from probative,

this finding is consistent with the conjecture that Web authoring tools account for much of the aliasing in Web transactions.

7.3 Performance Implications of URL-Indexed Caching

In any reference sequence, the first access to a given payload must result in a compulsory miss; subsequent transactions involving the same payload, however, can in principle be served from cache. In this section I compare the miss rates of conventional URL-indexed caches with compulsory miss rates. The use of URLs to organize and locate payloads in conventional caches accounts entirely for the difference between the two. As explained below, compulsory miss rates represent an *achievable* bound on the performance of a sufficiently large cache.

As noted above, I use the terms "hit" and "miss" to mean respectively "required payload is obtained from cache" and "payload must be fetched from elsewhere." This section assumes infinite-capacity caches that require no removal policy and suffer no capacity misses. It furthermore considers only semantically transparent caches, i.e., caches where hits supply exactly the same payload as the origin server would; this can easily be achieved in practice by revalidating every request.

7.3.1 Abstract Cache Models

A conventional "URL-indexed" cache stores in association with each requested URL the most recently received reply payload. In addition to compulsory misses, an infinite-capacity URL-indexed cache suffers a miss, and resulting redundant transfer, if it has previously received the payload that satisfies the current request, but this payload is not cached in association with the current request URL.

Just as it is possible to design a cache that wastes no storage on multiple copies of payloads [17, 107], it is possible to design a cache that suffers *only* from compulsory misses (assuming infinite capacity). One can construct such a cache by assuming that the cache computes a message digest of every stored payload, that the cache maintains an index mapping digest values to stored payloads, and that every payload-bearing response message received by the cache is preceded by a digest of that message's payload, computed by the

message sender (i.e., proxy or origin server). This allows the cache to avoid the payload transfer if it already stores a copy of the payload. The strategy is very simple: "1) cache forever every payload you receive, and 2) before receiving a payload, verify via digest lookup that you don't already have it." This approach ensures that the current request is served from cache if the required payload has ever been received before, and therefore only compulsory misses occur; redundant transfers cannot. Jeff Mogul and I devised this scheme independently and call it "Duplicate Transfer Detection" (DTD).

Figure 7.5 describes, in pseudocode, how conventional and DTD caches process requests. A DTD cache conceptually maintains *two* caches of received payloads: a "u_cache" indexed by URLs and a "d_cache" indexed by payload digest. (A reasonable implementation, of course, would involve two indices into a single cache of payloads.) Consistency checks (revalidations) are omitted from the pseudocode for brevity and clarity; in practice such checks are necessary to ensure semantic transparency. Section 7.5 discusses DTD as a realizable protocol design. For the present we simply note that compulsory miss rates represent an *achievable* bound on the performance of infinite-capacity caches and that DTD is one way of achieving this bound.

Our idealized cache models differ only in their susceptibility to misses due to the way they associate stored payloads with URLs. In other words, URL-indexed caching accounts for the difference between the miss rate of an infinite-capacity conventional cache and the compulsory miss rate inherent in the reference sequence. By comparing the two we can quantify precisely what fraction of payload transfers to a URL-indexed cache are redundant; all such redundant transfers are are due to the interplay between server-end content-naming practices, client access patterns, and conventional URL-indexed caching.

Aliased payloads can cause redundant transfers for URL-indexed caches, but are not the only cause. For example, a single resource (URL) whose value alternates between two payloads can also cause redundant transfers. Multiple causes may occur together, so a redundant transfer could fall into several categories. The abstract models in Figure 7.5, and the simulations in Section 7.3.2, do not isolate the contribution of aliasing to redundant transfers. Instead they consider the performance penalty that arises from the interaction between URL-indexed caching and complexities in the namespace, e.g., aliasing and resource modification.

Conventional URL-indexed cache

```
if cache[URL] = correct payload
    conventional_hit++
else
    compulsory_miss_or_redundant_transfer++
    send URL
    receive payload
    cache[URL] ← payload
```

DTD cache

```
if u_cache[URL] = correct payload
    conventional_hit++
else
    send URL
    receive payload digest
    if d_cache[digest] = correct payload
        DTD_hit++
        send "don't bother"
    else
        compulsory_miss++
        send "proceed"
        receive payload
        d_cache[digest] ← payload
        u_cache[URL] ← payload
```

Figure 7.5: URL-indexed and DTD caches.

7.3.2 Simulation Results

I computed compulsory and URL-indexed miss rates and byte miss rates for 1) the aggregate browser cache population of over 37,000 clients, 2) a shared proxy cache serving cacheless clients, 3) a proxy serving infinite URL-indexed browser caches, and 4) a proxy serving infinite DTD browser caches. Table 7.4 shows the results, including the percentage of redundant payload retrievals made by an infinite URL-indexed cache. The table separately reports cold and warm proxy simulation results; I used the first nine days of transactions in the sixteen-day WebTV trace to warm the simulated proxy for the latter. (I did not simulate warm clients because at no time are all client caches equally warm: At any given point in the overall trace, some clients have issued many requests while others have issued few or none.)

The results show that conventional URL-indexed caching entails large numbers of redundant transfers at both levels of the cache hierarchy: Nearly 10% of payload transfers to clients and over 20% of payload transfers to a shared proxy are redundant. Even if redundant proxy-to-browser payload transfers are eliminated by DTD browser caches, nearly 12% of payload transfers to a URL-indexed proxy would be redundant. Under 4% of the network traffic between the proxy and infinite-capacity conventional clients is redundant, as is over 13% of the traffic reaching a URL-indexed proxy serving infinite URL-indexed or cacheless clients.

The WebTV system is a somewhat peculiar environment, and it is reasonable to suspect that thin-client surfing might differ systematically from browsing with conventional rich clients. I therefore repeated the performance evaluations using a large proxy trace recorded on the Compaq corporate network. This trace, described in detail by Mogul [116], is summarized in Table 7.5 (the number of client hosts in the table is an underestimate, due to the loss of backup tapes at Compaq). Like the WebTV trace, the Compaq trace contains payload digests, but because browser caches were enabled it reflects only browser cache misses. Therefore the Compaq trace cannot be used to evaluate browser cache performance. As with the WebTV data, I use a reduced Compaq trace containing only status-200 transactions; a small number of erroneous transactions are also excluded.

The last two rows of Table 7.4 show simulated miss rates and byte miss rates for an infinite-capacity shared proxy serving the Compaq workload. For the warm-proxy results I warm the simulated cache with the first 50 million transactions and tabulate miss rates

Simulated cache	Miss Rates			Byte Miss Rates		
	URL-indexed	compulsory	% redundant	URL-indexed	compulsory	% redundant
cold ∞-cache clients	29.45	26.57	9.78	54.02	52.00	3.75
cold proxy serving cacheless clients	14.35	11.22	21.85	37.37	32.40	13.31
∞ URL-indexed clients	48.55	38.08	21.55	69.02	59.97	13.11
∞ DTD clients	47.83	42.21	11.75	69.27	62.30	10.06
warm proxy serving cacheless clients	12.93	9.93	23.14	35.58	30.40	14.55
∞ URL-indexed clients	46.30	35.74	22.80	67.42	57.77	14.32
∞ DTD clients	45.48	40.09	11.85	67.79	60.24	11.14
cold Compaq proxy, caching clients	46.84	38.77	17.24	67.49	59.53	11.79
warm Compaq proxy, caching clients	44.90	36.58	18.54	65.50	56.56	13.65

Table 7.4: URL-indexed and compulsory miss rates and % of URL-indexed payload transfers that are redundant.

Clients	at least 21,806
Server hostnames	at most 454,424
URLs	19,644,961
Unique payloads	30,591,044
(URL, payload) pairs	34,848,044
Transactions	78,913,349
Bytes transferred	
Total	902,792,408,397
Unique payloads	537,460,558,056

Table 7.5: Compaq reduced trace summary statistics.

based only on subsequent transactions. The Compaq results are roughly similar to the WebTV results: Over 17% of a URL-indexed cache's payload retrievals are redundant, as is roughly 12% of origin-to-proxy traffic.

