

2002

Adaptive and secured resource management in distributed and Internet systems

Li Xiao

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Xiao, Li, "Adaptive and secured resource management in distributed and Internet systems" (2002).
Dissertations, Theses, and Masters Projects. Paper 1539623406.
<https://dx.doi.org/doi:10.21220/s2-deqc-ew25>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Adaptive and Secured Resource Management in Distributed and Internet Systems

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William & Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

by

Li Xiao

2002

APPROVAL SHEET

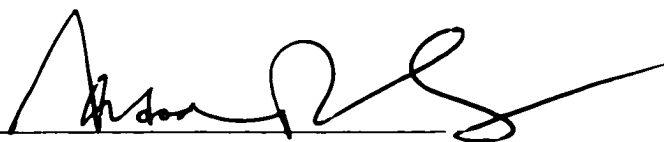
This dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

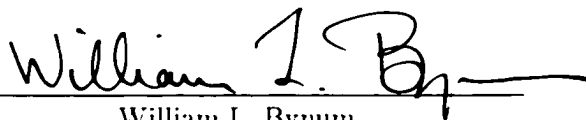


Li Xiao

Approved. July 2002



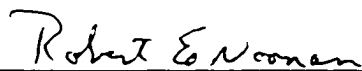
Xiaodong Zhang
Thesis Advisor



William L. Bynum



Phil Kearns



Robert E. Noonan



Marc Sher
Department of Physics

To my parents . . .

Table of Contents

Acknowledgments	xi
List of Tables	xiv
List of Figures	xxii
Abstract	xxiii
1 Introduction	2
1.1 Background	2
1.2 Problems	4
1.3 Statements of Contributions	5
1.4 Organization of the Dissertation	7
2 Application level resource management of memory systems	8
2.1 Literature overview on memory utilization in centralized servers	8
2.2 Improving Memory Performance of Sorting Algorithms	10
2.3 Architectural and Algorithmic Parameters and Evaluation Methodology . .	13
2.3.1 Architectural and algorithmic parameters	13

2.3.2	Performance evaluation methodology	13
2.3.3	Data sets	15
2.4	Cache-Effective Mergesort Algorithms	16
2.4.1	Tiled mergesort and multimergesort	16
2.4.2	New mergesort alternatives	17
2.4.2.1	Tiled mergesort with padding	17
2.4.2.2	Multimergesort with TLB padding	21
2.4.3	Trade-offs relating to an instruction count increase and the performance gain	24
2.5	Cache-Effective Quicksort	25
2.5.1	Memory-tuned quicksort and multiquicksort	26
2.5.2	New quicksort alternatives	26
2.5.2.1	Flash Quicksort	27
2.5.2.2	Inplaced flash quicksort	27
2.5.3	Simulation results	28
2.6	Measurement Results and Performance Evaluation	29
2.6.1	Mergesort performance comparisons	31
2.6.2	Quicksort performance comparisons	34
2.7	A Prediction Model of Performance Trade-Offs	36
2.8	Chapter Conclusion	41
3	Load Sharing for Global Memory System Management	43
3.1	Literature overview on load sharing for global memory in distributed systems	43

3.2	CPU-memory-based Load Sharing	47
3.2.1	CPU-Memory-Based Load Sharing Policies	49
3.2.2	Performance Evaluation Methodology	53
3.2.2.1	A simulated cluster	54
3.2.2.2	Workload Traces	55
3.2.2.3	System conditions	56
3.2.3	Performance Results and Analysis	57
3.2.3.1	Overall Performance Comparisons	57
3.2.3.2	Paging and Queuing	59
3.2.3.3	High Performance and High Throughput	61
3.2.4	Summary	62
3.2.5	Brief description of our study on heterogeneous systems	62
3.2.5.1	CPU/Memory Weights and Heterogeneity	63
3.2.5.2	Summary of Our Heterogeneous Study	65
3.3	Incorporation job migration and network RAM to share memory resource .	66
3.3.1	Objectives of the study	66
3.3.2	Job-migration-based load sharing vs. network RAM	68
3.3.2.1	Network RAM organizations	68
3.3.2.2	CPU-Memory-based load sharing	69
3.3.3	Performance Evaluation Methodology	70
3.3.3.1	Performance metrics	70
3.3.3.2	A simulated workstation cluster	71
3.3.3.3	Workloads	72

3.3.4	Simulation Results and Analysis	73
3.3.4.1	Impact of limited network bandwidths	73
3.3.4.2	Trade-offs between page fault reductions and load sharing	77
3.3.5	An improved load sharing scheme	79
3.3.6	Summary	80
4	Resource Management in Internet Caching Systems	84
4.1	Overview of existing caching system structures	84
4.2	Changes in Both Workload and Internet Technologies	86
4.2.1	Workload Changes	86
4.2.1.1	Trend in NLANR Workload	86
4.2.1.2	Trend in BU Workload	88
4.2.2	Technology Changes	91
4.3	Overview of the Limits on Existing Caching System Structures	92
5	Locality and Information Sharing among Browsers	95
5.1	Browsers-Aware Proxy Server	96
5.2	Simulation Environment	97
5.2.1	Traces	98
5.2.2	A browsers-proxy caching environment	100
5.3	Performance Evaluation	102
5.3.1	Sizes of browser and proxy caches	102
5.3.2	How much is browser cache data sharable?	103
5.3.3	Performance of browsers-aware proxy server	107

5.3.4	Performance Impact of Scaling the Number of Clients	108
5.4	Overhead Estimation	110
5.5	Chapter Conclusion	113
6	Reducing Duplications in a Proxy and Its Browsers	116
6.1	Background and Related Work	116
6.2	A simulated proxy-browser caching environment	120
6.3	Case Studies of Duplications in Web Caching	120
6.4	Cooperatively Shared Proxy-Browser Caching Scheme	122
6.4.1	An outline of the scheme	122
6.4.2	Data structures and operations	124
6.4.2.1	The structure in each browser	124
6.4.2.2	The structure in the proxy	125
6.4.3	Offline Algorithms for Performance Comparisons	127
6.5	Performance Evaluation	129
6.5.1	Evaluation of the sensitivity to the proxy cache size	130
6.5.2	Evaluation of the sensitivity to a browser cache size	132
6.5.3	Evaluation of the sensitivity to the replacement threshold	134
6.5.4	Performance Impact of Scaling the Number of Clients	135
6.5.5	Latency Reduction	137
6.6	Overhead Analysis	138
6.6.1	Intra-network Overhead	138
6.6.2	Space Overhead	141

6.6.3	CPU Overhead	143
6.7	Chapter Conclusion	144
7	Data Integrity and Communication Anonymity	145
7.1	Introduction	145
7.2	Data Integrity	146
7.3	Anonymity Issue	147
7.4	Related Work on Anonymity Studies	149
7.4.1	Publisher and Sender Anonymity	149
7.4.2	Existing mutual anonymity protocols: their merits and limits	150
7.5	Anonymity with Trusted Third Parties	153
7.5.1	A Mix-based Protocol: an intuitive solution	154
7.5.2	Center-Directing	157
7.5.3	Label-Switching	160
7.5.4	Multiple Trusted Index Servers	162
7.6	Anonymity in Pure P2P	165
7.7	Analysis	168
7.7.1	Security Analysis	168
7.7.2	Cost of the Different Protocols	171
7.8	Performance Evaluation	171
7.8.1	Data Transfer Time through Peer Nodes	172
7.8.2	Overhead of MD5, DES and RSA	173
7.8.3	Additional Storage	174

7.8.4	Comparisons of Protocols	174
7.9	Discussion	176
7.10	Chapter Conclusion	180
8	Prototype Implementations and Initial Results	182
8.1	A client daemon to interface the browser and communicate with the proxy .	182
8.2	A browsers-aware proxy server	184
8.3	Overhead Measurement and Analysis	185
9	Final Conclusions and Future Work	188
9.1	Summary	188
9.2	Future Work: Balancing the Power between Centralized and Decentralized	
	Resources in Distributed Computing	191
9.2.1	Non-uniform parallel computing	192
9.2.2	Resource Indexing on Grid Systems	193
9.2.3	Resource Management on Peer-to-peer systems	194
	Bibliography	196

ACKNOWLEDGMENTS

My first acknowledgment goes to my advisor, Xiaodong Zhang. He provides me with an excellent research environment both materially and spiritually: by deeply involving in guiding my research, by continuously securing funding, by persistently setting a high standard for quality, by actively promoting a discussion atmosphere and encouraging collaborations, and insightfully sharing his visions of research. He is my role model of hardworking and devotion to his career and to his students. He is always with his students at every moment of facing difficulties. I also appreciate deeply for his help on building my confidence, and for his efforts on pursuing every opportunity for his students in their career development. I am grateful to his diligent efforts on preparing and training us to face and enjoy real-world challenges. I am very fortunate to have him as my advisor.

I would like to acknowledge Professors Bill Bynum, Phil Kearns, and Bob Noonan for serving the dissertation committee, and Marc Sher for serving as the external member of the committee and making helpful comments. I thank Bill for reviewing many of my manuscripts, and for his help and encouragement in my study. Evgenia Smirni has made helpful suggestions to me for both technical study and career development. My dissertation is experimental-oriented research, and has been dependent on strong technical support from Phil and his techie team. I would also like to thank Vanessa Godwin for her organized management and help in my graduate study.

I have spent 4 years with many fellow students in the High Performance Computing and Software Lab (HPCS), and enjoy working, discussing, and joking with them: Songqing Chen, Xin Chen, Lei Guo, Song Jiang, Stefan Kubricht, Yanxia Qu, Zhao Zhang, and Zhichun Zhu. I have productively collaborated with Songqing, Stephan, and Yanxia on several research projects, and learned a lot from them. I enjoy my friendships with many other fellow graduate students: Wei Ding, Andrew LaRoy, Shanling Peng, Alma Riska, and

Wei Sun.

My work as a research intern at the Hewlett Packard Laboratories in the summer of 2001 gave me a valuable experience in the graduate study. I would like to thank Dr. Zhichen Xu, my mentor of the internship and a former member of the HPCS lab, for his advice and collaboration. My thanks also goes to Artur Andrzejak from HP Labs for his help and collaboration. I was also beneficial to many discussions with Dr. Yong Yan, a researcher at HP Labs and another former member of the HPCS lab. I had opportunities to work and discuss with several people when I was at the HP Labs: Martin Arlitt, Lucy Cherkasova, Yun Fu, Minaxi Gupta, Vana kalogeraki, Malena Mesarina, Manohar Prabhu, and Wenting Tang. I also thank Beveley Yang for her invitation to attend their group discussions in Stanford University.

I want to thank the funding agencies that provided funds and equipment for my research: Air Force Office of Scientific Research, National Science Foundation, and Sun Microsystems. I would like to give a special acknowledgment to the Fellowship awarded by the USENIX Association. I am honored to be a USENIX Scholar.

My gratitude to my parents and my two brothers is forever. My parents endured many hardships in the difficult time of China. But they were always optimistic to the future, unselfishly protective to us, and pleasantly tie the family together. I am deeply grateful to my parents' high expectations and their constant helps on developing my all-rounded abilities and personality. With their unconditional love, my family always understands my decisions, and devotes whatever they could to support my pursuits.

List of Tables

2.1	Architectural parameters of the 4 machines we have used for the experiments.	30
3.1	Trace Description	56
3.2	Summary of the 4 schemes and their impact on different system and workload conditions/requirements.	83
4.1	Average Hit ratio and coverage comparisons of year 1998 and 2000, where the average hit ratio is calculated from proxy “pb”, “bo1”, “bo2”, “sv” and “sd”, which have their statistical reports in both years, and the coverage of top 20 servers is the percentage of the number of requests to top 20 servers over the total number of requests.	87
5.1	Selected Web Traces.	100
5.2	Representative proxy cache configurations reported in [105].	103
6.1	Trace analysis on document duplications and sharing based on the proxy-browser system hit ratios, intra-sharing ratios, and inter-sharing ratios. . . .	121
6.2	Intra-network Overhead	141

7.1	Path Table	160
7.2	Sub-Tables	161
7.3	Degree of Anonymity	170
7.4	Comparison of Protocols with k middle nodes in each covert path	171
7.5	Latency	173

List of Figures

2.1	Data layout of subarrays is modified by padding to reduce the conflict misses.	20
2.2	Simulation comparisons of the L1 cache misses (left figure) and L2 misses (right figure) of the mergesort algorithms on the <i>Random</i> data set on the simulated Sun Ultra 5. The L1 cache miss curves (left figure) of the base mergesort and tiled-mergesort are overlapped.	21
2.3	Padding for TLB: the data layout is modified by inserting a page space at multiple locations, where $K_{TLB} = 1$, and $T_s = 8$	23
2.4	Simulation comparisons of the L2 cache misses (left figure) and TLB misses (right figure) of the mergesort algorithms on the <i>Random</i> data set on the simulated Pentium II.	24
2.5	Simulation comparisons of the instruction counts (left figure) and saved cycles in percentage (right figure) of the mergesort algorithms on the <i>Random</i> data set on the simulated Pentium II. The instruction count curves (left figure) of the base mergesort and the tiled mergesort are overlapped.	25

2.6	Simulation comparisons of the instruction counts (left figure) and the L1 misses (right figure) of the quicksort algorithms on the <i>Unbalanced</i> data set on the simulated Pentium III. The instruction count curve of the flashsort was too high to be presented in the left figure.	28
2.7	Execution comparisons of the mergesort algorithms on SGI O2 and on Sun Ultra 5.	31
2.8	Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Equilikely data set (left figure) and the Bernoulli data set (right figure). . .	32
2.9	Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Geometric data set (left figure) and the Pascal data set (right figure). . . .	33
2.10	Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Binomial data set (left figure) and the Poisson data set (right figure). . . .	34
2.11	Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Unbalanced data set (left figure) and the Zero data set (right figure).	35
2.12	Execution comparisons of the mergesort algorithms on Pentium II and on Pentium III.	36
2.13	Execution comparisons of the quicksort algorithms on the <i>Random</i> data set (left figure) and on the <i>Unbalanced</i> data set (right figure) on the SGI O2. (The timing curve of the flashsort is too high to be presented in the right figure).	37

2.14	Execution comparisons of the quicksort algorithms on the <i>Random</i> data set (left figure) and on the <i>Unbalanced</i> data set (right figure) on the Ultra 5. (The timing curve of the flashsort is too high to be presented in the right figure).	38
2.15	Execution comparisons of the quicksort algorithms on the <i>Random</i> data set (left figure) and on the <i>Unbalanced</i> data set on the Pentium II. (The timing curve of the flashsort is too high to be presented in the right figure).	39
2.16	Execution comparisons of the quicksort algorithms on the <i>Random</i> data set (left figure) and on the <i>Unbalanced</i> data set on the Pentium III. (The timing curve of the flashsort is too high to be presented in the right figure).	40
3.1	Mean slowdowns of the 4 load sharing policies as the page fault rate increases on traces MAY and JUNE.	58
3.2	Mean slowdowns of the 4 load sharing policies as the page fault rate increases on traces JULY and AUGUST.	59
3.3	Paging time reduction (left figure) and queuing time reduction (right figure) of policies MEM, CPU_MEM_HP and CPU_MEM_HT over policy CPU.	60
3.4	The average execution times per job (the left figure) and the networking portions in the execution times (right figure) of "trace 0" with job migration restrictions running on clusters of 6, 12 and 18 workstations.	75
3.5	The average execution times per job (the left figure) and the networking portions in the execution times (right figure) of "trace 0" without job migration restrictions running on clusters of 6, 12 and 18 workstations.	76

3.6	The average execution times per job of “trace 0” without job migration restrictions running on a 10 Mbps cluster (the left figure), and a 100 Mbps cluster (the right figure) of 6 workstations.	78
3.7	The average execution times per job of “trace 0” without job migration restrictions running on a 10 Mbps cluster (the left figure), and a 100 Mbps cluster (the right figure) of 12 workstations.	79
3.8	The average execution times per job of all the 8 traces (the left figure for the 8 traces where some jobs are non-migratable, and the right figure for the 8 traces where all the jobs are migratable), running on a 10 Mbps cluster of 6 workstations.	81
3.9	The average execution times per job of all the 8 traces (the left figure for the 8 traces where some jobs are non-migratable, and the right figure for the 8 traces where all the jobs are migratable), running on a 100 Mbps cluster of 6 workstations.	82
4.1	The percentage of the requests to each of the top 20 servers over the total number of requests versus each rank of servers.	87
4.2	The percentage of the requests to each server or document over the total requests versus server ranking or document ranking.	89
5.1	Organizations of the browsers-aware proxy server.	98
5.2	The hit ratios and byte hit ratios of the five caching policies using NLNR-uc trace, where the browser cache size is set <i>minimum</i>	104

5.3	The breakdowns of the hit ratios and byte hit ratios of the browsers-aware proxy using NLANR-uc trace, where the browser cache size is set <i>minimum</i> .	106
5.4	The hit rates and byte hit rates of the <i>browsers-aware-proxy-server</i> and <i>proxy-and-local-browser</i> scheme using NLANR-uc trace, where the browser cache size is set <i>average</i> .	108
5.5	The hit rates and byte hit rates of the <i>browsers-aware-proxy-server</i> and <i>proxy-and-local-browser</i> scheme using NLANR-bo1 trace, where the browser cache size is set <i>average</i> .	109
5.6	The hit rates and byte hit rates of the <i>browsers-aware-proxy-server</i> and <i>proxy-and-local-browser</i> scheme using NLANR-pa trace, where the browser cache size is set <i>average</i> .	110
5.7	The hit rates and byte hit rates of the <i>browsers-aware-proxy-server</i> and <i>proxy-and-local-browser</i> scheme using Boeing-4 trace, where the browser cache size is set <i>average</i> .	111
5.8	The hit rates and byte hit rates of the <i>browsers-aware-proxy-server</i> and <i>proxy-and-local-browser</i> scheme using Boeing-5 trace, where the browser cache size is set <i>average</i> .	112
5.9	The hit rates and byte hit rates of the <i>browsers-aware-proxy-server</i> and the <i>proxy-and-local-browser</i> scheme using the BU-95 trace, where the browser cache size is set <i>average</i> .	113
5.10	The hit rates and byte hit rates of the <i>browsers-aware-proxy-server</i> and the <i>proxy-and-local-browser</i> scheme using the BU-98 trace, where the browser cache size is set <i>average</i> .	114

5.11	The hit ratios and byte hit ratios of the <i>browsers-aware-proxy-server</i> and <i>proxy-and-local-browser</i> scheme using the CA*netII trace.	115
5.12	The hit ratio and byte hit ratio increments of the <i>browsers-aware-proxy-server</i> over the <i>proxy-and-local-browser</i>	115
6.1	Duplication among a proxy and its client browsers.	117
6.2	The management operations in each browser when a remote client request hits in it.	125
6.3	The Management operations in the proxy when a client request hits in the proxy.	126
6.4	Hit ratio and byte hit ratio of the three caching schemes versus relative proxy cache sizes using BU-95 trace ($\beta=10$, $th=0.5$).	130
6.5	Hit ratio and byte hit ratio of the three caching schemes versus relative proxy cache sizes using BU-98 trace ($\beta=10$, $th=0.5$).	131
6.6	Hit ratio and byte hit ratio of the three caching schemes versus relative proxy cache sizes using Boeing-4 trace ($\beta=10$, $th=0.5$).	132
6.7	Hit ratio and byte hit ratio of the three caching schemes versus relative proxy cache sizes using Boeing-5 trace ($\beta=10$, $th=0.5$).	133
6.8	Hit ratio and byte hit ratio of the three caching schemes versus relative browser cache sizes using BU-95 trace ($ps=1\%$, $th=0.5$).	134
6.9	Hit ratio and byte hit ratio of the three caching schemes versus relative browser cache sizes using BU-98 trace ($ps=1\%$, $th=0.5$).	135

6.10	Hit ratio and byte hit ratio of the three caching schemes versus relative browser cache sizes using Boeing-4 trace ($ps=1\%$, $th=0.5$).	136
6.11	Hit ratio and byte hit ratio of the three caching schemes versus relative browser cache sizes using Boeing-5 trace ($ps=1\%$, $th=0.5$).	137
6.12	Hit ratio and byte hit ratio of the three caching schemes versus the replacement threshold using BU-95 trace ($ps=1\%$, $\beta=10$).	138
6.13	Hit ratio and byte hit ratio of the three caching schemes versus the replacement threshold using BU-98 trace ($ps=1\%$, $\beta=10$).	139
6.14	Hit ratio and byte hit ratio of the three caching schemes versus the replacement threshold using Boeing-4 trace ($ps=1\%$, $\beta=10$).	140
6.15	Hit ratio and byte hit ratio of the three caching schemes versus the replacement threshold using Boeing-5 trace ($ps=1\%$, $\beta=10$).	141
6.16	The hit ratio and byte hit ratio increments of the <i>cooperative-caching</i> over the <i>proxy-and-local-browser</i> .	142
7.1	Integrity Protocol	147
7.2	An example of the Mix-Based Protocol	156
7.3	An example of the Center-Directing Protocol	159
7.4	An example of the shortcut-responding Protocol	167
7.5	Breakdown of data transfer and protocol overhead with 2 and 5 middle nodes for Boeing March 4 trace (left) and Boeing March 5 Trace (right). $MB(k)$ represent mix-based protocol with k middle nodes. Similarly, CD, LS and SR represent center-directing, label-switching, shortcut-responding, respectively.	175

8.1	The organization of client daemon to interface with a client browser and the proxy.	184
8.2	The organization of proxy daemon to interface with a client browser and the proxy.	186

ABSTRACT

The effectiveness of computer system resource management has been always determined by two major factors: (1) workload demands and management objectives, (2) the updates of the computer technology. These two factors are dynamically changing, and resource management systems must be timely adaptive to the changes. This dissertation attempts to address several important and related resource management issues.

We first study memory system utilization in centralized servers by improving memory performance of sorting algorithms, which provides fundamental understanding on memory system organizations and its performance optimizations for data-intensive workloads. To reduce different types of cache misses, we restructure the mergesort and quicksort algorithms by integrating tiling, padding, and buffering techniques and by repartitioning the data set. Our study shows substantial performance improvements from our new methods.

We have further extended the work to improve load sharing for utilizing global memory resources in distributed systems. Aiming at reducing the memory resource contention caused by page faults and I/O activities, we have developed and examined load sharing policies by considering effective usage of global memory in addition to CPU load balancing in both homogeneous and heterogeneous clusters.

Extending our research from clusters to Internet systems, we have further investigated memory and storage utilizations in Web caching systems. We have proposed several novel management schemes to restructure and decentralize the existing caching system by exploiting data locality at different levels of the global memory hierarchy and by effectively sharing data objects among the clients and their proxy caches.

Data integrity and communication anonymity issues are raised from our decentralized Web caching system design, which are also security concerns for general peer-to-peer systems. We propose an integrity protocol to ensure data integrity, and several protocols to achieve mutual communication anonymity between an information requester and a provider.

The potential impact and contributions of this dissertation are briefly stated as follows: (1) two major research topics identified in this dissertation are fundamentally important for the growth and development of information technology, and will continue to be demanding topics for a long term. (2) Our proposed cache-effective sorting methods bridge a serious gap between analytical complexity of algorithms and their execution complexity in practice due to the increasingly deep memory hierarchy in computer systems. This approach can also be used to improve memory performance at different levels of the memory hierarchy, such as I/O and file systems. (3) Our load sharing principle of giving a high priority to the requests of data accesses in memory and I/Os timely adapts the technology changes and effectively responds to the increasing demand of data-intensive applications. (4) Our proposed decentralized Web caching framework and its resource management schemes present a comprehensive case study to examine the P2P model. Our results and experiences can be used for related and further studies in distributed computing. (5) The proposed data integrity and communication anonymity protocols address limits and weaknesses of existing ones, and place a solid foundation for us to continue our work in this important area.

Adaptive and Secured Resource Management in Distributed and Internet Systems

Chapter 1

Introduction

1.1 Background

System resource management has been seriously considered since the computer was born. The original objective of resource management is to make good use of computer resources for high performance. Today the objective has been extended beyond performance, to security, availability and reliability. An effective resource management must be adaptive to the changes of workload and technology. For example, resource management between cache and memory became an essential topic after the cache was installed. A cache could be useless if its locality is not exploited. Resource management in a distributed system came hand in hand with networking systems, and it differs from that in a centralized system in a fundamental way [122]. Resource management on the Internet has become another focused research topic. The security issue is becoming a major concern as global distributed resource sharing dramatically increases.

The two major themes of this dissertation are (1) to make resource allocation be adaptive to the changes of workload and technology, and (2) to make resource sharing secure and protected.

Resource management should target the major system components affecting system performance. Fundamental resources in a modern computer system are CPU cycles, memory, input/output, network interface, and Internet bandwidth. Workloads are becoming increasingly data-intensive, while the speed gap between processing and data accessing continues to widen as the development of memory and disk lags farther behind that of CPU [42]. The speed of microprocessors has been steadily improving at a rate of between 50% and 100% every year, over the last decade. Unfortunately, the memory speed has not kept pace with this, improving only at the rate of about 10% per year during the same period [67]. Thus, the memory hierarchy in both servers and distributed/Internet systems becomes a computing crucial resource. The first focused dissertation topic is to provide new solutions to effectively utilize the memory hierarchy in computing servers and distributed and Internet systems for data-intensive application workloads.

Resource sharing of both computing and information over the Internet is dramatically increasing. This system decentralization trend challenges the existing client/server model, and leads to a new distributed computing paradigm, the peer-to-peer (P2P) model. In a P2P system, a client is no longer a pure consumer but also an information producer or disseminator. This can solve some of problems caused by the client/server model, especially, hot spots surrounding big servers and underutilization of resources [62]. Examples of P2P systems include Napster [90], Freenet [52], Gnutella [60]. So P2P can offer an attractive alternative to the traditional client/server model, and can better utilize networked resources. P2P, however, also creates several challenges, including how to implement distributed controls and how to enforce trust and anonymity. In a P2P system, a peer expects the information that she receives is genuine. A peer may desire to remain anonymous with

respect to the content she possesses or requests. A peer has many reasons to remain anonymous: to keep her life away from danger, to avoid being prejudged by other people, to hide sensitive information from competitors, or simply to keep her privacy [94]. For example, with cooperative proxy caches, the information held in the proxy cache of an organization can be a trade secret. Leaking this information could compromise its competitive advantage. The second focused topic in this dissertation is to propose effective methods to enforce data integrity and communication anonymity for P2P resource sharing.

1.2 Problems

We have identified four related problems of resource management in distributed and Internet systems:

- Memory system utilization in a centralized server. Efforts have been made at the level of program and algorithm optimization. This is a preliminary work to set up a foundation for memory hierarchy management of distributed and Internet systems. Our case study is to improve memory performance of sorting algorithms.
- Load sharing for global memory utilization in distributed systems. Efforts have been made at the level of middleware/system scheduler to migrate jobs. Our case studies are (1) resource allocation for jobs with known and unknown memory demands, and (2) incorporating network RAM and job migrations.
- Data management of Internet caching systems. Efforts have been made at the application level on Web caching. We propose a P2P framework for browser-aware caching to effectively share browser caches. One algorithm aimed at reducing unnecessary

document duplications in the caching storage hierarchy is also proposed. In order to safely and reliably share browser caches, the issues of data integrity and communication anonymity must be addressed. This can be extended as a general problem for P2P systems, which is the last identified problem in this dissertation.

- Data integrity and communication anonymity for P2P Internet systems. Efforts have been made to ensure the data integrity of multiple data copies among the peer nodes, and to provide privacy protection for each peer by enforcing anonymous communications in different types of P2P systems.

1.3 Statements of Contributions

Contributions and potential impact of this dissertation are stated as follows:

- The two major research topics identified in this dissertation, the memory-centric resource management and security/privacy protection in distributed and Internet systems are fundamentally important to the growth and development of information technology, and will continue to be demanding topics for a long term.
- We propose a group of cache-effective software techniques to improve sorting algorithms, which experimentally outperform existing sorting algorithms with cache optimizations. This experimental approach bridges a serious gap between analytical complexity of algorithms and their execution complexity in practice due to the increasingly deep memory hierarchy in computer systems. This approach can also be used to improve memory performance at different level of the memory hierarchy, such as I/O and file systems.

- The traditional principle of “load balancing” in resource management of distributed systems had been highly effective before the memory hierarchy became a performance bottleneck in computer systems. We propose new load sharing policies by not only taking memory and I/O into considerations but also giving high priority to the requests of data accesses. Our resource allocation principle timely adapts the technology changes and effectively responds the increasing demand of data-intensive applications. Our resource management policies for known and unknown memory allocations can be widely applicable, and implemented as a user software or a middleware for high performance cluster computing.
- Computing and information sharing has inevitably and globally decentralized with the rapid advancement of Internet infrastructure. We believe that the P2P model will soon become a standard paradigm co-existing with the client/server model in distributed systems. We present a comprehensive case study to examine the P2P model by proposing the browser-aware Web caching framework and its resource management schemes. Our results and experiences can be used for related studies in distributed computing.
- For a highly decentralized system, the issue of security and privacy has become crucially important. The research and IT community has started to pay more serious attention to this issue since the Tragedy of September-11. The last effort we have made in this dissertation on data integrity and communication anonymity targets on this important issue in P2P systems. The algorithms and protocols we have proposed address limits and weaknesses of existing ones, and place a solid foundation for us to

further improve Internet security.

1.4 Organization of the Dissertation

Chapter 2 presents our study on memory system utilization in centralized servers by improving memory performance of sorting algorithms. Chapter 3 presents our studies on load sharing for global memory utilization in distributed systems. Chapter 4 raises resource management issues in the Internet caching system. The existing caching system structures are first overviewed. This chapter presents our motivation and rational to study on effectively sharing browser caches using the peer-to-peer model. In Chapter 5, we propose a peer-to-peer Web document sharing technique, called *Browsers-Aware Proxy Server* that makes the browsers and their proxy share the contents to fully utilize the Web contents and network bandwidth among clients. In order to further improve performance, a peer-to-peer Web caching management scheme, called *cooperatively shared proxy-browser caching* is proposed in Chapter 6 to reduce document duplications among a proxy and its client browsers. Chapter 7 addresses two problems of data integrity and communication anonymity in browser-aware systems and general peer-to-peer systems. Chapter 8 presents a prototype implementation of the P2P browser sharing system and initial measurement results. Chapter 9 concludes the dissertation and presents future work.

Chapter 2

Application level resource management of memory systems

2.1 Literature overview on memory utilization in centralized servers

Caches could help to bridge the speed gap between fast CPU and slow memory, but scientific applications typically exhibit poor performance in cache-based systems [46]. The reason is that scientific programs run on large data sets and have traversal patterns that may not exploit data locality. Intensive studies have been done in 90's to effectively exploit the benefits of caches at four different levels.

- **Hardware Techniques:**

Examples of hardware techniques to improve cache performance include set-associate caches [73], pseudo-associative caches [3][17], group-associative caches [97], victim caches [72], and multi-column caches [147]. These techniques aim at reducing conflict cache misses for general purpose application programs.

- **Compiler-time Techniques:**

Compiler transformations have been developed to restructure the computation sequence and to improve data locality [18][21][85][103]. Loop interchange, loop reversal, and loop skewing perform loop restructuring to improve data locality [130]. Loop tiling reduces capacity misses by enhancing data locality [30][77][23]. Tile size is an essential factor affecting performance. A compiler should be able to select a right tile size for a given problem and a given cache size, because improperly selected tiling can introduce misses due to cache mapping conflicts.

- **Run-time Techniques:**

Run-time techniques are also effective in reducing cache misses, especially for dynamic applications (see e.g. [9], [71], [140]). A combination of compiler and run-time support for a class of run-time data reordering techniques is studied in [32], where an access sequence is examined and used to reorder data to improve spatial locality as the access sequence is traversed.

- **Programming-level Techniques:**

There have been many studies and implementations at the programming level to improve cache performance of application programs. Many such implementations have been done in a format of scientific libraries. The PhiPAC project [11] aims at producing highly tuned code for specific BLAS 3 [34] kernels such as matrix multiplications that are tiled for multiple levels of the memory hierarchy. An implementation of recursive matrix multiplication is provided by [53]. Paper [65] discusses the role of recursive control strategies in automatic variable blocking of dense linear algebra codes.

and shows dramatic performance gains compared to implementations of the same routines in IBM's Engineering and Scientific Subroutine Library (ESSL). Authors in [24] explore nonlinear array layout functions to improve reference locality, and show high performance benefits on a benchmark suite with dense matrix kernels. Papers [57] and [146] provide cache-optimal methods for bit-reversals. Paper [87] evaluates the impact of data and computation reordering using space-filling curves, and introduces a multi-level blocking technique as a new computation reordering strategy for irregular applications.

The first three techniques provide automatic services to users. But generally one specific technique can only benefit several classes of applications and may not be beneficial to performance of some applications. For example, an improperly selected tile size can degrade performance of applications and the technique proposed in [32] has the side effect of improving TLB performance. In contrast, techniques at the programming design level using application-specific knowledge of the data structures can be highly effective, and are expected to outperform optimizations using the first three system methods. This chapter presents our work on improving memory performance of sorting algorithms at the programming design level. This work was in collaboration with Stefan Kubricht.

2.2 Improving Memory Performance of Sorting Algorithms

Sorting operations are fundamental and are often repeatedly used in many large-scale scientific and commercial applications. Because of this prominence, any effort to maximize the efficiency in these programs requires ensuring that the sorting algorithms used have been

correctly selected and are precisely implemented. Restructuring standard efficient sorting algorithms (such as mergesort and quicksort) to exploit cache locality has proven to be an effective approach for improving performance on high-end systems. Since sorting algorithms are highly sensitive to both the memory hierarchy of the computer architecture and the types of data sets, care must be taken when choosing an algorithm to fully optimize the performance for a specific sorting operation. Existing restructured algorithms (e.g., [78]) mainly attempt to reduce capacity misses on direct-mapped caches. In this chapter, we present several restructured mergesort and quicksort algorithms that exhibit substantial performance improvements by further increasing the locality of the memory references to reduce other types of cache misses, such as conflict misses and TLB misses. These new algorithms utilize both tiling and padding techniques, data set repartitioning, and knowledge of the processor hardware (such as cache and TLB associativity) to fully optimize the performance. Thus, in order to maximize efficiency, it is necessary to implement the cache-effective algorithms carefully and precisely at the algorithm design and programming levels.

Our efforts focus chiefly on restructuring mergesort and quicksort algorithms to more effectively utilize the cache. Our results and contributions are summarized below:

- By applying padding techniques we are able to reduce significantly cache conflict misses and TLB misses, which are not fully addressed in the algorithm designs of tiled mergesort and multimergesort [78]. For our two mergesort alternatives, the optimizations improve both cache and overall performance. Our experiments on different high-end workstations show that our algorithms achieve up to a 70% reduction in

execution time compared with the base mergesort, and up to a 54% reduction versus the fastest of the tiled and multimergesort algorithms.

- By partitioning the data set based on data ranges, we are able to improve the cache locality of quicksort on unbalanced data sets. Our two quicksort alternatives significantly outperform the memory-tuned quicksort [78] and flashsort [92] on unbalanced data sets.
- Cache-effective sorting algorithm design depends on the computer architecture as well as the type of data set. The algorithm design should include parameters such as the size and associativity of both the data cache and TLB, the ratio between the data set size and the cache size, and possibly other factors. Using our measurements and simulations, we show the importance of considering these factors by demonstrating how machines interact differently with the various algorithms.
- A major issue that must be considered when designing a sorting algorithm for practical use concerns the trade-offs resulting from increasing the instruction count in order to reduce cache misses and other high-latency memory operations. To address this, we give an execution timing model to quantitatively predict the performance of an algorithm. We also give analytical predictions of the number of cache misses for the sorting algorithms before and after the cache optimizations. We show that cycles lost from increasing the instruction count to maximize cache reuse can be a negligible price to pay when compared to the many cycles that would otherwise be lost from different types of cache misses.

2.3 Architectural and Algorithmic Parameters and Evaluation Methodology

In this section, we first list and describe the architectural-dependent parameters we used in designing the algorithms. We then introduce the performance evaluation methodology and present the data sets used in the experiments.

2.3.1 Architectural and algorithmic parameters

A data set consists of a number of elements. One element may be a 4-byte integer, an 8-byte integer, a 4-byte floating point number, or an 8-byte double floating point number. We use the same generic unit, an element, to specify the cache capacity. Because the size of caches and cache lines are always a multiple of an element in practice, a general unit is practically meaningful to both architects and application programmers. The algorithmic and architectural parameters we will use to describe cache-effective sorting algorithms are as follows: N : the size of the data set, C : the data cache size, L : the size of a cache line, K : the cache associativity, T_s : the number of entries in a TLB set, K_{TLB} : the TLB associativity, and P_s : the size of a memory page.

2.3.2 Performance evaluation methodology

Directly monitoring and measuring a program's cache behavior is an important task for providing insights and guidance for optimizing the memory performance of an algorithm. Since current systems are not able to directly report memory related performance statistics (such as the number of cache hits or misses) during program execution, users must use

tools to gather these statistics. ATOM [118] is a system utility for DEC Alpha machines for instrumenting and analyzing program executables. The ATOM analysis tool accepts the results of an instrumented program and presents the cache performance statistics. Using the ATOM utility, users can directly monitor and measure the cache performance on DEC Alpha machines. The analysis of sorting algorithms in [78] uses the ATOM tool. Due to its platform dependence, memory performance studies using ATOM are not feasible on other types of machines.

The need for studying a broad range of platforms necessitates an alternative approach. We conducted our performance evaluation in two steps: (1) completing algorithm analysis and measuring performance on different high-end workstations, and (2) utilizing execution-driven simulations to gather insight into the memory performance of the algorithms on these machines. Employing the first step, we are able to measure the algorithm performance on a wide variety of machines. From the second step we are able to gather a deeper understanding of how the cache behavior affects the execution performance.

For our simulation environment, we used the SimpleScalar tool set [15], a family of simulators for studying interactions between application programs and computer architectures. The simulation tools take an application program's binaries compiled for the SimpleScalar Instruction Set Architecture (a close derivative of the MIPS instruction set) and generate statistics during the execution of the program on the simulated architecture. The statistics generated include many detailed execution traces that are not available from measurements on a computer, such as the number of cache misses in the L1, L2 and TLB caches.

We ran the compared sorting algorithms on different simulated memory architectures with memory hierarchies similar to those on typical high-end workstations to observe the

following performance factors:

- *L1 or L2 cache misses per element*: to compare the number of data cache misses.
- *TLB misses per element*: to compare the number of TLB misses.
- *Instruction count per element*: to compare the algorithmic complexities.
- *Reduction rate of total execution cycles*: to compare the percentage of cycles saved in comparison to the base mergesort or the memory-tuned quicksort.

2.3.3 Data sets

The algorithms are compared and evaluated experimentally and analytically. We tested the sorting algorithms on a variety of data sets consisting of 8-byte integer elements. The 9 data sets we used are enumerated below. (Probability Density Functions and Inverse Distribution Functions of some of the number generators used can be found in [95].)

1. *Random*: the data set is obtained by calling the random number generator **random()** from the C library, which returns integers in the range of 0 to $2^{31} - 1$.
2. *Equilikely*: function **Equilikely(a,b)** returns integers in the range a to b.
3. *Bernoulli*: function **Bernoulli(p)** returns integers 0 or 1.
4. *Geometric*: function **Geometric(p)** returns integers 0, 1, 2, ...
5. *Pascal*: function **Pascal(N,p)** returns integers 0, 1, 2, ...
6. *Binomial*: function **Binomial(N,p)** returns integers 0, 1, 2, ..., N.
7. *Poisson*: function **Poisson(μ)** returns integers 0, 1, 2, ...
8. *Zero*: the data set consists entirely of 0s.

9. Unbalanced: function returns integers in the range of 0 to $2^{15} - 1$ for $i = 0$ to $\frac{127}{128}N - 1$, by calling `rand()` from the C library, where i is the data element index and N is data set size; and returns integers $MAX/100 + i$ for $i = \frac{127}{128}N$ to N , where $MAX = 2^{31} - 1$.

2.4 Cache-Effective Mergesort Algorithms

In this section, we first briefly evaluate the two existing mergesort algorithms on their cache locality, as well as their merits and limits. We present two new mergesort alternatives to address these limits. The experimental performance evaluation done through measurements will be presented in Section 2.6.

2.4.1 Tiled mergesort and multimergesort

LaMarca and Ladner [78] present two mergesort algorithms to effectively utilize the data cache. The first one is called *tiled mergesort*. The basic idea is to partition the data set into subarrays that are sorted individually. This is mainly done for two reasons: to avoid capacity misses and to fully use the data loaded in the cache before it must be replaced. The algorithm is divided into two phases. In the first phase, subarrays of length $C/2$ (half the cache size) are sorted by the base mergesort algorithm to exploit temporal locality. The algorithm returns to the base mergesort without considering cache locality in the second phase to complete the sorting of the entire data set.

The second mergesort, called *multimergesort*, addresses the limits of the tiled mergesort. In this algorithm, the first phase is the same as the first phase of the tiled mergesort. In the second phase, a multiway merge method is used to merge all the sorted subarrays together in a single pass. A priority queue is used to hold the heads of the lists (the sorted subarrays

from the first phase) to be merged. This algorithm exploits cache locality well when the number of subarrays in the second phase is less than $C/2$ (half the cache size). However, the instruction count is significantly increased in this algorithm.

Our analysis of the two mergesort algorithms shows two areas for improvement. First, both algorithms significantly reduce capacity misses, but do not sufficiently reduce conflict misses. In mergesort, the basic idea is to merge two sorted subarrays to a destination array. In a cache with low associativity, mapping conflicts occur frequently among the elements in the three subarrays. Also, reducing TLB misses is not considered in the algorithm designs. Even when the data set is only moderately large, TLB misses may severely degrade execution performance, compounding the effect of normal data cache misses. Our experiments show that the performance improvement of the multimerge algorithm on several machines is modest—although it decreases the number of data cache misses, the heap structure significantly increases the number of TLB misses.

2.4.2 New mergesort alternatives

We present two new restructured mergesort alternatives for reducing conflict misses and TLB misses with a minimized instruction count increase: *tiled mergesort with padding* and *multimergesort with TLB padding*.

2.4.2.1 Tiled mergesort with padding

Padding is a technique that modifies the data layout of a program so that conflict misses are reduced or eliminated. The data layout modification can be done at run-time by system software [9, 140] or at compile-time by compiler optimization [103]. However, padding done

at the algorithm design level using a full understanding of the data structures is expected to outperform optimizations using the two methods above [146].

In the second phase of the tiled mergesort, pairs of sorted subarrays are sorted and merged into a destination array. One element from each of the two subarrays is selected at a time for a sorting comparison in sequence. These three data elements in the two different subarrays and the destination array can potentially be in conflicting cache blocks because they may be mapped to the same block in a direct-mapped cache and in a 2-way associative cache. This phenomenon occurs most often when the source array (containing the two subarrays) and the destination array are allocated contiguously in memory.

On a direct-mapped cache, the total number of conflict misses for the tiled mergesort in the worst case is approximately

$$(1 + \frac{1}{2C})N \lceil \log_2 \frac{2N}{C} \rceil. \quad (2.1)$$

where $\log_2 \frac{2N}{C}$ is the number of passes in the second phase of the sorting and $1 + \frac{1}{2C}$ represents 1 conflict miss per comparison and $\frac{1}{2C}$ conflict misses for every time an element is placed into the destination array following a comparison, respectively.

In order to change the base addresses of these potentially conflicting cache blocks, we insert L elements (or a spacing the size of a cache line) to separate every section of C elements in the data set in the second phase of the tiled mergesort. These padding elements can significantly reduce the cache conflicts in the second phase of the mergesort. The memory used by the padding elements is trivial when compared to the size of the data set. The increase in the instruction count (resulting from having to move each element in a subarray to its new position for the padding) is also minor. We call this method as *tiled*

mergesort with padding.

On a direct-mapped cache, the total number of conflict misses for the tiled mergesort with padding is at most

$$\frac{3}{4}N \lceil \log_2 \frac{2N}{C} \rceil. \quad (2.2)$$

where $\log_2 \frac{2N}{C}$ is the number of passes in the second phase of the sorting and $\frac{3}{4}$ represents the number of conflict misses per element. After the padding is added, the one conflict miss per comparison is reduced to $\frac{3}{4}$, and the $\frac{1}{2C}$ conflict misses from the placement in (2.1) are eliminated. Comparing the two approximations in (2.1) and (2.2), we see that tiled mergesort with padding reduces the conflict misses of tiled mergesort by about 25%. (Our experimental results on the Sun Ultra 5, a workstation with a direct-mapped cache, show that execution times of tiled mergesort were reduced 23% to 68% by tiled mergesort with padding. These execution time reductions mainly come from the abatement of conflict misses.)

Figure 2.1 shows an example of how the data layout of two subarrays in the second phase of tiled mergesort is modified by padding to reduce conflict misses. In this example, a direct-mapped cache holds 4 elements. In the figure, identical lines represent a pair comparison and the corresponding action to store the selected element in the destination array. The letter “m” in the figure represents a cache miss. Without padding, there are 8 conflict misses when merging the two sorted subarrays into the destination array; there are only 4 after padding is added.

Figure 2.2 shows the L1 misses (see the left figure) and the L2 misses (see the right figure) of the base mergesort, tiled mergesort, and tiled mergesort with padding on a simulated machine with the cache architecture of a Sun Ultra 5 using SimpleScalar. On this machine,

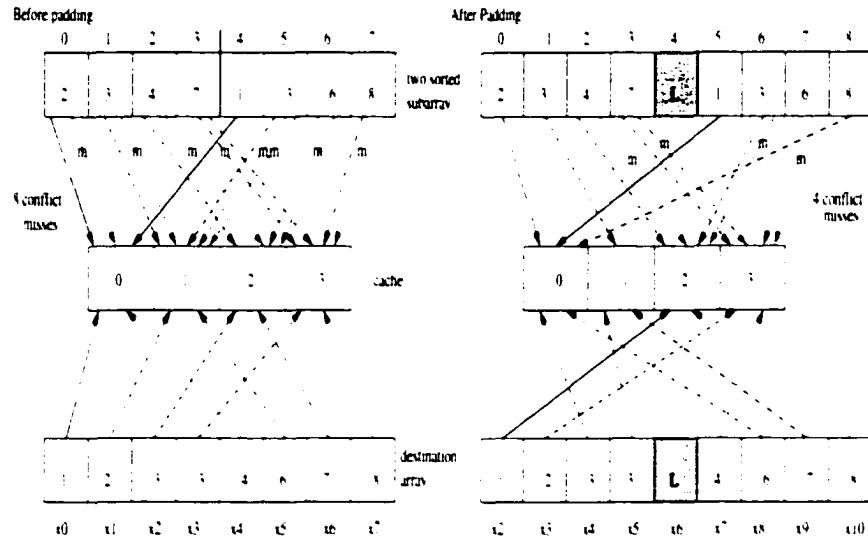


Figure 2.1: Data layout of subarrays is modified by padding to reduce the conflict misses.

the L1 cache is direct-mapped and contains 16 KBytes, and the L2 cache is 2-way associative holding 256 KBytes. The experiments show that the padding reduces the L1 cache misses by about 23% compared with the base mergesort and tiled mergesort. These misses are conflict misses that cannot be reduced through tiling. The L2 cache miss reduction by tiled mergesort with padding is almost the same as that by tiled mergesort, which shows that the padding is not very effective in reducing L2 conflict misses on this machine. This is because the 2-way associativity in the L2 cache significantly reduces the percentage of conflict misses, in comparison to the direct-mapped L1 cache.

Capacity misses in the second phase of the tiled mergesort are unavoidable without a complex data structure, because the size of the working set (two subarrays and a destination array) is normally larger than the cache size. As we have shown, conflict misses can be reduced by padding in this phase. However, the padding may not completely eliminate all conflict misses due to the randomness of the order in the data sets. Nevertheless, our

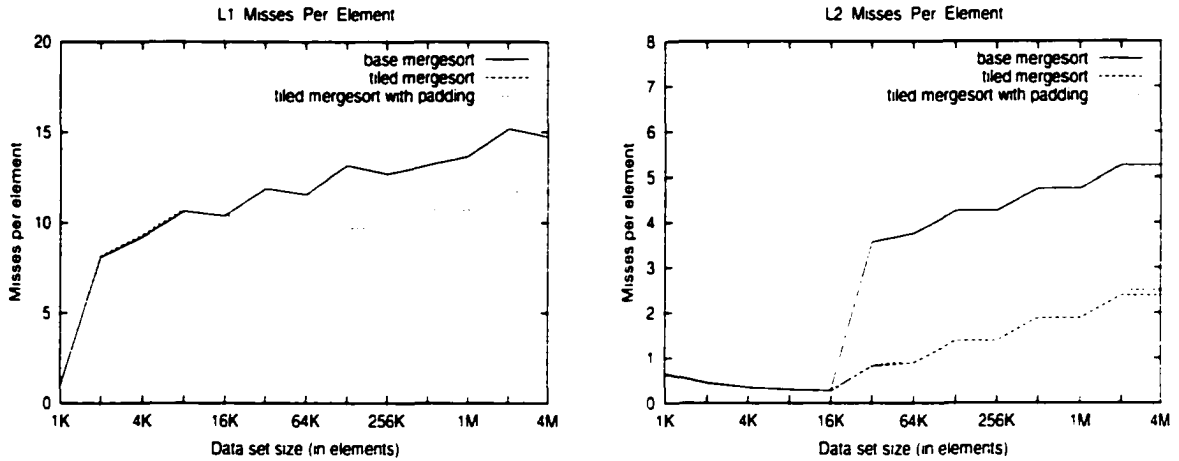


Figure 2.2: Simulation comparisons of the L1 cache misses (left figure) and L2 misses (right figure) of the mergesort algorithms on the *Random* data set on the simulated Sun Ultra 5. The L1 cache miss curves (left figure) of the base mergesort and tiled-mergesort are overlapped.

experimental results presented in Section 2.6 using the 9 different data sets consistently show the effectiveness of the tiled mergesort with padding on the Sun Ultra 5.

