Techniques to Enhance Web Performance in Fixed Networks and Mobile Networks.

THESIS

Submitted in partial fulfillment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY

by

T S B SUDARSHAN

Under the Supervision of
Prof. G. Raghurama

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN) INDIA
2007

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN) INDIA

CERTIFICATE

This is to certify that the thesis entitled "**Techniques to Enhance Web Performance in Fixed Networks and Mobile Networks**" and submitted by **Mr. T S B Sudarshan** ID No. **1999PHXF009** for award of Ph.D degree of the Institute embodies original work done by him under my supervision.

____

Signature of the Supervisor

Date:

Name: Prof. G. RAGHURAMA

Designation: Deputy Director (Academic)

Dedicated to all my Teachers

ACKNOWLEDGEMENTS

I am most indebted to my supervisor, Professor G. Raghurama for his guidance, wisdom, valuable suggestions and encouragement throughout my work. Talking to him has always been enlightening and highly productive. Throughout the years, it has been a great learning experience working under him. His constructive criticism when required has greatly helped me in my career.

I would like to thank Prof Sundar Balasubramanian, Group Leader, CS & IS Group, Prof Rahul Banerjee, Unit Chief, Software Development & Education Technology Unit, for all the support and encouragement throughout. I also thank all my colleagues who directly and indirectly helped me in completing my thesis.

A special appreciation to Nokia Research Center, Boston for funding my research work. In particular, I would like to thank Dr. Sudhir Dixit for providing necessary assistance, encouragement and suggestions on several occasions. I would like to thank Prof J.P Mishra, Chief, Information Processing Centre, for his support and facilities provided to do this work.

Thanks are due to Prof. L.K. Maheshwari, Vice-Chancellor, BITS, for the interest shown in my work and constant encouragement. A word of thanks to Prof Ravi Prakash, Dean, Dr. S.S. Deshmukh, Mr. Dinesh Kumar and other staff members of Research & Consultancy Division who directly or indirectly assisted me during this period.

Beyond doubt, pursuing higher studies has been possible only due to the support and encouragement from my parents. Being teachers themselves, they have been source of inspiration for me to aim higher in teaching and research. Thank You Mom and Dad.

Lastly, but always first in my heart, I appreciate the support of my wife, Dr. Shikha Tripathi throughout. Thank you for your incredible support. My daughters Kriti and Prakhya have been wonderful, co-operative and less demanding.. Thanks kids!

Finally, I give thanks to Almighty GOD, for guiding me to this stage in my life.

ABSTRACT

The World Wide Web has become an integral part of every day life and rules the world's economy. It has changed the way people work, communicate, and share information. The growth of Internet, in terms of the number of users and the type of objects accessed, has been phenomenal and fast. The content accessed has undergone a change from plain HTML pages to more dynamic pages with multimedia content, while the user-end equipments have evolved from desktops to laptops and mobile devices. The fixed networks have transformed into a combination of fixed and mobile networks. This has led to slower speeds of web access and hence, lesser user satisfaction. Factors which contribute to the slower speeds of the web also include the heterogeneity of network connectivity, origin server location and distance from the users, traffic congestion, unexpected rise in demand and dynamic updating of information on the web servers. Reducing the latency in web access is critical for user satisfaction and productivity. Web caching and Prefetching are two methods being investigated recently by researchers, for improving the response times experienced by the user.

This thesis discusses some aspects of caching and prefetching techniques to enhance web performance in fixed networks and mobile networks. These techniques are adopted for static web objects, multimedia web objects and wireless Internet access. Web caching is similar to memory system caching; differences being the nonuniformity of Web object sizes, retrieval costs and cacheability. A Web cache stores Web resources in anticipation of future requests. The replacement policies designed for Web caching thus must be characteristically different from that of memory systems. Caching techniques for streaming multimedia objects and mobile networks are recent research issues. This thesis concentrates on four aspects of Web caching: QoS cache replacements, caching streaming multimedia objects, adaptive cache replacements for static and streaming objects and caching in mobile networks.

Web caching does not support quality of service (QoS) and therefore can be seen as a best-effort service. All objects are handled equally. Introducing QoS to caching and

replacement is an interesting research issue. Cache-on-Demand protocol is an effort in that direction. This thesis proposes and evaluates one such caching technique with replacement policies to enhance the web performance. Also, it proposes a dual-caching scheme for caching web objects, which enhances the web performance. The caching techniques popularly used for static web pages cannot be used for streaming multimedia objects. These objects cannot be cached in its entirety due to their large size. This thesis proposes caching and replacement techniques, for both the static and the adaptive kinds. It proposes a frequency index based method for replacing partially cached multimedia objects using Cut-off and Optimal caching methods.

Application of soft computing methods for caching and replacements in web caching is an interesting approach. This thesis proposes adaptive methods such as Fuzzy logic and Genetic algorithm for web caching and replacement. A Genetic algorithm is adopted for streaming multimedia web objects replacement and analyzed for its performance on different workloads.

Accessing the World Wide Web data by using mobile devices is increasing due to the deployment of 2.5G and 3G services. A mobile user's web access is largely determined by the user-specific preferences and the presentation of data is constrained by the capabilities of the device used. For reducing latency and disconnectivity while using Internet by a mobile client, prefetching and Quality of Service (QoS) can be deployed. A caching architecture for combining prefetching and QoS is proposed in the thesis. An attempt has been made to derive a dynamic cache invalidating scheme which uses 'Bit-Sequence' and 'Bit-Sequence with bit count' depending on the number of cache objects being updated.

All the proposed schemes are tested for performance through simulation studies using benchmark web access logs. The thesis also proposes and analyses various hardware designs and implementation for LRU replacement policy in high-associativity processor cache for caching uniform objects.

TABLE OF CONTENTS

LIST OF FIGURES ---------- vi
LIST OF TABLES ---------- xi
LIST OF ACRONYMS ---------- xii

1. INTRODUCTION
   1.1 Web Performance Enhancement Techniques: An Overview ------- 1
      1.1.1 Caching Technique ---------- 2
      1.1.2 Prefetching Technique ---------- 2
   1.2 Objectives and Approach ---------- 3
   1.3 Thesis Outline and Contributions ---------- 5

2 BACKGROUND & LITERATURE SURVEY
   2.1 Introduction ---------- 7
   2.2 Processor caching ---------- 8
   2.3 Web Caching vs. Processor Caching ---------- 10
   2.4 Web Caching Techniques ---------- 12
      2.4.1 Desirable properties of WWW Caching ---------- 14
      2.4.2 Types of Caching ---------- 15
      2.4.3 Caching Architectures ---------- 18
      2.4.4 Cache Replacement Policies ---------- 21
      2.4.5 Cache consistency ---------- 24
   2.5 Summary ---------- 27

3 REPLACEMENT POLICIES FOR CACHING STATIC WEB OBJECTS
   3.1 Introduction ---------- 28
   3.2 Modifications to Existing Replacement Policies ---------- 29
      3.2.1 Dual Stage Caching with Victim Cache ---------- 29
         3.2.1.1 Size adjusted LRU Replacement Policies ----- 29
         3.2.1.2 Access cost and Expiration time ---------- 30
         3.2.1.3 Admission control policy ---------- 31
         3.2.1.4 Dual stage caching with victim cache ---------- 32
         3.2.1.5 Discussion of Results ---------- 34
      3.2.2 Randomized History Based Caching and Replacement --- 39
         3.2.2.1 Randomized Algorithm ---------- 39
         3.2.2.2 Randomized LRU ---------- 40
         3.2.2.3 Randomized SLRU ---------- 41
         3.2.2.4 History based Cahe replacement algorithm --- 41
         3.2.2.5 History based RLRU ---------- 42
         3.2.2.6 History based RSLRU ---------- 42
         3.2.2.7 Discussion of Results ---------- 43
      3.2.3 Modified Cache-on-Demand protocol ---------- 53
         3.2.3.1 Service Discovery ---------- 54
         3.2.3.2 Protocol Messages ---------- 54
         3.2.3.3 Request Handling ---------- 55

3.2.3.4 Admission Control -------------------------------- 58
3.2.3.5 Modifications to CoD Protocol ------------------ 59
3.2.3.6 Discussion of Results ----------------------------- 61
3.3 Summary --- 67

4 REPLACEMENT POLICIES FOR CACHING STREAMING MULTIMEDIA OBJECTS
4.1 Introduction --- 68
4.2 Cutoff Caching & Optimal Caching Techniques ---------------------- 71
4.2.1 CC algorithm -- 73
4.2.2 OC Algorithm --- 74
4.3 Popularity Function Based Replacement Policy ----------------------- 78
4.3.1 Popularity Function or Frequency Index ------------------- 78
4.3.2 The Replacement Algorithm --------------------------------- 80
4.3.3 Discussion of Results -- 82
4.4 Summary -- 87

5 SOFT COMPUTING TECHNIQUES IN WEB CACHING
5.1 Introduction -- 88
5.2 Fuzzy Replacement Algorithm --- 89
5.2.1 Implementation -- 95
5.2.2 Discussion of Results --- 100
5.3 Genetic Algorithm Replacement Policy --------------------------------- 101
5.3.1 Fitness Calculation --- 102
5.3.2 Pseudo Code --- 102
5.3.3 Implementation -- 103
5.3.4 Discussion of Results --- 104
5.4 GAR For Streaming Multimedia Objects -------------------------------- 107
5.4.1 Fitness Calculation --- 107
5.4.2 Implementation -- 108
5.4.3 Discussion of Results --- 109
5.5 Summary -- 111

6 CACHING IN MOBILE NETWORKS
6.1 Introduction -- 112
6.2 Intelligent Proxy Server with Cache-On-Demand protocol ----------- 114
6.2.1 Latency Reduction Schemes --------------------------------- 115
6.2.2 Issues in Mobile Communication Environment ------------ 115
6.2.3 An Intelligent Proxy Server ---------------------------------- 116
6.2.3.1 User Profile Database --------------------------- 118
6.2.3.2 Intelligent Proxy Server ------------------------- 118
6.2.4 Cache-On-Demand Protocol ---------------------------------- 120
6.2.5 IPS-COD Combined Protocol -------------------------------- 121
6.2.6 Discussion of Results --- 124
6.3 Dynamic Cache Invalidation Protocol ----------------------------------- 126
6.3.1 Taxonomy of Cache Invalidation Strategies ---------------- 128
6.3.2 Content of Invalidation Report ------------------------------- 130

6.3.3 Invalidation Mechanism -------------------------------------- 131
6.3.4 Cache Invalidation Scheme ---------------------------------- 132
6.3.5 Complexity of BS & BB Schemes -------------------------- 140
6.3.6 Dynamic Cache Invalidation Scheme ---------------------- 143
6.3.7 Discussion of Results --- 144
6.4 Summary -- 147
7 CONCLUSIONS -- 148

APPENDIX:

LRU IMPLEMENTATIONS FOR UNIFORMED OBJECTS ------------------------------ 153
A.1 Introduction -- 153
A.2 Higher Associativity with LRU Policy ---------------------------------- 154
A.3 Implementation Complexity --- 155
A.4 Proposed Design Approaches for LRU Implementation -------------- 156
A.4.1 Square Matrix Implementation -------------------------------- 157
A.4.2 Skewed Matrix Implementation ------------------------------- 159
A.4.3 Counter Implementation -- 160
A.4.4 Phase Implementation -- 161
A.4.5 Link List Implementation --------------------------------------- 163
A.4.6 Systolic Array Implementation -------------------------------- 165
A.5 Discussion of Results -- 167
A.6 Summary -- 172

REFERENCES -- 173

PUBLICATIONS -- 187

BRIEF BIOGRAPHY OF CANDIDATE AND SUPERVISOR ------------------------------ 190

LIST OF FIGURES

FIG NO.	*CAPTION*	*PAGE NO.*

2.1 Possible locations for deploying web caching -------------------------------------- 12

2.2 HTTP transfer between client and server -- 24

2.3 HTTP request and response headers --- 27

3.1 Hit Ratio Vs Size of the Cache for LRU, Clock pin and SLRU without Admission Control -- 35

3.2 Hit Ratio Vs Size of the Cache for LRU, Clock pin and SLRU with Admission Control -- 36

3.3 Hit Ratio Vs Size of the Cache for LRU and SLRU with Admission Control and Dual Stage Caching -- 37

3.4 Comparison of LRU(Without Admission control), LRU(With Admission control) and LRU(With Admission control and Dual Cache) -------------------------------- 38

3.5 Comparison of SLRU(Without Admission control), SLRU(With Admission control) and SLRU(With Admission control and Dual Cache) ------------------------------ 38

3.6 Cache Size vs Hit ratio for LRU & RLRU 1) N=30, M=5 2) N=8, M=2 ---------- 44

3.7 Cache size vs Hit ratio for SLRU & RSLRU 1) N=30, M=5 2) N=8, M=2 -------- 45

3.8 Cache size vs. Hit ratio for RLRU & RSLRU 1) N=30, M=5 2) N=8, M=2 ------ 46

3.9 Cache size vs. Hit ratio for HRLRU, RLRU & LRU for N=30, M=5 ------------- 47

3.10 Cache size vs. Hit ratio for HRLRU, RLRU & LRU for N=8, M=2 -------------- 48

3.11 Cache size vs Hit ratio for HRSLRU, RSLRU & SLRU for N=30, M=5 -------- 49

3.12 Cache size vs Hit ratio for HRSLRU, RSLRU & SLRU for N=8, M=2 ---------- 50

3.13 Cache size vs. Hit ratio for HRSLRU & HRSLRU for N=30, M=5 --------------- 51

3.14 Cache size vs Hit ratio for HRSLRU & HRSLRU for N=8, M=2 ----------------- 52

3.15 Cache on Demand Service Discovery -- 54

FIG NO.	CAPTION	PAGE NO.

3.16 Cache on Demand Message Formats -- 55

3.17 Maximum disk space available -- 59

3.18 When request s=50 MB, t1=10, t2=30, accepted ---------------------------------- 59

3.19 When request s=50MB, t1=20, t2=40, accepted ------------------------------------ 59

3.20 When request s=20 MB, t1=25, t2=50, rejected ------------------------------------ 59

3.21 Effect of CoD on the hit ratio of the one time admission policy normal cache for cache size 50MB --- 62

3.22 Effect of CoD on the hit ratio of the two time admission policy normal cache for cache size 50MB --- 62

3.23 Effect of CoD on the hit ratio of the three time admission policy normal cache for cache size 50MB --- 63

3.24 Effect of CoD on the hit ratio of the one time admission policy normal cache for cache size 100 MB -- 63

3.25 Effect of CoD on the hit ratio of the two time admission policy normal cache for cache size 100 MB -- 64

3.26 Effect of CoD on the hit ratio of the three time admission policy normal cache for cache size 100 MB -- 64

3.27 Effect of CoD on the hit ratio of the one time admission policy normal cache for cache size 200 MB -- 65

3.28 Effect of CoD on the hit ratio of the two time admission policy normal cache for cache size 200 MB -- 65

3.29 Effect of CoD on the hit ratio of the three time admission policy normal cache for cache size 200 MB -- 66

4.1 Video Frames and the notations --- 72

4.2 Illustration of CC Algorithm -- 73

4.3 Illustration of OC algorithm --- 75

Fig No.	*Caption*	*Page No.*

4.4 Cache Size vs. Bandwidth for streaming multimedia objects ----------------------- 77

4.5 Network Utilisation vs. Bandwidth for streaming multimedia objects ------------- 77

4.6 Hit ratio vs. Cache Size for a constant bandwidth of 40kbps ----------------------- 83

4.7 Hit Ratio vs. Cache Size for a constant bandwidth of 56kbps ---------------------- 83

4.8 Hit Ratio vs. Cache Size for a constant bandwidth of 100kbps --------------------- 84

4.9 Hit Ratio vs. Bandwidth for a constant cache size of 40GB ------------------------ 84

4.10 Hit Ratio vs. Bandwidth for a constant cache size of 80GB ----------------------- 85

4.11 Hit Ratio vs. Bandwidth for a constant cache size of 160GB --------------------- 85

4.12 Hit Ratio vs. Bandwidth for a constant cache size of 200GB --------------------- 86

4.13 Comparison of performance of LFU, LRU and FIR -------------------------------- 86

5.1 Membership Function for Input Variable Frequency --------------------------------- 91

5.2 Membership Function for Input Variable Time --------------------------------------- 91

5.3 Membership Function for Input Variable Size -- 92

5.4 Membership Function for Replacement Probability --------------------------------- 94

5.5 Hit Ratio Obtained from the trace1 for LRU and FUZZY12 ------------------------ 95

5.6 Hit Ratio obtained from the trace1 for LFU and FUZZY12 ------------------------- 96

5.7 Hit Ratio Obtained from the trace1 for SLRU and FUZZY12 ---------------------- 96

5.8 Hit Ratio Obtained from the trace1 for LRU and FUZZY24 ------------------------ 97

5.9 Hit Ratio Obtained from the trace1 for LFU and FUZZY24 ---------------------- 97

5.10 Hit Ratio Obtained from the trace1 for SLRU and FUZZY24 ------------------- 98

5.11 Hit Ratio Obtained from the trace2 for LRU and FUZZY24 --------------------- 98

5.12 Hit Ratio obtained from the trace2 for LFU and FUZZY24 ---------------------- 99

FIG NO.	CAPTION	PAGE NO.

5.13 Hit Ratio Obtained from the trace2 for SLRU and FUZZY24 -------------------- 99

5.14 Hit Ratio of GAR and LRU for varying cache sizes -------------------------------- 105

5.15 Hit Ratio GAR and LFU for varying cache sizes ---------------------------------- 105

5.16 Hit Ratio GAR and SLRU for varying cache sizes -------------------------------- 105

5.17 Time taken vs Cache Size for the GAR algorithm --------------------------------- 106

5.18 Hit Ratio of GAR-M, LRU and LFU for varying cache sizes for Sample-1 ---- 110

5.19 Hit Ratio of GAR-M, LRU and LFU for varying cache sizes for Sample-2 ---- 110

6.1 World Wide Web for Wireless Network Architecture ------------------------------ 111

6.2 Overall architecture of the IPS system -- 118

6.3 Architecture of the IPS -- 119

6.4 The GPRS Network with the proposed prefetching scheme ------------------------------- 120

6.5 Protocol working when Client requests for prefetching facility --------------------- 123

6.6 Protocol working when Client does not request for prefetching facility ----------- 123

6.7 No. of requests vs hit ratio for normal cache in an IPS-CoD system --------------- 125

6.8 Percentage of normal cache vs. Hit ratio in an IPS-CoD system -------------------- 125

6.9 Wireless Computing Environment -- 127

6.10 The Bit Sequence Scheme Protocol --- 134

6.11 Bit Sequence Example =--- 135

6.12 The Bit Sequence with Bit Count scheme protocol ------------------------------- 138

6.13 Bit Sequence with Bit Count (BB) Example ------------------------------------- 139

6.14 Structure of IR with bits sequence time stamps ---------------------------------- 140

6.15 No. of updates vs time for N=400 and cache size = 1% of the total objects --- 145

FIG NO.	CAPTION	PAGE NO.

6.16 No. of updates vs time for N=800 and cache size = 1% of the total objects --- 145

6.17 No. of updates vs time for N=4000 and cache size = 1% of the total objects -- 145

6.18 No. of updates vs time for N=200 and cache size = 3% of the total objects --- 146

6.19 No. of updates vs time for N=1000 and cache size = 3% of the total objects -- 146

6.20 No. of updates vs time for N=100 and cache size = 10% of the total objects -- 146

A.1 4 x 4 matrices initialized to zero -- 157

A.2 4 x 4 matrices with cache line 3 as the least recently used line ------------------ 157

A.3 Square Matrix Implementation -- 158

A.4 Skewed Matrix Implementation --- 159

A.5 Counter Implementation --- 161

A.6 Phase Implementation -- 162

A.7 Entry in the Previous list and Next list for Link List implementation ------------ 164

A.8 Link List Implementation -- 165

A.9 Systolic Node -- 165

A.10 Systolic Array Implementation --- 166

A.11 No. of Gates per cache set vs Associativity -------------------------------------- 169

A.12 No. of Gates vs Associativity for 128KB cache ---------------------------------- 169

A.13(a) Ratio of No. of Gates per Cache Set w.r.t 2-way Set Associativity vs Associativity --- 170

A.13(b) Ratio of No. of Gates for entire cache w.r.t 2-way Set Associativity vs Associativity --- 170

A.14 No. of Gates per Cache Line vs Associativity ---------------------------------- 171

A.15 No. of Gates vs Associativity for a 128 KB cache ----------------------------- 171

LIST OF TABLES

TABLE NO. *CAPTION* *PAGE NO.*

2.1. Comparison of Cache Mapping Functions. -- 9

2.2 Examples of commonly used parameters in cache replacement policies ---------- 14

2.3 Examples of Key-based Replacement policies --------------------------------------- 23

2.4 Summary of Existing Replacement Algorithms -------------------------------------- 25

3.1 Hit ratio of cache without admission control for various cache sizes ------------- 35

3.2 Hit ratio of cache with admission control for various cache sizes ------------------ 36

3.3 Hit ratio of cache with admission & dual stage caching for various cache sizes -- 37

3.4 LRU & RLRU Hit ratio for N=30 & 8 and M= 5 & 2 -------------------------------- 44

3.5 SLRU & RSLRU Hit ratio for N=30 & 8 and M= 5 & 2 ---------------------------- 45

3.6 RLRU & RSLRU Hit ratio for N=30 & 8 and M= 5 & 2 ---------------------------- 46

3.7 HRLRU, RLRU and LRU Hit ratio for N=30 and M= 5 ----------------------------- 47

3.8 HRLRU, RLRU and LRU Hit ratio for N=8 and M= 2 ------------------------------ 48

3.9 HRSLRU, RSLRU and SLRU Hit ratio for N=30 and M= 5 ----------------------- 49

3.10 HRSLRU, RSLRU and SLRU Hit ratio for N=8 and M= 2 ----------------------- 50

3.11 HRSLRU and HRSLRU Hit ratio for N=30, M=5 ---------------------------------- 51

3.12 HRSLRU and HRSLRU Hit ratio for N=8, M=2 ----------------------------------- 52

3.13 Precentage of CoD cache for normal cache having 40% and above hit ratio ---- 66

5.1 Fuzzy12 Rule Set -- 93

5.2 Fuzzy24 Rule Set -- 93

LIST OF ACRONYMS

BB: Bit sequence with Bit count

BS: Bit Sequence

CARP: Cache Array Routing Protocol

CC: Cut-off Caching

CGMP: Cache Group Management Protocol

CoD: Cache on Demand

CRP: Content Routing Protocol

FIFO: First in First out

FIR: Frequency Index based Replacement

GAR Genetic Algorithm based Replacement

GAR-M: Genetic Algorithm base Replacement for streaming Multimedia objects

GGSN: Gateway GPRS Support Node

GPRS: General Packet Radio Service

HDL: Hardware Description Language

HLR: Home Location Register

HRLRU: History-based Random Least Recently Used

HRLFU: History-based Random Least Frequently Used

HRSLRU: History-based Random Size-based Least Recently Used

HTML: Hyper Text Markup Language

HTTP: Hyper Text Transfer Protocol

IMSI: International Mobile station Subscriber Identity

IP: Internet Protocol

IPS: Intelligent Proxy Server

IR: Invalidation Report

ISP: Internet Service Provider

LAN: Local Area Network

LFU: Least Frequently Used

LRU: Least recently Used

MC; Mobile Client

MRU: Most Recently Used

MSS: Mobile Support Station

OC: Optimal Caching

PDA: Personal Digital Assistant

QoS: Quality of Service

RBC: Resource Based Caching

RLRU: Random Least Recently Used

RLFU: Random Least Frequently Used

RSLRU: Random Size-based LRU

RR: Random Replacement

RTT: Round Trip Time

SGSN: Serving GPRS Support Node

SLA: Service Level Agreement

SLRU: Size-Least Recently Used

TCP: Transmission Control Protocol

TS: Time Stamp

UPD: User Profile Database

UR: Update Report

URL: Universal Resource Locator

WWW: World Wide Web

CHAPTER 1

INTRODUCTION

The World-Wide Web has transformed much of the world's economy and will continue to do so. It provided a paradigm shift in the way people work and communicate. Termed as an “Information Superhighway”, the web or Internet as it is popularly known provides access to a wealth of information, which can be accessed instantaneously from one’s desktop or laptop. However, from the point of view of time - access to information on today's Web is rarely instantaneous. The growth of web resulted in a performance penalty for both the web services and its infrastructure, the Internet. While performance continues to improve over time from improvements in bandwidth and device latencies, users continue to desire yet faster response time. Likewise, content providers continue to make greater demands on bandwidth.

Good interactive response-time has long been known to be essential for user satisfaction and productivity [Brady 1986, Roast 1998]. This is also true for the Web [Bhatti 2000]. A widely-cited study from Zona Research [Zona 1999] provides evidence for the “eight second rule" in electronic commerce; ‘If a Web site takes more than eight seconds to load, the user is much more likely to become frustrated and leave the site’. Thus there is also significant economic incentive for many content providers to provide a responsive Web experience.

1.1 WEB PERFORMANCE ENHANCEMENT TECHNIQUES

There have been many studies to better understand characteristics of the web [Maltzahn 1997, Wills 1999, Brewington 2000]. The factors, which contribute to the slower speeds of the web, include the heterogeneity of network connectivity, origin server locations and distances, traffic congestion, unexpected demand and dynamic updating of information available on the web. Many researchers have considered the problem of improving web response times. Some of the proposed performance improvements are by increasing the

existing bandwidth by adopting alternative communication technologies. Also, the performance can be improved by efficiently using the existing infrastructure. Web caching and Prefetching are two such methods, which are used for improving the response times experienced by the user. The most extensively investigated solution is content caching [Aggarwal 1999, Cao 1997, Katsaros 2004]. Other solutions include the technique of prefetching [Davison 1999, Nanopoulos 2003] and cooperating caches.

1.1.1 CACHING TECHNIQUES

The web consists of Web Servers that accept requests from Web Clients for pieces of information called Web Objects. The interaction between clients and servers is by means of standard protocols, typically the Hypertext Transfer Protocol (HTTP). Any computer or device on the Internet can access web objects and thus become a Web client. A Web client that obtains Web content for the user or an application is called a Web browser. Examples of Web clients include personal computers, web-enabled phones, handheld computers, mobile devices that are web-enabled and so on. The web continues to grow rapidly and this growth puts great stress on the Internet and Web servers. Caching refers to a simple idea that if you use some information and think you might use it again in the near future, you store a copy of this information in some easily accessible place. Three features of Web caching according to Davison [Davison 2001] are:

- Caching reduces network bandwidth usage.
- Caching reduces user-perceived delays.
- Caching reduces loads on the origin server.

One central problem in Web caching is the cache replacement strategy. Cache replacement refers to the process that takes place when the cache becomes full and old objects must be removed to make space for the new one.

1.1.2 PREFETCHING TECHNIQUES

Prefetching is the cache-initiated speculative retrieval of a resource into a cache in the anticipation that it can be served from cache in the future. Most requests on the Web are made on behalf of human users, and like other human-computer interactions, the actions

of the user can be characterized as having identifiable regularities. Much of these patterns of activity, both within a user, and between users, can be identified and exploited by intelligent action prediction mechanisms [Wang 1996, Foxwell 1998]. These prediction mechanisms attempt to build a relatively concise model of the user so as to be able to dynamically predict the next action(s) that the user will take. One of the research focuses has been to apply machine-learning techniques to the problem of user action prediction on the Web, in particular, to predict the next Web page that a user will select. Such a system could anticipate each page retrieval and then, fetch that page ahead of time into a local cache so that the user experiences shorter response time [Avinoam 2000, Davison 2002]. However, the evaluation of such models in terms of response time improvement requires the incorporation of real-world considerations such as network characteristics and content caching.

1.2 OBJECTIVES AND APPROACH

Web caching is similar to memory system caching. A Web cache stores Web resources in anticipation of future requests. However, significant differences between memory system and Web caching result from nonuniformity of Web object sizes, retrieval costs and cacheability. The replacement policies designed for Web caching thus must be characteristically different from that of memory systems. There have been many replacement policies that have been proposed in the literature. In the early days of caching, simple replacement strategies were used. Therefore, research for more sophisticated replacement strategies was an important issue. Nowadays there exist several replacement policies. Although Web cache replacement in its general form seems to be a solved problem, there are new areas that need further investigation. This thesis concentrates on four aspects of Web caching:

- QoS-aware Cache Replacement: Original caching does not support quality of service (QoS). That means, caching can be seen as a best-effort service. All objects are handled equally. Introducing some sort of on-demand protocol can make the replacement process QoS-aware.
- Multimedia Cache Replacement: Multimedia cache requires new strategies to be adopted for replacement. Multimedia objects being very large in size, caching

them in entirety will lead to poor performance. Therefore, partial caching techniques have to be adopted to make caching effective and to enhance Web performance. Also, multimedia caches can use adaptation techniques to augment replacement strategies.

- Adaptive Cache Replacement: The efficiency of replacement strategies depends on the actual workload. Differences in workloads can lead to varying performance for replacement strategies. Therefore the replacement policy can be made adaptive for different workloads. Adaptive algorithms like Fuzzy Logic and Genetic Algorithms can be used for cache replacement.
- Caching in Mobile Networks: Accessing the World Wide Web data by using mobile devices is increasing due to the deployment of 2.5G and 3G services. A mobile user's web access is largely determined by the user specific preferences and the presentation of data is constrained by the capabilities of the device used. For reducing latency and disconnectivity while using Internet through a mobile client, prefetching and QoS services can be deployed.

This thesis deals with the above issues, and others, to various degrees. The study has led to investigation into a variety of areas, including soft computing techniques like Fuzzy Logic and Genetic Algorithms, simulation, networking, computer architecture and information retrieval. Web system performance has been evaluated for various replacement strategies. Generally one may choose from three general approaches to studying system performance: analytic modeling, simulation, or direct measurement. Each provides unique insights into the problem. Analytic approaches provide tools to model systems and scenarios to find trends and limits. Simulation allows for the rapid testing of a variety of algorithms without causing undue harm on the real world. Direct measurements provide grounding in reality with existence proofs and challenges for explanations. To realistically consider response times in the Web, however, strict analytic models become unmanageably complex, and thus the thesis concentrates our efforts on the latter two approaches.

The primary focus of this work is the design and development of replacement algorithms and the evaluation of such techniques for proxy caches. In addition to explorations and surveys of cache replacement techniques, the major part of the thesis will propose, implement, validate, and give examples of the use of each of the cache replacement policies for QoS-aware Web caching, Streaming Multimedia Caching and Adaptive Caching. Some of these methods can be adopted for mobile networks also. The thesis proposes a few algorithms which support caching and replacement in mobile networks.

1.3 THESIS OUTLINE AND CONTRIBUTIONS

The thesis comprises of eight chapters. Although there has been significant attention paid to cache replacement policies from the research community in the time since this thesis was conceived, this thesis makes a number of contributions.

Chapter 2 gives a review of various caching techniques for Processor architecture and World Wide Web with a summary of the literature survey and background to work reported in the thesis. In Chapter 3, some modifications to the existing replacement polices for static web objects, are discussed. The policies like Dual-Stage caching and Randomized History based techniques are explained. Also QoS-aware replacement technique like Cache-On-Demand based caching is explained. The adaptation of this algorithm for client side caching is implemented and the results are analyzed. The main contribution in this chapter is that the cache replacement has been designed to provide quality of service (QoS) replacement policies. This chapter identifies and enumerates key aspects of idealized QoS-enabled cache. It proposes and demonstrates the utility of a simple approach of Cache-On-Demand (CoD) protocol and its adaptation from client's point of view. It also will analyze the performance of cache replacement policy for small caches for CoD. Then, the performance of the basic cache in the presence of CoD caching strategy has been evaluated to show that it does not affect the performance of the cache adversely, but improves the performance for the user with CoD-enabled cache.
In Chapter 4 the thesis examines the problem of caching streaming multimedia objects and proposes and analyzes a popularity function or frequency index based cache replacement policy. Also, it explores adaptive techniques for cache replacement using

Fuzzy Logic and Genetic Algorithm on caching static web objects. After analyzing these techniques and with encouraging results, these techniques are adopted for caching streaming multimedia objects in Chapter 5. These chapters deal with the complementary idea of content-based caching in fixed network domain. Here we establish the potential of caching streaming multimedia objects with different known schemes. Then propose the various design strategies for cache replacement polices for such schemes. The analysis of the proposed policies with different set of workloads and bandwidth to establish the stability of the policy also has been explained.

Having established the potential for content-based caching, the thesis considers in Chapter 6 the approach of caching web objects for a mobile user while on the move. The adaptation of CoD approach in mobile networks has been explored and adaptivity of cache invalidation algorithm with the workload has been analyzed. Finally, Chapter 7 summarizes the results of the thesis work, with conclusions.

The main work reported in this thesis involves study of the web enhancement techniques using simulation studies. During this work, it was felt that a hardware realization of a replacement scheme should be attempted. As a first step, an LRU implementation for uniform objects was done and is reported in the Appendix of the thesis. It proposes various designs of LRU implementation for cached uniform objects in a Processor cache. These implementations have been carried out in Verilog-based simulation and synthesis using Mentor Graphics tools.

CHAPTER 2

BACKGROUND AND LITERATURE SURVEY

2.1 INTRODUCTION

Computer architectures, operating systems, and databases all use caching mechanisms to alleviate the speed gap of hierarchical storage. The prevalence of the World Wide Web has made remote-object caching increasingly important. Cache performance depends heavily on replacement algorithms, which dynamically select a suitable subset of objects for caching in a finite space. Developing such algorithms for wide-area distributed environments is challenging because, unlike traditional paging systems, retrieval costs and object sizes are not necessarily uniform. A replacement algorithm's general goal in a uniform caching environment is to reduce cache misses, usually by replacing an object with the least likelihood of re-reference. In contrast, reducing total cost incurred due to cache misses is more important in nonuniform caching environments [Bahn 2002]. A replacement algorithm in these environments should:

- make good use of observations from past references to distinguish between objects likely and not likely to be referenced in the near future. These include distinguishing not only "hot" (frequently referenced) and "cold" (infrequently referenced) objects but also those that are hot but getting colder and those that are cold but getting hotter.

- allow for efficient implementation in terms of both space and time complexities. The space needed to maintain an object's reference history should be constant, preferably a few bytes per object, and the algorithm's time complexity should not, for all practical purposes, exceed $O(\log n)$, where n is the number of objects in the cache.

- incorporate the nonuniformity factor—cost and size—fairly and effectively.

2.2 PROCESSOR CACHING

Today's high performance microprocessors operate at speeds that far outpace even the fastest of the memory bus architectures that are commonly available. One of the biggest limitations of main memory is the wait state: period of time between operations. This means that during the wait states the processor waits for the memory to be ready for the next operation. The most common technique used to match the speed of the memory system to that of the processor is caching. Cache Memory is the level of computer memory hierarchy situated between the processor and main memory. It is a very fast memory the processor can access much more quickly than main memory or RAM. Cache is relatively small, but expensive. Its function is to keep a copy of the data and code (instructions) currently used by the CPU. By using cache memory, waiting states are significantly reduced and the work of the processor becomes more effective. Cache memories remain one of the hot topics in the research community, since the ever-increasing speed gap between processor and memory only emphasizes the need for a more efficient memory hierarchy. As modern processors include multiple levels of caches, and as cache associativity increases it is important to know the effectiveness of common cache replacement policies [Al-Zoubi 2004].

In general, cache memory attempts to predict which memory elements the processor is going to need next, and loading those memory elements before the processor needs it, and saving the results after the processor is done with it. Whenever the byte at a given memory address is needed to be read, the processor attempts to get the data from the cache memory. If the cache doesn't have that data, the processor is halted while it is loaded from main memory into the cache. At that time, memory elements around the required data are also loaded into the cache. In the "real world", the direct mapped and set associative caches are by far the most common. Direct mapping is used more for level 2 caches on motherboards, while the higher-performance set-associative cache is found more commonly on the smaller primary caches contained within processors.

Table 2.1: Comparison of Cache Mapping Functions

Cache Type	**Hit Ratio**	**Search Speed**
Direct Mapped	Good	**Best**
Fully Associative	**Best**	Moderate
N-Way Set Associative, N>1	Very Good, Better as N Increases	Good, Worse as N Increases

Cache Line Replacement Algorithms

When a new line is loaded into the cache, one of the existing lines must be replaced. In a direct mapped cache, the requested block can go in exactly one position, and the block occupying that position must be replaced. In an associative cache we have a choice of where to place the requested block and hence a choice of which block to replace. In a fully associative cache, all blocks are candidates for replacement. In a set associative cache, we must choose among the blocks in the selected set. Therefore a line replacement algorithm is needed which sets up well defined criteria upon which the replacement is made. A large number of algorithms are possible and many have been implemented. Four of the most common cache line replacement algorithms are:

- *Least Recently Used* (LRU) - the cache line that was last referenced in the most distant past is replaced.
- *FIFO* (First In- First Out) - the cache line from the set that was loaded in the most distant past is replaced.
- *LFU* (Least Frequently Used) - the cache line that has been referenced the fewest number of times is replaced.
- *Random* - a randomly selected line from cache is replaced

The most commonly used algorithm is LRU replacement. It is implemented by keeping track of when each element in a set was used relative to the other elements in the set. For a two-way set associative cache, tracking when the two lines were used can be easily implemented in hardware by adding a single bit (*use* bit) to each cache line. Whenever a

cache line is referenced its use bit is set to 1 and the use bit of the other cache line in the same set is set to 0. The line selected for replacement at any specific time is the line whose use bit is currently 0. Based on the principle of the locality of reference that a recently used cache line is more likely to be referenced again, LRU tends to give the best performance. In practice, as associativity increases, LRU is too costly to implement, since tracking the information is costly. Even for four-way set associativity, LRU is often approximated – for example, by keeping track of which of a pair of blocks is LRU (which requires one bit), and then tracking which line in each pair is LRU (which requires one bit per pair). For large associativity, implementing LRU hardware is complex.

The FIFO replacement policy is again easily implemented in hardware by the cache lines as queues. The LFU replacement algorithm is implemented by associating with each cache line a counter which increments on every reference to the line. Whenever a line needs to be replaced, the line with the smallest counter value is selected, as it will be the cache line that has experienced the fewest references. Random replacement is simple to build in hardware. While it may seem that this algorithm would be a poor replacement line selection method, in reality it performs only slightly worse than any of the other three algorithms that we mentioned. For a two-way set associative cache, random replacement has a miss rate of 1.1 times higher than LRU replacement. The reason for this is easy to see. Since there are only two cache lines per set, any replacement algorithm must select one of the two, therefore the random selection method has a 50-50 chance of selecting the same one that the LRU algorithm would select yet the random algorithm has no overhead (i.e., there wouldn't be any use bit). As the cache associativity becomes higher, the miss rate for both replacement strategies become more significant, and the difference becomes higher. Hence for higher associativity cache LRU replacement policy is considered to be better than other replacement policies ignoring the complexity of the LRU hardware.

2.3 WEB CACHING VS. PROCESSOR CACHING

Web caching, where a Web cache stores Web resources in anticipation of future requests, is similar to memory system caching. However, significant differences between memory system and Web caching result from the nonuniformity of Web object sizes, retrieval

costs, and cacheability. To address object size, cache operators and designers track both the overall object hit rate (percentage of requests served from cache) and the overall byte-hit rate (percentage of bytes served from cache). Traditional replacement algorithms often assume a fixed object size, so variable sizes can affect their performance. Retrieval cost varies with object size, distance traveled, network congestion, and server load. Finally, some Web resources cannot or should not be cached, for example, because the resource is personalized to a particular client or is constantly updated. Caching is performed in various locations throughout the Web, including at the two endpoints known to a typical user — the Web browser and Web server.

Unlike CPU caches or virtual memory, which cache objects of identical size, objects in a proxy cache may have widely varying sizes – from text files of a few bytes to videos of several megabytes. Also object types such as audio or image, may be considered separately by a replacement policy, in contrast to CPU caches, which treat all data as homogeneous. On the other hand, proxy caches are simpler in that there are no “dirty” objects to write back. Obviously caching will improve the overall performance of the system as long as the *hit ratio*, i.e. the ratio of locally available information to total volume of requests, is sufficiently high. However, unlike traditional low level caching, as used in most current computer architectures, a relatively low hit ratio suffices to make using a web caching system worthwhile. This is true because the overhead of a miss (getting the object from the remote server) can be very high compared to the speed of a local search and transfer and thus the savings on a few hits are sufficient to make up for the overhead needed for searching the cache storage first.

There are several aspects, which clearly differentiate web caching from traditional caching environments. For example the fact that hit ratio considered significant even when it is low as mentioned earlier, also the fact that computation and memory at the proxy come relatively cheap and thus sophisticated cache management strategies are possible, including algorithms with different approaches for each class of objects. Another significant difference is that the bandwidths to the various servers are different (and indeed can change over time) and thus the cost of a miss does not depend on the size

of the object alone. Increased levels of performance can be achieved with web caching strategies specifically geared towards traditional web objects like web pages with static images and feature rich pages with multimedia objects.

2.4 WEB CACHING TECHNIQUES

Caching can be deployed at various points in the Internet: within the client browser, at or near the server (reverse proxy) to reduce the server load, or at a proxy server. A proxy server is a computer that is often placed near a gateway to the Internet as shown in Fig. 2.1, and that provides a shared cache to a set of clients. Client requests arrive at the proxy regardless of the Web servers that host the required web objects. The proxy either serves these requests using previously cached responses or obtains the required web objects from the original Web servers on behalf of the clients. It optionally stores the responses in its cache for future use. Hence, the goals of proxy caching are twofold: first, proxy caching reduces the access latency for a web object; second, it reduces the amount of "external" traffic that is transported over the wide-area network (primarily from servers to clients), which also reduces the user's perceived latency. A proxy cache may have limited storage in which it stores "popular" objects (web objects that users tend to request more frequently than other web objects).

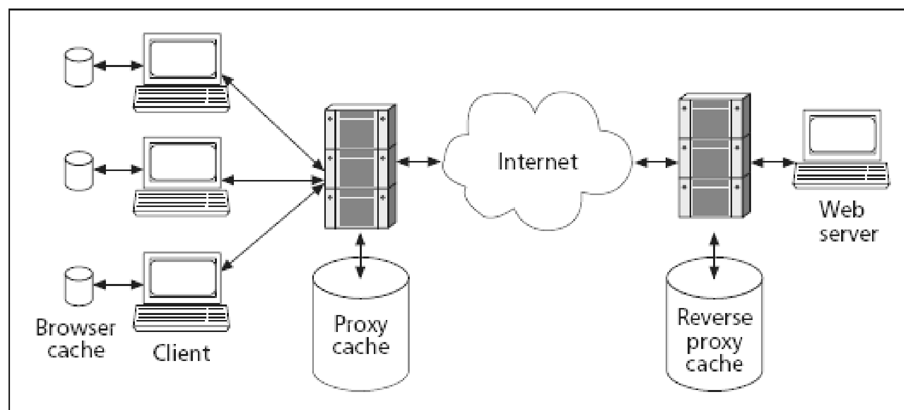


Fig 2.1. Possible locations for deploying web caching [Balamash 2004]

Caching policies for traditional memory systems do not necessarily perform well when applied to World Wide Web traffic for the following reasons:

- In memory systems, caches deal mostly with fixed-size pages, so the size of the page does not play any role in the replacement policy. In contrast, web objects are of variable size, and object size can affect the performance of the policy.
- The cost of retrieving missed web objects from their original servers depends on several factors, including the latency between the proxy and the original servers, the size of the object, and the bandwidth between the proxy and the original servers. Such dependence does not exist in traditional memory systems.
- Web objects are frequently updated, which means that it is very important to consider the object expiration date at replacement instances. In memory systems, pages are not generally associated with expiration dates.
- The popularity of web objects generally follows a Zipf-like law (i.e., the relative access frequency for an object is inversely proportional to the "rank" of that object) [Breslau 1999]. This essentially says that popular web objects are very popular and a few popular objects account for a high percentage of the overall traffic. Accordingly, object popularity needs to be considered in any Web caching policy to optimize a desired performance metric. A Zipf-like law has not been noticed in memory systems. While memory systems are known to exhibit temporal locality, this concept is quiet different from object popularity.