7.4 Explaining Redundant Transfers

The original motive for my investigation of aliasing was to explain the high rates of redundant proxy-to-browser payload transfers in the WebTV system described in Section 6.2: Actual client miss rates are far higher than predicted by a simulated client-cache model that includes only compulsory and capacity misses. Redundant transfers can result from at least three causes: 1) faulty metadata supplied by origin servers or intermediaries, 2) poor browser cache management, and 3) URL-indexed caching. It now appears that URL-indexed caching accounts for a substantial fraction of redundant payload transfers to WebTV clients, but not all of them. A thorough investigation of the remaining possibilities is the subject of my ongoing research; I offer a few tentative observations below.

Inappropriate metadata appears frequently in reply headers, and sometimes takes surprising forms. For instance, in the WebTV trace, different replies from the same server containing the same payload sometimes contain *different* entity tags. This curious phenomenon can cause "`If-None-Match`" revalidation attempts to fail needlessly, resulting in redundant payload transfers. Other researchers have explained this problem, which arises when large server farms fail to harmonize entity tags across server replicas [169].

Mogul investigated erroneous HTTP timestamps in a large trace and reported that 38% of responses contained impossible `Date` header values and 0.3% had impossible

`Last-Modified` values [115]. Some timestamp errors might cause transparency failures; others might cause needless revalidations. Wills & Mikhailov report a different kind of timestamp error: The `Last-Modified` reply header of a resource sometimes changes even when the reply body does not [169].

Anecdotal evidence suggests that Web design tools do not encourage content creators to associate reasonable expiration dates with pages. This is unfortunate because many commercial sites might incorporate business rules into `Expires` headers, e.g., "resources are modified only during business hours"; however, such practices seem to be rare. Finally, origin servers are not the only source of faulty metadata. The popular Squid proxy, for example, does not update cached object headers after revalidations [126].

The WebTV browser cache truncates expiration dates to a maximum of 24 hours; this may increase the number of revalidations but in itself will not cause redundant payload transfers. However the WebTV browser furthermore *evicts* expired items rather than revalidating them with conditional GET requests [37, 151]. The Mozilla browser cache, by contrast, is designed to comply with the letter and spirit of HTTP/1.1 [65, 120]. WebTV's strategy prevents transparency failures when expiration dates are overly optimistic and might simplify implementation in memory-constrained client devices, but it might also inflate client miss rates. It appears to follow the obsolete HTTP/1.0 standard (RFC 1945 of May 1996), which explicitly states that expired cache entries should be discarded:

> The Expires entity-header field gives the date/time after which the entity should be considered stale. .... *Applications must not cache this entity beyond the date given.* [24] (emphasis added)

The current standard specifies far more reasonable behavior:

> The Expires entity-header field gives the date/time after which the response is considered stale. A stale cache entry may not normally be returned by a cache ... *unless it is first validated with the origin server*.... [64] (emphasis added)

7.5 Avoiding Redundant Transfers

Section 7.3.1 described an abstract model for a cache that suffers only compulsory misses. Here I sketch how this can be realized in a practical protocol design, as an extension

to HTTP. This design averts *all* redundant payload transfers, including but not limited to those caused by aliasing. Jeff Mogul and I conceived this simple scheme independently; we call it "Duplicate Transfer Detection" (DTD). DTD can be applied both to client and proxy caches.

First, consider the behavior of a traditional HTTP cache. Such a cache is URL-indexed: If the cache finds that it does not currently hold an entry for a requested URL U, this is a cache miss. On a miss, the cache issues or forwards a request for the URL toward the origin server, which would normally send a response containing payload P. If the cache holds an expired entry for U, it may send a "conditional" request, and if the server's view of the resource has not changed, it may return a "Not Modified" response without a payload. Real HTTP caches differ from the abstract URL-indexed model defined in Section 7.3.1 because they implement HTTP's cache-consistency mechanisms, and so may suffer redundant transfers resulting from inappropriate metadata and from self-inflicted wounds, so to speak (see Section 7.4), as well as from "namespace confusion" (aliasing and resource modification).

Now consider an idealized, infinite cache that retains in storage every payload it has ever received, even those that a traditional HTTP cache would not treat as valid cache entries. A finite, URL-indexed cache differs from this idealization because it implements both an update policy (it stores only the most recent payload received for any given URL), and a replacement policy (it stores only a finite set of entries, selected for maximum expected value).

The concept behind Duplicate Transfer Detection is quite simple: If our idealized cache can determine, before receiving the payload, whether it had ever previously received P, then we can avoid transferring that payload. Such a cache would experience only compulsory misses and would never suffer redundant payload transfers. A finite-cache realization of DTD would, of course, also suffer capacity misses.

How does the cache know whether it has received a payload P before the server sends the entire response? DTD follows the model of the abstract cache described in Section 7.3.1. The cache maintains one set of cache entries but two lookup tables: one indexed by URL, and one indexed by the digest of each stored payload. If a DTD cache finds no fresh entry under the requested URL U, it forwards a (possibly conditional) request to the origin server. If the server replies with a payload, it first sends the digest D of the payload,

and the cache checks for a stored entry with a matching digest value. On a digest match, the cache can signal the server not to send the payload (although the server must still send the HTTP message headers, which might be different). Thus, while DTD does not avoid transferring the request and response message headers, it can avoid any redundant payload transfer. We say that a "DTD hit" occurs when DTD prevents a payload transfer that would have occurred in a conventional URL-indexed cache.

An idealized, infinite DTD cache stores *all* payloads that it has received. In particular, it does not delete a payload P from storage simply because it has received a different payload P' for the same URL U. A realistic, finite DTD cache will eventually delete payloads from its storage, based on some replacement policy. A DTD cache might benefit from retaining old cache entries that other cache replacement and update algorithms would discard, speculating that such an entry will yield a future DTD hit.

DTD can be implemented in at least two ways. One minimizes bandwidth consumption by ensuring that redundant payload transfers never occur; however it sometimes entails an extra round-trip time (RTT) between data receiver and sender. The other implementation strategy minimizes client latency by ensuring that data is transmitted as rapidly as possible; however it sometimes entails redundant data transmission. To minimize bandwidth, on receiving a request the server sends the response headers (including the payload digest) but defers sending the payload until the client sends an explicit "proceed" request. To minimize latency, the server sends the payload immediately, but stops if the client sends an "abort" message. The "proceed" model imposes an extra RTT on every compulsory and capacity miss, but never sends any redundant payload bytes. (A more intricate form of the "proceed" model could amortize this delay over several misses, as described in Section 7.5.1.) The "abort" model does not impose additional delays, but the abort message may fail to reach the server in time to prevent redundant traffic from being transmitted.

Finally, note that DTD caches allow us to dispense entirely with the curious notion of "uncachable content." No special treatment is required for "dynamic" content generated by CGI scripts or content customized via cookies. With DTD, a payload is a payload, and all payloads are equally cachable. DTD guarantees that redundant transfers never occur, and it requires no sacrifice in semantic transparency.

7.5.1 Latency Analysis

It is straightforward to analyze the relative latencies of a conventional URL-indexed cache and a DTD cache (using the "proceed" model). Ignoring revalidations and assuming that cache hits are instantaneous and request messages are negligibly small, the average-case latencies of these two approaches are given by the following expressions:

$$\begin{array}{rrcl} & \text{DTD hit} & & \text{miss} \\ \text{URL}-\text{indexed}: & P_d(R+S_p/B) & + & P_m(R+S_p/B) \\ \text{DTD}: & P_d(R+S_d/B) & + & P_m(R+S_d/B+R+S_p/B) \end{array}$$

where P_d is the probability that a request misses in a URL-indexed cache but would hit in a DTD cache, P_m is the probability of a compulsory or capacity miss, R and B are the round-trip latency and bandwidth of the communications medium, respectively, S_p is the mean size of a reply payload, and S_d is the size of a payload digest.

The latency of a URL-indexed cache is greater when

$$\frac{P_d}{P_m} > \frac{BR+S_d}{S_p-S_d}$$

Substituting into this equation parameter estimates for the WebTV system with infinite client caches ($P_d = 2.89\%$, $P_m = 26.57\%$, $B = 33.6$ Kbps, $R = 100$ ms, $S_d = 32$ bytes, $S_p = 13,441$ bytes) we find that DTD offers lower latency (0.908 sec/transaction versus 0.972 sec/transaction, on average). DTD offers lower latency for S_p values larger than roughly 3600 bytes.