2.4.2.2 Multimergesort with TLB padding

In the second phase of the multimergesort algorithm, the multiple subarrays are completely sorted in a single pass. Multiple subarrays are used only once to complete the sorting of the entire data set to effectively use the cache. This single pass makes use of a heap structure to hold the head elements of the multiple subarrays. (Because of this structure, we will often refer to these subarrays as lists.) However, since the heads come from all the lists being multimerged, the working set is much larger than that of the base mergesort (where only three subarrays are involved at a time). This large working set causes TLB misses that degrade performance. (We will explain the TLB structure in the following paragraph.) Our experiments indicate that multimergesort does significantly decrease the number of data cache misses; however, it also increases the TLB misses, which offsets the

performance gain. Although a rise in the instruction count leads to additional CPU cycles in *multimergesort*, this has a minimal effect. The performance of the algorithm is degraded mainly from the high number of TLB misses—memory accesses are far more expensive than CPU cycles.

The TLB (Translation-Lookaside Buffer) is a special cache that stores the most recently used virtual-physical page translations for memory accesses. The TLB is generally a small fully associative or set-associative cache. Each entry points to a memory page of 4 to 64Kbytes, depending on the architecture. A TLB cache miss forces the system to retrieve the missing translation from the page table in memory, and then to replace an existing TLB entry with this translation. The TLB can hold a limited amount of data for sorting. When the data to be accessed spans more memory pages mapping to the same TLB set than the TLB can hold, TLB misses will occur. For example, the TLB cache of the Sun UltraSparc-III processor holds 64 fully associative entries ($T_s = 64$), each of which points to a page of 8 KBytes ($P_s = 1024$ 8-byte elements). The 64 pages in the TLB of the Sun UltraSparc-III processor hold $64 \times 1024 = 65536$ elements, which represents a small-sized data set for sorting. Furthermore, in practice we often have more than one data array being operated on at a time. Some processors' TLBs are not fully associative, but set-associative. For example, the TLBs in the Pentium II and Pentium III processors are 4-way associative ($K_{TLB} = 4$).

In the second phase of *multimergesort*, we insert P_s elements (or a page space) to separate every sorted subarray in the data set in order to reduce or eliminate the TLB cache conflict misses. The padding changes the base addresses of these lists in page units to avoid potential TLB conflict misses.

Figure 2.3 exemplifies how padding for the TLB works: in this case the TLB is a direct-mapped cache of 8 entries, and the number of elements in each list is a multiple of 8 page elements. Before padding, each of the lists in the data set is mapped to the same TLB entry. After padding, these lists are mapped to different TLB entries.

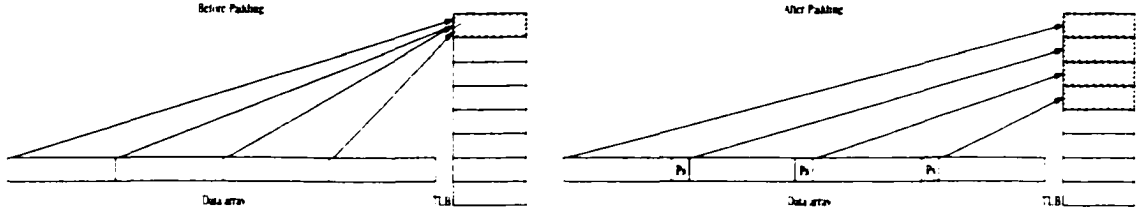


Figure 2.3: Padding for TLB: the data layout is modified by inserting a page space at multiple locations, where $K_{TLB} = 1$, and $T_s = 8$.

After padding, these lists are mapped to different TLB entries. When multimergesort is run on a large data set and the size of each list is a multiple of T_s , the number of TLB misses per element is close to 1. After the TLB padding, the average number of TLB misses per element for the multimergesort algorithm becomes approximately

$$\frac{A}{A + K_{TLB}} \quad (2.3)$$

where $A = \frac{C}{T_s}$ is the number of average misses for each TLB set entry. The above approximation is further derived to

$$\frac{C}{C + K_{TLB} \times T_s} \quad (2.4)$$

Figure 2.4 shows the number of L2 misses and TLB misses for the five mergesort algorithms on the Pentium II memory architecture as simulated using SimpleScalar, where the L1 cache is 4-way set-associative with 16 KBytes, the L2 cache is 4-way set-associative with 256 KBytes, and the TLB is a 4-way set-associative cache with 64 entries. The simulation shows that multimergesort and multimergesort with TLB padding have the lowest L2 cache

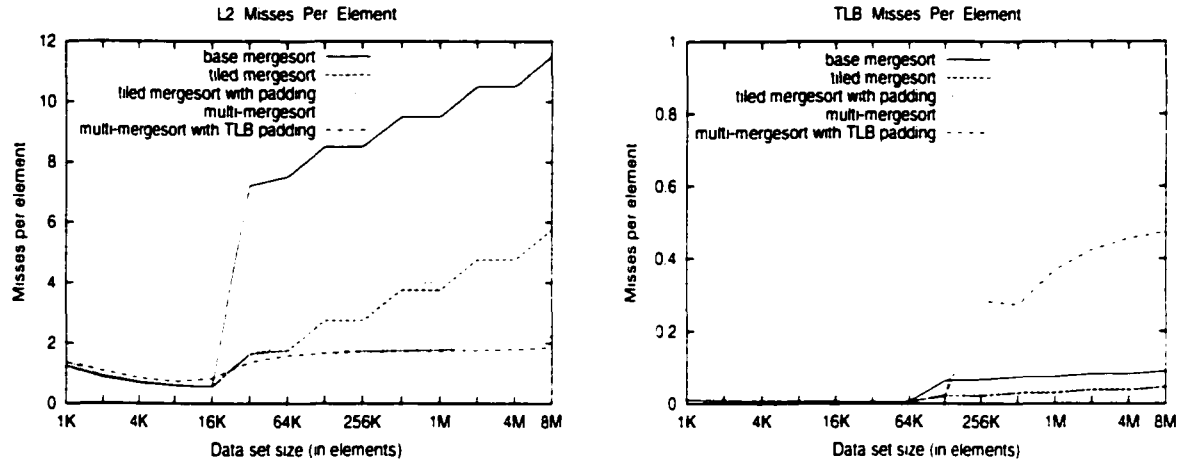


Figure 2.4: Simulation comparisons of the L2 cache misses (left figure) and TLB misses (right figure) of the mergesort algorithms on the *Random* data set on the simulated Pentium II.

misses among the different algorithms (see the left figure in Figure 2.4). Multimergesort also had the highest number of TLB misses, but these misses are considerably reduced by the TLB padding (see the right figure in Figure 2.4).

Here is an example verifying the approximation in (2.4) of TLB misses of multimergesort. Substituting the parameters of Pentium II to the approximation, $C = 256$, $K_{TLB} = 4$, and $T_s = 64$, we get 0.5 TLB misses per element for multimergesort with TLB padding, which is very close to our experimental result, 0.47 (in the right figure of Figure 2.4). We will show in Section 2.6 that multimergesort with TLB padding significantly reduces TLB misses and improves overall execution performance.

2.4.3 Trade-offs relating to an instruction count increase and the performance gain

Figure 2.5 shows the instruction counts of the five mergesort algorithms and the percentage of total cycles saved by the four improved mergesort algorithms compared to the base

mergesort on the simulated Pentium II. The simulation shows that multimergesort had the

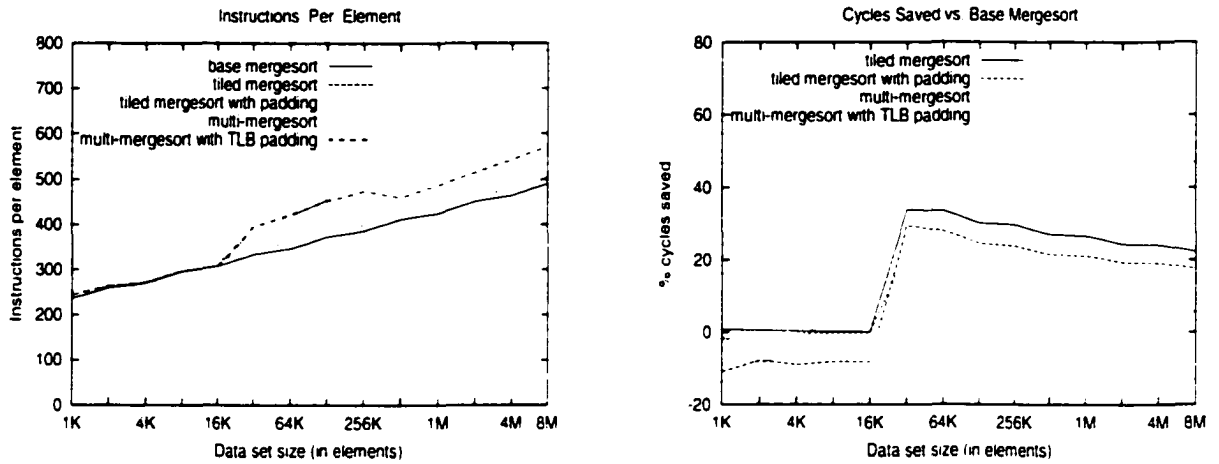


Figure 2.5: Simulation comparisons of the instruction counts (left figure) and saved cycles in percentage (right figure) of the mergesort algorithms on the *Random* data set on the simulated Pentium II. The instruction count curves (left figure) of the base mergesort and the tiled mergesort are overlapped.

highest instruction count, while tiled mergesort had the lowest instruction count. Taking advantage of the low number of L2 cache misses in multimergesort and by reducing the TLB misses through padding, multimergesort with TLB padding saved cycles by about 40% on large data sets compared to the base mergesort even though it has a relatively high instruction count. Tiled mergesort with padding did not improve performance on the Pentium II. This is because this machine has a 4-way set associative cache where conflict misses are not major concerns.

2.5 Cache-Effective Quicksort

We first briefly assess the merits and limits of the two existing quicksort algorithms, especially considering their cache locality. We present two new quicksort alternatives for improving memory performance further. Experimental results will be reported in the next

section.

2.5.1 Memory-tuned quicksort and multiquicksort

LaMarca and Ladner in the same paper [78] present two quicksort algorithms for cache optimization. The first one is called *memory-tuned quicksort*, which is a modification of the base quicksort [110]. Instead of saving small subarrays to sort in the end, the memory-tuned quicksort sorts these subarrays when they are first encountered in order to reuse the data elements in the cache.

The second algorithm is called *multiquicksort*. This algorithm applies a single pass to divide the full data set into multiple subarrays, with the hope that each subarray will be smaller than the cache capacity.

The performance gain of these two algorithms from experiments reported in [78] is modest. We implemented the two algorithms on simulated machines and on various high-end workstations and obtained consistent performance. We also found that the performance of quicksort and its cache-optimized alternatives are very sensitive to the types of the data set being used. These algorithms were not efficient on unbalanced data sets.

2.5.2 New quicksort alternatives

In practice, the quicksort algorithms exploit cache locality well on balanced data. A challenge is to make the quicksort perform well on unbalanced data sets. We present two cache-optimized quicksort alternatives that work well on both balanced and unbalanced data sets.

2.5.2.1 Flash Quicksort

Flashsort [92] is extremely fast for sorting balanced data sets. The maximum and minimum values are first identified in the data set to identify the data range. The data range is then evenly divided into classes to form subarrays. The algorithm consists of three steps: “classification” to determine the size of each class, “permutation” to move each element into its class by using a single temporary variable to hold the replaced element, and “straight insertion” to sort elements in each class by using Sedgewick’s insertion sort [110]. This algorithm works very well on balanced data sets because the sizes of the subarrays after the first two steps are similar and are small enough to fit in the cache. This makes flashsort highly effective ($O(N)$). However, when the data set is not balanced, the sizes of the generated subarrays are disproportionate, causing ineffective usage of the cache, and making flashsort as slow as insertion sort ($O(N^2)$) in the worst case.

In comparison with the pivoting process of quicksort, the classification step of flashsort is more likely to generate balanced subarrays, which favors better cache utilization. On the other hand, quicksort outperforms insertion sort on unbalanced subarrays. By combining the advantages of flashsort and quicksort, we present a new quicksort alternative, *flash quicksort*, where the first two steps are the same as in flashsort and the last step uses quicksort to sort the elements in each class.

2.5.2.2 Inplaced flash quicksort

To further improve overall performance, we employ another cache optimization to improve temporal locality in flash quicksort. We call this alternative *inplaced flash quicksort*. In this algorithm, the first and third steps are the same as in flash quicksort. In the second

step, an additional array is used as a buffer to hold the permuted elements. In the original flashsort, a single temporary variable is used to hold the replaced element. A cache line normally holds more than one element. The data structure of the single variable minimizes the chance of data reuse. Using the additional array, we attempt to reuse elements in a cache line before their replacement and to reduce the instruction count for copying data elements. Although this approach increases the required memory space, it improves both cache and overall performance.

2.5.3 Simulation results

Figure 2.6 shows the instruction counts (left figure) and the L1 misses (right figure) of memory-tuned quicksort, flashsort, flash quicksort, and inplace flash quicksort, on the *Unbalanced* data set on the simulated Pentium III memory architecture, which has a higher memory latency and a larger L2 cache (512 KBytes) than the Pentium II. The instruction

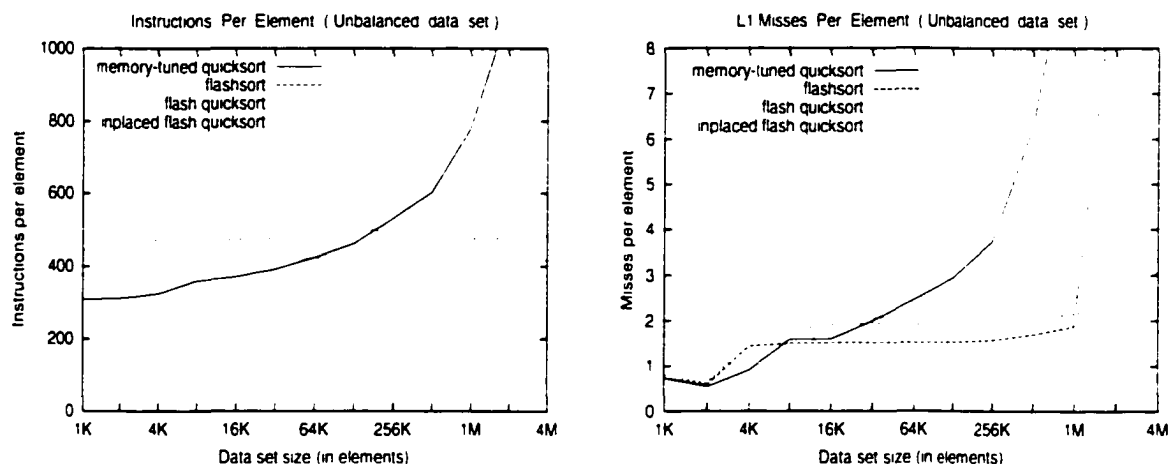


Figure 2.6: Simulation comparisons of the instruction counts (left figure) and the L1 misses (right figure) of the quicksort algorithms on the *Unbalanced* data set on the simulated Pentium III. The instruction count curve of the flashsort was too high to be presented in the left figure.

count curve of flashsort was too high to be presented in the left figure of Figure 2.6. The

same figure shows that the instruction count of memory-tuned quicksort also increases rapidly as the data set size grows. In contrast, the instruction counts of flash quicksort and inplaced flash quicksort change little as the data set size increases. The simulation also shows that the number of L1 misses increases much more rapidly as the size of the data set grows in the memory-tuned quicksort and flashsort than in the flashsort and inplaced flashsort algorithms. The simulation results are consistent with our algorithm analysis, and show the effectiveness of our new quicksort alternatives on unbalanced data sets.

2.6 Measurement Results and Performance Evaluation

We have implemented and tested all the sorting algorithms discussed in the previous sections on all the data sets described in Section 2.3 on a SGI O2 workstation, a Sun Ultra-5 workstation, a Pentium II PC, and a Pentium III PC. The data sizes we used for experiments are limited by the memory size available on the experimental machines since we focus on cache-effective methods. We used “lmbench” [86] to measure the latencies of the memory hierarchy at its different levels on each machine. The architectural parameters of the four machines are listed in Table 2.1, where all the L1 cache specifications refer to the L1 data cache; all the L2 caches are uniform. The hit times of the L1 and L2 caches and the main memory measured by lmbench have been converted to the corresponding number of CPU cycles.

We compared all our algorithms with the algorithms in [78] and [92]. The execution times were collected by “gettimeofday()”, a standard Unix timing function. The reported

Table 2.1: Architectural parameters of the 4 machines we have used for the experiments.

Workstations	SGI O2	Sun Ultra 5	Pentium	Pentium
Processor type	R10000	UltraSparc-IIi	Pentium II 400	Pentium III Xeon 500
clock rate (MHz)	150	270	400	500
L1 cache (KBytes)	32	16	16	16
L1 block size (Bytes)	32	32	32	32
L1 associativity	2	1	4	4
L1 hit time (cycles)	2	2	2	3
L2 cache (KBytes)	64	256	256	512
L2 associativity	2	2	4	4
L2 hit time (cycles)	13	14	21	24
TLB size (entries)	64	64	64	64
TLB associativity	64	64	4	4
Memory latency (cycles)	208	76	68	67

time unit is cycle per element (*CPE*):

$$CPE = \frac{\text{execution time} \times \text{clock rate}}{N}.$$

where *execution time* is the measured time in seconds, *clock rate* is the CPU speed (in cycles per second) of the machine where the program is run, and *N* is the number of elements in the data set.

Each execution time reported in this chapter represents the mean of 20 runs. The variances range from 0.096 to 23.72 cycles² (corresponding to standard deviations ranging from 0.31 to 4.87 cycles). As a result, the coefficients of variation calculated by the ratio of the standard deviation to the mean is in a range of 0.00035 to 0.01.

The performance results on all the data sets are fairly consistent. Because of this, we only present the performance results of the mergesort algorithms using the *Random* data set on the four machines (plus performance results of the other data sets on the Ultra 5 to

show the effectiveness of the tiled mergesort with padding). We present the performance results of the quicksort algorithms using the *Random* and the *Unbalanced* data sets on the four machines.

2.6.1 Mergesort performance comparisons

We compared five mergesort algorithms: the base mergesort, tiled mergesort, multimerge-sort, tiled mergesort with padding, and multimergesort with TLB padding. Proportional to each machine's memory capacity, we scaled the mergesort algorithms from $N=1K$ up to $N=16M$ elements. All our algorithms demonstrated their effectiveness as the data set size grew. Figure 2.7 shows comparisons of cycles spent per element for the five algorithms on the SGI O2 and the Sun Ultra 5. Multimergesort with TLB padding performed the best

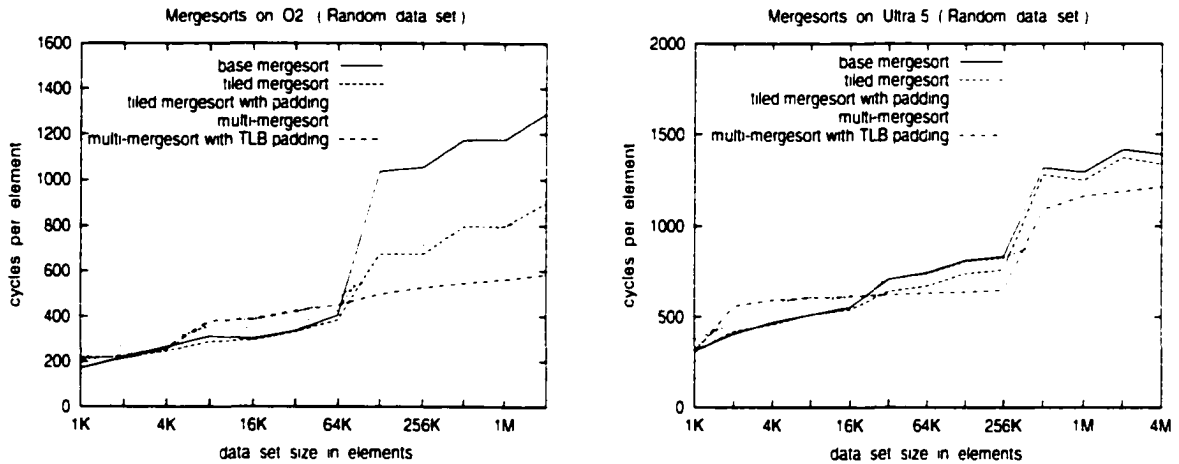


Figure 2.7: Execution comparisons of the mergesort algorithms on SGI O2 and on Sun Ultra 5.

on the O2, with execution times reduced 55% compared to the base sort, 35% compared to tiled mergesort, and 31% compared to multimergesort on 2M elements. On the other hand, tiled mergesort with padding performed the best on the Ultra 5, reducing execution times 45% compared to multimergesort, 26% to the base mergesort, and 23% to tiled mergesort

on 4M elements. Multimergesort with TLB padding on Ultra 5 also did well, with a 35% improvement over multimergesort, 13% over the base mergesort, and 9% over tiled mergesort on 4M elements. The reason for the incredible performance improvements on the O2 is its long memory latency (208 cycles): this makes the cache miss reduction techniques very effective in improving the overall performance of the sorting algorithms. The L2 cache size of the SGI is relatively small (64 KBytes) and the TLB is frequently used for memory accesses. Thus, TLB padding is very effective. In addition, both L1 and L2 caches are 2-way set associative, where the padding is not as effective as on a direct-mapped cache. In contrast, the Ultra 5's L1 cache is direct-mapped and the L2 cache is 4 times larger than that of the O2. On this platform data cache padding is more effective than TLB padding.

In order to show the effectiveness of tiled-mergesort with padding on a cache system with a low associativity, the performance curves of the five mergesort algorithms from the Sun Ultra 5 on the other 8 data sets are provided in Figure 2.8 - 2.11. Our experiments

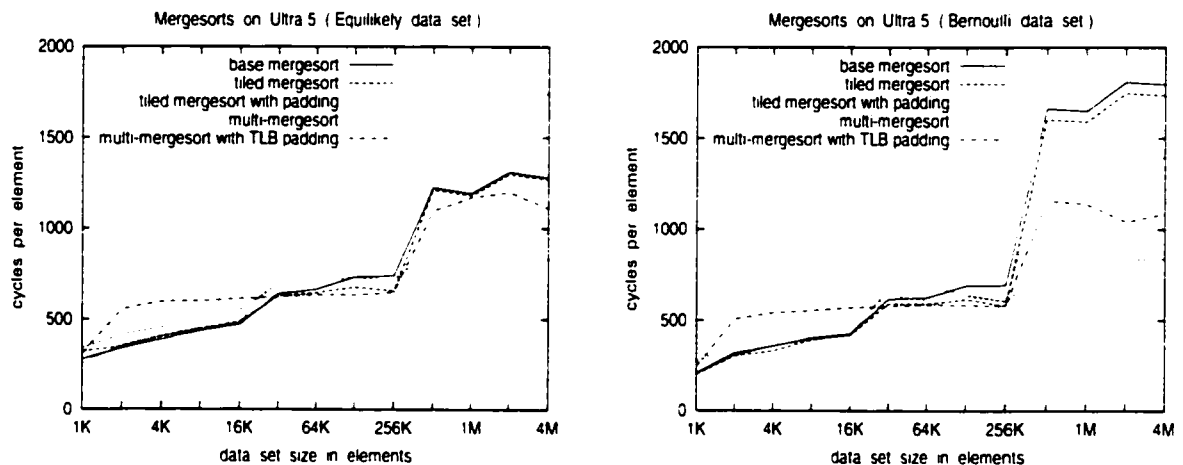


Figure 2.8: Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Equilikely data set (left figure) and the Bernoulli data set (right figure).

show that tiled-mergesort with padding consistently and significantly outperforms the other

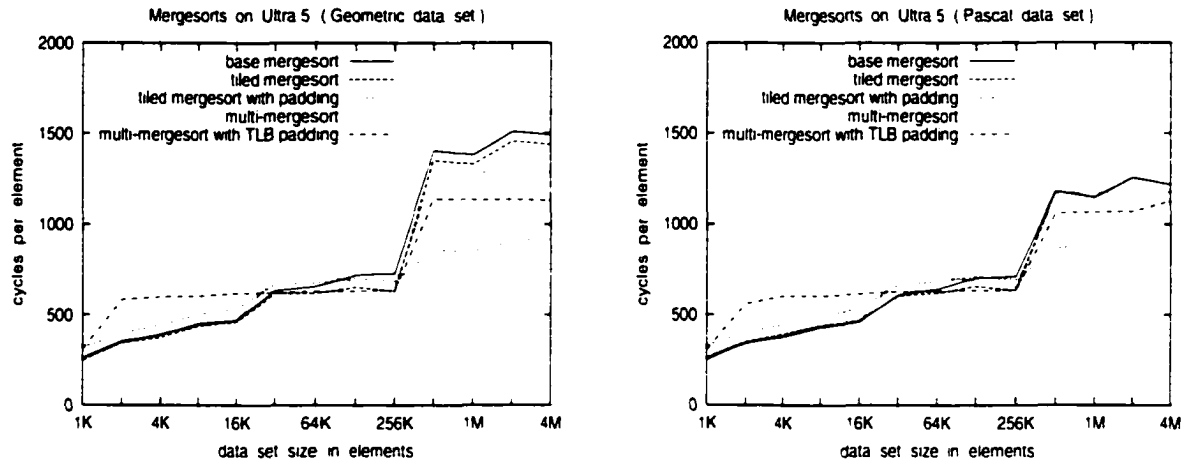


Figure 2.9: Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Geometric data set (left figure) and the Pascal data set (right figure).

mergesort algorithms on the Ultra 5. For example, tiled mergesort with padding achieved 70%, 68%, and 54% reductions in execution time on the *Zero* data set compared with the base mergesort, tiled mergesort, and multimergesort, respectively. Using other data sets, tiled mergesort with padding achieved 24% to 53% reductions in execution time compared with the base mergesort, 23% to 52% reductions compared with tiled mergesort, and 23% to 44% reductions compared with multimergesort.

Figure 2.12 shows the comparisons of cycles per element between the five mergesort algorithms on the Pentium II 400 and the Pentium III 500. The measurements on both machines show that multimergesort with TLB padding performed the best, reducing execution times 41% compared with multimergesort, 40% compared with the base mergesort, and 26% compared with tiled sort on 16M elements. The L1 and L2 caches of both machines are 4-way set associative so the issue of data cache conflict misses is not a concern (as discussed in Section 2.4.1). Since TLB misses are the main source of inefficiency in the multimergesort algorithm, padding for the TLB is very effective in improving the performance.

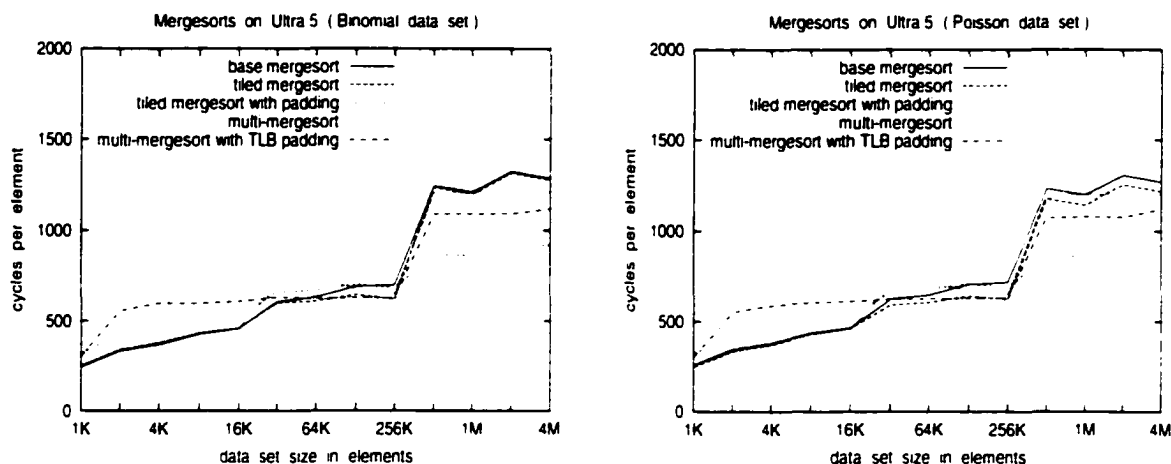


Figure 2.10: Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Binomial data set (left figure) and the Poisson data set (right figure).

In summary, tiled mergesort with padding is highly effective in reducing conflict misses on machines with direct-mapped caches and multimergesort with TLB padding performs very well on all types of architectures.

2.6.2 Quicksort performance comparisons

We present the results of quicksort algorithms on the 4 machines using the *Random* data set and the *Unbalanced* data set. The 4 quicksort algorithms are: the memory-tuned quicksort, flashsort, flash quicksort, and the inplace flash quicksort.

Figure 2.13 shows the comparisons of cycles per element between the four quicksort algorithms on the *Random* data set (left) and the *Unbalanced* data set (right) on the SGI O2 machine. The performance results of the four quicksort algorithms using the *Random* data set are comparable, with the memory-tuned algorithm slightly outperforming the others. The performance results using the *Unbalanced* data set are much different. As we expected, the number of cycles spent to sort each element is relatively stable for flash quicksort and

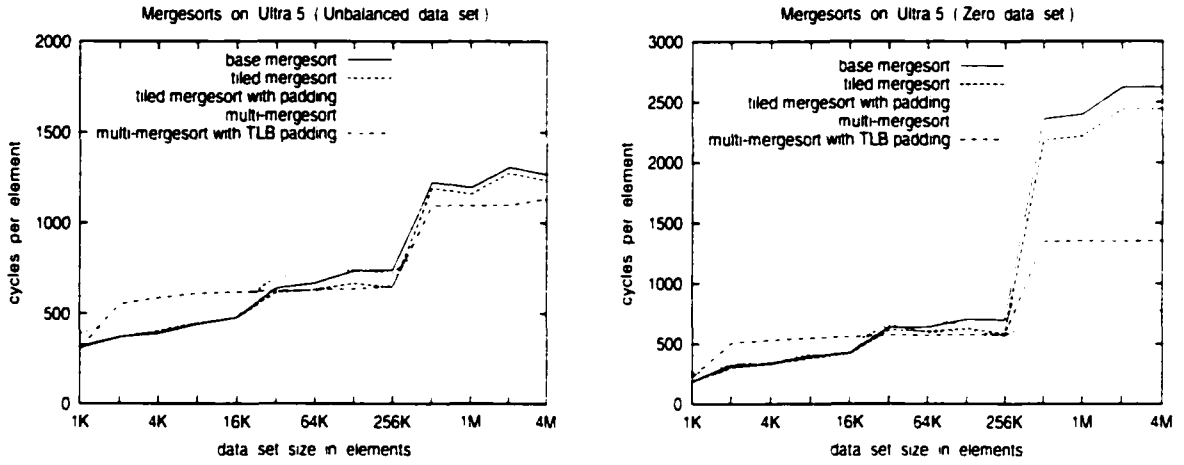


Figure 2.11: Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Unbalanced data set (left figure) and the Zero data set (right figure).

the inplaced flash quicksort as the size of the data set increases, while the performance of memory-tuned quicksort and flashsort greatly diminishes. The timing curves of flashsort are even too high to be shown in the right figure in Figure 2.13.

Figure 2.14 shows the comparisons of cycles per element among the four quicksort algorithms on the *Random* data set (left) and the *Unbalanced* data set (right) on the Sun Ultra 5 machine. On the Ultra 5, all four algorithms showed little difference in their execution times on the *Random* data set. On the other hand, the flash and inplaced flash quicksorts exhibited much better performance on the *Unbalanced* data set. For example, when the data set increased to 128K elements, the execution time of flashsort is more than 10 times higher than that of the other three algorithms (the curve is too high to be plotted in the figure). When the data set is increased to 4M elements, the execution time of the memory-tuned quicksort is more than 3 times higher than that of the flash quicksort and inplaced flash quicksort, and the execution time of the flashsort is more than 100 times higher than that of the others.

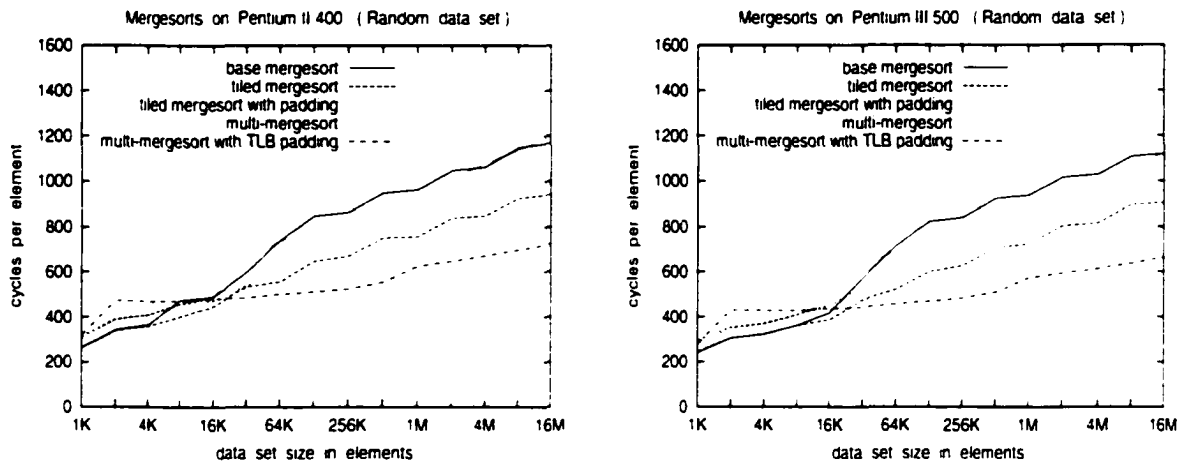


Figure 2.12: Execution comparisons of the mergesort algorithms on Pentium II and on Pentium III.

Figure 2.15 and Figure 2.16 show the comparisons of cycles per element between the four quicksort algorithms on the *Random* data set (left) and the *Unbalanced* data set (right) on the Pentium II and the Pentium III machine respectively. On both Pentiums using the *Random* data set, flashsort, flash quicksort, and inplace flashsort displayed similar execution performance and reduced execution times around 20% compared to the memory-tuned quicksort on large data sets. Again, flash quicksort and inplace flash quicksort significantly outperformed the memory-tuned quicksort algorithm on the *Unbalanced* data sets on the two Pentium machines.

2.7 A Prediction Model of Performance Trade-Offs

The essential issue that must be considered when designing an algorithm that has an efficient memory access pattern is the trade-off between the optimization achievement --the reduction of cache misses, and the optimization effort--the increment in the instruction count. The optimization objective is to improve overall performance--to reduce the ex-

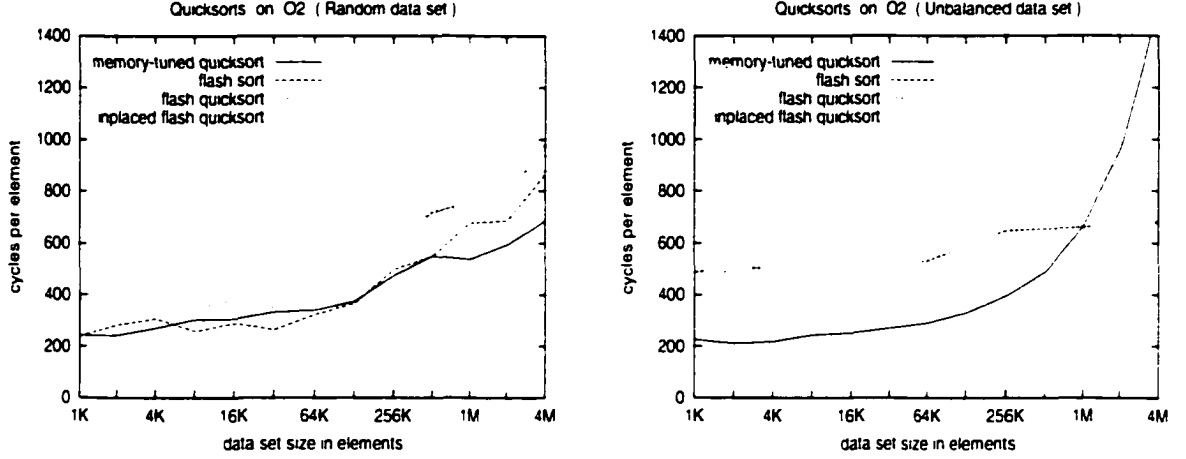


Figure 2.13: Execution comparisons of the quicksort algorithms on the *Random* data set (left figure) and on the *Unbalanced* data set (right figure) on the SGI O2. (The timing curve of the flashsort is too high to be presented in the right figure).

ecution time of a base algorithm. This trade-off and the objective can be quantitatively predicted through an execution timing model. The execution time of an algorithm on a computer system based on Amdahl's Law [67] is expressed as

$$T = CPU \text{ clock cycles} + memory \text{ stall cycles} = IC \times CPI + CA \times MR \times MP. \quad (2.5)$$

where IC is the instruction count of the algorithm, CPI is the number of CPU cycles per instruction for the algorithm, CA is the number of cache accesses of during the algorithm's execution, MR is the cache miss rate of the algorithm, and MP is the miss penalty in cycles of the system. The execution time for a base algorithm, T_{base} , is expressed as

$$T_{base} = IC_{base} \times CPI + CA_{base} \times MR_{base} \times MP. \quad (2.6)$$

and the execution time for an optimized algorithm, T_{opt} , is expressed as

$$T_{opt} = IC_{opt} \times CPI + CA_{opt} \times MR_{opt} \times MP. \quad (2.7)$$

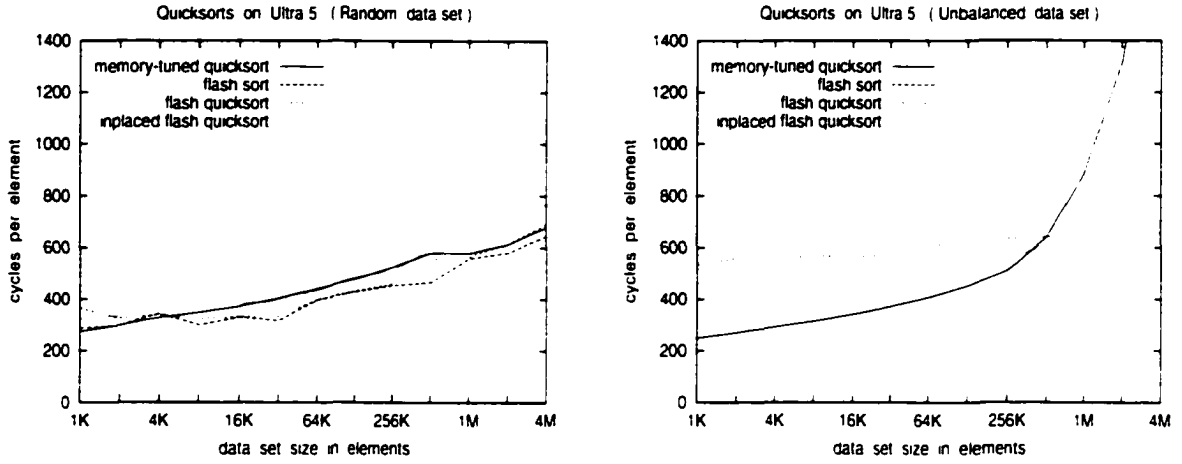


Figure 2.14: Execution comparisons of the quicksort algorithms on the *Random* data set (left figure) and on the *Unbalanced* data set (right figure) on the Ultra 5. (The timing curve of the flashsort is too high to be presented in the right figure).

where IC_{base} and IC_{opt} are the instruction counts for the base algorithm and the optimized algorithm. CA_{base} and CA_{opt} are the numbers of cache accesses of the base algorithm and the optimized algorithm, and MR_{base} and MR_{opt} are the cache miss rates of the base algorithm and the optimized algorithm, respectively.

In some optimized algorithms such as tiled mergesort and tiled mergesort with padding, the total number of cache accesses may be nearly the same as for the base algorithm. For this type of algorithms, we combine equations (2.6) and (2.7) with $CA_{base} = CA_{opt} = CA$ to predict the execution time reduction rate of an optimized algorithm as follows:

$$R = \frac{T_{base} - T_{opt}}{T_{base}} = \frac{\Delta MR \times CA \times MP - \Delta IC \times CPI}{IC_{base} \times CPI + CA_{base} \times MR_{base} \times MP}. \quad (2.8)$$

where $\Delta MR = MR_{base} - MR_{opt}$ represents the miss rate reduction, and $\Delta IC = IC_{opt} - IC_{base}$ represents the instruction count increment. In order to obtain a positive reduction

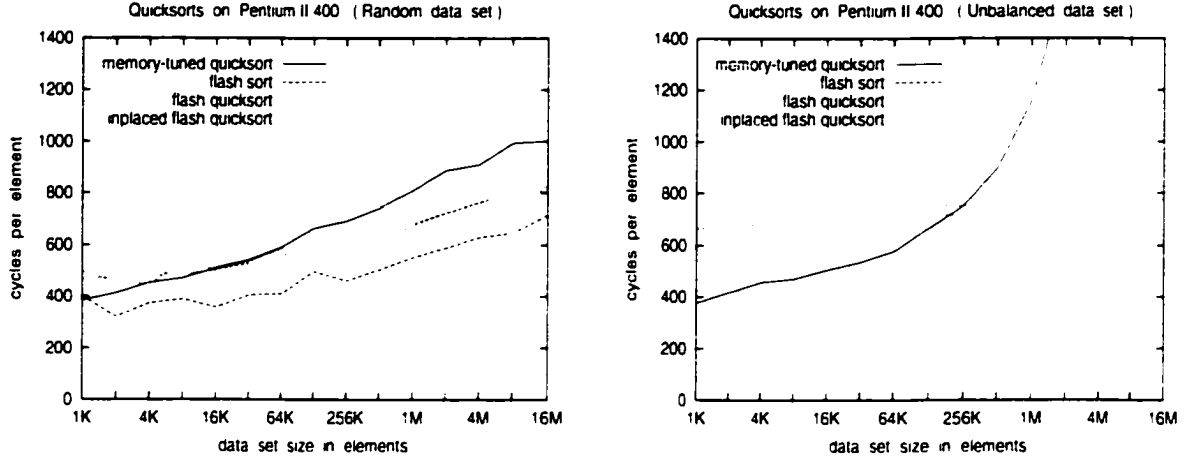


Figure 2.15: Execution comparisons of the quicksort algorithms on the *Random* data set (left figure) and on the *Unbalanced* data set on the Pentium II. (The timing curve of the flashsort is too high to be presented in the right figure).

in execution time, the following must hold true:

$$\Delta MR \times CA \times MP > \Delta IC \times CPI.$$

This model describes the quantitative trade-off between instruction count increase and miss rate reduction, and gives the condition for an optimized algorithm to improve the performance of a base algorithm:

$$\frac{\Delta IC}{\Delta MR} < \frac{CA \times MP}{CPI}. \quad (2.9)$$

For multiphase optimized algorithms which have different cache access patterns in each phase, such as multimergesort and multimergesort with TLB padding, we combine equations (2.6) and (2.7) with $CA_{base} \neq CA_{opt}$ to obtain the condition for an optimized algorithm to improve the performance of a base algorithm:

$$\frac{\Delta IC}{\Delta(MR \times CA)} < \frac{MP}{CPI}. \quad (2.10)$$

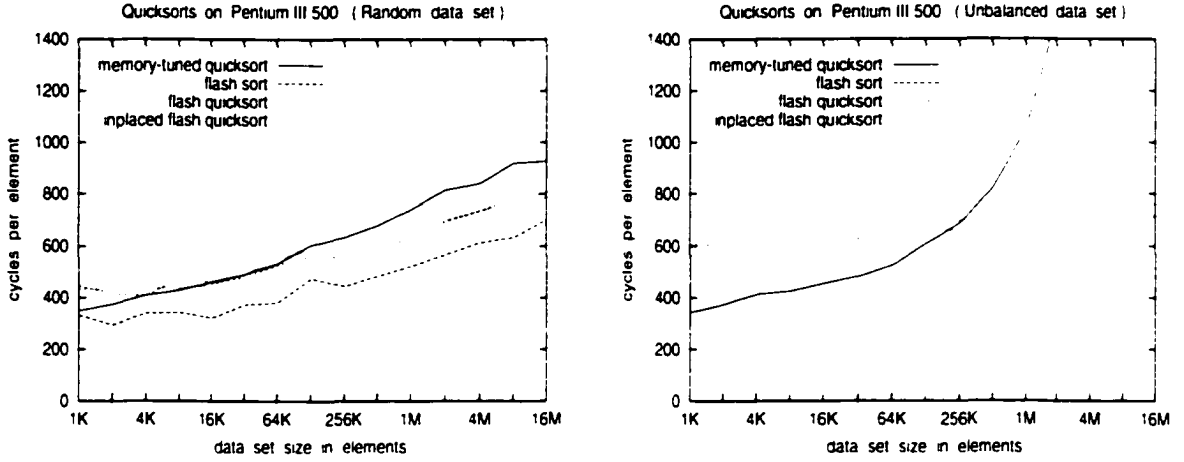


Figure 2.16: Execution comparisons of the quicksort algorithms on the *Random* data set (left figure) and on the *Unbalanced* data set on the Pentium III. (The timing curve of the flashsort is too high to be presented in the right figure).

where $\Delta(MR \times CA) = MR_{base} \times CA_{base} - MR_{opt} \times CA_{opt}$.

There are architecture related and algorithm related parameters in this prediction model. The architecture related parameters are *CPI* and *MP*, which are machine-dependent and can be easily obtained. The algorithm related parameters are *IC*, *CA*, and *MR*, which can be either predicted from algorithm analysis or obtained from running the program on a simulated architecture, such as SimpleScalar. The algorithm related parameters can also be predicted by running the algorithms on relatively small data sets that are larger than the cache capacity on a target machine.

Using the prediction model and the parameters from the SimpleScalar simulation, we are able to predict the execution time rate of reduction for the optimized algorithms. Our study shows that the predicted results using the model are close to the measurement results, with a 6.8% error rate.

2.8 Chapter Conclusion

We have examined and developed cache-effective algorithms for both mergesort and quicksort. These algorithms have been tested on four representative processors dating from 1995 to 1999 to show their effectiveness. We also use simulations to provide additional evaluation of performance. We have shown that the memory architecture plays the largest role in affecting the performance of various mergesort algorithms, while the type of data set used affects quicksort algorithms the most.

Our techniques of padding, partitioning, and buffering can also be used for other algorithms for optimizations directed at the cache. Whenever a program regularly accesses a large data set that cannot be entirely stored in the cache, the danger of conflict misses exists, particularly when the algorithm partitions the data sets in sizes that are a power of 2. Padding is effective for this type of program to eliminate or reduce conflict misses. Examples include matrix accesses and manipulations and data reordering and swapping between data sets. When a program sequentially and repeatedly scans a large data set that cannot be stored in the cache in its entirety, the program will suffer capacity cache misses. Partitioning the data set based on the cache size to localize the memory used by a stage in execution is effective for this type of program. Tiling for mergesort is one example where this is used; other tasks where this optimization approach can be used include data accesses by loops and data manipulations of a large data file in a sequential order. The buffering technique is effective to reduce or eliminate conflict misses by using an additional buffer to temporarily hold data elements for later reuse that would otherwise be swapped out of the cache. Examples where this can be employed include programs manipulating data in an

inplaced fashion and programs where data accesses easily cause conflict cache misses.

The only machine-dependent architecture parameters for implementing the four methods we presented in this chapter are the cache size (C), the cache line size (L), cache associativity (K), the number of entries in the TLB cache, and a memory page size (P_s). These parameters are becoming increasingly known to users. They can be defined as variables in the programs, making migration from one platform to another easy for a user. In this way, the programs are easily portable—all that is required is the knowledge of the four required parameters.

There are several ways to provide sorting algorithms with architecture-dependent parameters. One approach leaves the work to an informed user who is familiar with the machine architecture: this user could simply input the required parameters into the programs. A second possibility: users could conduct some brief executions using a runtime library to obtain estimated architectural parameters for the program optimizations. The overhead caused by this approach is normally acceptable [140]. ATLAS [129] uses a tool to first automatically determine architectural parameters by extensive tests on the target machine. The program is then recompiled with these parameters included. A third possibility would be to utilize the ATLAS approach to support our sorting program optimizations, easing the burden on an end-user.