Several web replacement policies have been proposed in the literature. Such policies attempt to optimize various performance metrics, including the byte hit ratio and the average download time [Bahn 2002]. Replacement policies rely on key metrics (parameters) to achieve their goals. Many of them use the recency or frequency information of past references; which are well exhibited in World Wide Web traffic [Jin 2000a, Jin 2000b]. For example, the well known least recently used (LRU) caching policy employs the time since last access as its only parameter. Some policies combine both recency and frequency information, along with some other parameters such as the size of the object and the cost associated with each object. Since web objects are of variable size, two objects with different sizes and with the same likelihood of being referenced can have different costs. The cost of an object includes the time and processing overhead associated with retrieving the object from the original server. The

lifetime of the object and the cache space overhead associated with the object size are also considered as cost factors [Balamash 2004]. Table 2.2 summarizes some of the parameters used in cache replacement policies.

Table 2.2: Examples of commonly used parameters in cache replacement policies

Parameter	Rationale
Last access time	Web traffic exhibits strong temporal locality
Number of previous accesses	Frequently accessed objects are likely to be accessed in the near future
Average retrieval time	Caching objects with high retrieval times can reduce the average access latency
Object Size	Caching small objects can increase the hit ratio
"Expires" or "Last Modified" HTTP header values	Caching an expired object wastes cache space and results in a miss when object is accessed.

In [Wang 1999], Wang provides a good survey of Web caching schemes. It addresses several topics related to Web caching, including cache architectures, protocols, replacement policies, prefetching, cache coherency, proxy placement, user access prediction, and dynamic objects caching.

2.4.1. DESIRABLE PROPERTIES OF WEB CACHING

Besides the obvious goals of a caching system, a web caching system must have a number of properties from the user's perspective and from the server's perspective.
The desirable properties from the user's perspective are:

- *Fast access*: Access latency is an important parameter to measure the quality of web service. A web caching system must aim at reducing the access latency so that it makes user experience of surfing the internet much better as compared to the network which does not use a caching system.
- *Robustness*: This indicates the availability of the system, which is another important parameter to measure quality of web service. This includes the availability of the service even when proxies crash, eliminating single point of failure. When a failure occurs, the system must gracefully fail so that it is easy to

recover from failure. The caching system design should ensure such a fault tolerant system design.

- *Transparency*: The operation of caching system must be transparent for the user, who is concerned only about faster response, higher availability and easier access.

The desirable properties from the server perspective are:

- *Scalability*: We know that the amount of growth the web has seen is exponential and it will continue to be so in the years to come. Any caching system designed must be should scale well with the increasing size and density of network. This requires all the protocols employed in the caching system to be as lightweight as possible.
- *Load balancing*. It's desirable that the caching scheme distributes the load evenly through the entire network. A single proxy/server shouldn't be a bottleneck (or hot spot) and thereby degrades the performance of a portion of the network or even slow down the entire service system.
- *Simplicity*. Simplicity is always an asset. Simpler schemes are easier to implement and likely to be accepted as international standards. We would like an ideal Web caching mechanism to be simple to deploy.

2.4.2. TYPES OF CACHING

There has been several research and study work being undertaken in the field of Web Caching. They deal with different caching architectures and cache deployment options. Some deployments go hand in hand with the caching system architecture, whereas some architectures allow for a variety of deployment options [Barish 2000, Wang 1999, Rabinovich 2002]. Web caching can be classified on the basis of the cache deployment as follows:

Proxy Caching

A proxy cache server intercepts HTTP requests from clients, and if it finds the requested object in its cache, it returns the object to the user. If the object is not found, the cache

goes to the object's home server, the originating server, on behalf of the user, gets the object, possibly deposits it in its cache, and finally returns the object to the user. Proxy caches are usually deployed at the edges of a network (i.e., at company or institutional gateway or firewall hosts) so that they can serve a large number of internal users. The use of proxy caches typically results in wide-area bandwidth savings, improved response time, and increased availability of static Web-based data and objects. One disadvantage to this design is that the cache represents a single point of failure in the network. When the cache is unavailable, the network also appears unavailable to users. The other disadvantage is with respect to scalability. As demand rises, one cache must continue to handle all requests. There is no way to dynamically add more caches when needed, as is possible with transparent proxy caching.

Reverse Proxy Caching

The other way of deploying proxy cache is the notion of reverse proxy caching, in which caches are deployed near the origin of the content instead of near clients. This is an attractive solution for servers that expect a high number of requests and want to ensure a high level of quality of service. Reverse proxy caching is also a useful mechanism when supporting Web hosting farms (virtual domains mapped to a single physical site), an increasingly common service for many Internet service providers (ISPs). Note that reverse proxy caching is totally independent of client-side proxy caching. In fact, they may coexist and collectively improve overall performance.

Transparent Caching

Transparent proxy caching is similar to the proxy server approach. Transparent caches work by intercepting HTTP requests and redirecting them to Web cache servers or cache clusters. This style of caching establishes a point at which different kinds of administrative control are possible; for example, deciding how to load balance requests across multiple caches. The filtering of HTTP requests from all outbound Internet traffic may add additional latency. There are two ways to deploy transparent proxy caching: at the switch level and at the router level. Router-based transparent proxy caching uses policy-based routing to direct requests to the appropriate cache(s). For example, requests

from certain clients can be associated with a particular cache. In switch-based transparent proxy caching, the switch acts as a dedicated load balancer. This approach is attractive because it reduces the overhead normally incurred by policy-based routing. Although it adds extra cost to the deployment, switches are generally less expensive than routers.

Adaptive Web Caching

Adaptive Web caching [Michel 1998] views the caching problem as one of optimizing global data dissemination. A key problem adaptive caching targets is the “hot spot” phenomenon, where short-lived Internet content can, overnight, become massively popular and in high demand. Adaptive caching consists of multiple distributed caches which dynamically join and leave cache groups (referred to as *cache meshes*) based on content demand. Adaptivity and the self-organizing property of meshes are a response to those scenarios where demand for objects gradually evolves and those where demand spikes, or is otherwise unpredictably high or low. Adaptive caching uses the Cache Group Management Protocol (CGMP) and Content Routing Protocol (CRP). CGMP specifies how meshes are formed, and how individual caches join and leave those meshes. CRP is used to locate cached content from within the existing meshes. This technique relies on multicast communication between cache group members and makes use of URL tables to intelligently determine to which overlapping meshes requests should be forwarded. One of the key assumptions of the adaptive caching approach is that the deployment of cache clusters across administrative boundaries is not an issue. If the virtual topologies are to be most flexible and have the highest chance of optimizing content access, administrative boundaries must be relaxed so that groups form naturally at proper points in the network.

Push Caching

As described in [Bhide 2002], the key idea behind push caching is to keep cached data close to the clients requesting that information. Data is dynamically mirrored as the originating server identifies where requests originate. As with adaptive caching, one main assumption of push caching is the ability to launch caches that may cross administrative

boundaries. However, push caching is targeted mostly at content providers, which will most likely control the potential sites at which the caches could be deployed. Unlike adaptive caching, it does not attempt to provide a general solution for improving content access for all types of content from all providers. One study [Tiwari 1999] found that well-constructed push-based algorithms can lead to speedups of between 1.27 and 2.43 as compared to traditional cache hierarchies. This study also notes the general dilemma that push caching encounters: forwarding local copies of objects incurs costs (storage, transmission), while overall performance and scalability are only seen as improved if those objects are indeed accessed. Also combination of push and pull caching is going to yield a higher quality of service to the end user.

Active Caching

The WisWeb project at the University of Wisconsin explored how caching can be applied to dynamic objects [Cao 1998]. Their motivation is that the increasing amount of personalized content makes caching such information difficult and not practical with current proxy designs. Indeed, a recent study [Creres 1998] of a large ISP trace revealed that over 30 percent of client HTTP requests contained cookies, which are HTTP header elements typically indicating that a request be personalized. As Web servers become more sophisticated and customizable, and as one-to-one marketing e-commerce strategies proliferate the Internet, the level of personalization is anticipated to rise. Active caching uses applets, located in the cache, to customize objects that could otherwise not be cached. When a request for personalized content is first issued, the originating server provides the objects and any associated cache, cache the applets. When subsequent requests are made for that same content, the cache applets perform functions locally (at the cache) which would otherwise (more expensively) be performed at the originating server. Thus, applets enable customization while retaining the benefits of caching.

2.4.3. CACHING ARCHITECTURES

A caching architecture should provide the paradigm for proxies to cooperate efficiently with each other. Caches sharing mutual trust may assist each other to increase the hit rate.

Various caching architectures are hierarchical caching, distributed caching and hybrid caching

Hierarchical Caching Architecture

Hierarchical caching was pioneered in the Harvest Cache [Chankhunthod 1996]. A series of caches are hierarchically arranged in a tree like structure; these caches leverage from each other when an object request arrives and the receiving cache experiences a miss. Caches are placed at multiple levels of the network. Requests for a object travel up the caching hierarchy until the object is hit at some cache level. When the object is found, either at a cache or at the original server, it travels down the hierarchy, leaving a copy at each of the intermediate caches along its path. In hierarchical design, child caches can query parent caches, children can query each other but parents can never query their children.

A hierarchical architecture is more bandwidth efficient, particularly when some cooperating cache servers do not have high-speed connectivity. In such a structure, popular Web pages can be efficiently diffused towards the demand. With hierarchical caches, it has been observed that parent nodes can become heavily swamped during child query processing. Commercial caches such as Network Appliances NetCache employ clustering to avoid this swamping effect.

However, there are some disadvantages associated with this caching architecture

- Additional delays may be introduced at different levels.
- An object may be duplicated at different levels, so multiple copies of the same object may exist leading to coherency problems.
- Significant coordination may be required among the caches placed at key access points in the network.
- Long queuing delays may be introduced at high levels because with increase in levels, the parent nodes become heavily swamped during the child query processing.

Distributed Caching Architecture

In distributed Web caching systems, there are no caches at intermediate levels other than the institutional caches which serve each others' misses [Povey 1997]. All the institutional level caches maintain metadata information about the contents of other caches at the same level, in order to decide from which institutional cache a miss object can be retrieved. This metadata information can be distributed in the system using a hierarchical mechanism. With distributed caching, most of the traffic flows through low network levels, which are less congested and no additional disk space is required at intermediate network levels. Web caching systems are composed of multiple distributed caches to improve

- *System Scalability*: Caches can serve high degree of concurrent client requests
- *Availability*: Systems can survive the failure of some caches there by becoming more fault tolerant.
- *Leveraging physical locality*: Having caches closer in proximity to certain groups of users helps in reducing network latencies.
- *Load Balancing*: Caches can query each other; distributing objects among them and intercache communication helps in load balancing and resolving requests internally.

There are several approaches to the distributed caching. The Harvest group designed the Internet Cache Protocol (ICP) [RFC 2186], which supports discovery and retrieval of objects from neighboring caches as well as parent caches. Another approach to distributed caching is the Cache Array Routing protocol (CARP) [Valloppillil 1998], which divides the URL-space among an array of loosely coupled caches and lets each cache store only the objects whose URL are hashed to it. Another technique related to cache-to-cache communication is the notion of cache digests, such as those implemented by Squid [Wessels 1998] and the Summary Cache [Fan 2000]. Digests can be used to reduce intercache communication by summarizing the objects contained in peer caches. Thus, request forwarding can be more intelligent and more efficient. This approach is

similar to the use of URL routing tables in adaptive caching as a more intelligent way to forward requests.

Hybrid Caching Architecture

In hierarchical caching the connection time is shorter. So placing additional copies at intermediate levels reduces the retrieval latency for small objects. Distributed caching has shorter transmission times and higher bandwidth usage than hierarchical caching. A well-configured hybrid scheme can combine the advantages of both hierarchical and distributed caching. In a hybrid scheme, caches may cooperate with other caches at the same level or at a higher level using distributed caching. ICP is a typical example. The object is fetched from a parent/neighbor cache that has the lowest RTT. Rabinovich et al. [Rabinovich 1998] proposed to limit the cooperation between neighbor caches to avoid obtaining objects from distant or slower caches, which could have been retrieved directly from the origin server at a lower cost.

2.4.4. CACHE REPLACEMENT POLICIES

Effectiveness of proxy caches depends on object placement and replacement algorithms that can yield high hit rate. A good admission control policy is especially important while caching non-uniformly sized objects, because a considerable amount of disruption can be caused when an object is added and others are purged from the cache. Highly frequent replacements may cause space and time wastage, and storage of objects, which are never hit. Therefore, an optimal cache replacement policy is essential.

To summarize, the important factors (characteristics) of web objects that can influence the replacement process are

- Recency: time of (since the last reference of the object)
- Frequency: number of requests to an object
- Size: size of the web object in bytes.
- Cost: cost of fetching an web object from its origin server
- Modification time: time of (since) last modification
- Expiration time: time when an object gets stale and can be replaced immediately.

Most of the replacement policies designed for web caches use some of these factors for decision making. They can be classified as suggested in [Wang 1999, Aggarwal 1999].

Traditional Replacement Policies

- *Least Recently Used (LRU):* It removes the object which was least recently requested by using a reference count to store information about time of last access.
- *Least Frequently Used (LFU):* It evicts the object whose frequency of access is the least.
- *Pitkow/Recker:* [Pitkow 1994] evicts objects in LRU order, except if all objects are accessed within the same day, in which case the largest files are removed first.

Key-Based Replacement Policies

The objects are replaced based on a primary key. In case there is a tie, it is broken using a secondary key and tertiary key, etc. Few of these policies are

- *Size:* [Williams 1996] This strategy removes the object having largest size. The LRU strategy is applied for the objects with the same size. One variant is LOG2_SIZE which uses $\lfloor \log_2(size) \rfloor$ instead of size.
- *LRU-MIN:* [Abrams 1995] A particular size S is chosen and if there are any objects in the cache which have size of at least S, the least recently used such object is removed from the cache. In case there are no such objects, then starting from the objects with size at least S/2 in LRU order, objects are evicted i.e. the object which has the largest log(size) and is the least recently used among all such objects will be evicted first.
- *LRU-Threshold:* [Abrams 1995] It is the same as LRU, but objects larger than a certain threshold size are never cached.
- *Hyper-G:* [Wooster 1997] It is a refinement of LFU, breaks ties using the recency of last use and size.
- *Lowest Latency First:* [Williams 1996] It minimizes average latency by evicting the object with the lowest download latency first.

Table 2.3 shows examples of the key based replacement policies.

Cost-Based Replacement Policies

In these policies, a potential cost function derived from different factors such as time since last access, size of the object, entry time of the object in the cache, transfer time cost, object expiration time etc. is used to choose the objects to be replaced. Some of the algorithms based on this policy are

- *Greedy Dual-Size(GD-Size) :* [Cao 1997] It associates a cost with each object and evicts the object with the lowest cost to size ratio.
- *Hybrid:* [Wooster 1997] It associates a utility function with each object and evicts the one with the least utility to reduce the total latency.
- *Least Normalized Cost Replacement(LNC-R-W3):* [Scheuermann 1997] This algorithm employs a rational function of the access frequency, transfer time cost and the size and removes the object with the lowest value for this function.
- *Server-assisted scheme:* [Cohen 1998] The server generates a histogram of inter-request times by observing its request logs. It calculates a value for an object in terms of its fetching cost, size, next request time and cache prices during the time period between requests and evicts the object having the least value.

Table 2.3 Examples of Key-based policies

Name	Primary Key	Secondary Key	Tertiary Key
LRU	Time Since Last Access		
FIFO	Entry Time of Object in Cache		
LFU	Frequency of Access		
SIZE	Size	Time Since Last Access	
LOG2-SIZE	$\lfloor \log_2(Size) \rfloor$	Time Since Last Access	
HYPER-G	Frequency of Access	Time Since Last Access	Size

There have been other proposals for classification of replacement policies in the literature. [Jin 2000a, Bahn 2002, Podlipnig 2003]. There have been other replacement strategies proposed and discussed. To sum up, a great deal of effort has been made to maximize the web object hit rate and minimize the latency in delivery of contents to the clients. One popular caching product realized in software and is the freely available Squid proxy cache [Squid 1998]. In its original implementation Squid uses LRU with

some modifications. The replacement algorithm is not triggered on demand but runs periodically every second. Squid has a low and a high water mark. When the disk usage is close to the low water mark, the replacement is less aggressive (fewer objects removed). When the usage is close to the high water mark, the replacement is more aggressive (more objects removed). The replacement depends among other things on an LRU-threshold that is dynamically calculated, based on the current cache size and the low and high water marks. An object is removed if the time since last access is greater than this threshold. Furthermore, Squid supports LFU and GD-Size replacement policies. Table 2.4 gives the summary of the algorithms described so far.

2.4.5. CACHE CONSISTENCY

In its simplest form, the web is a set of servers and clients. To retrieve a particular web resource, the client attempts to communicate over the internet to the origin web server as shown in Fig 2.2.

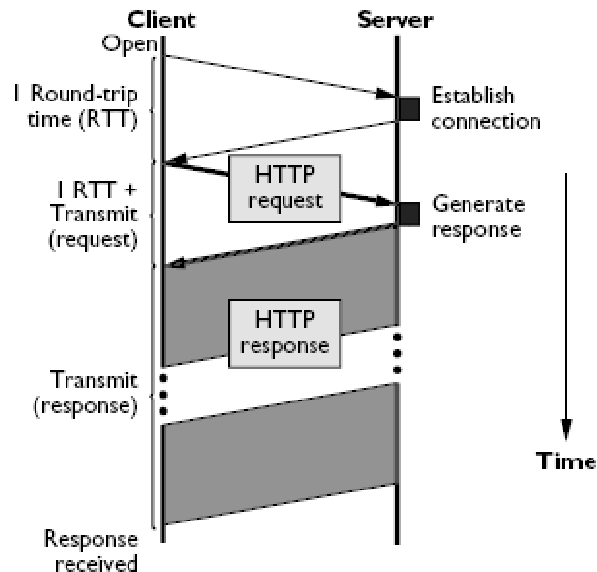


Fig 2.2 HTTP transfer between client and server. [Davison 2001]

Table 2.4 Summary of Existing Replacement Algorithms

Algorithm	Reference based on recency history	Reference based on frequency history	Consideration of nonuniformity of the object	Time Complexity	Space Complexity	Advantages	Disadvantages
LRU	Last reference time	No	No	O(1)	O(1)	Simple to implement	Fixed performance measure;
LFU	No	Number of references	No	O(log n)	O(1)	Keeps many objects in cache	Fixed performance measure;
Size	No	No	Size in biased manner	O(log n)	O(1)	Keeps many objects in cache	Fixed performance measure; doesnot consider reference history
LRU-min	Last reference time	No	Size in biased manner	O(n)	O(1)	Keeps many objects in cache	Time complexity; Fixed performance measure
Hybrid	No	Number of references	Size and latency	O(log n)	O(1) + per-server information	Good estimation of download latency	Per-server information overhead; fixed performance measure
LNC-R-W3	k-th reference time	Based on k-th reference time	Normalised manner	O(n)	O(k)	Normalized contribution to cost saving ration	Time Complexity
GD-Size	Last reference time	No	Weighted manner	O(log n)	O(1)	No parameter; Can optimize any performance measure	Does not consider frequency

To connect to the server, the client needs the host's numerical identifier. It queries the domain name system (DNS) to translate the hostname to its Internet Protocol (IP) address, with which it can establish a connection to the server and request the content. Once the Web server has received and examined the client's request, it can generate and transmit the response. As Fig 2.2 shows, each step in this process takes time. The hypertext transfer protocol (HTTP) specifies the interaction among Web clients, servers, and intermediaries. Requests and responses are encoded as headers that precede optional bodies containing content. Fig 2.3 shows one set of request and response headers. The first request header shows the method used (GET), the resource requested ("/"), and the version of HTTP supported (1.1).

Another commonly used method is POST, which allows clients to send content with a request (for instance, to carry variables from an HTML form). The first line of the response header shows the HTTP version supported and a response code with standard values. The headers of an HTTP transaction also specify aspects relevant to an object's cacheability. The relevant headers from the example in Fig 2.3 include Date, Last-Modified, ETag, Cache-Control, and Expires. For example, in HTTP GET requests that include an If-Modified-Since header, Web servers use the Last-Modified date on the current content to return the object only if the object changed after the date of the cached copy. The origin server needs an accurate clock to calculate and present modification and expiration times in the other tags. An ETag (entity tag) represents a signature for the object and allows for a stronger test than If-Modified-Since: If the signature of the current object at this URL matches the signature of the cached one, the objects are considered equivalent. The Expires and Cache-Control: max-age headers specify how long the object can be considered valid. For slowly or never-changing resources, an explicit expiration date tells caches how long they can keep the object (without requiring the cache to contact the origin server to validate it).

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

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

Fig 2.3 HTTP request and response headers [Davison 2001]

2.5. SUMMARY

In this chapter, the basics of caching techniques as applied to computer architecture and web has been discussed. Further a detailed description of types of caching in web and caching architectures is also presented. The replacement policies with merits and demerits for each replacement strategy have been analysed.

CHAPTER 3

REPLACEMENT POLICIES FOR CACHING STATIC WEB OBJECTS

3.1 INTRODUCTION

The size and cost concerns make web caching a much more complicated problem than traditional caching. In the previous chapter, we discussed variety of web caching algorithms proposed so far. Effectiveness of proxy caches depends on document placement and replacement algorithms that yield high hit rate. A good admission control policy is also important while caching non-uniformly sized objects, because a considerable amount of disruption can be caused when an object is added and others are purged from the cache. Highly frequent replacements may cause space and time wastage, and storage of objects, which are never hit. Therefore, an optimal cache replacement policy needs to be designed.

There are a number of results on the optimal offline replacement algorithms and online competitive algorithms, on simplified versions of the Web caching problem. The variable document sizes in web caching make it complicated to determine an optimal offline replacement algorithm. If one is given a sequence of requests to uniform size blocks of memory, it is well known that the simple rule of evicting the block whose next request is farthest in the future will yield the optimal performance [Belady 1966]. In the variable-size case, no such efficient offline algorithm is known.

For the cost consideration, there have been several algorithms developed for the uniform-size variable-cost paging problem. GreedyDual [Young 1994], is actually a range of algorithms which include a generalization of LRU and a generalization of FIFO. The name GreedyDual comes from the technique used to prove that this entire range of algorithms is optimal according to its competitive ratio. The competitive ratio is essentially the maximum ratio of the algorithms cost to the optimal offline algorithm's cost over all possible request sequences.

3.2 MODIFICATIONS TO EXISTING REPLACEMENT POLICIES

Some improvements to the existing replacement policies and the admission policies, which will help in improving the hit ratio for static web documents, are presented here. Three caching schemes for static web documents are discussed in detail. The first one is a dual-stage victim based replacement policy which replaces a single-level web caching to dual-level web caching. The victims of the replacement can be cached using a second level of cache memory to enhance the life of a web document. The second one is a randomized history based caching and replacement, where a history based approach is combined with a randomized LRU approach. The third one is a Cache-on-Demand (CoD) protocol based caching which has been modified to include few more features to effectively cache the objects by demand and provide quality of service (QoS) in place of best effort service. This later scheme has been extended to wireless networks also.

3.2.1. DUAL-STAGE CACHING WITH VICTIM CACHE

The simplest form of a replacement policy is Least Recently Used (LRU). Several extensions to this simplest form have been implemented. The size adjusted LRU is one such algorithm [Aggarwal 1999], which is popular and has a good performance. We extend this algorithm to include an admission control policy and a multilevel cache (victim cache), which produces better performance than the ordinary schemes. In fact this extension can be used with any other caching algorithm. Here, first the Size-adjusted LRU policy is discussed.

3.2.1.1 Size adjusted LRU Replacement Policy

When an object is to be inserted into the cache, more than one object may need to be removed in order to create sufficient space. In the LRU, objects are greedily removed from the cache in the order of recency of last access until enough space is created for the incoming object. But such a policy is not the only possible LRU generalization for handling objects of non-uniform size. Charu Agarwal et.al [Aggarwal 1999] proposed a

heuristics to solve an optimization problem, which mimics but generalizes the LRU criteria for uniform sized objects. A brief discussion of the scheme is given here.

Assume that there are *N* objects, and that object *i* has size S_i. A counter is maintained and incremented each time there is a request for an object. This counter is named as the Dynamic Count (d_i). Let *i* be the object requested. The object will be fetched if present in the cache; a hit. If miss, assuming *i* satisfies the admission control requirements, it has to be decided that which objects have to be purged from the cache.

The following steps are defined:

Dynamic count (d_i) is maintained for all the objects in the cache table
Object requested (irrespective of cache hit or miss) {
    d_i^{++} for all the objects}
When a new object enters:
    $d_i = 1$
Objects arrangement:
    In the order of size x dynamic count $(S_i \times d_i)$
When a new object enters:
    Insert correctly in the cache table.
    Object(s) with the highest $(S_i \times d_i)$ count value is thrown off to make space for the incoming object.

Also the objects can be ordered by the ratio of cost to size. If so, then choose the objects with the highest cost-to-size ratio, one by one, until no more objects are to be purged. The cost-to-size ratio for the object *i* is $\frac{1}{(S_i \times d_i)}$. So, we reindex the objects in order of non-decreasing values of $(S_i \times d_i)$. Then we greedily pick the highest index objects one by one and purge them from the cache until we have created sufficient space for the incoming object. This is defined as Size-Adjusted LRU, or SLRU replacement scheme.

3.2.1.2 Access Cost and Expiration time

The scheme discussed here attempts to maximize the probability of a cache hit. If c_i is the access cost of object i and y_i be the decision variable to decide the object should be thrown out or not from the cache where $y_i = 0$ (for not throwing out) and $y_i = 1$ (for throwing out), then the generalized objective function is defined as $\sum \frac{c_i \times y_i}{d_i}$. Similarly,

for arranging objects in the Size adjusted LRU, considering the access cost, the generalized function for arrangement will be $\frac{(S_i \times d_i)}{c_i}$. We can observe that if all values of c_i are uniform (1, for example), then the replacement policy will be the SLRU.

To define the expiry time we define δt_{i1} to be the difference between current time (*t)* and the time when it was *last* accessed and δt_{i2} be the difference between the object expiration time and *t*. Then the refresh overhead factor for an incoming object *i* is defined to be $r_i = \min\left\{1, \frac{\delta t_{i1}}{\delta t_{i2}}\right\}$. This value is approximately the reciprocal of the number of expected accesses before the object needs to be refreshed. We can incorporate the refresh overhead factor into the replacement policy by ordering objects in terms of nondecreasing values of $\frac{(S_i \times d_i)}{c_i \times (1 - r_i)}$, and greedily purging those objects with the highest indexes.

3.2.1.3 Admission control Policy

When a requested object is obtained by the proxy, to place in a cache, we have to check whether the object entry into the cache is profitable or not. An admission control policy decides whether or not it is profitable to cache an object. A good admission control policy is very important when caching non-uniform size objects, because an object, which is added, may replace not one but several objects in the cache. When Objects are replaced frequently it may lead to wastage of space. A small additional cache, called the auxiliary cache, which maintains the identities of some X number of objects can be used. For each object in this auxiliary cache we also maintain timestamps of the last access, measured both in terms of the number of object accesses and time, together with access cost and expiration time data. The access counter is incremented each time an object is requested from the cache, whether or not that request can be fulfilled. Because the auxiliary cache contains identities of objects rather than the objects themselves, its size is negligible compared to that of the main cache. LRU Order is used for this auxiliary cache.

An algorithm for this can be defined as:

```
object i requested
  if (object i present in auxiliary cache){
        determine the objects to be thrown out to make space for the incoming object
                if (caching of object profitable by c_i (1 - r_i)/d_i )
                put in the main cache
  }
        else{
           don't cache
  }
Update auxiliary cache by LRU
```

The sum $\sum c_i \frac{(1-r_i)}{d_i}$ of the set of candidate outgoing objects is determined using the replacement scheme. We admit an object only if it is profitable to do so. Observe that the information needed can be obtained from the auxiliary cache. After this iteration, the time stamp of the object i is updated.

3.2.1.4 Dual-stage caching with Victim Cache

The above scheme is modified by including Dual-stage Caching scheme. So far, we have considered only a single level of caching objects. Though the term auxiliary cache may be misleading, we can note that it does not maintain the actual cache entities. As the cache space in proxy is increasing and space not being a constraint, we propose a Dual-stage caching scheme wherein we maintain an additional cache called the Victim cache**.** The admission control policy decides whether the object entry into the cache is worthwhile or not. The cache replacement policy determines the objects to be cached considering various parameters, mainly the recency of use and the size.

While the object i_k enters the cache, after undergoing admission control, the cache replacement policy determines the objects to be purged to make space for the incoming object. These objects that are candidates for purge are given a second chance. This is accomplished with the help of a victim cache. The victim cache is a subordinate cache memory in addition to the main cache which will have the objects that are the victims of the replacement. The size of the victim cache is taken as (1/6)th of the proxy caching

memory as a thumb rule, which is similar to the victim caches in computer architecture proposed by Norman Jouppi [Jouppi 1990].

The concept of victim cache as proposed by Jouppi [Jouppi 1990, Hennessy 2003] is popular in computer architecture; it has been proved that it improves the hit rate by considerable amount due to locality of reference. In World Wide Web, considering the workload characteristics as discussed in [Arlitt 1996], some of the web objects, especially in institutional workload, have a fair number of references after the object is purged. By using SLRU the caching space is effectively used but to achieve a greater hit rate a second chance for the purged object is essential. In this method, whenever an object leaves the main cache, instead of immediately purging it, is stored in the victim cache. The victim cache has a simple replacement algorithm like the First in First out (FIFO) to reduce the computational time and to have a simple data structure. Thus the object in the victim cache is purged on the first come first purge basis. The algorithm for the proposed caching scheme is:

```
Object i is requested (after the admission control) {
        Check in the main cache
        Check in the victim cache
            If present in the victim cache {
                        Place the object in the main cache
                    }
            If not present in the main cache {
              decide the objects to be removed from the main cache
            place these objects in the victim cache in the order of FIFO
                    }
      }
```

This method of cache replacement accomplished by a dual stage caching will give a second chance to the objects which are to otherwise purge thus enhancing the performance of the Proxy Cache.

3.2.1.5 Discussion of Results

To compare the performance of various replacement policies with dual-stage admission control and victim cache, we chose three different caching schemes, the naive LRU replacement policy, the CLOCK PIN policy and the SLRU. For obtaining the experimental results, we employed trace driven simulations.

These schemes were implemented in conjunction with the admission control that was discussed earlier in this section. We compare these results with those obtained without the admission control policy. The final extension proposed, namely the dual stage caching with the victim cache was also simulated and the results are discussed. We compare SLRU, in order to show that the two schemes are virtually identical in terms of performance. We are interested in examining the performance of the algorithms under the assumption that objects have varying sizes, relative frequencies, and combinations of these two factors.

It is well known that the performance of caching policies for Web objects often depends on whether smaller objects have higher frequency or vice versa. The LRU scheme is very robust for uniform size objects and varying distributions of relative frequencies. All the schemes that we compare are in fact generalizations of LRU in one way or the other and, consequently, it is useful to see how the correlation of size and frequency factors into the robustness of the proposed schemes. For the simulation, real time traces have been used. We considered the logs for two weeks, traces taken from an institutional proxy (BITS Squid server) with number of entries in the log being 70,000 user accesses. Most of the frequently accessed pages had relatively smaller sizes. We ran the simulation for varying values of the cache capacity. The performance curves for the different replacement policies of the traces are illustrated in Fig. 3.1 to Fig 3.5

Trace driven simulation without Admission control

As shown in Fig 3.1, the simulation results for the LRU, SLRU and the CLOCK PIN without the admission control policy clearly shows that the SLRU outperforms the other

two schemes of cache replacement. However the hit ratio obtained is found to be relatively very low. The SLRU algorithm is found to be better than the other algorithms.

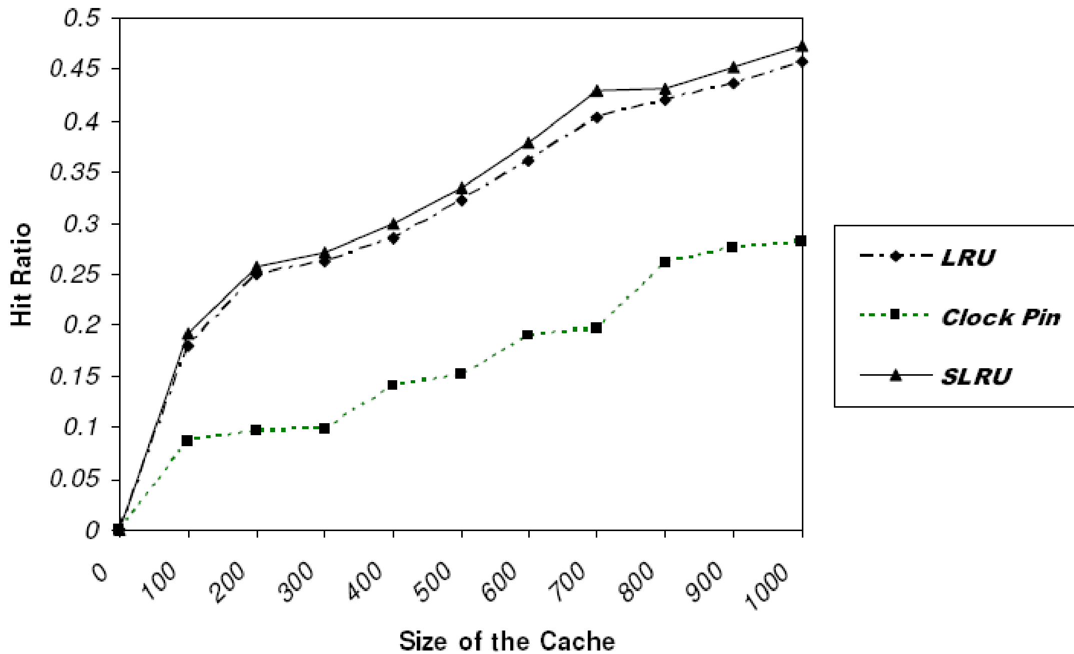


Fig. 3.1. Hit Ratio Vs Size of the Cache for LRU, Clock Pin and SLRU policies without Admission Control

Table 3.1 Hit ratio of cache without admission control for various cache sizes

Size (MB)	100	200	300	400	500	600	700	800	900	1000
LRU	0.1799	0.2492	0.2623	0.2860	0.3218	0.3610	0.4033	0.4203	0.4364	0.45
Clock Pin	0.0866	0.0966	0.0988	0.1407	0.1522	0.1895	0.1967	0.2610	0.2756	0.2814
SLRU	0.1918	0.2568	0.2711	0.2990	0.3343	0.3781	0.4282	0.4319	0.4531	0.4729

Trace driven simulation with Admission control

After adopting the admission control policy discussed earlier, the trace driven simulation results using the same set of traces, is as shown in Fig 3.2. It can be seen that the SLRU policy is still found to perform better than the other two algorithms. As expected the cache becomes more stable as it results in the entry of the objects only after checking of a

set of conditions in the admission control. This stable cache memory is found to give better hit ratio when compared to the one without the admission control policy.

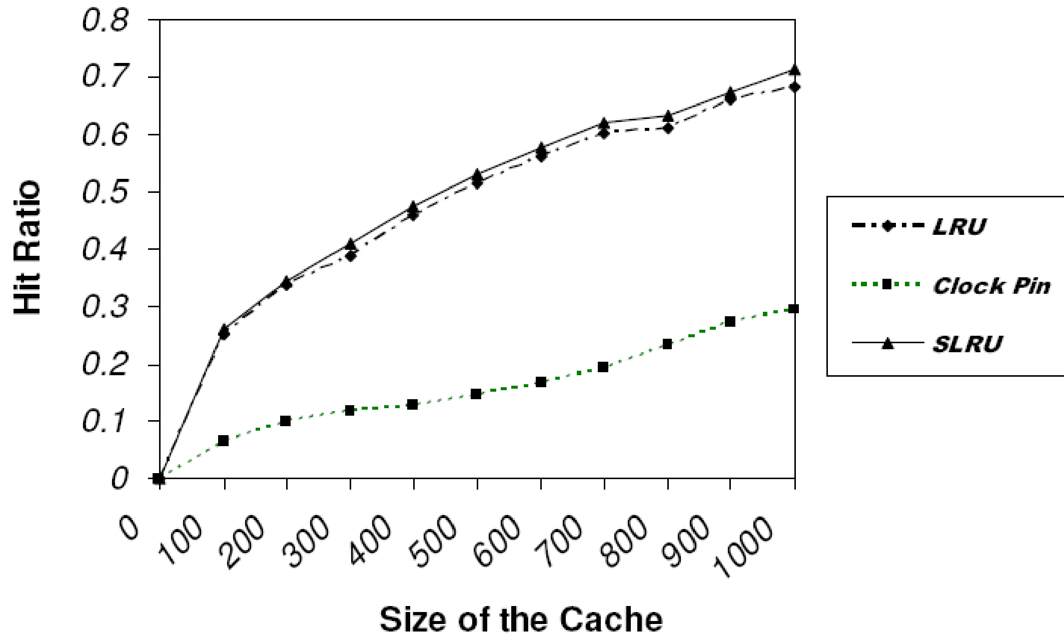


Fig. 3.2. Hit Ratio Vs Size of the Cache for LRU, Clock pin and SLRU with Admission Control

Table 3.2. Hit ratio of cache with admission control for various cache sizes

Size (MB)	100	200	300	400	500	600	700	800	900	1000
LRU	0.2500	0.3375	0.3889	0.4603	0.5159	0.5600	0.6007	0.6111	0.6599	0.6818
Clock Pin	0.0644	0.0987	0.1193	0.1269	0.1472	0.1689	0.1924	0.2322	0.2734	0.2941
SLRU	0.2609	0.3454	0.4080	0.4758	0.5292	0.5770	0.6190	0.6319	0.6139	0.7130

Trace driven simulation for dual stage caching

Finally the extension to the replacement scheme proposed with a dual stage of caching is simulated using the same set of traces.

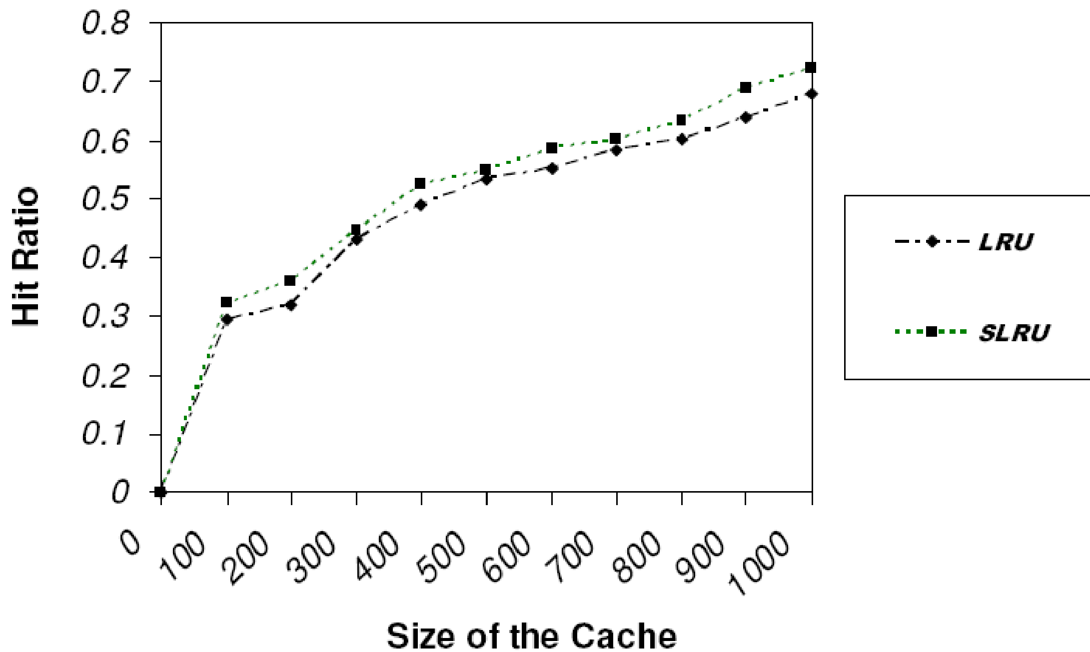


Fig. 3.3. Hit Ratio Vs Size of the Cache for LRU and SLRU with Admission Control and Dual Stage Caching

Table 3.3. Hit ratio of cache with admission & dual stage caching for various cache sizes

Size (MB)	100	200	300	400	500	600	700	800	900	1000
LRU	0.295	0.319	0.43	0.491	0.533	0.552	0.583	0.603	0.639	0.678
SLRU	0.321	0.359	0.445	0.524	0.548	0.587	0.603	0.632	0.689	0.723

As shown in Fig 3.4 and Fig 3.5, comparing the LRU and SLRU without admission control policy, with admission control policy and with admission control dual-stage cache, it can be seen that by introducing admission policy and dual-stage cache the performance of the cache can be improved significantly. As the cache size increases there is a considerable amount of improvement.