This average analysis overestimates the RTT overhead of the "proceed" model because it considers transactions individually. A reasonable browser-proxy implementation of DTD would likely "bundle" requests and payload digests using HTTP pipelining over persistent connections, in the common case where the client requests an HTML container page followed by many embedded page components. In such an implementation the number of wasted RTTs would not exceed two per compound Web page. In the WebTV workload the ratio of HTML replies to GIF and JPEG replies is roughly 1:9, which implies a worst case of two DTD RTTs wasted per ten HTTP requests. Trans-continental RTT is roughly

100 ms in the United States, and the additional latency of two RTTs would barely exceed the threshold of perception.

Note that the "abort" model for DTD, in which the server simply transmits a normal reply preceded by a payload digest rather than waiting for a "proceed" message, *always* results in receiver latency less than or equal to that of URL-indexed caching. However the "proceed" model conserves bandwidth by ensuring that no fraction of a payload is ever transmitted to a receiver that already has it. Because of the complexity of TCP behavior, it is nearly impossible to use an average-case analysis to estimate the bandwidth savings in the "abort" model.

The foregoing average-case analysis omits many details and therefore may not accurately predict the performance of actual DTD implementations. A more detailed analytic and empirical evaluation of the costs and benefits of DTD is the subject of my ongoing work with Jeff Mogul and Yee Man Chan. However the simplified performance model above formalizes our intuition that in low-bandwidth/low-RTT environments the "proceed" model entails little latency penalty.

7.5.2 Security Issues

Measures that improve the performance of computing systems often create subtle security vulnerabilities, and caching is a prime example. Timing attacks on processor memory hierarchies have been known for decades, e.g., the famous TENEX vulnerability described in Tanenbaum's OS text [153, pp. 183–4]. Recently Felten et al. have described variants applicable to Web cache hierarchies [63]. At least two new security problems arise when DTD is used in cache hierarchies.

First, if an attacker can generate payload digest collisions, then she can cause a DTD proxy to deliver incorrect payloads, as illustrated by the following scenario:

1. User Alice's browser forwards requests to a shared proxy. DTD is employed in the server-to-proxy hop.

2. Alice issues a request for which the correct reply payload is P.

3. Evildoer Eve has previously created a payload P' such that $\text{digest}(P') = \text{digest}(P)$ and has caused it to be cached in the proxy (this can occur in the course of an ordinary

transaction; the proxy need not deviate from its prescribed behavior nor "collude" with Eve in any sense).

4. The proxy forwards Alice's request to the origin server, which replies with digest(P).

5. The proxy finds that it already has a cached payload P' with a matching digest, and therefore does not retrieve payload P from the origin server.

6. The proxy transmits Eve's incorrect payload P' to Alice.

This attack can be prevented through the use of secure message digest functions such as MD5 [139] and SHA1 [127], for which it is thought to be computationally infeasible to generate collisions.

A more subtle problem involves information leakage; interestingly, the attack does *not* rely on timing information of any kind. A server can exploit DTD to learn the contents of a client's cache:

1. User Bob's browser and the `nosy.com` server employ DTD.

2. Bob issues a request for URL `http://nosy.com/humdrum.html`.

3. `nosy.com` replies with digest(`naughty.gif`), even though it never receives or serves requests for this interesting payload.

4. Bob's browser tells server "already got that," thereby revealing something about Bob's past surfing.

Sophisticated implementations of this attack might employ JavaScript within HTML pages to systematically search a client's cache for interesting payloads, analogous to the timing attacks described by Felten et al [63]. Attacks of this form can be detected by simply always sending "proceed" instead of "already got that," retrieving the reply payload, and verifying that its digest matches the one sent by the server. Of course, this countermeasure negates the benefits of DTD and should perhaps be done by "privacy auditors" using simulated clients, rather than by ordinary users. Another countermeasure is to permit users to "opt out" of DTD by disabling this feature, perhaps on a selective, site-by-site basis. A still more cautious countermeasure would be to employ DTD only within individual sites. However this approach may severely limit the benefits of DTD, because as noted in Section 7.2.2 most aliasing occurs *across* sites rather than within sites.

7.5.3 Similar Proposals

DTD is an application-level analogue of the network-level approach proposed by Santos & Wetherall [143] and Spring & Wetherall [150], in which packet-payload digests are used to avert redundant data transfers between *routers*. The two approaches are in some sense complementary, because each can avoid some redundant data transfers eliminated by the other. For example, the router-based approach can eliminate transfers of common prefixes of slightly different payloads, while DTD does not suffer from the re-packetization potentially caused by pipelining in HTTP. The approaches also differ in adoption dynamics: The router-level technique is easier to deploy for an organization that controls network infrastructure, while the application-level technique may be preferable for a single organization that controls two levels of a cache hierarchy (e.g., the client and proxy caches of AOL or WebTV).

7.5.4 Alternatives to DTD

While DTD is a simple and fully general solution to the problem of redundant payload transfers, it comes with a price. As shown in Section 7.5.1, the bandwidth-saving "proceed" variant of DTD typically imposes additional latency on clients, and the latency-minimizing "abort" variant sometimes saves little bandwidth. If we can attribute most redundant transfers to a single cause, perhaps we can devise a less general and less costly solution than DTD.

As noted in Sections 7.2.1 and 7.2.2, the aliasing observed in the WebTV trace is consistent with the hypothesis that Web authoring tools such as FrontPage have filled the Web with aliased images, and that these account for much of the aliasing in Web transactions. The WebTV and Compaq traces are anonymized in such a way as to preclude definitive investigation of this issue, but if future research confirms the authoring-tool conjecture several special-purpose solutions are available.

Instead of installing redundant copies of "clip art" images at every customer site, Web authoring tools might serve each image from a central server, or from a content distribution network such as Akamai. The former approach eliminates aliasing but creates a single point of failure for all Web sites that use the authoring tool. It might also raise privacy concerns, because the image server might learn about the workloads of sites that use the authoring tool

(e.g., through Referer headers). The CDN approach eliminates aliasing without introducing fragility, and furthermore may improve end-user latency.

Going one step further, we can imagine bundling the clip art images *with browsers* instead of with Web authoring tools. (In other words, transmit images from FrontPage to Internet Explorer once, in Redmond, rather than millions of times via the Web.) This approach eliminates even *compulsory* misses and requires no modifications to the HTTP protocol. One implementation strategy would be to create a new type of HTML image tag that contains both a payload digest and a URL. The meaning of the new tag is that an image with the given digest should be inserted. If such an image cannot be found in the browser cache it can be fetched using the given URL, ensuring that older browsers not bundled with newer clip art will still function properly.

7.6 Discussion

My results show that aliasing is a surprisingly prevalent phenomenon in Web transactions; by several measures it is present to a greater extent than resource modification, yet it has received far less attention in the research literature. I have also shown that namespace phenomena such as aliasing and resource modification are not merely academic curiosities. My analysis of two large workloads from very different environments demonstrates that content-naming practices interact with client access patterns in such a way as to cause substantial numbers of redundant retrievals in conventional URL-indexed Web caches. The costs of redundant transfers in terms of end-user latency and server load are difficult to quantify in monetary terms, but the monetary cost in terms of wasted bandwidth can be computed easily in environments where bandwidth costs are straightforward. The Financial Director of IT at the University of Michigan reports that roughly 15% of traffic between U-M and the outside world is Web-related, that our institution currently pays $1.9 million per year for bandwidth, and that this cost is roughly linear in bandwidth consumed [76, 132]. If roughly 14% of origin-to-proxy Web traffic is redundant in the warm steady state, as suggested by Table 7.4, it costs U-M nearly $40,000 annually.

The Mogul/Kelly Duplicate Transfer Detection protocol extension provides a fully general way to eliminate redundant transfers, regardless of cause. It is conceptually very simple; veteran Web browser and proxy implementors have assured me that it would be easy to

implement and deploy in a production environment [101, 151]. Because it is a hop-by-hop mechanism, adoption dynamics pose few difficulties in any environment where adjacent HTTP endpoints are inclined to cooperate, e.g., between the proxy and browser caches of AOL or WebTV, or between a content provider and the edge servers of a CDN. Finally, DTD permits so-called "dynamic" content to be cached safely and consistently without any special treatment.