Chapter 3

Load Sharing for Global Memory System Management

3.1 Literature overview on load sharing for global memory in distributed systems

A major performance objective of implementing a load sharing policy in a distributed system is to minimize execution time of each individual job, and to maximize the system throughput by effectively using the distributed resources, such as CPUs, memory modules, and I/Os.

CPU-based Policy:

Most load sharing schemes (e.g., [36], [38], [41], [66], [68], [74], [149]) mainly consider CPU load balancing by assuming each computer node in the system has a sufficient amount of memory space. These schemes have proved to be effective on overall performance improvement of distributed systems. However, with the rapid development of CPU chips and the increasing demand of data accesses in applications, the memory resources in a

distributed system become more and more expensive relative to CPU cycles. We believe that the overheads of data accesses and movement, such as page faults, have grown to the point where the overall performance of distributed systems would be considerably degraded without serious considerations concerning memory resources in the design of load sharing policies. We have following reasons to support our claim. First, with the rapid development of RISC and VLSI technology, the speed of processors has increased dramatically in the past decade. We have seen an increasing gap in speed between processor and memory, and this gap makes performance of application programs on uniprocessor, multiprocessor and distributed systems rely more and more on effective usage of their entire memory hierarchies. In addition, the memory and I/O components have a dominant portion in the total cost of a computer system. Second, the demand for data accesses in applications running on distributed systems has significantly increased accordingly with the rapid establishment of local and wide-area Internet infrastructure. Third, the latency of a memory miss or a page fault is about 1000 times higher than that of a memory hit. Therefore, minimizing page faults through memory load sharing has a great potential to significantly improve overall performance of distributed systems. Finally, it has been shown that memory utilizations among the different nodes in a distributed system are highly unbalanced in practice, where page faults frequently occur in some heavily loaded nodes but a few memory accesses or no memory accesses are requested on some lightly loaded nodes or idle nodes [2].

Memory-based Policy:

Some work has been reported on memory resource considerations of load sharing. Using analytical models, researchers have studied performance impact of memory allocations in scheduling parallel jobs on both shared-memory multiprocessors and distributed memory

systems. (e.g., see [96] and [98]). The impact of the memory demands in parallel scientific workloads on different parallel machines are also studied in [112] and [113]. However, memory demands of workloads on distributed systems could be much higher and more random than many parallel scientific programs running on MPPs. Basically, there are two major approaches to more effectively use global memory resources in a workstation cluster: (1) *job-migration-based load sharing schemes* and (2) *network RAM*. A job-migration-based load sharing system attempts to migrate jobs from a workstation without sufficient memory space to a lightly loaded workstation with large idle memory space for the migrated jobs. When a job migration is necessary, the migration can be either a remote execution (where a job is initiated on a remote workstation), or a preemptive migration which suspends the selected job and moves it to a remote workstation where it is restarted. In a network RAM system [48], if a job cannot find sufficient memory space for its working sets, it will utilize idle memory space from other workstations in the cluster through remote paging. Since accessing remote memory is slower than accessing local memory but much faster than local disk access, the idle global memory space or the network RAM can be considered as another layer between the local memory and the local disk in the memory hierarchy of a workstation.

Regarding network RAM implementations, the Global Memory System (GMS) [44] [124] and the Remote Memory Pager [84] attempts to reduce the page fault overhead by remote paging techniques. Although the page fault cost is reduced, remote paging may also increase network contention. DoDo [2] is designed to improve system throughput by harvesting idle memory space in a distributed system. The owner processes have the highest priority for their CPUs and memory allocations in their nodes, which divides the global memory system into different local regions.

Regarding job-migration-based load sharing systems, an early study in [74] considers using the free memory size in each node as an index for load sharing. Compared with CPU-based policies, this study did not find the memory-based policy particularly effective. This is because the workloads were CPU intensive, and the processors then were much slower than what we are using today. In the MOSIX load sharing system, a memory ushering algorithm is used when the free memory of a node is lower than a certain amount (e.g. 1/4 MBytes) [5]. A preemptive migration is then applied to the smallest running job in the node by moving it to a remote node with the largest free memory. A load sharing policy that only considers memory resource without considering CPU resource is very likely to cause uneven job distributions among workstations, which is not favorable for optimizing the average job queuing time.

We propose **CPU-Memory-based load sharing policies** [133] that will be presented in the next Section. These policies are job-migration-based. They share both CPU and memory services among the nodes in order to minimize both CPU idle times and the number of page faults caused by unbalanced memory allocations of distributed jobs in distributed systems so that overall performance can be significantly improved. The new load sharing policies not only improve performance of memory-bound jobs, but also maintain the same load sharing quality as the CPU-based policies for CPU-bound jobs. The load sharing design is extended on heterogeneous distributed systems [137]. Performance results show that the CPU-based load sharing policy is not robust in a heterogeneous system, and performs poorly for memory-intensive workloads. The performance of the memory-based and CPU-Memory-based load sharing policies are quite independent of system heterogene-

ity changes for memory-intensive workloads and independent on several different memory demand distributions.

Effective usage of global memory resources is an important consideration in the design of load sharing policies for cluster computing. When a workstation does not have sufficient memory space for its assigned jobs, the system will experience a large number of page faults, resulting in long delays for each job. In this case, job-migration-based load sharing approach is not sufficient. Section 3.3 presents how we optimize designs of cluster resource management systems by effectively combining the job-migration-based load sharing system approach and network RAM system approach. We also propose a software method incorporating with dynamic load sharing, which adaptively reserves a small number set of workstations through virtual cluster reconfiguration to provide special services to the jobs demanding large memory allocations. This study can be found in [28].

3.2 CPU-memory-based Load Sharing

Aiming at reducing the memory resource contention caused by page faults and I/O activities, we have developed and examined load sharing policies by considering effective utilization of global memory in addition to CPU load balancing in clusters. Our study consists of two parts: load sharing policies dealing with known memory demands, and with unknown memory workloads.

For the first part of the study, we use real-world application traces obtained from the public domain, which contain average requested and used CPU times, and average requested and used memory space for each job. Relying on the knowledge of memory demands,

we develop several load sharing policies with coordinated utilizations of both CPU and memory resources. Trace-driven simulations are conducted for performance comparison and evaluation. The practical basis of this part is that memory demands of some applications can be known or can be predicted based on users' hints.

Since memory demands of many other applications may not be known in advance or may be hard to predict, and memory accesses and allocations can be dynamically changed, it is highly desirable to develop load sharing schemes with unknown memory demands. We have addressed this issue in the second part of this study. This investigation requires workloads with dynamic memory access and allocation traces. To our knowledge, there have not been workload traces with dynamic memory information available in the public domain. Thus, we have conducted kernel instrumentation to collect application workload execution traces to capture dynamic memory access patterns, and have proposed load sharing schemes dynamically monitoring the jobs status of resource utilizations, and making resource allocation decisions timely and adaptively.

A detailed study with unknown memory demands can be found in [27], in which we present three new results and contributions in the study. (1) Conducting Linux kernel instrumentation, we have collected different types of workload execution traces to quantitatively characterize job interactions, and have modeled page fault behavior as a function of the overloaded memory sizes and the amount of jobs' I/O activities. (2) Based on experimental results and collected dynamic system information, we have built a simulation model that accurately emulates the memory system operations and job migrations with virtual memory considerations. (3) We have proposed a memory-centric load sharing scheme and its variations to effectively process dynamic memory allocation demands, aiming at

minimizing execution time of each individual job by dynamically migrating and remotely submitting jobs to eliminate or reduce page faults and to reduce the queuing time for CPU services. Conducting trace-driven simulations, we have examined these load sharing policies to show their effectiveness.

We focus on presenting the load sharing study with known memory demands in this dissertation, where the performance evaluation methodologies are described, and the performance results are reported. In practice, some jobs' memory demands are known in advance or predictable based on users' hints [7]. In this part of our study, the job's memory demand is assumed to be known, and the memory allocation for this job is done at the arrival of the job. A job's working set size is assumed to be stable during its execution. This part of the work was in collaboration with Yanxia Qu.

3.2.1 CPU-Memory-Based Load Sharing Policies

In a multiprogramming environment, multiple jobs share a node for both its CPU and memory space. There are two types of page replacement policies in a multiprogramming environment: global replacement and local replacement. A global replacement allows the paging system to select a memory page for replacement throughout the memory space of a node. A local replacement requires that the paging system select a page for a job only from its allocated memory space. Most time-sharing operating systems use a global LRU replacement policy. We use node index j to represent one node a cluster, and use variable P to represent the total number of nodes in the cluster. We give the following memory related characterizations in a multiprogramming environment using a global LRU replacement policy on a single node:

- RAM_j : the amount of user available memory space on node j for $j = 1, \dots, P$.
- U_j : the memory usage is the total amount of requested memory space accumulated from jobs the node j . This requested or declared amount of space reflects the maximum amount of the working set, but not the real memory load in executions.
- ML_j : the memory load in bytes is the total amount of memory loads accumulated from running jobs on node j . (After a job is in its stable stage, its working set size should also be stable [116]. We call the memory space for the stable working set the memory load of the job. If $RAM_j > ML_j$, page faults would rarely occur, otherwise, paging would be frequently conducted during the executions of jobs in node j .)
- σ_j : the average page fault rate caused by all jobs on a node is measured by the number of page faults per million instructions when the allocated memory space equals the memory load.

When a job migration is necessary in load sharing, the migration can be either a remote execution, which makes jobs be executed on remote nodes in a non-preemptive way, or a preemptive migration, which may suspend the selected jobs, move them to a remote node, then restart them. We have compared the performance of the remote executions with preemptive migrations for load sharing in a homogeneous environment [145]. Our study indicates that an effective preemptive migration for a memory-intensive workload is not only affected by the workload's lifetime, but also by its data access patterns. Without a thorough understanding of workloads' execution patterns interleaving among the CPU, the memory and the I/O, it is difficult to effectively use preemptive migrations in load sharing policies. For this reason, we have decided to only use the remote execution strategy in this

part. This part focuses on the three policies using remote executions: the first one is based on CPU resource information, the second one uses information on memory usage, and the third one is based on data concerning both CPU and memory resources. Descriptions of the three policies are given as follows.

CPU-based load sharing. The load index in each node is represented by the length of the CPU waiting queue, L_j . A CPU threshold on node j , denoted as CT_j , is the maximum number of jobs the CPU is willing to take, which is set based on the CPU computing capability. For a new arriving job in a node, if the waiting queue is shorter than the CPU threshold ($L_j < CT_j$), the job is executed locally. Otherwise, the load sharing system tries to find the remote node with the shortest waiting queue to remotely execute this job. This policy is denoted as CPU in performance figures.

memory-based load sharing. Instead of using L_j , we propose to use the memory load, ML_j to represent the load index. For a new arriving job, if the memory load is smaller than the user memory space ($ML_j < RAM_j$), the job is executed locally. Otherwise, the load sharing system tries to find the remote node with the lightest memory load to remotely execute this job. This policy is denoted as MEM.

CPU-memory-based load sharing. We have proposed a load index that considers both CPU and memory resources. The basic principle is as follows. When a node has sufficient memory space for both running and requesting jobs, the load sharing decision is made by a CPU-based policy. When the node does not have sufficient memory space for the jobs, the system will experience a large number of page faults, resulting in long delays for each job in the node. In this case, a memory-based policy makes the load sharing decision to either

submit jobs to suitable nodes or to hold the jobs in a waiting queue if necessary.

The load index of node j ($j = 1, \dots, P$) combining the resources of CPU cycles and memory space is defined as

$$Index_{hp}(j)(L_j, ML_j) = \begin{cases} L_j, & ML_j < RAM_j, \\ CT_j, & ML_j \geq RAM_j. \end{cases}$$

When $ML_j < RAM_j$, CPU-based load sharing is used. When $ML_j \geq RAM_j$, the CPU queue length (the load index) is set to CT_j as if the CPU is overloaded so that the system refuses to accept jobs. In our implementation, when $ML_j \geq RAM_j$, the local scheduler immediately searches for the most lightly loaded node in the system as the job's destination.

Since the load index of node j is set to CT_j when $ML_j \geq RAM_j$, it may not allow a node with the overloaded memory to accept additional jobs. This approach attempts to minimize the number of page faults in each node. This load index option is in favor of making each job execute as fast as possible, which is a principle of *high performance computing*. That is the reason we define this option as an high performance computing load index, defined as $Index_{hp}$.

However, this policy may not be in favor of *high throughput computing* which emphasizes on effective management and exploitation of all available nodes. For example, when $ML_j \geq RAM_j$ on one node, this condition may be true in several nodes. If the load indices in many nodes have been set to CT and consequently they may refuse to accept jobs, the amount of node resources accessible to users would be low. For this reason, we design an alternative load index for high-throughput-computing. Instead of aggressively setting the load index to CT_j , we conservatively adjust the load index by a memory utilization status parameter when $ML_j \geq RAM_j$. The memory utilization parameter is defined as $\gamma_j = \frac{U_j}{RAM_j}$. When

$\gamma_j < 1$, the memory space of the node is sufficiently large for jobs. When $\gamma_j > 1$, the memory system is overloaded. This option is designed for high throughput computing, and its load index is defined as follows:

$$Index_{ht}(j)(L_j, ML_j) = \begin{cases} L_j, & ML_j < RAM_j, \\ L_j \times \gamma_j, & ML_j \geq RAM_j. \end{cases}$$

Memory utilization parameter γ_j is used to proportionally adjust the load index. When $ML_j \geq RAM_j$, the CPU queue length is enlarged by a factor of γ_j as if the CPU were increasingly loaded. The increase of the load index would reduce the chance of this node being selected soon for a new job assignment.

Both load index options have their merits and limits, and they are workload and system dependent. The load sharing policy based on above two load indices can be expressed as follows:

$$LS(Index(j)) = \begin{cases} \text{local execution.} & Index(j) < CT_j, \\ \text{remote execution.} & Index(j) \geq CT_j. \end{cases}$$

where $Index$ is either $Index_{hp}$ or $Index_{ht}$. This policy is denoted as CPU_MEM_HP or CPU_MEM_HT.

3.2.2 Performance Evaluation Methodology

Our performance evaluation is simulation-based, consisting of two major components: a simulated cluster and workloads ¹.

¹The simulator can be accessed at <http://www.cs.wm.edu/hpcs/WWW/HTML/publications/abs00-1.html>.

3.2.2.1 A simulated cluster

Workstations in a cluster could be heterogeneous with different CPU powers and memory capacities. In a heterogeneous system, load indices of a node can be adjusted based on the node's relative computing capability and memory capacity in this system [137]. In order to simplify the description, We focus on presenting our study on a homogeneous system and give a brief summary of our study on heterogeneous system at the end of Section 3.2.

We simulated a homogeneous cluster with 32 nodes, where each local scheduler holds all the load-sharing policies we just discussed: CPU-based, Memory-based, CPU-Memory-based and their variations. The simulated system is configured with workstations of 800 MHZ CPUs and 1GBytes Memory each. The memory page size is 4Kbytes. The Ethernet connection is 100Mbps. Each page fault service time is 10 *ms*, and the context switch time is 0.1 *ms*. The overhead of a remote execution is 0.05 second.

The widening speed gap between CPU and memory makes memory accesses and page faults increasingly expensive. Using SPEC CPU 1995 and SPEC CPU 2000 benchmark programs, and execution-driven simulations of modern computer architectures, researchers have quantitatively evaluated their execution time portions for CPU operations and memory accesses [79] and [148]. For example, using the SPEC CPU 2000 benchmarks on a simulated 1.6 GHz, 4-way issue, out-of-order core with 64 KB split L1 caches, a 1 MB on-chip L2 cache, and an infinitely large main memory, in average, the system spends 57% total execution time serving memory accesses (L2 misses), 12% of its time serving L1 misses, and only 31% of its time for CPU operations. If we consider a small percentage of the memory accesses experiences page faults in a system with limited main memory space, the percentage of the

execution time spent for CPU operations can be significantly low. Our cluster simulation environment is consistent with reported results.

The CPU local scheduling uses the round-robin policy. Each job is in one of the following states: “ready”, “execution”, “paging”, “data transferring”, or “finish”. When a page fault happens in the middle of a job execution, the job is suspended from the CPU during the paging service. The CPU service is switched to a different job. When page faults happen in executions of several jobs, they will be served in FIFO order. The page faults in each node are simulated as follows. When the memory load of jobs in a node is equal to or larger than the available memory space ($ML_j \geq RAM_j$), each job in the node will cause page faults at a given page fault rate, $\sigma_j \times \frac{ML_j^i}{MA_j^i}$, where ML_j^i is the memory load of job i in node j , and MA_j^i is the allocated memory space for job i in node j .

In practice, application jobs have page fault rates from 1 to 10.

3.2.2.2 Workload Traces

We select a workload from the Los Alamos National Lab, which contains detailed information about resource requests and usage including memory. This workload was collected from a 1024-node Connection Machine CM-5 during October 1994 through September 1996. This workload can be downloaded from Feitelson’s Workload Archive [45]. We extract 4 traces from this workload, which are summarized in Table 3.1. Trace “MAY”, “JUNE”, “JULY” and “AUGUST” include jobs submitted in May 1996, June 1996, July 1996 and August 1996, respectively. The parallel workloads have been converted to sequential workloads by accumulating CPU and memory demands of all parallel tasks of each job to a sequential job. Each job in our trace has 4 items: (1) arrival time, (2) arrival node, (3) requested memory

size, and (4) requested CPU time. Item 1 can be obtained from the original CM-5 workload directly. Item 3 and 4 are the total amount of requested CPU time and memory size of a job. Item 2 is assigned to a node whose number is the same as the job's submission date. For example, if a job is submitted on May 16, 1996, this job is assumed to be submitted to node 16. We specially assign jobs to node 31 and/or 32 as follows. Node 32 in trace "MAY", "JULY" and "AUGUST" contains all jobs submitted on June 1, 1996, August 1, 1996 and September 1, 1996, respectively. Node 31 and node 32 in trace "June" include all jobs submitted on June 1, 1996 and June 2, 1996, respectively. We converted the job duration time into Million Instructions according to the CPU speed.

trace name	duration	# jobs	avg. CPU demand	avg. memory demand
MAY	May. 1996: June 1, 1996	4177	10166236 MIPS	1006 MB
JUNE	June. 1996: July 1-2, 1996	3738	9783912 MIPS	735 MB
JULY	July. 1996: Aug. 1, 1996	8639	5121149 MIPS	552 MB
AUGUST	Aug.. 1996: Sept. 1, 1996	3209	11428627 MIPS	901 MB

Table 3.1: Trace Description

3.2.2.3 System conditions

We have following conditions and assumptions for evaluating the load sharing policies in the cluster:

- Each node maintains a global load index file which contains both CPU and memory load status information of other nodes. The load sharing system periodically collects and distributes the load information among all nodes.
- The location policy determines which node to be selected for a job execution. The

policy we use is to find the most lightly loaded node in the cluster.

- Similar to assumptions in [59] and [124], we assume that page faults are uniformly distributed during job executions.
- We assume that the memory load of a job is 40% of its requested memory size. The practical value of this assumption has also been confirmed by the studies in [59] and [124].

3.2.3 Performance Results and Analysis

Slowdown is the ratio between the wall clock execution time and the CPU execution time of a job. A major timing measurement we have used is the mean slowdown, which is the average of each program's slowdown in a trace. In the rest of the chapter, "slowdown" means the "mean slowdown". Major contributions to the slowdown come from the delays of page faults, waiting time for CPU service, and the overhead of remote execution. The mean slowdown measurement can determine the overall performance of a load sharing policy, but may not be sufficient to provide performance insights. We have also looked into the total execution time and its breakdowns. For a given workload scheduled by a load sharing policy (or without load sharing), we have measured the total execution time. The execution time is further broken into CPU service time, queuing time, paging time, and migration time.

3.2.3.1 Overall Performance Comparisons

We have experimentally evaluated the 4 load sharing policies, and present performance comparisons of all the traces. Figure 3.1 and 3.2 present the mean slowdowns of 4 traces scheduled by different load sharing policies. The average memory demand of a job is known

in advance, but memory access interactions of multiple running jobs are unknown. We use different page fault rates to characterize different interactions. Intensive interactions mean that memory accesses of multiple running jobs happened at the same time, which could cause more page faults than those in less intensive interactions. Before getting into details, we present two general observations based on the results in the figures. First, the slowdown are proportionally increased as the page fault rate increases. Second, when average page fault rates are low, the performance differences among the load sharing policies are insignificant. However, when average page fault rates are high, the CPU-Memory based load sharing policies significantly outperform both CPU-based and Memory-based policies.

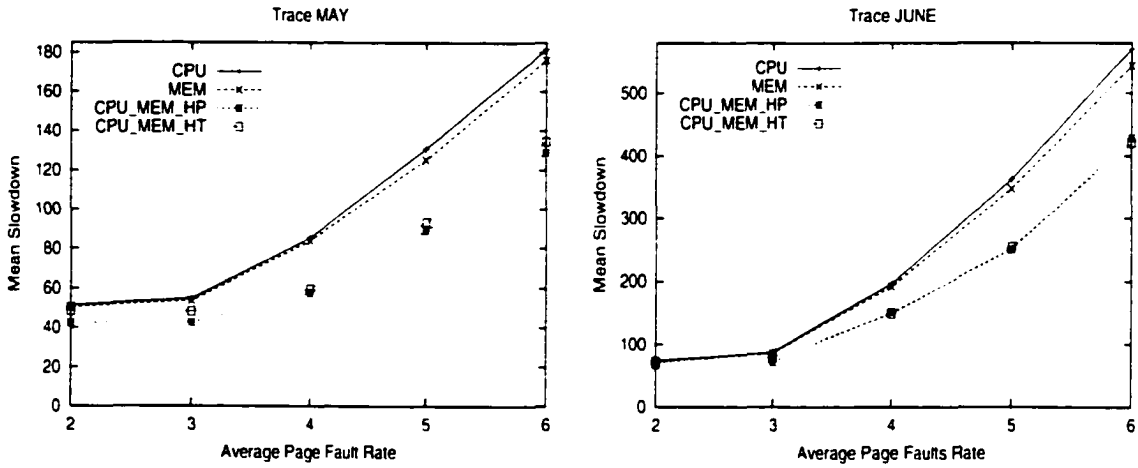


Figure 3.1: Mean slowdowns of the 4 load sharing policies as the page fault rate increases on traces MAY and JUNE.

Policy CPU does reasonably well when the page fault rate is low, but does poorly when the rate is high. Policy MEM performs slightly better than CPU, but it still far below the performance of CPU-MEM based policies. Policies CPU-MEM-HP and CPU-MEM-HT perform well under all conditions, and do show their effectiveness. Here is an example on trace AUGUST. When the page fault rate is 4, the slowdowns of the last three policies

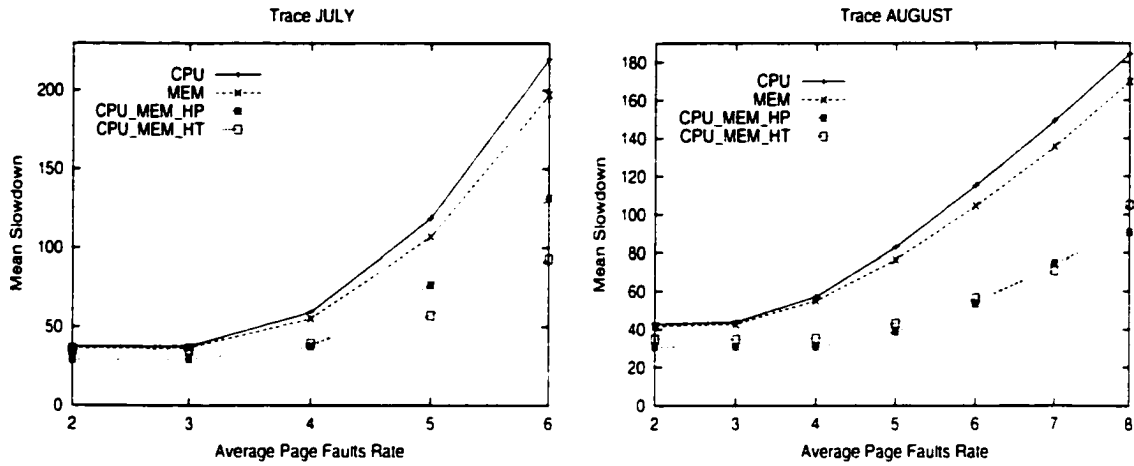


Figure 3.2: Mean slowdowns of the 4 load sharing policies as the page fault rate increases on traces JULY and AUGUST.

are about 1.04 times lower, 1.84 times lower, 1.62 times lower than that of CPU policy, respectively. When the page fault rate is increased to 8, the slowdowns of these three policies are about 1.09 times lower, 2.03 times lower, 1.75 times lower than that of CPU policy, respectively.

3.2.3.2 Paging and Queuing

Our simulator also records execution breakdowns. Our experiments confirm that in different load sharing policies the CPU service time is not changed. The migration time spending on remote execution is neglected. So paging time and queuing time become the major parts to evaluate performance of load sharing policies. Figure 3.3 presents the paging time reduction and queuing time reduction of policies MEM, CPU_MEM_HP and CPU_MEM_HT over policy CPU for different traces when the average page fault rate is 6.

Surprisingly, the paging time reduction of policy MEM is very small. This is because policy MEM does not consider CPU load balancing at all so that some nodes may hold

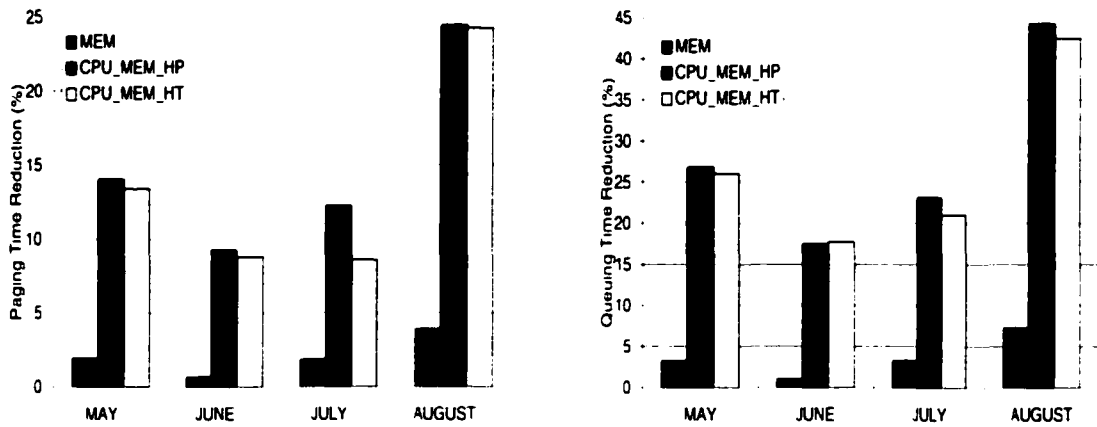


Figure 3.3: Paging time reduction (left figure) and queuing time reduction (right figure) of policies MEM, CPU_MEM_HP and CPU_MEM_HT over policy CPU.

a large number of running jobs. However, these nodes could be viewed as lightly loaded because idle memory may still be available there. The heavy CPU load tends to make these running jobs stay longer, and cause more page faults in these nodes when more jobs have to move in, which offsets the page fault reduction gained from other nodes holding fewer running jobs. In contrast, the paging time reductions of policies CPU_MEM_HP and CPU_MEM_HT are significant. For example, in trace AUGUST, the paging time reduction of policy MEM is only 3.88%. The reductions of CPU_MEM_HP and CPU_MEM_HT are 24.43% and 24.27%, respectively.

Queuing time reductions for different policies follow the same trend. The reduction of MEM is very small. On one hand, some nodes hold a small number of running jobs with large memory demands. The queuing time could be significantly reduced in these nodes. On the other hand, it's very likely that a large number of jobs running in the same node in a time-sharing mode because these jobs demand small memory space. The queuing time in these nodes significantly increases. The right figure of Figure 3.3 clearly shows that the

two parts are comparable so that the queuing time reduction of MEM is only modest. The queuing time reductions of Policies CPU_MEM_HP and CPU_MEM_HT balancing both CPU and memory loads are very effective. For example, In trace AUGUST when page fault rate is 6, the queuing time reduction of MEM is 7.25%, while reductions of CPU_MEM_HP and CPU_MEM_HT are 44.16% and 42.43%, respectively.

3.2.3.3 High Performance and High Throughput

We have further compared the high performance (HP) approach and the high throughput (HT) approach in our load sharing policies (see Figure 3.1 and 3.2). Generally, the high performance approach is comparable with, but is slightly more effective than, the high throughput approach for all cases. This is because the high throughput approach tends to encourage more jobs to be executed in a time-sharing mode in a cluster so that it could cause slightly more page faults compared with the high performance approach. Occasionally, the high throughput approach outperforms the high performance approach. A cluster managed by CPU_MEM_HP refuses to accept jobs when either CPU or memory is overloaded. This approach attempts to make each running job execute as fast as possible. But if many jobs are refused or some jobs are delayed for a very long period of time, the overall performance could be affected. In these cases, the high throughput approach can outperform the high performance approach. For example (see Figure 3.1 and 3.2), the performance results in trace JULY with page fault rate of 5 and 6, in trace JUNE with page fault rate of 4, and in trace AUGUST with page fault rate of 7 give such examples.

3.2.4 Summary

We summarize our study on load sharing with known job memory demands as follows.

- The performance of a load sharing policy considering both CPU or memory resources is robust for all traces in this part of the study, and is much better than the performance of a load sharing policy considering only CPU or only memory resource, particularly when the memory access interactions are intensive.
- The reason that CPU-MEM-based policies perform well is that these policies effectively reduce the paging time and queuing time. Meanwhile, CPU policy suffers large paging overhead, and MEM policy could not reduce queuing time.
- The high performance approach is slightly more effective than the high throughput approach for all traces in this part.

3.2.5 Brief description of our study on heterogeneous systems

Practical systems are often heterogeneous with a large variation in the computing power and memory capacities of different workstations. We have designed and evaluated load sharing policies by considering both system heterogeneity and effective usage of CPU and memory resources. The detailed study can be found in [137]. We first present how we characterize heterogeneity, then give a brief summary of our study on Heterogeneous Systems in this Subsection.

3.2.5.1 CPU/Memory Weights and Heterogeneity

In this study, heterogeneity only refers to the variations of CPU powers and memory capacities, but not the variations of operating systems, network interfaces and hardware organizations among the workstations. In this section, we quantitatively characterize heterogeneous CPU powers and memory capacities in a network of workstations. The simple models to be discussed here have been used in the designs and evaluation of load sharing policies. We use node index j to represent one of the nodes in a heterogeneous network of workstations. We also use variable P to represent the total number of nodes in the system.

The CPU weight of a workstation refers to its computing capability relative to the fastest workstation in a distributed system. The value of the CPU weight is less than or equal to 1. Since the CPU weight is a relative ratio, it can also be represented by the CPU speed of each node measured by millions of instructions per second (MIPS). If $V_{cpu}(j)$ is the speed of workstation M_j in MIPS, $j = 1, \dots, P$, the CPU weight can be expressed as follows:

$$W_{cpu}(j) = \frac{V_{cpu}(j)}{\max_{i=1}^P V_{cpu}(i)}. \quad (3.1)$$

The total CPU power of a system is defined as the sum of the CPU speeds of each workstation, which represents the accumulated computing capability of the system:

$$TP_{cpu} = \sum_{j=1}^P V_{cpu}(j). \quad (3.2)$$

Similarly, the memory weight is calculated by comparing the memory sizes among the

computing nodes:

$$W_{mem}(j) = \frac{RAM_j}{\max_{i=1}^P RAM_i}. \quad (3.3)$$

where RAM_j is the amount of user available memory space on node j for $j = 1, \dots, P$.

The total memory capacity of a system is defined as

$$TP_{mem} = \sum_{j=1}^P MS_j. \quad (3.4)$$

where MS_j is the memory size of node j .

The system heterogeneity can be quantified as the variance of computing powers and memory capacities among the workstations. Using standard deviation and CPU weights, we define the CPU heterogeneity as follows:

$$H_{cpu} = \sqrt{\frac{\sum_{j=1}^P (\bar{W}_{cpu} - W_{cpu}(j))^2}{P}}. \quad (3.5)$$

where $\bar{W}_{cpu} = \frac{\sum_{j=1}^P W_{cpu}(j)}{P}$, is the average CPU weight. Similarly, we define memory heterogeneity as follows:

$$H_{mem} = \sqrt{\frac{\sum_{j=1}^P (\bar{W}_{mem} - W_{mem}(j))^2}{P}}. \quad (3.6)$$

where $\bar{W}_{mem} = \frac{\sum_{j=1}^P W_{mem}(j)}{P}$, is the average memory weight in the system. Higher values of H_{cpu} and H_{mem} in a distributed system correspond with a higher variation in the CPU capability and memory capacity among different nodes. A homogeneous system is characterized by $H_{cpu} = H_{mem} = 0$.

3.2.5.2 Summary of Our Heterogeneous Study

We have experimentally examined and compared CPU-based, memory-based and CPU-Memory-based load sharing policies on heterogeneous networks of workstations. Based on our experiments and analysis we have following observations and conclusions:

- The CPU and memory weights of workstations can be effectively used to characterize heterogeneity of a distributed system for designs of load sharing policies. For given total CPU power and total memory capacity, we can have different homogeneous and heterogeneous configurations with a roughly equivalent purchase cost. Under such a condition, the performance evaluation and comparisons are meaningful and useful.
- The CPU-based load sharing policy is not robust in a heterogeneous system, and performs poorly for memory-intensive workloads.
- The performance of the memory-based and CPU-Memory-based load sharing policies are quite independent of system heterogeneity changes for memory-intensive workloads. This is because the job migrations considering both memory and CPU resources offset the negative effects of the system heterogeneity. As the system heterogeneity increases to a certain degree, the remote executions and page faults also increase proportionally for the two policies, resulting a moderate degradation of the performance. However, our experiments also show that changes of the heterogeneity do not affect the functionality and nature of the two policies.
- An initial job pool assignment which uses information regarding system heterogeneity can allocate system resources effectively.

- We also show that the CPU-based, memory-based and CPU-Memory-based load sharing policies are independent on several different memory demand distributions.

3.3 Incorporation job migration and network RAM to share memory resource

3.3.1 Objectives of the study

The job-migration-based load sharing system approach and the network RAM system approach share the same objective of reducing page faults in each local workstation. The two approaches have another common technical feature in their implementations. Both systems maintain a global load index record for each workstation about how its CPU and/or memory resources are being utilized. This record is either stored in a master workstation or distributed among the workstations, and is updated periodically by the cluster workstations.

There are several major differences between the two approaches in the ways that the global memory resources are shared. Because of these differences, each approach has its own merits and limits. First, in a network RAM cluster system, a workstation is provided with a huge global memory space for its jobs. The global memory space could be even larger than its local disk space. Thus, it is possible to eliminate accesses to local disks due to page faults in a network RAM cluster. In contrast, memory allocations of a job could be limited by the local memory size of a workstation in a migration-based load sharing cluster system where local memory modules are not shared by other workstations. A network RAM cluster system could be more beneficial to large or non-migratable data-intensive jobs than a migration-based load sharing cluster system. Second, the effectiveness of global paging operations in

a network RAM cluster system is heavily dependent the cluster network speed. In contrast, the network, in general, is less frequently used in a remote-execution-based load sharing cluster system. In other words, a remote-execution-based load sharing system relies less on network speed than a network RAM system. Finally, a migration-based load sharing system is able to balance the workloads among workstations by sharing both CPU and memory resources, while a network RAM system only considers global memory resources for load sharing. Without job migrations, job executions may not be evenly distributed in a cluster -- some workstations can be more heavily loaded than others. Although the lightly loaded workstations in a network RAM cluster system can be used as memory servers for heavily loaded workstations, their CPU resources are not fully utilized by the cluster.

Conducting trace-driven simulations, we have compared the performance and trade-offs of job-migration-based load sharing policies and Network RAM, and their impact on job execution time and cluster scalability. In this study, we quantitatively address the following three questions: (1) Under what cluster and workload conditions is a migration-based load sharing policy or the network RAM beneficial for performance improvement? (2) What are the performance effects of limited network bandwidths and cluster size to the two system approaches? (3) How do we optimize designs of cluster resource management systems by effectively integrating and combining the two system approaches?

The rest of the section is organized as follows. We describe the job-migration-based load sharing policies and the network RAM implementations we have used in this study in Section 3.3.2. We present the performance evaluation methodology and experimental environments in Section 3.3.3. The performance comparisons and analyses are presented in Section 3.3.4. We propose an improved load sharing policy supported by network RAM in

Section 3.3.5. Finally, we summarize the work in Subsection 3.3.6.

3.3.2 Job-migration-based load sharing vs. network RAM

Network RAM and job-migration-based load sharing related operations on workstation j ,

for $j = 1, \dots, P$, are characterized by the following variables:

- RP_j : the amount of remote paging in Mbytes from the workstation.
- FM_j : the idle memory space in Mbytes of the workstation.

3.3.2.1 Network RAM organizations

A network RAM organization makes each workstation not only have its own local memory, but also be able to access idle memory space of other workstations through remote paging in a cluster. The memory allocation decision for a job on workstation j is made by

$$\text{memory allocation} = \begin{cases} \text{local memory} & \text{if } ML_j < RAM_j \\ \text{global memory} & \text{if } ML_j \geq RAM_j. \end{cases}$$

where the global memory allocation is implemented by finding the most lightly loaded workstation one by one for remote paging based on the following search algorithm:

Allocate the idle local memory space to the arrival job:
 $MD_j = ML_j$;
While ($MD_j \geq RAM_j$) and (idle memory space is available elsewhere)
 do
 find node i with the largest idle memory space among $P - 1$ nodes (excluding node j):
 allocate $RP_i = \min\{MD_j - RAM_j, FM_i\}$ MBytes from node i to the job in node j :
 $FM_i = FM_i - RP_i$;
 $MD_j = MD_j - RP_i$;
 end do

where MD_j represents the current local memory demand on workstation j . The while loop continues until the memory demand is met or no idle memory available in the system.

If $ML_j \geq RAM_j$ after the global allocations, disk accesses due to page faults will occur in workstation j . In order to minimize the global paging, we give local memory accesses the highest priority. The global paging is only conducted when the remote workstation has additional idle memory space. Therefore, when a new local job arrives, the network RAM paging services for remote jobs will be transferred to other workstations if any memory space occupied by remote pages is needed for this new job.

3.3.2.2 CPU-Memory-based load sharing

The job-migration-based load policy we have selected for this comparative study is the CPU-Memory-based load sharing scheme previously described, which makes a job migration decision by considering both CPU and memory resources. The basic principle of this scheme is as follows. When a workstation has sufficient memory space for both running and requesting jobs ($ML_j < RAM_j$), the load sharing decision is made by a CPU-based policy where the load index in each workstation is represented by the length of the CPU waiting queue. As long as the CPU waiting queue is not larger than the threshold which is set based on the CPU capability, the requesting jobs will be locally executed in the workstation. Otherwise, the load sharing system finds the remote workstation with the shortest waiting queue to either remotely execute this job or to preemptively migrate an eligible job from the local workstation to the remote workstation. When the workstation does not have sufficient memory space for the jobs ($ML_j \geq RAM_j$), the load sharing scheme attempts to migrate jobs to suitable workstations or even to hold the jobs in a waiting pool if necessary. Again, the migration can be either remote execution or preemptive migration.

During an execution of a memory-intensive job, page faults may occur periodically. Each

such period is called a *transition*, where page faults are conducted to bring a working set into memory. The data references will then be memory hits for a while before the working set changes and page faults are conducted in the next transition period. The local reference period is called a *phase*. If the phases of a job are disjoint, or almost disjoint, the best time to do a preemptive migration is at the end of a phase and before starting another transition period for a new working set. The migrated job would carry no data or a small data set to a remote workstation. However, in practice, it may be difficult to predict the data access phase and transition patterns of so many different jobs. If this prediction is impossible, remote executions should be a practically optimal solution for load sharing of memory-intensive jobs [145]. For this reason, remote executions are used in our CPU-Memory-based load sharing policy.

3.3.3 Performance Evaluation Methodology

Our performance evaluation is simulation-based. We discuss performance evaluation metrics, the simulation model, and the workloads in this subsection.

3.3.3.1 Performance metrics

For a given workload scheduled by a job-migration-based load sharing policy, supported by network RAM, or without load sharing and network RAM, we target evaluating and comparing their performance merits and limits under various system and workload conditions.

The following performance metrics are used in our evaluation:

- *average execution time per job* is defined as $\bar{t} = \frac{\sum_{i=1}^n t_i}{n}$, where t_i is the measured execution time of an individual job, and n is the number of jobs in a given workload.

- *execution time breakdowns*: The average execution time is further broken into CPU service time, queuing time, disk access time due to page faults, and networking time for job migrations or remote pagings including network contention time.

3.3.3.2 A simulated workstation cluster

We have developed a simulator for a bus-based workstation cluster that has multiple functions by: (1) supporting different job-migration-based load sharing policies including the CPU-Memory-based policy, (2) simulating a remote paging system for a network RAM in the cluster, (3) simulating bus contention, and (4) having system heterogeneity. The simulated cluster is scalable and is configured by 6 to 18 workstations of 300 MHz CPUs with local memory of 128 MBytes. The cluster network is an Ethernet bus of 10 Mbps and 100 Mbps. Each disk access time due to a page fault is 10 *ms*. The size of a memory page is 4 KBytes. The CPU local scheduling uses the round-robin policy.

When a page fault happens during job execution, the job is suspended from the CPU during the paging service. The CPU service is switched to a different job. When page faults happen in the executions of several jobs, they will be served in FIFO order. The overhead of a remote execution is 0.1 second.

The bus service and contention are simulated as follows. Each workstation is given a sequence number, which also represents its priority rank to access the bus. The priority increases as the sequence number decreases. As multiple requests for bus services arrive in sequence, the requests will be served in FIFO order. If the requests arrive simultaneously, they will be served in an order based on their workstations' bus access priorities.

3.3.3.3 Workloads

The workloads we have used are the 8 traces from [66]. Each trace was collected from one workstation on different daytime intervals. The jobs in each trace were distributed among 6 homogeneous workstations. Each job has the following three major parameters:

`< arrival time. arrival workstation. duration time >`

The 8 traces have different inter-arrival distributions and different Pareto service time distributions.

We have made the following modifications of the traces for our study. We converted the job duration time into Million Instructions according to CPU speed. The memory demand of each job in the traces is generated from a Pareto distribution with the mean sizes of 1 MBytes. Each job has the following 4 items:

`< arrival time. arrival workstation. requested memory size. duration time >`

The number of jobs is doubled and tripled in each trace as the number of workstations is scaled from 6 to 12, and scaled from 12 to 18, respectively.

For the job-migration-based load sharing system, the page faults in each workstation are conducted in our simulation as follows. When the memory load of jobs in a workstation is equal to or larger than the available memory space ($ML_j \geq RAM_j$), each job in the workstation will cause page faults at a page fault rate that is proportional to the memory usage of this workstation. In practice, application jobs have page fault rates from 1 to 10 per million instructions. We set the rate in the same range in our experiments.

For the network RAM system, when $ML_j \geq RAM_j$ in a workstation, remote paging is conducted as described in Subsection 3.3.2.1. The remote paging rate of a job is proportional

to the size of the global memory space allocated to this job. If the aggregate global memory space in the cluster is not sufficient for the job, the job in the workstation will cause page faults to access the local disk at a page fault rate.

3.3.4 Simulation Results and Analysis

Our performance evaluation targets understanding the effects of network bandwidth changes to both the job-migration-based load sharing scheme and network RAM supported by remote paging. We have also quantitatively evaluated two performance trade-offs for comparing the two schemes: (1) the trade-off between reducing local disk accesses due to page faults and increasing network contention and delay due to remote paging; (2) the trade-off between reducing more local disk accesses by network RAM and balancing job execution among workstations by job migrations.

3.3.4.1 Impact of limited network bandwidths

It is widely known that CPU speeds are increasing much more rapidly than network speeds. For example, the 10 Mbits per second (Mbps) Ethernet has been a common network infrastructure for many years, while the CPU speed of workstations/PCs connected to networks of this speed has been updating every year. Both job migrations and remote paging rely on the cluster network for data transfers. However, the performance of each scheme is affected differently by changes of the network speed.

Figure 3.4 presents the average execution times per job (the left figure) and the network contention portions in the execution times (right figure) of “trace 0” running on clusters of 6, 12 and 18 workstations, where the jobs are executed without load sharing (denoted

as “Base”), scheduled by CPU-Memory-based load sharing policy with remote executions (denoted as “LS_RE”), and executed on a network RAM system (denoted as “Net_RAM”). The bus speed varies from 10 Mbps to 100 Mbps. The mean memory demand of jobs is 1 MBytes.

The page fault rates was set to 5.96 per million instructions for all the experiments on “trace 0”. Since the number of jobs proportionally increases as the number of workstations increases in the cluster, the average execution times per job of “trace 0” by “Base” are identical on clusters of 6, 12, and 18 workstations. Using the same page fault rate, we conducted the experiments to compare the execution time performance between “LS_RE” and “Net_RAM” for the same workload of “trace 0”.

We have the following observations based on the experimental results in Figure 3.4. First, the performance of “LS_RE” is not significantly affected as the cluster is scaled from 6 to 12, and from 12 to 18 workstations. The performance improves only slightly as the bus speed increases from 10 Mbps to 100 Mbps. This is because data communication via the network by remote executions is a small portion in the total execution time (0% to 0.005%, see the right figure in Figure 3.4). Second, the performance of “Net_RAM” supported by remote paging is highly sensitive to the network speed and the number of workstations in the cluster. For example, the average execution time of “Net_RAM” is 46% lower than that of “LS_RE” on the cluster of 6 workstations where the bus speed is 10 Mbps. As the bus speed increases to 100 Mbps, the average execution time of “Net_RAM” is further reduced 45%. However, as the cluster of 10 Mbps increases to 12 workstations, the average execution time of “Net_RAM” sharply increases (about 3.6 times higher than that of “LS_RE”). As the cluster speed increases to 100 Mbps, the execution time is significantly reduced, and is

69% lower than that of “LS_RE”. Similar performance data are collected as the number of workstations increases to 18. Our experiments show that the cluster scalability and workload performance when using network RAM are highly dependent on the speed of the cluster, because the network latency due to data transfer and contention is a significant portion in the total execution time (0.05% to 46.03%, see the right figure in Figure 3.4). Finally, in the workload of “trace 0”, some jobs are marked as non-migratable. Therefore, the power and benefits of job migrations may be limited.

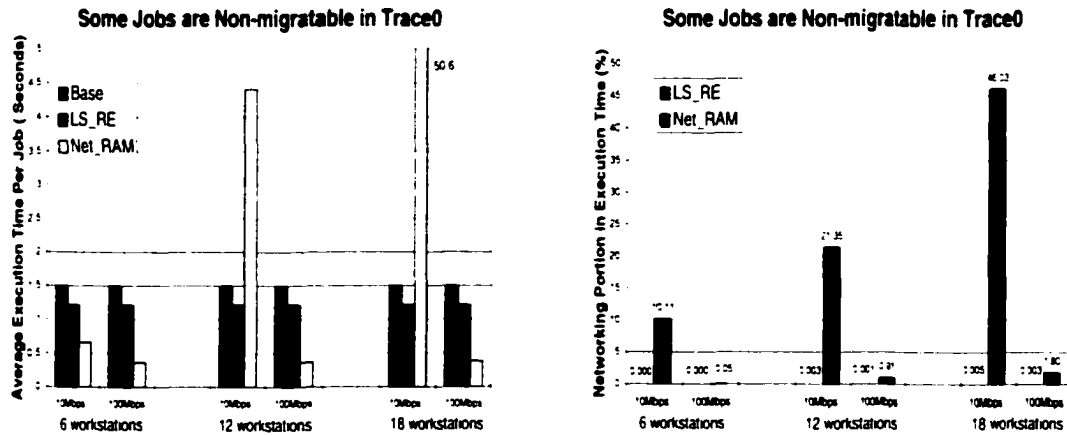


Figure 3.4: The average execution times per job (the left figure) and the networking portions in the execution times (right figure) of “trace 0” with job migration restrictions running on clusters of 6, 12 and 18 workstations.

In order to fully take advantage of job migrations, we released the restrictions on the non-migratable jobs so that remote executions can be applied to all the jobs in “trace 0”. Figure 3.5 presents the average execution time per job (left figure) and the networking portions in the execution times (right figure) of the modified “trace 0” scheduled by “LS_RE” in comparisons with “Base” and “NET_RAM” on the clusters of 6, 12 and 18 workstations.

We show that the performance of “LS_RE” is significantly improved. The execution times of “LS_RE” using a 10 Mbps cluster are slightly lower than the execution times of “Net_RAM” using 100 Mbps clusters of 6, 12, and 18 workstations. In this case, the remote-execution-based load sharing policy not only outperforms the network RAM, but is also more cost-effective.