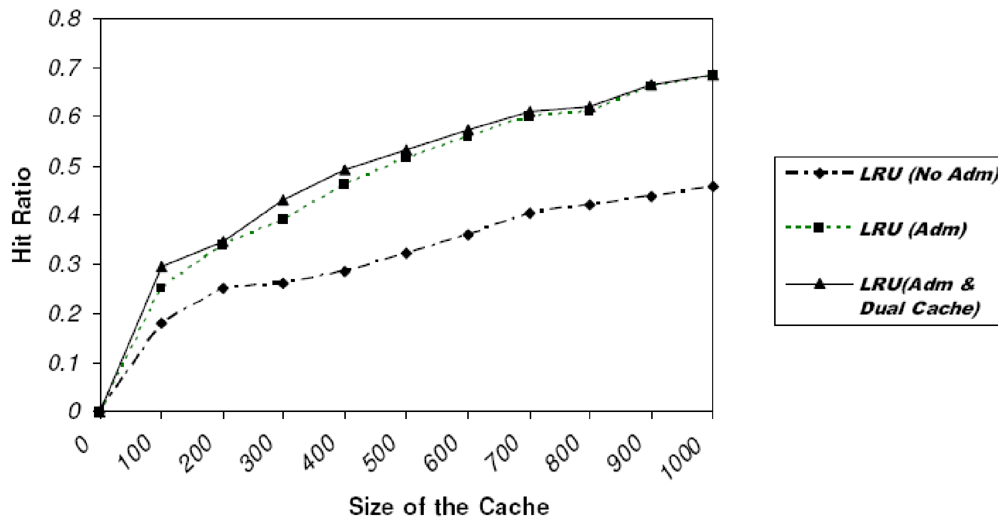


Fig 3.4 Comparison of LRU(Without Admission control), LRU(With Admission control) and LRU(With Admission control and Dual Cache)

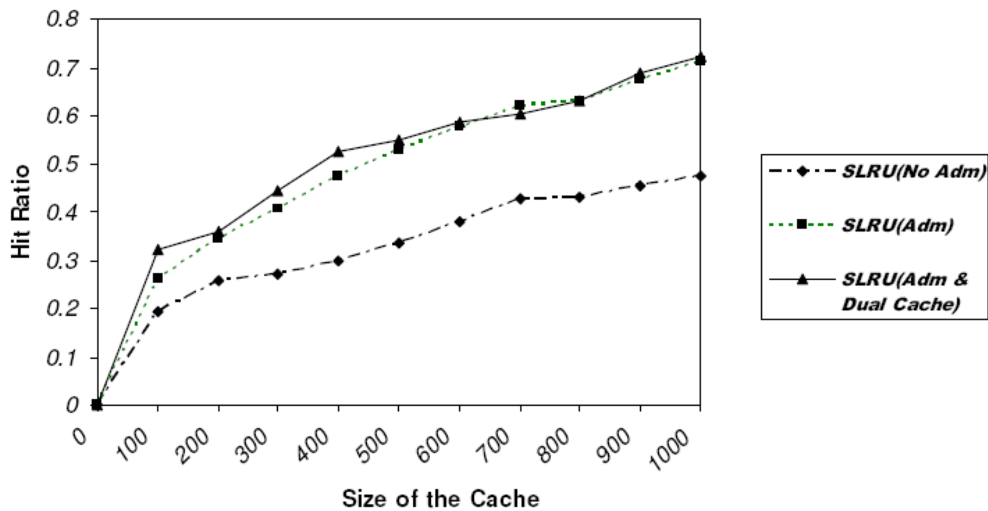


Fig 3.5 Comparison of SLRU(Without Admission control), SLRU(With Admission control) and SLRU(With Admission control and Dual Cache)

Based on these results, it can be concluded that Dual-Stage with Victim Cache policy is a practical and viable caching algorithm. It has better hit ratio performance and is also robust to varying workload characteristics.

3.2.2. RANDOMIZED HISTORY BASED CACHING AND REPLACEMENT

As discussed earlier the LRU and its variants have worked well for processor caches [Silberschatz 2001]. It has been shown by Cao [Cao 1997] that the eviction rule "replace the least recently used document" performs poorly in web-caches, instead using a combination of several criteria, such as recency, frequency, the size, and the cost of fetching a document, leads to sizable improvement in hit rate and latency reduction. However, in order to implement these novel schemes, one needs to maintain complex data structures. Most of them require a priority queue in order to reduce the time to find a replacement from O(k) to O(log k), where k is the number of documents in the cache. Further these data structures need to be constantly updated (ie. even when there is no eviction), although they are solely used for eviction. A simple Random Replacement (RR) algorithm evicts a document drawn at random from the cache [Motwani 1995]. This algorithm does not need any data structure to support the eviction decisions. However, as might be expected, the RR algorithm does not perform well. So recently a scheme was proposed in [Psounis 2002] to combine the benefits of both the utility function based schemes like LRU, LFU and SLRU with RR schemes.

To better the performance of the above scheme we combine the history based scheme with the randomized utiity based schemes.

3.2.2.1 Randomized Algorithm

Here we briefly describe the Randomized Web cache replacement scheme [Psounis 2002]. Consider a scheme which draws N documents from the cache and evicts the least useful document in the sample, where the usefulness of a document is defined by a utility function. After replacing the least recently used of N samples, the identity of the next M (usually less than N) least useful samples is retained in memory. At the next eviction time, N-M samples are drawn from the cache and the least recently used of these N-M and M previously retained samples are combined to form N samples. The identities of the M least useful of these samples are retained in memory and so on. Intuitively, the performance of the algorithm that works on few randomly chosen samples depends on

the quality of the samples. Therefore, by deliberately tilting the distribution of the samples towards the good side, considerable improvement in performance can be achieved. It is found that the improvement in performance can be exponential for small values of M. As the value of M increases, degradation in the performance can be observed because bad samples are being retained and not enough new samples are being chosen.

The Randomized Algorithm

```
If (eviction) {
If (first_iteration) {
   Sample (N);
   Evict_least_useful;
   Keep_least_useful (M);
}
Else  {
   Sample (N-M);
   Evict_least_useful;
   Keep_least_useful (M);
      }
}
```

3.2.2.2 Randomized Least recently Used (RLRU)

The randomized LRU approximates the deterministic LRU. More the samples better will be the approximation towards LRU by RLRU. In this scheme the data structure is not maintained for the eviction purposes. The parameter used by the utility function i.e. the time since the pages last used is stored in order to be available when the document is chosen as a sample. Moreover the whole cache need not be sorted according to the last accessed time of the document. The number of updations that take place in this case are less when compared to deterministic LRU.

```
If (eviction) {
If (first_iteration) {
   Sample (N);
   Arrange these N samples in the increasing order of last accessed time
   Evict the first sample of these sorted samples
   Keep the next M samples for the succeeding iteration
}
```

```
Else  {
   Sample (N-M);
   Sort these N-M new samples and M previously retained in the increasing order of last
                                                                                    accessed time
   Evict the first sample of these sorted samples
   Keep the next M samples for succeeding iteration
      }
}
```

3.2.2.3 Randomized Size Adjusted Least recently Used (RSLRU)

The randomized SLRU approximates SLRU. Just like RLRU in this scheme the data structure is not maintained for the eviction purposes. The parameter used by the utility function i.e. the product of size and dynamic count is stored in order to be available when the document is chosen as a sample. Moreover the whole cache need not be sorted according to the product of size and dynamic count of the document. The number of updations that take place in this case are less when compared to SLRU.

```
If (eviction) {
If (first_iteration) {
   Sample (N);
   Arrange these N samples in the increasing order product of size and dynamic count.
   Evict the first sample of these sorted samples
   Keep the next M samples for the succeeding iteration
}
Else  {
   Sample (N-M);
   Sort these N-M new samples and M previously retained in the increasing order of
product of size and dynamic count.
   Evict the first sample of these sorted samples
   Keep the next M samples for succeeding iteration
      }
}
```

3.2.2.4 History based Cache Replacement Algorithm

A typical cache replacement approach involves updating the cache content under a certain criterion or over a considered time period. One of the disadvantages of the LRU is that it only considers the time of the last reference and it has no indication of the number of references for a certain Web object. In the previous section we presented a random

cache replacement policy, which overcomes the difficulty of maintaining the data structure. Here we introduce a scheme to support a "history" of the number of references to a specific Web object.

Definition of History function

Suppose that $r_1, r_2, \ldots, r_n$ are the requests for cached Web objects as logged at the time units $t_1, t_2, \ldots, t_n$ respectively. A history function for a specific cached object x is defined as follows:

hist(x, h) = t_i if there are exactly h - 1 references between times t_i and t_n, and
= 0 otherwise.

The above function hist(x, h) is a time metric and defines the time of the past h^{th} reference to a specific cached object x. Furthermore, the time t_i identifies the first of the last h references to x. To analyze the performance of History based Randomized algorithm, a combination of random replacement with the history based algorithm has been implemented and the results have been analyzed.

3.2.2.5 History based RLRU (HRLRU)

Randomly chosen documents are arranged on the basis of the parameter, history. The object with the least history value is evicted first. In case of a tie between the history value the LRU scheme is used to choose the object for eviction. The history of an object is defined as the first of the last h^{th} reference to that object.

3.2.2.6 History based RSLRU (HRSLRU)

Implementation is the same as HRLRU except that if two or more objects have the same history value then the tie is broken using SLRU scheme.

HRLRU and HRSLRU algorithm will replace the cached object with minimum history value from the randomly selected sample. If two or more objects have the same history value then the tie is broken by the LRU algorithm or SLRU algorithm respectively.

```
If (eviction) {
If (first_iteration) {
   Sample (N);
   Arrange these N samples in the increasing order of the history value.
   Evict the first sample of these sorted samples
   Keep the next M samples for the succeeding iteration
}
Else  {
   Sample (N-M);
   Sort these N-M new samples and M previously retained in the increasing order of the
history value.
   Evict the first sample of these sorted samples
   Keep the next M samples for succeeding iteration
     }
}
```

3.2.2.7 Discussion of Results

The simulation has been carried out for varying sizes of cache capacity and also for different values of N and M. As shown in Fig 3.6, it is observed that the behaviour of LRU for N=8, M=2 and N=30, M=5 are identical. Thus the values of N and M do not affect the LRU curve significantly. However the RLRU curves are affected. The average behavior of the RLRU curve for N=8, M=2 is better than N=30, M=5.

From Fig 3.7, it is observed that the behavior of SLRU for N=8, M=2 and N=30, M=5 are identical. Thus the values of N and M do not affect the SLRU curve significantly. However the RSLRU curves are affected. The average behavior of the RSLRU curve for N=30, M=5 is better than N=8, M=2. For case 1 the object for eviction is chosen from 25 objects and for case 2 from 6 objects. So the probability of retention of the object with smaller product of size and the dynamic count is more and hence the hit ratio is more for the case 1.

In Fig 3.8, we can observe the behavior of Randomized LRU and Randomized SLRU for 2 cases mentioned earlier. It is observed that for case 1 and case 2 that RSLRU performs better that RLRU. Now comparing the behavior of History-Based RLRU, Randomized

LRU and LRU, as shown in Fig 3.9, we observe that HRLRU performs better than RLRU which in turn performs better than LRU. Similarly comparing the behavior of History-Based Randomized SLRU, Randomized SLRU and SLRU, as shown in Fig 3.11, we observe that RSLRU performs better than HRSLRU which in turn performs better than SLRU. Finally we compare the History based implementation of RLRU and RSLRU, as shown in Fig 3.12 and Fig 3.13 for case 1 and case 2, we observe that HRSLRU performs better for smaller cache sizes in both the cases. After a certain cache size HRLRU performs better.

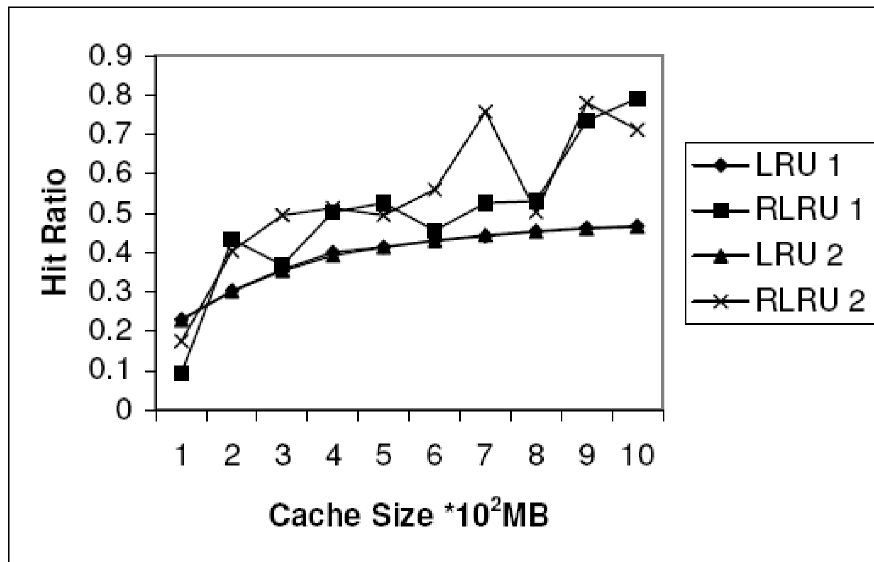


Fig 3.6 Cache Size vs Hit ratio for LRU & RLRU 1) N=30, M=5 2) N=8, M=2

Table 3.4 LRU & RLRU Hit ratio for N=30 & 8 and M= 5 & 2

Size(MB)	LRU1	RLRU1	LRU2	RLRU2
100	0.2307	0.0946	0.2294	0.1748
200	0.3032	0.4329	0.3022	0.404
300	0.3565	0.3689	0.3542	0.4943
400	0.4001	0.5021	0.3929	0.5119
500	0.4131	0.5249	0.4134	0.4953
600	0.4293	0.4556	0.43	0.5596
700	0.4407	0.5255	0.4439	0.7576
800	0.4534	0.5301	0.4527	0.5031
900	0.4628	0.7345	0.4612	0.7793
1000	0.4673	0.792	0.466	0.7117

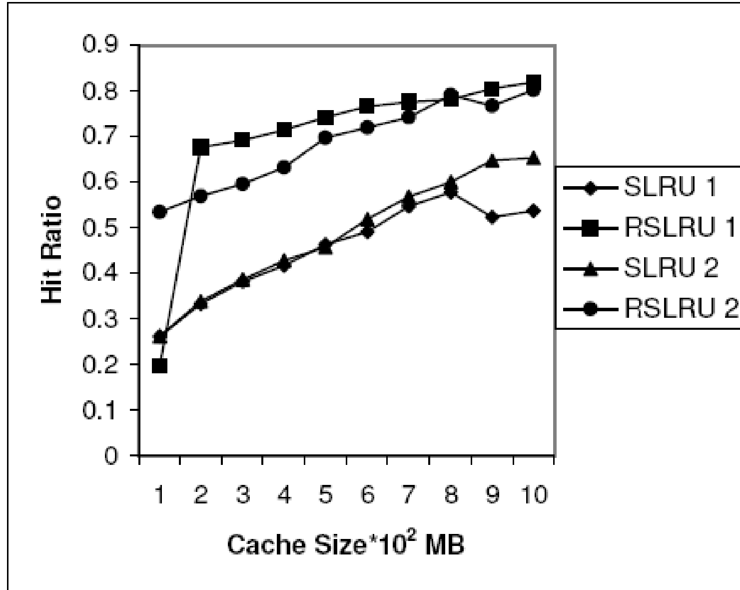


Fig 3.7 Cache size vs Hit ratio for SLRU & RSLRU 1) N=30, M=5 2) N=8, M=2

Table 3.5 SLRU & RSLRU Hit ratio for N=30 & 8 and M= 5 & 2

Size(MB)	SLRU1	RSLRU1	SLRU2	RSLRU2
100	0.263	0.1961	0.2617	0.5342
200	0.3341	0.6772	0.3393	0.5689
300	0.3817	0.6924	0.3865	0.5962
400	0.4168	0.714	0.4285	0.6326
500	0.4632	0.7426	0.4571	0.698
600	0.4905	0.7665	0.5186	0.7205
700	0.5472	0.776	0.568	0.7426
800	0.578	0.7816	0.6003	0.7912
900	0.5238	0.8055	0.6478	0.7673
1000	0.5368	0.8185	0.6526	0.8029

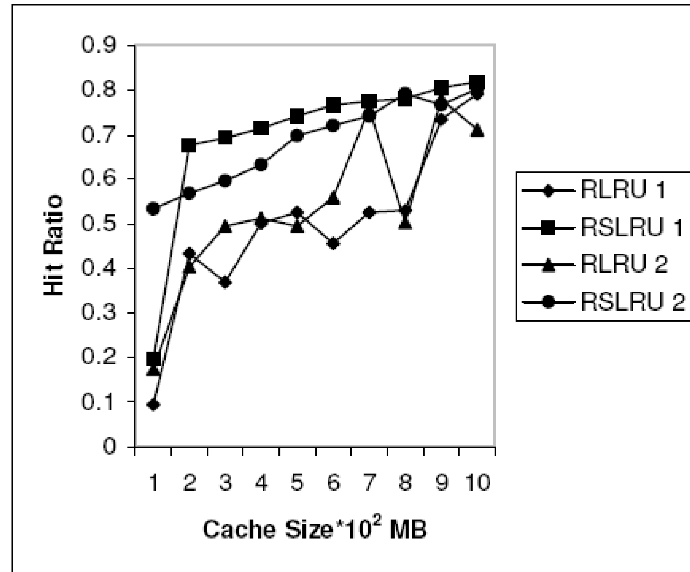


Fig 3.8 Cache size vs. Hit ratio for RLRU & RSLRU 1) N=30, M=5 2) N=8, M=2

Table 3.6 RLRU & RSLRU Hit ratio for N=30 & 8 and M= 5 & 2

Size(MB)	RLRU1	RSLRU1	RLRU2	RSLRU2
100	0.0946	0.1961	0.1748	0.5342
200	0.4329	0.6772	0.404	0.5689
300	0.3689	0.6924	0.4943	0.5962
400	0.5021	0.714	0.5119	0.6326
500	0.5249	0.7426	0.4953	0.698
600	0.4556	0.7665	0.5596	0.7205
700	0.5255	0.776	0.7576	0.7426
800	0.5301	0.7816	0.5031	0.7912
900	0.7345	0.8055	0.7793	0.7673
1000	0.792	0.8185	0.7117	0.8029

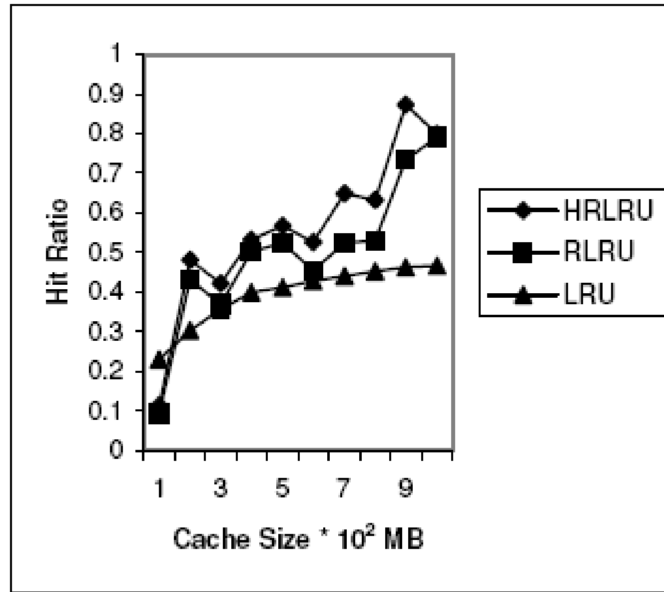


Fig 3.9 Cache size vs. Hit ratio for HRLRU, RLRU & LRU for N=30, M=5

Table 3.7 HRLRU, RLRU and LRU Hit ratio for N=30 and M= 5

Size(MB)	HRLRU	RLRU	LRU
100	0.1134	0.0946	0.2207
200	0.4807	0.4329	0.3032
300	0.4225	0.3689	0.3565
400	0.5323	0.5021	0.4001
500	0.5665	0.5249	0.4131
600	0.5265	0.4556	0.4293
700	0.6497	0.5255	0.4407
800	0.6328	0.5301	0.4534
900	0.8729	0.7345	0.4628
1000	0.8005	0.792	0.4673

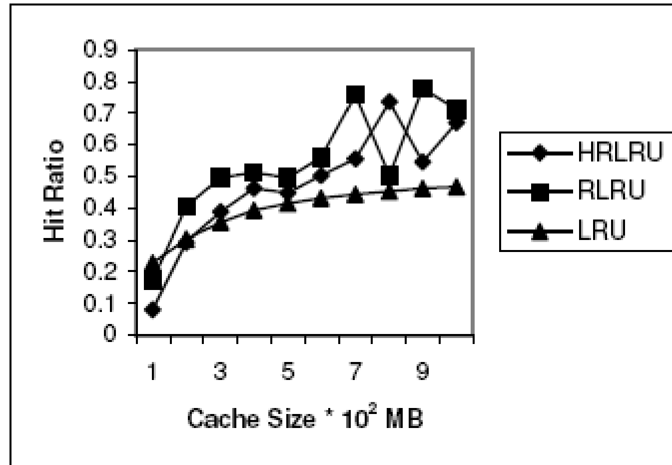


Fig 3.10 Cache size vs. Hit ratio for HRLRU, RLRU & LRU for N=8, M=2

Table 3.8 HRLRU, RLRU and LRU Hit ratio for N=8 and M= 2

Size(MB)	HRLRU	RLRU	LRU
100	0.0819	0.1748	0.2294
200	0.2918	0.404	0.3022
300	0.3874	0.4943	0.3542
400	0.4615	0.5119	0.3929
500	0.4465	0.4953	0.4134
600	0.5031	0.5596	0.43
700	0.5544	0.7576	0.4439
800	0.7348	0.5031	0.4527
900	0.5447	0.7793	0.4612
1000	0.6692	0.7117	0.466

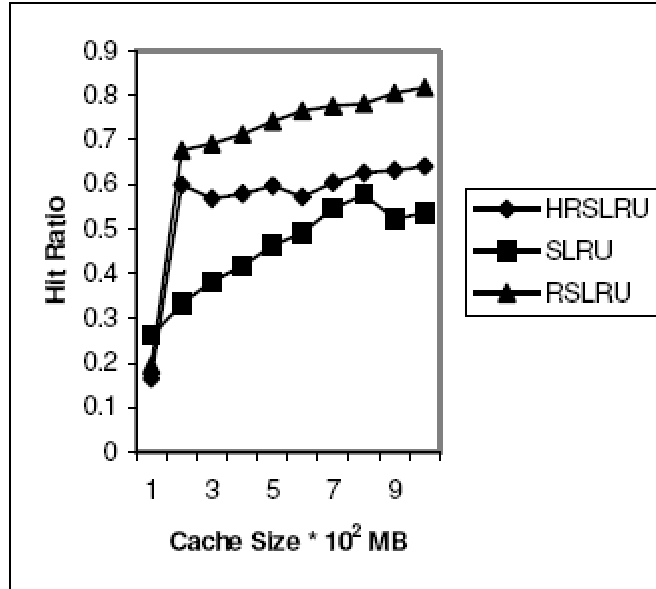


Fig 3.11 Cache size vs Hit ratio for HRSLRU, RSLRU & SLRU for N=30, M=5

Table 3.9 HRSLRU, SLRU and RSLRU Hit ratio for N=30 and M= 5

Size(MB)	HRSLRU	SLRU	RSLRU
100	0.1664	0.263	0.1961
200	0.6001	0.3341	0.6772
300	0.5693	0.3817	0.6924
400	0.5802	0.4168	0.714
500	0.5975	0.4632	0.7426
600	0.5724	0.4905	0.7665
700	0.6053	0.5472	0.776
800	0.6261	0.578	0.7816
900	0.6317	0.5238	0.8055
1000	0.6412	0.5368	0.8185

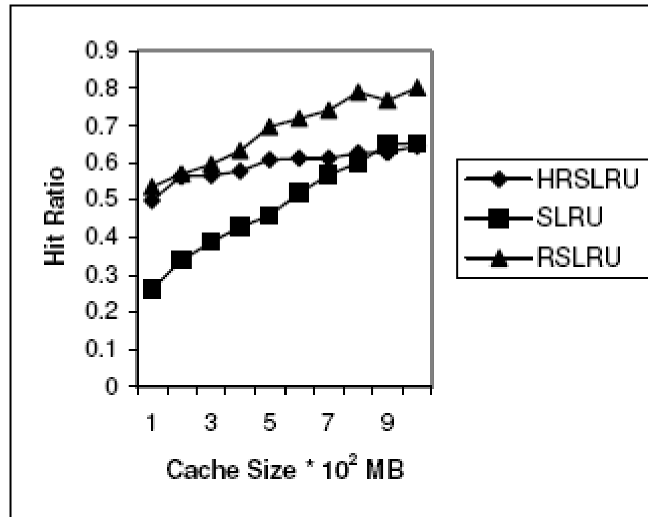


Fig 3.12 Cache size vs Hit ratio for HRSLRU, SLRU & RSLRU for N=8, M=2

Table 3.10 HRSLRU, SLRU and RSLRU Hit ratio for N=8 and M= 2

Size(MB)	HRSLRU	SLRU	RSLRU
100	0.4983	0.2617	0.5342
200	0.5628	0.3393	0.5689
300	0.5663	0.3865	0.5962
400	0.578	0.4285	0.6326
500	0.6075	0.4571	0.698
600	0.6122	0.5186	0.7205
700	0.6122	0.568	0.7426
800	0.6269	0.6003	0.7912
900	0.6274	0.6478	0.7673
1000	0.6447	0.6526	0.8029

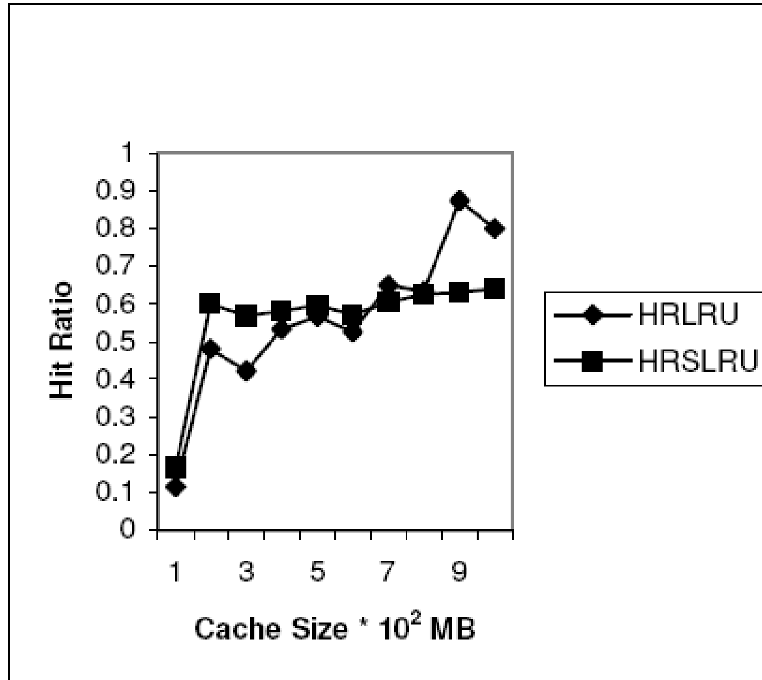


Fig 3.13 Cache size vs. Hit ratio for HRSLRU & HRSLRU for N=30, M=5

Table 3.11 HRLRU and HRSLRU Hit ratio for N=30, M=5

Size(MB)	HRLRU	HRSLRU
100	0.1134	0.1664
200	0.4807	0.6001
300	0.4225	0.5693
400	0.5323	0.5802
500	0.5665	0.5975
600	0.5265	0.5724
700	0.6497	0.6053
800	0.6328	0.6261
900	0.8729	0.6317
1000	0.8005	0.6412

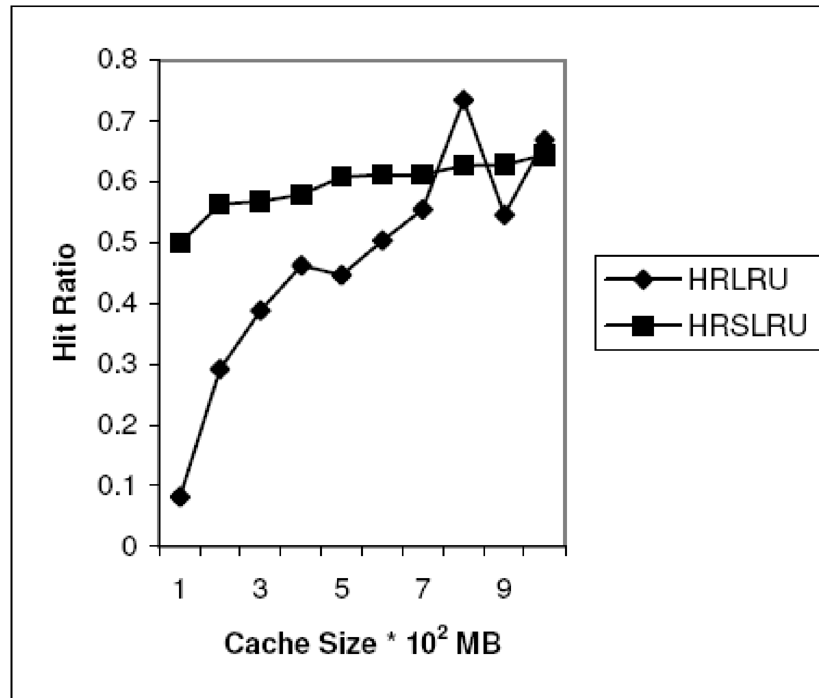


Fig 3.14 Cache size vs Hit ratio for HRLRU & HRSLRU for N=8, M=2

Table 3.12 HRLRU and HRSLRU Hit ratio for N=8, M=2

Size(MB)	HRLRU	HRSLRU
100	0.0819	0.4983
200	0.2918	0.5628
300	0.3874	0.5663
400	0.4615	0.578
500	0.4465	0.6075
600	0.5031	0.6122
700	0.5544	0.6122
800	0.7348	0.6269
900	0.5447	0.6274
1000	0.6692	0.6447

Being randomized, the performance of this algorithm depends crucially on the quality of the samples it obtains. Further the utility function in these algorithms considers only the recency and it does not consider the number of accesses of the object in the past, which is also very important for the popularity of the web object. To consider this we have adopted a History based replacement algorithm.

It has been observed that randomized replacement policy with LRU or SLRU performs better than only LRU or SLRU. A size based replacement is more efficient than a Least recently used policy. This is confirmed by our results which prove that RSLRU has a higher hit ratio than RLRU. On using random replacement with history based policy we observe that HRLRU performs better than RLRU but this does not hold for the size based replacement. In the case of HRSLRU and HRLRU it is observed that for smaller cache sizes the former performs better. HRSLRU is almost stable with increasing cache sizes.

3.2.3. MODIFIED CACHE-ON-DEMAND PROTOCOL

Cache-on-Demand(CoD) is a new protocol for web caching, which allows a web cache to allocate its local resources (e. g. disk space) upon external requests from either content provider or web users themselves, and thus provides Quality of Service (QoS) and service level agreement (SLA) in delivering content [Ahuja 2002]. The advantage to the content provider is QoS guarantees like fresh content being available to a web user from a CoD enabled web cache. The model also provides for a new value added service that can be offered by network operators and ISPs (usually the cache owners), who can build revenue in return for providing caching resources to the requesters. And above all, it improves the user experience.

The CoD concept can also be applied between the cache and the web clients. This protocol also supports strong consistency by giving complete content management control to the content provider. The CoD client can reserve resources for a specified duration of time and push its content to the cache. It can explicitly update the cached content in order to maintain strong consistency between the original and the cached copies of the content. It can request the CoD enabled cache to invalidate the content and free up the reserved resources. It can also request for content specific access log information.

3.2.3.1 Service Discovery

Before the client can use the CoD services of a cache, it has to be aware of which are the proxy caches that are CoD enabled. This is done by harnessing the ease with which HTTP allows new extension headers to be added. When forwarding a request/reply, a CoD enabled cache adds new HTTP headers for which it sends information like its hostname or IP address as well as the port number on which it listens for CoD requests. When a client receives these extension headers, it may simply ignore them if it is not configured to interpret them. If it is, on the other hand, enhanced to be a CoD client, it extracts the values of these headers and then the two entities can communicate using CoD protocol. This initial communication mechanism is shown in Fig 3.15, where the origin web server is capable of using the CoD protocol, but the browser is not.

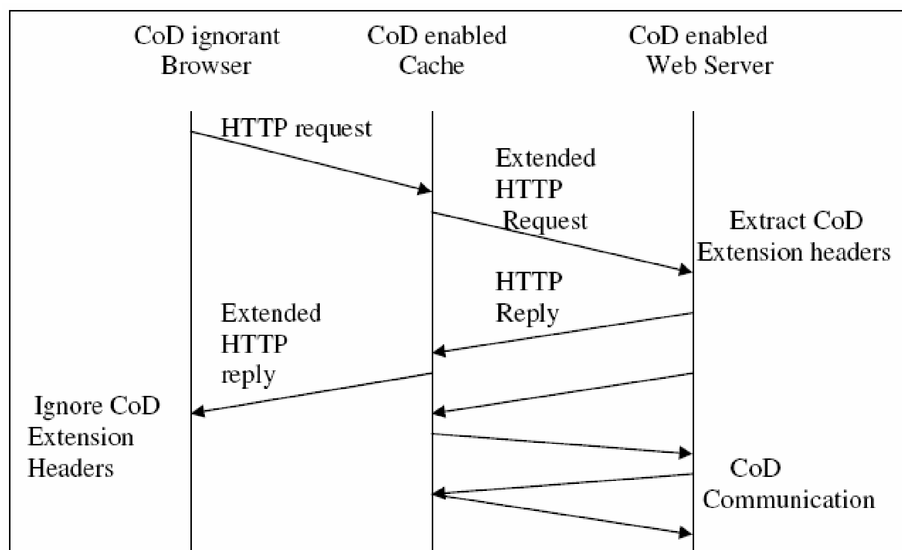


Fig 3.15: Cache on Demand Service Discovery

3.2.3.2 Protocol Messages

Once a client wiling to use the CoD services has determined the location of a CoD enabled cache, it can use the following types of messages to communicate with the cache: RESERVE, UPDATE, RELEASE, DELETE and LOG. The format of these CoD messages is shown in Fig. 3.16. These messages are sent by the requestor to the CoD

enabled cache for a request to reserve resources, update content, free up the reserved resources, delete the stored content and obtain an access log respectively.

RESERVE	UPDATE	RELEASE	DELETE	LOG
RESVDISK<diskspace> [BEGIN<time>]END<time> [<URL1>…<URLn>]	UPDT<request_id> [<URL1>…<URLn>]	RELS<request_id> [<URL1>…<URLn>]	DEL<request_id> [<URL1>…<URLn>]	LOG[<request_id] [<URL1>…<URLn>]

Fig. 3.16: Cache on Demand Message Formats

3.2.3.3 Request Handling

The CoD server continuously waits for requests from its clients. A CoD client first authenticates with the CoD server and then submits a CoD request. When the CoD server receives a request from a client, it reads in the request message and passes it to determine the type of the request and also checks the validity of the request. The way a request is handled further depends on the type of the request.

RESV

A CoD client uses RESV message to reserve disk space on the cache for a specific duration of time. The client can provide a list of URLs to be fetched along with RESV requests or it can decide to send a URL list through an UPDT message at a later time. When the CoD server receives the RESV request, it reads the time at which the client would like the reservation to start (*begin_time*), the time duration for which the resources have to be reserved and the amount of resources requested. The request is run through an admission control algorithm, discussed later in this chapter, to determine whether or not it should be accepted. If the request is rejected, a REJECT message is sent back to the requestor. If accepted, a random *request_id* is generated and sent to the requestor. The client can use this id number with other CoD commands to take further actions on the reservation. The next step is to determine when to fetch the content. If the *begin_time* for the request is the current time or a past time, content is fetched immediately. If it is a time in the future, a timer is set to indicate when the content should be fetched. When ready to fetch, the URLs listed in the RESERVE message are read one by one and content is

fetched from the origin server. This involves creating an HTTP request for each URL and forwarding it to the origin server. When content is fetched, the standard HTTP cachability rules are used to check whether or not the object can be cached. Another check is performed to make sure that the object does not exceed the disk quota reserved for that particular CoD client. If either of these checks implies that the object should not be stored in the cache, then an error is logged in the access log file. A RESV request without any URL request is also allowed. In this case, it is the CoD client's responsibility to send a list of URLs in an UPDT message.

UPDT

Web caches traditionally maintain a weak consistency between the original content and its cached copies. Consistency checks are performed only when a user agent forces to do so by sending an IMS (If Modified Since) request. Such a model does not guarantee the freshness of cached content. But CoD provides functionality to support strong content consistency. In this case, the content provider, as a CoD client has strong control over what content is cached at the CoD caches. If content stored in a CoD cache is modified at the origin server, the origin server can either send an updated copy of the content or send invalidation messages to CoD server to discard the CoD content. To update the content stored in the cache an UPDATE request can be sent to the CoD enabled cache. An UPDT message must always be accompanied with the id number of the initial reservation that needs to be updated. It may also include a list of URLs that the CoD client wants the cache to refresh. If no URL list is given, all the URLs in the original request are fetched again. The UPDT message can also be used in another scenario. As mentioned above, a RESV request might or might not have a URL list included with it. If no URLs are provided with RESV then the URLs can be specified with an UPDT request. Depending on the *begin_time* of the RESV request, the content will either be fetched immediately or in the future.

RELS

It is guaranteed that the reserved content will be stored at the web cache at least for the time period requested in the RESERVE request. After the expiration of the agreement,

the status of that content changes from strongly guaranteed to weakly guaranteed. This implies that the content may still be reserved to a user from the cache as long as it is fresh, but if another client requests for resources and cache does not have enough resources available, then this content will be replaced as per the cache replacement algorithm. However a CoD client can send a RELS request to free up some/all of the used disk space from within its quota before the agreement expires, without changing the original amount of disk space reserved, so that it can accomodate new content in the cache. This helps it to change the content stored in the cache without having to first delete the existing reservation and then send a new RESV request. If any URLs are included in the RELS request, only those URLs are released. If no URLs are specified, then all of the content corresponding to that *request_id* is released. The content will not be actually deleted from the cache immediately, but will become weakly guaranteed content. The initial amount of resources (disk space) reserved remains unaffected.

DEL

The DEL request is used when a CoD client wants to actually release partly/ wholly the resources that it had reserved. By deleting content with the DEL command, it can ensure that the content will not be available from the cache to other clients. Also, by deleting the resources before the reservation agreement expires, the CoD client can save itself some billing charges. Like the RELEASE message, if URLs are specified, only those URLs are deleted. If no URLs are specified, all the URLs are associated with that *request_id* are deleted from the cache. This will ensure that the content is not even available as best-effort content from the cache. In this case, the client also loses the corresponding reserved disk space. A URL released with the RELS command may still be served as a HIT to web users from the cache, but a URL deleted with the DEL command will be flushed out from the cache physically and will result in a MISS if a user requests it.

LOG

The access log file on a CoD enabled proxy server can have log messages in three different formats. The first format is used when a web user accesses an object through the cache. The other two log messages can be logged as a result of fetching content according

to a CoD agreement. A CoD client can request the access logs specific to its content by sending a LOG request to the cache. The request may include either a request id number, or a list of URLs, or both of them. This flexibility is provided to ensure that access logs can be retrieved even after the reservation has expired, in which case the *request id* would be invalid and the URL list could be used to retrieve the log.

3.2.3.4 Admission Control

A Cache on Demand server uses the admission control algorithm to determine if the requested resources can be allocated to a client. Current implementations allow clients only to request disk space on the CoD caches, so the admission control algorithm is used to ascertain that the total amount of disk space reserved by various clients is always less that or equal to the total hard disk capacity available on the server.

If we express this mathematically, we have,

$$req(t) + rsvd(t) \leq D$$

where, req(t) = disk space being requested at time t

rsvd(t) = disk space already reserved at time t

D = constant total disk space available

For example, considering a cache with a total 100 MB disk space available for its clients. When no reservations have been made, the system can be represented by the graph in Fig.3.17, where the dotted line represents the total disk space available. If suppose the first reservation is made for s=50 MB, starting at time t1=10 and ending at t2=30. The disk reservation is shown with a solid line in Fig.3.18. The next request is to reserve another 50 MB, starting at t1=20 and ending at t2=40. This request will be accepted because, as can be seen in Fig.3.19, the total reserved disk space at all times is less than or equal to the total available (100 MB). If the CoD server receives a third request to reserve 20 MB, starting at t1=25 and ending at t2=50, then this request will be rejected because in this case, the step function representing the reserved disk space crosses over the maximum limit as shown in Fig 3.20.

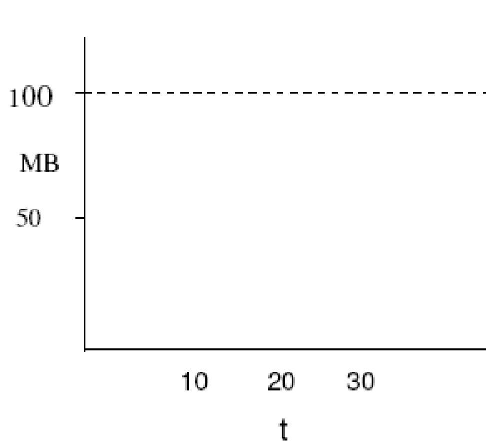


Fig.3.17 Maximum disk space available

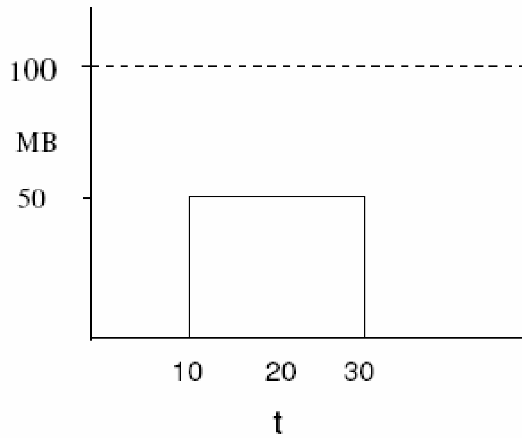


Fig.3.18 When request s=50MB, t1=10, t2=30, accepted

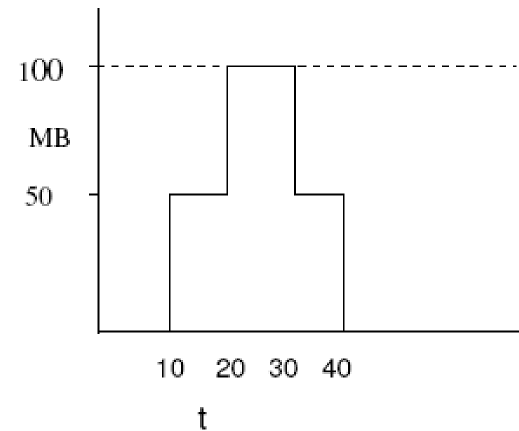


Fig.3.19 When request s=50MB, t1=20, t2=40, accepted

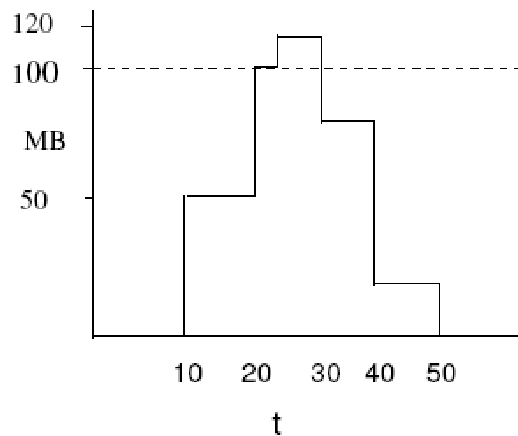


Fig.3.20 When request s=20 MB, t1=25, t2=50, rejected

This admission control algorithm can be extended for other types of resources, like CPU power and network bandwidth, as needed.

3.2.3.5 Modifications to the CoD Protocol

- When the total size of the URLs exceeds the total cache space reserved for a particular client, following policy can be adopted instead of plainly rejecting the request. The URLs are arranged in increasing order of object size and they are allocated in that order till no URL can be fit into the reserved space.