When DTD is used to eliminate redundant transfers (the "proceed" model), it sometimes introduces an extra round trip into transactions. Although in typical cases this RTT is small and can furthermore be amortized over several HTTP requests, in some cases it might be too high a price to pay. My analysis is consistent with the hypothesis that many of the redundant transfers that occur in practice may be due to aliasing caused by the "clip art" images bundled with Web authoring tools like FrontPage, DreamWeaver, etc. If this proves to be the case, then DTD is not the only remedy available; the little images could be served from a single site, served from a CDN, or bundled with browsers.

CHAPTER 8

Summary and Future Work

To scale and to serve offered workload efficiently, distributed information retrieval systems such as the World Wide Web must employ caching to the fullest extent permitted by offered workload. In this thesis I demonstrate that today's Web caches fall short of the performance bounds they might attain, and I contribute several methods for optimizing or improving the performance of Web caches along several dimensions, including

1. a removal policy that strives to maximize the run-time value of caches as defined by stakeholder (content provider) preferences;

2. a method of precisely computing cost-minimizing cache sizes under realistic and very general workload and cost assumptions; and

3. a fully general method of eliminating redundant data-payload transfers.

One distinctive feature of my research is that I adopt generalized notions of cost and performance that support sensitivity to system stakeholder preferences. Another is that I focus on system-wide interactions, end-to-end performance, and fundamental exogenous system workloads rather than on component workloads and performance. Finally, for my empirical work I have obtained several of the most detailed and the largest data sets in existence; these in turn required me to advance the state of the art in analytic techniques.

Beyond my practical contributions, I make a number of noteworthy observations. Contrary to conclusions drawn from studies of distributed file systems, I find that sharing across Web client reference streams is so strong that a shared proxy cache can attain high hit rates

even if dedicated client caches are large. In a similar vein, I report that the *marginal* benefits of proxy caching *increase* with browser cache size. Finally, I prove that under certain circumstances a decentralized algorithm can compute globally-optimal cache sizes at every node in a large two-level branching hierarchy.

In Section 8.1 I describe how my contributions fit together to improve our understanding of Web caching and in Section 8.2 suggest avenues for future work.

8.1 Summary of Contributions

Several of my contributions can be viewed as generalizations of conventional notions of "workload" and "performance." Biased Web cache removal policies have been proposed before, but the insight that miss penalties can reflect stakeholder (e.g., content provider) preferences and that such policies can therefore provide variable QoS through dynamic adaptation to preferences is original to the MARX project [105], which provided the first detailed evaluation of such a scheme [86, 87] (Chuang discusses variable-QoS cache management schemes in very general terms but does not evaluate specific methods [46].) This perspective broadens our concept of workload to include heterogeneous stakeholder preferences and allows us to accommodate these preferences within the existing framework of removal policies. Similarly the idea that capacity planning can optimize the caching tradeoff between the costs of bandwidth and storage is not new. However the optimal sizing method I introduced generalizes existing data-engineering rules of thumb to non-uniform document sizes and completely arbitrary per-access miss costs. The latter may reflect any criteria whatsoever, including the preferences of the content consumers and providers involved in each transaction.

Furthermore my research has investigated in novel ways the interplay between different workload characteristics as they affect caching. For instance, I have demonstrated (Section 3.4) that it will be difficult for weighted-LFU removal policies to adapt to heterogeneous client preferences if document popularity and per-client hit valuations are uncorrelated. This is only a partial negative result; it proves that one class of biased replacement policies is ill-suited to a particular combination of workload characteristics, but does not preclude the possibility that client preferences can guide welfare-maximizing cache management. To take another example, I have shown that the fundamental exogenous workload

entering the Web from opposite ends—content naming and client access patterns—interact in such a way as to cause many unnecessary misses in conventional URL-indexed caches. In response to this finding I devised a straightforward and comprehensive payload-transfer optimization mechanism that completely eliminates redundant transfers. These results underscore the importance of a system-level perspective and the perils of focusing exclusively on a subset of components or workload characteristics.

In several ways my results focus our attention specifically on *large-scale* caching systems. My proof that decentralized self-interested computations yield globally-optimal cache sizes throughout a two-level branching cache hierarchy holds only in the limit as the client population grows large; this result is somewhat reminiscent of the competitiveness assumption underlying key results in microeconomic theory [106, Section 12.F]. In terms of empirical methodology the scale of my investigations is in many ways unprecedented. The cache-busting proxy method of trace collection is neither highly original nor highly sophisticated. Yet I was the first to employ this technique and demonstrate its viability by collecting a client trace two orders of magnitude larger than any other reported in Web-related literature. To analyze this vast quantity of data I developed fundamentally more efficient simulation methods, e.g., a parallel general-purpose simulator and a generalized fast single-pass stack simulation algorithm. The latter represents an incremental advance over techniques that are well known among hardware designers, but it represents a dramatic improvement in efficiency compared with the methods that Web caching researchers had previously employed.

Finally, from a practical standpoint one of my most important contributions is to affirm that shared intermediate proxy caches can substantially improve important performance metrics for today's Web workloads. Sharing across Web client request streams is far stronger than in distributed file systems, and the merged miss sequences of many Web clients contain sufficient re-referencing to support high proxy hit rates even if client caches are large. Furthermore I have shown that the *marginal* benefits of proxy capacity expansion are *greater* when browser caches are large. These findings demonstrate that proxies can play an important role in reducing server loads and network traffic, and that client capacity expansion *increases* the motivation for proxy capacity expansion.

8.2 Future Work

In the near term I intend to evaluate more thoroughly the latency effects of Duplicate Transfer Detection using microbenchmarks and prototype implementations. Jeff Mogul and I have been joined in our ongoing DTD work by MARX project veteran Yee Man Chan, who at the time of this writing is implementing DTD in the Squid freeware proxy. In the longer term DTD should be deployed in a well-instrumented production environment. My efforts to persuade the WebTV browser, proxy and network operations teams to implement and deploy DTD continue. A critical mass of enthusiasm for such a project has not yet accumulated among key personnel at WebTV, but I'm confident that a persuasive detailed case for DTD will lead to large scale testing.

Content naming and its impact on conventional cache performance remains a promising area for future research. The "proceed" variant of DTD can completely eliminate redundant payload transfers resulting from the combination of namespace complexity, user access patterns and URL-indexed caching. However it entails a latency penalty, and if a single underlying cause that accounts for the bulk of redundant transfers can be identified and eliminated, we might obtain all the benefits of DTD without this penalty. My findings in Chapter 7 suggest the intriguing possibility that authoring tools have populated the Web with large numbers of "clip art" image files that are accessed via multiple URLs. A recent trace that records both payload checksums and *plaintext* URLs would be required to definitively investigate this hypothesis (the WebTV trace is unsuitable because both URLs and payload digests are irreversibly anonymized). If it proves to be correct, a number of interventions are possible. The most obvious is to serve each clip art image from a single server site, e.g., `images.frontpage.com`, rather than from every site that uses the tool. The images might also be served from a CDN's edge servers, or even bundled with browsers. A detailed evaluation of the relative merits of these approaches and their potential benefits on the real Web would be a significant contribution.

The cache-busting proxy method of Chapter 6 seems destined for further use among researchers. Adam Bradley of Boston University has implemented a cache-busting proxy, and his team intends to use it to collect data for their research [30]. An obvious enhancement to the basic technique would be to alter replies to actively flush client caches, e.g., by appending semantically insignificant newline characters to HTML pages served.

The optimal cache sizing method of Section 4.4 can be applied only in an environment where both cache workload and miss penalties are available. Unfortunately my (admittedly limited) experience has shown that environments where miss penalties are readily available, e.g., institutions like the University of Michigan where bandwidth costs increase linearly with traffic volume, happen to be places where workload is very difficult to measure on a large scale. Conversely, environments like WebTV where large-scale workload measurement is feasible have proven to be those where credible monetary miss penalties are very difficult to estimate. A natural next step in the evaluation of the method is to identify a production system where workload measurement and cost estimation can be done at reasonable cost and quantify the monetary savings that would result from optimally-sized caches in such an environment. An orthogonal avenue of development for the optimal sizing method is to explore ways of generalizing it to accept per-reference penalties for successful revalidations as well as payload fetches; such an extension is desirable for situations where costs reflect the disutility of latency.