From the scalability point of view, “LS_RE” demands less network bandwidth in order to scale the cluster by connecting more workstations than “Net_RAM” does. For example, “LS_RE” is scalable from 6 to 18 workstations for both 10 and 100 Mbps buses, while “Net_RAM” is only scalable for the 100 Mbps bus.

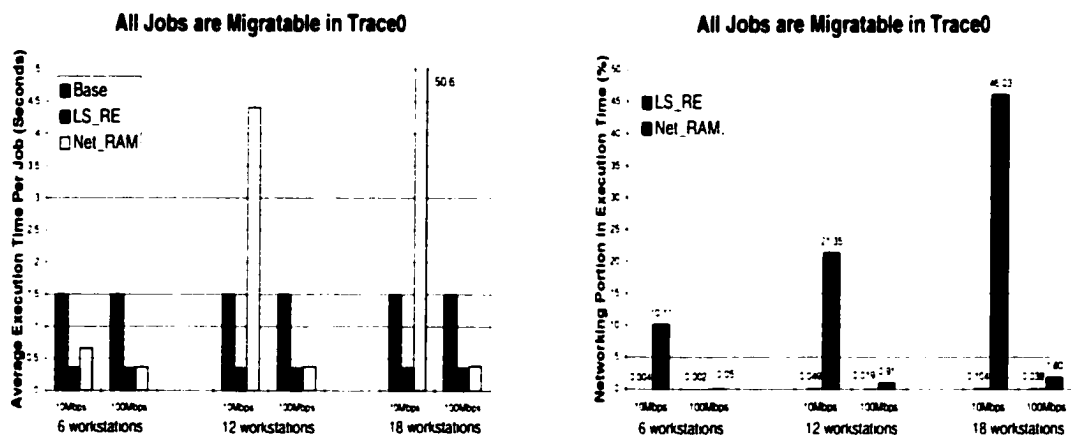


Figure 3.5: The average execution times per job (the left figure) and the networking portions in the execution times (right figure) of “trace 0” without job migration restrictions running on clusters of 6, 12 and 18 workstations.

3.3.4.2 Trade-offs between page fault reductions and load sharing

Page faults in the network RAM are reduced at the cost of additional network contention and delay. Although page fault reductions may be limited by the remote-execution-based load sharing scheme for large data-intensive jobs, the scheme requires less additional network support compared with the network RAM. In order to provide insights into the trade-offs between the two schemes, we present the execution time breakdowns of “trace 0” where all jobs are migratable in Figures 3.6 and 3.7. The execution time of a workload consists of “CPU”, “networking”, “page faults”, and “queuing” portions. “CPU” is the execution time by the CPU for the workload. “Networking” is the time spent on the cluster network, which is used for remote pagings by the network RAM, or for remote executions by the load sharing scheme (including network contention time). “Page faults” is the local disk delay time for both schemes. “Queuing” is the average waiting time for a job to be executed on a workstation.

When the workload is executed on a 10 Mbps cluster of 6 and 12 workstations, the networking time for remote pagings by the network RAM is one of the major portions in the execution time. For example, the networking times contribute 15.5%, and 23.08% to the execution times on the 6 workstation cluster, and the 12 workstation cluster (see the left figures in Figures 3.6 and 3.7), respectively. In contrast, the networking time for remote executions by the load sharing scheme is insignificant (0.06% and 0.11%). Consequently, the queuing time for each job in the network RAM is significantly increased by networking delay, causing much longer execution times than for the remote-execution-based load sharing scheme.

We have also shown that the networking time portions in the executions of the workload by the network RAM are significantly reduced by increasing the bus speed from 10 Mbps to 100 Mbps. Consequently, the queuing time for each job is also significantly reduced (see the right figures in Figures 3.6 and 3.7).

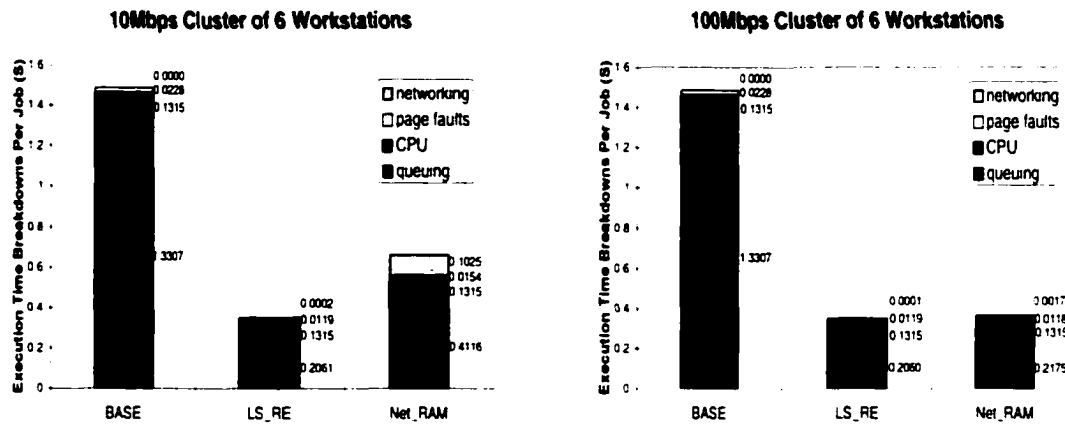


Figure 3.6: The average execution times per job of “trace 0” without job migration restrictions running on a 10 Mbps cluster (the left figure), and a 100 Mbps cluster (the right figure) of 6 workstations.

Another trade-off of the two schemes is between page fault reductions and load sharing. Without job migrations, job executions may not be evenly distributed among the workstations by the network RAM although page faults can be significantly reduced through remote pagings. The unbalanced loads among workstations in network RAM is another reason for the long queuing times for the workload executed on the 10 Mbps clusters of 6 and 12 workstations.

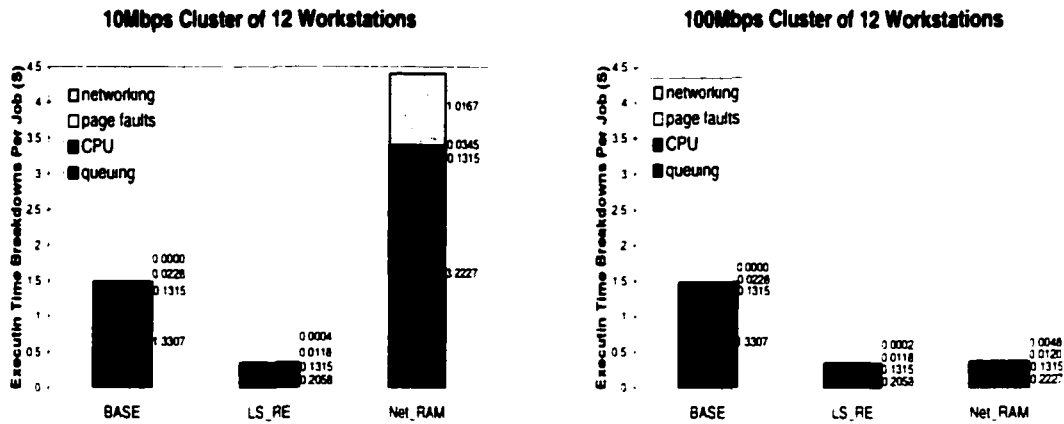


Figure 3.7: The average execution times per job of “trace 0” without job migration restrictions running on a 10 Mbps cluster (the left figure), and a 100 Mbps cluster (the right figure) of 12 workstations.

3.3.5 An improved load sharing scheme

Our experiments show advantages and limits of the network RAM and the remote-execution-based load sharing scheme. A natural optimization step for overcoming the limits of each scheme is to combine them. Here is the basic idea of this improved load sharing scheme. When a workstation has sufficient space for both current and requesting jobs, the job execution location will be determined by the CPU-based policy. When a workstation runs out of memory space for both current and requesting jobs, we first attempt to migrate the new arrival job to the most lightly loaded workstation. If the workstation does not have sufficient memory space for the job, the network RAM will be used to satisfy the memory allocation of the job through remote paging. The memory allocation combining both remote executions and network RAM of the scheme is outlined as follows:

```

If ( $ML_j \geq RAM_j$ )
  find workstation  $i$  with the largest idle memory space among  $P$  workstations:
  If  $i \neq j$ 
    remotely execute the job at workstation  $i$ :
  If ( $ML_i \geq RAM_i$ ) and ( $Q_{net} < NT$ )
    allocate global memory by using network RAM:
else
  execute the job locally:

```

Variable Q_{net} is the number of jobs waiting for network access, and NT is the network threshold, which functions to allow only a limited number of network accesses at a time. The purpose of setting NT is to prevent a large number of bus requests during a small time interval. Such bursty bus requests will cause network contention to sharply increase.

The improved load sharing scheme is denoted as “LS_Net_RAM”. Each workload trace is further divided into two types: (1) some jobs are restricted for migrations in a trace and (2) all the jobs in a trace are migratable. Figures 3.8 and 3.9 present the average execution times of all the 8 traces of both type 1 (left figure) and type 2 (right figure) executed on the 10 Mbps and 100 Mbps clusters of 6 workstations, respectively.

Our experiments show that “LS_Net_RAM” performs well for all the 8 traces of both types, while “LS_RE” or “Net_RAM” only performs well on one type of traces. We obtained consistent results on clusters of 12 and 18 workstations.

3.3.6 Summary

We have experimentally examined and compared job migrations and network RAM for sharing global cluster memory resources. Based on our experiments and analysis we have the following observations and conclusions:

- Providing a large memory space through remote paging, the network RAM is par-

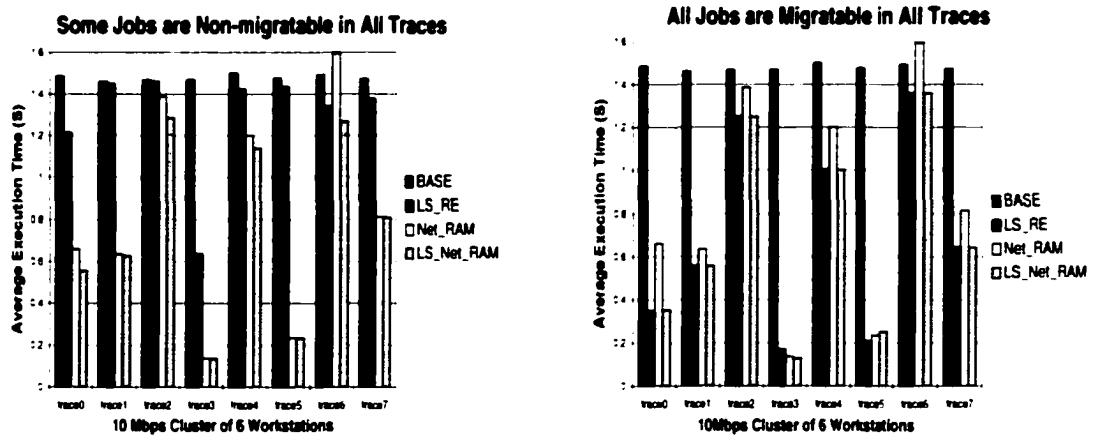


Figure 3.8: The average execution times per job of all the 8 traces (the left figure for the 8 traces where some jobs are non-migratable, and the right figure for the 8 traces where all the jobs are migratable), running on a 10 Mbps cluster of 6 workstations.

ticularly beneficial to large or data-intensive workloads where some jobs may not be migratable. However, the network RAM performance is heavily dependent on the cluster speed and the availability of the idle memory space in the cluster. Since load balancing is not considered, uneven job distributions may degrade the overall performance of cluster computing using network RAM.

- Dynamically migrating jobs by considering both the CPU and memory resources of the cluster, the load sharing policy using remote executions is particularly beneficial to data-intensive workloads where most jobs are migratable, and where each job fits in a memory space of a single workstation. The requirement of network speed by the remote-execution-based load sharing scheme is not as high as the network RAM. However, if the memory allocation of a job does not fit in any single workstation in the cluster, the additional memory requirement has to be satisfied by local disks, causing

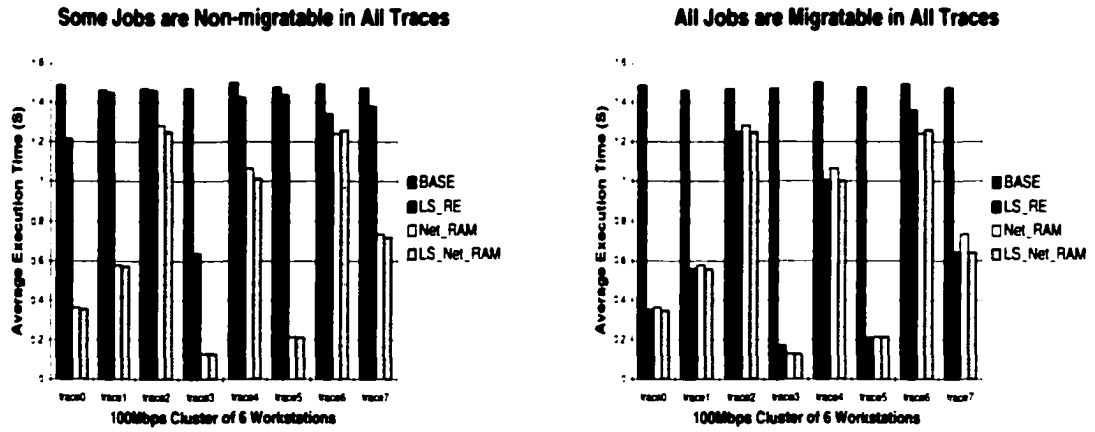


Figure 3.9: The average execution times per job of all the 8 traces (the left figure for the 8 traces where some jobs are non-migratable, and the right figure for the 8 traces where all the jobs are migratable), running on a 100 Mbps cluster of 6 workstations.

longer execution time.

- The improved load sharing scheme overcomes the limits and combines the advantages of the both schemes. We have shown that this scheme is effective for scalable cluster computing.

The impact of the 4 schemes (“Base”, “Net_RAM”, “LS_RE”, and “LS_Net_RAM”) on conditions/requirements of clusters and workloads such as CPU sharing, memory sharing, non-migratable jobs, network usage demand, and network speed demand, are summarized in Table 3.2. In the table, the relationship between a scheme and each type of system and workload condition/requirement is represented by symbol “+” (beneficial), “++” (highly beneficial), and “-” (non-beneficial). A blank represents minor or no effects.

Memory allocations of jobs are generated by a Pareto distribution in the experiments presented in this subsection. We have also run the simulations on the workloads with

	Base	Net_RAM	LS_RE	LS_Net_RAM
CPU sharing	–	–	+	+
memory sharing	–	++	+	++
non-migratable jobs	+	+	–	+
lightly loaded cluster		+	+	+
heavily loaded cluster	–	+	++	++
large individual jobs	–	++	+	++
network speed demand		high	moderate	moderate
network usage demand		high	moderate	moderate

Table 3.2: Summary of the 4 schemes and their impact on different system and workload conditions/requirements.

different memory demand distributions, and observed consistent performance results with that of the workloads by the Pareto memory demand distributions. The other distributions we have used for comparisons are uniform distribution, exponential distribution, and erlang distribution.

This study also has two limits. First, we assume that the memory requirement is known in the beginning of the execution. Workloads with dynamic memory allocations may not be accurately handled by our memory-based policies. Second, there is only one working set in each job. Although the simulated memory access patterns in our workloads could sufficiently show the performance impact of the memory demand, the memory access patterns may not be the same as those of some practical jobs.

In order to address these limits and to scale the trace-driven simulations, we have collected large real-world traces for experiments, and have investigated the effects of more dynamic memory and I/O related activities to global memory resource management for cluster computing in [133].

Chapter 4

Resource Management in Internet Caching Systems

4.1 Overview of existing caching system structures

Web caching, that is, the temporary storage of Web objects, is an effective solution to reduce bandwidth consumption, reduce server load, and reduce latency [128]. Much work has been done on Web caching at different levels.

Caching can be performed on the server side to reduce the number of requests that the server must handle. The caching issues on the server side are representatively discussed in [10] and [93]. These methods aim at improving server caching performance by balancing load and reusing requested documents.

Caching can be used in the middle of the client and the server as part of a proxy. Proxy caches are often located near network gateways to reduce the bandwidth required over expensive dedicated Internet connections. Proxy caches can be organized in a hierarchy for greater performance, e.g. the caching scheme in Harvest [31]. This is a very similar organization to the memory hierarchy in a computer system. Proxy caches can also cooperate to

achieve better performance. Cooperative proxy caches are discussed in papers [43], [83], and [56], [141]. They use different approaches to improve cache performance and reduce collaborative overhead. In adaptive Web caching schemes [144], cache servers are self-organizing and form a tight mesh of overlapping multicast groups and adapt as necessary to changing conditions.

Caching can also be performed by the client application, and is built into most Web browsers. Designs and implementations of browsers have been studied in papers [80], [101], [109], and [127]. The work in [142] attempts to transfer the server's functions to clients. They aim at larger storage, more features and better performance.

Client access patterns are characterized by several research groups (see e.g. [6], [39], [63], and [131]). The hit ratios to proxy caches have been observed in a decreasing trend in practice for a few years. There are two major reasons for the decrease. First, e-commerce and personalized services have increased the percentage of dynamic documents. Dynamic documents are usually noncacheable. Many recent studies (e.g. [22], [20], [35], [117]) have shown that requests for dynamic Web content also contain substantial locality for identical requests, and have provided several methods to cache dynamic Web contents. Second, the increase of proxy cache size has not been sufficient to keep up with the rapid increase in the numbers and types of Web servers and clients' diverse interests. For example, paper [6] gives a comprehensive study of the changes in Web client access patterns based on the traces collected from the same computing facility and the same nature of the user population separated by three years. Their experiments show that, compared with the data three years ago, the hit ratios are reduced and the most popular documents are less popular in the transfer dataset. That means accesses to different types of Web servers have become

more evenly distributed. One reason for this, we believe, is that the access variations have increased as more and more Web servers are emerging. The number and types of Web servers have increased and will continue to increase dramatically, providing services to a wider range of clients with more diverse interests. Thus, the number of unique documents has increased and will continue to increase. It will be more difficult to retain an optimal hit ratio by only increasing proxy cache size. Cache size enlargement will be expensive and may not be cost-effective.

Therefore, in such a trend we should consider effective resource management methods to well utilize the limited caching space.

4.2 Changes in Both Workload and Internet Technologies

4.2.1 Workload Changes

4.2.1.1 Trend in NLANR Workload

In order to further understand and confirm the changes in Web access patterns, we have analyzed proxy access pattern statistics of National Lab of Applied Network Research available in public domain [91]. The "Status of NLANR Caches" report indicates that a lot of system upgrade work had been done from the summer of 1997 to the summer of 1998.

¹ In order to maintain the fairness of comparisons, we decided to compare the caching patterns between year 1998 and year 2000 when there were no major upgrade events. We

¹ All of their proxies have been upgraded. For example, the old proxy "sd" DEC alpha has been replaced with a new Pentium-II with 512MB RAM and 36GB of disks on July 27, 1998. Proxy "pa" has been replaced with a Digital AlphaStation 500/500 with 512MB of RAM and 60GB of disk on October 23, 1997. They also upgraded the software. They upgraded caches to Squid-1.1.16 on August 22, 1997 and begin running Squid-1.2.beta17 on March 20, 1998. All caches have been converted over to unicast ICP on April 2, 1998.

randomly selected days in both 1998 and 2000 to do the comparisons. Since most results are consistent, we only present representative comparisons of two days here.

Metrics	August 17			First Wednesday of October		
	1998	2000	reduction	1998	2000	reduction
average hit ratio	27.60%	18.40%	33.33%	24.50%	21.40%	12.65%
coverage of top 20 servers	14.02%	13.33%	4.92%	13.68%	12.00%	12.28%

Table 4.1: Average Hit ratio and coverage comparisons of year 1998 and 2000, where the average hit ratio is calculated from proxy “pb”, “bo1”, “bo2”, “sv” and “sd”, which have their statistical reports in both years, and the coverage of top 20 servers is the percentage of the number of requests to top 20 servers over the total number of requests.

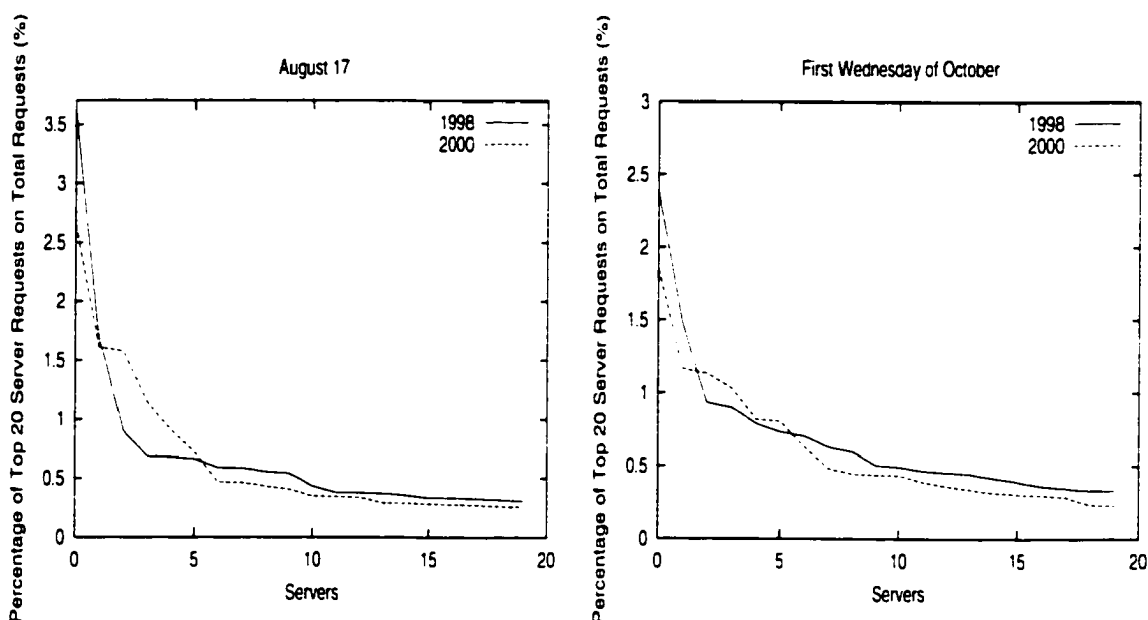


Figure 4.1: The percentage of the requests to each of the top 20 servers over the total number of requests versus each rank of servers.

Based on the hit ratio for each proxy, the number of requests to the top 20 requested servers, and other statistics reported in [91], we present comparisons of proxy access pattern changes between 1998 and 2000. Table 4.1 compares the average hit ratio per proxy and the percentage of accesses to the top 20 servers over the total number of accesses (coverage) on August 17 and the first Wednesday of October between 1998 and 2000. We show that the

hit ratios have decreased by 33.33% and 12.65%, respectively, and the coverage percentage of the top 20 servers was reduced by 4.92% and 12.28%, respectively.

In Figure 4.1, we plot the percentage of the requests to each of the top 20 servers over the total number of requests versus each server rank by sorting the numbers of accesses to the top 20 servers in decreasing order. The tail distributions of requesting accesses and server accesses of most traces (only one exception) in the study of [14] were reported to fit the Zipf-like distribution Ω/i^α . We also roughly fit the curves in Figure 4.1 into Zipf-like distributions.

In order to estimate the cache size requirement difference between 1998 and 2000 for a given hit ratio, we assume that the request distributions are identical to the server access distributions, and the file size with the same rank in 1998 is the same as that in 2000. We also assume that the priority of caching a document is based on the document popularity. Thus, for a 20 GByte proxy cache on August 17, 1998, in order to keep the same the hit ratio in 2000, the cache size needs to be enlarged to 37.29 GBytes (a 86% increase). A 20 GByte proxy cache on the first Wednesday of October in 1998 needs to be enlarged to 107.58 GByte (an increase of 5 times) in order to keep the same hit ratio in 2000.

4.2.1.2 Trend in BU Workload

Boston University collected traces from the similar computing facility and user population in 1995 and 1998, which can be found in [13]. We select the traces in a period of two months of the two years, which are denoted as BU-95 and BU-98, respectively. We have also analyzed the browser access pattern of BU traces. The difference between the total number of requests of BU-95 trace and that of BU-98 trace is very big. In order to make fair

comparisons, we compare request ratios instead of the numbers of requests between these two traces. We compare two statistical results: (1) the percentages of requests to different servers over the total requests, and (2) the percentages of requests to different documents over the total requests, both reflecting access distributions.

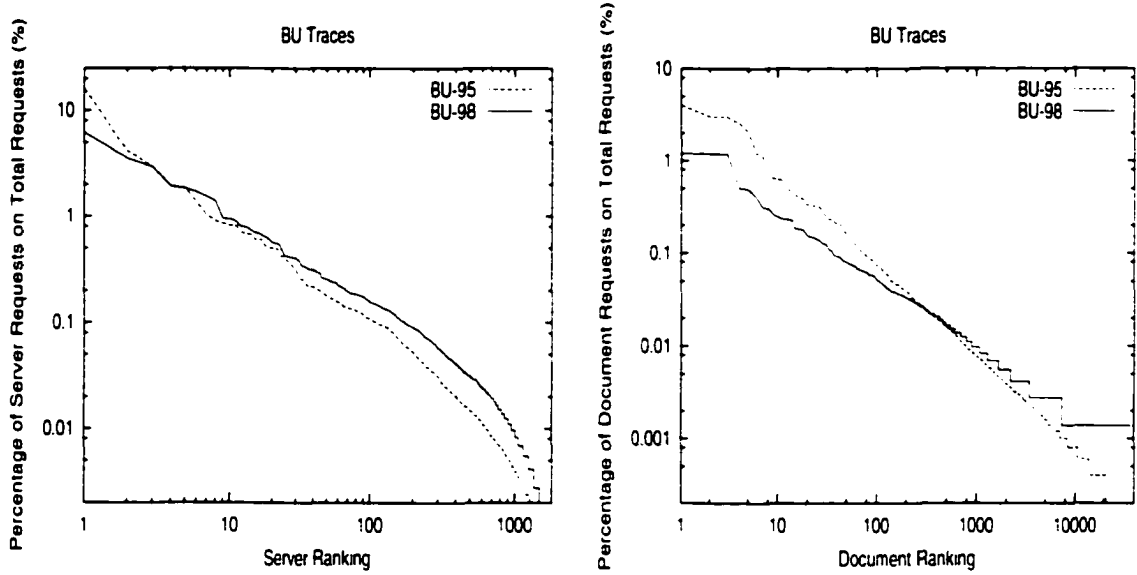


Figure 4.2: The percentage of the requests to each server or document over the total requests versus server ranking or document ranking.

In the left figure of Figure 4.2, we plot the percentage of the requests to each server over the total requests versus server ranks. The ranks are obtained by sorting the percentages of accesses to servers in decreasing order. In the right figure of Figure 4.2, we plot the percentage of the requests to each document over the total requests versus document ranks by sorting the percentages of accesses to documents in decreasing order. We have two observations. First, the access distributions presented in the two figures are consistent with an observation reported in [14] — the distributions of requesting accesses and server accesses follow the Zipf-like distribution Ω/i^α . Second, the accesses had been changed to be more evenly distributed during the 3 year period. Specifically, the request percentages of BU-95

trace are higher than those of BU-98 for very high rank servers and documents. But for lower rank servers and documents, the request percentages of BU-98 exceed those of BU-95. So for the same cache size, BU-95 could get higher hit ratio than that BU-98. However, the hit ratios of BU-98 can be increased in a faster pace than that of BU-95 if the cache is larger than a certain size.

This type of access pattern change demands progressive increase of the cache size in order to retain a fixed hit ratio during a period of time. To estimate the cache size requirement difference between BU-95 and BU-98 for a given hit ratio, we fit the curves in Figure 4.2 into Zipf-like distributions. We assume that the file size with the same rank in BU-95 is the same as that in BU-98, and the priority of caching a document is based on the document popularity. From the fit Zipf-like distribution curves, we estimate that a 12.7 times larger cache is needed for BU-98 to achieve a given hit ratio in BU-95. (In fact, we obtained a number of 10 that is smaller than 12.7 from simulation results. This is because the average document size in BU-98 is smaller than that in BU-95.)

Both workloads show the same trend. The assumptions and numbers may not be directly used to guide the proxy cache design, but we attempt to show the trends of decreasing hit ratios in proxies and the diversity of the Web contents. We envision that the access distribution is becoming more evenly distributed. Thus, in order to retain the proxy cache hit ratios, we have to enlarge the cache size as time passes. However, the proxy cache size enlargement will be no longer sufficient. Therefore, we should consider alternative methods to effectively utilize the limited caching space.

4.2.2 Technology Changes

A browser cache was initially developed as a small data buffer with a few simple data manipulation operations. Users may not effectively retain the cached data with a high quality of spatial and temporal locality.

There are two types of browser caches: persistent and non-persistent. A persistent cache retains the cached document unless a user manually deletes it. A non-persistent cache deallocates the cached document when a user quits the browser. Persistent caches are commonly used in almost all the commercial browsers, such as AOL, Communicator, Internet Explorer, and Netscape browsers.

Current technologies have improved browsers in the following three areas. First, browsers provide a function for users to set the browser cache size. With the rapid increase of memory and disk capacity in workstations and PCs, and with the rapid growth of Web applications, user browser cache size will tend to increase as time passes. In addition, several new software techniques are introduced for users to effectively increase the browser cache size. For example, "browser cache switch" [49], allows users to set multiple browser caches in one machine, and to switch them from one to another during Web browsing. Thus, different caches can be used for different contents and for different time periods. This technique significantly increases the size of a browser cache for an effective management of multiple data types. However, the larger the browser cache size is set, the more spatial locality will be neglected by the proxy cache server.

Secondly, in order to help Web users to effectively use and manage large browser cache data, browser software has been upgraded to include several sophisticated database func-

tions, such as file searching, folding, and grouping. With the aid of these functions, users will pay more attention to the organized browser cache data objects, and tend to keep them in the cache much longer than to keep the unorganized data objects. However, the longer the cached data is retained, the more temporal browser cache locality will be neglected by a proxy cache server.

Thirdly, in order to improve the browsing speed, a large memory drive can be configured to store the entire browser cache. This technique of “browser cache in memory”, has been implemented in several commercial browsers, such as Internet Explore and Netscape. This technique can be further extended to periodically save the cached data objects in a special directory in the disk. The data will be brought back from the disk to the special memory drive whenever the system is restarted or rebooted. Several studies (see e.g. [44], [93], and [135]) have shown that transferring data through a moderate speed network will be significantly faster than obtaining the same amount of data from a local disk through page faults. The high speed memory access is not only beneficial to a local user, but also speeds up data accesses for remote users to share browser caches.

Therefore, the browsers are becoming more powerful and the more powerful resources are not fully utilized.

4.3 Overview of the Limits on Existing Caching System Structures

The configuration of a *proxy-browser system* is a commonly used infrastructure for Web caching, where a group of networked clients connects to a proxy cache server and each

client has a browser cache. A standard Web caching model built on a proxy-browser system has the following data flows. Upon a Web request of a client, the browser first checks if the requested document exists in the local browser cache. If so, the request will be served by its own browser cache. Otherwise the request will be sent to the proxy cache. If the requested document is not found in the proxy cache, the proxy server will immediately send the request to its cooperative caches, if any, or to an upper level proxy cache, or to the Web server, without considering if the document exists in other browsers' caches.

This model has two features that prevent it from effectively utilizing the rapid improvement in Internet technologies and from adapting in a timely manner the changes of the supply and demand of Web contents. First, with a significant increase of memory and disk capacity in workstations and PCs, and with the fine improvement of Web browser caching capability, users are able to enlarge browser cache size for more frequent accesses to cached documents and to retain the documents in an organized manner for a longer period of time. Furthermore, there exist some documents which are already replaced in the proxy cache but still retained in one or more browser caches, because the request rates to the proxy and to browsers are different, causing the replacement in the proxy and browsers at a different pace. However, the browser caches are not shared among the browsers and the available locality in browsers is neglected in Web proxy caching. When a requested document misses in a local browser cache and the proxy cache, it may have been cached in other browser caches. Second, with the rapid increase of Web servers and the huge growth of Web client populations in both numbers and types, the requested Web contents have become, and will continue to be, more diverse, causing a decrease of proxy hit ratios and an increase in the amount of document duplications among the proxy cache and the browser caches. These

two limits prevent clients from effectively utilizing the increasingly limited caching space.

We will focus on the resource management of a proxy-browser system to address the limits of existing caching systems, aiming at adapting the changes of both workload and technologies (e.g. peer-to-peer). Chapter 5 presents how we address the neglected locality in browsers using the peer-to-peer model. Chapter 6 presents how we address the duplications among the proxy cache and browser caches. Chapter 7 handles the new problems caused by sharing browser caches. Chapter 8 presents a prototype implementation.

Chapter 5

Locality and Information Sharing among Browsers

We will first address the first limit in current proxy-browser system, the neglected locality in browsers. We believe there are three practical reasons for a proxy server to exclude the consideration of neglected locality in browsers. First, the browser caches are not shared for software simplicity and user privacy reasons; and the dynamic status in each cache is unknown to the proxy server. Second, the possibility of a proxy cache miss that is a browser cache hit may have been considered low, although no such a study has been found in literature. Finally, a browser cache was initially developed as a small data buffer with a few simple data manipulation operations. Users may not effectively retain the cached data with a high quality of spatial and temporal locality. But changes in workload and technology show that potential benefit gain in caching performance by exploiting the neglected locality is increasing. The quality of spatial and temporal locality in browser caches has been and will continue to be improved, inevitably providing a rich and commonly sharable P2P storage among trusted Internet peers. In this study, we introduce a P2P technique to fully utilize browser caches, called “browsers-aware proxy server”. Conducting trace-

driven simulations, we quantitatively evaluate its potential benefits for further improving proxy caching performance. Our effort shares the same objective of building effective P2P infrastructure that lets users easily and reliably share files and processing power over the Internet.

5.1 Browsers-Aware Proxy Server

In the design of the browsers-aware proxy server, the proxy server connecting to a group of networked clients maintains an index file of data objects of all clients' browser caches. If a user request misses in its local browser cache and the proxy cache, the browsers-aware proxy server will search an index file attempting to find it in a client's browser cache before sending the request to an upper level server. If such a hit is found in a client, we propose two alternative implementations to let the requesting client access the data object. First, the proxy server will inform this client to directly forward the data object to the requesting client. In order to retain user browsers' privacy, the message passing from the source client to the requesting client should be anonymous to each other. The second implementation alternative is to make the proxy server provide the data by loading the data object from the source client and then storing it to the requesting client.

In order to implement the browsers-aware concept in a proxy server, we create a *browser index file* in the proxy server. This index file records a directory of cached file objects in each client machine. Each item of the index file includes the ID number of a client machine, the URL including the full path name of the cached file object, and, if any, a time stamp of the file or the TTL (Time To Live) provided by the data source. Since the dynamic changes

in browser caches are only partially visible to the proxy server (when a file object is sent from the proxy cache to the browser), the browser index file will be updated periodically by each browser cache. Here is another alternative. After a file object is sent from the proxy server to a client's browser cache, its index item is added to the browser index file. Whenever this file object is replaced or deleted from the browser cache, the client sends an invalidation message to the proxy server. After that, the proxy deletes the corresponding index item.

Figure 5.1 presents the organization of the browsers-aware proxy server by an example. A group of client machines is connected by a local area network. For a given Web service request with a specific URL in client machine i , the browser cache is first searched attempting to satisfy the request. After the request misses in the browser cache, client i sends the request to the proxy server, where the proxy cache is searched for the same purpose. After the request misses again in the proxy cache, the browser index file is searched, where the URL is matched in client machine j . The proxy server informs client machine j to forward the cached file object to client i , or fetches the cached object from machine j and then forwards it to client i .

5.2 Simulation Environment

The browsers-aware proxy server and related performance issues are evaluated by trace-driven simulations. The evaluation environment consists of different Web traces, simulated clustered client machines, and a proxy server having aware or unaware browser caches. We will discuss the selected Web traces and our simulation model in this section.

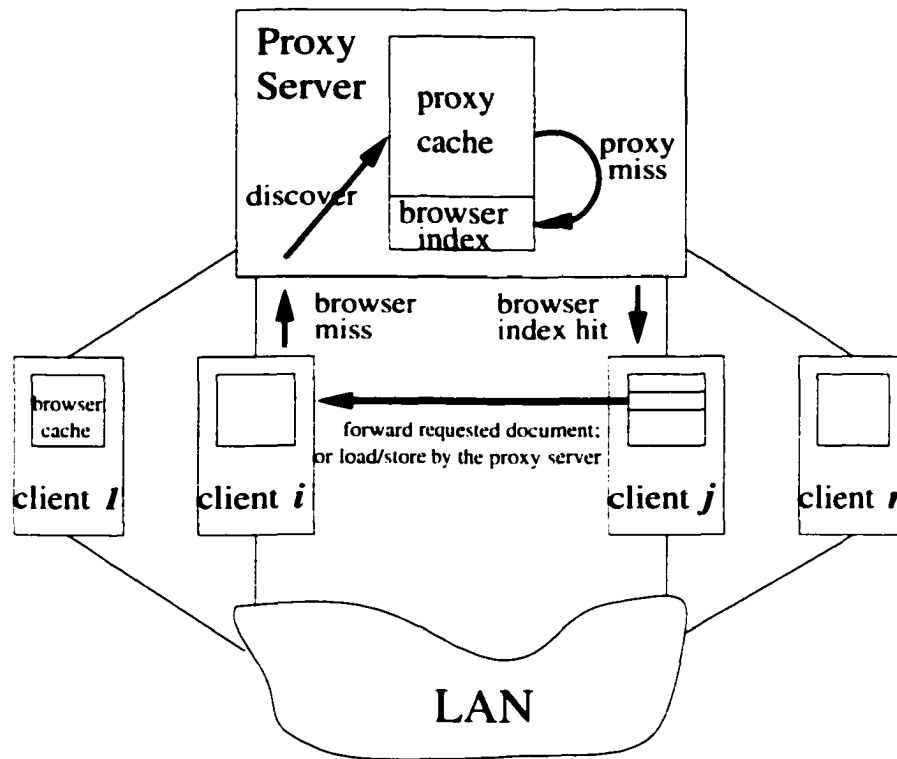


Figure 5.1: Organizations of the browsers-aware proxy server.

5.2.1 Traces

Table 5.1 lists the Web traces we have used for performance evaluation, where “*Inf. Cache*” denotes *infinite* cache size that is the total size storing all the unique requested documents, and “Max HR” and “Max BHR” denote maximal hit ratio and maximal byte hit ration, respectively.

1. NLANR traces: NLANR (National Lab of Applied Network Research) provides sanitized cache access logs for the past seven days in the public domain [91]. NLANR takes special steps to protect the privacy of those participating in their cache mesh. Client IP addresses are randomized from day-to-day, but consistent within a single log file. Client IP addresses are very important in our study, so we use traces based on

one day's log file. NLANR provides about ten proxies' traces. We have used one day's trace of July 14, 2000 from the "uc" proxy and one day's trace of August 29, 2000 from the "bol" proxy and "pa" proxy, which are denoted as NLANR-uc, NLANR-bol and NLANR-pa.

2. Boeing traces: The Boeing Company collected anonymized logs from Boeing's Puget Sound perimeter (firewall) proxies by using an anonymizer tool (log2anon) and made these logs available in [12]. For privacy reasons, client IP addresses are not identical between two different days, so we use traces based on one day's log file. We have used one day's trace on March 4, 1999, and one day's trace on March 5, 1999, which are the most recent traces in this site and denoted as Boeing-4 and Boeing-5.
3. BU traces: Boston University collected traces from the similar computing facility and user population in 1995 and 1998, which can be found in [13]. We select the traces in a period of two months of the two years, which are denoted as BU-95 and BU-98, respectively.
4. CA*netII traces: The CA*netII (Canada's coast to coast broadband research network) parent cache provides sanitized log files in [16]. The client IDs are consistent from day to day, so we concatenate two days' logs together as our trace. The two logs we used are the traces collected on September 19, 1999 and September 20, 1999, which are the most recent traces in this site.

Traces	Period	# Requests	Total GB	Inf. Cache	# Clients	Max HR	Max BHR
NLANR-uc	7/14/00	360806	4.36	3.72 GB	95	19.11%	14.80%
NLANR-bo1	8/29/00	263942	1.71	1.22 GB	115	21.32%	28.79%
NLANR-pa	8/29/00	310939	2.52	1.85 GB	145	22.78%	26.71%
Boeing-4	3/4/99	219951	7.54	6.21 GB	3996	44.91%	17.69%
Boeing-5	3/5/99	184476	7.00	5.50 GB	3659	45.07%	21.63%
BU-95	Jan.95-Feb.95	502424	1.31	0.90 GB	591	64.14%	31.37%
BU-98	Apr.98-May 98	72626	0.45	0.29 GB	306	40.62%	35.94 %
CA*netII	9/19-/9/20/99	745943	0.089	0.062 GB	3	34.20%	29.84%

Table 5.1: Selected Web Traces.

5.2.2 A browsers-proxy caching environment

We have built a simulator to construct a system with a group of clustered clients connecting to a proxy server. The cache replacement algorithm used in our simulator is LRU. We do not simulate document aging. However, all the traces have the size of a document for each request. If a user request hits on a document whose size has been changed, we count it as a cache miss. We have implemented and compared the following five Web caching organizations using the trace-driven simulations:

1. *Proxy-cache-only*: Each client does not have a browser cache. Every client request is sent directly to the proxy cache server.
2. *Local-browser-cache-only*: Each client has a private browser cache, but there is no proxy cache server for client machines.
3. *Global-browsers-cache-only*: Each client has a browser cache that is globally shared among all the clients by maintaining an index file in each client machine. The index file records a directory of cache documents of all clients. A browser does not cache documents fetched from another browser cache. If a request is a miss in its local browser, the client will check the index file to see if the requested file is stored in

other browser caches before sending the request to a Web server. There is no proxy cache server for the group of client machines.

4. *Proxy-and-local-browser*: Each client has a private browser cache, and there is a proxy cache server for the group of client machines. If a request misses in its local browser, it will be sent to the proxy to check if the requested document is cached there. If it misses again, the proxy will send the request to an upper level server.
5. *browsers-aware-proxy-server*: This is the enhanced proxy caching technique presented in section 5.1.

We have validated our simulator motivated by the method in [39]. We simulated each NLANR trace with an *infinite* proxy cache size, which is the total size storing all the unique requested documents. We compared the simulated and actual daily hit ratios in the public domain [91]. The reason we use an *infinite* cache size for comparisons is as follows. All the proxies of NLANR allocate about 16 GB of the disk for caching. But, for privacy and protection reasons, we are only able to use one day's log file, whose total requested document size is less than 16 GB. Our experiments show that the average hit ratio difference is about 6% for six NLANR traces. In the actual daily statistics of NLANR traces, some of today's requests hit the documents cached "yesterday". The simulation does not reflect this small number of special hits. This is a major reason for the 6% error. We also validated our simulator by comparing the hit ratios and byte hit ratios of above schemes 4 and 5 with infinite cache proxy cache and browser cache. They all join to the same point.

We use two performance metrics. *Hit ratio* is the ratio between the number of requests that hit in browser caches or in the proxy cache and the total number of requests. *Byte hit*

ratio is the ratio between the number of bytes that hit in browser caches or in the proxy cache and the total number of bytes requested.

5.3 Performance Evaluation

Before presenting performance results, we will first look into a browser and proxy cache size related issue to provide a basis and a rationale for us to configure our simulated Web caching system.

5.3.1 Sizes of browser and proxy caches

Rousskov and Soloviev [105] have studied seven Squid proxies covering several levels of the caching hierarchy from leaf university proxies, to top level proxies for large country-wide networks, and to the international root proxy located at NLNR. Three of them are leaf proxies which are related to our study: *run* from Netherlands, *uit* from Norway, and *adfa* from Australia. Their proxy cache related configurations are listed in the second to fourth columns of Table 5.2. Squid uses a two level cache. The first level is a small and hot memory in which very popular and recently requested documents are kept. The second level is a disk cache where the majority of documents reside. The second and third columns in Table 5.2 are the sizes of the hot memory and disk caches. The last two columns are the average proxy cache size in hot memory per client and the average proxy cache size in disk per client. We assume each client's browser has a cache. If we use the average proxy cache size per client in Table 5.2 as the browser cache size of each client, the memory space ranging from 0.04 MB to 0.08 MB is certainly too small, and the total cache size ranging from 7.34 MB to 10.86 MB is also not large enough in practice for today's computer systems. Therefore,

Proxies	hot memory	disk cache	# clients	memory cache/client	disk cache/client
<i>ruu</i>	32MB	5.6GB	518	0.0618MB	10.8MB
<i>uit</i>	32MB	3.8GB	378	0.0847MB	10.1MB
<i>adfa</i>	32MB	5.8GB	798	0.0401MB	7.3MB

Table 5.2: Representative proxy cache configurations reported in [105].

in our study we define a *minimum* browser cache size as

$$\text{Min}(\text{Cache}_{\text{browser}}) = \frac{\text{Cache}_{\text{proxy}}}{m}. \quad (5.1)$$

where $\text{Cache}_{\text{browser}}$ is the size of a client browser cache, m is the number of clients, and $\text{Cache}_{\text{proxy}}$ is the size of the proxy cache responsible for the m clients. We also conservatively define an *average* browser cache size as $\frac{\beta \text{Cache}_{\text{proxy}}}{m}$.

$$\text{Average}(\text{Cache}_{\text{browser}}) = \frac{\beta \text{Cache}_{\text{proxy}}}{m}. \quad (5.2)$$

where β is in a range of 2 to 10. Since the accumulated browser cache size increases faster than the increase of the proxy cache size, the value of β tends to increase if both clients and the proxy server are upgraded as time passes.

5.3.2 How much is browser cache data sharable?

To answer this question, we have operated the five caching policies with different traces on a simulated Web caching environment where the browser cache size of each client is set to *minimum* by (5.1). Performance results of all the traces we have used are quite consistent. We only representatively present the results of hit ratios and byte ratios from the NLNR-uc trace in Figure 5.2, where the size of the proxy cache is scaled from 0.5%, 5%, 10%, and

to 20% of the *infinite* proxy cache size, the browser cache size is also scaled up accordingly by (5.1).

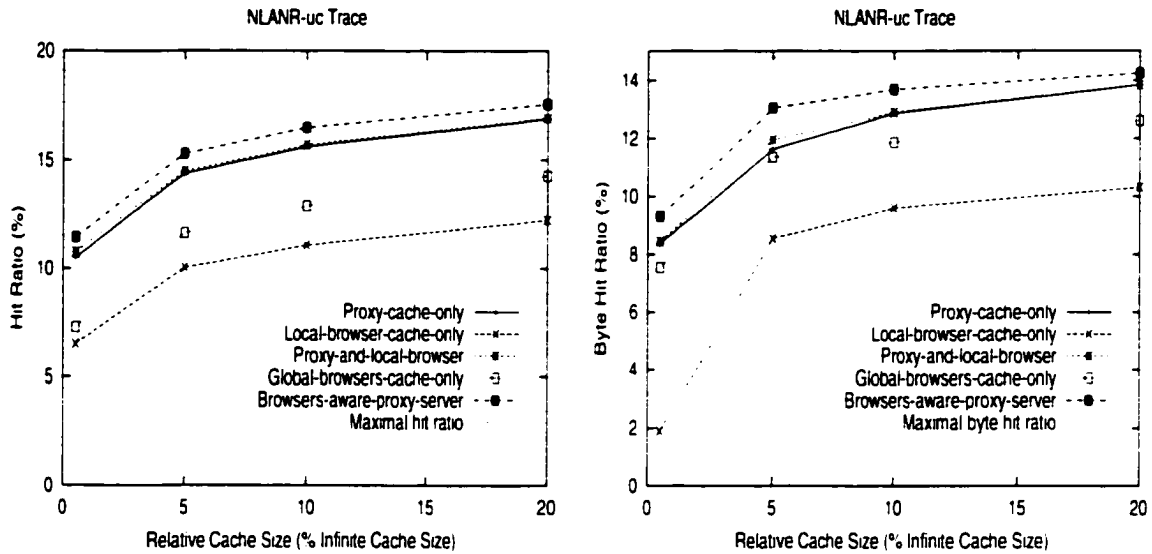


Figure 5.2: The hit ratios and byte hit ratios of the five caching policies using NLANR-uc trace, where the browser cache size is set *minimum*.

Figure 5.2 shows that the hit ratios (left) and byte hit ratios (right) of the *browsers-aware-proxy-server* are the highest, particularly, the hit ratios are up to 5.94% higher and the byte hit ratios are 9.34% higher than those of the *proxy-and-local-browser*, even when the browser cache size is set to *minimum*. This means that sharable data locality does exist, even for a small browser cache size. The sharable data locality proportionally increases as browser cache size increases and as the number of unique file objects cached in browsers increases, both of which are the trends in Web computing. In next subsection, we will show that significant proxy cache performance improvement can be achieved by the proposed browsers-aware proxy server to exploit sharable data locality.