- When a URL requested through UPDATE is already present in the CoD cache, the CoD server fetches the updated copy of the URL. If the URL is a new request it is fetched from the appropriate origin server and cached for the first time. If the size of the web object obtained from the URL is greater than the free CoD cache space available for the user, then that particular request is rejected.
- When a particular web object is freed using RELS, the space occupied by it is added on to the free CoD cache space and its corresponding record in the client structure is removed.
- If there are no URLs in the RELS message, then the space occupied by all the currently cached URLs for that client is added on to the free space and the records corresponding to them are removed from the client structure.
- If URLs are specified in the DEL message, the client loses the corresponding disk space. The CoD server ensures that the content will not be available from the cache to web servers. By deleting the resources before the reservation agreement expires, the CoD client can save itself some billing charges.
- If URLs are not specified in the DEL message all the URLs that are associated with that client are deleted from the cache and the freed CoD cache space is added to the common CoD cache pool.
- If the request is a normal *http* request and not a CoD request, it is cached in the *http* portion of the cache. After every *http* request all the objects are sorted using an algorithm similar to the SLRU wherein all the cached objects are sorted in the decreasing order of object size. Once that is done the array of objects is scanned for objects having the same size. If found these objects with same size are sorted in the decreasing order of number of times they have been accessed previously. What effectively happens in the above algorithm is that very large sized and least frequently used objects are pushed to the top of the array. When a fresh *http* request requires an object to be brought into the cache and there isn't enough free space to accommodate the new object, then the objects from the top of the array are removed till there is just enough cache space for the new web object.

3.2.3.6 Discussion of Results

A proxy cache has been simulated which can accept Cache on Demand requests as well as normal caching requests. A portion of the cache (in terms of percentage of the total size of the cache) is always reserved for CoD requests. The rest of the cache is used as a normal cache. The admission and replacement for the CoD part of the cache is done through an appropriate message from the user who reserved the cache. The simulation was done for 3 admission policies. One-time caching i.e. caching the URL the first time it is requested by the user, Two-time caching i.e. caching the URL when it is requested for the second time and Three-time caching i.e. caching the URL when it is requested for the third time. The replacement policy followed is Size – Adjusted Least Recently Used (SLRU). We assume that there are N objects, and that object i has size S_i. A counter is maintained and incremented each time there is a request for an object. This counter has been named as the ***Dynamic Count(d_i)***. The steps followed in this policy are as follows:

Dynamic count(d_i) is maintained for all the objects in the cache table.

When a new object enters : $d = 1$

The object is inserted correctly in the cache table Object/s with the maximum S_i x d_i count value is thrown off to make space for the incoming object.

Object requested (irrespective of cache hit or miss) : d_i ++ for the object

Object arrangement: In the order of Size * dynamic count (S_i x d_i)

Inserted correctly in the cache table Object/s with the maximum S_i x d_i count value is thrown off to make space for the incoming object.

The log data used in the simulations was the access log of the Institutional Squid Proxy Server. The number of access logs used was 50000 logged over a period of 6 days. The simulation was done for total cache sizes of 50 MB, 100 MB and 200 MB. The percentage of the Cache on Demand cache was varied for the different sizes and the hit ratio of the normal cache was noted. The minimum hit ratio was fixed as 0.4 and based on this, the maximum percentage of the cache on demand cache was observed.

The graphs obtained for the hit ratio of various cache sizes are shown in Fig 3.21 to Fig 3.29.

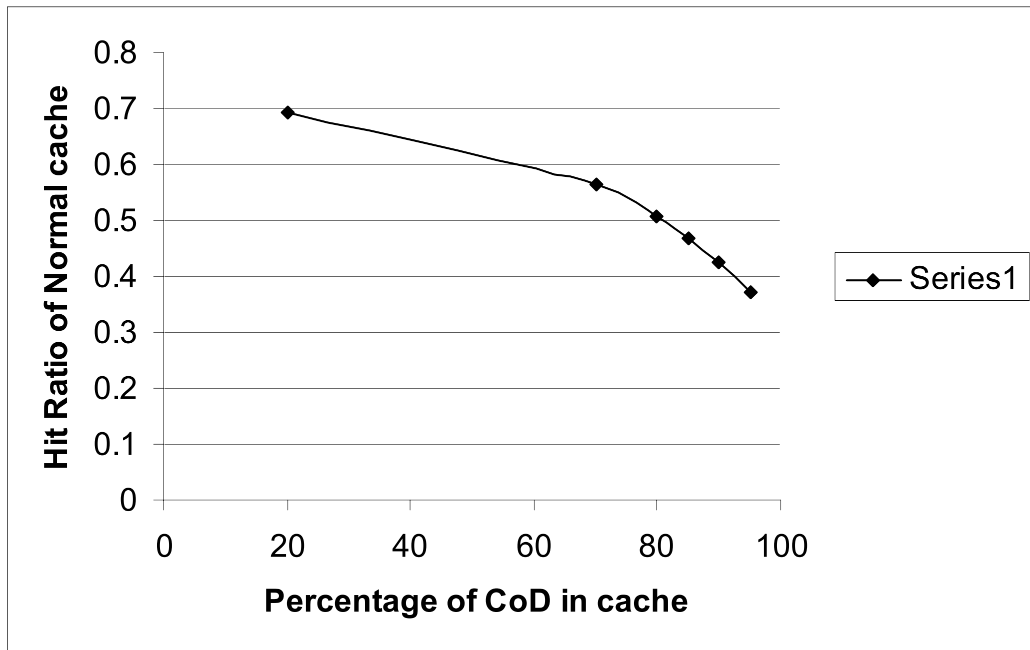


Fig 3.21 Effect of CoD on the hit ratio of the one time admission policy normal cache for cache size 50MB

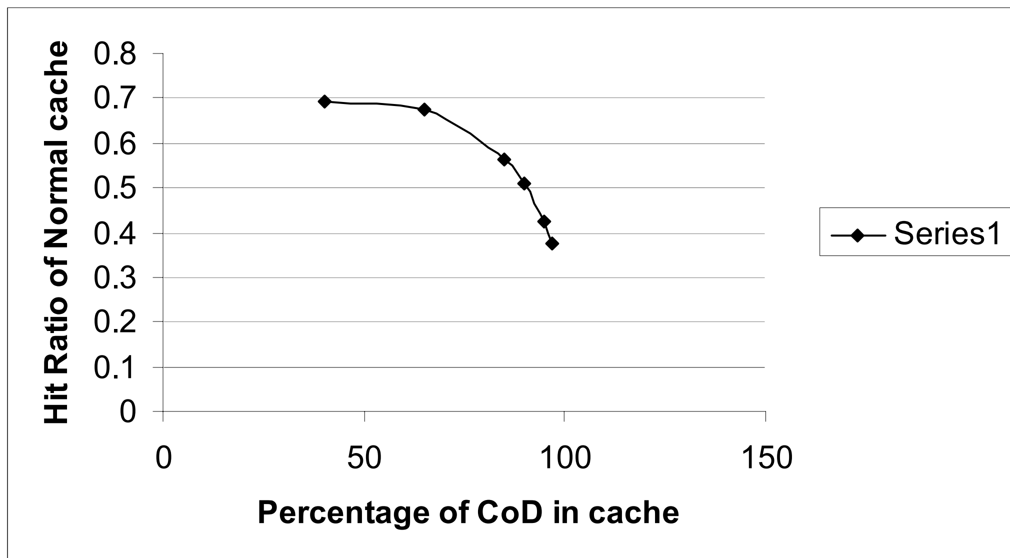


Fig 3.22 Effect of CoD on the hit ratio of the two time admission policy normal cache for cache size 50 MB

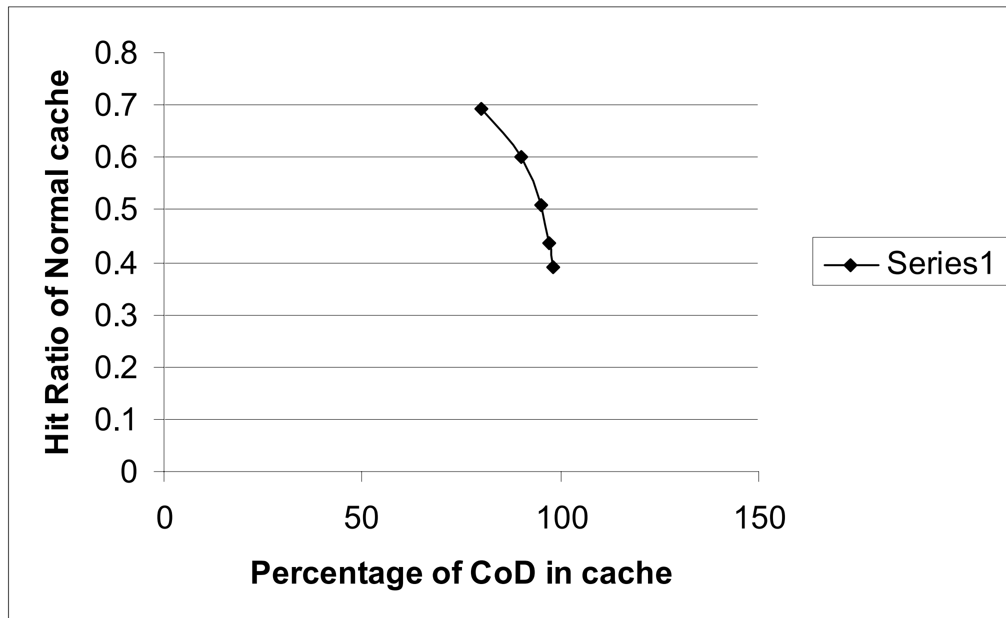


Fig 3.23 Effect of CoD on the hit ratio of the three time admission policy normal cache for cache size 50 MB

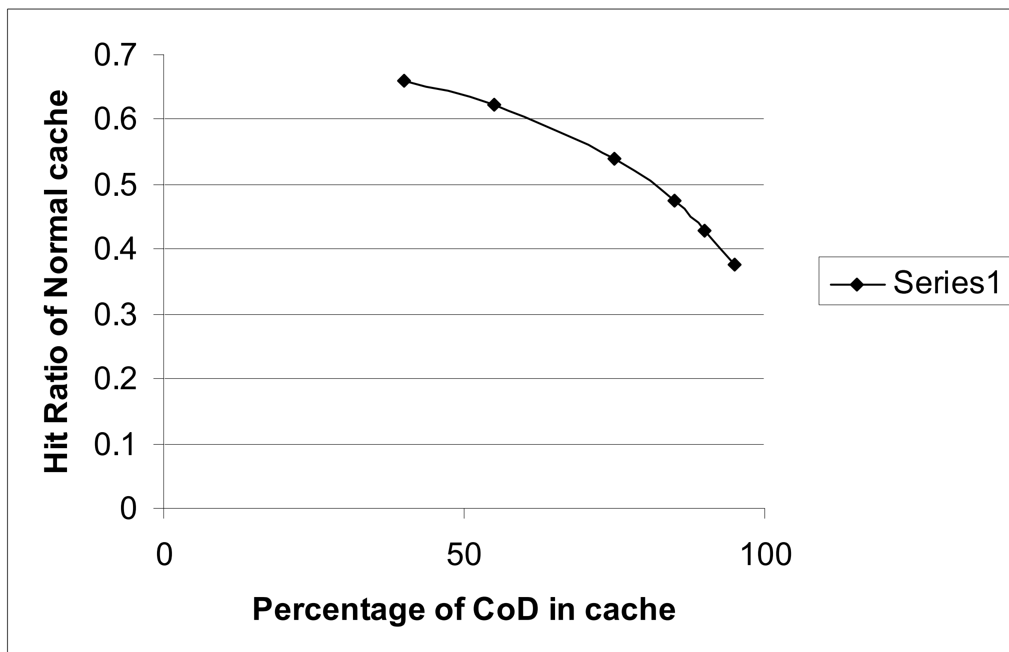


Fig 3.24 Effect of CoD on the hit ratio of the one time admission policy normal cache for cache size 100 MB

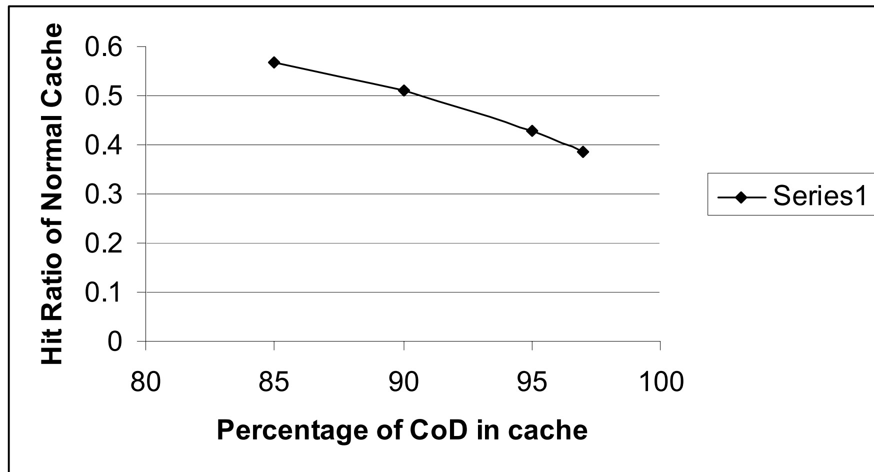


Fig 3.25 Effect of CoD on the hit ratio of the two time admission policy normal cache for cache size 100 MB

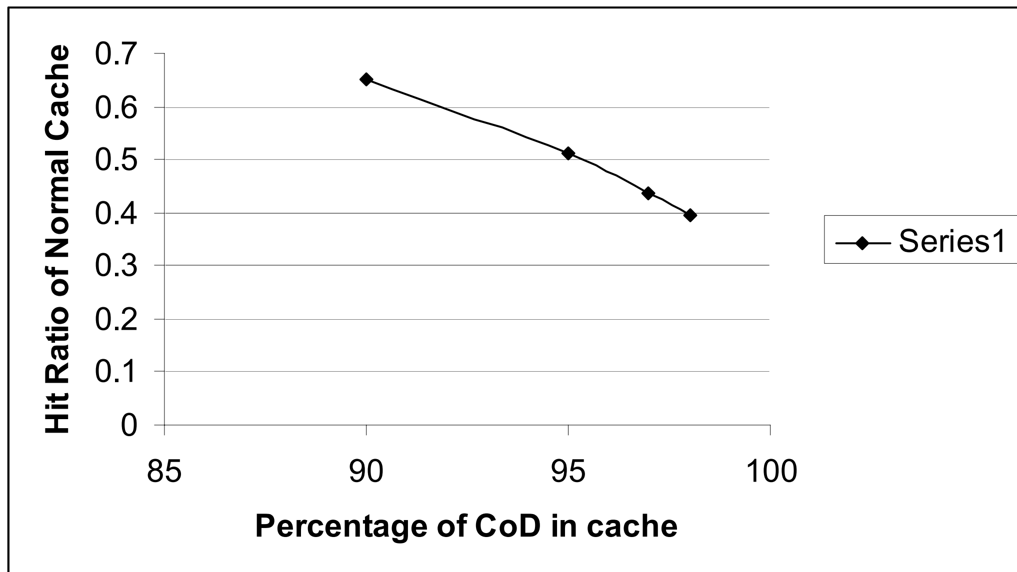


Fig 3.26 Effect of CoD on the hit ratio of the three time admission policy normal cache for cache size 100 MB

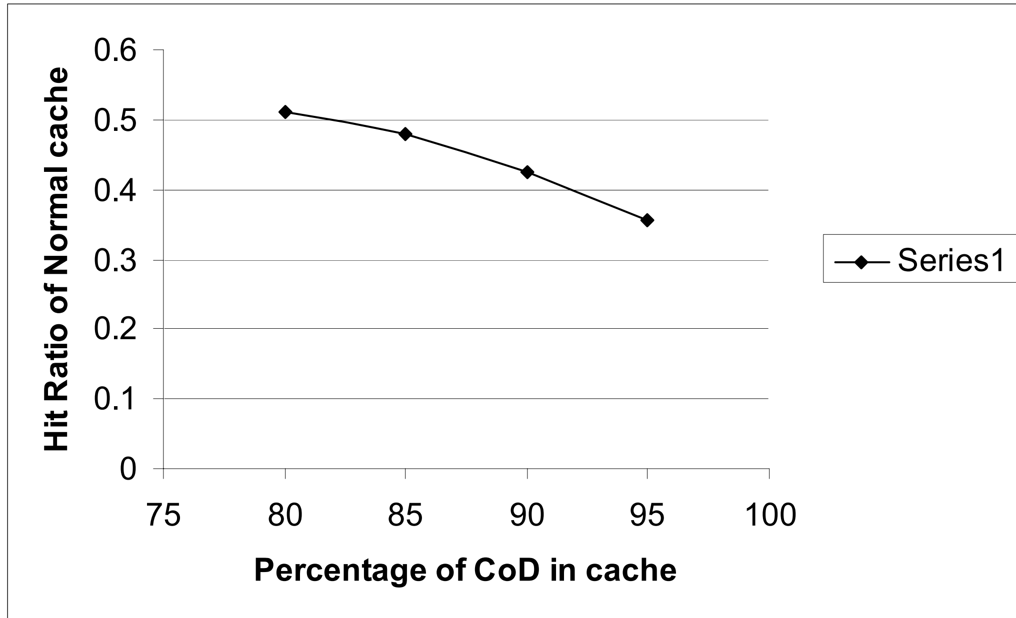


Fig 3.27 Effect of CoD on the hit ratio of the one time admission policy normal cache for cache size 200 MB

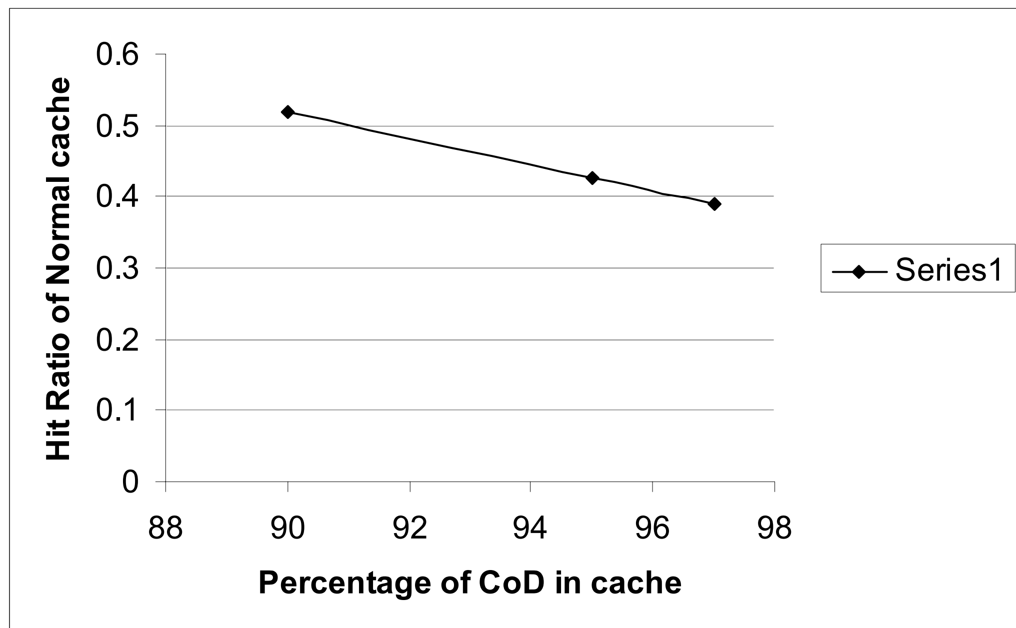


Fig 3.28 Effect of CoD on the hit ratio of the two time admission policy normal cache for cache size 200 MB

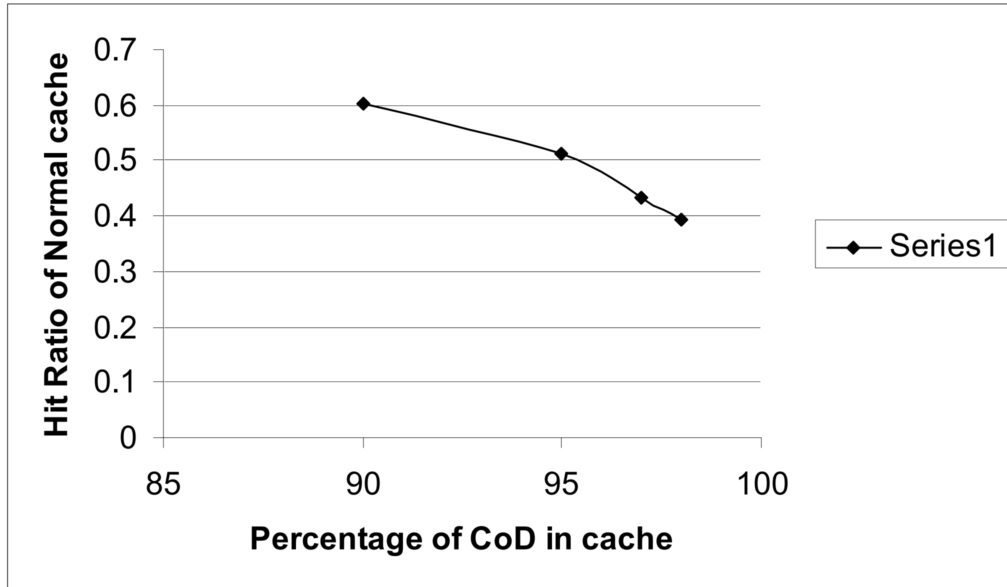


Fig 3.29 Effect of CoD on the hit ratio of the three time admission policy normal cache for cache size 200 MB

The results obtained as shown in Fig 3.21 to Fig 3.29, the effect of the presence of CoD cache as a percentage of normal cache on the hit ratio of normal cache. The table shows the amount of CoD cache size as a percentage of the normal cache that is available for the normal cache hit ratio which should remain above 40% are,

Table 3.13 Precentage of CoD cache for normal cache having 40% and above hit ratio

Size(MB)	One-time policy CoD present (%)	Two-time policy Hit ratio (%)	Three-time policy Hit ratio (%)
50	90	90	90
100	95	95	95
200	97	97	97

Thus it can be said that having a CoD Enabled Proxy Caches which can dedicate its own cache from 50% to 90% depending on number of users using this facility, does not affect the normal cache hit ratio drastically. Thus the CoD protocol enhances traditional web caches with the capability of reserving resources to store external content for a specified period of time. The major benefit of this feature is that a third party, such as a content provider or a business partner can have guaranteed content presence in the network, and also strong control on the content

delivered to web users. Furthermore, a third party can enforce strong content consistency since it can keep track of distributed content at different CoD cache locations.

Although the current implementation of Cache on Demand allows a cache to allocate only disk space to its clients, in the future this can be extended to include features like network QoS provisioning and system level resource allocation. Guaranteeing network QoS would be useful for applications like streaming media, where it would be desirable to be able to reserve network resources along the path so that the end-to-end delay can be controlled. System-level resource allocation could include leasing out memory or CPU utilization. This could facilitate services such as edge application hosting.

3.3 SUMMARY

In this chapter the Dual-Stage Victim based Web Caching method to enhance the performance of the web caches has been discussed. Next the significance of randomized algorithms and the enhancement of this scheme to include History scheme is discussed. This improves the performance of Randomized replacement policies. Also Cache on Demand protocol, which is used for improving the quality of service to the end user has been discussed. It is also shown that using CoD is not going to affect the effectiveness of the normal caching service.

CHAPTER 4

REPLACEMENT POLICIES FOR CACHING STREAMING MULTIMEDIA OBJECTS

4.1 INTRODUCTION

During recent years, the rapid increase in commercial usage of the Internet has resulted in explosive growth in demand for web-based streaming applications. As requests and delivery of streaming video and audio over the Web becomes more popular, caching of media objects on the edge of the Internet has become increasingly important. Recently, several commercial companies have announced media distribution services on the Internet using a number of proxy caches. Examples include Akamai (www.akamai.com), Digital Island (www.digisle.com), Enron Broadband Services (www.enron.net) and others. Companies that provide hardware and software caching products include Inktomi (www.inktomi.com), CacheFlow (www.cacheflow.com), Network Appliance (www.netapp.com) and others. This trend is expected to continue, and justifies the need for caching popular streams at a proxy server close to the clients.

However, techniques for caching text and image objects are not appropriate for caching media streams. The main reason is due to the large sizes of typical media objects, variable-bit rate property and real time constraints. For a large media file, such as a 2-hour video, treating the whole video as a single web object to be cached is impractical. Just storing the entire contents of long streams would exhaust the capacity of a conventional proxy cache. Hence, only the very few video objects that are 'hot' should be cached entirely. Most media objects probably should only be cached partially. Because of the high start-up overhead and isochronous requirement, a streaming media request typically is not started by a proxy server until sufficient blocks of data are cached locally. Such delayed starts can frustrate users and make customers unhappy. To overcome this problem, the beginning portions of most media objects should be cached. Hence, from the caching perspective, the beginning portion of a media stream is more important than the later portion.

From the caching perspective, multimedia-streaming objects are the most challenging ones. Other major concern while dealing with multimedia objects is the variable bit rate property. The uncompressed video that is composed of standard sized frames, become variable sized when compressed using techniques like MPEG. This makes them variable bit rate objects. Thus, while caching such objects one has to take into consideration the variable sized frames present in the videos. Multimedia objects have critical timing requirements. Any network congestion and other delays would heavily degrade the quality of service. These objects have started proliferating across the Internet recently. For this reason, user access patterns to these objects are not clearly known, like the normal web objects. This makes even the replacement of these objects, a challenging task. The video objects are mostly static in nature. Thus the cache consistency is not much of an issue in case of these objects.

One of the first works done in the field of caching techniques for multimedia objects was Resource Based Caching (RBC) algorithm [Tiwari 1998].

The RBC algorithm

(i) Characterizes each object by its resource requirement and a caching gain,

(ii) Dynamically selects the granularity of the entity to be cached that minimally uses the limited cache resource (i.e., bandwidth or space), and

(iii) If required, replaces the cached entities based on their cache resource usage and caching gain.

But this has the disadvantage of caching the objects in their entirety, which puts ever higher demand on the cache size.

Few other caching techniques for multimedia objects have been proposed in the literature. In Prefix caching [Sen 1999], the proxy stores a *prefix* consisting of the initial frames of each clip. Upon receiving a request for the stream, the proxy immediately initiates transmission to the client, while simultaneously requesting the remaining frames from the server.

Prefetching [Rejaie 1999, Rejaie 2000] is another technique wherein the objects are fetched based on prediction thus involving a lot of overhead in network bandwidth. Here, the video is prefethced and cached in its entirety.

A recent variation for the Prefetching technique is layered video format caching according to the QoS requested by the client [Jerkins 2003]. Also, the layered approach used ensures that the basic layer delivered to the client is the one cached and then the quality of the stream is improved on subsequent requests depending on the bandwidth availability. Even though it provides the benefit of caching according to the QoS requirement of user, this method stores the object in its entirety, which puts restriction on number of objects that can be stored.

Video summarization [Lee 2002] proposes to deliver a summary of the video before the delivery of the actual video. The content analysis service performs shot boundary detection, key-frame selection, and face detection and tracking. When a client requests a streaming video file, proxy system initially provides the video summary to the user. The user quickly browses these summary images to decide whether to download the video. If the user selects to download, the user can also choose which part of the video to download. This system has been designed to utilize the content analysis service that is currently applied to videos of format MPEG-1 and MPEG-2. A similar technique has to be used for other streaming formats. All this amounts to a lot of processing overhead on the proxy server and overhead of maintaining video summary for all the requests.

In Segment based caching [Wu 2001] blocks of a media stream received by a proxy server are grouped into variable-sized segments. The cache admission and replacement policies then attach different caching values to different segments, taking into account the segment distance from the start of the media. These caching policies give preferential treatments to the beginning segments. As such, users can quickly play back the media objects without much delay. This works well when the set of hot media objects changes over time.

4.2 CUT-OFF AND OPTIMAL CACHING TECHNIQUES

Multimedia objects, more specifically video objects, with their inherent properties cannot be cached in their entirety. A technique called Video Staging was proposed for caching video objects, where the video proxy would cache only a part of the video content. A video staging algorithm would decide what part of the video should be cached in the proxy. Any such algorithm would look at video object as set of video frames, each of which can be of different size.

Two of the video staging algorithms are explored here: cut-off caching algorithm (**CC**) [Zhang 2000] and optimal caching algorithm (**OC**) [Chang 2001, Chang 2002]. These are described below and their performances are analyzed through implementation of both the algorithms. Standard benchmark videos have been used for the comparison purposes.

The video staging algorithms in the study have considered cache size and available external bandwidth as two important resources, for video proxy caching. Since each frame is of different size, and should be served in the given frame period, the algorithms intend to cache only parts of those frames which cannot be completely supported by the external network bandwidth. A cutoff size must be chosen so that all the frame parts above this cutoff are cached as explained in the following part.

NOTATIONS

The CC and OC algorithms are almost similar except that OC algorithm has an added feature called prefetching by which its performance is enhanced. This section first discusses the CC algorithm which is fairly simple, followed by the more complex OC algorithm.

A cutoff size or cutoff rate is an important factor in any video staging algorithm. A cutoff size is decided depending upon the external bandwidth and some other factors. This cutoff size indicates the size that can be fetched in the given frame period from the server in the worst-case scenario. Thus, for most of the time cutoff size lies substantially lower

than the number of bits that can be supported by the external bandwidth. Any frame that is smaller in size than the cutoff need not be cached; for any frame that is larger than cutoff, the portion above the cutoff needs to be cached.

According to the CC algorithm, any video file F is composed of several frames (no layering of frames has been assumed). For each frame, some portion of it can be fetched using the available network bandwidth, which is called as cutoff size or rate and is denoted by c(i). Below are the notations that would be followed in this chapter as shown in Fig.4.1

For a given video file F

1. $F = \{ s_i \mid 0 <= i <= n \}$
2. s_i represents the i-th frame
3. *s(i)* : size of i-th frame
4. *sc(i)* : size of i-th frame that must be cached
5. *c(i)* : cutoff size for any frame i, supported by bandwidth
6. *sc(i) = s(i) - c(i)*

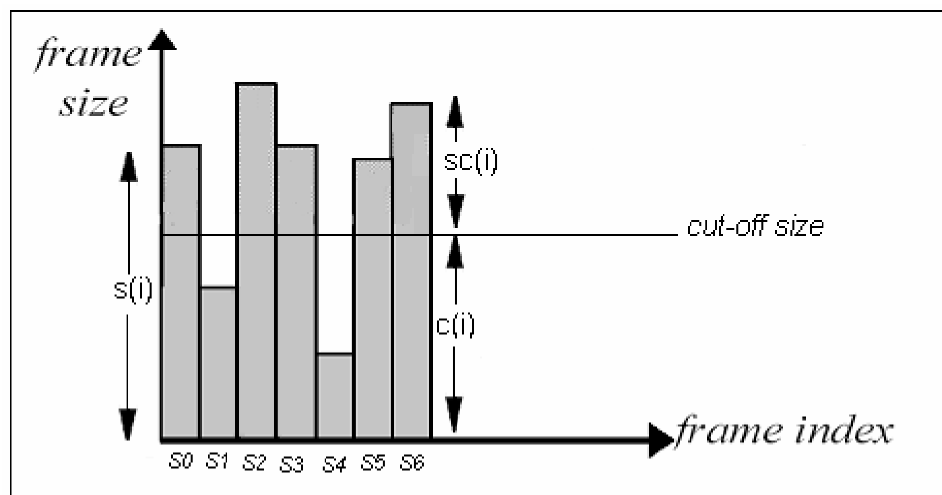


Fig.4.1: Video Frames and the notations

4.2.1 CC ALGORITHM

The CC algorithm proposes to cache only those parts of video frame, which are above the cutoff size for each frame. When a client requests a video file and there is a cache-miss at the proxy, parts of this file must be cached. A request is sent to the server for the video file and proxy starts receiving the video frames, one each per frame period. In the proxy, the CC algorithm is run for each frame obtained. The size of the frame obtained is compared with the cutoff size. If the size is less than the cutoff size, no part of this frame needs to be cached. If the frame size is larger than the cutoff size, then the bits of the frame that fall in the excess part (above the cutoff) are cached into the proxy as shown in Fig. 4.2.

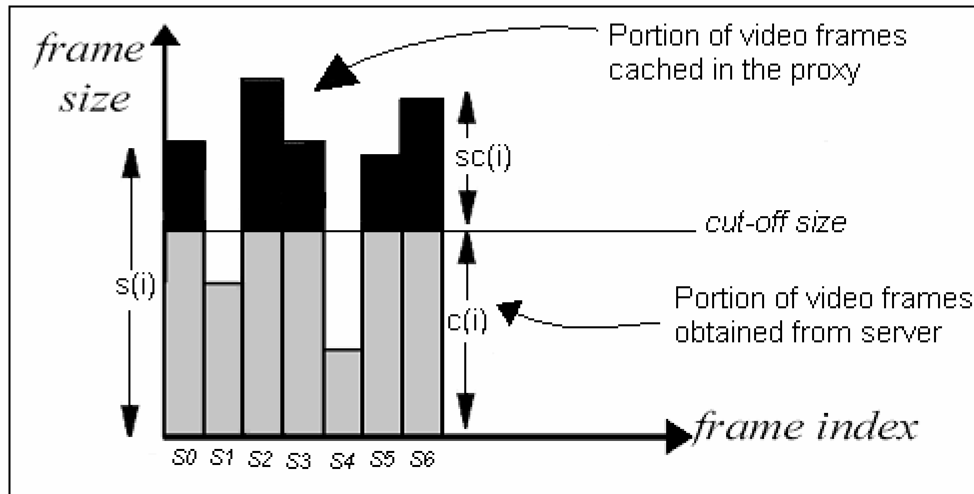


Fig.4.2: Illustration of CC Algorithm

Following the above given notations, CC algorithm can be written down as follows:

```
i=0;
repeat
    i = i+1;
    if ( s(i) < c(i) )
    {
        /* The frame size is less than cutoff */

        sc(i) = 0;
        /* Nothing to cache */
    }
```

```
    else
    {
        /* The frame size is larger than
         cutoff */
        sc(i) = s(i) - c(i ) ;
        /* cache sc(i) bits of frame I
       into the proxy */
     }
until (i>(n-1));
```

Once the file is cached, the proxy would serve any other request for the same file in the following manner. Any frame, which is less than cutoff size, would be completely obtained from the server. For any frame which is greater in size than cutoff, the first *c(i)* bits are obtained from the server and the remaining *sc(i)* bits which are already present in the proxy are concatenated to the first part. This is served to the client. Thus, the client will not experience any delay even for a frame greater than cutoff size because only the portion supported by the bandwidth is obtained from the server and rest is present in the proxy already. The quality of service is guaranteed in this type of delivery, while caching only a part of the video frame.

4.2.2 OC ALGORITHM

The basic disadvantage of the CC algorithm is that the network bandwidth available is not completely utilized. This happens for frames that are less than the cutoff size, where in only a portion of the available bandwidth is utilized and the rest of it is wasted. The OC algorithm proposes to utilize even this bandwidth for prefetching some part of the next frame. The algorithm follows the same procedure as CC algorithm for the frames that are larger than cutoff size. But for the frames that are smaller than cutoff, some portion of the next frame is prefetched. The amount to be prefetched is chosen so as to fill the network bandwidth to *c(i)* bits completely. Since some portion of next frame is prefetched, less number of bits are needed to be cached for any given frame (if its size is greater than cutoff). Thus, this algorithm not only utilizes the network bandwidth more efficiently but also reduces the required cache size for each file. This means more files can be cached in the proxy for the same given storage space as illustrated in Fig. 4.3.

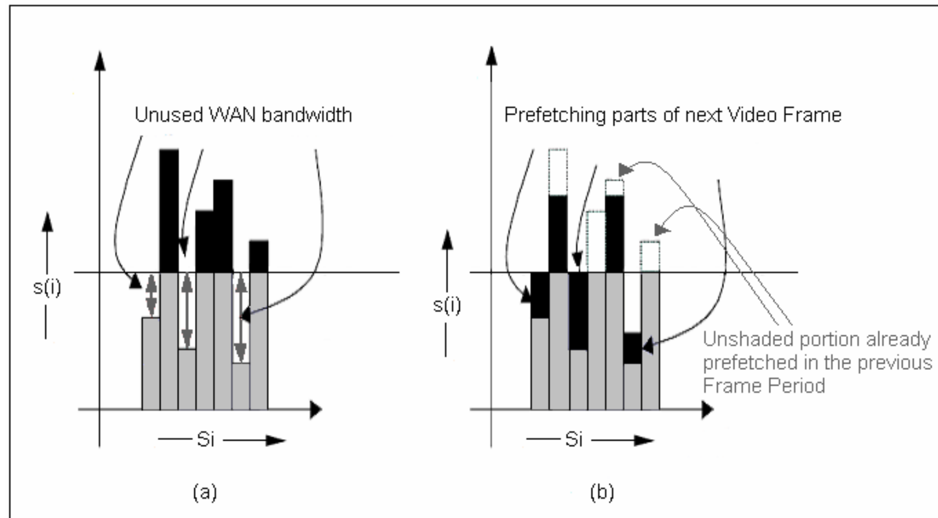


Fig.4.3: Illustration of OC algorithm

Following the above given notations, OC algorithm can be written down as follows:

```
i=0;
repeat
      i = i+1;
      if ( s(i) < c(i) )
      {
           /* The frame size is less than cutoff */
           sc(i) = 0 ;
           /* Nothing to cache */
           /* prefetch some portion of
           next frame */
          /* prefetch size = c(i) - s(i); from
         next frame */
      }
      else
      {
           /* The frame size is larger than
           cutoff */
          sc(i) = s(i) - c(i ) ;
          /* Cache sc(i) bits of frame i
         into the proxy */
       }
until (i>(n-1));
```

Once the file is cached, the proxy would serve any other request for the file in the same manner as in the case of CC Algorithm. But the concatenation process becomes slightly

complicated because of prefetching. For any frame, prefetched portion along with cached portion should be concatenated to the frame obtained from the server.

Benchmark videos for this purpose were obtained from [Traces 1995]. These video files contain a series of numbers indicating the size of the given video frame. A client's request for a video is forwarded to the video proxy. If it is a cache miss, another request will be sent to the origin video server. In response, the origin server will send the sizes of the video frames. The way proxy handles the caching of these video frames depends upon the algorithm (OC or CC). Once the video object is cached, the proxy maintains a file that contains details about the amount of each frame cached and the portion that needs to be fetched from the server. The proxy under consideration is assumed to have infinite cache size to accommodate any number of video files.

Two indices have been used to evaluate the performance of the algorithms. The first one is total cache size required to store a video file. This is obtained by summing all *sc(i)*'s for a given video file. The second performance index is the bandwidth utilization. This is obtained by taking the ratio of bandwidth utilized for fetching portions of the video from the origin server and the total external bandwidth available.

Cache Size:

$$\mathbf{C} = \sum_{i=0}^{n} sc(i)$$

Bandwidth Utilization:

$$\mathbf{B} = \sum_{i=0}^{n} \frac{[(s(i) - sc(i)) / T(i)]}{[r(WAN) \times \sum_{i=0}^{n} T(i)]}$$

Two graphs have been plotted one for each performance index with bandwidth on the x-axis. In case of the total cache size requirement, for all the bandwidths used, OC algorithm occupied less cache space as shown in Fig.4.4. Considering the total bandwidth utilized, OC algorithm outperforms the CC algorithm. It utilizes the external bandwidth

more than the CC algorithm for all the cases of bandwidth, as illustrated in Fig.4.5. By these results it can be seen that though OC algorithm is slightly difficult to implement, it definitely is a better performer.

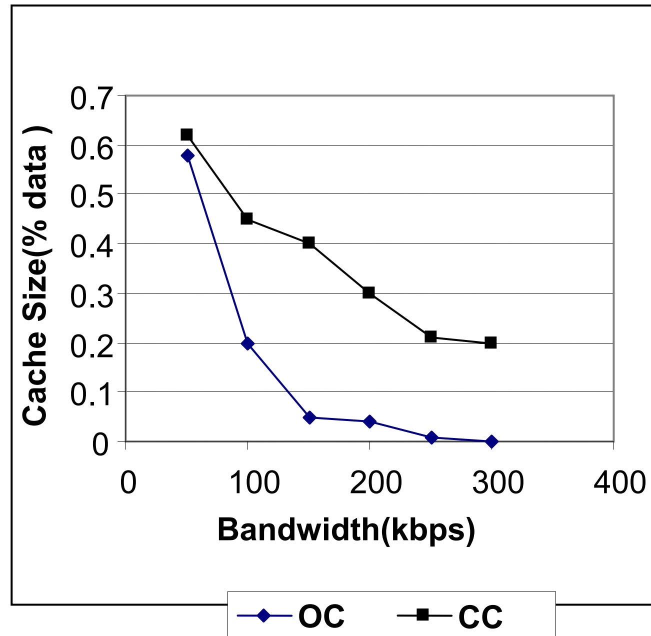


Fig.4.4: Cache Size vs. Bandwidth for streaming multimedia objects

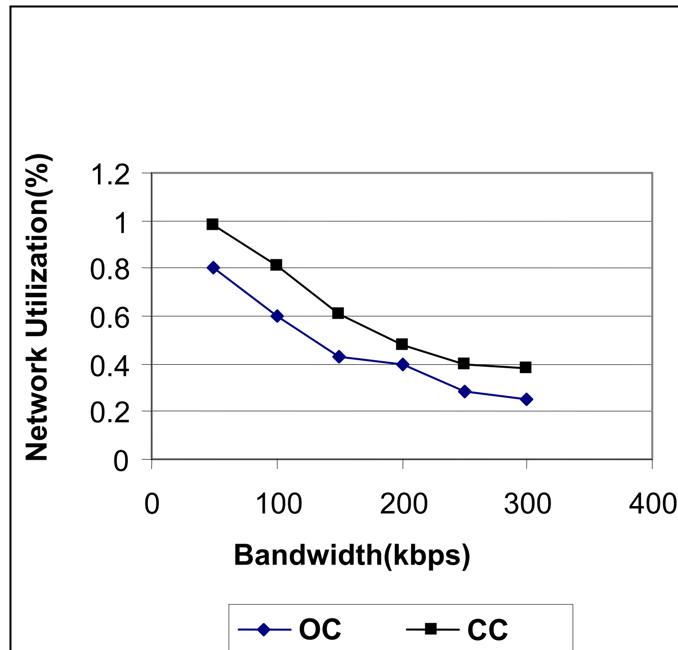


Fig 4.5: Network Utilisation vs. Bandwidth for streaming multimedia objects

The request pattern was formulated using standard benchmark videos obtained specifically for this purpose. Also the proxy in the consideration was assumed to have infinite cache size to accommodate any number of video files. Replacement of the video files has not been considered in the above implementation. In the following sections a replacement policy based on a popularity index is proposed to improve the performance multimedia caching.

4.3 POPULARITY FUNCTION BASED REPLACEMENT POLICY

Current replacement algorithms usually make a binary decision on the caching of an atomic object. The object is cached or flushed in its entirety based on the time or frequency. The above two algorithms cache the objects partially, (i.e. some portion in a frame is cached). A replacement algorithm, which uses just time or frequency, would not suffice. The algorithm should take into account the size because size is the most important factor in the case of multimedia objects. Also, in the case of videos, popularity of the videos is quite essential, because once a video becomes popular the requests for such videos grow exponentially and less popular videos are almost ignored.

The caching of the videos using the above two algorithms, CC and OC, happens frame-by-frame. Thus, logically the replacement also can be frame-by-frame. Once the victim video is selected, the deletion of frames can be done from first frame or the last frame. The best choice would be to start deleting from the last frame, because any remaining frames could be used to serve a future request, at least partially. This also serves another purpose; if any request for the video being deleted arrives, it could be locked and other file could be chosen for deletion.