Storage costs have plummeted in recent years and the decline is expected to continue; some might therefore argue that optimal cache sizing methods will become unnecessary. This line of reasoning is faulty for several reasons. First, cost *ratios* rather than absolute costs should guide design, and technology price ratios relevant to a wide range of computer system design problems have in some cases remained remarkably constant over periods of many years [70, 72]. Furthermore workloads often expand at rates that keep pace with technological improvements and cost decreases. As entertainment content moves from broadcast media to retrieval-on-demand systems, optimal capacity planning will take on new importance. The music stores and video rental outlets of tomorrow are shared caches, and they must be designed well to compete.

Future research on preference-sensitive removal policies should emphasize measuring stakeholder preferences rather than comparisons among biased replacement policies using randomly-generated preferences. As I have shown in Chapter 3, the basic method requires preferences that vary over large scales when applied to the problem of maximizing value to content providers. It furthermore requires that preferences be correlated with document reference counts when applied to the problem of maximizing client value, as explained in Section 3.4. The next step for research on applications of biased removal policies to run-time value maximization is to determine whether these requirements are satisfied in

practice. If they are, then it will be possible to conduct credible evaluations of the overall method on real workloads and real client preferences using the most recent cost-aware removal policies, e.g., GreedyDual* [82]. Readers interested in pursuing such work are warned that the Web caching community has grown weary of studies comparing biased removal policies. The first panel discussion at the 2001 Web Caching Workshop, which occasioned little debate and much nodding, considered the proposal, "Resolved: Publish no more papers on Web cache replacement policies." Future research should emphasize the measurement of user value and potential benefits of run-time maximization rather than comparisons of replacement policies.

Much of my empirical research follows a simple paradigm characteristic of research in experimental computer science: 1) measure fundamental, exogenous, system-level workload on a large scale in an important production environment; 2) measure performance bounds inherent in the workload itself, independent of the system currently serving it; 3) identify gaps between the actual and potential performance of the system under study; and 4) devise ways of closing these gaps that are compatible with existing components, architectures, protocols and standards. This approach has proven fruitful for my study of Web cache hierarchies, and I believe that it is applicable to many other systems. In particular I conjecture that a wide range of emerging "Web services" [44] will not initially be optimized for the applications that are built on them. Like early Web cache hierarchies, first-generation Web services will likely be based on sub-optimal system architectures and protocols, designed with insufficient workload knowledge, and deployed in haste. Much low-hanging fruit will grow as applications based on Web services mature, and the methods I have applied in my thesis research are a promising way of harvesting it.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Atul Adya, Paramvir Bahl, and Lili Qiu. Analyzing the browse patterns of mobile clients. In *SIGCOMM Internet Measurement Workshop*, November 2001. `http://research.microsoft.com/~liliq/papers/pub/IMW2001.pdf`.

[2] Inc. Alexa Internet. `http://www.alexa.com/`.

[3] Jussara Almeida, Mihaela Dabu, Anand Manikutty, and Pei Cao. Providing differentiated levels of service in Web content hosting. In *Proceedings of the ACM SIGMETRICS Workshop on Internet Server Performance (WISP)*, 1998.

[4] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS96)*, December 1996. Reference [5] is longer and older.

[5] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the WWW. Technical Report TR-96-11, Boston University Computer Science Department, 1996. `http://www.cs.bu.edu/techreports/`.

[6] Virgílio Almeida, Daniel Menascé, Rudolf Riedi, Flávia Peligrinelli, Rodrigo Fonseca, and Wagner Meira Jr. Analyzing Web robots and their impact on caching. In *Proceedings of the Sixth International Workshop on Web Caching and Content Delivery*, June 2001.

[7] Jörn Altmann, Björn Rupp, and Pravin Varaiya. Internet demand under different pricing schemes. In *Proceedings of the ACM Conference on Electronic Commerce (EC'99), Denver, CO*, November 1999.

[8] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. MINERVA: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, November 2001.

[9] Martin Arlitt. Personal communication.

[10] Martin Arlitt, Ludmila Cherkasova, John Dilley, Richard Friedrich, and Tai Jin. Evaluating content management techniques for Web proxy caches. In *Proceedings of the Second Workshop on Internet Server Performance (WISP '99)*, May 1999.

[11] Martin Arlitt, Ludmila Cherkasova, John Dilley, Richard Friedrich, and Tai Jin. Evaluating content management techniques for Web proxy caches. Technical Report HPL-98-173, HP Labs, March 1999. `http://www.hpl.hp.com/techreports/98/HPL-98-173.html`.

[12] Martin Arlitt, Rich Friedrich, and Tai Jin. Workload characterization of a Web proxy in a cable modem environment. Technical Report HPL-1999-48, Hewlett-Packard Laboratories, 1999. `http://www.hpl.hp.com/techreports/1999/HPL-1999-48.html`.

[13] Martin Arlitt and Tai Jin. Workload characterization of the 1998 World Cup Web site. Technical Report HPL-1999-35R1, Hewlett-Packard Labs, September 1999. `http://www.hpl.hp.com/techreports/1999/HPL-1999-35R1.html`.

[14] Martin Arlitt and Carey Williamson. Web server workload characterization: The search for invariants. In *Proceedings of ACM SIGMETRICS*, May 1996.

[15] Martin F. Arlitt and Carey L. Williamson. Internet Web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–644, October 1997.

[16] Jean-Loup Baer and Wen-Hann Wang. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 73–80, 1988.

[17] Hyokyung Bahn, Hyunsook Lee, Sam H. Noh, Sang Lyul Min, and Kern Koh. Replica-aware caching for Web proxies. *Computer Communications*, 25(3):183–188, February 2002.

[18] Hyokyung Bahn, Sam H. Noh, Kern Koh, and Sang Lyul Min. Using full reference history for efficient document replacement in Web caches. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, November 1999. `http://www.cs.hongik.ac.kr/~dnps/research/pub.html`.

[19] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in Web client access patterns: Characteristics and caching implications. *World Wide Web Journal, Special Issue on Characterization and Performance Evaluation*, 1999. Also available as Boston U. CS tech report 1998-023 at `http://www.cs.bu.edu/techreports/`.

[20] Paul Barford and Mark Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the 1998 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 151–160, July 1998. `http://www.cs.bu.edu/faculty/crovella/paper-archive/sigm98-surge.ps`.

[21] J. Fritz Barnes. DavisSim: Another Web cache simulator, October 1999. http://arthur.cs.ucdavis.edu/projects/qosweb/DavisSim.html.

[22] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[23] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, July 1975.

[24] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext transfer protocol – HTTP/1.0, May 1996. See Reference [138].

[25] T. Berners-Lee, L. Masinter, and M. McCahill. RFC 1738: Uniform resource locators (URL), December 1994. See Reference [138].

[26] Azer Bestavros, Robert Carter, Mark Crovella, Carlos Cunha, Abdelsalam Heddaya, and Sulaiman Mirdad. Application-level document caching in the Internet. In *Proceedings of the Second International Workshop on Services in Distributed and Networked Environments (IEEE SDNE'95)*, June 1995. http://www.cs.bu.edu/fac/best/res/papers/sdne95.ps.

[27] Krishna Bharat and Andrei Broder. Mirror, mirror on the Web: A study of host pairs with replicated content. In *Proceedings of the Eighth International World Wide Web Conference*, May 1999. http://www8.org/w8-papers/4c-server/mirror/mirror.html.

[28] Krishna Bharat, Andrei Broder, Jeffrey Dean, and Monika R. Henzinger. A comparison of techniques to find mirrored hosts on the WWW. In *Proceedings of the Workshop on Organizing Web Space at the Fourth ACM Conference on Digital Libraries 1999*, August 1999.

[29] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. Technical Report HPL-2000-3, HP Labs, January 2000. http://www.hpl.hp.com/techreports/2000/HPL-2000-3.html.

[30] Adam Bradley. Personal communication.