We also show that methods of *proxy-cache-only*, *local-browser-cache-only*, and *global-browsers-cache-only* are not as effective as the method of *proxy-and-local-browser*. *Local-*

browser-cache-only had the lowest hit and byte hit ratios due to the minimum caching space. *proxy-and-local-browser* only slightly outperforms *proxy-cache-only*, which implies that performance gain from a local browser cache is limited. Another observation worth mentioning is that *proxy-and-local-browser* and *global-browsers-cache-only* had lower hit and byte hit ratios than *browsers-aware-proxy-server*. This observation confirms the existence of two types of misses. First, there exist some documents that are already replaced in the proxy cache but still retained in one or more browser caches, because the request rates to the proxy and to browsers are different, causing the replacement in the proxy and browsers at a different pace. Second, there are some documents that are already replaced in browser caches but still retained in the proxy cache, because a browser cache is much smaller than the proxy cache. The *browsers-aware-proxy-server* effectively addresses these two types of misses.

Figure 5.3 presents the breakdowns of the hit ratios and the byte hit ratios of the *browsers-aware-proxy-server* using NLANR-uc trace. There are three types of hits: hits in the local browser cache, hits in the proxy cache, and hits in remote browser caches. We show that the hit ratio and byte hit ratio in remote browser caches should not be neglected even when the browser cache size is very small.

The *browsers-aware-proxy-server* has another advantage over the *proxy-and-local-browser* policy in terms of “memory” byte hit ratios. In other words, for the same byte hit ratio, a higher percentage of requests will hit in the main memory of browser caches and the proxy cache provided by the *browsers-aware-proxy-server*. To quantitatively justify this claim, we have compared the memory byte hit ratios of the two policies for an equivalent byte hit ratio.

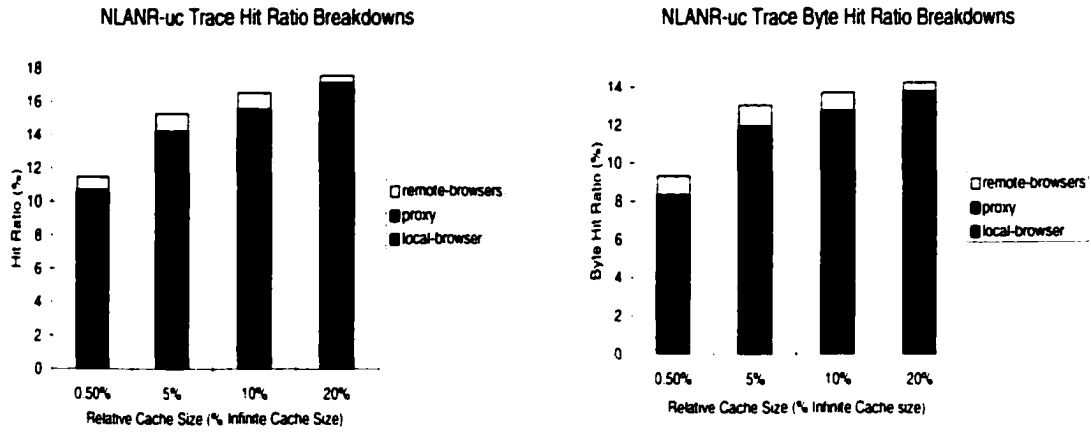


Figure 5.3: The breakdowns of the hit ratios and byte hit ratios of the browsers-aware proxy using NLNR-uc trace, where the browser cache size is set *minimum*.

In our simulation, we set the memory cache size in the proxy as 1/150 of the proxy cache size based on the memory ratio reported in Table 5.2. We also set the memory size of a browser cache as 1/150 of the browser cache size, which is not in favor of the *browsers-aware-proxy-server* because the memory cache portion in a browser can be much larger than that for the proxy cache in practice. We also conservatively assume that one memory access of one cache block of 16 Bytes spends 200 *ns* (the memory access time is lower than this in many advanced workstations), and one disk access of one page of 4 KBytes is 10 *ms*.

Figure 5.2 shows that the hit and byte hit ratios of the *browsers-aware-proxy-server* at 5% of the *infinite* cache size are very close to those of the *proxy-and-local-browser* policy at 10% of the *infinite* cache size (the hit ratio comparison is 15.3 v.s. 15.7, and byte hit ratio comparison is 13.06 v.s. 12.91). However, the memory byte hit ratios of the two schemes are quite different under the same condition, which are 3.5% for the *browsers-aware-proxy-server*, and 1.9% for the *proxy-and-local-browser* policy, respectively. The larger memory

byte hit ratio of the *browsers-aware-proxy-server* in this case would reduce 15.2% of the total hit latency compared with the *proxy-and-local-browser*. The latency reduction due to the higher percentage memory accesses will be larger in practice because the memory cache size of each browser is much larger than the assumed size.

5.3.3 Performance of browsers-aware proxy server

We have evaluated and compared the performance of the *browsers-aware-proxy-server* and *proxy-and-local-browser* schemes using the NLANR-bo1 trace and two BU traces. For experiments of each trace, the proxy cache size is set to 0.5%, 5%, 10%, and 20% of the *infinite* proxy cache size. Accordingly, each browser cache is also set to 0.5%, 5%, 10%, and 20% of the average *infinite* browser cache size calculated from all the browsers. The *infinite* cache size of a browser is the total size of all uniquely requested documents in this client. For example, if the proxy cache is set to 0.5% of the *infinite* proxy cache, all browsers' caches will also be set to 0.5% of the average size of the *infinite* browser size of all browsers. The value of β calculated from each trace falls into the *average* range of 2 to 10.

Figures 5.4 - 5.10 present the hit ratios (left) and byte hit ratios (right) of the two policies on NLANR-uc trace, NLANR-bo1 trace, NLANR-pa trace, Boeing-4 trace, Boeing-5 trace, BU-95 trace, and the BU-98 trace, respectively. Compared with the *proxy-and-local-browser* scheme, *browsers-aware-proxy-server* consistently and significantly increases both hit ratios and byte hit ratios on all the traces.

The limit of the Browsers-Aware Proxy Server

When the number of clients is small, and their accumulated size of the browser caches is much smaller or not comparable to the proxy cache size, the cache locality inherent in

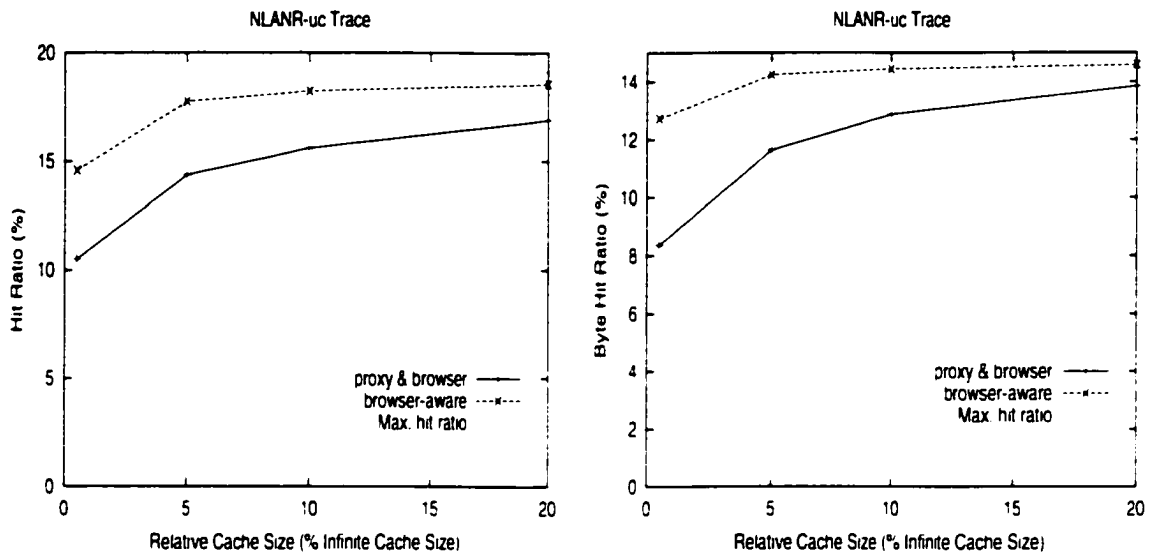


Figure 5.4: The hit rates and byte hit rates of the *browsers-aware-proxy-server* and *proxy-and-local-browser* scheme using NLANR-uc trace, where the browser cache size is set *average*.

browsers is low, so the performance gain from the browsers-aware proxy cache will also be insignificant. Figure 5.11 presents such an example, where the total number of clients of the CA*netII trace is only 3, the accumulated size of three browser caches is small. The increases of both average hit ratio and byte hit ratio of this trace by the *browsers-aware-proxy-cache* are below 1%, compared with the *proxy-and-local-browser* scheme.

5.3.4 Performance Impact of Scaling the Number of Clients

We have also evaluated the effects of scaling the number of clients to browsers-aware proxy servers. For each trace, we observe its hit ratio (or byte hit ratio) increment changes by increasing the number of clients from 25%, to 50%, to 75%, and to 100% of the total number of clients. We also regard each percentage as a relative number of clients. For all relative numbers of clients of each trace, the proxy cache size is fixed to 10% of the *infinite* proxy cache size when the relative number of clients is 100%. The byte hit ratio increment or the

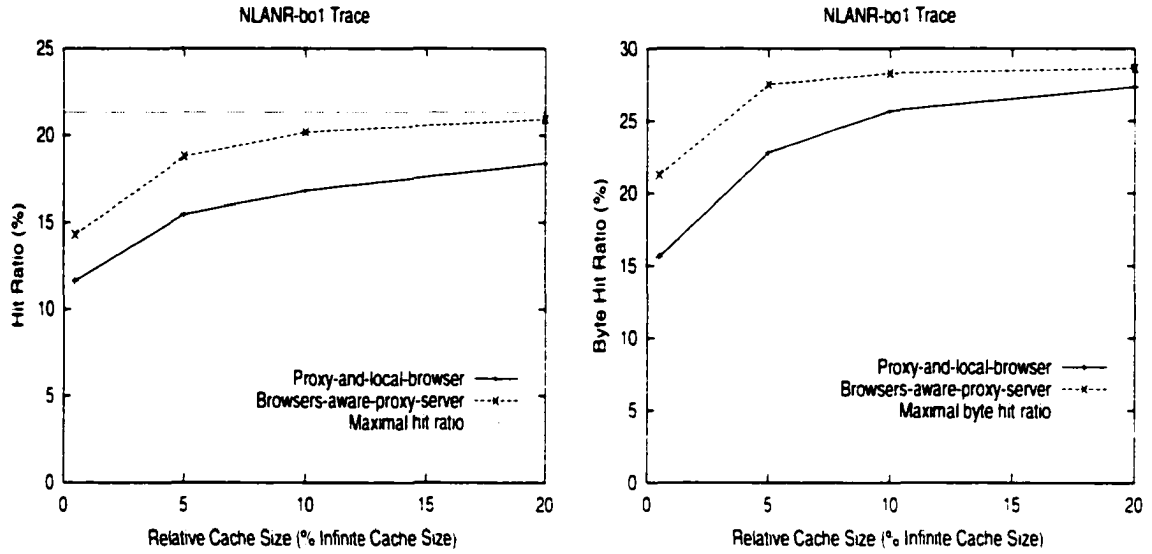


Figure 5.5: The hit rates and byte hit rates of the *browsers-aware-proxy-server* and *proxy-and-local-browser* scheme using NLANR-bo1 trace, where the browser cache size is set *average*.

hit ratio increment of the browsers-aware proxy server for a given trace is defined as

$$\frac{(\text{byte}) \text{ hit ratio of browse-aware} - (\text{byte}) \text{ hit ratio of proxy-and-local-browser}}{(\text{byte}) \text{ hit ratio of proxy-and-local-browser}}.$$

Figure 5.12 presents the hit ratio increment curves (left figure) and the byte hit ratio increment curves (right figure) of the three traces as the relative number of clients changes from 25% to 100%. Our trace-driven simulation results show that both hit ratio increment and byte hit ratio increment of the browsers-aware proxy server proportionally increases as the number of clients increases. For some traces, the increments are significant. For example, the hit ratio increment of BU-98 trace increases from 10.70% to 13.35%, to 16.87%, and to 19.35%, as the relative number of clients increases from 25% to 50%, to 75%, and to 100%, respectively. The byte hit ratio increment of BU-95 trace increases from 4.33% to 20.17%, to 24.82%, and to 28.08%.

The performance results indicate that a browsers-aware proxy server is performance

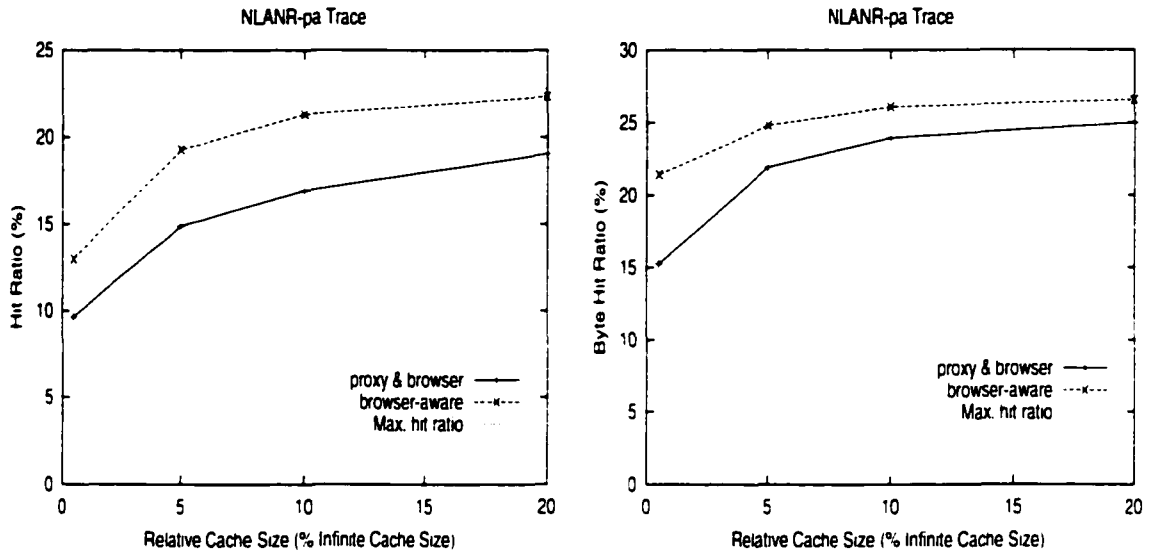


Figure 5.6: The hit rates and byte hit rates of the *browsers-aware-proxy-server* and *proxy-and-local-browser* scheme using NLANR-pa trace, where the browser cache size is set *average*.

beneficial to client cluster scalability because it exploits more browser locality and utilizes more memory space as the number of clients increases.

5.4 Overhead Estimation

The additional overhead of the browsers-aware proxy cache comes from the data transferring time for the hits in remote browsers. The simulator estimates the data transferring time based on the number of remote browser hits and their data sizes on a 10 Mbps Ethernet. The browser access contention is handled as follows. If multiple requests hit documents in a remote browser simultaneously, the bus will transfer the hit documents one by one in the FIFO order distinguished by each request's arrival time. Our experiments using the *ping* facility show that the startup time of data communications among the clients in our local area university network is less than 0.01 second. Setting 0.01 second as the network connection time, we show that the amounts of data transferring time and the bus contention

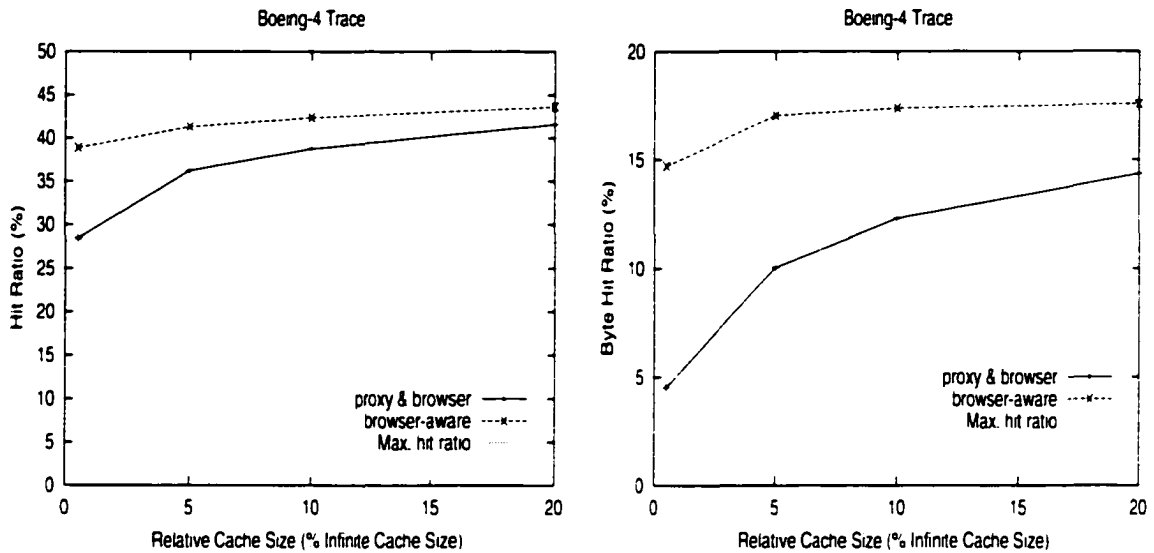


Figure 5.7: The hit rates and byte hit rates of the *browsers-aware-proxy-server* and *proxy-and-local-browser* scheme using Boeing-4 trace, where the browser cache size is set *average*.

time spent for communication among browser caches of the browsers-aware proxy server on all the traces is very low. For example, the largest accumulated communication and network contention portion out of the total workload service time for all the traces is less than 1.25%. In addition, the contention time only contributes up to 0.12% of the total communication time, which implies that the browsers-aware proxy server does not cause bursty hits to remote browser caches.

Another potential overhead is the update of the browser index file if the update is not conducted at a suitable time or conducted too frequently. There have been some solutions to address this concern. For example, the browser could send its update information when the path between the browser and the proxy is free to avoid contention. The study in [43] shows that the update of URL indices among cooperative caches can be delayed until a fixed percentage of cached documents are new. The delay threshold of 1% to 10% (which corresponds to an update frequency of roughly every 5 minutes to an hour in their exper-

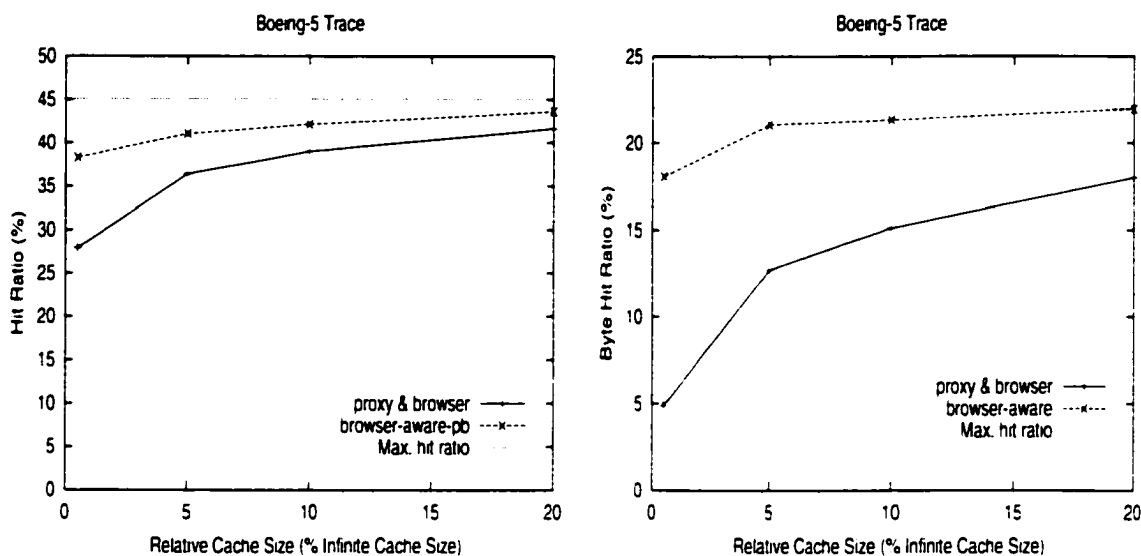


Figure 5.8: The hit rates and byte hit rates of the *browsers-aware-proxy-server* and *proxy-and-local-browser* scheme using Boeing-5 trace, where the browser cache size is set *average*.

iments) results in a tolerable degradation of the cache hit ratios. In their experiments, the degradation is between 0.02% to 1.7% for the 1% choice. Our concerns should be less serious because the updates are only conducted between browsers and the proxy without broadcasting. Thus, the index file update overhead between browsers and proxy is very low.

The last potential concern is the space requirement of the proxy cache to store the browser index. We address this concern by an example. Each URL is represented by a 16-byte MD5 signature [88]. Assume there are 1000 clients connected to one proxy. Each client has a browser with a 8MB cache. We assume that an average document size is 8 KB. Each browser has about 1 K Web pages. The proxy server only needs about $1000 \times (8MB/8KB) \times 16 = 16MB$ to store the whole browser index file for the 1000 browsers. If we apply the compression methods presented in [43] or [89], the browsers-aware proxy server requires even less space to store the index file. (e.g. a storage of 2 MB is sufficient for the 1000 browsers with a tolerant inaccuracy).

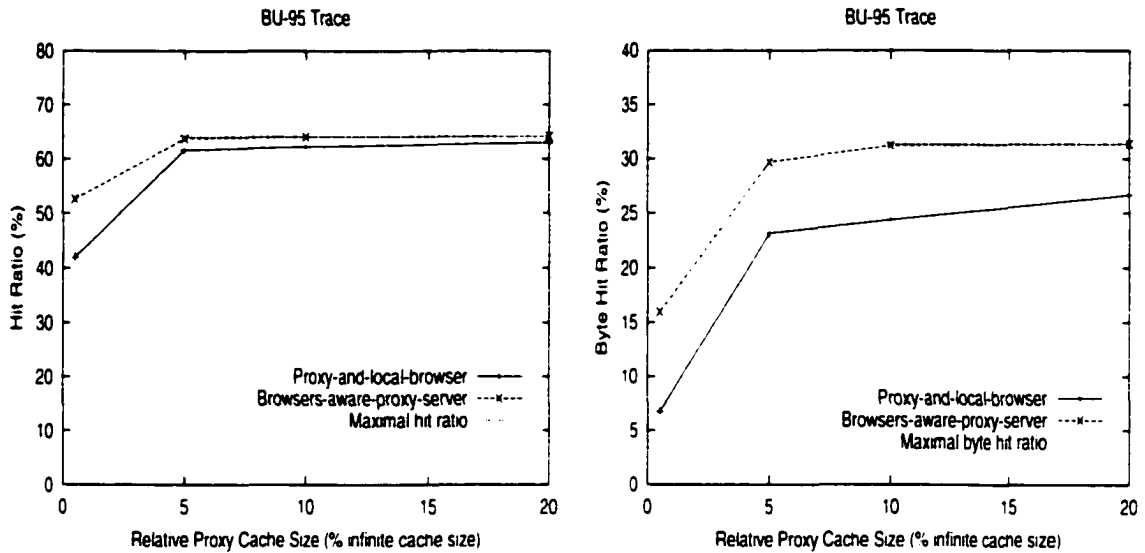


Figure 5.9: The hit rates and byte hit rates of the *browsers-aware-proxy-server* and the *proxy-and-local-browser* scheme using the BU-95 trace, where the browser cache size is set *average*.

We can also take advantage of a Bloom filter that is used to keep URL indices of cooperative caches in [43]. Assume that there are 1000 clients connected to one proxy. Each client has a browser in which has a 8MB cache. Similar to [43], we also assume that an average document size is 8KB. Each browser has about 1K Web pages. The Bloom filter needs 2KB to represent 1K pages of each browser. The proxy needs only about 2000KB \approx 2MB to store the whole browser index file.

5.5 Chapter Conclusion

We have proposed and evaluated a browsers-aware proxy server to provide a distributed P2P Web document sharing service. We have also quantitatively answered two questions: how much browser data is sharable? and how much proxy caching performance improvement can we gain by this P2P approach? Could the browsers-aware proxy server be scalable and

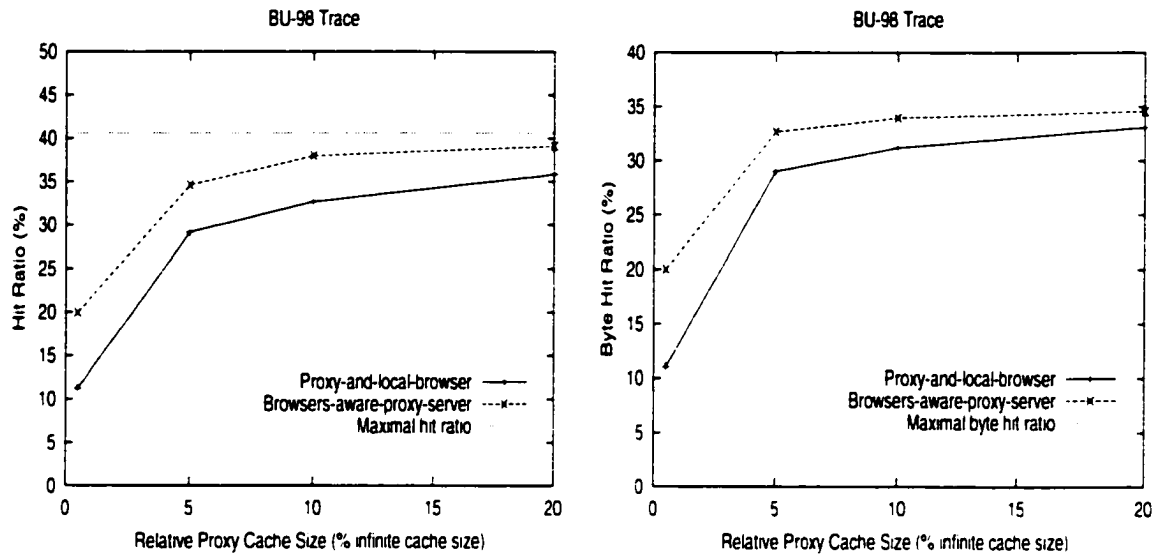


Figure 5.10: The hit rates and byte hit rates of the *browsers-aware-proxy-server* and the *proxy-and-local-browser* scheme using the BU-98 trace, where the browser cache size is set *average*.

reliable? Our study shows that the amount of sharable data is significant and should be utilized and the proxy caching performance can be significantly improved by the proposed browsers-aware structure that is scalable and reliable.

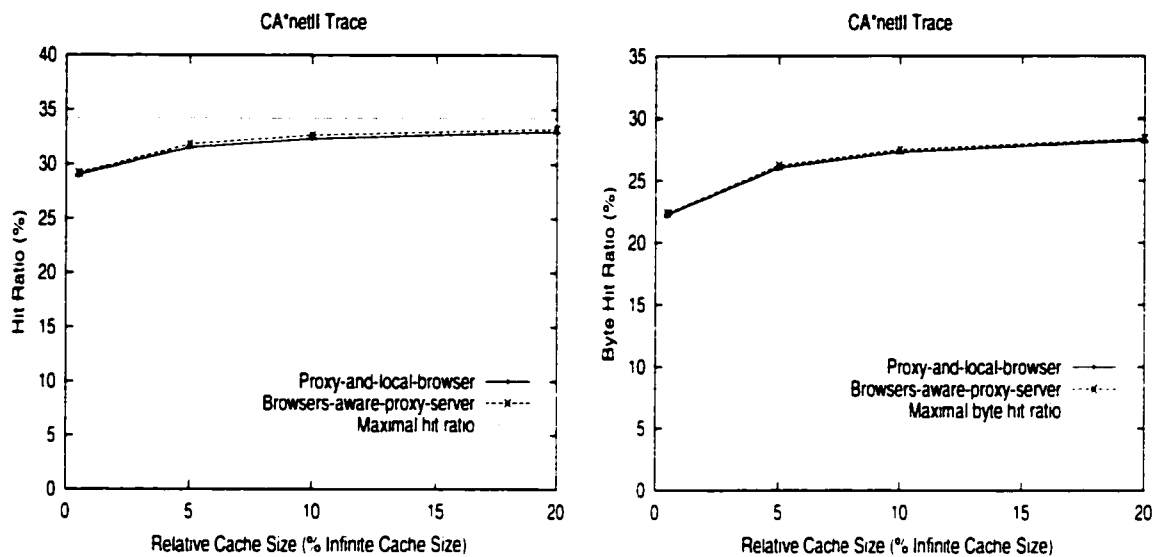


Figure 5.11: The hit ratios and byte hit ratios of the *browsers-aware-proxy-server* and *proxy-and-local-browser* scheme using the CA*netII trace.

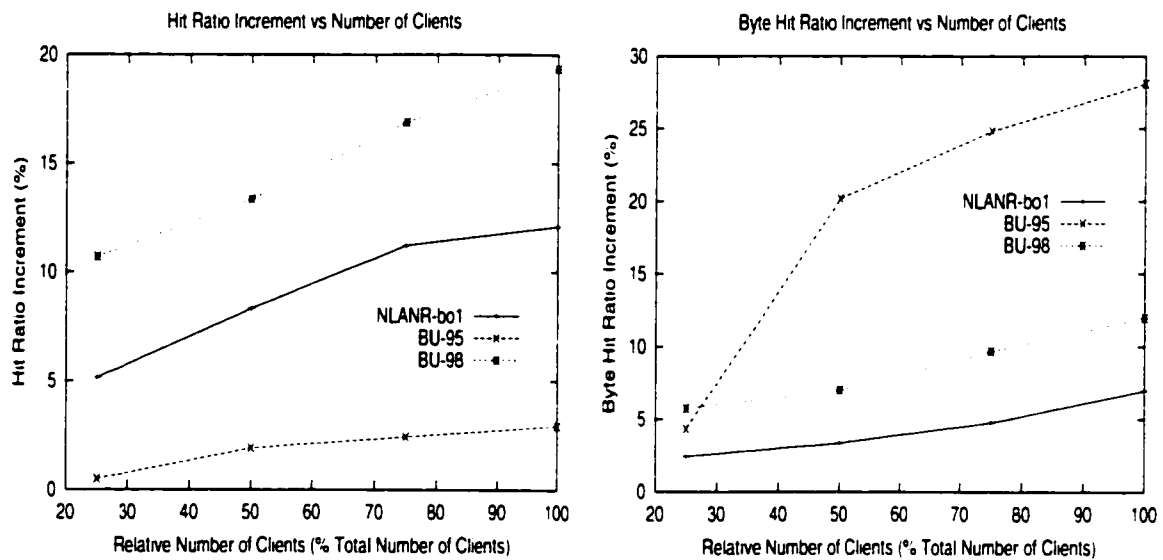


Figure 5.12: The hit ratio and byte hit ratio increments of the *browsers-aware-proxy-server* over the *proxy-and-local-browser*.

Chapter 6

Reducing Duplications in a Proxy and Its Browsers

6.1 Background and Related Work

This chapter focuses on addressing the second limit in current proxy-browser system, the duplications among the proxy cache and the browser caches. The amount of document duplication between the proxy and browser caches is generally very large because the requested document is cached in both the proxy and a requesting client browser in most cases. It is also highly possible to generate a large amount of document duplication among browsers for two reasons. First, multiple clients request some popular documents cached in the proxy. Each requesting client will duplicate these documents in its local browser cache. Second, when a request from a client is missed in the proxy cache but is a hit in another client's browser cache, the requesting client will duplicate the document in its own browser cache.

Envisioning the rapid advancement of networking technology, we argue that the duplication issue can seriously limit potential benefits to be gained from the current structure of Web caching systems. Here are the reasons. First, high speed networking technology will

soon close the speed gap between local and remote accesses. Therefore, data duplications over the Internet will truly not necessary. Second, data duplications will significantly cause additional overhead, such as global data invalidations and broadcasting. Minimizing the number of owners for a data document also strengthens security and privacy protections. Finally, unnecessary data duplications over the Internet can widely waste storage space. Both the additional operation and space overheads will certainly limit the scalability of Internet computing.

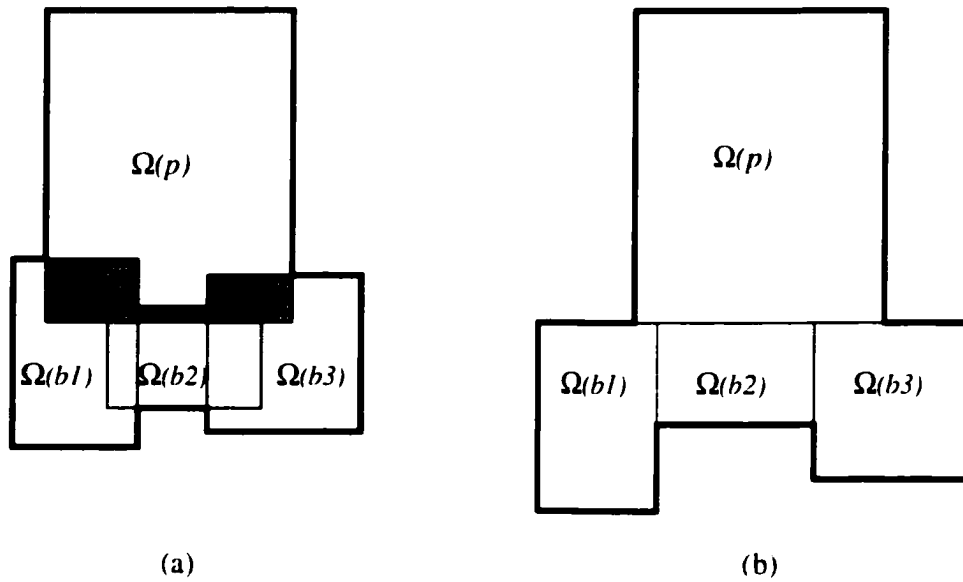


Figure 6.1: Duplication among a proxy and its client browsers.

Figure 6.1 (a) is a duplication caching scenario among the browsers and the proxy. We use $\Omega(p)$ to represent the set of documents cached in the proxy. $\Omega(b_i)$ to represent the set of documents cached in the i th browser of P browsers, where $i = 1, 2, \dots, P$. We also use $U(p)$ to represent the set of documents **only** cached in the proxy. $U(b_i)$ to represent the set of documents **only** cached in the i th browser. We aim at minimizing $\Omega(p) \cap \Omega(b_i)$ (the dark overlapped part in Figure 6.1 (a)), and minimizing $\Omega(b_i) \cap \Omega(b_j)$

(the lightly dark overlapped part in Figure 6.1 (a)), where $i, j = 1, 2, \dots, P, i \neq j$ such that $U(p) \cup U(b_1) \cup U(b_2) \dots \cup U(b_P)$ is maximized. Figure 6.1 (b) is an ideal case after the duplicated documents among the proxy and its browsers have been eliminated.

Numerous studies focus on local caching replacement policies. For example, papers [4] and [69] provide theoretical bases for approximate optimal performance and designing effective online algorithms. Papers [19] and [70] propose practical caching replacement strategies and showed promising experimental performance results. However, cooperative caching can significantly improve performance compared to local replacement [75], and has been studied in both horizontal and vertical directions.

In the horizontal direction, cooperative proxy caches are studied in many papers (e.g. [43], [56], [83], [141]), which focus on the proxies at the same level. These papers provided different approaches attempting to effectively sharing files among same level proxies, such as how to locate a file cached in another cache precisely and quickly, and how to place a file as close as possible to a proxy requesting the file with highest probability. None of these studies consider the file duplications among same level proxies. A practical reason for allowing file duplications among proxies is because proxies are normally far from each other in locations. Emphasizing eliminating file duplications too highly could cause too many requests to remote proxies so that the overall response time might be hurt. However, browsers connecting to the same proxy are usually located nearby, thus, reducing file duplications among browsers enables more files to be shared to improve overall performance.

In vertical direction, Web proxy workloads from different levels of a caching hierarchy are studied in [82]. Paper [76] develops an optimal algorithm for hierarchical placement problem. Papers [75] [123] propose practical schemes to cooperate hierarchical proxies by

hierarchical GreedyDual replacement algorithm and placement algorithm that can cache file close to clients. They conclude that hierarchical cooperative caching can significantly improve performance. The study in [40] is not so optimistic about hierarchical cooperative caching, and concludes that the performance in terms of response time can not be improved without paying careful attention to details of cooperation design to eliminate overhead, such as better distributing network traffic and avoiding congested routes. Two previous studies attempt to reduce file duplications in hierarchical cooperative caching. Paper [26] proposes a hierarchical cooperative caching architecture to avoid a requested file cached in each intermediate cache. A cache is viewed as a filter with its cutoff frequency equal to the inverse of the characteristic time. Files with access frequencies lower than this cutoff frequency have a good chance to pass through the cache without cache hits. A collaborative method is proposed in [143] for hierarchical caching in proxy servers to reduce duplicate caching between a proxy and its parent or higher level proxies in the hierarchy. In particular, a collaboration protocol passes caching decision information along with the document to the next lower level proxy to facilitate its caching decision. Our work focuses on a proxy-browser system, which is a different issue of reducing duplication in the different level proxies. Our proposed scheme not only reduces the duplications between different level caches (between proxy and browsers), but also reduces the duplications at the same level caches (among browsers).

6.2 A simulated proxy-browser caching environment

The cooperatively shared proxy-browser caching scheme is evaluated by trace-driven simulations. We use the traces of Boeing-4, Boeing-5, BU-95 and BU-98 that are described in Chapter 5. We have implemented and compared the following four Web caching organizations using the trace-driven simulations in this chapter:

1. *Proxy-and-local-browser*: If a request misses in its local browser, it will be sent to the proxy to check if the requested document is cached there. If the request is a hit in the proxy, the hit document will be cached in the browser cache of the requesting client. If the request misses in the proxy, the proxy will send the request to an upper level server. The document fetched outside the proxy-browser system will be cached both in the proxy and the browser of the requesting client.
2. *Browser-aware-proxy-server*: This is browser-aware-proxy caching technique without duplication reduction consideration, which is discussed in Chapter 5.
3. *Cooperatively shared proxy-browser caching server*: This is the cooperative caching technique proposed in this chapter, which will be discussed in Section 6.4.
4. *Offline-algorithm*: These are the offline algorithms close to optimal performance for comparisons with our proposed schemes, which will be discussed in Section 6.4.

6.3 Case Studies of Duplications in Web Caching

We have analyzed the 4 different Web traces: BU-95, BU-98, Boeing-4 and Boeing-5. These traces have been operated in a simulated system with an infinite proxy cache and infinite

Traces	BU-95	BU-98	boeing-4	boeing-5	average
hit ratio (%)	64.14	40.62	44.91	45.07	48.69
intra-sharing (%)	27.64	35.18	39.55	42.42	36.20
inter-sharing (%)	72.36	64.82	60.45	57.58	63.80

Table 6.1: Trace analysis on document duplications and sharing based on the proxy-browser system hit ratios, intra-sharing ratios, and inter-sharing ratios.

browser caches. There are two types of data sharing in Web surfing: individually requested documents by a single client, and commonly requested documents by multiple clients. We define “intra-sharing” ratio as the percentage of the requests only hit in local browsers for individual usage of clients out of the total **hit requests** in the proxy-browser system. We further define the “inter-sharing” ratio as the percentage of the requests coming from multiple clients but hitting the same documents out of the total **hit requests** in the proxy-browser system.

We have three observations based on the trace analysis results reported in Table 6.1. First, the average hit ratio of the 4 traces is 48.69%, which means that 51.31% of requested documents are only accessed once and remained in both proxy and browser caches. Second, among the total **hit requests** in the proxy-browser system, the average intra-sharing ratio is 36.20%. Since this large portion of documents is only for individual usage, the documents do not need to be cached in the proxy, but only need to be cached in the requesting local browser caches. Unfortunately, the standard Web caching model stores this high percentage of documents in the proxy. Finally, the hits for inter-sharing by multiple clients that need to be cached in the proxy is 63.80%. However, documents of this type are duplicated in the proxy cache and multiple browser caches.

Our analysis and case studies show that a significant amount of document duplication

exists in commonly used Web caching models. If supply and demand of diverse Web contents are continually increased, this duplication will soon limit the effective utilization of caching space. In addition, current Web caching models lack data sharing mechanism between the proxy and browsers with which to further exploit data locality and utilize caching space. This preliminary trace analysis motivates us to propose new caching management schemes for reducing the document duplications among a proxy and its browsers for performance improvement by utilizing more caching space.

Using the browser-aware caching model as a framework, we propose a new Web caching management model, called *cooperatively shared proxy-browser caching*, where the proxy cache is managed mainly to store the shared documents requested by multiple clients, and browsers are managed mainly to store the individually used documents. The objective of this caching management scheme is to effectively enlarge the caching space for clients by significantly reducing the document duplications among a proxy and its client browsers, and to significantly reduce the traffic to Web servers.

6.4 Cooperatively Shared Proxy-Browser Caching Scheme

6.4.1 An outline of the scheme

Upon a client request, the cooperatively shared proxy-browser caching scheme provides the following data flows for document service and storage management:

1. If the request is a hit in the local browser, the document will be read from the browser cache.
2. If the request misses in its local browser cache but hits in the proxy, then, in addition

to providing the document, the proxy will increment the counter of the number of remote accesses to this document from this requesting client. The proxy will inform the requesting client to cache this document only if the value of this counter is larger than a pre-determined threshold, TH_BROWSER.

3. If the request is a miss in the local browser and the proxy, the index file (which maintains the records of all documents cached in client browsers) in the proxy will be searched to see if the document is cached in another browser cache. If the request is a hit in another client's browser cache, then the hit browser will do two bookkeeping operations besides providing the document: (1) increment the counter of the total number of distinct remote requesting clients to this document if the requesting client accesses this document for the first time, and (2) increment the counter of the number of remote accesses to this document from this client. If the first counter is larger than a pre-determined threshold, TH_PROXY, it means this document is shared by a sufficient number of clients so that the hit browser will transfer and cache the document to the proxy. The requesting browser is informed to cache this document only if the value of the second counter is larger than TH_BROWSER.
4. When the request is missed in the entire proxy-browser system, the requested document will be provided by an upper level proxy or a Web server. The initial document coming externally will be cached only in the requesting browser. However, if the proxy cache has enough free cache space for the document, it can be cached in the proxy at the same time.

In the above items 2, 3, and 4, the document may be cached in either a browser cache or the proxy. When the browser cache or the proxy cache does not have sufficient space to store the document, one or more currently cached documents have to be replaced. LRU_Threshold (which does not cache a document larger than a threshold size) is used as the basic replacement policy for our scheme. (Most practical systems use algorithms similar to LRU_Threshold [105]). For a document larger than the threshold, our scheme also caches it as long as the cache has enough free space, but it is marked as an LRU document. The cache size threshold used in LRU_Threshold in the proxy and a browser cache is different due to significant difference of their cache sizes.

6.4.2 Data structures and operations

Two structures are maintained to facilitate this scheme. One structure allocated in each browser is used to manage cached documents in it. Another structure allocated in the proxy is used to manage all documents cached there.

6.4.2.1 The structure in each browser

A counter and an array is allocated for each cached document that has been requested by other clients. The counter *CC* keeps the number of other clients that have accessed the document. Its value will be used to check if this document should be cached in the proxy. Each element of the array *AC* has two fields: *Client_Alias* and *Access_Count*. An *AC.Client_Alias* records a client who has accessed the document. *AC.Client_Alias* is produced by the proxy to hide the true identity of the requesting client. The aliases are consistent and untraceable as in LPWA [54]. *AC.Access_Count* records the number

of accesses from the corresponding client. The array is of size TH_PROXY, of which CC elements are in use. It is allocated for a document only if there is a remote client requesting this document. When a document is replaced, the counter and array for this document are also replaced.

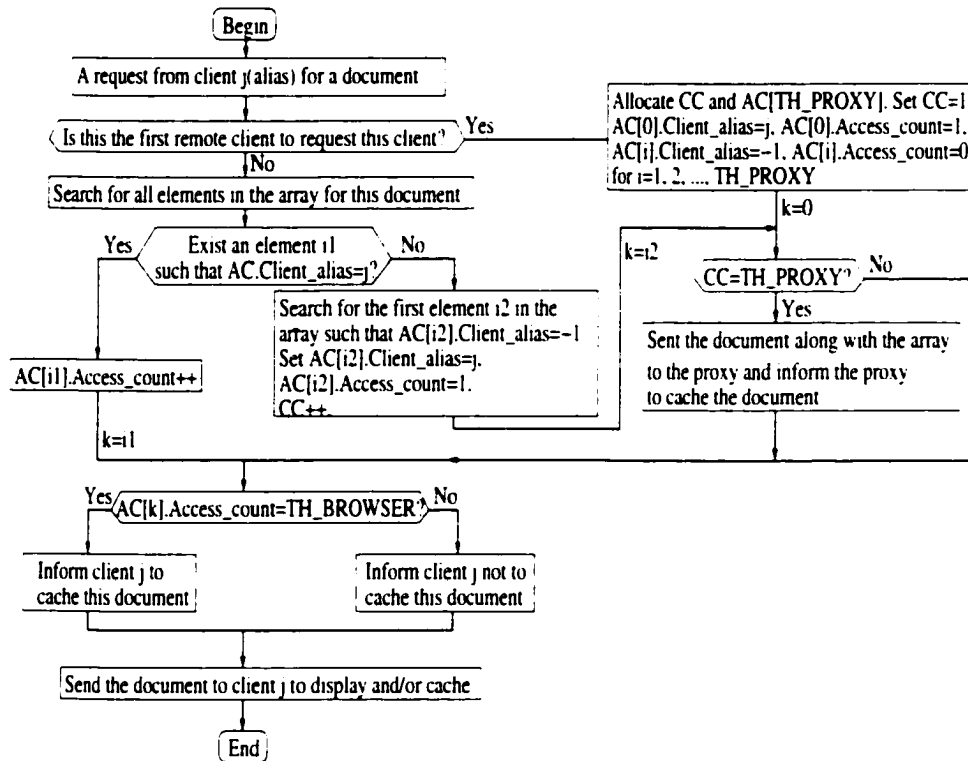


Figure 6.2: The management operations in each browser when a remote client request hits in it.

Figure 6.2 presents the management operations when a remote client requests a document cached in this browser. When a browser is informed to cache a document sent by another client, it will cache this document.

6.4.2.2 The structure in the proxy

Each cached document in the proxy needs to count the number of accesses to this document from different requesting clients. This is used to check if this document should be duplicated

in a requesting client browser. We use a linked list for each document. Each element of the list *LL* includes three fields: (1) *Client_ID*: the ID number of a requesting client; (2) *Access_Count*: the number of requests from this client; and (3) *Pointer*: a pointer to link to the next element. A new element is allocated to the linked list of a document only if this document is requested by a client for the first time. When a document is replaced, its linked list is also replaced.

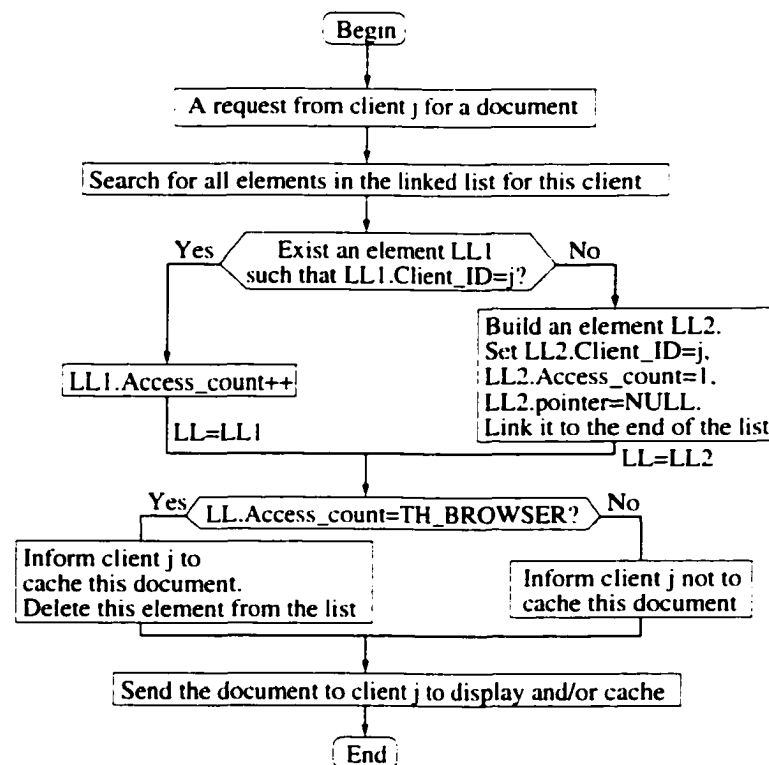


Figure 6.3: The Management operations in the proxy when a client request hits in the proxy.

Figure 6.3 presents the management operations when a client request hits in the proxy. When the proxy is informed to store a document sent by a client, the proxy first caches the document, and then copies the necessary elements of the array sent by the client to the corresponding fields in the newly created linked list. The necessary elements are those

with $AC.Client_ID \neq -1$ and $AC.Access_Count < TH_BROWSER$. When the proxy has to fetch a document outside the proxy-browser system, it will pass the document to the requesting client and inform the client to cache it. The proxy will not cache a new arrival document.