4.3.1 POPULARITY FUNCTION OR FREQUENCY INDEX

As mentioned above, the size and frequency of the video are essential for knowing the popularity of the video, and we choose a popularity function that takes into account both the parameters. We use a modified *hit ratio* of a cached stream as a metric to measure its popularity. The proxy can easily count the number of byte hits for every cache resident

stream during an interval. Most of the current schemes assign a binary value to a hit, i.e. 0 for lack of interest and 1 for each request. However not all bytes of a stream are present in the proxy. A hit in this case means the hit for bytes present in the cache. The *bytes requested* would not be the size of the entire video requested, but the number bytes cached for that video. The *bytes hit* would then become the number of bytes present in the cache for that particular video (bytes present is different from bytes cached, because deletion starts from the last frames, and in some cases only last few frames are removed).

Intuitively, the popularity of each stream must reflect the level of interest that is observed through this interaction. We assume that the total bytes requested for each stream indicates the level of interest in that stream. For example if a client only watches half of one stream, his level of interest is half of a client who watches the entire display. Based on this observation, we extend the semantic of a hit and introduce a *Byte Hit*, called a *bhit*, which is defined as follows:

For the video V,

$$\mathbf{bhit} = \frac{Bytes\,Delivered(V)}{Bytes\,Requested(V)} = \frac{Bytes\,Present(V)}{Bytes\,Cached(V)} \quad 0 \le bhit \le 1$$

The proxy server keeps track of byte hits for each request from the client. The cumulative value of Byte Hit is used as a Frequency Index of a cached stream. The popularity of each video is given by its Frequency Index and is recalculated at the end of a session as follows:

$$\mathbf{FI} = \sum_{i=0}^{n} bhit(i)$$

Where FI is the popularity of the video, n is the number of times the video is requested for in the given time interval and *bhit(i)* is the Byte Hit for i^{th} request.

Size of the video is taken is into account by the Byte Hit, because it’s a byte-hit ratio. Since we are cumulating Byte Hit for each hit, the frequency is also taken care of. Thus the above formula has the two important factors: size and frequency, which makes it a true measure of the popularity of the video.

4.3.2 THE REPLACEMENT ALGORITHM

The replacement algorithm is completely based on popularity. If there is a cache hit, then the Byte Hit of the video is calculated for the present request. The Frequency Index (FI) of that video would be updated by adding the Byte Hit to its previous FI.

The issue of replacement comes into picture if there is a cache miss. On a cache miss, the requested file would be cached into the proxy. While caching, if the cache size exceeds the maximum permissible amount, one of the files should be replaced. A data structure that contains FI's of the videos is maintained. The video with least FI is selected as victim video for replacement. Victim video would then be replaced by the requested video frame-by-frame. The victim video is deleted starting from the last frame.

While deleting, if a request arrives for the victim video then it is locked and is not allowed to be deleted. Another victim file would be selected and the replacement resumes again. This not only satisfies the users' requests but also increases the hit-rate. The replacement continues till the first frame of the victim video is replaced (i.e. till the video is completely deleted). If the requested video isn't completely cached yet, another victim video should be selected from the list and replacement process continues as above. Hit-Ratio of the proxy would be calculated for each request received from the client.

If (Cache Hit)
{
*//Calculate the **bhit** for the requested video.*

$$\textbf{bhit} = \frac{Bytes\,Delivered(V)}{Bytes\,\mathrm{Re}\,quested(V)} = \frac{Bytes\,\mathrm{Pr}\,esent(V)}{Bytes\,Cached(V)} \quad 0 \le bhit \le 1$$

//Update the Frequency Index
FI = FI + bhit;
}

If (Cache Miss)
{
// Cache the frames of the requested video
Cache (OC or CC);
If (Cache Size + NextFrameSize >
MaxCacheSize)
{
REPLACE:
//Replace a video.
//Select the least popular video as

```
          "victim video".
          Select (V) where FI = FImin.

          //Replace the victim video from last
          frame with the requested video.
          Replace (Victim Video with
          Requested  Video);

          //If a request arrives for the victim video
          while being deleted, lock it.
         If (Request (Victim Video) == TRUE)
          {
               Lock (Victim Video);
            // Select another Victim video that
              comes next in the list and replace it.
              Goto REPLACE;
          }
         // Delete the entire victim video if required
          if (Victim Video Frame No. = = 1)
          {
              Delete(Victim Video);
   // if the Requested video is not cached
              completely yet

            If (Cache (Requested Video)! = Finished)
            {
              // Select another Victim video that
              comes next in the list and  replace it.
              Goto REPLACE;
            }
          }
      }
}
```

Implementation of the replacement algorithm in the video proxy was done using tree and stack structures. Tree was used to maintain the Frequency Indices of the videos (i.e. as a popularity table). Since cache placement or replacement is done frame-by-frame, stack was used to store the frames within a video in a LIFO order. Hit ratio was the performance metric for comparing the two algorithms. The benchmark videos for this purpose were obtained from [Traces 1995] and are the same as used earlier.

The sequence of actions is as follows. The client, first, requests for a particular video file from the proxy server. The proxy server maintains a cache of the video files. On receipt of a request, the proxy server first searches in its local cache whether it has the requested file. In case of the file being present, part of the file is obtained from the cache and the portion greater than the cut-off limit is supplied from the cache itself. The cut-off limits and the cache size are obtained from the user during run-time. In case the file is not present (cache miss), the proxy contacts the origin server. The origin server sends the video frame sizes to the proxy server. For each frame size received from the server, the proxy runs the OC or CC algorithm. In case of a cache miss, the replacement algorithm is run. Hit Ratio is calculated for each client request, be it a cache hit or a miss.

4.3.3 DISCUSSION OF RESULTS

The Frequency Index replacement (*FIR*) algorithm has been run for request sequences formed by benchmark videos mentioned earlier. The same request sequences were also tested with standard replacement algorithms like Least Recently Used (LRU) and Least Frequently Used (LFU). Graphs indicating their performance (hit-ratio) have been plotted.

The replacement algorithm has been run for two request sequences. Two sets of graphs have been plotted for each request pattern. The hit ratio has been plotted with respect to changing bandwidth and max cache size. In the first case, the cache size was kept constant and bandwidth was varied over a range of values. The bandwidths chosen for simulation were 40 Kbps, 56 Kbps and 100 Kbps. The obtained hit ratio for both OC and CC algorithms are plotted against bandwidth. Similarly in the second case the bandwidth was kept constant and maximum cache size was varied over a range of values. The cache sizes chosen for simulation were 40, 80, 160 and 200 GB. The obtained hit ratio for both OC and CC algorithms are plotted against cache size.

It is observed that the hit-ratio increases with increase in bandwidth. This is because as the available bandwidth increases, the cut-off size increases and hence the number of bytes cached per video decreases. Consequentially, more number of videos can be

cached, thereby improving the hit-ratio. It is also observed that the OC algorithm gives a better hit ratio compared to the CC algorithm. As the bandwidth increases, the hit ratios of both the algorithms become nearly the same, because both the algorithms behave the same way at high bandwidths.

Refer to Fig. 4.6 to 4.8

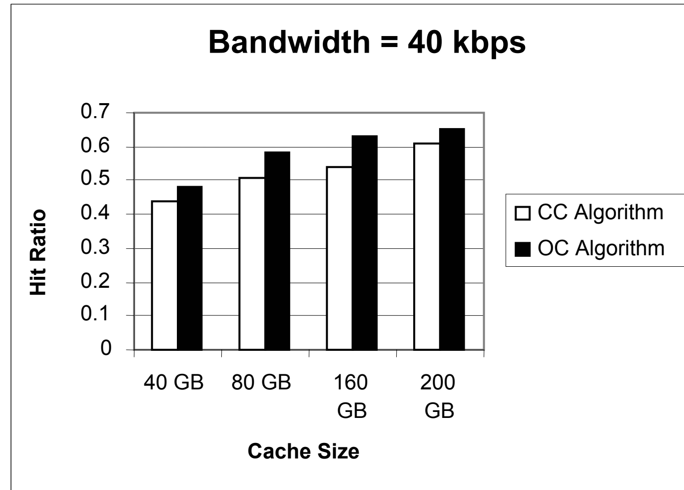


Fig. 4.6: Hit Ratio vs Cache Size for a constant bandwidth of 40kbps

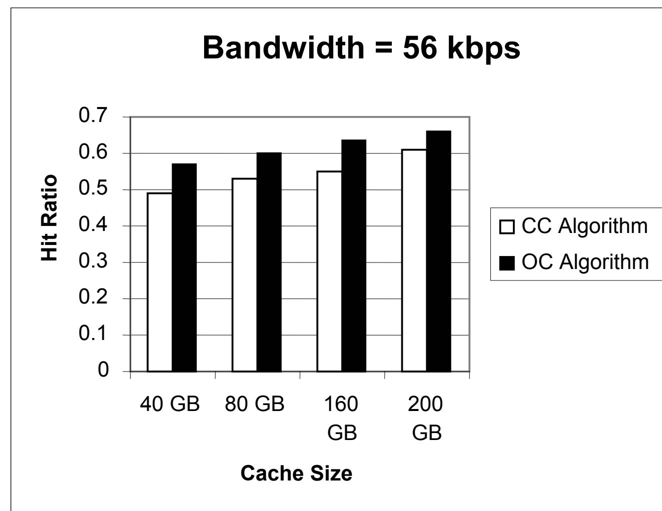


Fig. 4.7: Hit Ratio vs Cache Size for a constant bandwidth of 56kbps

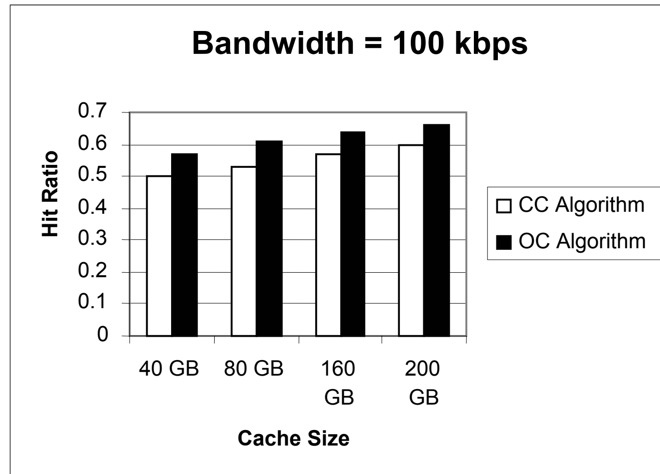


Fig. 4.8: Hit Ratio vs Cache Size for a constant bandwidth of 100kbps

In the second set of graphs (Fig. 4.9 to 4.12), the hit-ratio increases with increase in cache size. This can be explained by the argument that for a constant bandwidth, more the available cache size, more the number of videos that can be cached and hence more the hit-ratio. In this case as well, the OC algorithm gives a better performance compared to the CC algorithm. As the cache size increases, the hit ratios of both the algorithms become nearly the same, because at high cache sizes any number of videos can be cached, irrespective of the algorithm.

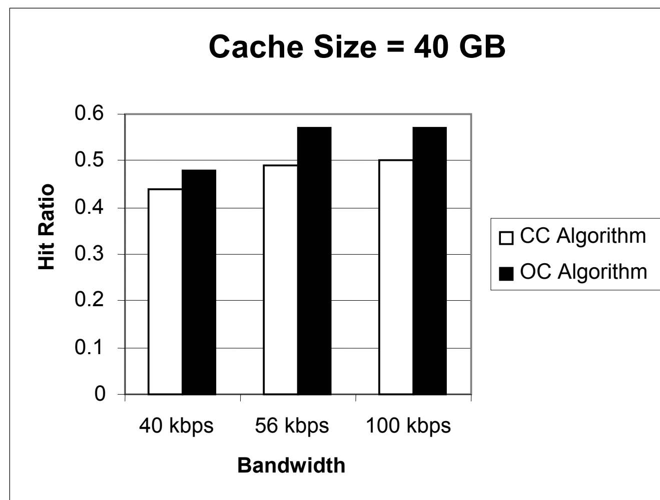


Fig 4.9: Hit Ratio vs Bandwidth for a constant Cache Size of 40GB

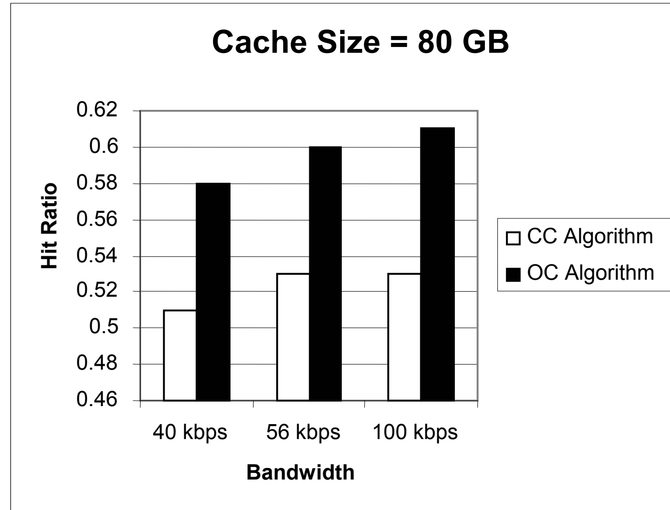


Fig 4.10: Hit Ratio vs Bandwidth for a constant Cache Size of 80GB

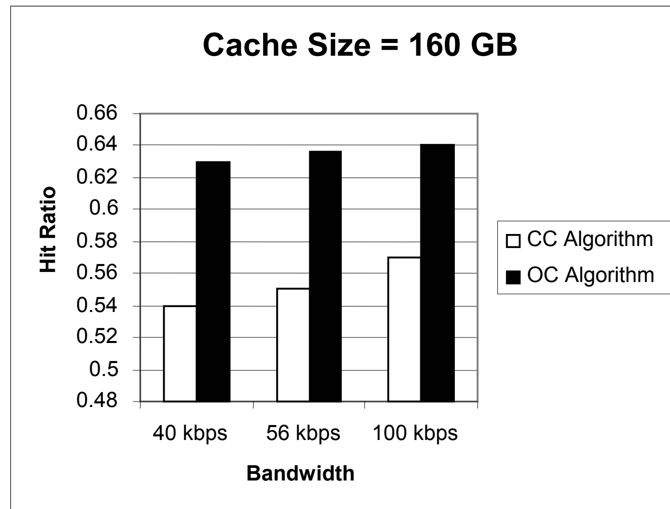


Fig 4.11: Hit Ratio vs Bandwidth for a constant Cache Size of 160GB

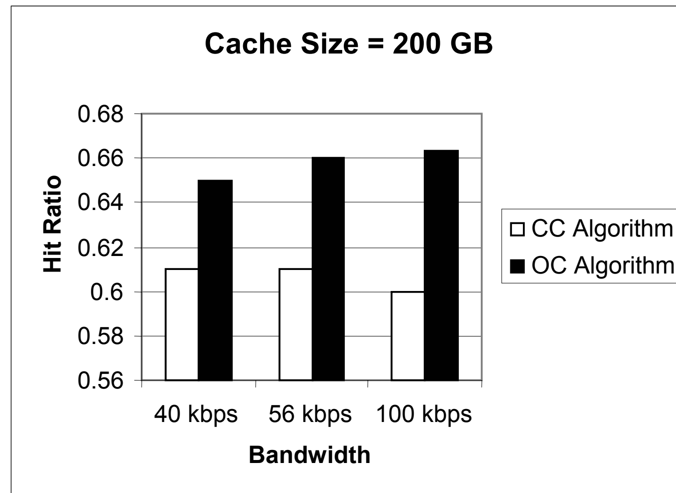


Fig 4.12: Hit Ratio vs Bandwidth for a constant Cache Size of 200GB

In comparison with LFU and LRU under similar memory availability, FIR algorithm yields better hit-ratio as shown in Fig.4.13. This is primarily due to the fact that replacement here happens frame by frame as opposed to complete Boolean replacement in the other two algorithms.

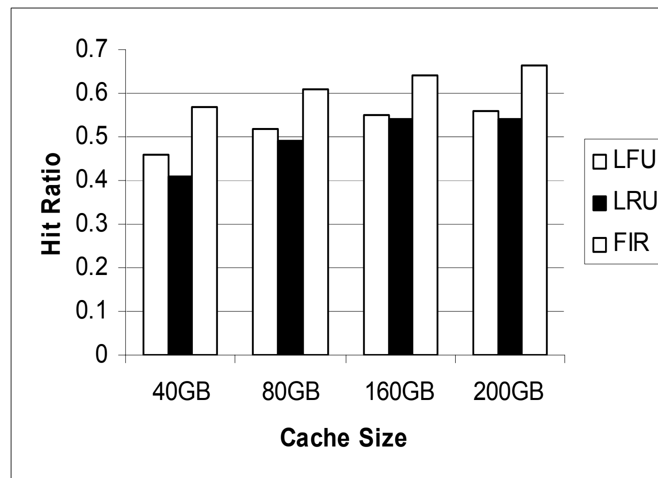


Fig 4.13 Comparison of the performance of LFU, LRU and FIR

Also to verify the reliability of the proposed FIR algorithm, the algorithm was tested under varying conditions of bandwidth and cache size using both OC and CC for caching. As seen in the results obtained, algorithm did not indicate extreme swings in performance when the parameters were varied. The hit ratios obtained varied between 0.5 and 0.7, which can be construed as an absolute indication of the efficient functioning and reliability of the algorithm under all conditions.

4.4 SUMMARY

In this chapter we discussed various methods used for Multimedia objects Web Caching. Video staging algorithm is a caching technique which allows partial caching of multimedia objects. We also discussed the performance of Cut-off Caching (CC) and Optimal Caching (OC) techniques. A Frequency Index based Replacement policy was proposed and its performance was tested for CC and OC algorithms.

CHAPTER 5

SOFT COMPUTING TECHNIQUES IN WEB CACHING

5.1 INTRODUCTION

Unlike the traditional hard computing methods, soft computing techniques such as fuzzy logic, neural networks and genetic algorithms are aimed at accommodating the pervasive imprecision of the real world. The guiding principle of soft computing technique is to exploit the tolerance for imprecision, uncertainty, and partial truth to achieve tractability, robustness, and low solution cost. Evolutionary programming has been successfully applied to numerous problems from different domains. Fuzzy Logic and Genetic algorithms are popular techniques, which can be applied to many computational problems requiring adaptation to a changing environment or search through a huge number of possibilities for solutions [Vakali 1999a].

Fuzzy Logic representations founded on fuzzy set theory try to capture the way humans represent and reason with real-world knowledge in the face of uncertainty. Uncertainty could arise due to generality, vagueness, ambiguity, chance or incomplete knowledge [Rajasekaran 2003]. In recent years, attempts have been made to use soft computing techniques for web caching and replacement. There is a need to base the replacement process on both qualitative and quantitative information [Calzarossa 2003]. The algorithms must take care of the characteristics and properties of workloads of proxy servers and must apply some qualitative reasoning to identify the pages to evict from the cache. Recently the Fuzzy Algorithm for web caching has been proposed by M. Calzarossa and G. Valli [Calzarossa 2003]. Here the variables describing each web object cached are first fuzzified. A set of fuzzy control rules is then applied and their outputs are defuzzified as to identify the object to evict.

Genetic Algorithm (GA) has been used to solve scientific problems demanding optimization and adaptation to a changing environment. The idea in this approach is to evolve a population of candidate solutions to a given problem, using operations inspired

by natural genetic variation and natural selection, expressed as "survival of the fittest". GAs are being applied to many computational problems requiring either search through a huge number of possibilities for solutions, or adaptation to a changing environment. More specifically, GAs have been applied in the areas of scientific modeling and machine learning, but recently there has been a growing interest in their application in other fields [Vakali 1999b, Goldberg 2004]

This chapter describes the application of fuzzy logic with two different rule sets and application of Genetic Algorithm technique for replacement policies for caching static web objects and an algorithm for caching streaming multimedia web objects. Our choice is motivated by the need to take both qualitative as well as quantitative information into account for replacement while using soft computing method. These algorithms consider the nature and properties of the workloads of institutional proxy servers and apply some qualitative reasoning to identify the object to evict from the cache.

5.2 FUZZY REPLACEMENT ALGORITHM

In the Fuzzy Inference Method, whenever the cache is full and a cache miss occurs, the fuzzy algorithm determines the objects to be evicted by computing a mathematical merit called Replacement Probability for each of the objects, depending on certain parameters of input viz. Size, Frequency and Access recency. The fuzzy knowledge base includes the input and output variables, their respective membership functions, and the fuzzy rule base. The algorithm involves fuzzification, rule inference and defuzzification.

Application of Fuzzy Logic consists of three stages: Input, Processing and Output. The input stage maps sensor or other inputs to the appropriate membership functions and truth-values. The process of converting a crisp input value into a fuzzy value is known as "fuzzification". The processing stage invokes each appropriate rule from a 'set of rules' and generates a result for each, then combines the results of the rules. Finally the output stage converts the combined result back into a specific control output value. The collection of logic rules on which the processing stage is based is a bunch of If- Then statements where the 'IF' part is the "antecedent" and the 'THEN' part is called the

"consequent". In practice, fuzzy rule sets usually have several antecedents that are combined using fuzzy operators such as AND, OR and NOT. There are several different ways to define the result of a rule, but one of the most common and simplest is the "max-min" inference method, in which the output membership function is given the truth-value generated by the premise. The results of all the rules that have fired are "defuzzified" to a crisp value by one of several methods such as the "centroid method" in which the "center of mass" of the result provides the crisp value. In centroid defuzzification, the values are ORed, that is, the maximum value is used and values are not added, and the results are then combined using a centroid calculation [Kosko 1994].

The proposed approach is that the variables describing each web document are "fuzzified", fuzzy rules from a given rule set are applied and then their outputs are "defuzzified" to identify the objects to be expelled from the cache. Based on the above approach we propose a cache replacement policy for web proxy servers. The replacement policy is as follows: when a cache miss occurs and the cache is full, the algorithm determines the objects to evict by computing for each object in the cache a figure of merit, namely, its probability of replacement. Among the objects ranked according to their probability of replacement, the algorithm chooses the one with the highest rank.

The operation of any Fuzzy-based system depends on the proper choice of process state input variables and control output variables. Here, we have chosen three input variables to represent the process state. These variables describe each web object in terms of its size, access frequency and access recency i.e. time elapsed since last access. As output variable, we have chosen the probability of replacement, RP of each object. For each of these variables, the fuzzy sets with the membership functions are designed which describes the degree of membership of the variable to the corresponding fuzzy set. Figs 5.1, 5.2 and 5.3 show the membership functions of the three input variables. Here, membership functions having triangular or trapezoidal shapes have been used.

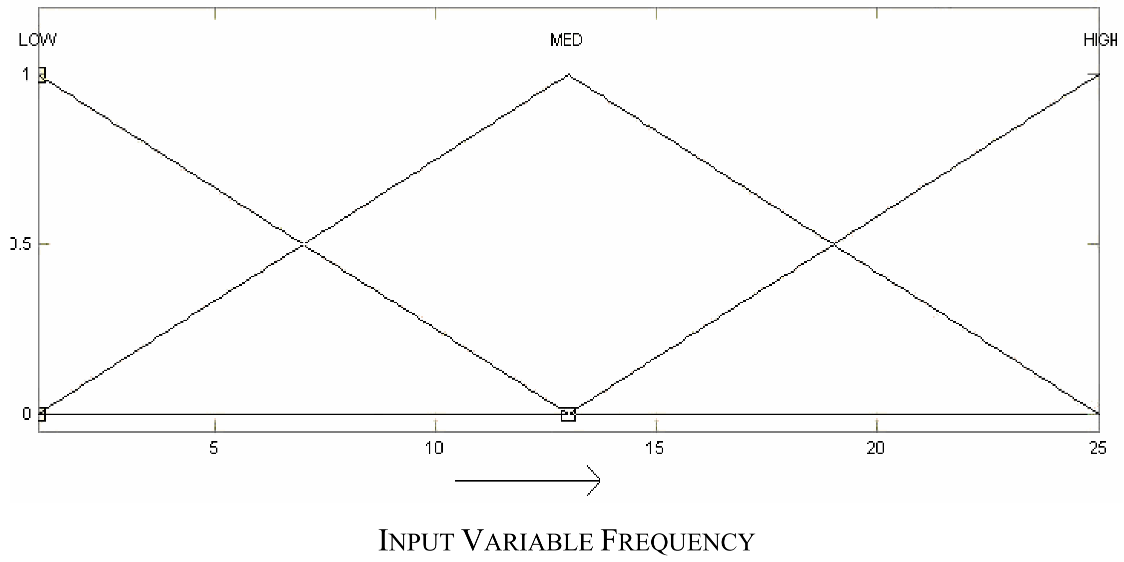


Fig 5.1: Membership Function for Input Variable Frequency

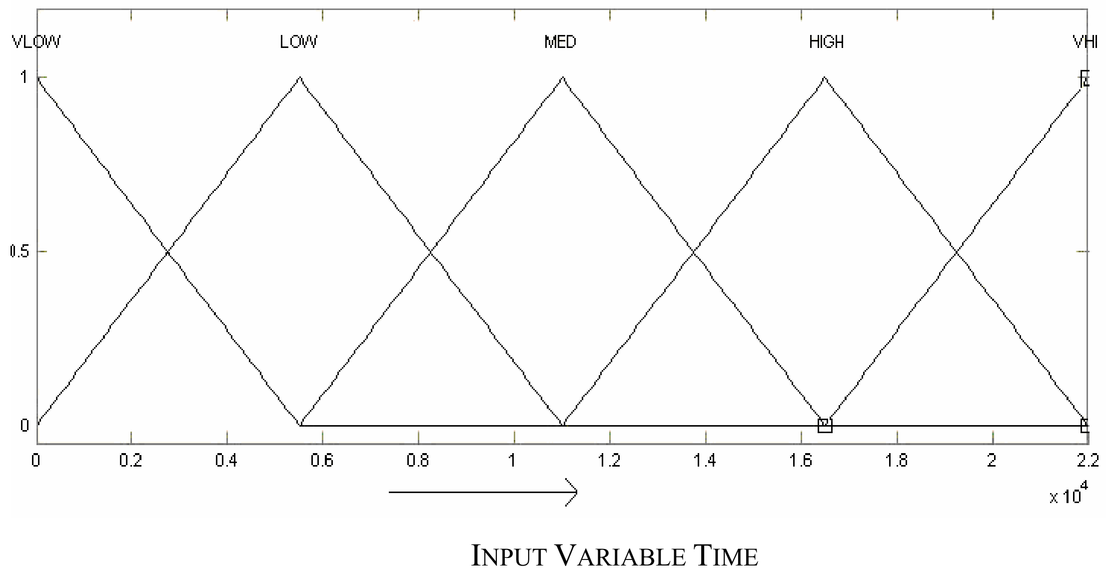
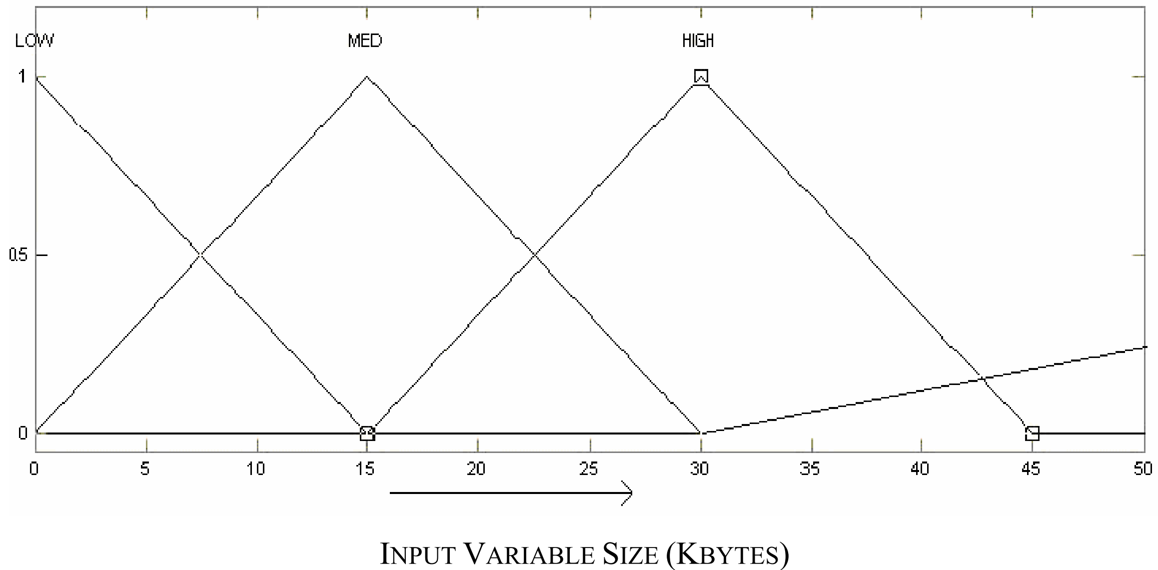


Fig 5.2: Membership Function for Input Variable Time



INPUT VARIABLE SIZE (KBYTES)

Fig 5.3: Membership Function for Input Variable Size

The simulations have been carried out with 2 rule sets: Fuzzy12 with 12 rules and Fuzzy24 with 24 rules as shown in Table 5.1 and Table 5.2 respectively. In both cases, there are 3 membership functions associated with the variable Frequency. LOW, MEDIUM and HIGH have been used as labels i.e. descriptive lingual values. In Fuzzy12, there are 3 membership functions LOW, MEDIUM and HIGH associated with variable Size whereas in Fuzzy24, an additional membership function VERY HIGH has also been added. To describe the variable Time, in both cases, we have chosen 5 variables. This is because the algorithm requires a finer control of this variable. The corresponding descriptive labels are VERY LOW, LOW, MEDIUM, HIGH, and VERY HIGH. The centers and the left and right limits of the membership functions have been obtained as a result of analysis of proxy workloads. Fig 5.4 shows the membership functions of the output variable, that is, the probability of replacement RP. As can be seen, for Fuzzy12 and Fuzzy24, 4 membership functions, with descriptive variables LOW, MEDIUM, HIGH, VERY HIGH, have been associated with this variable.

Table 5.1: Fuzzy12 Rule Sets

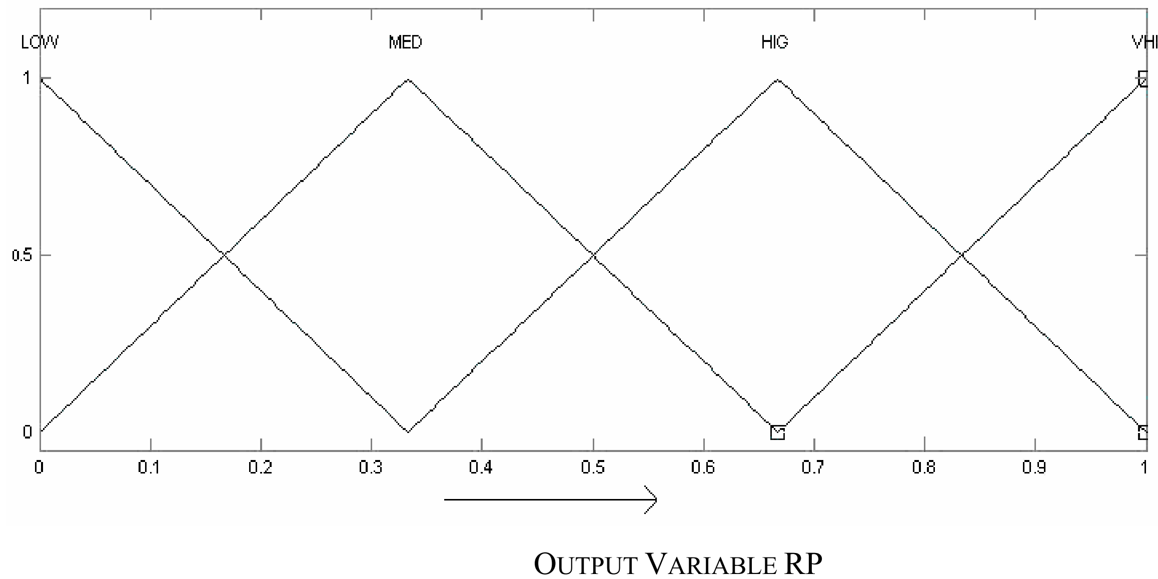
FUZZY12

If (Frequency is LOW) and (Time is VHI) and (Size is MED) then (RP is VHI)
If (Frequency is LOW) and (Time is HIG) and (Size is HIG) then (RP is VHI)
If (Frequency is MED) and (Time is VHI) and (Size is HIG) then (RP is VHI)
If (Frequency is LOW) and (Time is VHI) and (Size is HIG) then (RP is VHI)
If (Frequency is LOW) and (Time is HIG) and (Size is LOW) then (RP is HIG)
If (Frequency is MED) and (Time is HIG) and (Size is LOW) then (RP is MED)
If (Frequency is MED) and (Time is VHI) and (Size is MED) then (RP is HIG)
If (Frequency is MED) and (Time is HIG) and (Size is HIG) then (RP is HIG)
If (Frequency is HIG) and (Time is VHI) and (Size is HIG) then (RP is LOW)
If (Frequency is HIG) and (Time is HIG) and (Size is HIG) then (RP is LOW)
If (Frequency is LOW) and (Time is MED) and (Size is HIG) then (RP is HIG)
If (Frequency is MED) and (Time is HIG) and (Size is MED) then (RP is MED)

Table 5.2: Fuzzy24 Rule Sets

FUZZY24

If (Frequency is LOW) and (Time is VHI) and (Size is MED) then (RP is VHI)
If (Frequency is LOW) and (Time is HIG) and (Size is HIG) then (RP is VHI)
If (Frequency is MED) and (Time is VHI) and (Size is HIG) then (RP is VHI)
If (Frequency is LOW) and (Time is VHI) and (Size is HIG) then (RP is VHI)
If (Frequency is LOW) and (Time is HIG) and (Size is LOW) then (RP is HIG)
If (Frequency is MED) and (Time is HIG) and (Size is LOW) then (RP is MED)
If (Frequency is MED) and (Time is VHI) and (Size is MED) then (RP is HIG)
If (Frequency is MED) and (Time is HIG) and (Size is HIG) then (RP is HIG)
If (Frequency is HIG) and (Time is VHI) and (Size is HIG) then (RP is LOW)
If (Frequency is HIG) and (Time is HIG) and (Size is HIG) then (RP is LOW)
If (Frequency is LOW) and (Time is MED) and (Size is HIG) then (RP is HIG)
If (Frequency is LOW) and (Time is VLOW) and (Size is LOW) then (RP is MED)
If (Frequency is MED) and (Time is VLOW) and (Size is LOW) then (RP is MED)
If (Frequency is HIGH) and (Time is VLOW) and (Size is MED) then (RP is LOW)
If (Frequency is MED) and (Time is VLOW) and (Size is LOW) then (RP is MED)
If (Frequency is MED) and (Time is MED) and (Size is LOW) then (RP is MED)
If (Frequency is LOW) and (Time is MED) and (Size is MED) then (RP is HIG)
If (Frequency is HIG) and (Time is MED) and (Size is LOW) then (RP is MED)
If (Frequency is MED) and (Time is VHI) and (Size is MED) then (RP is MED)
If (Frequency is HIGH) and (Time is MED) and (Size is MED) then (RP is MED)
If (Frequency is MED) and (Time is VLOW) and (Size is MED) then (RP is MED)
If (Frequency is LOW) and (Time is VLOW) and (Size is VHI) then (RP is HIG)
If (Frequency is LOW) and (Time is MED) and (Size is VHI) then (RP is HIG)
If (Frequency is MED) and (Time is VLOW) and (Size is VHI) then (RP is MED)



OUTPUT VARIABLE RP

Fig 5.4: Membership Function for Replacement Probability

Having defined the membership functions, we construct the fuzzy rule base. As mentioned earlier in this section, the rule base consists of fuzzy conditional statements in the form “if-then” in which the antecedent is a condition pertaining to the particular application and the consequent is an action for the controlled system. Each rule in the antecedent involves one or more variables. There is no general procedure for deciding on the optimal number of fuzzy control rules and the role of each variable. We have defined two sets of rules, namely Fuzzy24 and Fuzzy12 having 24 and 12 rules respectively. The aim of Fuzzy24 as well as Fuzzy12 is to keep in the cache objects that have been accessed very recently, and to evict large objects. Moreover, among objects with similar size, the rules penalize objects characterized by a small number of accesses or objects accessed very recently. In Fuzzy24, 24 rules have been designed so as to take into account the Size of the object when it is large.

Fuzzification does the job of mapping crisp input data into fuzzy sets by means of the corresponding membership functions. The input values Size, Frequency and Recency related to each object are converted into linguistic labels. Next, for each rule, the antecedent is evaluated based on the descriptive label and the degree of truth is computed by applying the fuzzy AND operator, the product. The aggregation process combines the

outputs of the rules by applying the maximum operator to each descriptive label of the output variable RP (i.e. the probability of replacement). The defuzzification transforms these four values into a non-fuzzy control action corresponding to the probability of replacement of the object. The defuzzification used in our algorithm is based on the centroid method. The masses, obtained as a result of the aggregation process, have been placed at the three points where the membership functions of the output variable RP intersect, that is, at the points 0.25, 0.5 and 0.75. Moreover, the mass corresponding to the label VERY HIGH has been placed at 1. Finally, the objects are ranked according to their probability of replacement. The algorithm evicts the objects with the highest rank.

5.2.1 IMPLEMENTATION

The performance analysis of the Fuzzy algorithm has been carried out using the simulation of a trace obtained from the logs from Duke University [Traces 1995]. This trace contains a day's worth of all the HTTP requests to the Environmental Protection Agency (EPA) WWW server. During the simulation, only cacheable static objects were considered. The proposed Fuzzy algorithm is compared with the traditional cache replacement algorithms viz. LRU, LFU and SLRU Algorithms. Figs 5.5 to 5.10 illustrate the results of this simulation.

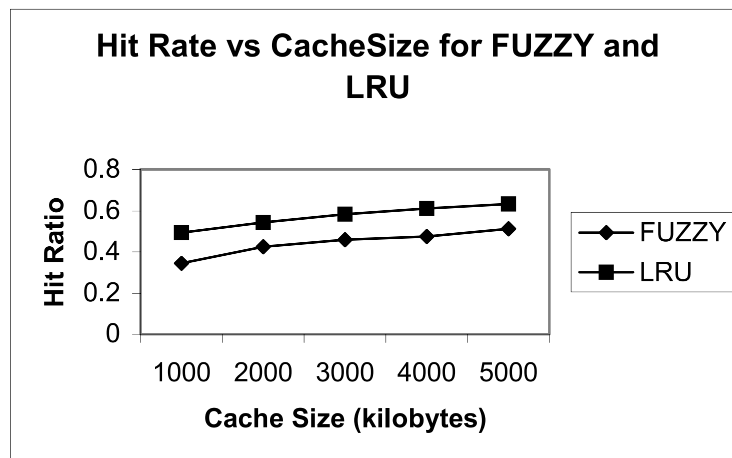


Fig 5.5 Hit Ratio Obtained from the trace1 for LRU and FUZZY12

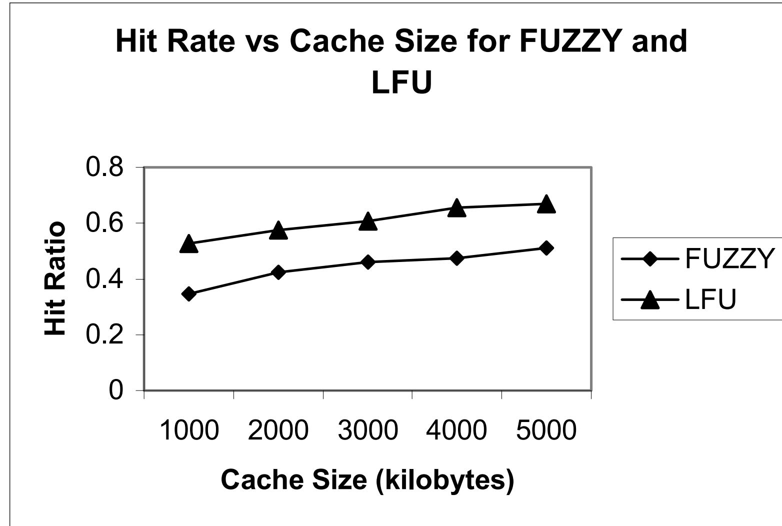


Fig 5.6 Hit Ratio Obtained from the trace1 for LFU and FUZZY12

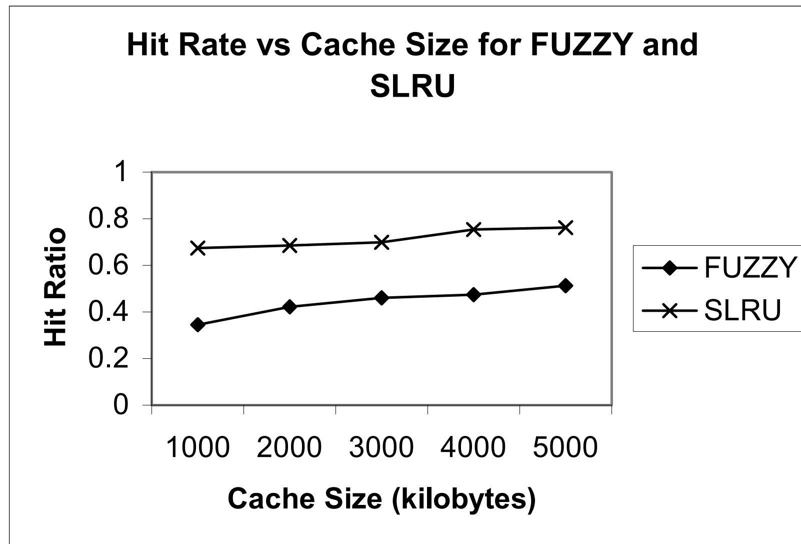


Fig 5.7 Hit Ratio Obtained from the trace1 for SLRU and FUZZY12

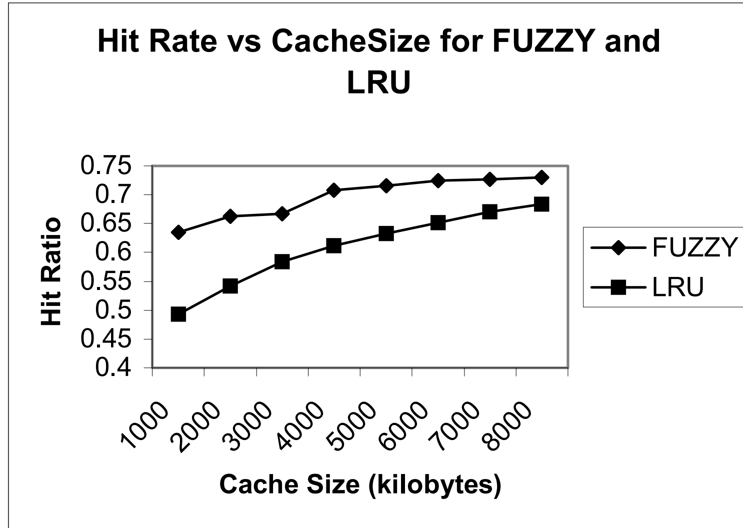


Fig 5.8 Hit Ratio Obtained from the trace1 for LRU and FUZZY24

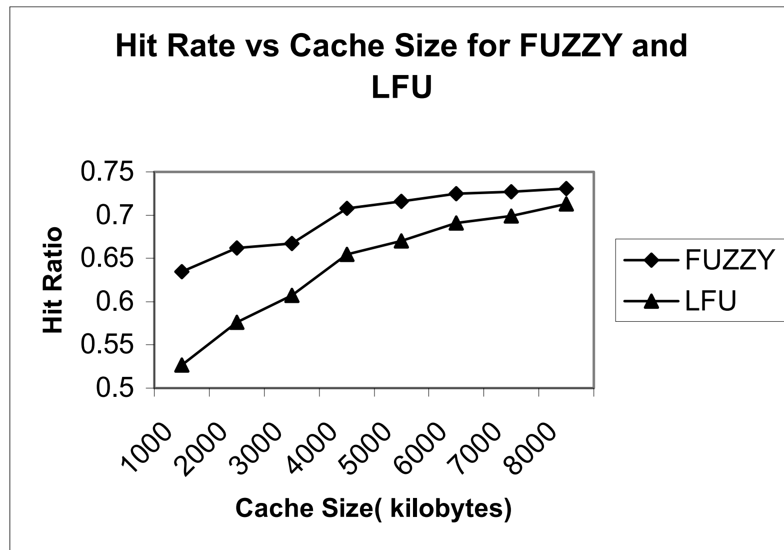


Fig 5.9 Hit Ratio obtained from the trace1 for LFU and FUZZY24

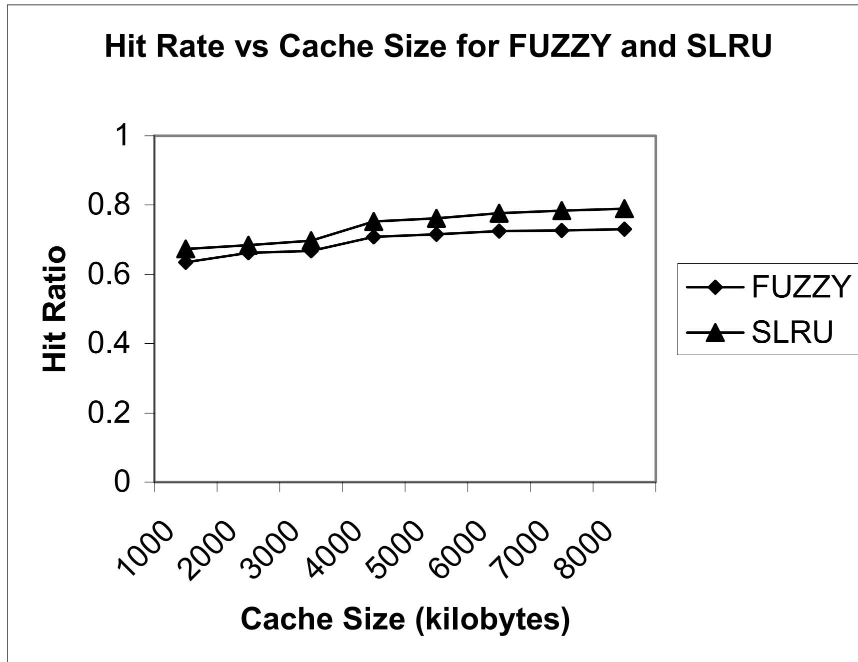


Fig 5.10 Hit Ratio Obtained from the trace1 for SLRU and FUZZY24

Tests have also been done on small logs obtained from BITS proxy server. For these tests, since the log has fewer requests, the cache size is kept small. This was to experiment with smaller individual caches for the clients in some cases, which can be allowed on demand. Figs 5.11 to 5.13 show these results.