[31] Adam Bradley. Cache-busting HTTP/1.1 caching proxy, July 2002. Source code: http://cs-people.bu.edu/artdodge/research/reflex/release/0.99/ Instructions: http://cs-people.bu.edu/artdodge/research/reflex/release/docs/cachebustingproxy.php.

[32] Richard A. Brealey and Stewart C. Meyers. *Principles of Corporate Finance*. McGraw-Hill, sixth edition, 2000. ISBN 0-07-117901-1.

[33] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Infocom99*, March 1999. Tech report version available at http://www.cs.wisc.edu/~cao/papers/.

[34] Brian E. Brewington. *Observation of changing information sources*. PhD thesis, Dartmouth, June 2000. http://actcomm.dartmouth.edu/papers/brewington:thesis.ps.gz.

[35] Brian E. Brewington and George Cybenko. How dynamic is the web? In *Proceedings of the Ninth International World Wide Web Conference*, May 2000. http://www9.org/w9cdrom/264/264.html.

[36] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the Web. In *Proceedings of the Sixth International World Wide Web Conference*, April 1997. http://www.scope.gmd.de/info/www6/technical/paper205/paper205.html.

[37] Jake Brutlag. Personal communication.

[38] Ramón Cáceres, , Fred Douglis, Anja Feldman, Gideon Glass, and Michael Rabinovich. Web proxy caching: The devil is in the details. In *Proceedings of ACM SIGMETRICS Workshop on Internet Server Performance*, 1998. Reference [62] is a later version.

[39] Ramón Cáceres, Balachander Krishnamurthy, and Jennifer Rexford. HTTP 1.0 logs considered harmful. Position Paper, W3C Web Characterization Group Workshop, November 1998. http://www.research.att.com/~jrex/papers/w3c.passant.ps.

[40] Pei Cao and Gideon Glass. Wisconsin Web cache simulator, May 1997. http://www.cs.wisc.edu/~cao/webcache-simulator.html.

[41] Pei Cao and Sandy Irani. Personal communication.

[42] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, pages 193–206, December 1997. http://www.cs.wisc.edu/~cao/papers/gd-size.html.

[43] Lara D. Catledge and James E. Pitkow. Characterizing browsing strategies in the World-Wide Web. *Computer Networks and ISDN Systems*, 27(6):1065–1073, April 1995.

[44] Ethan Cerami. *Web Services Essentials*. O'Reilly, February 2002. An online edition is available at http://safari.oreilly.com/main.asp?bookname=webservess.

[45] Yee Man Chan, Jeffrey K. MacKie-Mason, Jonathan Womer, and Sugih Jamin. One size doesn't fit all: Improving network QoS through preference-driven Web caching. In *Proceedings of the Second Berlin Internet Economics Workshop*, May 1999.

[46] John Chung-I Chuang. *Economies of Scale in Information Dissemination over the Internet*. PhD thesis, Carnegie-Mellon University, November 1998.

[47] Thomas H. Cormen, Charles E. Leiserson, and Ronald R. Rivest. *Introduction to Algorithms*. MIT Press, 1990. ISBN 0-262-03141-8.

[48] Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001. ISBN 0-262-03293-7.

[49] CacheFlow Corporation. White paper: Creating a cache-friendly Web site, April 2001. `http://www.cacheflow.com/technology/whitepapers/index.cfm`.

[50] Microsoft Corporation. Cache array routing protocol and microsoft proxy server 2.0. Technical report, Microsoft Corporation, 1997. `http://www.microsoft.com/ISN/whitepapers.asp`.

[51] 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. `http://www.cs.bu.edu/faculty/crovella/paper-archive/self-sim/journal-version.ps`.

[52] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW client-based traces. Technical Report BU-CS-95-010, Boston University Computer Science Department, July 1995. `http://www.cs.bu.edu/techreports/`. See Reference [19] for a follow-up study.

[53] 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. `http://www.cs.rutgers.edu/~davison/pubs/inet99/`.

[54] Brian D. Davison. Index of Web traces and logs, April 2000. `http://www.web-caching.com/traces-logs.html`.

[55] John Dilley. Personal communication.

[56] John Dilley and Martin Arlitt. Improving proxy cache performance—analyzing three cache replacement policies. Technical Report HPL-1999-142, HP Labs, October 1999.

[57] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of ACM SIGMETRICS*, 1999.

[58] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey 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*, pages 147–158, December 1997.

[59] Ronald P. Doyle, Jeffrey S. Chase, Syam Gadde, and Amin M. Vahdat. The trickle-down effect: Web caching and server request distribution. In *Proceedings of the Sixth International Workshop on Web Caching and Content Distribution*, June 2001. `http://www.cs.bu.edu/techreports/2001-017-wcw01-proceedings/121_doyle.pdf`.

[60] Bradley M. Duska, David Marwood, and Michael J. Feeley. The measured access characteristics of World-Wide-Web client proxy caches. In *Proceedings of the First USENIX Symposium on Internet Technologies and Systems*, pages 23–35, December 1997.

[61] Anja Feldmann. Continuous online extraction of HTTP traces from packet traces. In *Proceedings of W3C Web Characterization Group Workshop*, 1999. `http://www.research.att.com/~anja/feldmann/papers.html`.

[62] 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 '99*, March 1999. Reference [38] is an earlier version.

[63] Edward W. Felten and Michael A. Schneider. Timing attacks on Web privacy. In *Proc. of 7th ACM Conference on Computer and Communications Security*, November 2000. `http://www.cs.princeton.edu/sip/pub/webtiming.pdf`.

[64] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol—HTTP/1.1, June 1999. See Reference [138] or `http://www.w3.org/Protocols/Overview.html`. RFC 2616 obsoletes the January 1997 draft, RFC 2068. Original HTTP 1.0 is described in RFC 1945, dated May 1996. RFC 1945 is 60 pages long; RFC 2616 is 172 pages long. Reference [95] explains some of the additions and changes.

[65] Darin Fisher. Personal communication.

[66] National Laboratory for Applied Network Research. Anonymized access logs. `ftp://ftp.ircache.net/Traces/`.

[67] Syam Gadde. Personal communication.

[68] J. Gecsei. Determining hit ratios for multilevel hierarchies. *IBM Journal of Research and Development*, 18(4):316–327, July 1974.

[69] Jim Gray. Personal communication.

[70] Jim Gray and Goetz Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. Technical Report MSR-TR-97-33, Microsoft Research, September 1997. `http://www.research.microsoft.com/scripts/pubs/trpub.asp`.

[71] Jim Gray and Franco Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *Proceedings of ACM SIGMOD*, May 1987.

[72] Jim Gray and Prashant Shenoy. Rules of thumb in data engineering. Technical Report MS-TR-99-100, Microsoft Research, February 2000. Revised version dated February 2000 http://www.research.microsoft.com/scripts/pubs/trpub.asp.

[73] S. Gribble and E. Brewer. System design issues for Internet middleware services: Deductions from a large client trace. In *Proc. 1st USITS*, pages 207–218, December 1997.

[74] W3C Web Characterization Activity Working Group. Web Characterization Repository. http://researchsmp2.cc.vt.edu/cgi-bin/reposit/index.pl.

[75] Brendan Hannigan, Carl D. Howe, Sharon Chan, and Tom Buss. Why caching matters. Technical report, Forrester Research, Inc., October 1997.

[76] Kurt Hillig. Personal communication. Hillig is with the Network Administration group at the University of Michigan's Information Technology Division.

[77] Saied Hosseini-Khayat. On optimal replacement of nonuniform cache objects. *IEEE Transactions on Computers*, 49(8):769–778, August 2000. ISSN 0018-9340.

[78] The Internet Demand Experiment (INDEX). http://www.INDEX.Berkeley.EDU/public/index.phtml.

[79] Sandy Irani. Page replacement with multi-size pages and applications to Web caching. In *29th ACM STOC*, pages 701–710, May 1997.

[80] Arun Iyengar and Jim Challenger. Improving Web server performance by caching dynamic data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 49–60, December 1997.

[81] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley, 1991. ISBN 0-471-50336-3.

[82] Shudong Jin and Azer Bestavros. GreedyDual* Web Caching Algorithm: Exploiting the Two Sources of Temporal Locality in Web Request Streams. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, May 2000. http://www.cs.bu.edu/fac/best/res/papers/wcw00.ps.