6.4.3 Offline Algorithms for Performance Comparisons

The goal of obtaining optimal hit ratio and byte hit ratio in a proxy-browser system is equivalent to finding optimal replacement algorithms for objects with different sizes in a single cache whose size is the accumulated size of the proxy and all browsers. Studies in [69] provide two offline algorithms that are close to the optimal replacement algorithms. These offline algorithms are not viable in practice due to their requirement of knowing future requests. However, in order to evaluate the effectiveness of the proposed schemes, we compare their performance with that of the offline algorithms. This section gives an overview of two models of cost measurement for offline Web caching algorithms and discusses the respective approaches. They both follow [69], and are also discussed in [4].

1. The **Fault Model**, where the cost of an algorithm for a request sequence σ equals the number of cache misses.
2. The **Bit Model**, where we sum up the sizes of the documents each time they are brought into cache.

Both models do not discriminate whether a document is stored in a proxy or in a browser cache.

If all documents have the same size and same costs of bringing them into cache, the *Belady's Rule* [8] is known to be optimal strategy for evicting pages from the cache: on a fault evict the *most distant page* that is the page whose next request is farthest in the future. However, for caching of Web documents with different sizes these assumptions are not appropriate. We consider therefore for each of the above models a separate offline algorithm.

For the Fault Model, we use the Offline Fault Model Algorithm (OFMA) [69], which is shown in Algorithm 1. It guarantees that for any request sequence σ the number of cache misses is within the factor $2 \log k$ of the number of cache misses for an optimal offline algorithm. Here k is the ratio between the largest and the smallest document in σ .

Algorithm 1 Offline Fault Model Algorithm [69]

Divide the documents into at most $\lfloor \log k \rfloor + 1$ l -classes C_l ,
 where C_l holds the documents of sizes $[2^l, \dots, 2^{l+1} - 1]$.
 for each request to a document d in a l -class:
 if d is not in the cache:
 bring it in.
 if size of the cache is exceeded:
 for all j , do twice:
 if C_j is not empty:
 evict the most distant document in C_j .

As for the BIT Model, we use the Offline Bit Model Algorithm (OBMA) from [69]. The cost of this algorithm is essentially within the factor $5(\log k + 4)$ of the optimal offline algorithm. As in OFMA, the documents are first divided into l -classes. If the cache capacity is exceeded by h when a new object is cached, OBMA evicts from every class most distant objects until there is enough room or there are no objects in that class. In order to avoiding evicting a large page when the cache is only exceeded by a small amount, OBMA maintains a counter for each class. If h is smaller than the most distant object in a class, it is added

to the counter of this class. When the counter is larger than the size of the object, OBMA evicts the object and subtracts the size of the object from the counter.

The goals of the Fault Model and the Bit Model are to maximize hit ratios and byte hit ratios, respectively. In order to show how close the performance of the proposed caching scheme is to the optimal one, we compare the performance of our scheme with that of the two offline algorithms.

6.5 Performance Evaluation

We have evaluated and compared the performance of the *proxy-and-local-browser*, *browser-aware-proxy-server*, *cooperatively shared proxy-browser caching* (which will be simplified as *cooperative-caching* in the rest of the chapter) schemes, and the *offline-algorithm* using the two BU browser traces and two Boeing traces. In the comparisons of hit ratios, the offline Fault Model Algorithm is used, while in the comparisons of byte hit ratios, the offline BIT Model Algorithm is used. We will discuss the performance sensitivity to four important parameters: proxy cache size, browser cache size, cache size threshold for replacement, and the number of clients. We use *ps* to denote proxy cache size, which is based on the percentage of infinite proxy cache size. We use *bs* to denote browser cache size, which is based on the value of β . We assume that all browsers have the same size. We use *th* to denote cache size threshold used in LRU_Threshold cache replacement policy, which is a ratio of a given cacheable document threshold size over the proxy (or browser) cache size.

6.5.1 Evaluation of the sensitivity to the proxy cache size

We have examined how sensitive the hit ratios and byte hit ratios are to the changes of the proxy cache size. For the experiments of each input trace, we set ps to 1%, 2%, 3%, 5% and 10% of the infinite proxy cache size. We set $\beta = 10$. We also choose $th = 0.5$, which means the proxy size threshold is half of the proxy cache size, and the browser size threshold is also half of browser cache size. Our trace-driven simulations show that our *cooperative-caching* consistently outperforms *browser-aware-proxy-server* and *proxy-and-local-browser* for all the traces measured by hit ratios and byte hit ratios, in Figures 6.4 - 6.7, respectively.

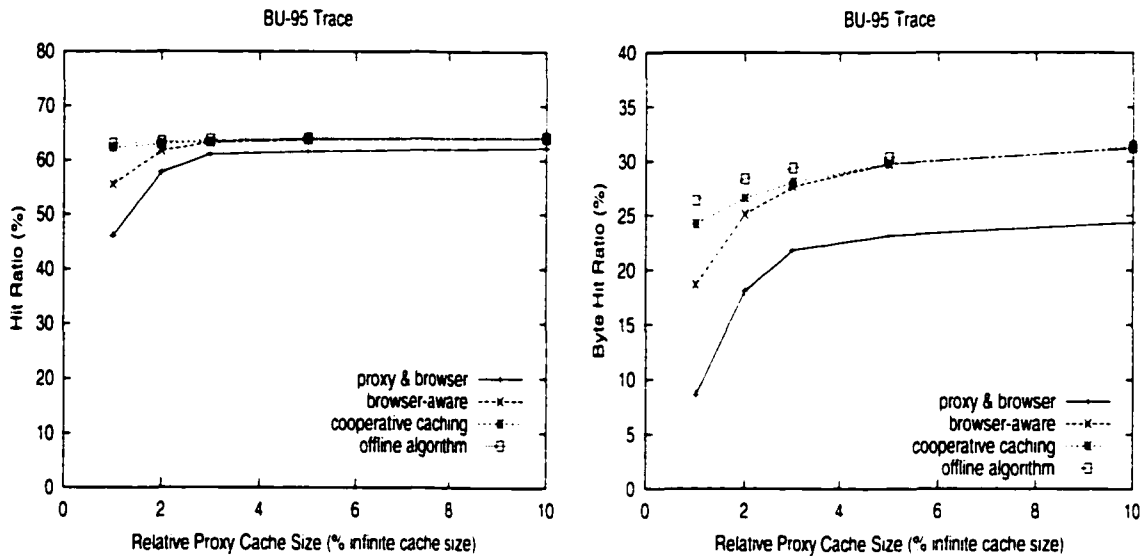


Figure 6.4: Hit ratio and byte hit ratio of the three caching schemes versus relative proxy cache sizes using BU-95 trace ($\beta=10$, $th=0.5$).

We first compare the performance of browser traces BU-95 and BU-98 (see Figures 6.4 and 6.5). For *proxy-and-local-browser*, BU-98's hit ratio is much lower than BU-95's hit ratio, but is also much lower than BU-98's hit ratio of *offline-algorithm*, which means that the hit ratio of the BU-98 trace has much more potential for improvement, while the hit

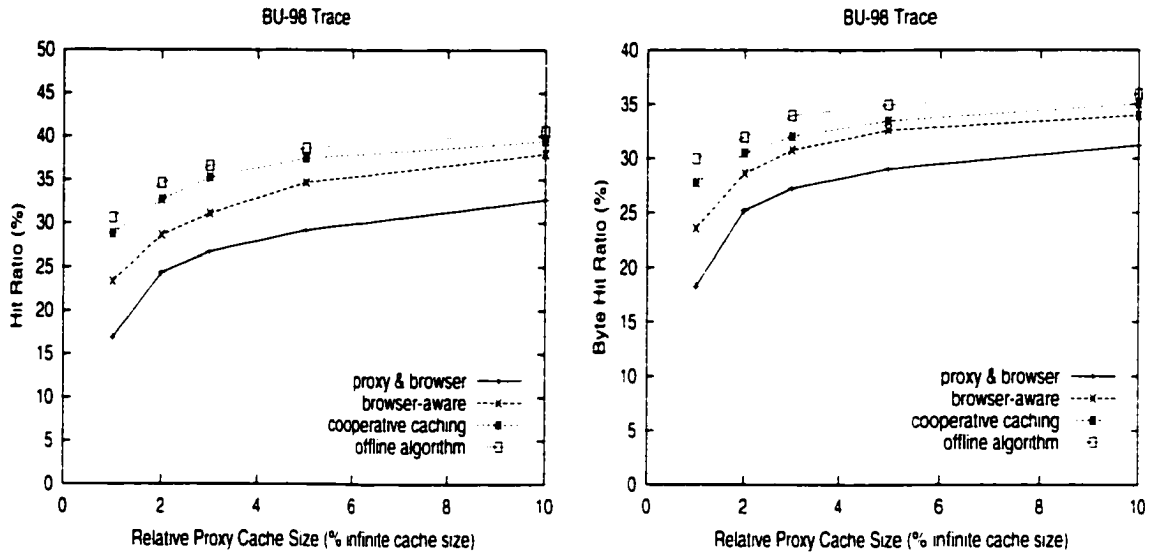


Figure 6.5: Hit ratio and byte hit ratio of the three caching schemes versus relative proxy cache sizes using BU-98 trace ($\beta=10$, $th=0.5$).

ratio of BU-95 trace has almost no room for improvement because it is so close to *offline-algorithm*. Both traces' byte hit ratios of *proxy-and-local-browser* have similar performance gaps with *offline-algorithm*. Our scheme of *cooperative-caching* improves hit ratios and byte hit ratios of both traces, which are very close to *offline-algorithm*. As an example of $ps = 5\%$, the *offline-algorithm* outperforms *cooperative-caching* by only 3.03% and 4.26% measured by hit ratio and byte hit ratio for BU-98. So *cooperative-caching* is more promising to improve year 1998's trace than the trace three years before, because requests in the year 1998's trace are more evenly distributed.

Boeing-4 and boeing-5 are proxies traces, but we still see a big performance gain from *cooperative-caching*. The intra-network overhead simulation for these two traces in Section 6.6.1 shows that the increase of intra-network overhead of *cooperative-caching* is trivial, which does not offset the (byte) hit ratio gain from this scheme. So reducing document duplications among cooperative proxies in the same organization is still promising for per-

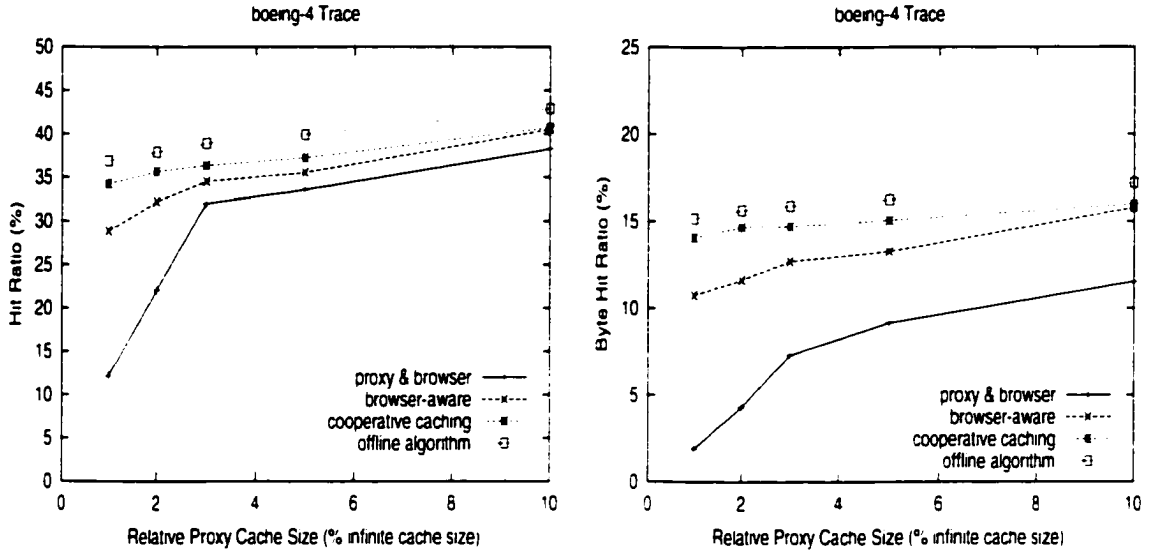


Figure 6.6: Hit ratio and byte hit ratio of the three caching schemes versus relative proxy cache sizes using Boeing-4 trace ($\beta=10$, $th=0.5$).

formance. But it is not desirable for higher level proxies, which are closer to servers and farther to clients, because long distances among these proxies and potential networking congestion may offset (byte) hit ratio gains so that response time can not be improved [40].

The percentage (ps) reflects the ratio between the actual proxy cache size and the accumulated size of unique documents. If the increase of the numbers of servers and of the diverse client populations is faster than the increase of the proxy cache size, the relative proxy cache size (ps) will continue to decrease. In other words, our *cooperative-caching* scheme will be more performance-beneficial as Web servers and Web client populations continue to increase in both numbers and types.

6.5.2 Evaluation of the sensitivity to a browser cache size

We have examined how sensitive the hit ratios and byte hit ratios are to the changes of a browser cache size. For the experiments of each input trace, we set β to 0.1, 1, 5, 10, 15, 20.

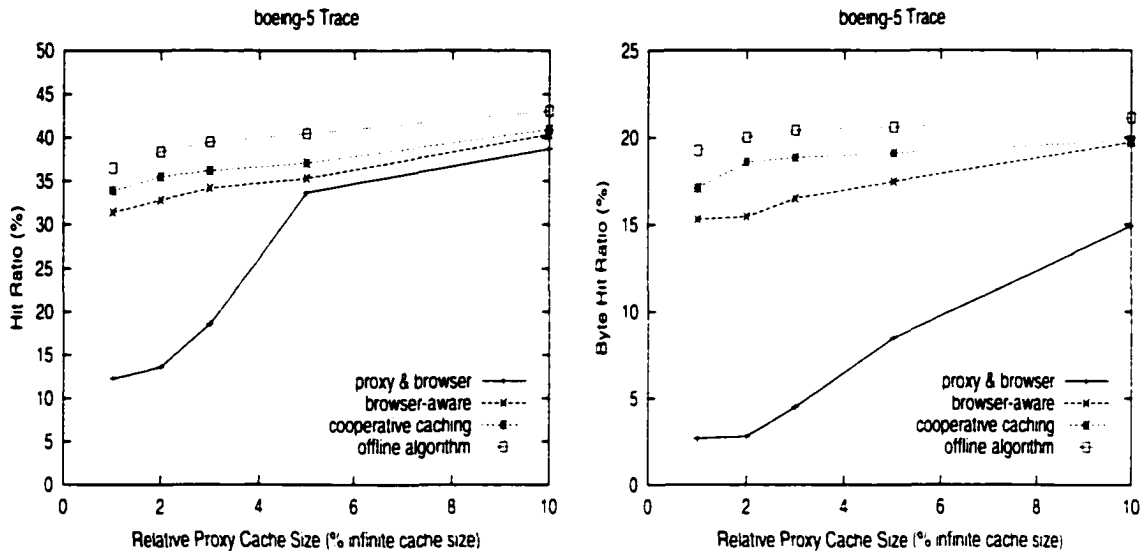


Figure 6.7: Hit ratio and byte hit ratio of the three caching schemes versus relative proxy cache sizes using Boeing-5 trace ($\beta=10$, $th=0.5$).

and 50, respectively. We set ps to 1% of the infinite proxy cache size, and choose $th=0.5$.

Our trace-driven simulations show that our *cooperative-caching* consistently outperforms *browser-aware-proxy-server* and *proxy-and-local-browser* for all the traces with all the given β values measured by hit ratios and byte hit ratios in Figures 6.8 - 6.11, respectively. The performance gain of all the schemes is improved slowly after β reaches a certain value. The best performance gain was achieved for β in the range of 1 to 15. If β is too small, such as less than 0.1, the accumulated browser cache is not large enough to be effective for both *cooperative-caching* and *browser-aware-proxy-server*. It is also not desirable to increase β to a very large value. (The paper in [131] also points out this). For the examples in section 5.3.1, the range of 1 to 15 of β corresponds a browser cache size in the range of 10 MBytes to 150 MBytes, which is a reasonable range of browser cache size in current disk storage capacity of workstations.

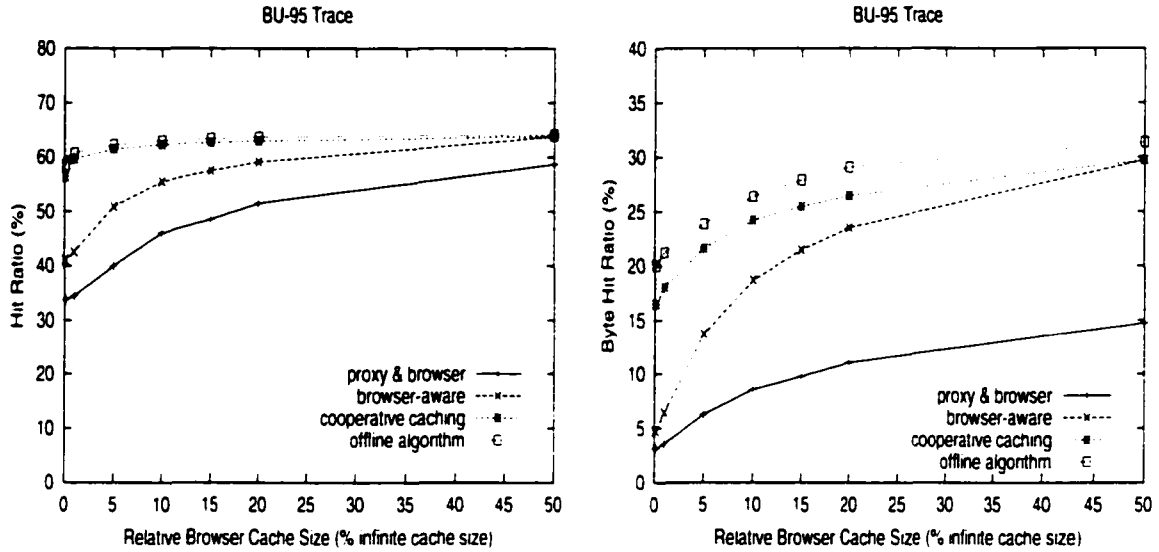


Figure 6.8: Hit ratio and byte hit ratio of the three caching schemes versus relative browser cache sizes using BU-95 trace ($ps=1\%$, $th=0.5$).

6.5.3 Evaluation of the sensitivity to the replacement threshold

We use the basic *LRU_threshold* cache replacement policy in both *proxy-and-local-browser* and *browser-aware-proxy-server*. We have revised the *LRU_Threshold* policy for *cooperative-caching*, where a document larger than the threshold could be cached as long as enough free caching space is available but is marked as an LRU document. We have examined how sensitive the hit ratios and byte hit ratios are to changes of the replacement threshold. For experiments of each trace, the th variable is set to $\frac{1}{64}$, $\frac{1}{16}$, $\frac{1}{8}$, $\frac{1}{4}$, $\frac{1}{2}$, $\frac{3}{4}$, $\frac{5}{6}$, and 1, respectively. We set ps to 1% of the infinite proxy cache size, and choose $\beta = 10$.

Our trace-driven simulations show that *cooperative-caching* consistently outperforms *browser-aware-proxy-server* and *proxy-and-local-browser* for all the traces with all the given relative thresholds measured by hit ratios and byte hit ratios in Figures 6.12 - 6.15.

Our experiments show that in general small cache threshold values are more effective

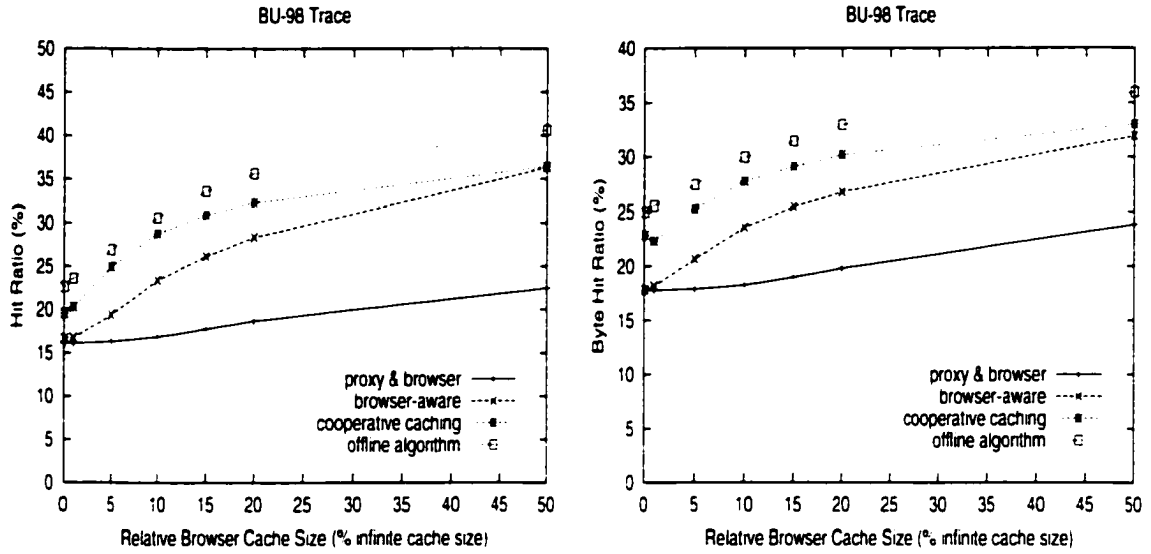


Figure 6.9: Hit ratio and byte hit ratio of the three caching schemes versus relative browser cache sizes using BU-98 trace ($ps=1\%$, $th=0.5$).

for *cooperative-caching* than large threshold values measured by the hit ratios. This is because file size distribution is heavy-tailed [6]. The average size of popular documents is smaller than that of unpopular documents. But a very small threshold is not beneficial to performance measured by byte hit ratios. Comparing hit ratios of *cooperative-caching* for trace BU-95 and BU-98, we show that small cache threshold values are more effective for BU-98 trace. This can be explained by the findings in [6]: BU-98 trace shows a shift toward smaller sizes overall than BU-95 trace. The threshold impact to (byte) hit ratios of *cooperative-caching* is much less sensitive than those of *browser-aware-proxy-server* and *proxy-and-local-browser* for browser traces.

6.5.4 Performance Impact of Scaling the Number of Clients

We have also evaluated the effects of scaling the number of clients to *cooperative-caching*.

Figure 6.16 presents the hit ratio increment curves (left figure) and the byte hit ratio

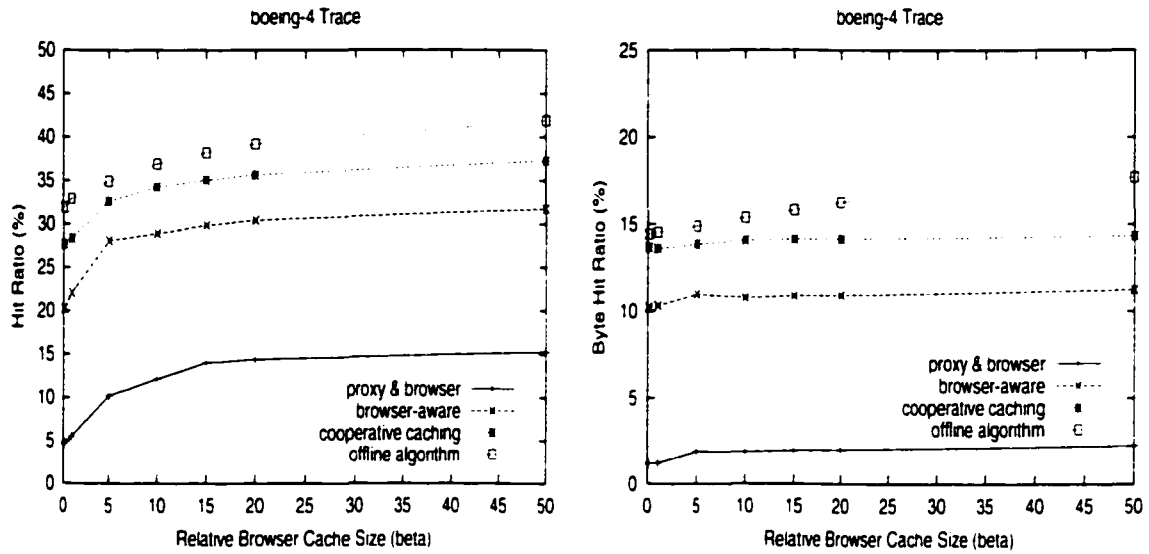


Figure 6.10: Hit ratio and byte hit ratio of the three caching schemes versus relative browser cache sizes using Boeing-4 trace ($ps=1\%$, $th=0.5$).

increment curves (right figure) of the five traces as the relative number of clients changes from 25% to 100%. Our trace-driven simulation results show that both hit ratio increment and byte hit ratio increment of the browser-aware proxy server proportionally increases as the number of clients increases. For some traces, the increments are significant. For example, the hit ratio increment of BU-98 trace increases from 16.89% to 23.85%, to 28.13%, and to 34.13%, as the relative number of clients increases from 25% to 50%, to 75%, and to 100%, respectively. The byte hit ratio increment of Boeing-5 trace increases from 36.35% to 46.34%, to 52.92%, and to 66.02%.

The performance results indicate that *cooperative-caching* scales very well for the traces with up to 3996 clients because it will exploit more browser locality and utilize more space as the number of clients increases in the cluster.

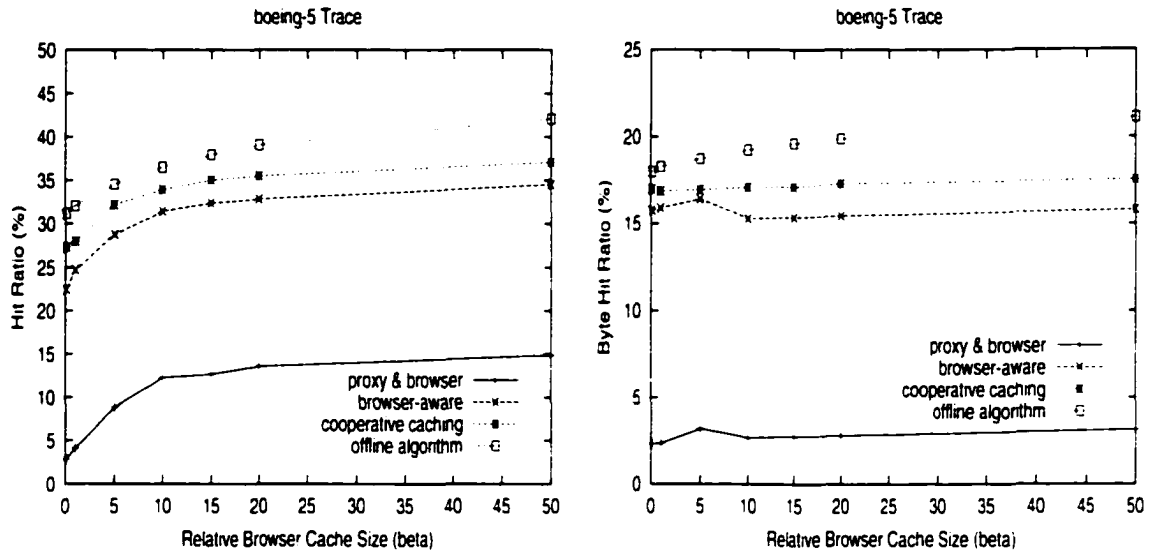


Figure 6.11: Hit ratio and byte hit ratio of the three caching schemes versus relative browser cache sizes using Boeing-5 trace ($ps=1\%$, $th=0.5$).

6.5.5 Latency Reduction

The access delay for fetching a missed document in the proxy cache from a remote server can be estimated by the summing the network connection time and the data transferring time in the Internet. We estimated connection times and data transferring times by using the method presented in [70], where the connection time and the data transferring time are obtained by applying a least squares fit to measured latency in traces versus the size variations of documents fetched from different remote servers. The access latency to remote servers reduced by the *cooperative-caching* can be further estimated by accumulating the latency times used to access remote servers for those requests missed in *browser-aware proxy server* or *proxy-and-local-browser*, but hit in *cooperative-caching*. Our experiments show that the *cooperative-caching* achieves average latency reduction of 21.25%, compared with the *browser-aware-proxy-server* scheme, and about 56.61%, compared with the *proxy-*

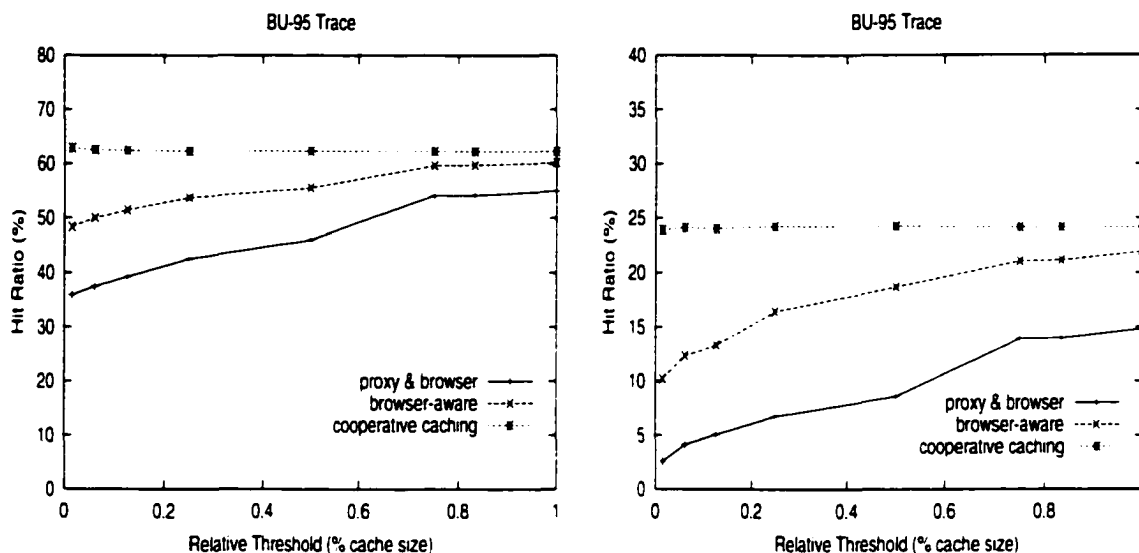


Figure 6.12: Hit ratio and byte hit ratio of the three caching schemes versus the replacement threshold using BU-95 trace ($ps=1\%$, $\beta=10$).

and-local-browser scheme.

6.6 Overhead Analysis

The overhead associated with the *cooperative-caching* comes from communications among the proxy and its client browsers (intra-network overhead), the additional space for building management data structures, and the CPU time spent on the management operations. We will discuss these three types of the overhead in this section.

6.6.1 Intra-network Overhead

The additional overhead of *cooperative-caching* comes from (1) the time spent on data transferring between two browsers for hits in remote browsers, (2) the time spent on data transferring from the proxy to a browser for hits in the proxy cache (these hit documents might be hit in requesting browsers by *proxy-and-local-browser* but not by the *cooperative-*

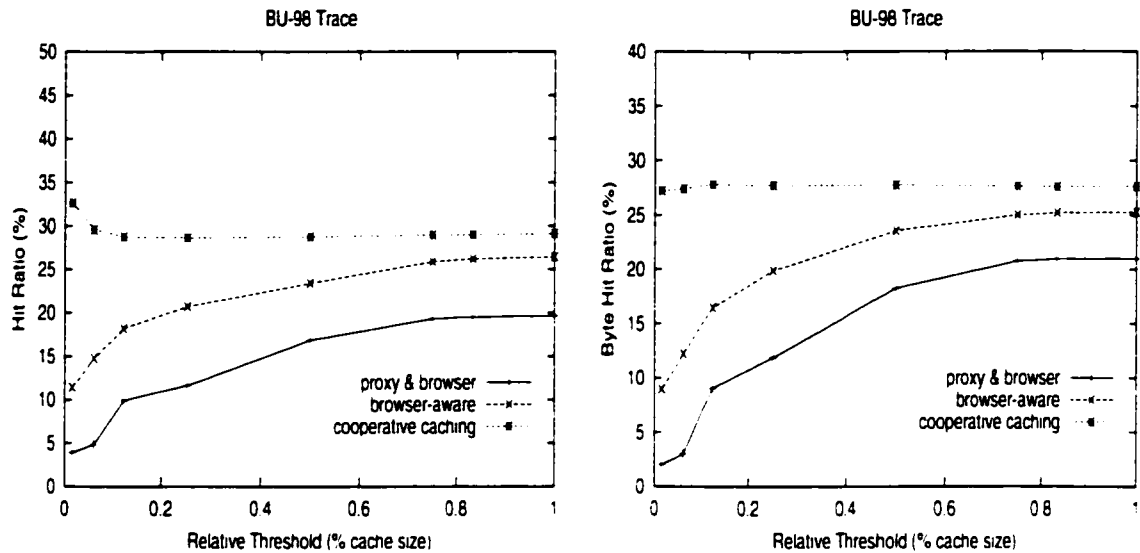


Figure 6.13: Hit ratio and byte hit ratio of the three caching schemes versus the replacement threshold using BU-98 trace ($ps=1\%$, $\beta=10$).

caching scheme). (3) the time spent to transfer documents from a client browser to the proxy due to requests by multiple clients.

We estimated the data transferring times from the above three sources on a 10 Mbps Ethernet in our simulation. The browser access contention is handled as follows. If multiple requests ask for bus service simultaneously, the bus will transfer documents one by one in FIFO order distinguished by each request's arrival time. Our experiments based on the *ping* facility show that the startup time of data communications among the clients in our local area network is less than 0.01 second. Setting 0.01 second as the network connection time. Table 6.2 presents the maximal intra-network overhead for each trace with different parameters in simulations. Column "Time" is the total workload service time. Column "communication" shows the additional intra-network latency time and the percentage of this latency out of the total workload service time. Column "contention" is the waiting time due to the additional intra-network communication contention for the bus and the

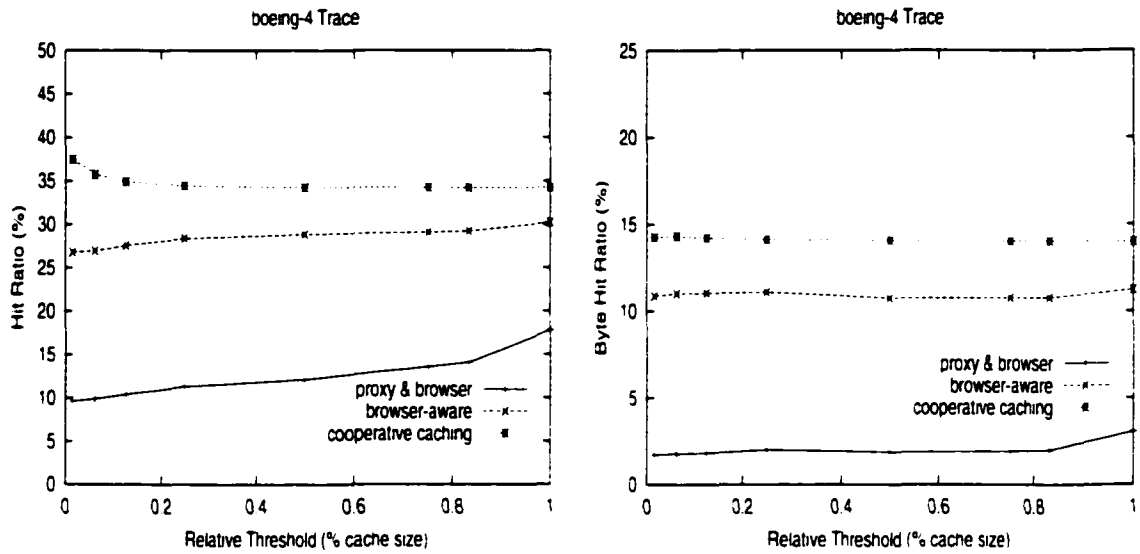


Figure 6.14: Hit ratio and byte hit ratio of the three caching schemes versus the replacement threshold using Boeing-4 trace ($ps=1\%$, $\beta=10$).

percentage of the waiting time out of the “communication” time. In this table, we also present the intra-network overhead of *browser-aware-proxy-server* that comes from the time spent on data transferring between two browsers for the hits in remote browsers.

We show that the amounts of data transferring times and the bus contention times spent for communications among the proxy and clients of the *cooperative-caching* scheme on all traces are very low. For example, the largest accumulated communication and network contention portion out of the total workload service time for all traces, is less than 1.51%. In addition, the largest contention time of *cooperative-caching* is 0.01 seconds, which only contributes up to 0.004% of the total communication time. This implies that *cooperative-caching* does not cause bursty communications in a proxy-browser system. Notice that “communication” times of *cooperative-caching* are even smaller than that of *browser-aware-proxy-server* for browser traces BU-95 and BU-98, which shows that *cooperative-caching* can place documents in more suitable places for these two traces than *browser-aware-proxy-*

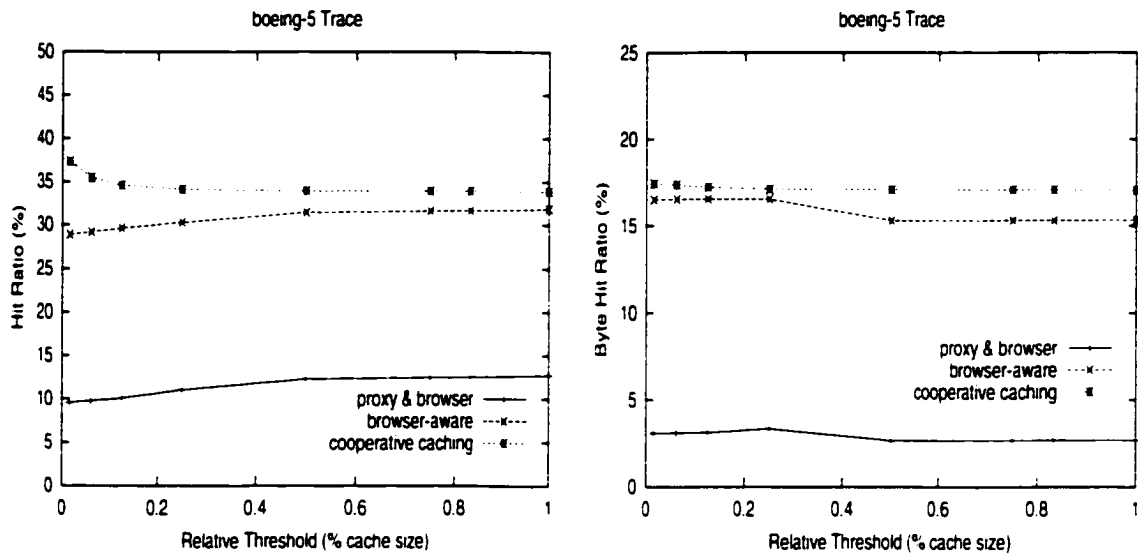


Figure 6.15: Hit ratio and byte hit ratio of the three caching schemes versus the replacement threshold using Boeing-5 trace ($ps=1\%$, $\beta=10$).

server.

Traces	Time (s)	Browser-aware-proxy-server		Cooperative-Caching	
		communication	contention	communication	contention
BU-95	3668313	2501s (0.068%)	0.01s (0.0004%)	232s (0.006%)	0.01s (0.004%)
BU-98	4164302	167s (0.005%)	0.01s (0.006%)	43s (0.001%)	0.01s (0.003%)
Boeing-4	86382	668s (0.77%)	0.0022s (0.0004%)	1304s (1.51%)	0.0004s (0.00003%)
Boeing-5	86176	741s (0.86%)	0.011s (0.0017%)	1255s (1.46%)	0.0052s (0.0005%)

Table 6.2: Intra-network Overhead

6.6.2 Space Overhead

The additional space of *cooperative-caching* is allocated for two data structures keeping track of reference counts to manage data placement.

First, linked lists are used in the proxy to count the number of accesses to the same documents from different requesting browsers. The size of this space requirement depends on TH_BROWSER and the number of clients to access this document. The value of

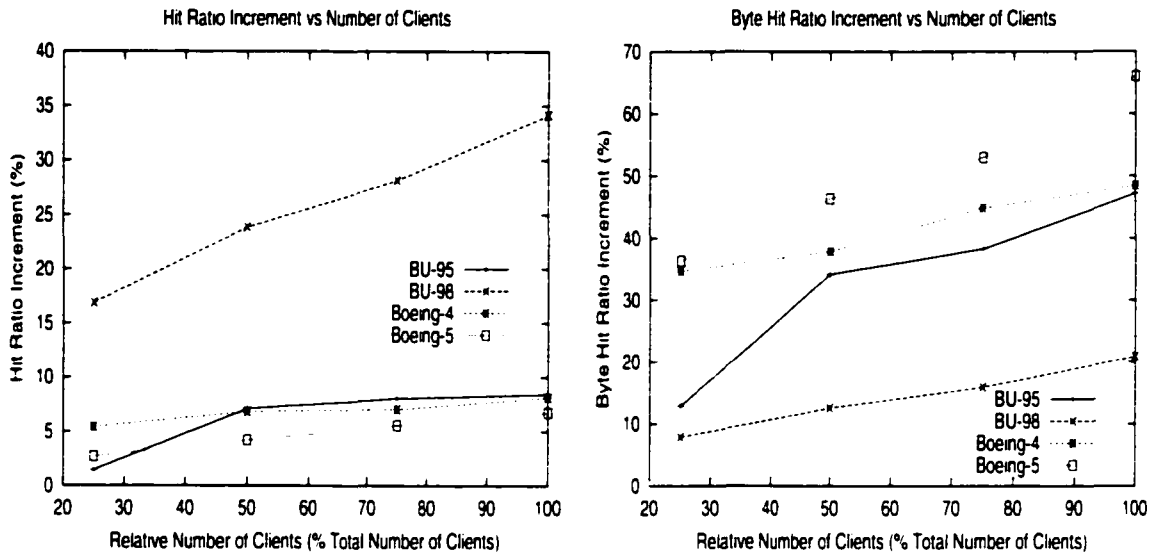


Figure 6.16: The hit ratio and byte hit ratio increments of the *cooperative-caching* over the *proxy-and-local-browser*.

TH_BROWSER reflects the trade-off between the amount of document duplications and intra-network communication overhead. Our simulation results show that an optimal range of TH_BROWSER is 3 to 5. We use 5 to estimate the space requirement. Our simulation results also show that the average number of clients to access one document is less than 6. For each element in the list, we use 2 bytes for *LL.Client.ID*, 1 byte for *LL.Access.Count*, and 5 bytes for *LL.Pointer*. The 2 bytes can record up to 65,536 different clients. The 1 byte can represent up to 256 accesses which is much larger than the optimal TH_BROWSER we used. The 5 bytes could represent up to 1024 G address space. We assume that the proxy has a 32 GByte cache, and an average document size is 8 KByte. The proxy has about 4 M Web pages. The proxy needs to allocate $(32GB/8KB) \times (2 + 1 + 5) \times 6 = 192$ MBytes for the linked lists, which only occupies 0.59% of the proxy cache, and can be easily placed in the main memory of a proxy server.

Second, a counter and a structure array is allocated for each cached document that has

been requested by other clients. The array size is `TH_PROXY`. The value of `TH_PROXY` also reflects the trade-off between the amount of document duplications and intra-network communication overhead. Our simulation results show that an optimal range of `TH_PROXY` is 3 to 7. We use 7 in our calculation, which will overestimate the space requirement. For each element of the array, we allocate 2 bytes for `AC.Client_ID`, and 1 byte for `AC.Access_Count`. One byte is also enough for the counter because we use `TH_PROXY=7` here. We assume that each client has a large browser cache with a 80 MByte cache, and an average document size is 8 KByte. Each browser has about 10 K Web pages. The browser needs to allocate about $(80MB/8KB) \times (7 \times (2 + 1) + 1) = 220$ KBytes, which only occupies 0.27% of a browser cache. This requirement is overestimated because the array and the counter are allocated to a document only if this document is accessed by other clients.

6.6.3 CPU Overhead

In a browser, the additional CPU overhead comes from searching structure arrays. The size of each array is `TH_PROXY`. As we mentioned previously, an optimal range of `TH_PROXY` is 3 to 7. So handling such a search for each request from a remote client requires $O(1)$ time.

In the proxy, the additional CPU overhead comes from searching a linked list for a hit request. The CPU time requirement for handling such a search for a document from a remote client depends on the number of clients that have requested the document. As we mentioned previously, the average number of clients to access one document is less than 6. Thus, handling such a search requires $O(1)$ time in average. But it is possible that there is a long list for one document. The following strategies have been applied to alleviate

this possible delay. First, an element for a client will be deleted from the list when the document has been requested as many times as TH_BROWSER because the client has been informed to cache this document. Second, the list search can be overlapped with passing the document to a client. In detail, when a client request hits in the proxy, the proxy first sends the requested document to the client. The client will spend some time to view the document. At the same time, the proxy searches for the list of this document to check how many times this client has requested this document. Afterwards, the proxy will inform the client whether to cache this document or not depending on the searching result. The searching process will not delay response times to the clients.

6.7 Chapter Conclusion

We have demonstrated trends of decreasing proxy hit ratios and increasing access diversity, and significant document duplications in Web caching systems. In order to effectively utilize the increasingly proxy and browser caching space, we propose a peer-to-peer Web caching management scheme, called *cooperatively shared proxy-browser caching*. We show that the performance of our scheme compares very favorably with the performance of near-optimal offline Web caching algorithms.

Chapter 7

Data Integrity and Communication Anonymity

7.1 Introduction

In order to make the browsers-aware proxy server feasible in practice, the reliability and security of the browser data must be seriously considered [134]. For example, the browser data files that have been modified by an owner client are not reliable for sharing among clients. In addition, the identities of a requesting browser and a hit browser, and the hit document should not be visible among clients to preserve the privacy of each client. These concerns can be addressed by ensuring data integrity and making anonymous communications between clients.

We have proposed protocols to enforce data integrity and communication anonymity [139]. This part of the work was in collaboration with Zhichen Xu at Hewlett Packard Laboratories. Our study shows that the associated overheads are trivial. These protocols are based on symmetric and public key encryptions [88]. In a symmetric key system, two communicating parties share an identical secret, the symmetric key, used for encryption

and decryption. DES (Data Encryption Standard) is such an example. In a public key system (e.g. RSA), such party has a public/private key pair. A public key can be accessed by everyone. A sender encrypts an outgoing message using a receiver's public key, and the receiver uses its private key to decrypt this ciphertext. DES is much faster than RSA. A practical way is to combine DES and RSA, for example, to use DES to encrypt a large message and use RSA to encrypt the DES key.

7.2 Data Integrity

To ensure that a document received by a client is tamper-proof, we need to find a way for a requesting browser to check whether the content it receives is intact. For this purpose, we use the proxy server to produce a digital water mark in the following manner: for a document f , the digital water mark is produced by first generating a message digest using MD5 [104], and then encrypt the message digest with the proxy server's private key. (We assume that the private key of the proxy is x , the corresponding public key is y , and the public keys of the browser caches are known to all peer clients. We use $K(M)$ to represent either (i) the message M being encrypted with the key K , or (ii) the ciphered message M being decrypted with decryption key K .)

Figure 7.1 shows the integrity protocol. Initially, when a client c_i sends a request to the proxy for a document, the proxy obtains the requested document, denoted as f , either from the server or an upper level proxy. The proxy generates a MD5 message digest, $h(f)$, of the document. It then encrypts $h(f)$ with its private key x to produce $x(h(f))$. The message $\{f, x(h(f))\}$ is sent to the client c_i and stored in its local cache. If another client c_j

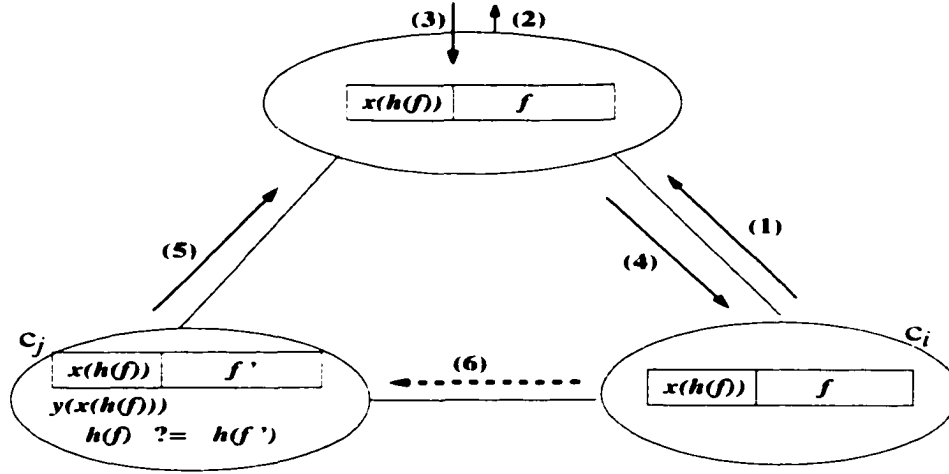


Figure 7.1: Integrity Protocol

requests the same document, and this document has been replaced in the proxy cache and is found to be in c_i 's cache, the proxy will instruct c_i to send the message $\{(h(f)), f\}$ to c_j . On receiving the message, c_j will produce a message digest of the document using MD5, and compare the message digest with $y(x(h(f)))$. No client can tamper with the document f and produce a matching digital water mark, because no client but the proxy server knows the private key of the proxy server.