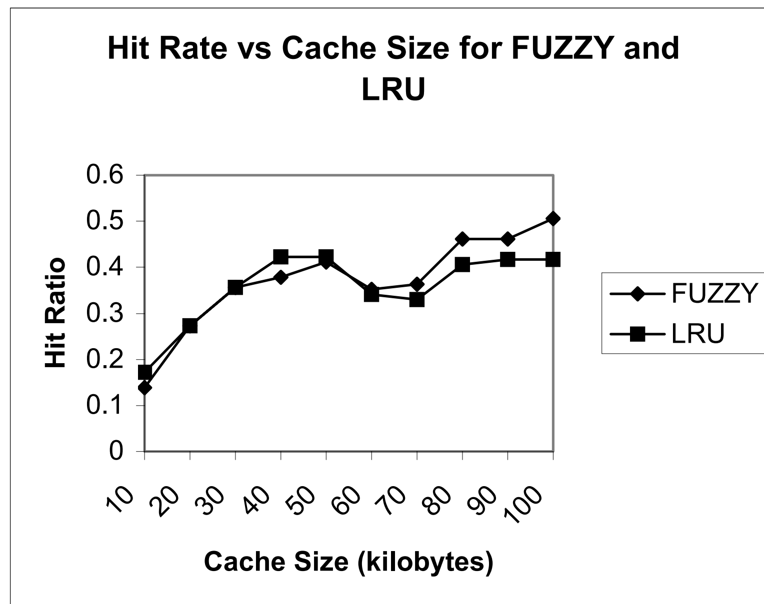


Fig 5.11 Hit Ratio Obtained from the trace2 for LRU and FUZZY24

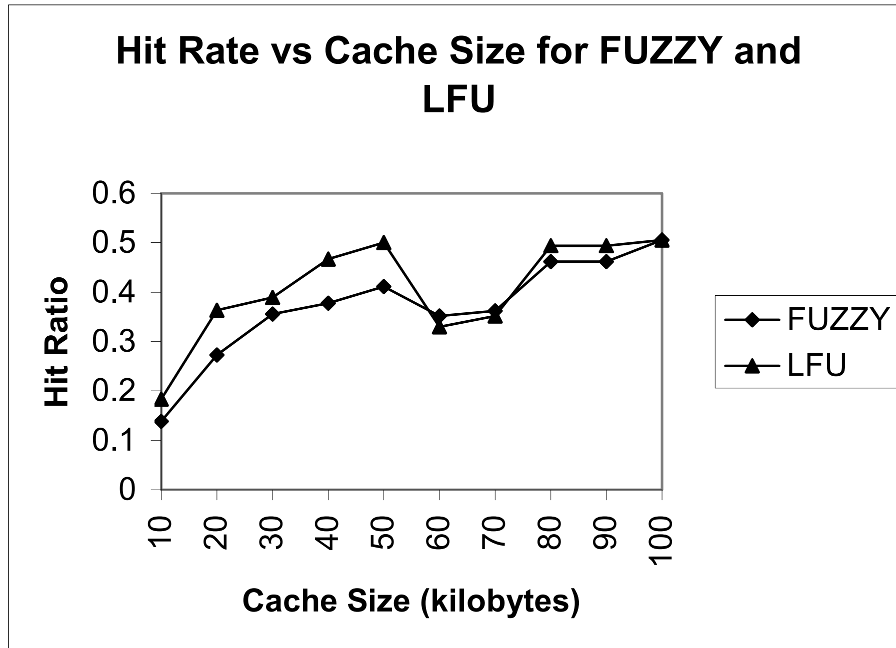


Fig 5.12 Hit Ratio Obtained from the trace2 for LFU and FUZZY24

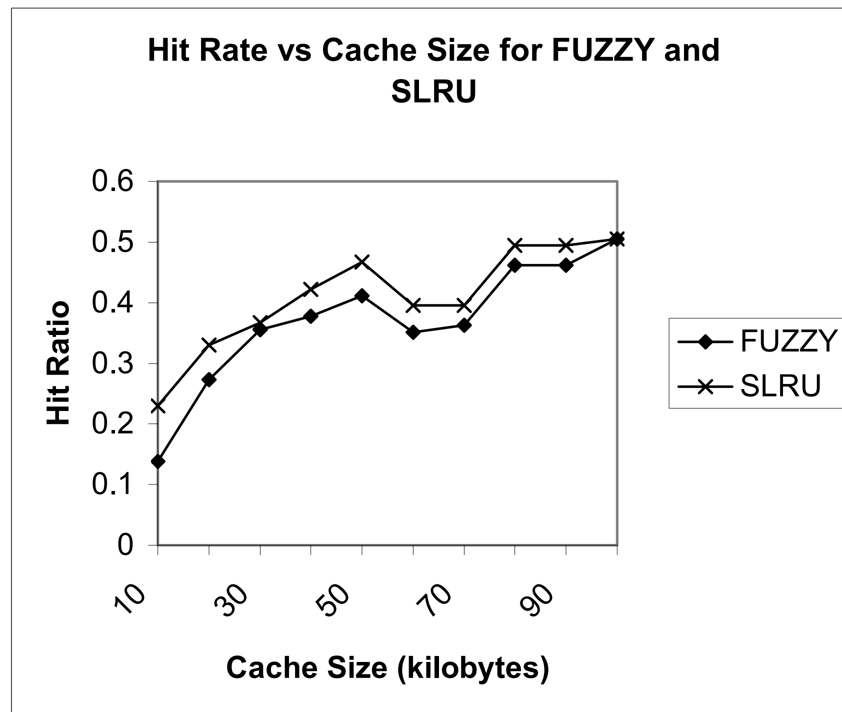


Fig 5.13 Hit Ratio Obtained from the trace2 for SLRU and FUZZY24

5.2.2 DISCUSSION OF RESULTS

The results of the simulations are depicted in the graphs. The results of the simulation show that;

- In case of Fuzzy24 rule set the FUZZY algorithm outperforms the LRU and LFU algorithms. However the Hit Rate of SLRU is nearly equal to that of the FUZZY algorithm as observed in Fig 5.8, 5.9 and 5.10.
- The Fuzzy Algorithm is outperformed by the size-based algorithm, SLRU.
- In case of Fuzzy12 rule set, the performance of FUZZY algorithm in terms of Hit Rate is poor as compared to LRU, LFU and SLRU as observed in Fig 5.5, 5.6 and 5.7.

The results of the simulation on the small log of fewer requests show that:

- For very small cache sizes, the performance of FUZZY is inferior to that of the other three algorithms as observed in Fig 5.11, 5.12 and 5.13.
- However for larger cache sizes, the hit rates for FUZZY are higher or nearly equal to that of LRU and LFU. The SLRU algorithm shows better performance as compared to FUZZY in most cases.

This shows that the FUZZY algorithm is suitable on a cache, which generally faces fewer requests, when the cache size is above a certain threshold value.

Byte Hit Rate is another metric of performance evaluation. It is the fraction of requested bytes retrieved directly from the cache. In case of Fuzzy24 rule set, the FUZZY Algorithm has mediocre performance with respect to this metric. The LRU has the least Byte Hit Rate and both SLRU as well as LFU has the highest Byte Hit Rate, with SLRU performing better for lower cache sizes and LFU performing better for higher cache sizes.

Thus, the replacement algorithm based on Fuzzy Logic that has been discussed here can lead to significant improvement in performance in certain situations such as when the cache size is small and more number of Fuzzy rules is used in the rule set. The performance of Fuzzy Algorithm improves greatly over the conventional methods as the number of rules in the rule set increases. Fuzzy algorithm’s performance is not so

favorable when there are fewer rules, as is evident with the Fuzzy12 rule set, except for particular cache sizes. Moreover the results of the simulations have shown that the Fuzzy algorithm achieves good performance even for small cache size, less than 5% of the cache capacity. However it remains inferior to SLRU.

The complexity of the algorithm is of the order of the number of objects O(n), in the cache as it evaluates for each page its probability of replacement. The running time complexity also depends upon the number of rules in the rule set. However, even though this complexity is larger than the complexity of most of the traditional algorithms, it is not much of an issue. The workload of a proxy server is typically I/O bound and the processor is never the bottleneck of the system. However, it is worth investing a few extra CPU cycles in a replacement policy that helps to save disk and network accesses.

5.3 GENETIC ALGORITHM REPLACEMENT POLICY (*GAR*)

The essential ingredients of the Genetic algorithm are population size (constant), string coding, fitness function, crossover, mutation and the number of generations. The standard genetic algorithm consists of an initial population chosen randomly. Every member of the population is coded into a unique string and has a definite fitness value attached to it. The fitness function used is dependent on the problem in hand.

This initial population is now made to go through the process of selection wherein members produce copies of themselves depending on their fitness value. Then the strings are made to go through the process of crossover and then mutated to finally end up with a new generation. The initial population set is now replaced with the new generation obtained and the whole procedure is repeated for a certain number of generations. The appropriateness of the fitness function is critical since this holds the key to members with low fitness values being eliminated in every generation. The selection process mimics the "Survival of the Fittest" theory. Also it is to be noted that the size of the population remains constant throughout the run of the algorithm in every phase. Genetic Algorithm is used because of two main reasons. First, the algorithm in its natural form itself identifies members of the population with low fitness and eliminates them. This fits into

caching and replication of internet data. This can be very well extended to replacement, as our objective in replacement is to find out objects with low fitness (popularity). Second, Genetic Algorithm is applied to problems demanding optimization out of spaces, which are too large to be exhaustively searched. A typical cache consists of millions of web objects. Conventional algorithms would require that we search through the entire space thereby taking more time for replacement. Genetic Algorithm works with just a sample population and typically the population size is far less compared to the size of the actual search space.

In our approach, the cache has been modeled as follows to fit in the usage of Genetic Algorithm. The cache can be visualized to consist of a set of individuals. This set is the complete search space. Each member of the cache is a specific web object and has a fitness value associated with it. This fitness is directly proportional to its popularity. The replacement algorithm is all about identifying the individuals (web objects) with low fitness (popularity) so that they could be evicted from the cache.

5.3.1 FITNESS CALCULATION

This is the most critical calculation upon which hinges the performance of the algorithm. We maintain a standard array, which stores the fitness of all the individual web objects in the cache. Every time a request for an object in the cache is made, its corresponding fitness value is incremented. If an object is absent in the cache, then it is brought in to the cache and its fitness value is initialized to zero. Hence at any point of time, this *fitness* array will contain the number of times every object in the cache has been accessed. Hence it would be the exact indicator of the popularity of the web object.

5.3.2 PSEUDO CODE

The Genetic Algorithm replacement scheme looks essentially as follows.

```
//definitions
old-pop, new-pop, popsize, fitness, noofgen, prob-cross, prob-mut

//pseudo-code
old-pop = initpop (popsize) // initialize a population randomly and assign it to old-pop
gen ← 1 //temporary variable
while (gen <= noofgen)
do
{
//selects a population of size popsize based on the fitness values
        parents = select (old-pop, fitness)
        crossover (parents, new-pop, prob-cross)
        mutation (new-pop, prob-mut)
        old-pop ← new-pop
        gen ← gen + 1
}
```

In the above mentioned replacement algorithm, *fitness* is an array which contains the fitness values of all the web objects in the cache. The *initpop( )* function randomly picks up *popsiz* number of individuals from the cache and stores it in *old-pop*. The function s*elect( )* ensures that only individuals with high fitness get selected. The unfit individuals are weeded out here.

5.3.3 IMPLEMENTATION

Every request from the trace was processed either locally or from the origin server. If the local cache did not have the requested file, then it is construed as a cache miss and the object is brought in and stored in the cache. Before bringing in a new object a check is made if the space is sufficient for accommodating the new object. If not, the replacement routine is called and objects with low popularity are evicted making way for the new object. Crossover was effected using arithmetic crossover. The probabilities for crossover and mutation were 0.8 and 0.05.

The performance analysis of the Genetic Algorithm based replacement policy has been carried out using the simulation of a trace provided by the logs from Duke University. This trace contains a day's log of all the HTTP requests to the EPA WWW server. In the simulation, the proposed replacement algorithm is compared with the traditional cache replacement algorithms viz. LRU, LFU and SLRU Algorithms. The performance metric used for comparison has been the hit-ratio. The cache sizes taken for simulation were in the order of a few megabytes (MB). This is far less than the total cache sizes available for the proxy caches. But when cache-on-demand scheme is used [Ahuja 2002], where a single user can get a cache space on demand for the time he wants to browse the net it is assumed that it can only be in the range of a few megabytes.

5.3.4 DISCUSSION OF RESULTS

The results of the simulation are depicted in the graphs in Figs 5.14 to 5.16. The salient features of the results are:

- The Genetic Algorithm based Cache Replacement Policy (GAR) has a higher hit-ratio compared to LRU and LFU.
- SLRU has a superior hit-ratio than GAR for lower cache sizes. For cache-sizes up to 2MB, SLRU performs better than GAR. Beyond this GAR scores over SLRU.
- The hit-ratio of GAR has been found to be between 0.6 and 0.8 which can be construed as an indication of the reliable performance of the algorithm absolutely.

Hence it can be concluded that GAR can be used beyond a certain threshold value of cache size. As the cache size increases, the performance of the algorithm increases. Also an important parameter to be varied for the algorithm is the population size. It has been clearly noted during simulation that as the population size increases, the hit-ratio also increases.

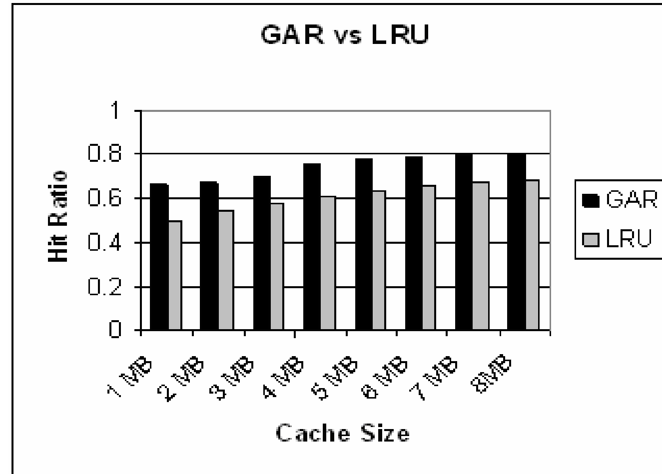


Fig. 5.14 Hit Ratio of GAR and LRU for varying cache sizes

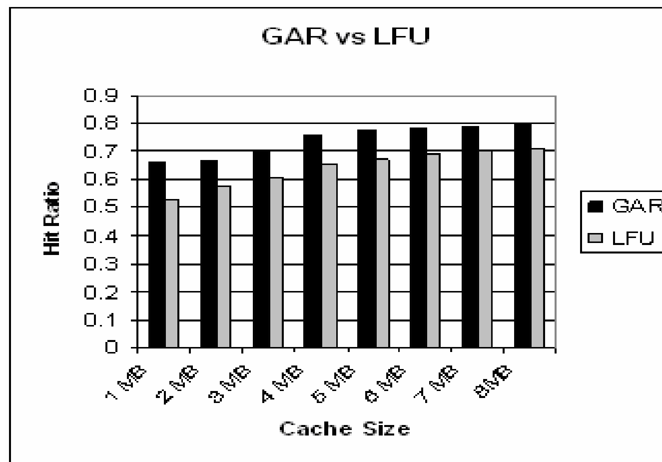


Fig. 5.15 Hit Ratio of GAR and LFU for varying cache sizes

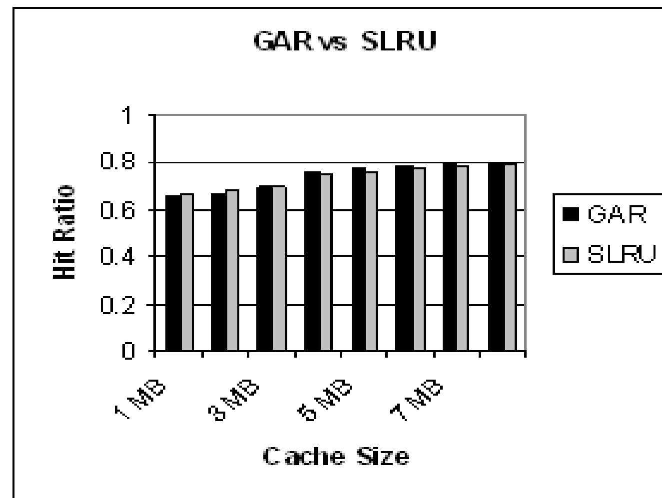


Fig. 5.16 Hit Ratio of GAR and SLRU for varying cache sizes

A study was also done comparing the time taken for the running of the replacement algorithm proposed with the Least Frequently Used Algorithm. It has been observed that as the number of objects in the cache increases, the time taken for the run of GAR is lower compared to LFU. Hence it is advantageous to use GAR especially when the cache size is large and there are a large number of objects to consider before picking a victim page. Another point is that the time taken for the run of GAR is not very dependent on the cache size. Since Genetic Algorithms deal only with a fixed population at a time, the size of the cache is immaterial. This fact can be exploited if GAR is deployed for really huge caches, with a large number of files to search from. The graph regarding the time analysis, is shown in Fig. 5.17.

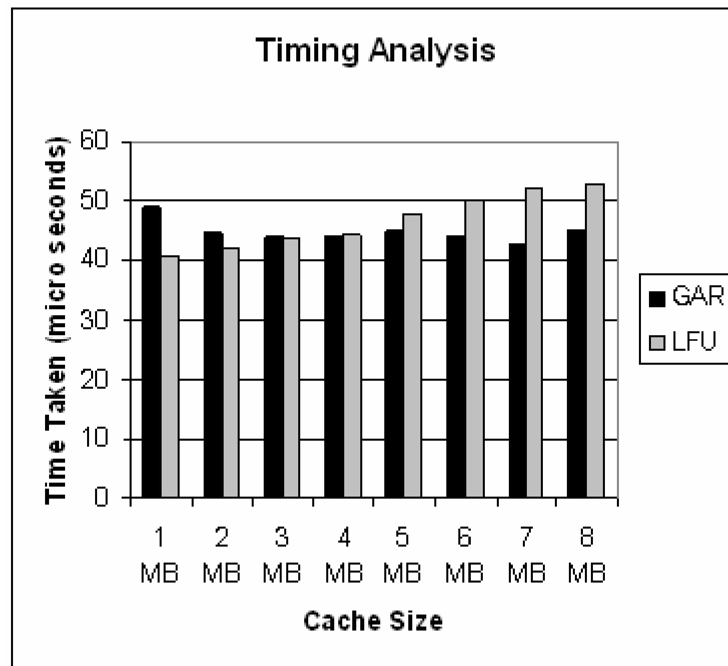


Fig. 5.17 Time taken vs Cache Size for the GAR algorithm

5.4 GENETIC ALGORITHM BASED REPLACEMENT POLICY FOR STREAMING MULTIMEDIA OBJECTS (*GAR-M*)

Replacement is all about identifying the "correct" frames to remove thereby making way for newer ones to be cached. Here, in our approach every frame in the video is handled and evaluated separately. A fitness value for every frame is calculated and the replacement is done of the less fit frames and not a whole video as such.

The cache is viewed as a set of frames. Every frame in the set is a member of one video file. This whole set is our search space. The replacement policy aims to identify and evict individuals (frames) with low fitness values.

5.4.1 FITNESS CALCULATION

The fitness of every frame is dependent of three factors, individual popularity, position and popularity of parent video. They are discussed below:

Individual popularity

Consistent with the frame-wise handling of the videos, every frame is monitored and assigned an individual "hit" value. This value is an indication of the number of times the particular frame was accessed when it was in the cache. Every frame collects some points whenever it is accessed while in the cache.

Position in the parent video

The position of a frame in the video that it is a part of is critical for assigning a fitness value. Frames, which occur earlier, have more weightage compared to the later frames in a video. This is because, when a user requests for a video, there is a very high probability that he would start viewing the earlier frames first. In the event of them being absent, the proxy has to fetch it from the origin server, thereby consuming more time. Hence absence of the earlier frames would severely lower the Quality of Service. If the later frames are absent, the proxy can fetch them from the origin server while it is sending the earlier

frames to the user. In the algorithm proposed, every time a frame in the cache is "hit" it collects points in proportion to its position.

- If a frame belongs to the first 20% of the file, it would pick up 10 points.
- If a frame belongs to the second 20% of the file, it would pick up 8 points.
- If a frame belongs to the third 20% of the file, it would pick up 6 points.
- If a frame belongs to the fourth 20% of the file, it would pick up 4 points.
- If a frame belongs to the last 20% of the file, it would pick up 2 points.

Popularity of the parent video

The fitness value of a frame is also governed by the popularity of the video that it is a part of. Frames with very low individual popularity will have their fitness shored up if the video that it is a part of is very popular. This is due to the fact that since the video is very popular, there is a high probability for requisition of any of its frames. Hence, fetching any of these frames from the origin server would reduce the quality of service, especially in the context that the video is to be requested multiple times.

This way, the number of points garnered by every frame (p_f) would be an indication of both its popularity as well as its position.

Let v_i be the popularity of the video, *i*.

$$v_i = \sum p_f \quad \text{where } p_f \text{ is the points garnered by frame, } f$$

for all the frames in the video *i*.

So, the 'fitness' f_f for any frame *f* would be,

$$f_f = w \times v_i + (1 - w) \times p_f \quad \text{where } 0 \le w \le 1 \text{ is the weightage of the video.}$$

5.4.2 IMPLEMENTATION

For evaluating the performance, the simulation has a proxy server which receives the requests for video objects from the client. The object is a video file, taken from the benchmark videos. The video is sent to the client frame by frame.

The origin server handles the request from the proxy server in case of a cache miss. If the video requested by the user is not present in the cache, then request is forwarded to the origin server. The origin server sends the video frame sizes to the proxy server. In case of a cache miss the replacement algorithm is run. Hit Ratio is calculated for each client request, be it a hit or a miss.

The performance analysis of the Genetic Algorithm based replacement policy has been carried out using benchmark videos obtained from [Video 1995]. The video files obtained were diverse enough to cover various categories like movies, sports, news, and animations. This was chosen so as to make the testing process comprehensive and also test the robustness of the algorithm. The files had approximately 40000 frames each, on an average. Every video file included size information per frame.

After every request, the individual fitness values of the relevant frames were updated. In the simulation, the proposed replacement algorithm was compared with the traditional cache replacement algorithms viz. LRU and LFU Algorithms. The performance metric used for comparison has been the hit-ratio. The testing was done for two different sample requests. Each of the request patterns had around 100 requests each. Hence the results obtained can be assumed as an indication of the performance of the algorithm over a sustained period of time. The hit-ratios were analyzed by varying the cache sizes as a percentage of the total possible size of all the files taken for simulation. The algorithm was tested for cache sizes, ranging from 20% to 80% of the total size of all the files put together.

5.4.3 DISCUSSION OF RESULTS

In the testing of the algorithm, it has been observed that the Genetic Algorithm based replacement policy gives a better hit-ratio compared to the standard replacement algorithms like LRU and LFU. The improvement in performance over LRU and LFU has been observed to increase for higher cache sizes. Hence it can be concluded that the algorithm is suited especially for high-end servers, which deal with a large number of

cache files. Also the hit-ratio of the algorithm hovers between 0.6 and 0.8, which can be construed as an absolute indication of the robustness and efficiency of the algorithm.

The graphs obtained from the simulation are as shown in Fig 5.18 and Fig. 5.19 are the results obtained by running the algorithm for the two sample requests mentioned earlier.

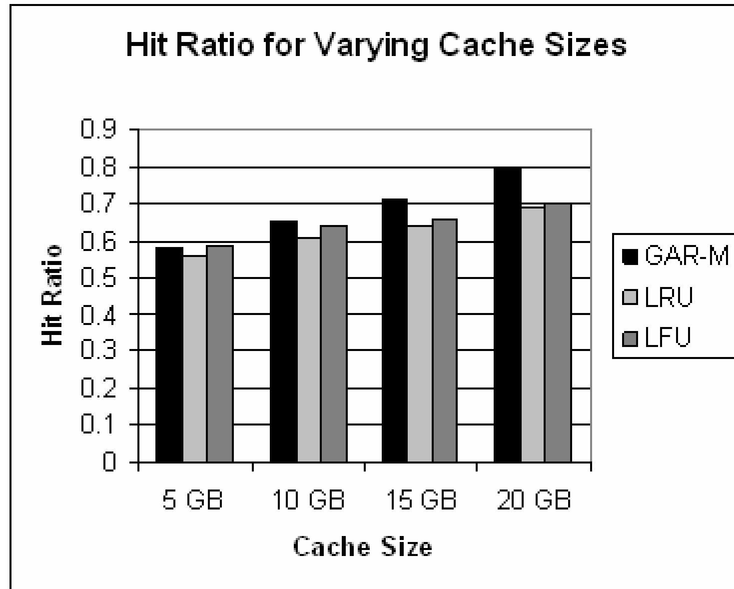


Fig 5.18: Hit Ratio of GAR-M, LRU and LFU for varying cache sizes for Sample-1

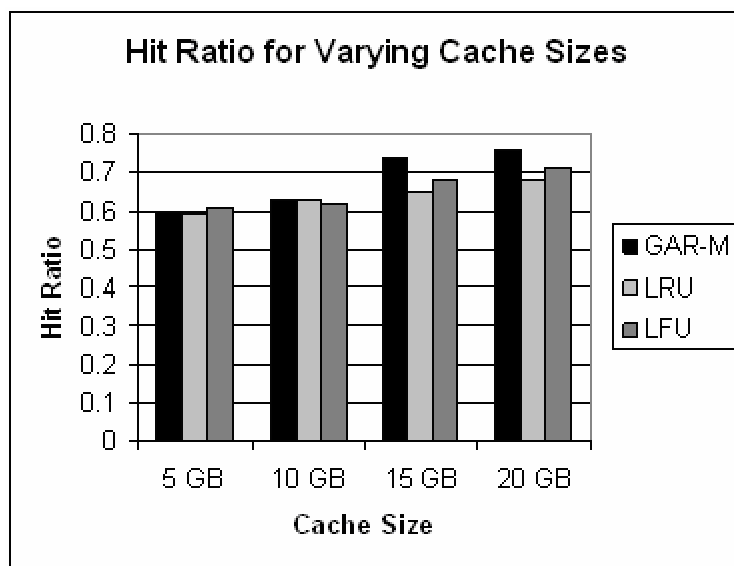


Fig 5.19: Hit Ratio of GAR-M, LRU and LFU for varying cache sizes for Sample-2

5.5 SUMMARY

The study of the soft computing techniques used in web caching is in its early stages. In this chapter an attempt has been made to use the Fuzzy logic and Genetic algorithms to improve the performance of web caching. The result obtained clearly indicates the advantage of using these techniques over conventional techniques. The implementation initially considered static web objects and was later extended to multimedia objects. It is observed that these algorithms have an edge over other, specifically as the proxy cache size and number of requests increase.

CHAPTER 6

CACHING IN MOBILE NETWORKS

6.1 INTRODUCTION

Accessing the World Wide Web data by using mobile devices is increasing due to the deployment of 2.5G and 3G services. The growth of wireless networks has made the wireless web applications more popular and sophisticated. Due to the strong demand for bringing web applications into wireless environments, much effort has been made to consolidate the WWW with wireless networks [Baquero 19995, Housel 1996]. Such integration is also referred to as W4 – World Wide Web for Wireless. Figure 6.1 depicts a typical W4 architecture.

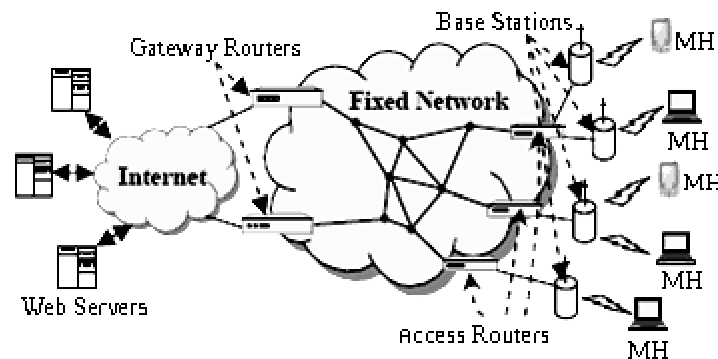


Figure 6.1: World Wide Web for Wireless Network Architecture

In this architecture, mobile hosts access the wireless network through base stations, which are inter-connected by access routers to form wireless LANs, and in turn are connected to the Internet through gateway routers. Among the numerous studies carried out on the enhancement of wireless internet performance, caching popular web data at locations close to the mobile clients is an effective solution to improving the quality of wireless web applications.

A mobile user's web access is largely determined by the user specific preferences and the presentation of data is constrained by the capabilities of the device used. To understand

Caching in wireless environment it is imperative to understand the key aspects involved in such networks and devices, specially their limitations over those that work in wired and fixed networks. The most important issue is the bandwidth in wireless networks. Bandwidth of wireless links that connect mobile units to the fixed networks is very limited and therefore is a major performance bottleneck. While a wireless LAN bandwidth is in the order of 11-50 Mbps, the typical bandwidth of wireless cellular systems is in the order of 10-55 Kbps.

User access devices such as PDAs (Personal Digital Assistants) or mobile phones have significantly smaller displays, slower processors and smaller memories, than more traditional end-user computers such as workstations and PCs. Given the trend towards wider variety and higher level of integration such as digital audio and video players in mobile units, the differences in power and expected capabilities will continue to grow larger. Importantly, Internet services are being integrated into mobile devices adding more constraints. This issue also affects content-providers and application programmers, as their model of the machine that allows the end-user to access content and run applications is no longer simple. That these devices already or will eventually connect to the Internet via wireless links introduces further complexity [Forman 1994, Satyanarayanan 1996, Shankarnarayanan 2002].

Disconnection is another distinguishing feature. Users may turn off their mobile computers to save battery energy, which is called voluntary foreseeable disconnection. Disconnection can be unpredictable too, as a result of wireless network failure. Wireless links have significantly different properties regarding performance, reliability, and security than the more prevalent wired links with which the Internet evolved. Wireless links generally have lower bandwidth and much higher error rates. Outages that result in disconnections are common and the ability to eavesdrop is qualitatively easier when compared to tapping a wired link [Parker 1998]

Bringing Internet connectivity to wireless devices offers a number of challenges. Basically, the Internet assumes powerful endpoints. The Internet expects endpoints to

carry out significant control functions, and this simplifies the internals of the Internet leading to increased efficiency and lowered costs. This implies that a user "pays" (in one way or another) only for what they need; as long as the Internet delivers packets, it is up to the endpoint to enhance this most basic service with such properties as in-order delivery, reliable transmission, and flow control, to name a few. Simply delivering bits at a high enough rate to untethered devices and creating devices with sufficient display, memory, and user interface features to allow efficient interaction with the Internet requires innovative technologies and new standards. [Saha 2001, Pasquale 2002].

6.2 INTELLIGENT PROXY SERVER WITH CACHE-ON-DEMAND PROTOCOL

Users are interested in high level services like e-commerce, with uninterrupted access and faster response time. In general, the usage of wireless access can be described by mobility scenario which is termed as true mobile access. Terminals can be moved within and between the range of multiple access points or base stations. Dynamic changes of the supporting access points or base stations during a session are expected to appear. Such changes are called handovers. For mobile access to be attractive, deployment of access points should be dense. The degree of service continuity in spite of handovers is one of the essential quality features. Continuity of service might be expressed in terms of the amount of information during handover. The case of frequent, possibly interruption-less handover usually implies a homogeneous system concept in which all the access points and the end-system are incorporated. This is the scenario for the majority of solutions deployed or considered today, like GSM, GPRS or the emerging UMTS [Wolisz 2000a, Wolisz 2000b].

Cache-On-Demand (COD), as explained earlier, is a protocol for web caching in a fixed network environment, which allows a proxy cache sever to allocate its local storage resources upon external requests from either content providers or clients themselves, and thus provides quality of service (QoS) in delivering content to users. The advantage to the content providers is QoS guarantees like fresh content being available to a web user from a Cache-On-Demand-enabled web cache. Advantage to the clients being reduced latency and, thus, better user experience.

The following sections discuss the Intelligent Proxy Server combined with the deployment of CoD Protocol.

6.2.1 LATENCY REDUCTION SCHEMES

Prefetching is one of the common approaches used to reduce network latency. When a user is idle, web pages can be prefetched from the remote sites ahead of their actual time of use. In case the user really requests those prefetched web pages, the requests can be satisfied by the local cache instead of the far away remote systems. This gives a much shorter user perceived latency. The criteria for deciding whether a web page should be prefetched can be *statistical* (the recent access logs can be used as a basic parameter of statistical prefetching decisions) or *deterministic* (just a set of user-predefined web pages are prefetched).

A dynamically constructed web page might constantly change, thus, any form of caching and prefetching is prohibited for such pages. Delta encoding reduces this problem by only sending the portion of the binary file, which changed since the last version stored on the proxy side. HTML Pre-Processing (HPP) is an HTML extension, which distinguishes a static and a dynamic portion [Douglis 1997]. While the static portion can be cached, the dynamic portion is generated for each request. Since a large portion of dynamic web pages is static, such a scheme can alleviate the perceived latency.

Here, Client Side Prefetching is used as it gives the user the control of the prefetch process. Advantages of this technique are that it does not increase network traffic, attempts to improve on all parts of latency, can be implemented on the client side, without the cooperation of any other tier and can work seamlessly with any other latency reduction technique [Eden 2000].

6.2.2. ISSUES IN MOBILE COMMUNICATION ENVIRONMENT

The unique characteristics of the mobile communication systems are as follows:

- In mobile communication systems, the scope of web access is relatively limited as compared to fixed network. Due to the limitations in processing power, size, and computational speed of the mobile devices and due to high communication cost, mobile user requests are only for a small amount of frequently accessed information from the Internet (e.g. travel information, financial report, weather report, daily news reports etc). These can be prefetched and put into the local disk, giving a higher cache hit ratio. No complex prediction algorithm is needed.
- Before a mobile station connects to a mobile network system, it must register itself to the system. The information about the station and the user can then be known by the system. This is the time to initialize the prefetching process.
- Transcoding is usually needed in mobile applications to convert an object from one format to another so as to present the object in a scale down format, see [Han 1998, Wong 2001]. However, transcoding spends system resources and causes system delay. If the system can transcode the prefetched objects before the user actually requests them, the user will experience a much shorter delay.

Based on these characteristics, a suitable prefetching scheme for the mobile applications needs to be designed.

6.2.3. AN INTELLIGENT PROXY SERVER (IPS)

Studies on prefetching show that most of the research work has been on statistical prefetching [Jiang 1998, Markatos 1998]. But an important observation is that deterministic prefetching is more suitable for mobile communication. It gives little or no bandwidth overhead because it is configured statically by the users. When the user is accessing the Internet by a mobile device, he accesses a limited set of web pages (unlike the case when he is using desktop or notebook PCs via the fixed network). The reason is due to the high communication cost and limited capacity of mobile devices, prefetching a set of user-predefined web pages is enough in most cases. This kind of prefetching not only gives higher prefetching hit rate, but is also simpler than the statistical approach.

The Intelligent Proxy System proposed by Yeung et. al [Yeung 2003] is based on unique characteristics of mobile communication system as discussed in the section 6.2.2.

The proposed prefetching scheme comprises of two basic components:

- The User Profile Database (UPD) and
- The Intelligent Proxy Server (IPS).

The scheme followed is:

- When a user subscribes the mobile service from an operator, his profile (i.e. URLs of his favorite web pages) will be created and stored in the User Profile Database (UPD).
- When a mobile device registers to the mobile network, the registration information will be recorded in the Home Location Register (HLR).
- The HLR registration information is then sent to the Intelligent Proxy Server (IPS).
- The IPS then prefetches the web pages for the user based on the information of the user profile in the UPD.
- Based on the preference of the user, the prefetched web pages can be stored in the IPS for later access or immediately sent to the mobile device of that user.
- Note that the HLR may already record the type of device being used by the user, thus, transcoding on the prefetched web pages can be performed immediately if needed.

The proposed prefetching scheme is built on top of the existing mobile communication infrastructure. As shown in Fig. 6.2, the IPS works as an add-on component of the Internet gateway of a mobile network. The UPD, on the other hand, can be viewed as an add-on to the HLR. Thus the scheme can be smoothly implemented in the existing mobile networks.

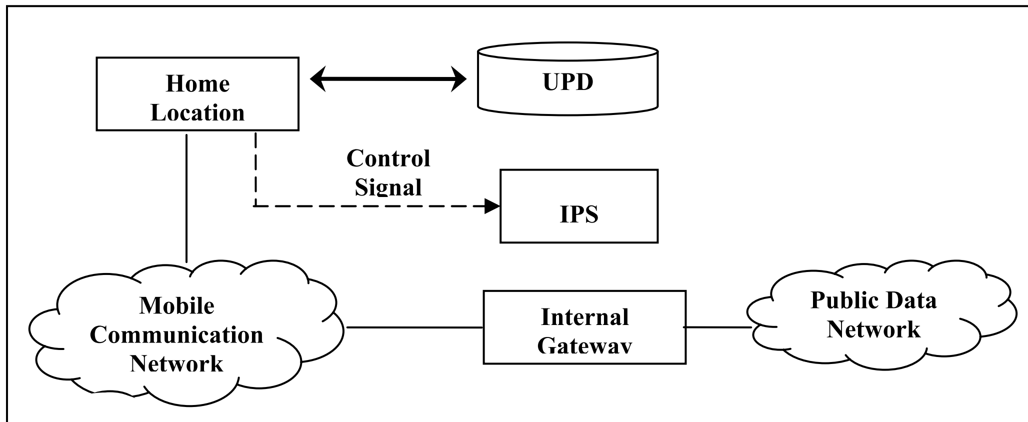


Fig 6.2 Overall architecture of the IPS system

6.2.3.1. User Profile Database

User Profile Database (UPD) is a database which stores user profiles. Each user profile consists of two fields:

- International Mobile Station Subscriber Identity (IMSI) of mobile device: a unique identifier of the mobile device.
- A list of URLs that are chosen by the users. Users can update their profiles by any means as provided by the operator. Sample contents of a UPD are as shown below:

IMSI_A http://www.sie1.com/
http://www.sie2.com/page1.htm
http://www.site3.com/page1.cfm

IMSI_B http://www.site2.com/page2.htm
http://www.site4.com/page1.asp

6.2.3.2. Intelligent Proxy Server

As explained earlier, the IPS accesses the user profiles stored in the UPD when the user registers to the mobile network. The accessed information is then used by the prefetching process to retrieve the objects.

There are two modules in the Intelligent Proxy Server (IPS) as shown in Fig 6.3.

The Caching Module

This module works as a traditional proxy server. When an Internet access request arrives (1)], the caching module will first check whether the local cache can satisfy the request(hit) or not(miss) (2). If the request is a hit, the requested page will be returned to the user (3). Otherwise, the requested page will be retrieved from the remote site (4), (5) and stored into the local cache (6). This completes a normal request process.