[83] Ted Julian and Brendan Hannigan. The cache appliance opportunity. Technical report, Forrester Research, Inc., January 1998.

[84] Terence Kelly. Priority depth (generalized stack distance) implementation in ANSI C, February 2000. http://ai.eecs.umich.edu/~tpkelly/papers/.

[85] Terence Kelly. Thin-client Web access patterns: Measurements from a cache-busting proxy. *Computer Communications*, 25(4):357–366, March 2002. http://ai.eecs.umich.edu/~tpkelly/papers/wtvwl_comcom.pdf.

[86] Terence Kelly, Yee Man Chan, Sugih Jamin, and Jeffrey K. MacKie-Mason. Biased replacement policies for Web caches: Differential quality-of-service and aggregate user value. In *Fourth International Web Caching Workshop*, March 1999. http://ai.eecs.umich.edu/~tpkelly/papers/wlfu.ps.

[87] Terence Kelly, Sugih Jamin, and Jeffrey K. MacKie-Mason. Variable QoS from shared Web caches: User-centered design and value-sensitive replacement. In *Proceedings of the MIT Workshop on Internet Service Quality Economics (ISQE 99), Cambridge, MA*, December 1999. http://www.marengoresearch.com/isqe/agenda_m.htm.

[88] Terence Kelly and Jeffrey Mogul. Aliasing on the World Wide Web: Prevalence and performance implications. In *Proceedings of the Eleventh International World Wide Web Conference*, pages 281–292, May 2002. http://ai.eecs.umich.edu/~tpkelly/papers/.

[89] Terence Kelly and Daniel Reeves. Optimal Web cache sizing: Scalable methods for exact solutions. *Computer Communications*, 24:163–173, February 2001. http://ai.eecs.umich.edu/~tpkelly/papers/.

[90] Tracy Kimbrel. Online paging and file caching with expiration times. *Theoretical Computer Science*, 268(1):119–131, October 2001.

[91] Donald E. Knuth. An analysis of optimum caching. *Journal of Algorithms*, 6:181–199, 1985.

[92] Mimika Koletsou and Geoffrey M. Voelker. The Medusa proxy: A tool for exploring user-perceived Web performance. In *Proceedings of the Sixth International Workshop on Web Caching and Content Delivery*, June 2001. http://www.cs.bu.edu/techreports/2001-017-wcw01-proceedings/134_koletsou.pdf.

[93] Dexter C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, 1992. ISBN 0-387-97687-6.

[94] Balachander Krishnamurthy and Martin Arlitt. PRO-COW: Protocol compliance on the Web—a longitudinal study. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems*, pages 109–122, March 2001. http://www.research.att.com/~bala/papers/usits01.ps.gz.

[95] 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*, May 1999. http://www8.org/w8-papers/5c-protocols/key/key.html.

[96] Balachander Krishnamurthy and Jennifer Rexford. *Web Protocols and Practice*. Addison-Wesley, May 2001. ISBN 0-201-71088-9.

[97] David M. Kristol and Lou Montulli. RFC 2109: HTTP state management mechanism, February 1997.

[98] James F. Kurose and Rahul Simha. A microeconomic approach to optimal resource allocation in distributed computer systems. *IEEE Transactions on Computers*, 38(5):705–717, May 1989.

[99] Chat-Yu Lam and Stuart E. Madnick. Propeties of storage hierarchy systems with multiple page sizes and redundant data. *ACM Transactions on Database Systems*, 4(3):345–367, 1979.

[100] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) policies. *Performance Evaluation Review*, 27(1):134–143, 1999.

[101] Jay Logue. Personal communication.

[102] Macromedia. Dreameaver, November 2001. `http://www.macromedia.com/support/dreamweaver/`.

[103] Anirban Mahanti and Carey Williamson. Web proxy workload characterization. Technical report, Department of Computer Science, University of Saskatchewan, February 1999. `http://www.cs.usask.ca/faculty/carey/papers/workloadstudy.ps`.

[104] Stephen Manley and Margo Seltzer. Web facts and fantasy. In *Proceedings of the First USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 125–133, December 1997.

[105] Michigan adaptive resource exchange (MARX) project. `http://ai.eecs.umich.edu/MARX/`.

[106] Andreu Mas-Colell, Michael D. Whinston, and Jerry R. Green. *Microeconomic Theory*. Oxford University Press, 1995. ISBN 0-19-507340-1.

[107] Peter Mattis, John Plevyak, Matthew Haines, Adam Beguelin, Brian Totty, and David Gourley. U.S. Patent #6,292,880: "Alias-free content-indexed object cache", September 2001. `http://patft.uspto.gov/`.

[108] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[109] R. Preston McAfee and John McMillan. Auctions and bidding. *Journal of Economic Literature*, XXV:699–738, June 1987.

[110] Daniel Menascé, Virgílio Almeida, Rodrigo Fonseca, and Marco A. Mendes. Resource management policies for e-commerce servers. In *Proceedings of Second Workshop on Internet Server Performance (WISP99)*, May 1999. `http://www.cc.gatech.edu/fac/Ellen.Zegura/wisp99/papers/menasce.ps`.

[111] Daniel Menascé, Flávia Ribeiro, Virgílio Almeida, Rodrigo Fonseca, Rudolf Reidi, and Wagner Meira Jr. In search of invariants for e-business workloads. In *Proceedings of the Second ACM Conference on Electronic Commerce*, pages 56–65, October 2000. ISBN 1-58113-272-7.

[112] Daniel A. Menascé and Virgílio A. F. Almeida. *Capacity Planning for Web Performance: Metrics, Models, and Methods*. Prentice Hall, 1998. ISBN 0-13-693822-1.

[113] Mikhail Mikhailov and Craig E. Wills. Change and relationship-driven content caching, distribution and assembly. Technical Report WPI-CS-TR-01-03, Worcester Polytechnic Institute, March 2001. `http://www.cs.wpi.edu/~mikhail/papers/tr01-03.pdf`.

[114] David L. Mills. RFC 1305: Network time protocol, March 1992.

[115] Jeffrey C. Mogul. Errors in timestamp-based HTTP header values. Technical Report 99/3, Compaq Western Research Laboratory, December 1999.

[116] Jeffrey C. Mogul. A trace-based analysis of duplicate suppression in HTTP. Technical Report 99/2, Compaq Western Research Laboratory, November 1999.

[117] Jeffrey C. Mogul. Squeezing more bits out of HTTP caches. *IEEE Network*, 14(3):6–14, May/June 2000.

[118] Jeffrey C. Mogul. Clarifying the fundamentals of HTTP. In *Proceedings of the Eleventh International World Wide Web Conference*, May 2002. `http://www2002.org/CDROM/refereed/444.pdf`.

[119] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP (corrected version). Technical Report 97/4a, Digital Western Research Lab, December 1997. `http://research.compaq.com/wrl/techreports/abstracts/97.4.html`.

[120] FAQ of the caching mechanism in [Mozilla] 331 release, April 2000. `http://www.mozilla.org/docs/netlib/cachefaq.html`.

[121] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems. Technical Report 91-3, University of Michigan Center for Information Technology Integration (CITI), August 1991. `http://www.citi.umich.edu/techreports/`.

[122] D. Muntz, P. Honeyman, and C. J. Antonelli. Evaluating delayed write in a multilevel caching file system. Technical Report 95-9, University of Michigan Center for Information Technology Integration (CITI), October 1995. `http://www.citi.umich.edu/techreports/`.

[123] Daniel A. Muntz, Peter Honeyman, and Charles J. Antonelli. Evaluating delayed write in a multilevel caching file system. In *Proceedings of the 1996 IFIP/IEEE International Conference on Distributed Platforms*, pages 415–429, February 1996.

[124] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, pages 174–187, October 2001. `http://www-cse.ucsd.edu/sosp01/papers/mazieres.pdf`.

[125] Roger B. Myerson. Incentive compatibility and the bargaining problem. *Econometrica*, 47:61–73, 1979.

[126] Henrik Nordstrom. Squid cache revalidation and metadata updates. Posting to `squid-dev` mailing list, October 2001. `http://www.squid-cache.org/mail-archive/squid-dev/200110/0054.html`.