7.3 Anonymity Issue

One important problem in peer-to-peer (P2P) systems is to enforce the trust of the data stored in the system and the security of the peers. So we extend our study on anonymity to generalized peer-to-peer systems. In a P2P system, each peer can play three different roles: as a *publisher* to produce documents; as a *provider* (or a *responder*) to host and deliver documents upon requests; as a *requester* (or an *initiator*) to request documents. In some systems, a provider and a publisher can be the same peer for the same document. In some

other systems, a provider and a publisher are different peers for the same document for various reasons. For example, a publisher can distribute its documents to other provider peers in order to resist censorship; and documents can also be cached in some non-producer peers.

Depending on circumstances, applications and users of a system may require different levels of anonymity. It is desirable in practice that the identity of a publisher be hidden to resist censorship (publisher anonymity), or that either a responder or an initiator be anonymous (responder or initiator anonymity), or that both responder and initiator be anonymous (mutual anonymity). In the most stringent version, achieving mutual anonymity requires that neither the initiator, nor the responder can identify each other, and no other peers can identify the two communicating parties with certainty. This is also the goal of our browser sharing system.

P2P systems can be classified into two classes: pure P2P systems, where peers share data without a centralized coordination; and hybrid P2P, where some operations are intentionally centralized, such as indexing of peers' files. Which form the system takes makes a difference. For instance, in a hybrid P2P, whether the indexing servers can be trusted or not has a critical implication on how anonymity is enforced.

In the next section (Section 7.4), we will overview the existing anonymity protocols, and present our motivation and objectives of the work.

Our goal is to achieve mutual anonymity between the initiator and responder with high efficiency. We consider two cases. In the first case, we assume the existence of trusted index servers (e.g., Napster[90], and browser-aware proxies [138]). In our work, instead of having both the initiator and responder each prepare their own covert path, we rely on the

index server to prepare a covert path for both of them, significantly reducing operations and communication overhead. We have proposed two new techniques: *center-directing*, where encryption cost is independent of the length of the covert path, and *label-switching* that eliminates potentially excessive messages in center-directing (Section 7.5).

In the second case, we assume a pure P2P setting. We propose an anonymity protocol called *shortcut-responding* that can greatly reduce communication overhead while preserving mutual anonymity (Section 7.6).

We analyze our proposed protocols in Section 7.7. We present our empirical experience of the techniques in a browser-sharing environment in Section 7.8. We discuss how to select the protocols based on their merits and limits from different aspects in Section 7.9. We conclude in Section 7.10.

7.4 Related Work on Anonymity Studies

The related work includes existing protocols for the three types of anonymity. We have paid special attention to the work on mutual anonymity, which has motivated us to develop new protocols.

7.4.1 Publisher and Sender Anonymity

Publisher Anonymity: In order to protect a publisher peer, many systems provide censorship resistance facility. In Freenet [29], each node in the response path may cache the reply locally, which can supply further requests and achieve publisher anonymity. Publius [126] splits the symmetric key used to encrypt and decrypt a document into n shares using Shamir secret sharing and store the n shares on various peers. Any k of the n peers must

be available to reproduce the key. Instead of splitting keys, FreeHaven[33] and [111] split a document into n shares and store them in multiple peers. Any k of the n peers must be available to reproduce the document. Tangler [125] and Dagster[120] make newly published documents depend on previously published documents. A group of files can be published together and named in a host-independent manner.

Initiator/responder Anonymity: Most existing anonymity techniques are for client/server models, which only hide the identities of the initiator (clients) from the responder (the server), but not vice versa. Anonymizer [55] and Lucent Personalized Web Assistant (LPWA) [54] act as an anonymizing proxy between a user and a server to generate an alias for a user, which does not reveal the true identity of the user. Many systems achieve sender anonymity by having messages go through a number of middle nodes to form a covert path. In Mix [25] and Onion [121], the sender part determines the covert path, and a message is encrypted in a layered manner starting from the last stop of the path. Instead of having the initiator select the path, Crowds [102] forms a covert path in such a way that the next node is randomly selected by its previous node. Hordes [115] applies a similar technique used in Crowd, but it uses multicast services to anonymously route the reply to the initiator. Freedom [51] and Tarzan [50] are similar to Onion Routing, but they are implemented at IP layer and transport layer rather than the application layer.

7.4.2 Existing mutual anonymity protocols: their merits and limits

Our study targets mutual anonymity between an initiator and a responder. There are two most related and recent papers aiming at achieving mutual anonymity: Peer-to-Peer Personal Privacy Protocol (P^5) [114], and Anonymous Peer-to-peer File Sharing (APFS)

[108].

Paper [114] first proposes to use a global broadcast channel to achieve mutual anonymity, where all participants in the anonymous communication send fixed length packets onto this channel at a fixed rate. Noise packets can be used to maintain a fixed communication rate. Besides enforcing both initiator and responder anonymity, this protocol pays a special attention to eliminate the possibility of determining the communication linkability between two specific peer nodes by providing equal and regular broadcast activities among the entire peer group. The broadcast nature of this framework can limit the size of the communication group. To address this limit, the authors further propose the P^5 scheme that creates a hierarchy of broadcast channels to make the system scalable. Different levels of the hierarchy provide different levels of anonymity at the cost of communication bandwidth and reliability. As authors stated in this paper, P^5 will not provide high bandwidth efficiency. But P^5 allows an individual peer to trade-off anonymity degree and communication efficiency.

In the APFS system, a coordinator node is set to organize P2P operations. Although this node is not considered as a highly centralized and trusted server, it should be in service all the time, and it plays an important role to coordinate peers for file sharing. APFS allows new peers to join and leave the system periodically by sending a message to the coordinator. Willing peers begin anonymously announcing themselves as servers to the coordinator. After contacting the coordinator, peers anonymously and periodically send lists of files using alias names to those servers. An initiator peer starts to request documents by anonymously querying the coordinator for available servers. The coordinator responds with a list of current servers. A peer then anonymously sends queries to some servers. Upon requests, these servers will send back N matches to the initiator peer. The initiator sends

the match request to a path, where the tail node is the last member. The tail node then forwards the request to the responder and returns the reply back to the initiator. APFS uses Onion as the base to build their protocol. There are two advantages of APFS. First, all the communications in the system are mutual anonymous. Even the coordinator does not know the physical identities of the peers. Second, the anonymous protocols are designed for a pure P2P where the trusted centralized servers may not be available.

However, there are also several disadvantages associated with APFS solely relying on volunteering. First, the suitability of a volunteering peer needs to be taken into account, which can significantly affect the performance of P2P systems. To do so, the coordinator needs to examine each volunteering peer before possibly assigning a task, such as peer indexing. The background checking of peers has to be done anonymously, increasing the communication overhead. Second, the number of servers can be dynamically changed. In the worst scenario, no qualified peers are available for a period of time, causing the P2P system to be in a weak condition. Thirdly, since any peer can be a server, a malicious node can easily become a server. Although the peer identities are hidden from a server, a server has the power to provide wrong indexing information to mislead the initiators. Finally, since no trusted servers are available, the anonymous communications have to be highly complicated.

Both P^5 and APFS provide unique solutions to achieve mutual anonymity in pure P2P systems without any trusted central controls. We believe that limited trusted and centralized services in decentralized distributed systems are desirable and necessary. In practice, trusted central parties exist and effectively function, such as proxies and firewalls in Internet and distributed systems. Utilizing these trusted parties and aiming at both

reliability and low-cost, we propose a group of mutual anonymity protocols. We show that with some limited central support, our protocols can accomplish the goals of anonymity, efficiency, and reliability. We have also proposed a mutual anonymity protocol solely relying on self-organizations among peers without any trusted central controls. In this protocol, the returning path can be shorter than the requesting path. Comparing with P^5 , this protocol does not need to broadcast the requested file back to the requester so that bandwidth is saved and efficiency is improved. Comparing with APFS, this protocol does not need special nodes to keep indices of sharing files, thus, eliminating the index maintenance overhead, and the potential problem of inconsistency between index records and peer file contents.

7.5 Anonymity with Trusted Third Parties

We present our techniques for achieving mutual anonymity of the initiator and responder with the help of trusted index servers that keep (but do not publicize) the whereabouts of the contents that are stored in the peers. Each peer sends an index of files they are willing to share with others peers to selected index servers periodically or when the percentage of updated files reaches to a certain threshold. We use I to represent the initiator, R to represent the responder, S to represent the index server that I contacts, and p_i ($i = 1, 2, \dots$) to represent a peer. For conciseness of the presentation, we assume there is only one index server. Section 7.5.4 discusses how multiple index servers will be involved in order to scale a P2P system.

A simple solution is to have an index server act as an anonymizing proxy hiding the identities of I and R from each other and other peers. But this index server may become

a bottleneck making the system not scalable. Instead, we have the index server randomly select several peers to act as middlemen. These middle nodes form a covert path for the peer that possesses the content to send the content to the peer that requests the content.

We describe one intuitive protocol using *mix*, and two new protocols, *center-directing* and *label-switching*, which are advanced alternatives. In the rest of the chapter, we use $X \rightarrow Y : M$ to represent X sending a message M to Y . We use K_X to denote the public key of X , and $\{M\}K$ to represent encrypting the message M with the key K .

7.5.1 A Mix-based Protocol: an intuitive solution

The detail of the mix-based protocol is shown below:

Step 1: The initiator sends a request to S . The request is encrypted with S 's public key.

$$I \rightarrow S : \{file_ID\}K_S$$

Step 2: S finds out that the file is possessed by R , it selects a list of peers p_0, p_1, \dots, p_k at random, and builds a *mix* with R as the first member of the path, I as the last member, and with p_i in the middle. We call this path *mix*. A *mix* is of the form $(p_0, (p_1 \dots (I, fakemix)K_{p_k} \dots)K_{p_0})K_R$. The item *fakemix* is introduced to confuse the last node in the *mix*, p_k , so that the format of a message passing through the middle nodes are the same. So p_k cannot be sure that she is the last stop. In addition, it generates a DES key K . It then sends a message to R . The message includes K encrypted with R 's public key, $\{file_ID\}$ encrypted with the DES key K , K encrypted with I 's public key, and the *mix*.

$$S \rightarrow R : \{K\}K_R, \{file_ID\}K, \{K\}K_I, mix$$

Step 3: R obtains K using its private key to decrypt $\{K\}K_R$; it uses K to decrypt the portion of the message $\{file_ID\}K$ and gets the file f based on the $file_ID$; it uses its private key to peel mix to obtain p_0 , and also the rest of the path, mix' , i.e. $(p_1 \dots (I, fakemix)K_{p_k} \dots)K_{p_0}$. It encrypts the file f with K and sends a message to p_0 :

$$R \rightarrow p_0 : \{f\}K, \{K\}K_I, mix'$$

Step 4: p_i decrypts mix' using its private key to obtain the address of the next member in the mix paths, and this also produces the rest of the path, mix'' . It then sends a message to p_{i+1} . For p_k , p_{k+1} is I .

$$p_i \rightarrow p_{i+1} : \{f\}K, \{K\}K_I, mix''$$

Step 5: I obtains K using its private key, and uses K to decrypt the encrypted file.

We omitted the details on how the initiator knows that the content is destined to it. This must be done efficiently. There are three alternatives: (i) to have S also encrypt $file_ID$ with the I 's public key and have R send this along with the content; (ii) to encrypt a magic number and the DES key with I 's public key; (iii) to encrypt $file_ID$ in $fakemix$ using the I 's public key. In the remainder of the chapter, we assume that our protocols choose one of the above alternatives.

The anonymizing path is selected by the trusted index server, and the mix routers are selected among the peers. Having the index server perform a path selection, this scheme becomes less vulnerable to traffic analysis since the peers' public keys need only be exposed

to the index server. Otherwise, an eavesdropper who knows the peers' public keys may reconstruct the path by applying the public keys in a reverse order. Furthermore, the index server has the opportunity to balance the load of the peers that act as mix routers. In this protocol, only the path is encrypted with an expensive public key encryption, and the content is encrypted with a less expensive DES key. This arrangement makes the scheme efficient. This scheme can be made more efficient by encrypting the mix path using secret keys that are shared between the index server and each of the peers. The content is encrypted by a key that is generated by the index server and is only known to I and R . This hides the content from anybody except I and R .

To well defend against traffic analysis, S can have the responder pad the contents, and the middle nodes can encrypt the DES-encrypted message pair-wise so that a message appears different along the path. These enhancements can be done to all our protocols. Figure 7.2 shows an example with two middle nodes.

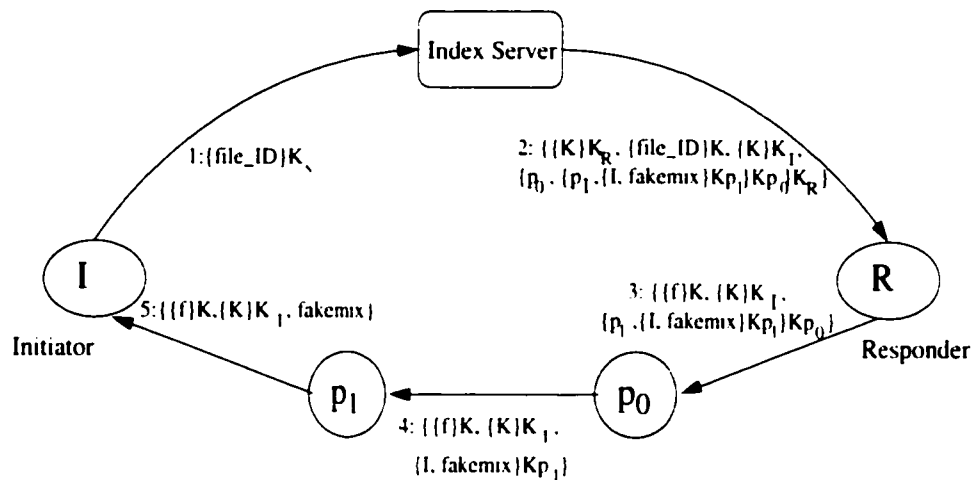


Figure 7.2: An example of the Mix-Based Protocol

7.5.2 Center-Directing

Alternatively, S can be used to reduce the number of encryption/decryption operations.

We describe two new protocols: *center-directing* and *label-switching*.

Instead of passing the *mix* through the whole covert path in *mix-based* protocol, the *center-directing* protocol has the index server send each node in the covert path its next hop individually. The basic idea of the center-directing protocol is as follows. The index server S selects several peers to form a covert path. It directs the content through the path by sending each middle node p_i a pair $\langle \text{label}(p_i), p_{i+1} \rangle$ that is encrypted with p_i 's public key. The labels can be generated such that $\text{label}(p_{i+1}) = \{\text{label}(p_i)\}K_{p_{i+1}}$. The labels uniquely identify a message, and p_{i+1} is the next member in the covert path. When the peer p_i sees a message from a peer labeled l , it will change the label to $\{l\}K_{p_{i+1}}$ and forward the message to p_{i+1} . Each p_i keeps a hash table to synchronize between the message from the index server and the message from its previous hop. The p_{i+1} is a random generated node number. Using the random node's public key to encrypt the request label each time, we can defend against traffic analysis in the sense that (1) labels for the same request appear differently along the covert path, and (2) the random generated node has no correlation with the nodes in the covert path. This protocol takes advantage of the fact that encryption cost is much lower than decryption cost in public key encryption. In contrast to the mix-based scheme, this protocol uses messages to set up the path. Although this incurs additional cost in hashing, setting up the path can be done in parallel. The big difference lies in the size of items being encrypted and decrypted. The server needs to encrypt $k \langle \text{label}, p_i \rangle$ pairs. Each peer decrypts once to reveal the next hop, and encrypts once to produce a label

for the next hop. Therefore, the sizes of items that need to be encrypted by public key encryption are independent of the path length.

The details of the protocol are shown below:

Step 1: The initiator I sends a request to S .

$$I \rightarrow S : \{file_ID\}K_S$$

Step 2: S first generates k that is the number of middle nodes in the covert path. S then generates a unique label for the request, n , and the first middle node in a covert path, p_0 . S also generates a DES key K . In addition, it randomly generates another node number used to convert the request label in node R , p_{j_0} . S then sends the following message to R :

$$S \rightarrow R : \{K\}K_R, \{n, file_ID, p_0, p_{j_0}\}K, \{K\}K_I$$

Step 3: S generates the next stop of p_0 , p_1 , and another random node number p_{j_1} . It converts the request label n to $\{n\}K_{p_{j_0}}$. S then sends a message to node p_0 :

$$S \rightarrow p_0 : \{n\}K_{p_{j_0}}, \{p_1, p_{j_1}\}K_{p_0}$$

Step 4: R obtains K using its private key to decrypt $\{K\}K_R$: it uses K to decrypt the portion of the message $\{file_ID\}K$ and gets the file f based on the $file_ID$: it converts the request label n to $\{n\}K_{p_{j_0}}$. It encrypts the file f with K and sends a message to p_0 :

$$R \rightarrow p_0 : \{n\}K_{p_{j_0}}, \{f\}K, \{K\}K_I$$

Step 5: S generates the next stop of p_1 , p_{i+1} , and another random node number $p_{j_{i+1}}$. It converts the request label $\{\dots \{n\}K_{p_{j_0}} \dots\}K_{p_{j_{i-1}}}$ to $\{\dots \{n\}K_{p_{j_0}} \dots\}K_{p_{j_i}}$. For p_k , p_{k+1}

is I . S then sends a message to node p_i :

$$S \rightarrow p_i : \{ \dots \{n\} K_{p_{j_0}} \dots \} K_{p_{j_i}} \cdot \{p_{i+1} \cdot p_{j_{i+1}}\} K_{p_i}$$

Step 6: p_i first matches the request label coming from the index server and the request label coming from last stop, $\{ \dots \{n\} K_{p_{j_0}} \dots \} K_{p_{j_i}}$, so that it finds the next stop for the request, p_{i+1} . It then converts the request label $\{ \dots \{n\} K_{p_{j_0}} \dots \} K_{p_{j_i}}$ to $\{ \dots \{n\} K_{p_{j_0}} \dots \} K_{p_{j_{i+1}}}$, and sends a message to p_{i+1} . For p_k , p_{k+1} is I .

$$R \rightarrow p_0 : \{ \dots \{n\} K_{p_{j_0}} \dots \} K_{p_{j_{i+1}}} \cdot \{f\} K \cdot \{K\} K_I$$

Step 7: I obtains K using its private key, and uses K to decrypt the encrypted file.

Figure 7.3 illustrates this protocol with two middle nodes. Each middle node uses an encryption operation to compute the label for setting up the path instead of using a decryption operation.

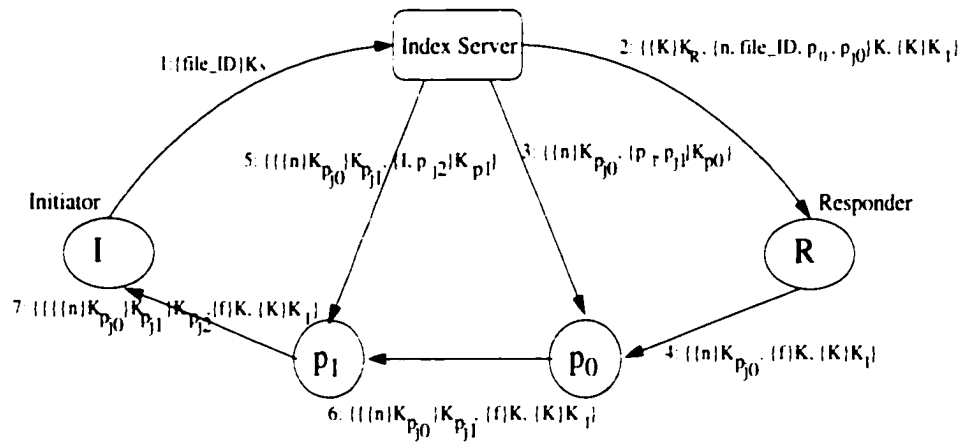


Figure 7.3: An example of the Center-Directing Protocol

7.5.3 Label-Switching

The label-switching protocol further reduces the messaging overhead of center-directing by putting more states on the peers. Rather than sending the middle nodes labels and next hop addresses on-the-fly, the index server produces a path table beforehand. The table is produced such that each peer p_i , as a destination, is associated with several path options. The path is of the form $p_x - p_y - \dots - p_i$ (L). This table is broken into sub-tables and distributed to peers (encrypted with their public keys). The sub-table of p_j consists of a list of pairs of the form $(L, nexthop)$. For every appearance of p_j in the path table, $\dots - p_j - p_w - \dots$ (L), the pair (L, p_w) is added to p_j 's sub-table.

Table 7.1 shows an example path table with 4 options for each peer. Table 7.2 shows some sub-tables derived from Table 7.1. In this example, each path option has two middle nodes. The number of middle nodes is not fixed in our design. It has already been shown that variable path-length strategies perform better than fix-length strategies[64]. Assuming that the index server needs to prepare a path from node 5 to 0, it can select among 4 paths from entry for node 0: $2-3-0(L8)$, $4-6-0(L3)$, $3-4-0(L4)$, and $1-7-0(L1)$. Suppose $L4$ is picked. The message will route to node 3, 4 and finally to 0, with each peer using their own sub-tables.

Table 7.1: Path Table

Peers	path			
0	2-3-0(L8)	4-6-0-(L3)	3-4-0(L4)	1-7-0(L1)
1
2

The detail of the protocol is shown below:

Peer1		Peer2		Peer3		Peer4	
L1	7	L8	3	L8	0	L3	6
...	L4	4	L4	0
...

Step 1: The initiator I sends a request to S .

$$I \rightarrow S : \{file_ID\}K_S$$

Step 2: S randomly selects a path in the entry for I in the path table (say $p_0 - p_1 \dots p_k - I$), and a key K . Assuming that this path has a label l . It sends the following message to R :

$$S \rightarrow R : \{l, p_0\}K, \{K\}K_R, \{K\}K_I$$

Here p_0 is the first middle node in the path.

Step 3: R sends a message (the label) to p_0 :

$$R \rightarrow p_0 : l$$

A persistent connection will be established between R and p_0 if the connection does not already exist. This connection is bound to the label l . Each p_i sends a message to p_{i+1} that is obtained from the sub-table of p_i .

$$p_i \rightarrow p_{i+1} : l$$

A persistent connection is set between p_i and p_{i+1} .

Step 4: A message is sent along the persistent connections from R to I .

$$R - l \rightarrow I : \{f\}K, \{K\}K_I$$

We use $-l \rightarrow$ to represent the persistent connection identified by the label l .

Step 5: I obtains K using its private key, and uses K to decrypt the encrypted file.

This protocol does not need the synchronization associated with center-directing protocol; it does not need as much encryption/decryption operations compared with the mix-based protocol: the only encryption and decryption occurs during the sub-table distribution. The overhead comes from the spaces for storing the path table and sub-tables and the time spending on table look-ups. Even though the path table kept in the trusted index may be a target of attack, multiple paths for a given source-destination pair adds one additional level of defense.

To simplify our presentation, we have assumed that we use a single label for the entire path. This protocol can be improved for stronger anonymity by introducing a pair of labels (like the center-directing protocol) for each hop rather than using a single label for the entire path, so that a label appears differently along the covert path.

7.5.4 Multiple Trusted Index Servers

In order to scale a P2P system to a large size, we will use multiple trusted index servers. Since multiple proxy servers are always available in an Internet region, this arrangement can be easily set up in practice. Besides scalability, the arrangement of multiple index servers will improve the reliability of a P2P system. As a peer node joins a P2P system, it will

register itself in multiple index servers. Servers may be down sometimes but unlikely at the same time. Thus, the indexing service is fault tolerant and much more reliable than the system with a single index server. However, use of multiple index servers also raises a load balancing issue. Without proper scheduling and redirection of peer requests, the workloads among the index servers can be unbalanced, generating some hot spot servers and leaving some others idle or lightly loaded.

We will adapt our own load sharing schemes [133] to make resource allocations in the P2P system. Each index server node maintains a current load index of its own and/or a global load index file that contains load status information of other index server nodes. The load status can be the number of registered peers, the average number of handled requests, storage for index of files to be shared, and so on. There are two alternatives to balance the workloads among the indexing servers **when a peer wants to join the system**.

- *index-server-based selections.* When a peer node joins the system and asks for an indexing service, it first randomly selects an index server. The load sharing system may periodically collect and distribute the load information among all index server nodes. Based on the load information of all index server nodes, the selected server will then suggest a list of lightly loaded index servers, including or excluding itself, for the peer node to be registered. One advantage of this approach is reliability. When a peer node leaves the system, it will inform one of the index nodes. This node will carry this message when it broadcasts its load status to other index server nodes. Since all index servers are trusted, a selection of most lightly servers is guaranteed. One disadvantage of this approach is that the global load statuses have to be updated

frequently among all the index servers to keep each node informed.

- *peer-node-based selections.* When a peer node joins the system and asks for an indexing service, it first broadcasts its request to all the index servers. Each index server will then return its status back to the peer node. The peer node will select a list of index servers to be the hosts, which are hopefully the most lightly loaded. When a peer node leaves the system, it will broadcast this status change message to all the index server nodes. In contrast to the alternative of index-server-based selections, this alternative does not require updating the load statuses globally among the index servers because a peer node will collect them each time it needs them. However, reliability is not guaranteed because peer nodes are not trusted, and they may not follow the load balancing principle when they select index server nodes.

There are also two alternatives **when a peer node requests a file**. The first alternative is straight forward. The peer node simply sends the request to index servers one by one. When it reaches the index server that has the index of the requested file, the file will be anonymously delivered to the peer node from a path arranged by the index server. The second approach involves two steps. The peer node first broadcasts a query message to all the index servers. The index servers that have the indices of the requested file will inform the peer node about their service availability. The peer node will then send the request to the index server that has responded earliest, for an anonymous file delivery. If the index server does not deliver the file for some reason, the peer node will try to send the request to other index servers that responded later than the first one. Although broadcast is not involved in the first alternative, the search is not as objective as the second alternative. In

general, we have no strong reasons favoring one approach over another.

7.6 Anonymity in Pure P2P

We now describe a technique to achieve mutual anonymity in a pure P2P setting without any trusted third party. We call it *shortcut-responding protocol*. In this protocol, a peer along the requesting path can elect itself to receive document on behalf of the initiator, thereby shortening the returning path.

We describe the details below:

Step 1: The initiator I randomly selects a list of peers, r_0, r_1, \dots, r_{k_T} , and builds a one-time *replyblock* with I as the last member of the path, and with r_i in the middle. The remainder *replyblock* is of the form $(r_{k_T} \cdot (r_{k_T-1} \dots (r_0, (I.fakemix)K_{r_0})K_{r_1} \dots)K_{r_{k_T}})$. Then I randomly selects a peer, p_0 , sends it the message:

$$I \rightarrow p_0 : \{r, \text{replyblock}, K_I\}$$

where r encodes the request.

Step 2: A peer p_i can elect itself to act as a *relay* of the returning path with a probability pv . We call pv the *shortcut probability*. If p_i has not elected itself, the request remains as $\{r, \text{replyblock}, K_I\}$. If p_i has self-elected, the *replyblock* and the request will be left in this node and the request format is changed to $\{r, \text{relay} : p_i, K_I\}$. It then decides whether to select p_{i+1} or broadcast the request with probability pb . If p_i has decided to broadcast the message, it will mark the message to avoid broadcasting it multiple times. Therefore, for p_i , the requests it can receive is one of the two formats: *format1* : $\{r, \text{replyblock}, K_I\}$ or *format2* : $\{r, \text{relay} : p_i, K_I\}$.

Step 3: If p_i can not find the content in local storage, it will save the request. We call p_i as R if p_i finds the content in local storage. R encrypts the found file content using K_I .

If R has the request format of *format1*, R contacts the first node in the *replyblock*, $r_{k\tau}$, then sends the encrypted file through the *replyblock* to I .

$$R \xrightarrow{\text{replyblock}} I : r, \{f\}K_I$$

If R has the request with the *format2*, it selects a list of peers o_0, o_1, \dots, o_{k_o} at random, and builds an *Onion* with o_0 as the first member of the path, *relay* as the last member, and with o_i in the middle. The *Onion* is of the form $(o_0, (o_1 \dots (o_{k_o}, (\text{relay}, \text{fakemix})K_{o_{k_o}})K_{o_{k_o-1}} \dots)K_{o_0})$. The R first sends the encrypted file through the *Onion* to the *relay*. If the request has not been discarded in the *relay*, the *relay* then sends the encrypted file through the *replyblock* to I . It discards the request so that duplicated responses can be dropped.

$$R \xrightarrow{\text{Onion}} \text{relay} \xrightarrow{\text{replyblock}} I : r, \{f\}K_I$$

Step 4: I uses her private key to decrypt the encrypted file.

Figure 7.4 illustrates the protocol with an example. Peer p_3 elects itself as a *relay* to receive the content on behalf of I . The peer that possesses the content R sends the response to p_3 through the *Onion*. The peer p_3 further sends the response to I through the *replyblock*.

This protocol has several advantages: (1) The response path can be shorter than the requesting path because a peer who receives the request and has the content will send the

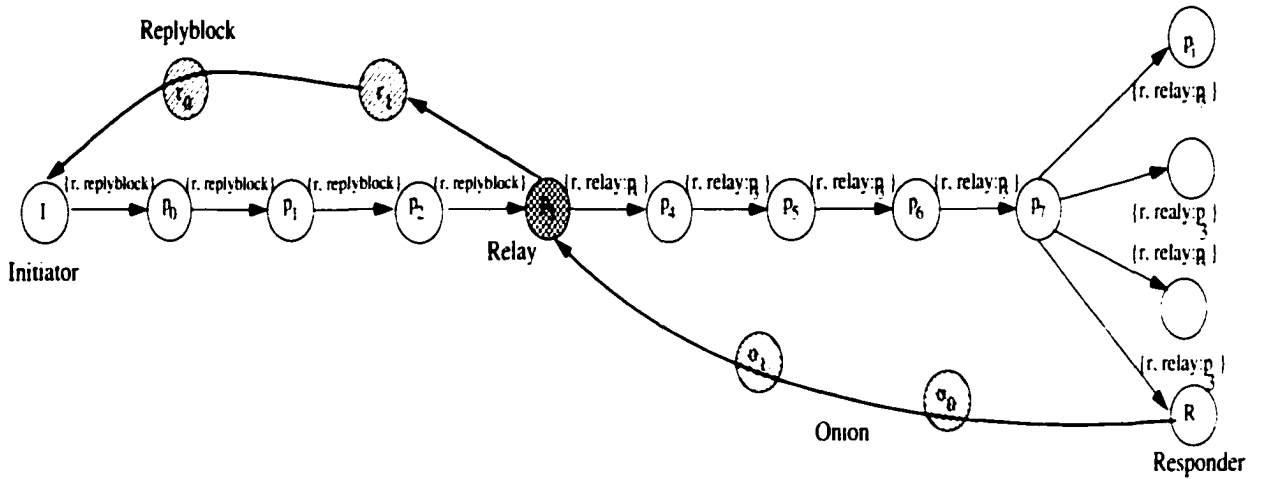


Figure 7.4: An example of the shortcut-responding Protocol

content through an *Onion* and a *replyblock* instead of going through the requesting path to the initiator. (2) Duplicated responses can be discarded earlier. (3) The protocol does not need special nodes to keep indices of sharing files like APFS, thus eliminating the index maintenance overhead and the potential problem of inconsistency between index records and peer file contents. (4) The protocol does not need to broadcast the requested file like P^5 while it still keeps mutual anonymity, so the efficiency is improved compared with P^5 . (5) The protocol uses *replyblock* that is also used in FreeHaven [33], where the responder contacts directly to the *replyblock* so that the first stop in the *replyblock* knows who the responder is. In contrast, *shortcut-responding* protocol has the responder send the requested file to a *relay* through an *Onion*, and then has the *relay* send the file back to the initiator through the *replyblock* so that nobody in the requesting path and responding path can guess the identity of initiator and responder with certainty. The initiator and responder also can not guess each other with certainty. Here is a related question to ask. If a node with a request of *format1* finds the requested file, it then contacts the *replyblock* and sends

back the file. In this case, can the first stop in the *replyblock* guess the one who contacts her is the responder? The answer is no, because the first stop in the *replyblock* can not distinguish whether the one who contacts her is the responder or a *relay*. Here is another proposed alternative. Upon receiving a request with *format1*, if a peer node realizes that the requested file is locally allocated, she will not send the file through the *relay* because the first stop in the *relay* can guess that the one who has just been connected is the responder. Instead of immediately providing the file, this peer forwards the request again. But this particular request is marked by her so that she will accept a later broadcast request. As soon as she receives this request again from a broadcast, she sends the file back through the *Onoin* and *replyblock*.

7.7 Analysis

We analyze the degree of anonymity each protocol can achieve, and compare their costs in terms of numbers of encryption/decryption operations.

7.7.1 Security Analysis

We analyze how the different protocols can defend against attacks from the various parties in the P2P networks. Because the situations for the initiator and the responder are symmetric, we consider only how different parties can guess the identity of the initiator.

The responder: To the responder, all other peers have the same likelihood of being the initiator. The probability that the responder correctly guess the identity of the initiator is $\frac{1}{n-1}$ (n is the total number of peers). Instead of making a random guess, the responder can bet that the peer to whom she sends the message is the initiator. She is only able to

make the right bet if there is no middle node selected. We assume that probability that there are k middlemen is $p(k)$, the probability that the responder makes the right bet is $p(0)$.

A middle node: We consider two cases: In the first case the middleman makes a random guess, because the only thing she can be sure about is that she is not the initiator. In this case, the probability it makes a correct guess is $\frac{1}{n-1}$. In the second case, the middleman bets that the peer to which it sends the message is the initiator. If there are k middlemen, only one of the k middlemen will make a correct bet. The probability that a middleman can make the correct bet is $\frac{1}{n-2} \sum_{k=1}^{n-2} \frac{p(k)}{k}$, and $p(k)$ is the probability that there are k middlemen.

In both cases, the probability will become smaller if multiple peers communicate simultaneously. For the protocols with the index server, even if a middle node can figure out who is communicating with whom, it still cannot figure out what is communicated.

A local eavesdropper: An eavesdropper is an entity that can monitor all local traffic. The worst case is when there is only one pair communicating (or the messages being communicated are so distinctive such that the eavesdropper is able to figure out who is communicating with whom). Even in this worst case, the eavesdropper still cannot figure out the content without the cooperation either from the responder or initiator (for the protocols with the index server) or one of the middlemen (for the shortcut-responding protocol).

Cooperating Peers: We consider cases where at least two middle nodes cooperate, and assume that neither the responder nor the initiator is involved. Two things make it hard for cooperating nodes to guess the identity of the initiator: (1) the middlemen do not know for sure how many communications are proceeding simultaneously, and (2) the format

of a message passing through the middle nodes is the same. If k collaborating peers were to make a random guess, the probability that they make the right guess is $\frac{1}{n-k}$, because all peers other than the k peers can be the initiator. If the collaborating peers were to make a bet, they can first eliminate all the peers that are communicating with peers that they know for sure is not the initiator. The worst scenario is when at least $m - 1$ out of all m middle nodes are involved. Even in this case, these middle nodes only have $\frac{1}{2}$ probability of correctly guessing that there is only one communication is conducted. The probability for them to correctly bet the identity of the initiator is $\frac{1}{2}$. Table 7.3 summarizes the results and compares them with P^5 and APFS.

Table 7.3: Degree of Anonymity

	Our Protocols		$P^5/APFS$	
	Guess	Bet	Guess	Bet
Initiator	$\frac{1}{n-1}$	$p(0)$	$\frac{1}{n-1}$	$p(0)$
Responder	$\frac{1}{n-1}$	$p(0)$	$\frac{1}{n-1}$	$p(0)$
Middle node	$\frac{1}{n-1}$	$\frac{1}{n-2} \sum_{k=1}^{n-2} \frac{p(k)}{k}$	$\frac{1}{n-1}$	$\frac{1}{n-2} \sum_{k=1}^{n-2} \frac{p(k)}{k}$
Cooperating Peers	$\frac{1}{n-k}$	$< \frac{1}{2}$, if they are not certain that there are multiple messages	$\frac{1}{n-k}$	$< \frac{1}{2}$, if they are not certain that there are multiple messages

For all protocols, we can add the following operations to increase the anonymity degree by introducing more confusion. The protocols prepare multiple covert paths for each request. The responder splits the requested file in multiple parts. The parts of the file can be sent back to the initiator through different covert paths. The different parts of the file can be easily combined, based on sequence numbers given by the responder. The Shamir algorithm can also be borrowed to split and combine files. With this algorithm, a file can be split into n parts and any k parts of them can reproduce the original file.

7.7.2 Cost of the Different Protocols

In Table 7.4, we summarize and compare the costs of the protocols in terms of numbers of encryption/decryption operations.

For the center-directing protocol, the time spent on RSA for setting up the anonymizing paths can be less than that of mix-based protocol for two reasons. First, RSA encryption is much faster than RSA decryption. Center-directing uses more encryption than decryption operations. Second, some steps are parallelizable. For the example in Figure 7.3, steps 3 and 4, and steps 5 and 6. The messages transferred in steps 3 and 5 are smaller than those in steps 4 and 6, so steps 3 and 5 may be finished before steps 4 and 6.

Table 7.4: Comparison of Protocols with k middle nodes in each covert path

Protocols		Mix-based	Center-directing	Label-switching	Shortcut
MD5		2	2	2	N/A
DES (Encrypt, Decrypt)	path	1, 1	1, 1	1, 1	0, 0
	content	1, 1	1, 1	1, 1	0, 0
RSA (Encrypt, Decrypt)	path	$4 + k, 4 + k$	$4 + 3k, 3 + k$	3, 3	$2k$
	content	0, 0	0, 0	0, 0	1, 1

7.8 Performance Evaluation

We estimate the additional overhead incurred in the protocols for achieving mutual communication anonymity. Our testbed is the browser-sharing environment where clients share cached Web contents [138]. The clients are the peers, and the proxy server is the index server. The proxy maintains an index of all files that are possibly cached in its clients' browser caches. If a user request misses both in the client's local cache and in the proxy

cache, the proxy will search the index file in an attempt to find the file in another client's browser cache. If the file is found in a client's cache, the proxy can then instruct this browser to forward the file to the requesting client. Our metric is the additional response time for each request hit in a remote browser cache compared with the response time of a request hit in the local browser cache. The increment comes from two major sources: time spent on transferring the requested data from the remote cache to the local cache, and time spent on the protocols.¹

We use trace-driven simulations and the Boeing traces [12] for the evaluation. We selected two days' traces (March 4 and March 5, 1999). There are 3996 and 3659 clients involved in these two days' traces, representing the total numbers of requests of 219,951 and 184,476, respectively. The total requested file sizes for the two traces are 7.54 and 7.00 Gbytes.

The results show that the average increment of the response time caused by the protocols is trivial. We present detailed performance results in the subsections that follow.

7.8.1 Data Transfer Time through Peer Nodes

We estimate the data transfer time through peer nodes based on a 100 Mbps Ethernet in our simulation. The bus contention is handled as follows. If multiple clients request bus service simultaneously, the bus will transfer documents one by one in FIFO order distinguished by each request's arrival time. Our experiments based on the ping facility show that the startup time of data communications among the clients in our local area network is less

¹We have neglected the costs for building and looking up the hash tables because the hashing cost is insignificant comparing with the other costs.

than 0.01 second. Setting 0.01 second as the network connection time, Table 7.5 presents the intra-network data transfer time for each trace. We can see that the amounts of data transfer times and the bus contention times spent for communications among clients on both traces are very low.

Traces	Total Workload Service Time	# Files Transferred among peers	Size of all files in Column 3	Data Transfer Time via 2 middlemen (% of Column 2)	Data Transfer Contention Time for Bus (% of Column 5)
Boeing.3/4	86,398.9 s	12,647	612 MB	177.82 s (.21%)	0.00003 s (.00002%)
Boeing.3/5	86,175.8 s	9,868	607 MB	149.64 s (.17%)	0.005 s (.0034%)

Table 7.5: Latency

7.8.2 Overhead of MD5, DES and RSA

The source programs of MD5, DES and RSA are obtained from [104] [106]. The machine we used for the experiments is a PC with a 1000 MHz Pentium III CPU and 128 Mbytes of memory. We used a large number of cached files in Microsoft's IE5 browsers as the input files for the tests. We ran each test 10 times. The average of 10 measurements is used.

The running times of MD5 and DES are proportional to the sizes of the input files. Our measurement results show that MD5 performs at 419 Mbps and DES's speed is 43.3 Mbps. The ratio of the RSA's running time to the input file size is not linear. RSA can encrypt/decrypt at a speed of 543/45.4 Kbps with a 512-bit value, 384/24.8 Kbps with a 768-bit value, and 275/14.6 Kbps with a 1024-bit value. It should be noted that the decryption speed of RSA is 12-19 times slower than the encryption speed. These measured results and the results in Tables 7.4 and 7.5 are used in our simulations to calculate the overheads of MD5, DES and RSA.

7.8.3 Additional Storage

The label-switching protocol requires additional storage to keep the path table in the index server and sub-tables in the peers. We allocate 2 bytes for each peer identification and 2 bytes for each path identification. The 2 bytes can represent up to 65536 different identifications. For each entry of destination described in Figure 7.1, 26 bytes are required in the index server. For the trace with 3996 peers, the total storage for the path table is 26×3996 , which equals to 101 Kbytes. There are a total of 3996×4 paths, and 4 bytes are needed for each entry of a path in a sub-table (see Figure 7.2). The storage needed for each peer is less than $3996 \times 4 \times 4$, which equals to 62 Kbytes. These storage requirements are sufficiently small for the path table and sub-tables to be held in memory for quick accesses.

7.8.4 Comparisons of Protocols

We have shown the data transfer times and the costs of MD5, DES and RSA operations. Here we compare the accumulated overheads of the protocols. Figure 7.5 compares the total increased response times and their breakdowns for the protocols using the “Boeing March 4 trace” and “Boeing March 5 trace” with 2 and 5 clients acting as middle nodes.

The performance results in Figure 7.5 show that center-directing and label-switching protocols generate very low overhead, while the other two have relatively higher overhead. The label-switching protocol shows its best performance. It is not desirable if the response time of a request hit in a remote browser cache is larger than that of the same request to the server. This is not a concern because our experiments show that the average response time increment is less than 8.4 ms when we use 5 middle nodes for both traces. The two protocols with lower overhead only increase the response time to about 3.4 ms when 5

middle nodes are used.

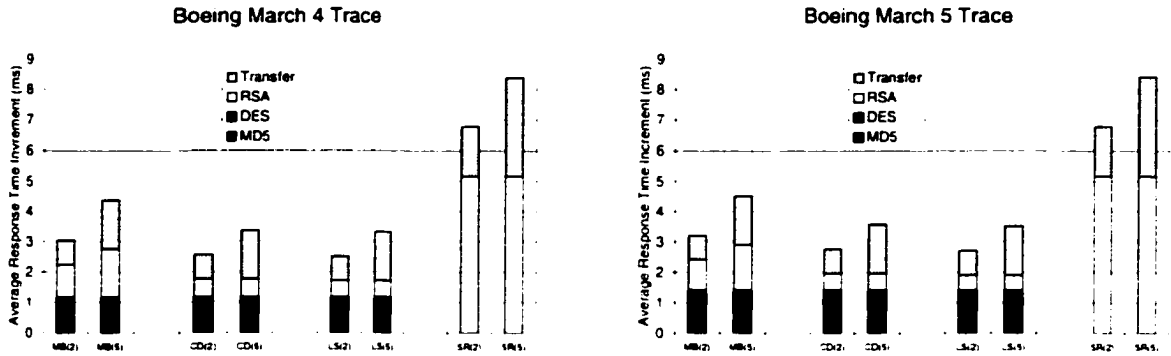


Figure 7.5: Breakdown of data transfer and protocol overhead with 2 and 5 middle nodes for Boeing March 4 trace (left) and Boeing March 5 Trace (right). $MB(k)$ represent mix-based protocol with k middle nodes. Similarly, CD, LS and SR represent center-directing, label-switching, shortcut-responding, respectively.

The time spent on RSA for the mix-based protocol increases as the number of middle nodes increases. In contrast, the times spent on DES and RSA for the center-directing and label-switching protocols are independent of the number of middle nodes. The number of RSA operations of center-directing protocol is the highest (see Table 7.4). However, most of them are low-cost encryption operations for small messages (such as a request, labels, node IDs), which are parallelizable. Both center-directing and label-switching protocols show very good scalability.

Compared with other three protocols, the time spent on RSA is considerably high for the shortcut-responding protocol. This is because we use a public key to encrypt the response content that is usually much larger than a message like a request, a label or a path. The efficiency can be improved if we encrypt the response content with DES keys that are encrypted using public keys in a pair-wise fashion. The traffic analysis can be defended.

but the content will be exposed to all middle nodes in a covert path.² The RSA cost is high, but it is a constant. So the shortcut-responding protocol scales well.

The data transfer time increases proportionally to the increase of the number of middle nodes. The transfer time of label-switching is lower than that of other protocols because it uses a persistent channel for continuous data transfers between the the same pairs of sending and receiving nodes. The data transfer time is still a dominant portion of the total overhead. We should limit the number of middle nodes to balance the two basic goals: achieving mutual anonymity and quick response time. Paper [64] shows that the anonymity degree may not always monotonically increase as the length of communication path increases.

7.9 Discussion

We have analyzed a mix-based scheme and several new protocols along with our empirical experience. We now discuss how to select the protocols based on their merits and limits.

How to select protocols by considering both efficiency and anonymity degree?

For a pure P2P system, the shortcut-responding protocol can be a good candidate, and its cost can be controlled by properly selecting the number of middle nodes in covert paths.

For a system with a trusted third party, such as a proxy and a firewall, this party can be utilized to provide some centralized support. With such limited support, both reliability and efficiency of mutual anonymity protocols can be significantly improved.

²In all of our protocols, the response content is only visible to the initiator and responder, but is not to any other nodes.

If storage space is not a concern, the label-switching protocol is the best choice in terms of efficiency. In fact, the storage requirement of this protocol can be acceptable for systems of moderate size (thousands of nodes). The other advantages of this protocol are: it uses minimal numbers of encryption/decryption operations; it does not need to keep all private keys in the third party, which can be vulnerable if the third party is attacked. Although the third party keeps a path table, there are multiple options for each destination. Therefore, even if the path table is exposed, it can still be very hard for an attacker to figure out which path is used for a specific data transfer.

If storage space is limited, the center-directing protocol is a good candidate. The mix-based protocol can be used if the RSA costs are tolerable.

Unlike the mix-based protocol, the cipher costs of center-directing and label-switching protocols are independent of the path length. In the case that a large number of middle nodes are required to enforce strong anonymity, center-directing and label-switching are the best choices.

What if a node in a covert path is down ?

All covert-path based protocols can have this problem. The center-directing protocol could handle this case very well. Since the trusted index server dynamically generates the next node in a covert path, it is easy for the index server to generate another node when it finds that the node it just generated is down.

APFS, shortcut-responding and mix-based protocols share the same concern for this problem. APFS and shortcut-responding protocol uses Onion as the base. A selected Onion passes through a whole covert path.

In the mix-based protocol, the trusted index server generates a *mix* that also needs to pass through the whole covert path. When a node in the covert path is down, the communication path needs to be recovered. One solution for this is to let the initiator send the request again when it can not get response within a certain period of time. Another covert path will be selected, in which all middle nodes are alive hopefully.

In the shortcut-responding protocol, if the *relay* can not get response within a certain period of time, it will send back a message of "NO RESPONSE". When the initiator receives a message of "NO RESPONSE", it means that the *Onion* part is down and the *replyblock* part works. If the initiator can not get anything within a certain period of time, she can not judge which path is down (maybe both are down). The request has to be sent again. Because the *replyblock* and *Onion* are one-time paths, hopefully all the selected nodes for the new request to form these paths are alive.

In APFS, for some initial requests, such as a request to volunteer to be a server, a request to ask for servers, or a request to update index, the requests will be resent if they can not get response within a certain period of time. For an initiator who already gets N matches for its request, there are also two covert paths between the initiator and responder. One is a path between the responder to its tail node, and another one is a path between the tail node and initiator. Because the communications are two-directional, the initiator can not judge which path is down if she can not get response, even with the help of the tail node. So the initiator has to send the same match request or another match request again. But the initiator will not need to start from the very beginning to request volunteer servers. APFS is more advanced than the shortcut-responding protocol in the sense that it will not sacrifice too much efficiency when a node in a covert path is down. But APFS still

can not compete with the center-directing protocol, because only one covert path needs to be handled in center-directing protocol.