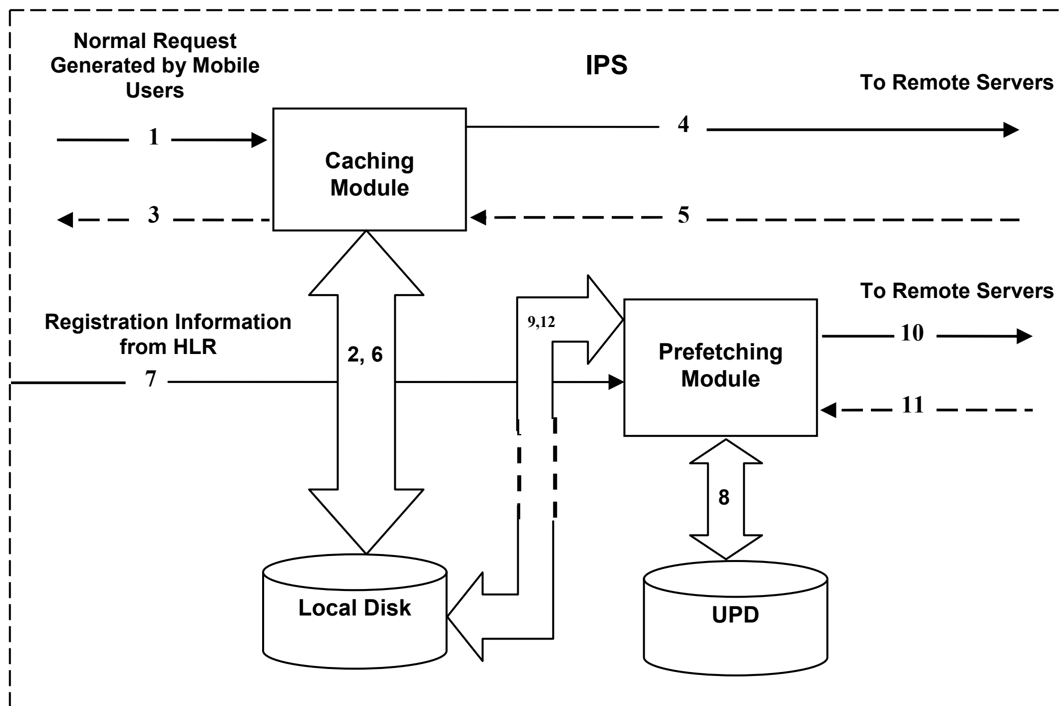


Fig 6.3 Architecture of the IPS

The Prefetching Module

When a user registers to a mobile network, the registration information including the *IMSI* of the user device will be sent from the HLR to the prefetching module (7). The module starts the prefetching process by looking up the registered profile of the user in the UPD (8). It then checks whether the pages are cached in the local cache or not (and whether the cached ones are updated ones) (9). For those pages that are not cached, the prefetching module requests them from the remote web sites (10, 11). The prefetched pages are then put into the local cache (12). This completes a prefetch request process.

We can see that the caching module and the prefetching module generate normal requests and prefetching requests respectively. Since the normal requests are more important than the prefetching requests, they should have a higher priority to be served first.

An example showing how the proposed prefetching scheme can be implemented in a General Packet Radio Service (GPRS) network is given in Fig. 6.4.

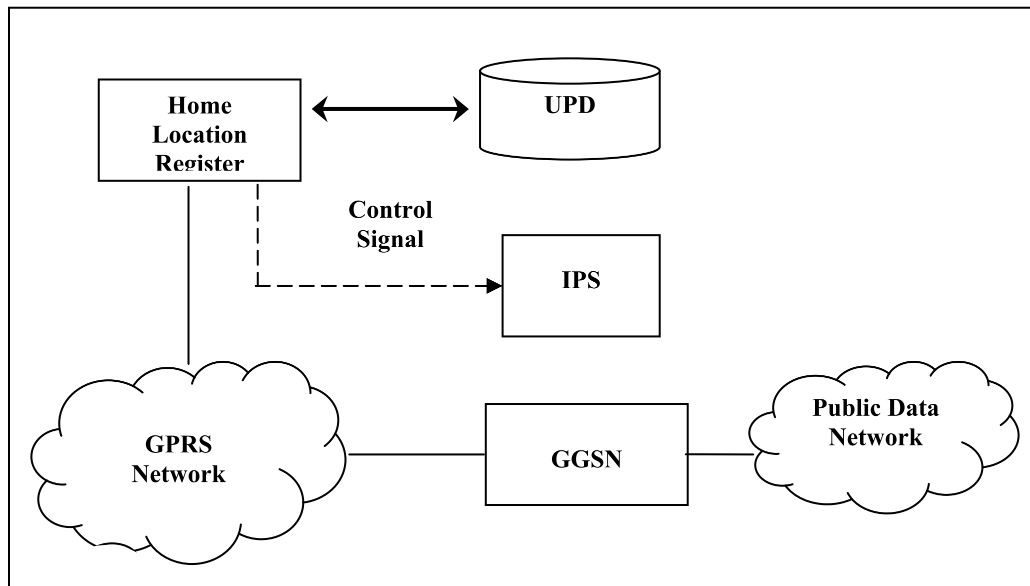


Fig 6.4 The GPRS Network with the proposed prefetching scheme

Before a mobile station can use the GPRS services, it must register to a Serving GPRS Support Node (SGSN) of a GPRS network. This procedure is called GPRS attach. The register information will then be sent to a Gateway GPRS Support Node (GGSN). GGSN is the Internet gateway between the GPRS backbone network and the external packet data network. Fig. 6.4 shows how the IPS and UPD works with the SGSN and the GGSN.

6.2.4. CACHE-ON-DEMAND PROTOCOL

The Cache-On-Demand protocol supports strong consistency by giving complete content management control to the content provider. The Cache-On-Demand client can reserve resources for a specified duration of time and pull required contents to the cache. It can explicitly update the cached content in order to maintain strong consistency between the

original and the cached copies of the content. It can request the Cache-On-Demand-enabled cache to invalidate the content and free up the reserved resources. Cache-On-Demand can be implemented as a paid service model. Clients are charged for the amount of resources reserved by them for the duration of the reservation.

6.2.5. THE IPS-COD COMBINED PROTOCOL

The IPS scheme lends us the concept of prefetching of web content based on the User's profile which may be specified by the client himself or may be gauged from the client's past access history.

The COD protocol testifies that the above would reduce the bandwidth usage and latency in servicing client requests as also the load placed on Origin Servers and the congestion in the mobile network. In addition, it indicates that redundant usage of storage space would be reduced if the web content is prefetched and maintained only for a requested period of time and for those clients alone, who request for the prefetching facility.

Leaving the option of using the prefetch facility open to the clients would also save the clients who do not use the facility from being unnecessarily charged for the same. Only interested clients use the facility and pay for the same. Building on the concepts from these protocols, we have implemented a new scheme which capitalizes on the advantages of both.

In this protocol, we assume that the Proxy Server is the Base Station. Starting from registration of the client with the mobile network to servicing of its requests by the Server, the protocol works as follows:

1. When a user registers to a mobile network, the registration information including the *IMSI* of the user device will be sent from the Home Location Register (HLR) to the Proxy.

2. The client communicates to the HLR whether it wants to use the prefetching facility or not.
3. If client does not wish to use the facility, then goto step 8.
4. If client communicates in the affirmative, then the Proxy starts the prefetching process by looking up the registered profile of the user in the UPD.
5. If the profile of the client is not registered in the UPD, then the same is stored by the client and then the prefetching process begins. When the client registers, then the profile would be available for the mobile network.
6. Proxy then checks whether the pages are already cached in its local cache (and whether the cached ones are updated copies).
7. For those pages that are not cached/ not updated, the Proxy Server requests them from the Origin Server. The prefetched pages are then put into the local cache. This completes a prefetch request process.
8. When an Internet access request arrives from the client, the Proxy first checks whether its local cache can satisfy the request or not.
9. If so, the requested page will be directly returned to the user. Otherwise (i.e. if the page is either not cached or cached but not updated), the requested page will be retrieved from the Origin Server and stored into the local cache before servicing the request.

This protocol design is implemented and verified. The working of the protocol is depicted in Fig 6.5 and Fig 6.6

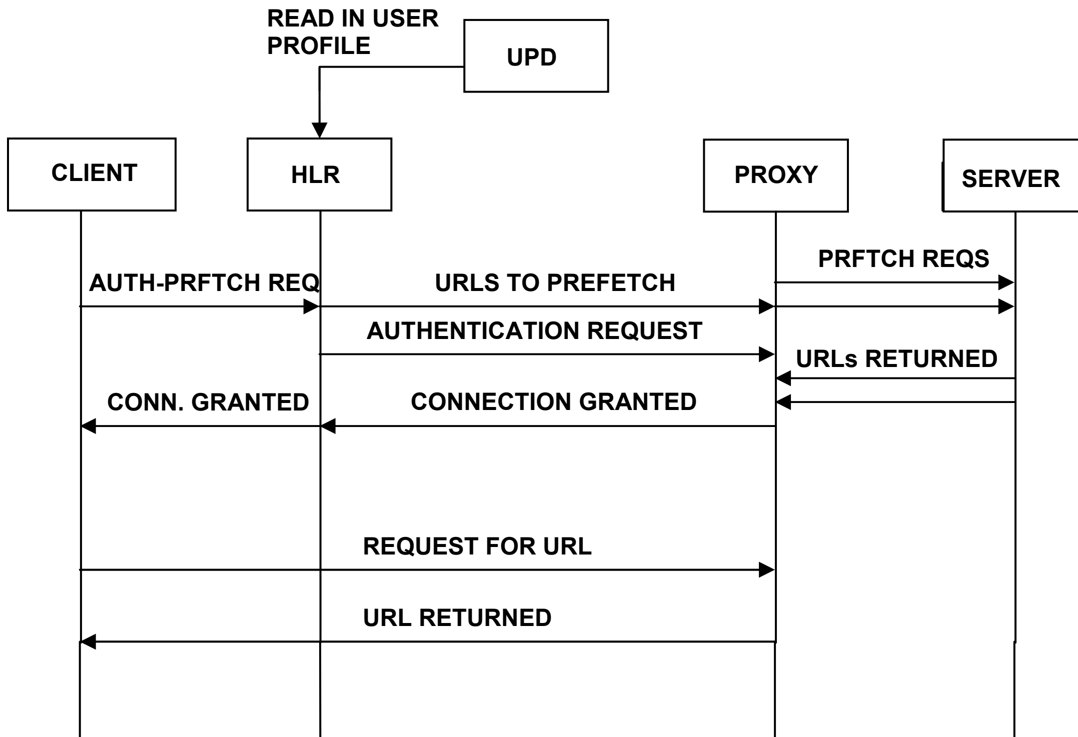


Fig 6.5 Protocol Sequence: Client requests for prefetching facility

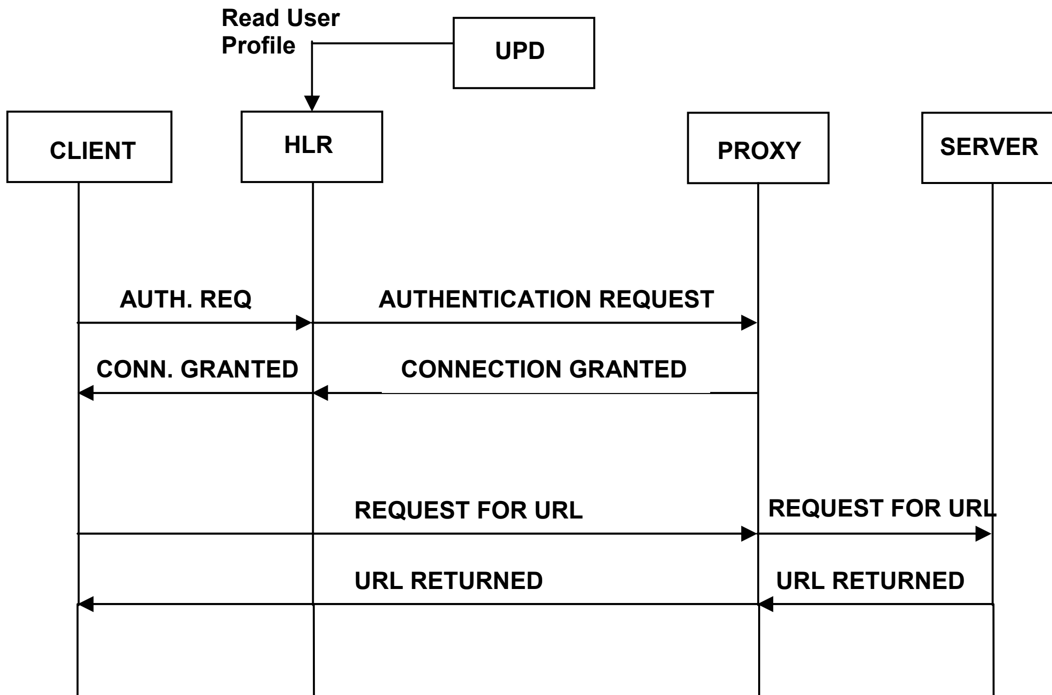


Fig 6.6 Protocol Sequence: Client does not request for prefetching facility

6.2.6. DISCUSSION OF RESULTS

To evaluate the performance of the CoD protocol in a mobile environment the performance of CoD is measured on a smaller cache size of 100K and 1000K. This cache size, will be easier to handle even when the mobile user is on the move. The factors that are considered for performance evaluation are; hit ratio, percentage of the normal cache for the CoD case and number of Requests by different users.

We consider two implementations. The normal proxy server implementation that caches all the requests from the clients and the CoD implementation of the proxy server. The whole process is simulated where the server generates a random size for the objects being requested for and then based on the caching strategy the objects are cached. The log file for the first case discussed above just consists of normal http requests where as the log file for the second case consists of both CoD and the normal http requests distinguished by the keyword 'COD' at the beginning of the requests. Now both the programs are run varying some of the parameters and then plotting the graphs.

In the normal proxy server implementation the hit ratio has been plotted for all the requests made. Fig 6.7 shows the variation of hit ratio with the number of requests for different cache sizes. The no of requests are on the X-axis and the hit ratio on the Y-axis. The black line shows the variation for a cache size of 100k and the grey line shows the variation for a cache size of 1000k. As expected the hit ratio for the 1000k cache is higher than that for the 100k cache.

In the CoD implementation, the CoD cache size reserved is expressed as the percentage of the normal cache. Fig 6.8 shows the variation of the hit ratio with the percentage of normal cache reserved. The percentage of the normal cache reserved is on the X-axis and the hit ratio on the Y-axis. The graphs have been plotted after satisfying 10000 requests from the client.

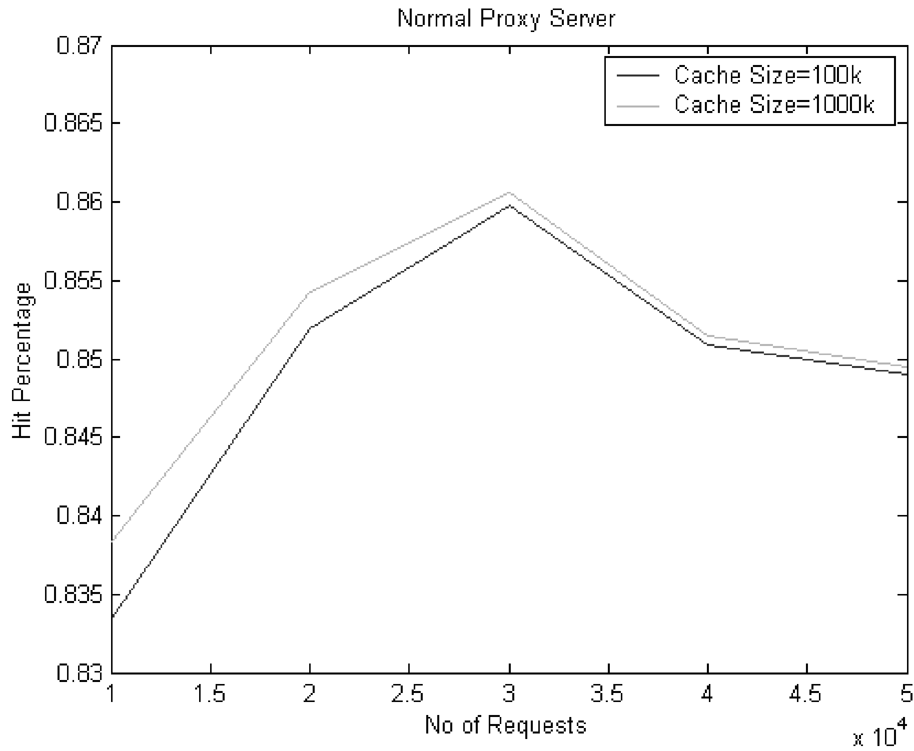


Fig 6.7 No. of requests vs hit ratio for normal cache in an IPS-CoD system

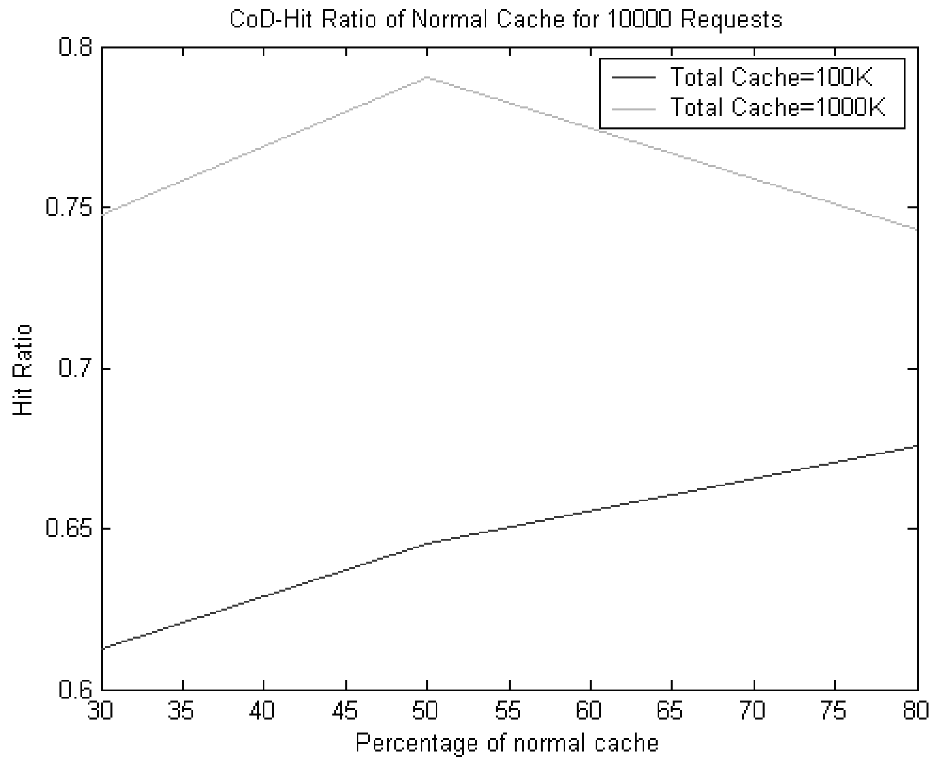


Fig 6.8 Percentage of normal cache vs. Hit ratio in an IPS-CoD system

This experimentation is to show the basic advantage of integrating the mobile network with the Cache-on-Demand protocol. Analyzing the results obtained, we can see that for smaller size of caches, the CoD protocol will help the user with a better Quality of Service and can be offered as a service by the Content Delivery vendors.

This problem gets escalated in a mobile communication environment which suffers from limitations in processing power, size, and computational speed of the mobile devices in addition to high communication cost. Due to these limitations, as mobile users request for frequently accessed information from the Internet the web content is prefetched based on the user's profile and stored in the proxy cache. This results a higher cache hit ratio, reduces the bandwidth usage and latency in servicing client requests. In addition, it also minimizes the congestion in the mobile network and the load placed on Origin Servers.

Here, the idea of using a novel but simple prefetching scheme with cache-on-demand protocol in mobile communication systems is discussed. The scheme was designed based on three major observations on the characteristics of mobile networks, and is therefore well suited to today's mobile applications. Simulation model on the scheme is built to study the system performance verification.

6.3 DYNAMIC CACHE INVALIDATION SCHEME

Caching of frequently accessed data at the mobile clients has been considered to be a very effective mechanism in reducing wireless bandwidth requirements as well as energy consumption, since no energy is expended to transmit and receive data. For caching to be effective, the cache content must be consistent with those stored in the server. This is difficult to enforce due to the frequent disconnection and mobility of clients. The basic approach adopted is for the server to periodically broadcast invalidation reports that contain information about objects that have been updated recently [Barbara 1994, Jing 1999]. Based on the report, clients can invalidate objects that have been updated and salvage their cache content that are still valid. Most of the existing algorithms address three issues. The first issue deals with the content of the invalidation reports. The second issue concerns how invalidation is performed. The third issue looks at the support the

server provides. The model for a mobile data access system adopted is as shown in Fig.6.9.

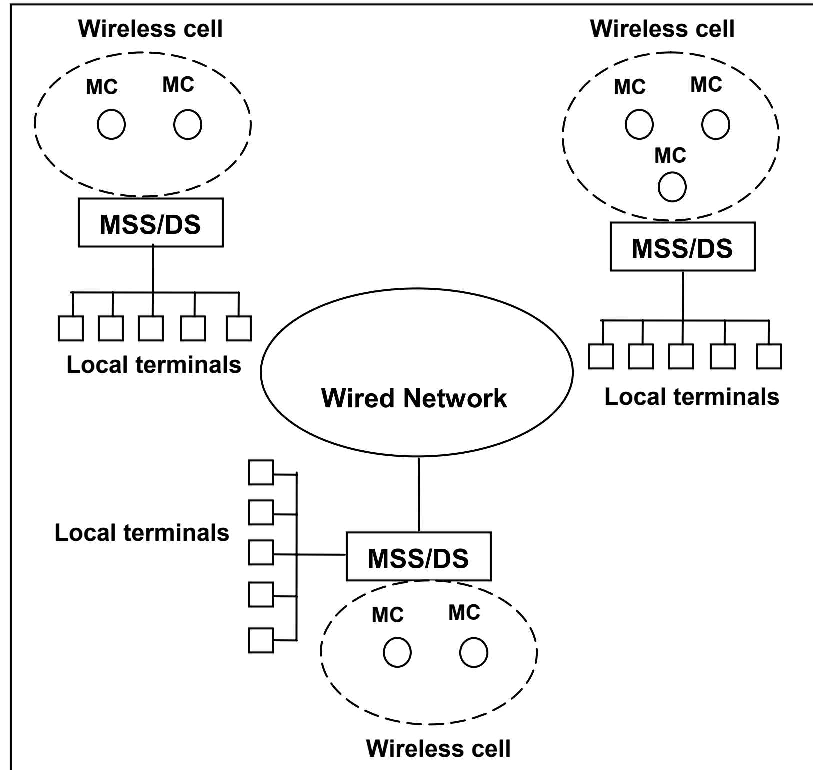


Fig 6.9 Wireless Computing Environment

The mobile environment consists of two distinct sets of entities, a larger number of mobile clients (MC) and relatively fewer, but more powerful, fixed hosts called mobile support stations (MSS) or database servers (DS). The fixed hosts are connected through a wired network and may also be serving local terminals. Some of the fixed hosts, like MSS, are equipped with wireless communication capability. An MC can connect to a server through a wireless communication channel. It can disconnect from the server by operating in a sleep mode or a power-off mode. Each MSS can communicate with MCs that are within its radio coverage area called a wireless cell. A wireless cell can either be a cellular connection or a wireless local area network. At any time, an MC can be associated with only one MSS and is considered to be local to that MSS. An MC can directly communicate with an MSS if the mobile client is physically located within the

cell serviced by the MSS. An MC can move from one cell to another. The servers manage service on-demand requests from mobile clients. Based on the requests, the objects are retrieved and sent via the wireless channel to the mobile clients. The wireless channel is logically separated into two sub-channels, an uplink channel which is used by clients to submit queries to the server via MSS, and a downlink channel which is used by MSS to pass the replies from the server to the intended clients. We assume that updates only occur at the server and mobile clients only read the data. To conserve energy and minimize channel contention, each MC caches its frequently accessed objects in its nonvolatile memory such as a hard disk. Thus, after a long disconnection, the content of the cache can still be retrieved. To ensure cache coherency, each server periodically broadcasts invalidation reports. All active mobile clients listen to the reports and invalidate their cache content accordingly. We assume that all queries are batched in a query list and are not processed until the MC has invalidated its cache with the most recent invalidation report. We assume that each server stores a copy of the database and broadcasts the same invalidation reports. In this way, clients moving from one cell to another will not be affected. Thus, it suffices for us to restrict our discussion to just one server and one cell. The following cache invalidation strategy is based on the model reported in [Tan 2001]

6.3.1 Taxonomy of Cache Invalidation Strategies

Two basic categories of cache invalidation strategies have been proposed in the literature. They are:

- Stateful approach
- Stateless approach

Stateful Approach

In the stateful approach, the server knows the objects that are cached by the mobile clients. As such, whenever there is any update to the database, the server will send invalidation messages to the affected clients.

Stateless Approach

In the stateless approach, the server is not required to be aware of the state of the client's cache. Instead, the server broadcasts information on objects that are most recently updated and the clients will listen for and use the reports to invalidate their caches. The invalidation methods in stateless approach can be classified into asynchronous and synchronous methods. In the asynchronous method, once a record is updated, the server will broadcast updated value immediately. The asynchronous method is effective for always connected clients, and allows them to be notified immediately of updates. However, for a client which reconnects after a period of disconnection, the client has no idea of what has been updated and so the entirety of its cache content has to be invalidated. To salvage the cache content, Barbara and Imielinski [Barabara 1994] have proposed that an invalidation report can be piggybacked with each invalidation notice. In this case, upon reconnection, clients will have to wait for the first asynchronous invalidation report. However, since the report is sent asynchronously, there is no guarantee on how long the client must wait. On the contrary, the synchronous method is based on the periodic broadcasting of invalidation reports. The server keeps track of the records that are recently updated and broadcasts this information to clients periodically. Based on the report, a client determines whether its cache is valid for the query; if it is, it can be used to answer the query, and otherwise, the query may have to be submitted to the server. Because of its periodic broadcast nature, synchronous methods provide a bound on the waiting time of the next report.

The Cache invalidation schemes reported in the literature mostly fall into the stateless category. Some common issues that have been addressed in designing cache invalidation schemes are given below. These are the content of the report, the invalidation process, and the information (log) that the server must maintain.

6.3.2 Content of the Invalidation Report

Ideally, the server should keep track of all updates and broadcast them to the mobile clients. But, this is costly and impractical in view of the limited bandwidth and short battery life of mobile clients. Instead, the server maintains a short (of reasonable length) history of updates and broadcasts an update report (UR) that reflects the most recent changes. Several issues need to be addressed with regards to the content of the report.

The granularity of the report refers to the level of details of information each record of the report captures. A record in the report can be an ⟨id, TS⟩ pair, where id is the identifier of the object that is updated and TS is the timestamp at which this object is updated. Alternatively, the report can reflect the full detail of the object that is updated at time TS, i.e., the record is the pair ⟨object, TS⟩. The former is commonly known as update invalidation as clients can only invalidate their cache content. The latter, on the other hand, allows clients to immediately update their invalid copy with the object that is broadcast. It is thus referred to as an update propagation mechanism. Clearly, there is a tradeoff between the two mechanisms. Under update propagation, when the disconnection time is short, clients can update its cache immediately. Under update invalidation, clients must still submit requests to retrieve the updated records even if the disconnection time is short. However, under update propagation, since the entire record is broadcast, the report is much larger and can take up a significant portion of the downlink channel capacity, which is a scarce resource in wireless environment. Moreover, given the same report size, update invalidation can afford to reflect a longer history of updates.

The size of the invalidation report can be fixed or varied. The update history refers to the history of the updates that are reflected in the report and can be fixed or varied too. These two factors are interrelated in the sense that one typically affects the other. It can also be fixed by the number of objects/groups to be included in the report. Obviously, under these cases, the update history cannot be predetermined (i.e., it has to be variable) since the number of updates varies over a fixed period of time. On the other hand, the report size can vary from broadcast to broadcast by fixing the update history being reflected.

To conserve energy, it may be necessary to organize the invalidation report to facilitate selective tuning. This can be done by interleaving the content of the report with “indexes” that can provide “direct” access to the targeted portion of the report. Thus, only the desired portion of the report needs to be examined.

6.3.3 Invalidation Mechanism

There are two issues to address here. The first concerns the scale of the invalidation, whether it is cache-level or query level. The second concerns the participants that are involved, whether the invalidation is performed by the client only, by the server only, or by collaboration between the two.

When a client receives an invalidation report, it can invalidate its cache content in two ways. First, it can perform cache-level invalidation, i.e., cache validity is performed for all objects cached. This requires scanning a large portion of the invalidation report, if not the entirety of the report. As a result, it is not particularly suited for selective tuning. In this approach, the cache content is associated with one timestamp - only the timestamp of the most recent invalidation report. On the other hand, the client can perform query-level invalidation, where validation is performed only on the objects queried. This reduces the number of objects to be invalidated and, hence, the report can be organized for selective tuning. However, each cached object has to be associated with a timestamp as compared to a single timestamp for all cached object in cache-level invalidation. The timestamp of an object represents the timestamp at which the object is last known to be valid. This is usually the timestamp of the invalidation report that was last used to validate the object. Thus, different cached objects will have different timestamps. So, each queried object may use a different list of objects for invalidation. When a query is issued, the query objects’ timestamps are checked against that of the invalidation report received. For each object queried, the appropriate list of objects is used to (in)validate it.

Invalidating the cache content can be performed by the client alone. This requires that the client based its invalidation purely on the invalidation reports. Thus, the effectiveness of such approaches is dependent on the content of the report. On the other extreme, we can

allow the server to perform the invalidation alone. This, however, will require the client to inform the server about its cache content which can be costly since transmitting this information consumes energy and bandwidth. Finally, the client and server can collaborate to identify the cache content that should be invalidated. The client uses the invalidation report to invalidate its cache content; for those that remain uncertain, the client submits their information to the server for invalidation.

Another important issue in the design of a cache invalidation scheme concerns the information (update logs) maintained at the server to reflect the updates on the database. The update logs may contain update information of each individual object or its identifier. For the former, the log record is of the form ⟨object, TS⟩ to reflect that object has been updated at timestamp TS. For the latter, the server only needs to maintain ⟨id, TS⟩ pairs, each of which indicates that the object with identifier id is updated at TS. Alternatively, each log record may reflect updates on a collection of objects. In this group-based approach, objects are organized into groups and the log record reflects the latest update to the group, i.e., each log record is of the form ⟨group-id, TS⟩ where TS is the most recent timestamp that an object in group identified by group-id has been updated. The second issue concerns the size and log history which is the duration that the update logs should be maintained which, like the content, are interrelated. The size can be fixed by restricting updates to be maintained for a fixed number of objects. In this case, the log history changes depending on the updates. On the other hand, variable sized logs can be maintained by fixing the log history to a fixed interval.

6.3.4 Cache Invalidation Scheme

Based on the above discussion, [Tan 2001] analyzed several cache invalidation schemes. The cache invalidation scheme reported in this thesis is a combination of Bit-Sequence Scheme and Bit-Sequences with Bit Count Scheme. Here we propose a Dynamic Cache Invalidation scheme which can dynamically adopt Bit-Sequence Scheme or Bit-Sequences with Bit Count Scheme.

The Bit-Sequence Scheme (BS)

The Bit-Sequence algorithm uses the following framework:

- Content: The report consists of a list of ⟨list of ids, TS⟩ pairs in a compact form. This allows the report size to be fixed, though the update history varies. There is no organization to support selective tuning.
- Invalidation Mechanism: The invalidation is performed by the client at cache-level.
- Log: The logs maintained at the server keeps track of individual object update information using ⟨id, TS⟩ pairs for up to half the database size, i.e., the size of the log is fixed but the update history is variable.

Let the number of database objects be $N=2^n$. In the BS algorithm, the invalidation report reflects updates for n different times T_n, T_{n-1}, . . . , T_1, where $T_i \langle T_{i-1}$ for $1 \langle i \leq n$. The report comprises n binary bit-sequences, each of which is associated with a timestamp. Each bit represents a data object in the database. A '**1**' bit in a sequence means that the data object represented by the bit has been updated since the time specified by the timestamp of the sequence. A '**0**' bit means that the object has not been updated since that time. The n bit-sequences are organized as a hierarchical structure with the highest level (i.e., bit sequence B_n) having as many bits as the number of objects in the database and the lowest level (i.e., bit-sequence B_1) having only two bits. For the sequence B_n, as many as half of the N bits (i.e., $\frac{N}{2}$) can be set to "1" to indicate the $\frac{N}{2}$ objects that have been updated. The timestamp of the sequence B_n is T_n. The next sequence in the structure, B_{n+1}, has $\frac{N}{2}$ bits. The kth bit in B_{n-1} corresponds to the kth '**1**' bit in B_n, and $\frac{N}{2^2}$ bits can be set to '**1**' to indicate that $\frac{N}{2^2}$ objects have been updated since T_{n+1}. In general, for sequence B_{n-i}, $0 \leq i \leq n - 1$, there are $\frac{N}{2^i}$ bits and the sequence will reflect that

$\frac{N}{2^{i+1}}$ objects have been updated after the timestamp T_{n-i}. The kth bit in sequence B_{n-i} corresponds to the kth '1' bit in the preceding sequence (i.e., B_{n-i+1}).

An additional dummy sequence B_0, with timestamp T_0, is used to indicate that no object has been updated after T_0. In general, N does not need to be a power of two and the number of lists can also be any value other than n. Furthermore, the list associated with timestamp T_i does not need to reflect the updates for half the number of objects in the list associated with T_{i+1}, $1 \leq i \leq n-1$. The Bit-Sequences structure is broadcast to clients periodically. The protocol for invalidating the cache is shown in Fig.6.10

```
//T – timestamp of current report
// T^C – timestamp of last valid report received by mobile client
if T0 ≤ T^C
      all cached objects are valid
else {
       if T^C < Tn
          remove the entire cache content
       else {
              determine the bit sequence Bi such that Ti ≤ T^C < Ti-1, 1 ≤ i ≤ n
              invalidate all the objects marked "1" in Bi
             }
       }
for every object O ∈ Qi {
      if (O is in the cache)
         use the cache's content to answer the query
      else
         submit request for O
}
T^C ← T
```

Fig 6.10 The Bit-Sequence scheme Protocol

The Bit-Sequence sceheme can be explained with an example. Consider a Bit-Sequence structure for an invalidation report as shown in Fig 6.11. As shown, the first level (B_4) has 16 bits, eight of which have been set to "1". These eight objects are the most recently updated objects. The timestamp for B_4 is 18. Similarly, bit sequence B_1 has two bits, reflects the most recently updated object 8, and has a timestamp of 32.

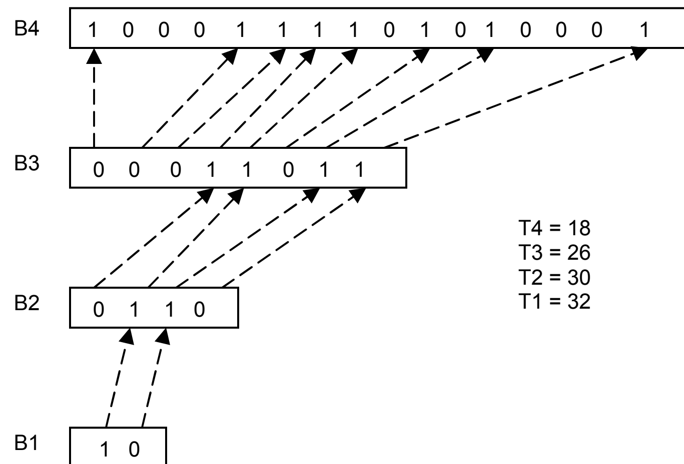


Fig 6.11 Bit Sequence Example

Assume that the client receives this invalidation report when it submits its query for objects 5 and 8. Suppose the last invalidation report received by the client before it disconnects is at time 31. Since the client's cache content is last valid at time 31, it should use the sequence B_2 to invalidate its cache. To locate those objects denoted by the two "1" bits, the client will check the sequences B_2-B_4. This is accomplished as follows: To locate the object corresponding to the second bit that is set to "1" in B_2, the client has to check the second "1" bit in B_3. Since the second "1" bit in B_3 is in the fifth position, the client will have to examine the 5th "1" bit in B_4. Because B_4 is the highest bit-sequence and the 5th "1" bit is in the eighth position, the client can conclude that object 8 has been updated since time 31. Similarly, the client can determine that the 12th object has also been updated since that time. Therefore, both objects will be invalidated by the client. Since the client requests for objects 5 and 8, object 5 remains valid and can be used to answer the query while the request for the invalid object 8 has to be submitted to the server.

Bit-Sequences with Bit Count (BB)

The other scheme, is the Bit-Sequences with Bit Count (BB) scheme. Like the Bit-Sequence approach, it comprises of a set of bit sequences organized in a hierarchical manner. However, only the relevant bits need to be examined. This is achieved by associating each bit sequence with a bit count array.

Let N be the number of objects in the database. Furthermore, let b_t denote the size of a timestamp. We also assume that a query Q returns the set of objects $\{O_1, O_2, \ldots, O_q\}$ as answers. Furthermore, we assume that the objects are already ordered in the same manner as the information reflected in the invalidation report, i.e., information for O_1 will be received before information for O_2 and so on. If the objects are not ordered accordingly, then they can be sorted. We also denote the corresponding timestamps when these objects are last valid in the client cache as $t_1, t_2, \ldots t_q$, respectively.

As in the BS scheme, the BB structure comprises a set of n bit sequences: Sequence B_n has a timestamp T_n indicates that updates after T_n are reflected and comprises N bits, half of which are set to '1'; sequence B_{n-1} has timestamp T_{n-1} and N/2 bits, of which $N/2^2$ bits are set to '1' and so on. In fact, the content of the bit sequences are exactly the same as those of the BS scheme. Like the BS scheme, if the bit sequence B_{n-i} is to be used to invalidate the cache, then the sequences $B_{n-i}, B_{n-i+1}, \ldots, B_n$ may have to be examined. However, the proposed BB strategy adopts a top-down examination of the sequences, i.e., from B_n to B_{n-i}, rather than the bottom-up approach (i.e., B_{n-i} to B_n) of BS scheme. Moreover, for some valid objects, it may not be necessary to examine all the sequences from B_n to B_{n-i} as it may be possible to determine their validity and terminate the search before sequence B_{n-i}. Furthermore, the proposed scheme only examines the relevant bits in each sequence. As the kth "1" bit in B_{n-i} corresponds to the kth bit in B_{n-i-1}, we need a mechanism that can count the number of "1" bits in a sequence, say B_{n-i}, without examining the entire sequence. To illustrate how selective tuning can be facilitated with such a mechanism, let us consider a query to validate an object O. The client first identifies the bit sequence that should be used. This is accomplished by examining the set of timestamps. Suppose the sequence is B_{n-i}. This means that we need to examine sequence B_n, followed by B_{n-1}, and so on until B_{n-i}. From object O, the client can selectively tune to the corresponding bit in B_n without scanning the entire B_n. If the bit is set to '0', then the object is valid, since object O will not be found in any subsequent sequences $B_{n-1}, B_{n-2}, \ldots$; otherwise, the client determines the number of '1' bits from the beginning of B_n to the bit corresponding to O. From this number, it can again selectively

tune to B_{n-1} and examine the corresponding bit of O in B_{n-1}. Again, if the bit is '0', then the object O is valid and the search terminates; otherwise, its position in the B_{n-2} is determined and this process is repeated until sequence B_{n-i}. We can terminate when we encounter '0' bit at any of the sequences from B_n to B_{n-i}. If the relevant bit at B_{n-i} is '1', then the object is invalid; otherwise, it is valid.

Now, the mechanism to facilitate selective tuning is simple. We associate with each bit sequence a bit count array, all of which have entries that are j bits. For bit sequence B_{n-i}, $0 \leq i \leq n - 1$, the sequence is partitioned into packets of 2^j bits.

In other words, there are $\left\lceil \frac{N/2^i}{2^j} \right\rceil$ packets. In general, for sequence B_{n-i}, the number of array entries is $\left\lceil \frac{N/2^i}{2^j} \right\rceil$. Essentially, the kth entry in the bit count array of sequence B_{n-i} represents the number of '1' bits that have been set for the kth packet in the sequence. Selective tuning is achieved as follows: Let packet i contain the bit which is the selected for the query sent. From the bit array count, we can determine the number of '1' bits that have been set for packets 1 to i - 1. The client can then tune into the ith packet and scan the ith packet until the relevant bit. In this way, we will be able to compute the number of "1" bits.

To check the validity of the answer objects to query Q, the client employs the protocol shown in Fig.6.12. The invalidation report is organized as follows: The counter is broadcast first, the timestamps are broadcast next, followed by the bit count arrays for sequences B_n, B_{n-1}, . . ., and, finally, the bit sequences B_n, B_{n-1}, . . ., B_1.

```
//Answer Set = {O1, O2, …, Oq}  --- objects for query Q
//tid - last known valid time of object with id Oid
download the counter, the timestamps and the bit count arrays
for each object Oi ∈ Answer set
        if TO ≤ ti     // object Oi is valid
           AnswerSet = AnswerSet – {Oi}
       else if ti < Tn    // object oi is invalid
           AnswerSet = AnswerSet – {Oi}
// AnswerSet contains the remaining objects whose validity is still uncertain
for each object Oi ∈ AnswerSet
         determine the bit sequence to be used to validate Oi
k = n
repeat {
         for each object Oi ∈ AnswerSet, examine bit sequence Bk {
           tune to packet p containing information on Oi
           examine the bits in packet p until position of Oi
           let the number of "1" bits set (in p inclusive of Oi ) be b
           if Bk is the bit sequence to be used to validate Oi {
                  if the bit corresponding to Oi is set to "1"
                     Oi is invalid
                  else
                     Oi is valid
                     AnswerSet = AnswerSet – {Oi}
           } else {
                  if the bit corresponding to Oi is set to "0" {
                     Oi is valid
                     AnswerSet = AnswerSet – {Oi}
              } else {
                     // we need to examine the next sequence Bk-1
                     from the bit count array of Bk
                          determine the number of "1" bit from packet 1 to p-1 of Bk
                     let this value be c
                     the position of Oi in bit sequence Bk-1 is (c+b)
              }
           }
         }
         k = k – 1
} until AnswerSet = ∅
```

Fig 6.12 The Bit-Sequence with Bit count scheme Protocol

To illustrate Bit Sequence with Bit count scheme consider the previous example discussed for Bit-Sequence scheme. The BB structure is as shown in Fig 6.13.