[127] National Institute of Standards and Technology. Secure hash standard. FIPS Pub. 180-1, U.S. Department of Commerce, April 1995. `http://csrc.nist.gov/publications/fips/fips180-1/fip180-1.txt`.

[128] Frank Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Electrical Engineering and Computer Science Department, University of California, Berkeley; and Computer Science and Mathematics Department, Lawrence Berkeley Lab, May 1981. This is the author's Berkeley Masters thesis.

[129] Morgan Oslake. Capacity model for Internet transactions. Technical Report MSR-TR-99-18, Microsoft Research, April 1999.

[130] Venkata N. Padmanabhan and Lili Qiu. The content and access dynamics of a busy Web server: Findings and implications. In *Proceedings of ACM SIGCOMM*, pages 111–123, August 2000. Reference [131] is a longer tech report version.

[131] Venkata N. Padmanabhan and Lili Qiu. The content and access dynamics of a busy Web server: Findings and implications. Technical Report MSR-TR-2000-13, Microsoft Research, February 2000. A shorter but more recent version is available as Reference [130].

[132] Andy Palms. Personal communication. Palms is Director of IT Communications at the University of Michigan's Information Technology Division.

[133] R. Pandey, J. Fritz Barnes, and R. Olsson. Supporting Quality of Service in HTTP Servers. In *Proceedings of the Seventeenth Annual SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 247–256, June 1998. `http://arthur.cs.ucdavis.edu/~barnes/Papers.html`.

[134] Web Polygraph. `http://www.web-polygraph.org/`.

[135] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992.

[136] Michael Rabinovich and Oliver Spatscheck. *Web Caching and Replication*. Addison Wesley, December 2001. ISBN 0201615703.

[137] B. Ramakrishna Rau. Properties and applications of the least-recently-used stack model. Technical Report CSL-TR-77-139, Digital Systems Laboratory, Department of Electrical Engineering and Computer Science, Stanford University, May 1977.

[138] Internet RFCs. `ftp://ftp.ietf.org/rfc/`.

[139] Ronald L. Rivest. RFC 1321: The MD5 message-digest algorithm, April 1992.

[140] Luigi Rizzo and Lorenzo Vicisano. Replacement policies for a proxy cache. Technical Report RN/98/13, University College London Department of Computer Science, 1998. `http://www.iet.unipi.it/~luigi/lrv98.ps.gz`.

[141] Alex Rousskov and Valery Soloviev. On performance of caching proxies. Technical report, NCAR, August 1998.

[142] Alex Rousskov and Duane Wessels. The third cache-off: The official report. Technical report, The Measurement Factory, Inc., October 2000. `http://www.measurement-factory.com/results/public/cacheoff/N03/report.by-meas.html`.

[143] Jonathan Santos and David Wetherall. Increasing effective link bandwidth by suppressing replicated data. In *Proceedings of the USENIX Annual Technical Conference*, June 1998. `http://www.usenix.org/publications/library/proceedings/usenix98/full_papers/santos/santos.pdf`.

[144] Narayanan Shivakumar and Hector Garcia-Molina. Finding near-replicas of documents on the Web. In *Proceedings of Workshop on Web Databases (WebDB'98)*, March 1998. `http://www-db.stanford.edu/~shiva/Pubs/web.ps`.

[145] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[146] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[147] Ben Smith, Anurag Acharya, Tao Yang, and Huican Zhu. Exploiting result equivalence in caching dynamic content. In *Proceedings of Second USENIX Symposium on Internet Technologies and Systems*, pages 209–220, October 1999. `http://www.cs.ucsb.edu/Research/swala/usits99/paper.html`.

[148] F. Donelson Smith, Félix Hernández Campos, Kevin Jeffay, and David Ott. What TCP/IP protocol headers can tell us about the Web. In *Proceedings of ACM SIGMETRICS*, pages 245–256, June 2001.

[149] SPECweb. `http://www.spec.org/osg/web99/`.

[150] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, pages 87–95, August 2000.

[151] David Surovell. Personal communication.

[152] I. E. Sutherland. A futures market in computer time. *Communications of the ACM*, 11(6):449–451, June 1968.

[153] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992. ISBN 0-13-588187-0.

[154] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Number 44 in CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1983. ISBN 0-89871-187-8.

[155] James Gordon Thompson. Efficient analysis of caching systems. Technical Report UCB/CSD 87/374, Computer Science Division (EECS), University of California at Berkeley, October 1987. This is the author's Ph.D. dissertation.

[156] Arthur van Hoff, John Giannandrea, Mark Hapner, Steve Carter, and Milo Medin. The HTTP distribution and replication protocol. Technical Report NOTE-DRP, World Wide Web Consortium, August 1997. `http://www.w3.org/TR/NOTE-drp-19970825.html`.

[157] Aad van Moorsel. Metrics for the internet age: Quality of experience and quality of business. Technical Report HPL-2001-179, HP Labs, July 2001. `http://www.hpl.hp.com/techreports/2001/HPL-2001-179.html` and `http://www.informatik.unibw-muenchen.de/PMCCS5/papers/moorsel.pdf`.

[158] Hal Varian and Jeffrey K. MacKie-Mason. Generalized vickrey auctions. Technical report, Dept. of Economics, University of Michigan, July 1994.

[159] Hal R. Varian. Economic mechanism design for computerized agents. In *Proceedings of the First USENIX Conference on Electronic Commerce*, July 1995. `http://www.sims.berkeley.edu/~hal/people/hal/papers.html`.

[160] William Vickrey. Counterspeculation, auctions and competitive sealed tenders. *Journal of Finance*, 16:8–37, 1961.

[161] Theimo Voight, Renu Tewari, Douglas Freimuth, and Anish Mehara. Kernel mechanisms for service differentiation in overloaded Web servers. In *Proceedings of the USENIX Annual Technical Conference*, pages 189–202, June 2001.

[162] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffery O. Kephart, and W. Scott Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.

[163] Michael P. Wellman. Market-oriented programming: Some early lessons. In S. Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, 1996. `http://ai.eecs.umich.edu/people/wellman/Publications.html`.

[164] Duane Wessels. Personal communication.

[165] Duane Wessels. *Web Caching*. O'Reilly, June 2001. ISBN 1-56592-536-X.

[166] 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, 1996.

[167] Carey Williamson. On filter effects in Web caching hierarchies. *ACM Transactions on Internet Technology*, 2(1):47–77, February 2002.

[168] Craig E. Wills and Mikhail Mikhailov. Examining the cacheability of user-requested Web resources. In *Proceedings of the Fourth International Web Caching Workshop*, April 1999. `http://www.cs.wpi.edu/~mikhail/papers/wcw99.ps.gz`.

[169] Craig E. Wills and Mikhail Mikhailov. Towards a better understanding of Web resources and server responses for improved caching. In *Proceedings of the Eighth International World Wide Web Conference*, May 1999. `http://www.cs.wpi.edu/~mikhail/papers/www8.ps.gz`.

[170] 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*, May 2000. `http://www.cs.wpi.edu/~mikhail/papers/wcw5.ps.gz`.

[171] Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Molly Brown, Tashana Landray, Denise Pinnel, Anna Karlin, and Henry Levy. Organization-based analysis of Web-object sharing and caching. In *Proceedings of the Second USENIX Conference on Internet Technologies and Systems*, October 1999. `http://www.cs.washington.edu/homes/wolman/`.

[172] 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. Originally in 17th ACM Symposium on Operating Systems Principles (SOSP '99). `http://www.cs.washington.edu/homes/wolman/`.

[173] Roland P. Wooster and Marc Abrams. Proxy caching that estimates page load delays. In *Proceedings of WWW6*, pages 325–334, April 1997. Also appeared in *Computer Networks and ISDN Systems* 29, 1997, 1497–1505. `http://vtopus.cs.vt.edu/~chitra/docs/www6r/`.

[174] 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*, March 1999.

[175] Xiaohui Zhang. Cachability of Web objects. Technical Report 2000-019, Boston University Computer Science Department, August 2000.

[176] Yuanyuan Zhou, James F. Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the USENIX Annual Technical Conference*, June 2001.