The label-switching protocol generates a path-table in a trusted index server beforehand, and peers keep relevant portions of the path table as subtables. Although the path table and subtables are updated periodically for security reasons, the protocol has to trade-off efficiency if a middle node is down. One solution for this is to let the initiator send the request again with a note to the trusted index server that its first request for the same file not been responded to, when the initiator can not get response within a certain period of time. The index server will select a different covert path in the path table. Hopefully all middle nodes in this covert path are alive.

What if a file can not be found due to an obsolete index ?

All index-based protocols, such as mix-based, center-directing, label-switching, and APFS, can have this problem. The index servers keep an index of files that peers are willing to share. The indices are updated by the peers periodically. It is possible that the file has already been replaced in a peer, but the index still shows its existence.

When this happens in the mix-based, center-directing, label-switching protocols, the responder just informs the trusted index server that she can not find the file. The index server then will contact another peer who has the file or send back a message of "NO FILE FOUND" to the initiator. Another alternative is that the responder sends the message of "NO FILE FOUND" to the initiator through the covert path as usual. Then the initiator sends the request again to the index server with a note that the responder can not find the file.

In APFS, the responder reply a message of “NO FILE FOUND” to initiator. Since the initiator was responded N matches for her request, she will just try to get the file from another match if she can not get the file from the first match.

Comprehensively considering all factors, the *center-directing* protocol is the best. If efficiency has a high priority over reliability, the *label-switching* and *shortcut-responding* protocols work well for a system with a trusted third party, and a system without any central controls, respectively.

7.10 Chapter Conclusion

Providing a reliable and efficient anonymity protection among peers is highly desirable in order to build a scalable and secured P2P system. In this chapter, we have presented several protocols to achieve mutual anonymity in a P2P file-sharing environment. Our first group of protocols take advantage of the existence of trusted third parties to improve efficiency and reliability, and use them to prepare the covert paths for anonymous communications. The other proposed protocol, shortcut-responding, combines both broadcast and self-organizing covert path techniques to achieve mutual anonymity in pure P2P systems without any trusted central controls. After several hop-to-hop requests, this protocol broadcasts the request that is normally a small message. It then sends back the requested file back to the initiator through a dynamically created covert path instead of broadcasting, achieving both communication anonymity and efficiency.

The protocols utilizing trusted third parties may have three potential limits. First, these

trusted third parties may become single points of failure. This potential problem can be addressed by our proposed methods of multiple index servers. In addition, we can enforce anonymous communications between any peer to the trusted servers, hiding their identities and locations.

Second, one may have a concern about scalability of P2P system with the involvement of trusted parties. Specifically, we may not have enough trusted parties to handle the increasingly growing Internet user community. We believe this is not a necessary concern. The client/server model will continue to play its important roles and continue to co-exist with the P2P model. Thus, the number of trusted servers will proportionally increase as the number of peers increases.

Finally, a P2P system with the involvement of trusted parties may not be completely open and free, but may put some restrictions on peers. For example, a peer has the freedom to join and leave a pure P2P system any time. Although a peer still has this freedom in our system, she needs to do registration to a pre-defined index server(s). In fact, we view the involvement of the trusted parties for this respect positively. Researchers in the distributed system community have made a long-term effort to attempt to build trustworthy systems out of untrusted peers. We believe that this principle applies to P2P systems.

The performance and robustness of a P2P system to a great extent depend on the capacity of trusted servers, and the suitability of peers to act as middle nodes. A strong P2P system should be self-organizing, and adaptive to dynamic application demands and network condition changes. When a peer is used for some centralized function (e.g., index servers), some reputation system must be used to regulate their use. We attempt to follow these principles in designing our protocols.

Chapter 8

Prototype Implementations and Initial Results

In order to implement the additional communication and computing functions in each client, the security and integrity protocols between clients, and the data management schemes for browsers-aware caching, we have built a system infrastructure based on existing client and proxy servers. The infrastructure consists of two parts: a client daemon to interface its browser and to communicate with the proxy, and a browsers-aware proxy server. Coordinating the operations between the two sites, we are able to build a secured browsers-aware caching system. The system is still in its prototype stage. Songqing Chen was involved in part of the design and implementation.

8.1 A client daemon to interface the browser and communicate with the proxy

We have selected the mozilla (<http://www.mozilla.org>, or netscape) software as the working browser system since it is widely used in applications. Instead of revising the browser source

code, we have built a client daemon interfacing the browser and communicating with the proxy. This approach makes the commercial browser software still portable, and keeps its independent functions. The client daemon consists of a pair of parent-child processes at the user level. The child process serves as a receptionist that “listens” at a reserved port to incoming messages of requesting data files from the proxy. If such a message is received, the receptionist searches and fetches the file from the client browser and sends it back to the proxy or sends it to a target client. The parent process serves as browser file index manager. This manager periodically checks the status of file changes in the browser, and timely sends the index updates to the proxy. Three major data management functions are implemented to coordinate caching activities between browsers and the proxy.

- *make_a_browser_caching_decision*. This function decides whether the arriving document should be cached in the local disk. The decision is made based on a threshold value of the local requesting counter.
- *make_a_proxy_caching_decision*. This function decides whether a document requested by another client should be cached as a shared document in the proxy. The decision is made based on a threshold value of the global requesting counter. For this purpose, a port is reserved for a dedicated communication between a browser and the proxy. When a client sends back a document that proxy requests, it will use the same reserved port.
- *index_file_management*. This user function dynamically monitors the status of the local browser document index files. Whenever sufficient amount of local files are replaced (for example, a 10% change) and the network is not busy, it will send the

related items based on replaced files to the proxy for updating its browser index file.

With these three functions, the client daemon adds simple and sufficient functions to a client browser so that it is able to actively communicate with other clients directly or through the proxy. The client daemon is activated at the time when the system is booted. Figure 8.1 illustrates the organization of the client daemon and its interface with the netscape browser.

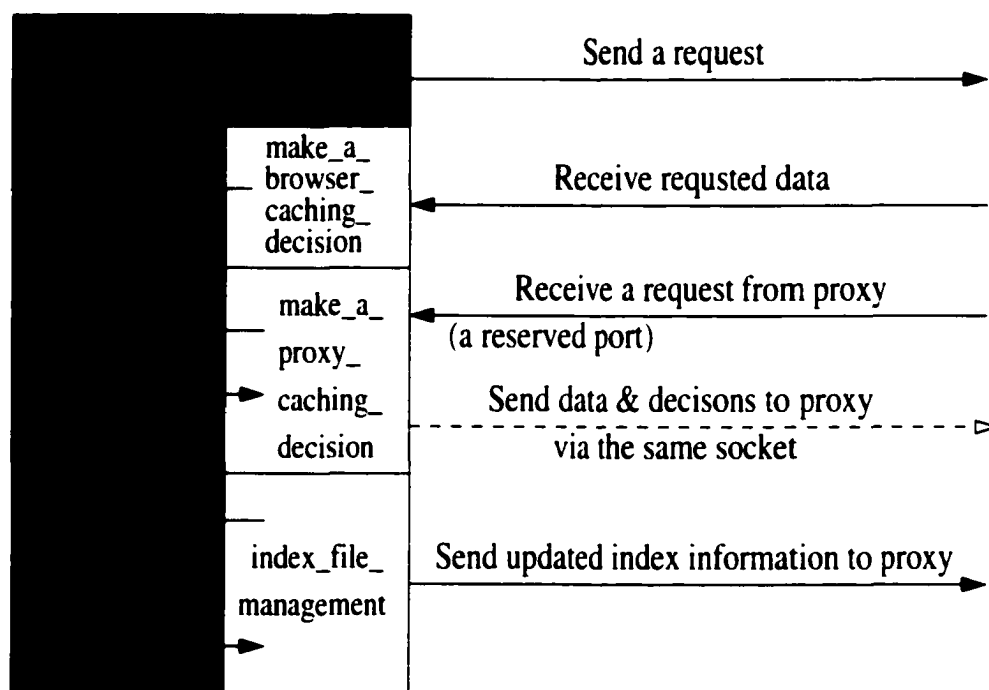


Figure 8.1: The organization of client daemon to interface with a client browser and the proxy.

8.2 A browsers-aware proxy server

We have selected the Squid proxy server (<http://www.squid-cache.org>) as the working system. Besides creating a global browser index file in the proxy, three additional functions

are added to the proxy server:

- *check_index_file*. This function checks the global browser index file after a miss occurs in the proxy. If the index file search is successful, it sends a data request to the target client.
- *cache_data_in_proxy*. This function caches the data after receiving a positive decision from a client.
- *global_index_file_management*. This user function maintains and updates the global browser index file upon receiving a new file from an upper level server or updated browser file status from a client. For this purpose, a port is reserved for a dedicated communication between a browser and the proxy.

Figure 8.2 illustrates the organization of the proxy daemon and its interface with the Squid proxy.

8.3 Overhead Measurement and Analysis

There are three major items of additional operations involved in browsers and proxy if an object can be provided by another browser instead of going to a Web server.

- *browser-index file searching*: The searching is done in the proxy after a request miss in the proxy. The browser index file consists of all the active URL's MD5 digests of browsers. We have used the searching facility for managing the cached documents in the Squid proxy. A hash function is used for the search, thus the search time is index

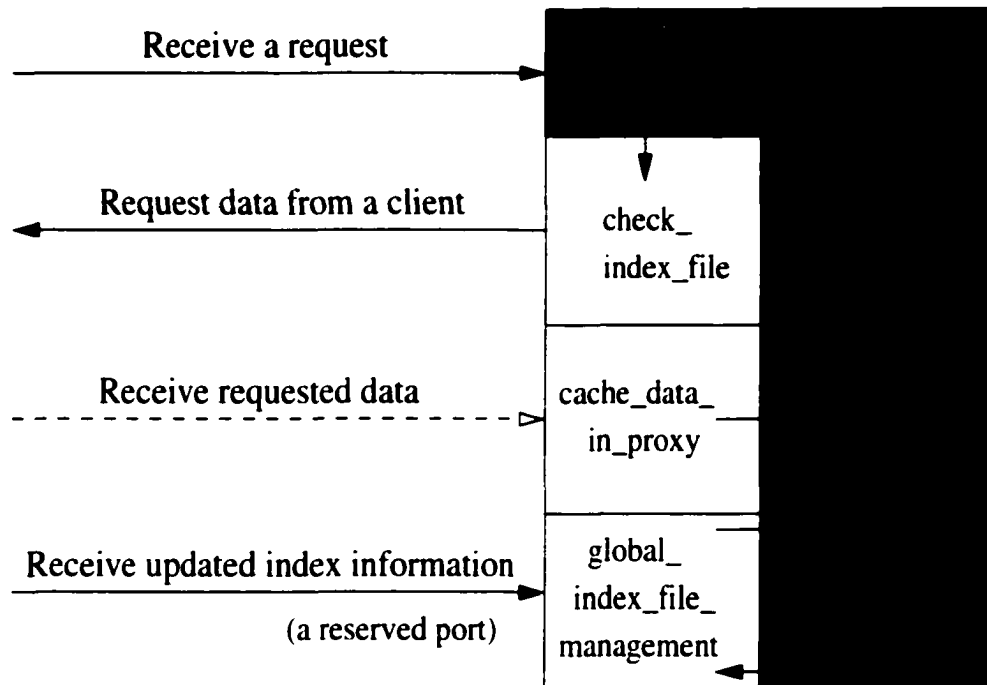


Figure 8.2: The organization of proxy daemon to interface with a client browser and the proxy.

file size independent. Specifically, function *storeGETPublic* is used, where function *hash.lookup* is called.

The searching time is denoted as T_{index} . Running the Squid proxy on a Pentium III of 1000 MHz machine, we obtained the average searching time. $T_{index} = 0.0076 \text{ ms}$.

- *requesting service from a client:* If the requested document is found in a browser cache after the index file searching, the proxy sends a request to the identified client. A requesting message is always 256 bytes. The communication time is denoted as T_{req} , and is dependent on a local area network speed.
- *data delivering between a client and the proxy:* The browser fetches the requested document and sends it back to the proxy that delivers it to the requesting client. The

data transferring time is denoted as T_{data} , and is dependent on the local area speed and size of the document.

The additional browsers-aware service time is $T_{overhead} = T_{index} + T_{req} + T_{data}$. We measured this service time by varying the size of the requested document from browsers on a 100 Mb Ethernet, and obtained $T_{overhead} = \alpha + \gamma D$, where $\alpha = 2.05$ ms, is the startup time including both T_{index} and T_{req} , and $\gamma = 1.10$ is the data transferring rate (ms/Kbytes), and D is the size in Kbytes of the document transferred between a browser and the proxy. Considering 8 Kbytes as the average size of a Web document, we obtain $T_{overhead} = 10.85$ ms from the model, which is very close to the measurement result.

There are also other types of unique operations in browsers-aware proxy. For example, the user daemon in each browser periodically sends the updated browser content information to proxy, and the proxy updates its index file accordingly. However, these operations are not in the critical path of the browsers-aware caching system, and can be done when the browser, proxy and networks are not in a heavy demand.

One important question we want to ask is how much latency time we can reduce with the support of the browsers-aware service. Without such a service, a proxy miss will consequently cause a request to a Web server and a data delivering from the server to the proxy. The average static HTML service time from a Web server is over 50 ms without considering the network congestion [150]. In contrast, our measurements show that the browsers-aware service can reduce this time to 10.85 ms, a reduction of more than 78%, if the document exists in one of the clients.

Chapter 9

Final Conclusions and Future Work

9.1 Summary

Effective resource management and its security issues have become crucial for the applications in distributed and Internet systems. Resource management covers a wide spectrum ranging from resource management in uniprocessor systems, to resource management on distributed and Internet systems. None of them can be ignored in order to significantly improve overall performance. An effective resource management must be adaptive to the changes of workload and technology. We have seen the rapid advancement of technology: the uniprocessor becomes increasingly fast, but the access speeds of memory and storage continue to lag behind. The high speed cluster and Internet technology have made the computing and information sharing widely decentralized and globalized. We have also observed several major changes of human demands. First, application workloads become increasingly data-intensive, relying on fast and efficient data accesses. Second, “computing” (including all the computer and Internet activities) has become an indispensable part of our daily life. Effective resource management directly improves the quality of life. Finally and most importantly, high performance is no longer the only resource management objective.

The objectives of security, availability, and reliability of the systems have become equally important!

We first study memory system utilization in centralized servers by improving memory performance of sorting algorithms. Memory hierarchy considerations during sorting algorithm design and implementation play an important role in significantly improving execution performance. Existing algorithms mainly attempt to reduce capacity misses on direct-mapped caches. To reduce other types of cache misses that occur in the more common set-associative caches and the TLB, we restructure the mergesort and quicksort algorithms further by integrating tiling, padding, and buffering techniques and by repartitioning the data set. Our study shows that substantial performance improvements can be obtained using our new methods.

After considering memory system utilization in centralized servers, we have further extended our study on load sharing for global memory utilization in distributed systems. The cluster system we consider for load sharing is a compute farm, which is a pool of networked server nodes providing high performance computing for CPU-intensive, memory-intensive, and I/O active jobs in a batch mode. Existing resource management systems mainly target balancing the usage of CPU loads among server nodes. Aiming at reducing the memory resource contention caused by page faults and I/O activities, we have developed and examined load sharing policies by considering effective usage of global memory in addition to CPU load balancing in both homogeneous and heterogeneous clusters. There are two major approaches to more effectively use global memory resources in a workstation cluster, aiming at minimizing page faults in each local workstation and improving overall performance of cluster computing: (1) job-migration-based load sharing schemes and (2)

network RAM. scalability. We also propose an improved load sharing scheme by combining job migrations with the network RAM for cluster computing. The improved scheme has the merits of both job migrations and network RAM. Our experiments show its effectiveness and scalability for cluster computing.

The Internet system is another branch of distributed systems. With the foundation of our load sharing study, we have further investigated memory and storage utilizations in Internet caching systems. Internet workload shows the trends of decreasing hit ratios in proxies and the diversity of the Web contents. Some limits of the existing caching system structure prevent them from effectively utilizing the rapidly improvement in Internet technologies and from adapting in a timely manner the changes of the supply and demand of Web contents. We propose a peer-to-peer Web document sharing technique, called *Browsers-Aware Proxy Server* that makes the browsers and their proxy share the contents to fully utilize the Web contents and network bandwidth among clients. Our study show that the amount of sharable data in browser caches is significant and can be utilized for peer-to-peer document sharing to improve Web caching performance and scalability.

In order to further improve the performance, a peer-to-peer Web caching management scheme, called *cooperatively shared proxy-browser caching* is proposed to reduce document duplications among a proxy and its client browsers. To evaluate this approach, we conduct trace-driven simulations with Web traces and compare the hit ratio, the byte hit ratio and the Web server access latency of the proposed Web caching management scheme with the values for the traditional approach and “Browsers-Aware Proxy Server”. As a result, we show that both the hit ratio and byte hit ratio of this scheme are indeed significantly higher, and the Web server access latency is reduced substantially. Finally, we empirically

show that the performance of our scheme compares very favorably with the performance of near-optimal offline Web caching algorithms.

New concerns are raised in the P2P browser sharing systems. Data integrity may not be trusted because a user is eligible to modify cached documents in the local browser cache. For security and privacy protections, our browsers-aware proxy system should hide the identities of both browser senders and receivers. General P2P systems also share the same concerns of data integrity and communication anonymity. We propose an integrity protocol to ensure data integrity in browser-aware systems. We also present several protocols to achieve mutual communication anonymity between an information requester and a provider in a P2P information-sharing environment such that neither the requester, nor the provider can identify each other, and no other peers can identify the two communicating parties with certainty. Our study shows that the average increase in response time caused by our protocols is trivial, and these protocols show both security and performance advantages over existing protocols in a P2P system.

We have built a system infrastructure based on existing client and proxy servers. The infrastructure consists of two parts: a client daemon to interface its browser and to communicate with the proxy, and a browsers-aware proxy server. Coordinating the operations between the two sites, we are able to build a secured browsers-aware caching system.

9.2 Future Work: Balancing the Power between Centralized and Decentralized Resources in Distributed Computing

We will discuss future work in the following three directions.

9.2.1 Non-uniform parallel computing

For a large scale application job demanding a huge memory space and I/O accesses, a single node server can not produce the results in a tolerant time period, or can not produce the correct results. If the CPU power in a single node is sufficient, memory/I/O is the fatal bottleneck. Here are two technical approaches to resolve this problem and to scale the application job for more computing power and space: (1) *single job using global memory/I/O in a cluster*, and (2) *parallelizing the job*.

Our studies presented in Chapter 3 belong to the first approach. Using migration-based load sharing schemes, we can try to migrate the job to a node with enough memory/I/O resources. Using network RAM, we can also utilize global memory/I/O resources from other nodes. Using our proposed scheme by combining job-migration and network RAM, we can further effectively utilize global memory/I/O resources. The single node service in this approach limits the scalability of computing power. In addition, the single node can easily become the hot spot slowing down the communication and computation.

The second approach is to let multiple nodes run this job, where the CPU and memory resources are evenly distributed and used. Local accesses or the data accesses between a CPU and its memory and disk are maximized. Another advantage is the nature of load balancing in parallel computing. However, these two advantages may not serve the best performance interests of parallel jobs because the balanced workload distribution among the nodes may result in a different unbalanced resource utilization in a cluster. Specifically, in a time-sharing environment, multiple parallel jobs may not have enough memory space, using local disk as the swapping site. But the CPU in each node may be under-utilized.

If we further increase the number of nodes for the parallel job, the memory space may be satisfied, but the CPUs among the nodes become even more under-utilized.

Having examined the limits of the two approaches, we propose a new approach called non-uniform parallel computing to better utilize the resources of both CPU and memory. Instead of evenly allocating parallel tasks among the nodes, we consider CPU and memory resource allocations separately. Under this non-uniform scheme, the number of CPUs to be assigned to a parallel job will be minimized in order to better utilize the increasingly powerful CPU cycles. Since a single job or a small number of jobs will be assigned in each node, the context switch overhead is also reduced. Regarding the memory resources, each job can not only utilize the local memory space from the assigned CPUs, but also remote memory space in other nodes. We do not limit the number of nodes for a job, but cautiously increase it so that the CPU of each node is fully utilized. In summary, the CPU cycles will be provided by a small number of nodes, while the global memory space of a cluster is open to the memory demand of the job. Since the speed gap between accessing a local memory and a remote memory is shrinking, and the speed gap between accessing a local disk and a remote memory continues to enlarge, the proposed scheme is expected to be highly performance beneficial.

9.2.2 Resource Indexing on Grid Systems

A Grid is a global cluster of clusters, which is a platform for large scale problems unsolvable by a single cluster. Resource management on Grids has several challenges because it needs to address more complex management issues.

Identifying and allocating available resources among the Grids is a challenge. Keeping

index of all nodes in a Grid system is not practical because updating index among different domain's clusters causes a high index maintenance overhead both in space and time. In addition, some Grids may not want to globally share the status information for security reasons. We propose solutions for this problem.

The first proposal is to create some special nodes in each Grid with the privilege of knowing resource availability of local cluster and its neighbor Grids. The advantage of this method is that the available resources can be quickly identified at an affordable expense. A future study will take the security into the consideration, and tradeoff the overhead and the performance.

The second proposal is related to allocating resources adaptively based on more dynamic changes of resource availability among different domain's Grids. The resources include CPU, memory, IO, network bandwidth, and others. In order to allocate resources among different domain's Grids, we need to predict available network bandwidths among different domains. But this prediction requires additional system effort. We will utilize the power of network bandwidth monitoring and measurement to collect the dynamic information for effectively allocating resources from different domains.

Reliability becomes more important for Grid computing and harder to handle than in a single cluster. Without enough reliability guarantee, the overall performance improvement is also hard to guaranteed. We plan to look into several reliability issues.

9.2.3 Resource Management on Peer-to-peer systems

P2P has recently attracted a lot of attention in the Internet community, and it represents a computing model that advocates decentralization. It also raises many new issues to be

addressed on resource management.

A pure P2P can be classified into Structured and Unstructured P2P [81]. Structured Pure P2P means that files are placed at some locations with specific rules, such as hash-table so that the queries can also follow corresponding rules for fast retrieval. Studying effective rules to place and retrieve a file is one topic for Structured P2P systems. An unstructured P2P means that nodes can join a peer group with its own files. The file location has no correlation with a node. Querying a file in an unstructured P2P system is quite different from that in a structured P2P system. A study for fast and scalable searching techniques is one topic for unstructured P2P systems.

In unstructured P2P systems, searching is not as effective as in structured systems because of the uncertainty of file locations. Another assistant technique for fast searching is to make certain replicas of files, such as caching a file in other nodes. Several related performance issues will be studied in unstructured systems.

Security is an important issue to be strongly addressed in P2P systems. We will continue our study in this direction based on our current work of communication anonymity. We are looking into combining different approaches to further and synergistically achieve the goal for both strong anonymity and high communication efficiency, as well as to adapt to application needs and network conditions.

Bibliography

- [1] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox. "Caching proxies: limitations and potentials". *Proceedings of the Fourth International World Wide Web Conference*, December, 1995.
- [2] A. Acharya and S. Setia. "Availability and utility of idle memory in workstation clusters". *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1999, pp. 35-46.
- [3] A. Agarwal and S. D. Pudar. "Column-associative caches: a technique for reducing the misses rate of direct-mapped cache". *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 21(2):179-190, May 1993.
- [4] S. Albers, S. Arora, and S. Khanna. "Page replacement for general caching problems". *Proceedings of 10th Annual ACM-SIAM Symposium Discrete Algorithms*, (SODA'99), 1999, pp.31-40.
- [5] A. Barak and A. Braverman. "Memory ushering in a scalable computing cluster". *Journal of Microprocessors and Microsystems*, Vol. 22, No. 3-4, August 1998, pp. 175-182.
- [6] P. Barford, A. Bestavros, A. Bradley, and M. Crovella. "Changes in Web client access patterns: characteristics and caching implications". *World Wide Web Journal*, 2(1):15-28, January, 1999.
- [7] A. Batat and D. G. Feitelson. "Gang scheduling with memory considerations". *Proceedings of 14th International Parallel & Distributed Processing Symposium (IPDPS'2000)*, May 2000, pp. 109-114.
- [8] L. A. Belady. "A study of replacement algorithms for virtual storage Computers". *IBM Systems Journal*, 5:78-101, 1966.
- [9] B. Bershad, D. Lee, T. Romer and B. Chen. "Avoiding conflict misses dynamically in large direct-mapped cache" *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October, 1994.
- [10] R. Bianchini and E. V. Carrera. "Analytical and experimental evaluation of cluster-based network servers". *World Wide Web Journal*, volume 3, number 4, December 2000.

- [11] J. Bilmes, K. Asanovic, C. W. Chin, and J. Demmel. "Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI coding methodology". *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997. pp.340-347.
- [12] Boeing log files. <ftp://researchsmp2.cc.vt.edu/pub/boeing/>
- [13] BU traces. <ftp://cs-ftp.bu.edu/techreports/1995-010-www.client-traces.tar.gz>
<ftp://cs-ftp.bu.edu/techreports/1999-011-usertrace-98.gz>
- [14] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. "Web caching and zipf-like distributions: evidence and implications". *Proceedings of IEEE INFOCOM*, 1999.
- [15] D. Burger and T. M. Austin. *The simplescalar tool set, version 2.0*. TR 1342, Department of Computer Sciences, University of Wisconsin, Madison, June 1997.
- [16] Canada's coast to coast broadband research network: <http://ardnoc41.canet2.net/>;
Sanitized log files: <http://ardnoc41.canet2.net/cache/squid/rawlogs/>
- [17] B. Calder, D.Grunwald, and J. Emer. "Predictive sequential associative cache". *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, February 1996.
- [18] B. Calder, C. Krintz, S. John, and T. Austin. "Cache-conscious data placement". *In 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October, 1998.
- [19] P. Cao and S. Irani. "Cost-aware WWW proxy caching algorithms". *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.
- [20] P. Cao, J. Zhang, and K. Beach. "Active cache: caching dynamic contents on the Web". *Proceedings of Middleware '98*, 1998, pp. 373-388.
- [21] S. Carr, K. S. McKinley, and C. W. Tseng. "Compiler optimizations for improving data locality". *In Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994. 28(5):252-262.
- [22] J. Challenger, P. Dantzig, and A. Iyengar. "A scalable and highly available system for serving dynamic data at frequently accessed Web sites". *Proceedings of SC'98*, November, 1998.
- [23] J. Chame and S. Moon. "A tile selection algorithm for data locality and cache interference". *Proceedings of International Conference of Supercomputing*, Rhodes Greece, June 1999, pp.492-499.
- [24] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra and M. Thottethodi. "Nonlinear array layouts for hierarchical memory systems". *Proceedings of International Conference of Supercomputing*, Rhodes Greece, June 1999, pp.444-453.
- [25] D. Chaum. "Untraceable electronic mail return addresses, and digital pseudonyms". *Communications of the ACM*, 24, 2, Feb.1981, pp.84-88.

- [26] H. Che, Z. Wang, and Y. Tung, "Analysis and design of hierarchical Web caching systems". *Proceedings of IEEE INFOCOM 2001*. April 2001.
- [27] S. Chen, L. Xiao, and X. Zhang, "Dynamic load sharing with unknown memory demands of jobs in clusters". *Proceedings of the 21st International Conference on Distributed Computing Systems*. (ICDCS'2001). Phoenix, Arizona, April 16-19, 2001, pp. 109-118.
- [28] S. Chen, L. Xiao, and X. Zhang, "Adaptive and virtual reconfigurations for effective dynamic resource allocations in cluster systems". *Proceedings of the 22nd International Conference on Distributed Computing Systems*. (ICDCS'2002). Vienna, Austria, July 2-5, 2002, pp.35-42.
- [29] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: a distributed anonymous information storage and retrieval system. Design Privacy Enhancing Technologies". *Workshop on Design Issues in Anonymity and Unobservability*, LNCS 2009, ed. by H. Federrath. Springer-Verlag (2001). pp.46-66.
- [30] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout". In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*. Lajolla, California, June 1995, pp.279-290.
- [31] P. Danzig, R. Hall, and M. Schwartz, "A case for caching file objects inside internetworks". *Proceedings of ACM Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 1993, pp.239-248.
- [32] C. Ding and K. Kennedy, "Improving cache performance of dynamic applications with computation and data layout transformations". *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999.
- [33] R. Dingledine, M. J. Freedman, and D. Molnar, "The Free Haven project: distributed anonymous storage service". *Workshop on Design Issues in Anonymity and Unobservability*, LNCS 2009, ed. by H. Federrath. Springer-Verlag (2001). pp.67-95.
- [34] J. J. Dongarra, J. D. Croz, I. S. Duff, and S. Hammarling, "A set of level 3 basic linear algebra subprograms". *ACM Transactions on Mathematical Software*, 16(1):1-17, Jan.1990.
- [35] F. Douglass, A. Haro, and M. Rabinovich, "HPP: HTML macro-preprocessing to support dynamic document caching". *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December, 1997, pp. 83-94.
- [36] F. Douglass and J. Ousterhout, "Transparent process migration: design alternatives and the sprite implementation". *Software - Practice and Experience*, Vol. 21, No. 8, 1991, pp. 757-785.
- [37] P. Druschel and A. Rowstron, "PAST: a large-scale, persistent P2P storage utility". *Proceedings of 8th workshop on Hot Topics in Operating Systems*, 2001.

- [38] X. Du and X. Zhang, "Coordinating parallel processes on networks of workstations". *Journal of Parallel and Distributed Computing*, Vol. 46, No. 2, 1997, pp. 125-135.
- [39] B. M. Duska, D. Marwood, and M. J. Feeley, "The measured access characteristics of world-wide-web client proxy caches". *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December, 1997.
- [40] S. G. Dykes and K. A. Robbins, "A viability analysis of cooperative proxy caching". *Proceedings of IEEE INFOCOM 2001*.
- [41] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "The limited performance benefits of migrating active processes for load sharing". *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1988, pp. 63-72.
- [42] J. H. Edmondson, P. I. Rubinfeld, P. J. Bannon, etc. "Internal organization of the Alpha 21164, a 300-Mhz 64-bit quad-issue CMOS RISC microprocessor". *Digital Technical Journal of Digital Equipment Corporation*, 7(1): 119-135, winter 1995.
- [43] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol". *Proceedings of 1998 SIGCOMM Conference*, pp. 254-265.
- [44] M. J. Feeley, et. al., "Implementing global memory management systems". *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995, pp. 201-212.
- [45] Dror Feitelson, The Parallel Workload Archive.
<http://www.cs.huji.ac.il/labs/parallel/workload/logs.html#lanlcm5>, 1998.
- [46] J. Ferrante, V. Sarkar, and W. thrash, "On estimating and enhancing cache effectiveness". In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, California, August 1991, pp.328-343.
- [47] R. A. Finkel, "Operating systems". *ACM Computing Surveys*, Vol.28, No.1, March 1996, pp.201-203.
- [48] M. D. Flouris and E. P. Markatos, "Network RAM". Chapter 16. *High Performance Cluster Computing*, Vol. 1, Edited by R. Buyya, Prentice Hall, New Jersey, 1999, pp. 383-508.
- [49] J. Fox, "Browser cache switch for internet explorer". *WebDeveloper Conference 2000*, San Francisco, California, September 2000).
- [50] M. J. Freedman, E. Sit, J. Cates, and R. Morris, "Introducing Tarzan, a peer-to-peer anonymizing network layer". *Proceedings of the 1st International Workshop on Peer-to-peer Systems*, March, 2002, MIT Faculty Club, Cambridge, MA, USA.
- [51] Freedom, <http://www.freedom.net/>
- [52] Freenet, <http://freenet.sourceforge.net/>, 2001

- [53] J. D. Frens and D. S. Wise. "Auto-blocking matrix-multiplication or tracking BLAS3 performance with source code". In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997. pp.206-216.
- [54] E. Gabber, P. Gibbons, D. Kristol, Y. Matias, and A. Mayer. "Consistent, yet anonymous. Web access with LPWA". *Communications of the ACM*. Vol. 42 No. 2. February 1999. pp.42-47.
- [55] E. Gabber, P. Gibbons, Y. Matias, and A. Mayer. "How to make personalized Web browsing simple, secure, and anonymous". *Proceedings of Conference on Financial Cryptography*. 1997.
- [56] S. Gadde, M. Rabinovich, J. Chase. "Reduce, reuse, recycle: an approach to building large internet caches". *Proceedings of the sixth Workshop on Hot Topics in Operating Systems*. May. 1997.
- [57] K. S. Gatlin and L. Carter. "Memory hierarchy considerations for fast transpose and bit-reversals". *Proceedings of 5th International Symposium on High-Performance Computer Architecture*. January 1999.
- [58] Pat Gelsingle. "Building the peer-to-peer community". *Intel Developer Forum Conference, Spring 2001*. Keynote Presentations. February 2001. <http://developer.intel.com/idf>
- [59] G. Glass and P. Cao. "Adaptive page replacement based on memory reference behavior". *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*. May 1997. pp. 115-126.
- [60] Gnutella. <http://gnutella.wego.com>. 2001.
- [61] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures in Pascal and C*. Addison-Wesley, 1991.
- [62] Li Gong. "JXTA: a network programming environment". *IEEE Internet Computing*. 5 3. May/June 2001. pp. 88-95.
- [63] S. D. Gribble, E. A. Brewer. "System design issues for Internet middleware services: deductions from a large client trace". *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*. December 1997.
- [64] Y. Guan, X. Fu, R. Bettati, and W. Zhao. "An optimal strategy for anonymous communication protocols". *Proceedings of the 22nd International Conference on Distributed Computing Systems*. (ICDCS'2002). July 2002.
- [65] F. G. Gustavson. "Recursion leads to automatic variable blocking for dense linear-algebra algorithms". *IBM Journal of Research and development*. 41(6):737-755. Nov.1997.

- [66] M. Harchol-Balter and A. B. Downey, "Exploiting process lifetime distributions for dynamic load balancing", *ACM Transactions on Computer Systems*, Vol. 15, No. 3, 1997, pp. 253-285.
- [67] J. Hennessy and D. Patterson, "Computer architecture: a quantitative approach", 2nd ed., Morgan Kaufmann Publishers, Inc., 1996.
- [68] C.-C. Hui and S. T. Chanson, "Improved strategies for dynamic load sharing", *IEEE Concurrency*, Vol. 7, No. 3, 1999, pp. 58-67.
- [69] Sandy Irani, "Page replacement with multi-size pages and applications to Web caching", *Proceedings of the twenty-ninth annual ACM Symposium on Theory of Computing 1997*, (STOC'97), pp.701-710.
- [70] S. Jin and A. Bestavros, "Popularity-aware greedy dual-size Web proxy caching algorithms", *Proceedings of 20th International Conference on Distributed Computing Systems*, (ICDCS'2000), April 2000.
- [71] T. Johnson, M. Merten, and W. Hwu, "Run-time spatial locality detection and optimization", *In 30th International Symposium on Microarchitecture*, December 1997.
- [72] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", *Proceedings of 17th Annual International Symposium on Computer Architecture*, May 1990, pp.364-373.
- [73] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill, "Inexpensive implementations of set-associativity", *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 17(3):131-139, 1989.
- [74] T. Kunz, "The influence of different workload descriptions on a heuristic load balancing scheme", *IEEE Transactions on Software Engineering*, Vol. 17, No. 7, 1991, pp. 725-730.
- [75] M. R. Korupolu and M. Dahlin, "Coordinated placement and replacement for large-scale distributed caches", *IEEE Transactions on Knowledge and Data Engineering*, 2001.
- [76] M. R. Korupolu, C. G. Plaxton and R. Rajaraman, "Placement algorithms for hierarchical cooperative caching", *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, (SODA'99), January 1999, pp.586-595.
- [77] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimization of blocked algorithms", *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, April 1991, pp.228-239.
- [78] A. LaMarca and R. E. Ladner, "The influence of caches on the performance of sorting", *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1997, pp.370-379.

- [79] W. Lin, S. K. Reinhardt, and D. Burger. "Reducing DRAM latencies with an integrated memory hierarchy design". *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, (HPCA-7) Nuevo Leon, Mexico, January 20-24 2001, pp.301-312.
- [80] T. S. Loon and V. Bharghavan. "Alleviating the latency and bandwidth problems in WWW browsing". *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, December 1997.
- [81] Qin Lv, Pei Cao, Edith Cohen, Kai Li, Scott Shenker. "Search and replication in unstructured peer-to-peer networks". *Proceedings of the 16th ACM International Conference on Supercomputing*, (ICS'02), New York, USA, June 2002.
- [82] A. Mahanti, C. Williamson and D. Eager, "Traffic analysis of a Web proxy caching hierarchy", *IEEE Network, Special Issue on Web Performance*, Vol.14, No.3, May/June 2000, pp.16-23.
- [83] R. Malpani, J. Lorch, D. Berger, "Making world wide web caching servers cooperate". *Proceedings of the 4th International World Wide Web Conference*, December, 1995.
- [84] E. P. Markatos and G. Dramitinos, "Implementation of a reliable remote memory pager". *Proceedings of the 1996 Usenix Technical Conference*, January, 1996, pp.177-190.
- [85] K. S. McKinley, S. Carr, and C. W. Tseng, "Improving data locality with loop transformations". *Transactions on Programming Languages and Systems*, 18(4), July 1996.
- [86] L. McVoy and C. Staelin, "Imbench: portable tools for performance analysis." *Proceedings of the 1996 Usenix Technical Conference*, San Diego, California, 1996, 279-295.
- [87] J. Mellor-Crummey, D. Whalley and K. Kennedy, "Improving memory hierarchy performance for irregular applications". *Proceedings of International Conference on Supercomputing*, June, 1999.
- [88] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
- [89] B. S. Michel, K. Nikoloudakis, P. Reiher, and L. Zhang, "URL forwarding and compression In adaptive Web caching". *Proceedings of IEEE INFOCOM 2000*, March, 2000.
- [90] Napster, <http://www.napster.com>.
- [91] National Lab of Applied Network Research. <http://www.ircache.net/>
Sanitized access logs: <ftp://ircache.nlanr.net/Traces/>
Statistics: <http://www.ircache.net/Cache/Statistics/>
- [92] K.-D. Neubert, "The Flashsort1 algorithm". *Dr. Dobbs's Journal*, February 1998, pp.123-125.

- [93] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. "Locality-aware request distribution in cluster-based network servers", *Proceedings of the Eighth Symposium on Architectural Support for Programming Languages and Operating Systems*, (ASPLOS-8), pp.205-216, October 1998.
- [94] A. Oram. *Peer-to-Peer Harnessing the Benefits of a Disruptive Technology*, O'Reilly, March 2001.
- [95] S. Park and L. Leemis. *Discrete-event simulation: a first course*, Lecture Notes, College of William & Mary, Revised Version, January 1999. Preprint of a Prentice-Hall book, August, 1999.
- [96] E. W. Parsons and K. C. Sevcik. "Coordinated allocation of memory and processors in multiprocessors". *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996, pp.57-67.
- [97] J. Peir, Y. Lee, and W. Hsu. "Capturing dynamic memory reference behavior with adaptive cache topology". In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
- [98] V. G. Peris, M. S. Squillante, and V. K. Naik. "Analysis of the impact of memory in distributed parallel processing systems". *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994, pp. 5-18.
- [99] K. Psounis and B. Prabhakar. "A randomized Web-cache replacement scheme". *Proceedings of IEEE INFOCOM 2001*.
- [100] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. "A scalable content-addressable network". ACM SIGCOMM 2001.
- [101] M. Reddy and G. P. Fletcher. "An adaptive mechanism for Web browser cache management". *IEEE Internet Computing*, 2(1), January 1998.
- [102] M. K. Reiter, and A. D. Rubin. "Crowds: anonymity for Web transactions". *ACM Transactions on Information and System Security*, 1.1, November 1998, pp. 66-92.
- [103] G. Rivera and C. W. Tseng. "Data transformations for eliminating conflict misses". In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.
- [104] R. Rivest. "The MD5 message-digest algorithm". *Internet RFC/STD/FYI/BCP Archives*, Request for Comments: 1321, April 1992. (<http://www.faqs.org/rfcs/rfc1321.html>).
- [105] A. Rousskov and V. Soloviev. "A performance study of the Squid proxy on HTTP/1.0". *World Wide Web*, 2(1-2):47-67, January 1999. "on Performance of Caching Proxies", *SIGMETRICS'98*, pp.272-273.
- [106] RSAREF20. <http://tirnanog.ls.fi.upm.es/Servicios/software/ap/crypt/disk3/rsaref20.zip>

- [107] Stefan Saroiu, P. Krishna Gunmadi, and Steven D. Gribble. "A measurement study of peer-to-peer file sharing systems". *Proceedings of Multimedia Computing and Networking 2002* (MMCN'02).
- [108] V. Scarlata, B. N. Levine and C. Shields. "Responder anonymity and anonymous peer-to-peer file sharing". *Proceedings of the 9th International Conference on Network Protocols* (ICNP 2001). November, 2001.
- [109] R. C. Seacord and S. A. Hissam. "Browsers for distributed systems: universal paradigm or siren's song". *World Wide Web Journal*. 1998, pp.181-191.
- [110] R. Sedgewick. "Implementing quicksort programs". *Communications of the ACM*. Vol. 21, No. 10, 1978, pp. 847-857.
- [111] A. Serjantov. "Anonymizing censorship resistant systems". *Proceedings of the 1st International Workshop on Peer-to-peer Systems*. March, 2002. MIT Faculty Club, Cambridge, MA, USA.
- [112] S. Setia. "The interaction between memory allocation and adaptive partitioning in message-passing multicomputers". *Proceedings of the IPPS Workshop on Job Scheduling Strategies for Parallel Processing*. 1995, pp. 146-164.
- [113] S. Setia, M. S. Squillante, and V. K. Naik. "The impact of job memory requirements on gang-scheduling performance". *Performance Evaluation Review*. March 1999.
- [114] R. Sherwood, B. Bhattacharjee, and A. Srinivasan. " P^5 : A protocol for scalable anonymous communication". *Proceedings of 2002 IEEE Symposium on Security and Privacy*. May 2002.
- [115] C. Shields and B. N. Levine. "A protocol for anonymous communication over the internet". *Proceedings of 7th ACM Conference on Computer and Communication Security* (ACM CCS 2000). November 2000, pp.33-42.
- [116] A. Silberschatz and P. B. Galvin. *Operating systems concepts*, 4th Edition, Addison-Wesley, 1994.
- [117] B. Smith, A. Acharya, T. Yang, and H. Zhu. "Exploiting result equivalence in caching dynamic Web content". *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*. October, 1999.
- [118] A. Srivastava and A. Eustace. "ATOM: a system for building customized program analysis tool." *Proceedings of ACM Symposium on Programming Languages Design and Implementation*. 1994, pp.196-205.
- [119] I. Stoica, R. Morris, D. Karger, M. F. Kasshoek, H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications". *ACM SIGCOMM* 2001.
- [120] A. B. Stubblefield and D. S. Wallach. "Dagster: censorship-resistant publishing without replication". Technical Report TR01-380, Department of Computer Science, Rice University, July 2001.

- [121] P. F. Syverson, D. M. Goldschlag, and M. G. Reed. "Anonymous connections and onion routing". *1997 IEEE Symposium on Security and Privacy (S&P'97)*. pp.44-53.
- [122] A. S. Tanenbaum and R. Van Renesse.. "Distributed operating systems". *ACM Computing Surveys*. Vol.17, No.4, December, 1985, pp.419-470.
- [123] R. Tewari, M. Dahlin, H. M. Vin and J. S. Kay, "Design considerations for distributed caching on the Internet", *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, (ICDCS), May 1999.
- [124] G. M. Voelker, H. A. Jamrozik, M. K. Vernon, H. M. Levy, and E. D. Lazowska. "Managing server load in global memory systems". *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1997, pp.127-138.
- [125] M. Waldman, D. Mazi, "Tangler: a censorship-resistant publishing system based on document entanglements". *Proceedings of the 8th ACM conference on Computer and Communications Security*, 2001, pp.126-135.
- [126] M. Waldman, A. D. Rubin, and L.F. Cranor, "Publius: A robust, tamper-evident, censorship-resistant web-publishing system". *Proceedings of the 9th USENIX Security Symposium*, August 2000, pp.59-72.
- [127] W. Wang and M. W. Mutka, "Intelligent browser initiated server pushing". *Proceedings of the IEEE International Performance, Computing and Communications Conference*, February 2000.
- [128] Web-caching site: <http://www.web-caching.com>.
- [129] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software." *Proceedings of Supercomputing'98*, November 1998.
- [130] M. E. Wolf, "Improving locality and parallelism in nested loops", PhD thesis, Dept. of Computer Science, Stanford University, August 1992.
- [131] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy, "Organization-based analysis of Web-object sharing and caching". *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, October, 1999.
- [132] Working Group on Peer-to-Peer Computing. <http://www.peer-to-peerwg.org>
- [133] L. Xiao, S. Chen, and X. Zhang "Dynamic cluster resource allocations for jobs with known and unknown memory demands" *IEEE Transactions on Parallel and Distributed Systems*, Vol.13, No.3, 2000, pp.223-240.
- [134] L. Xiao and X. Zhang, "Exploiting neglected data locality in browsers". *Proceedings of the 10th International World Wide Web Conference (WWW-10)*, Hong Kong, May 1-5, 2001 (an extended abstract).

- [135] L. Xiao, X. Zhang, and S. A. Kubricht. "Incorporating job migration and network RAM to share cluster memory resources". *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC-9)*. Pittsburgh, Pennsylvania, August 1-4, 2000, pp. 71-78.
- [136] L. Xiao, X. Zhang, and S. A. Kubricht. "Improving memory performance of sorting algorithms". *ACM Journal on Experimental Algorithmics*, Vol. 5, No. 3, 2000, pp. 1-23..
- [137] L. Xiao, X. Zhang, and Y. Qu. "Effective load sharing on heterogeneous networks of workstations" *Proceedings of 2000 International Parallel and Distributed Processing Symposium*, (IPDPS'2000), Cancun, Mexico, May 1-5, 2000.
- [138] L. Xiao, X. Zhang, and Z. Xu. "A reliable and scalable peer-to-peer Web document sharing system", *Proceedings of 2002 International Parallel and Distributed Processing Symposium*, (IPDPS'2002), Fort Lauderdale, Florida, April 15-19, 2002.
- [139] Z. Xu, L. Xiao, and X. Zhang. "Data integrity and communication anonymity in peer-to-peer networks". Hewlett Packard Laboratories, Technical Report HPL-2001-204, August 2001.
- [140] Y. Yan, X. Zhang, and Z. Zhang. "Cacheminer: a runtime approach to exploit cache locality on SMP". *IEEE Transactions on Parallel and Distributed Systems*, Vol.11, No.4, 2000, pp.357-374.
- [141] J. Yang, W. Wang, R. Muntz. "Collaborative web caching based on Proxy affinities". *Proceedings of ACM SIGMETRICS 2000*, Santa Clara, June, 2000, pp.78-89.
- [142] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, D. Culler. "Using smart clients to build scalable services". *Proceedings of the USENIX 1997 Annual Technical Conference*, January, 1997.
- [143] P. S. Yu and E. A. MacNair. "Performance study of a collaborative method for hierarchical caching in proxy servers". *Proceeding of the seventh International World Wide Web Conference*, April, 1998.
- [144] L. Zhang, S. Michel, K. Nguyen, A. Rosenstein, S. Floyd, and V. Jacobson. "Adaptive web caching: towards a new global caching architecture". *Proceedings of 3rd International WWW Caching Workshop*, Manchester, England, June, 1998.
- [145] X. Zhang, Y. Qu, and L. Xiao. "Improving distributed workload performance by sharing both CPU and memory resources". *Proceedings of 20th International Conference on Distributed Computing Systems*, (ICDCS'2000), Taipei, Taiwan, April 10-13, 2000.
- [146] Z. Zhang and X. Zhang. "Fast bit-reversals on uniprocessors and shared-memory multiprocessors". *SIAM Journal on Scientific Computing*, Vol.22, No.6, 2001.
- [147] C. Zhang, X. Zhang, and Y. Yan. "Two fast and high-associativity cache schemes". *IEEE Micro*, Vol. 17, No. 5, 1997, pp.40-49.

- [148] Z. Zhang, Z. Zhu, and X. Zhang, "Cached DRAM for ILP processor memory access latency reduction", *IEEE Micro*, Vol. 21, No. 4, July/August, 2001, pp.22-32.
- [149] S. Zhou, "A trace-driven simulation study of load balancing", *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, 1988, pp.1327-1341.
- [150] H. Zhu and T. Yang, "Class-based cache management for dynamic Web content", *Proceeding of IEEE INFOCOM 2001*.

VITA

Li Xiao

Li Xiao received her BS and MS degrees in Computer Science from the Northwestern Polytechnic University, China. She was enrolled the Ph.D. program of Computer Science at the College of William and Mary in the Fall semester of 1998. She has been a teaching assistant and a research assistant in the department since then. She was a research intern at the Hewlett Packard Labs in the summer of 2001. She is a recipient of USENIX Fellowship for Ph.D. dissertation research in her last year of graduate study. Her research interests are in the areas of distributed and Internet systems, system resource management, and designs and implementation of experimental algorithms. She is a member of the IEEE.