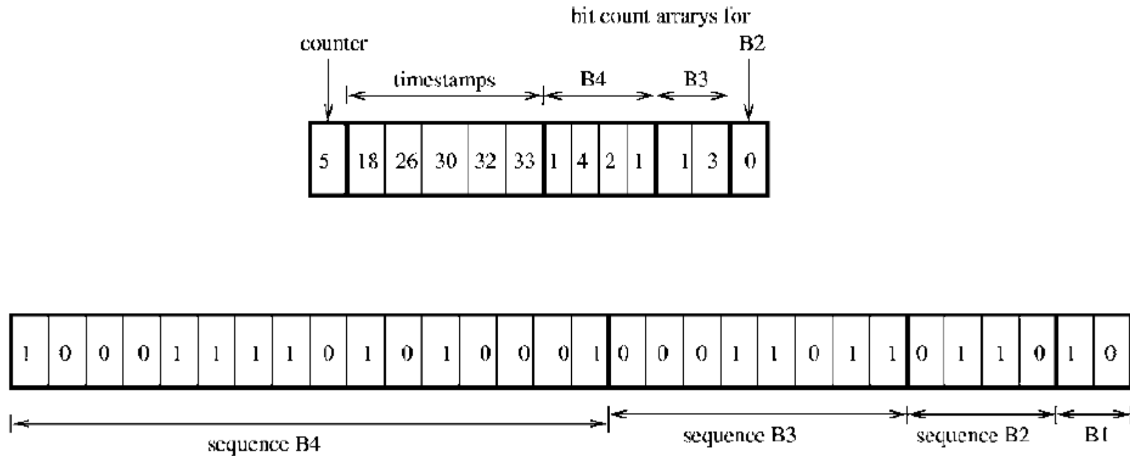


Fig 6.13 Bit Sequence with Bit Count (BB) Example

Each bit count array entry keeps track of the number of "1" bits set for four objects. Since there are 16 objects in the database, there are four entries in the bit count array corresponding to B_4, two entries in B_3's bit count array, and one entry in B_2's bit count array. Note that B_1 is not associated with a bit count array. Assume that a query requests for objects 5 and 8 whose cached timestamps are, respectively, 31 and 27. From the timestamps in the invalidation report, the client knows that it needs to check B_2 for the validity of object 5, and B_3 for the validity of object 8. The client first determines which two bits in B_3 correspond to the two queried objects. This is done as follows: As both objects 5 and 8 are in the same packet, from the first bit count array entry of B_4, the client knows that there is only one "1" bit among the first four objects in the bit-sequence B_4. Thus, it will tune to the beginning of the second packet of B_4 and examine the first bit in the second packet till the fourth bit. Since the first bit corresponds to object 5 and it is set to "1", the client knows that object 5 is the second bit in B_3. Similarly, the client can determine that object 8 is the fifth bit in B_3. For object 5, the client examines the corresponding bit in B_3 which has been set to "0" indicating that the object is valid. For object 8, the client first examines the bit count array for B_3 and knows that the first entry contains a value of 1. By examining the first bit of the second packet of B_3, it determines that the bit corresponding to object 8 is set to '1'. This also means that object 8 can be found in the second bit of B_2. It then examines the second bit of B_2 and finds that object 8 is invalid. The protocol is as shown in Fig. 6.13.

6.3.5 Complexity of BS & BB Schemes

Time Complexity of Bit-sequence algorithm:

In order to calculate the time complexity of bit-sequence algorithm, let us consider the following example. The structure of the invalidation report along with the bit sequences and time stamps are as given in Fig 6.14

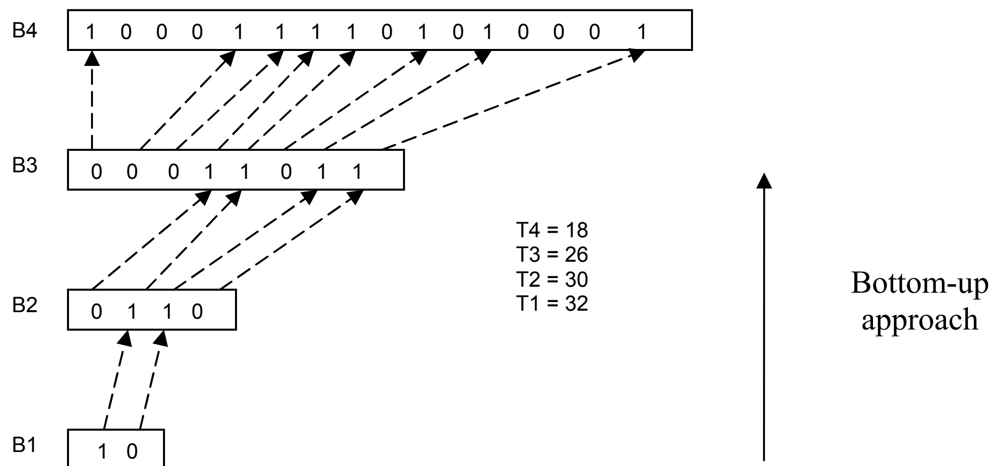


Fig 6.14 Structure of IR with bit sequence time stamps

Assume that the client receives this invalidation report when it submits its query and let the last invalidation report the client received be at time 31. So for answering the present query we start from the bit-sequence B_2 as the last valid time-stamp 31 is between time-stamp T_2 and T_1. The maximum number of updates possible at the bit-sequence B_2 is 2 which is equal to the number of bits in the next bit-sequence i.e. B_1. So the total number of iterations to be run for answering the validity of the present cache's objects is equal to the sum of the number of bits in bit-sequences B_2, B_3 and B_4.

Mathematically,

The total number of iterations = $s[B_2] + s[B_3] + s[B_4]$

where $s[B_i]$ denotes number of bits in bit-sequence B_i.

The above example shows us that knowing the last valid time-stamp is sufficient to get the total number of iterations. Generalizing the above example we calculate the time complexity of bit-sequence algorithm.

Let N be the total number of objects in the cache and k be the total number of updates. Let T_i be the last valid time-stamp. So, the corresponding last valid bit-sequence is B_i. The relation between k and i is given below:

$s(B_i) = 2^i = 2^k$

Using the above statements we get the following:

Time Complexity of bit-sequence algorithm, T (BS) = $(2k + 2^2k + \ldots + N)$

But , N = $\left(\frac{N}{k}\right) * k = k * \left(2^{\log_2\left(\frac{N}{k}\right)}\right)$

So,

$$T(BS) = \left[2k + 2^2k + \ldots + k * \left(2^{\log_2\left(\frac{N}{k}\right)}\right)\right]$$

$$= k\left[2 * \left(2^{\log_2\left(\frac{N}{k}\right)} - 1\right)\right]$$

$$= k\left[2\left(\frac{N}{k} - 1\right)\right]$$

$$\boxed{T(BS) = 2(N-k)}$$

Thus, the time complexity of the bit-sequence is 2(N-k).

Time Complexity of Bit Sequence with Bit Count (BB) algorithm:

Unlike the above algorithm this algorithm works on selective tuning top-down approach. The algorithm searches the validity of only one object unlike the case of bit-sequence which checks the validity of all objects at a time.

The time-complexity of the algorithm in finding the validity of one object:

Let, total number of objects in the server be N;

Let the Packet size be $\sqrt{N}$ and the number of updates be k;

In one bit-sequence,

Number of iterations = (Total no. of packets – 1) + (Packet size)

But, the number of bit-sequences to be checked for is equal to $\log_2\left(\frac{N}{k}\right)$

Therefore, for one object

$$T(BB) = [\ \sqrt{N} + \left(\frac{\sqrt{N}}{2}\right) + \left(\frac{\sqrt{N}}{4}\right) + \ldots \log_2\left(\frac{N}{k}\right) \text{times}] + \left(\sqrt{N}-1\right)\log_2\left(\frac{N}{k}\right)$$

$$= 2\sqrt{N}\left[1-\left(\frac{1}{2^{\log_2 \frac{N}{k}}}\right)\right]+\left(\sqrt{N}-1\right)\log_2\left(\frac{N}{k}\right)$$

Simplifying this we get,

$$\text{T (BB for one object)} = 2*\frac{(N-k)}{\sqrt{N}}+\left(\sqrt{N}-1\right)\log_2\frac{N}{k}$$

Let the number of objects in the cache be A,

$$\text{T (BB)} = A*\left(2*\frac{(N-k)}{\sqrt{N}}+\left(\sqrt{N}-1\right)\log_2\frac{N}{k}\right)$$

Optimal packet size for BB:

In the previous section, while deriving time complexity it was assumed that the packet size is √N.

Let us assume that the packet size in the BB algorithm be p.

Therefore,

$$\text{T (BB)} = 2*\frac{(N-k)}{p}+\left(p-1\right)\log_2\frac{N}{k}$$

For the optimal packet size $\frac{dT}{dp}=0$ and $\frac{d^2T}{dp^2}>0$,

$$\frac{dT}{dp}=2(N-k)(-p^{-2})+\log_2\left(\frac{N}{k}\right)=0$$

$$\frac{d^2T}{dp^2}=\frac{4(N-k)}{p}>0$$

minimum value of T at p,

$$p = \sqrt{\frac{2(N-k)}{\log_2\left(\frac{N}{k}\right)}}$$

But, maximum number of updates possible (worst case), k = N/2.

Therefore, $p = \sqrt{\frac{2(N-(N/2))}{\log_2 2}} = \sqrt{N}$

Thus for minimum time complexity, $p = \sqrt{N}$

6.3.6 Dynamic Cache Invalidation Scheme

The algorithm which we propose is a combination of both the algorithms BS and BB. Bit sequence algorithm works well when the number of objects in the cache is huge and the probability of the number of updates at the server is less. If the number of objects in the cache is huge then BB is less efficient because it has to traverse down the invalidation report for each object to find its validity. So, combination of both these strategies along with a proper switching condition which synergizes the advantages of both the algorithms will be much efficient.

The switch condition between BS and BB is explained below:

The main deciding factor for the efficiency of both the algorithms is time complexity. So for using BB algorithm the time complexity for invalidating the entire cache should be less than the time complexity of the BS algorithm.

i.e.,

T (BB) ≤ T (BS)

So, $A * \left(2 * \frac{(N-k)}{\sqrt{N}} + \left(\sqrt{N}-1\right)\log_2\frac{N}{k}\right) \leq 2(N\text{-}k)$

If the above condition is true then we use BB else we use BS.

6.3.7 Discussion of Results

It is found that the algorithm BS-BB works better for smaller cache sizes. The range for which it is valid is given by following proof:

For using the dynamic algorithm proposed here, the following condition should be satisfied..

$$A*\left(2*\frac{(N-k)}{\sqrt{N}}+\left(\sqrt{N}-1\right)\log_2\frac{N}{k}\right)\leq 2(\text{N-k})$$

Rearranging the above statement and substituting A = pN we get,

$$2N\left(p\sqrt{N}-1\right)+pN\left(\sqrt{N}-1\right)\log_2 N\leq 2k\left(p\sqrt{N}-1\right)+pN\left(\sqrt{N}-1\right)\log_2 k$$

We divide both sides of the inequation with $2pN\left(\sqrt{N}-1\right)\left(p\sqrt{N}-1\right)$ assuming it to be positive.

Thus we get,

$$\frac{N}{pN\left(\sqrt{N}-1\right)}+\frac{\log_2\sqrt{N}}{p\left(\sqrt{N}-1\right)}\leq\frac{k}{pN\left(\sqrt{N}-1\right)}+\frac{\log_2\sqrt{k}}{p\left(\sqrt{N}-1\right)}$$

This is not possible as k is always less than N. So,

$$2pN\left(\sqrt{N}-1\right)\left(p\sqrt{N}-1\right)<0$$

$(\sqrt{N}-1)(p\sqrt{N}-1) < 0$

Thus, $1 < \sqrt{N} < \left(\frac{1}{p}\right)$

But the k value is considered only half the N value at maximum. So the above equation becomes,

$$1 < \sqrt{N} < \left(\frac{1}{2p}\right)$$

The above condition gives the range of N. If the above condition is not satisfied then Bit Sequence algorithm is applied.

The time vs. updates for different N values are as shown in Fig 6.15 to Fig 6.19.

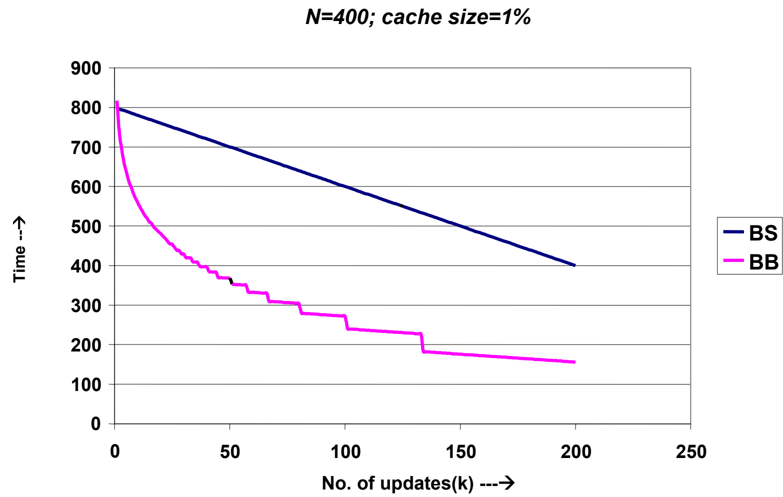


Fig 6.15 No. of updates vs time for N=400 and cache size = 1% of the total objects

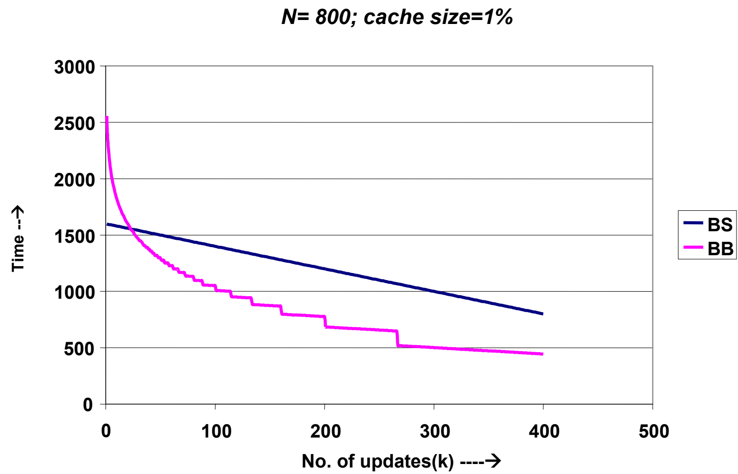


Fig 6.16 No. of updates vs time for N=800 and cache size = 1% of the total objects

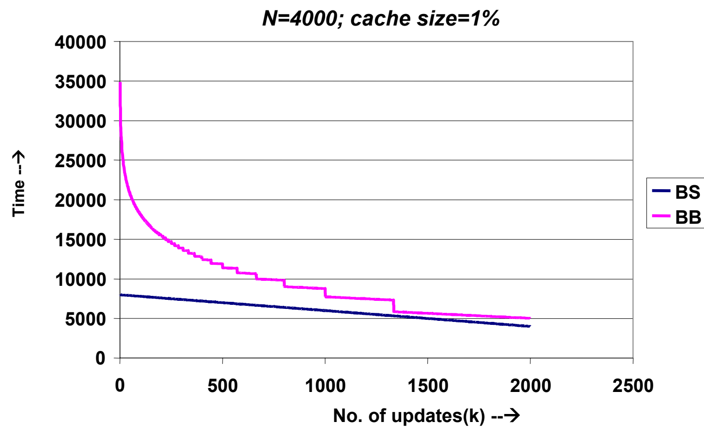


Fig 6.17 No. of updates vs time for N=4000 and cache size = 1% of the total objects

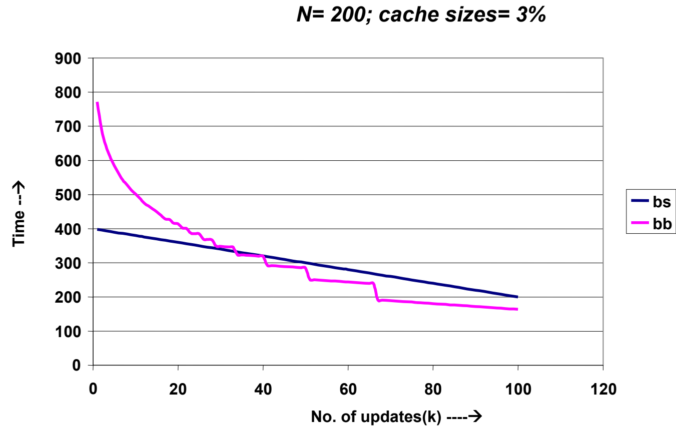


Fig 6.18 No. of updates vs time for N=200 and cache size = 3% of the total objects

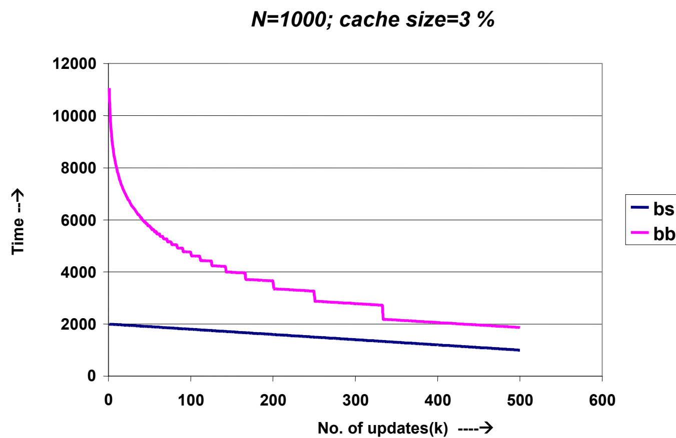


Fig 6.19 No. of updates vs time for N=1000 and cache size = 3% of the total objects

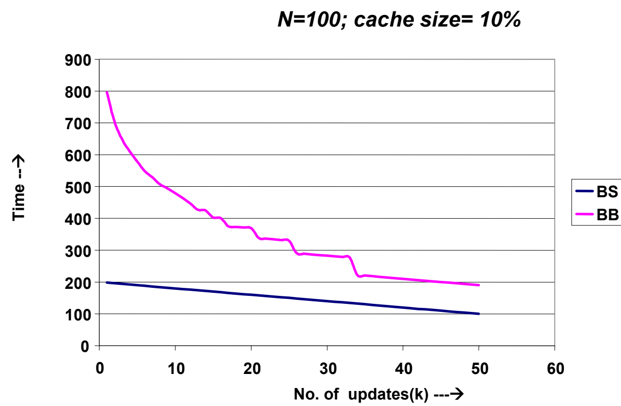


Fig 6.20 No. of updates vs time for N=100 and cache size = 10% of the total objects

6.4 Summary

In this chapter, we present a QoS strategy for Wireless Internet Access based on IPS-CoD protocol. The objects requested by clients are prefetched into the cache and made available beforehand, so that the latency for clients on the move can be significantly reduced. This strategy can be combined with hand-off management to give better results. We also showed that this strategy does not have significant impact on the cache hit for other clients who are not using CoD service. Also, later we discussed the cache invalidation strategy in a mobile environment and a dynamic cache invalidation strategy based on Bit Sequence and Bit Sequence with bit count invalidation schemes. We derived a condition for adopting one of these strategies depending on object invalidation parameter.

CHAPTER 7
CONCLUSIONS

The thesis deals with different caching techniques for enhancing the user experience in fixed and wireless web access. In this chapter, we summarize the main contribution made and point out some of the possible extension of the work.

The thesis, focused on the design and implementation of dual-stage victim cache policy (Chapter 3). Based on the results obtained, we can conclude that Dual-Stage with Victim Cache policy is a practical and viable caching algorithm. It has a good hit ratio performance and is also robust w.r.t varying workload characteristics. The study of how different caching algorithms would perform with smaller size caches which can be used for schemes like Cache-on-Demand proxies was carried out. Further a History-based randomized cache replacement policy has been analyzed. It is observed that randomized replacement policy with LRU or SLRU performs better than only LRU or SLRU. Size based replacement is known to be more efficient than a Least Recently Used policy. This is confirmed by our results, which shows that RSLRU has a higher hit ratio than RLRU. On using random replacement with history based policy we observe that HRLRU performs better than RLRU, but this does not hold for the size based replacement. In the case of HRSLRU and HRLRU it is observed that for smaller cache sizes the former performs better. HRSLRU is quite stable with increasing cache sizes. The Cache on Demand (CoD) protocol enhances traditional web caches with the capability of reserving resources to store external content for a specified period of time. The major benefit of this feature is that a third party, such as a content provider or a business partner can have guaranteed content presence in the network, and also strong control on the content delivered to web users. Furthermore, a third party can enforce strong content consistency since it can keep track of distributed content at different CoD cache locations.

The cache replacement in streaming multimedia is a prominent research problem and Chapter 4 of the thesis deals with this. The results obtained for OC algorithm and CC algorithm were analyzed. It was observed that in case of the total cache size requirement,

for all the bandwidths used, OC algorithm occupies less cache space than CC algorithm. In case of the total bandwidth utilized OC algorithm outperforms the CC algorithm. It utilizes the external bandwidth more efficiently than CC algorithm for all the cases of bandwidth. By these results it is evident that though OC algorithm is slightly difficult to implement, it definitely is a better performer. A replacement policy, based on frequency-index was proposed for replacing videos that are cached using the above algorithms. A popularity table is maintained in the proxy, which has the popularity index for all the video files stored in the proxy. The videos are replaced based on their popularity; the least popular video is removed from the proxy first. It has been observed that the hit-ratio increases with increase in bandwidth. This is due to the reason that, as the available bandwidth increases, the cut-off size increases and hence the number of bytes cached per video decreases. Consequentially, more number of videos can be cached thereby improving the hit-ratio. It was also observed that the OC algorithm gives a better hit ratio compared to the CC algorithm. As the bandwidth increases the hit ratios of both the algorithms are nearly the same, since both the algorithms behave the same way at high bandwidths. Further, the hit-ratio increases with increase in cache size. Also for a constant bandwidth, more the available cache size, more the number of videos that can be cached and hence higher the hit-ratio. In this case as well, the OC algorithm gives a better performance compared to the CC algorithm. FIR algorithm yields better hit-ratio as compared to traditional algorithms like LRU and LFU. This is primarily due to the fact that replacement here happens frame-by-frame as opposed to complete Boolean replacement in the other two algorithms.

In Chapter 5 we proposed soft computing techniques for cache replacement strategies both for static web objects and streaming multimedia objects. Size based policies normally have a higher hit rate eg.. SLRU outperforms even FUZZY algorithm, but suffer from a lower byte hit rate. The Fuzzy Algorithm has a higher hit rate for lesser cache sizes when compared to LRU and LFU. Considering Byte Hit Rate, for smaller cache sizes the traditional algorithms are better than Fuzzy approach proposed whereas for large cache sizes Fuzzy approach performed better than LRU and LFU but not better than SLRU. This made us to consider the other soft computing method, the Genetic

Algorithm. On analyzing the Genetic Algorithm based Cache Replacement Policy (GAR), it was observed that GAR has a higher hit-ratio as compared to traditional algorithms. SLRU has a superior hit-ratio than GAR for smaller cache sizes. As cache-size was increased, GAR scores over SLRU. The hit-ratio of *GAR* has been found to be between 0.6 and 0.8.

Genetic algorithms was then extended to cache streaming multimedia objects in place of frequency indexed replacement policy. We adopted Genetic Algorithm by assigning a fitness value to every frame and the replacement was done of the less fit frames and not whole videos. In the comparison, Genetic algorithm based replacement algorithm yielded better hit-ratios than LFU and LRU under similar memory availability, for a synthetic workload. This is primarily due to the fact that replacement here happen frame-by-frame as opposed to complete Boolean replacement in the other two algorithms. The number of replacements as compared to LRU and LFU has been minimal which shows that the network bandwidth is saved due to minimal replacements. It can be observed that the algorithm is suited especially for high-end servers, which deal with a large amount of memory for caching multimedia objects.

In Chapter 6, the issues of caching in a mobile environment were addressed. We proposed integrating intelligent proxy servers with cache-on-demand protocol which helps the cache to be reserved for the mobile client being registered in a new base station and prefetching the web objects requested by the client. So, the frequently accessed web content is prefetched based on the user's profile and put into the proxy cache. This was done to show the basic advantage of integrating the mobile network with the Cache-on-Demand protocol. Analyzing the results obtained, we can see that the CoD protocol will help the user with a better Quality of Service and can be offered as a service by the Content Delivery vendors, without affecting the clients who are not using the prefetching service. This gives a higher cache hit ratio, reduces the bandwidth usage and latency in servicing client requests. In addition, it also minimizes the congestion in the mobile network and the load placed on origin servers.

The thesis also discussed the cache invalidation schemes, which is to keep cache coherency in a wireless environment. The scheme 'bit-sequence' and 'bit-sequences with bit count' were discussed. After deriving the time complexity of these two algorithms, we proposed a dynamic scheme where the derived condition can be used to chose between the bit-sequence and bit-sequence with bit-count scheme. It is shown that this scheme works well with various workloads.

In Appendix A, implementation of replacement schemes for uniformed objects is discussed. LRU is the most popular and efficient scheme used for replacement scheme for uniformed objects, but it has been the most difficult scheme to be implemented especially when the associativity of the cache is higher. Higher associativity with the LRU replacement policy is a good configuration for reducing miss rate in the cache design and enriching the performance in many applications, high-end servers, workstation and modern processors. Implementing LRU policy in hardware for high associativity is difficult. Implementation objectives are identified and various implementations namely Square Matrix, Skewed Matrix, Counter, Phase, Link List and Systolic Array are discussed. The results of the different implementations for increasing associativity are analyzed. It is inferred that for higher associativity, conservation of space to store data of the schemes is important but the associated logic cannot be totally neglected. At higher associativity, Linked List, Systolic Array and Skewed Matrix are the designs most suitable for implementations. Delay also follows the same characteristics and with increase in associativity, the Link List, Systolic and Skewed Matrix would involve less delay. Although the implementation size for one set grows rapidly with increase in associativity, the growth is much less when considered for the entire cache. The results also show that the LRU implementations, which involve smaller storage space with little increase in component size or number of components, show better behavior with increasing associativity. Finally of all the implementations, Systolic and Link List show better results while Skewed Matrix with less information also exhibit similar performance.

Further Fuzzy Logic and Neural Network approach for caching & replacements of multimedia objects can be carried out as future work which will lead to comparative

evaluation of these soft-computing techniques. Efficient cooperative caching for streaming multimedia objects among peer-proxies needs to be investigated. Also the other area which has to be addressed is cost-optimization methods for mobile users where the contents cached can dynamically move with a mobile client.

APPENDIX

LRU IMPLEMENTATIONS FOR UNIFORMED OBJECTS

A.1 INTRODUCTION

Use of cache memories in computer architecture is well known and existed even before the Internet was envisaged. Proxy caching for enhancing web service performance has several similarities to caching in memory architectures to improve computer performance [Smith 1982]. Because central processing units operate at high speeds while memory systems operate at a slower rate, CPU designers provide one or more levels of cache – a small amount of memory that operates at or close to the speed of the CPU. When the CPU finds the information it needs in the cache, a hit, it doesn't have to slow down. When it fails to find the requested object in the cache, a miss, it must fetch the object directly and incur the associated performance cost.

Typically, when a cache miss occurs, the CPU places the fetched object in the cache, assuming temporal locality — that a recently requested object is more likely than others to be requested in the future. Memory systems also typically retrieve multiple consecutive memory addresses and place them in the cache in a single operation, assuming spatial locality — that nearby objects are more likely to be requested during a certain time span. At some point the cache will become full and the system will use a replacement algorithm to make room for new objects, for example, firstin/first-out (FIFO), least recently used (LRU), or least frequently used (LFU). The goal is to maximize the likelihood of a cache hit for typical memory architectures.

Modern processors, commercial systems, high performance servers and workstations have high-associative caches for performance improvement. The complexity of implementation of LRU policy for highly associative cache tends to increase as the associativity increases [Hennessy 2003, Patterson 2005, Hwang 1993, Deville 1992]. The increase in complexity additionally increases the delay incurred in detecting the line

for replacement degrading the cache performance. This work is an effort to implement and analyze efficient LRU implementations for high-associative caches. Various implementations of LRU were designed, simulated and synthesized for comparison. These designs are analyzed with respect to their implementation complexity.

A.2 HIGH-ASSOCIATIVITY CACHE WITH LRU POLICY

The classical approach to improve the cache behavior is reducing miss rate. Increasing associativity in the cache reduces conflict misses thereby reducing miss rates and improving performance. Studies have shown that conflict miss reduces from 28% to 4% when the associativity changes from 1-way to 8-way [Patterson 2005]. High-associative cache is more efficient when miss penalty is large and memory inter connect contention delay is significant and sensitive to the cache miss rate [ZhangC 1997]. Due to rapid changes in technology, the miss penalty is becoming smaller and thus, the replacement policies have to be faster. Better performance of high-associative cache depends on efficient replacement algorithm [Deville 1992]. The replacement algorithm LRU, that replaces the least used line in cache, has miss ratio and performance, comparable to optimal (OPT or MIN) algorithm.

LRU is currently the most common replacement strategy used in cache, which gives higher performance [Smith 1982]. Result from [Smith 1985] have shown that for many workloads FIFO and Random replacement policies yield similar performance but the miss ratio of LRU is 12% lower on the average thus yielding better performance than other policies. Studies [Sugumar 1993] have shown that in the case of larger associativity, LRU can be noticeably improved and made more optimal when compared to the off-line MIN [Belady 1966] or the equivalent OPT algorithms [Mattson 1996]. A high-associative cache with LRU is a better solution for reducing miss rate and improving performance. This combination has an added advantage of reducing thrashing provided that associativity value, N is greater than M, where M is the number of different blocks that map to the same set. Results from [Ailamaki 2000] reveal that cache design affects the behavior of database application and high-associativity gives better performance for database workload. Increasing associativity in Network processor cache

removes the problem of cache conflicts [Gopalan 2002] enhancing performance. High-associativity is a reasonable way to increase the physically addressed cache size for it does not increase the translation hardware. High-associativity also improves execution time of numerical intensive applications.

Commercial systems use processors with high-associative cache to obtain better performance. IBM POWER 3 architecture has 64KB data and 32KB instruction cache implemented as content addressable memories with each array having 128-way set associative and 8-way interleaved cache [Papermaster 1998]. Many Industrial Embedded processors like ARM3 has 64-way associativity, Strong ARM, Intel SA-110 and Intel X Scale has 32-way associativity [ZhangM 2000, SA110 2000]. Altera's Excalibur EPXA10 has a 200 MHz ARM 922T processor with 64-way set associative 8Kbyte instruction and 8KByte data caches [Excalibur 2002]. Thus, high-associativity has been employed commercially to enhance the performance of systems and applications.

A.3 IMPLEMENTATION COMPLEXITY

A 2-way set associative cache with LRU policy can be implemented with a one-bit counter called the access bit. When a line is accessed from the set, the access bit of the line is set representing most recently used line and the access bit of the other line is reset to zero representing least recently used line. If associativity is increased to four, LRU could be implemented as a counter where the number of access bits will be two. Beyond this implementing a LRU policy becomes difficult. The number of lines in a cache set increases, increasing the storage space to maintain the LRU history, thereby increasing cache size and cost. Complexity of the logic to implement the LRU also increases [Deville 1992]. Studies reveal that the performance impact of the LRU policy reduces as the associativity increases [Wong 2000]. [Mattson 1996] showed the LRU performs close to OPT replacement algorithm when associativity is less but has a large number of victims to choose from when the associativity is large decreasing performance. Although LRU is the best replacement policy, which can help to reduce miss ratio, it performs poorly due to inefficient implementation [Deville 1992, ZhangC 1997]. Efficient implementation of LRU in a high-associative cache will increase the performance of the

cache. It is shown in [Sugumar 1993, Deville 1992] that for FIFO and random replacement policies the complexity of implementation is relatively low whatever is the associativity. Sugumar and others gave a simpler implementation for FIFO and Random replacement policy. Eventhough LRU is the best policy, designers of embedded microprocessors for low power design chose other policies instead of LRU and made a compromise on performance in order to have simpler implementation [Clark 2001].

An efficient LRU implementation to improve the performance is necessary but implementation has many design constraints. LRU hardware should maintain a data structure where it logs every access to the cache. As the associativity increases the size of the data structure and associated logic also increases. But the storage size cannot be large due to space and time constraints. When the storage space is reduced, the complexity of the logic needed to log the access usually increases. Further the time taken to log every access and the time to find the line to replace when a miss occurs should be less in order to reduce the miss penalty. LRU hardware, with less storage space to log the access, with less complexity in circuit, less time to log the access, less time to detect the replacement line on miss is required for improved performance of cache.

A.4 PROPOSED DESIGN APPROACHES FOR LRU IMPLEMENTATION

The information of each access should be logged in a data structure that determines the performance of the LRU hardware. Each set in the associative cache has its own LRU hardware for implementing the LRU policy. On referencing this set the corresponding hardware is also invoked requiring no separate detection. The collection of this hardware for all the sets in the cache is the Global Set. And the hardware for the set, which is being referenced, is the Working Set. The cache line index in case of a hit is the index of the line whose tag matches with the tag bits of the referenced address and in case of a miss is the index of the line identified for replacement by LRU hardware. In this section, we compare six different implementations of LRU policy for attaining high performance for an N-way set associative cache with Square Matrix, Skewed Matrix, Counter, Link list, Phase and Systolic Array methods

A.4.1 SQUARE MATRIX IMPLEMENTATION

This scheme implements a simple data structure with a simple storage element, D flip-flop. The data structure used is a square matrix of this storage element and is of order N for an N-way set associative cache. The global set contains M replications of this data structure for a cache with M sets. Here each of the N rows of the data structure maps to one of the N cache lines of the set and logs the access information of that line. Initially all the bits in matrix are set to zero as shown in Fig A.1. The Square-Matrix implementation follows a simple logging scheme wherein, it sets the row of accessed line to one and after this sets the column of the accessed line to zero.

	1	2	3	4
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0

Fig A.1 4x4 matrix initialized to zero

	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0

Fig A.2 4x4 matrix with cache line 3 as the least recently used line.

The number of ones in each row is an indication of the order of the accessed cache lines for the set. A line with more number of 1's is more recently accessed than the one that has less number of 1's. The row in the matrix, which has the maximum number of 1's, is the line most recently used and the row, with all bits set to zero is the line least recently used as shown in Fig A.2. On a cache miss, LRU is detected by checking the row for which all the storage elements are zero. There will always be a line that has the entire row set to zero. The Matrix is made up of N x N storage elements. The hardware also has one n x N decoder, a 2 to 1 n-bit multiplexer and N x n priority encoder, where n is $\log_2 N$. The encoder gives priority to lines with lower index value. Fig A.3 shows the LRU implementation of N-way set-associative cache.

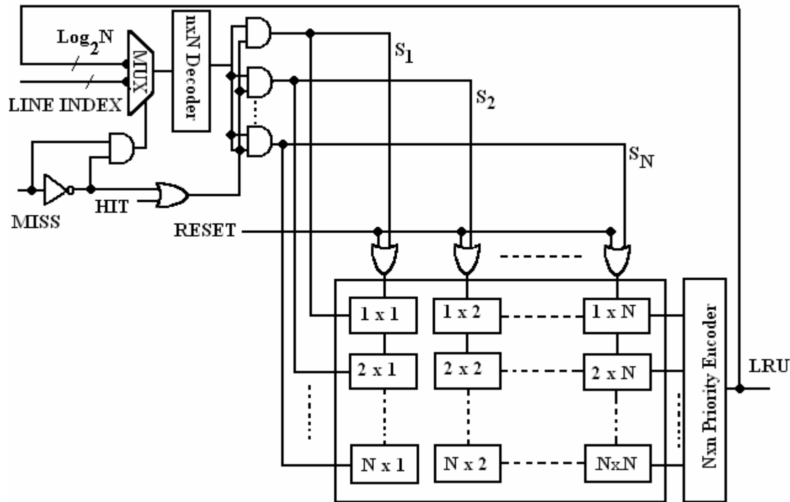


Fig A.3 Square Matrix Implementation

The cache line index is presented to the n x N decoder from the multiplexer, which is switched by the hit signal to accept it. The cache line index selects the correct corresponding row and column. The storage elements in the row are set and in the column are reset. The ANDed output of the values of the elements of each row is fed into a priority encoder, which detects the rows whose all elements are Zero and selects one amongst them as the LRU. In case of a miss this index is presented to the multiplexer, which is triggered by the miss signal to accept it and the corresponding row and the column are set and reset respectively. RESET pulse high initializes the matrix by setting all storage elements to zero. The hit or miss decides how the matrix information is to be altered. As very simple operations of set/reset are done on the basic storage elements, the delay involved and the time required to log the access is less when there is a hit. The replacement line is obtained from the priority encoder after the values in all the storage elements are ANDed causing considerable delay in the detection of the replacement. The data structure implemented is simple and a minimum of associated logic is required. But the design does not scale well because a large amount of space is required to hold the information that increases quadratically with N, the associativity.

A.4.2 SKEWED-MATRIX IMPLEMENTATION

Skewed-Matrix method is a variant of the previous implementation where a compromise is made in the amount of LRU history being stored. For large sets only a group of cache lines in the set may be active simultaneously. Not keeping the history of other lines would only affect performance slightly. The history is kept for a smaller number of lines B, where B is less than N and needs careful choosing with respect to N. If a line is not accessed in the last B accesses of the set it is considered to be the least recently used line. So, when B is less than N we have more than one line for replacement simultaneously. This differs from the previous implementation in choosing the column to set to zero and in the choice of the replacement row. Rows are set as in Square matrix but since the number of columns is less so more than one line maps to one column. N lines clear B columns and so after B accesses more than one row would have all zero values as N mod B lines would map to the same column. The Matrix itself has B columns and N rows as shown Fig A.4. A separate bxB decoder with lower order b lines from the Multiplexer as input, where b is $\log_2 B$, is used for the columns. The replacement mechanism chooses the row that is zero only if the one above it is not, so as to use all the rows, which would not be possible with the previous implementation.

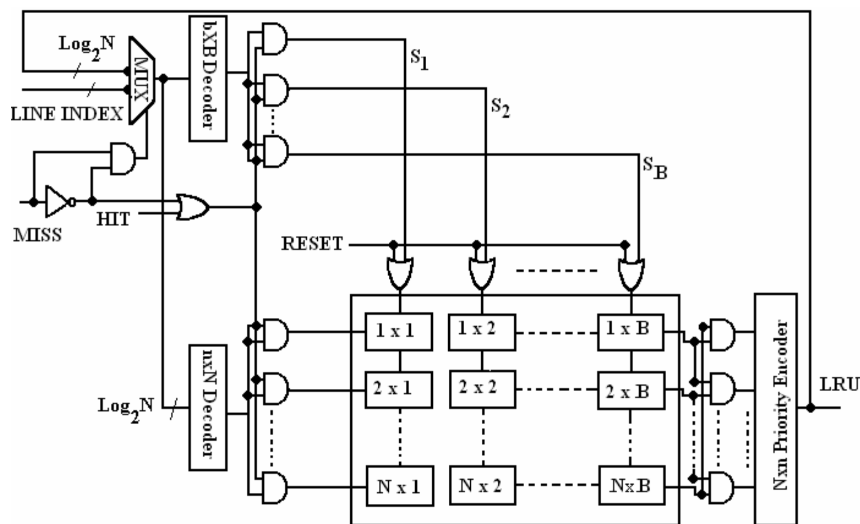


Fig A.4 Skewed Matrix Implementation

Skewed-Matrix needs less storage space than the Square Matrix though there is some increase in the complexity of the associated circuitry and the path to detect the replacement. It performs as well as the Square-Matrix implementation given the correct value of B but to predict the correct value of B is difficult.

A.4.3 COUNTER IMPLEMENTATION

Using a register for each row to maintain the LRU history can reduce the large space occupied by the Square and Skewed Matrix implementations. As the value of N becomes higher there is exponential drop in the storage space when compared with previous implementations. There is one to one mapping between the register, used to record LRU information and the cache line in a set. The values in the register indicate the order in which the cache lines within a set have been accessed. A register with a larger value means that corresponding cache line is more recently accessed than the line whose register has a lesser value. The smallest value, Zero in the register indicates the corresponding cache line is least recently accessed line and the highest value, N-1 indicates the corresponding cache line is most recently accessed line. Initially all the registers are set to zero. The value of the active register whose cache line is being accessed, is compared with the value of other registers .The registers whose value is greater than active register are decremented and the active register is set to the highest value N-1. The register can be reset to zero, decremented and loaded externally. Each cache line in every set is mapped to a register.

The hardware implementation for this data structure for set is as shown in Fig A.5 and needs one 1xN $\log_2 N$-bit demultiplexer, one 2x1 $\log_2 N$-bit multiplexer, one 1xN 1-bit demultiplexer, one Nx1 $\log_2 N$-bit multiplexers, N $\log_2 N$-bit comparators, N $\log_2 N$-bit registers and one Nx1 priority encoder. The comparator hardware determines the registers whose value is greater than the register of the indexed cache line and equal to it. A zero register means that the line can be replaced and there can be a number of registers that can be used as replacements, so a priority encoder decides which of the line can be replaced.

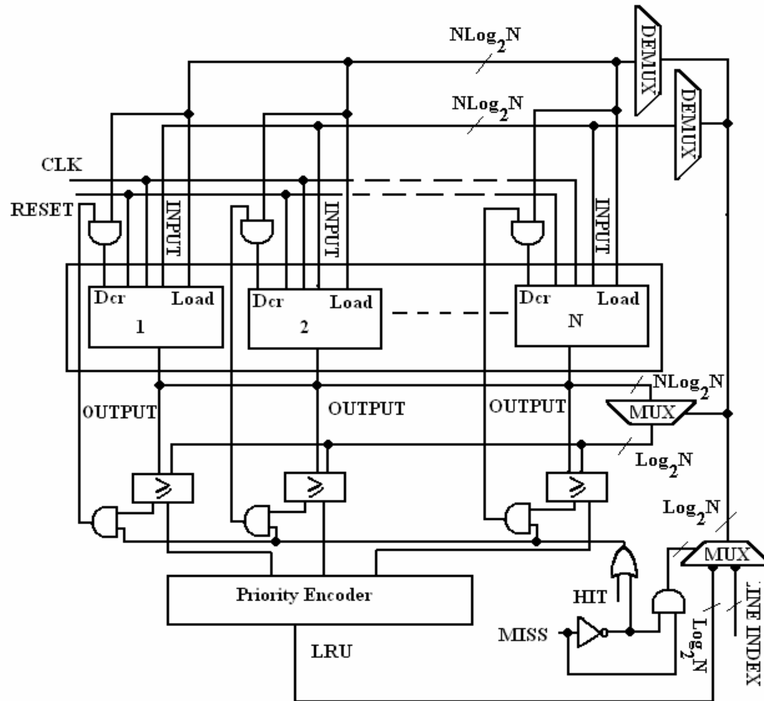


Fig A.5 Counter Implementation

The register is decremented if a comparator for that register signals that the register is greater than the indexed line provided the load signal for that register is not high. The indexed register is set to N-1 by using the input from the first demultiplexer and the register to be used for comparison is indicated by the second demultiplexer and is fed to all comparators using the multiplexer. The 2x1 multiplexer is used to select the indexed line, which is the replacement line in case of a miss or the accessed cache line index in case of a hit. This implementation uses the minimum, N number of Storage elements, among the various implementations but the associated logic to detect LRU and logging information is more as compared to other implementations. The implementation does not scale well as the complexity of the associated circuitry increases with N.

A.4.4 PHASE IMPLEMENTATION

Phase implementation uses the Matrix to implement the Phases concept. Phase is the period where a series of references is made in the set. This implementation is an adaptation from [Deville 1992] by Yannick Deville and Jean Gobert, which shows that

using phases improves the miss ratio. The rows are indexed from 0 to E-1 and the columns from 0 to B-1. E is associativity of the cache and B is a free parameter chosen depending on the design but it should be less than and multiple of E. The Matrix is set to zero initially. The column with the highest index is the active phase, so there is no need of the B-pointer, which is a pointer that points to the active phase to track of this phase. A counter, E-counter, is used to keep track of the number of lines that have entered the phase and when a maximum of E/B lines are in the phase, new phase starts. The change of phase is indicated by a shift in the Matrix. All the rows of the Matrix are shifted left by one element. At the start of the phase all the highest Index elements are set to zero. Every time a line is accessed its row is set to 1. E-counter is incremented only when a line, which is accessed, has a zero in the highest indexed column of the corresponding row, i.e. in the active phase. When the E-counter reaches E/B value, the phase ends. The LRU will be the row that has the least number of ones or the maximum number of zero's. There can be more than one such row.

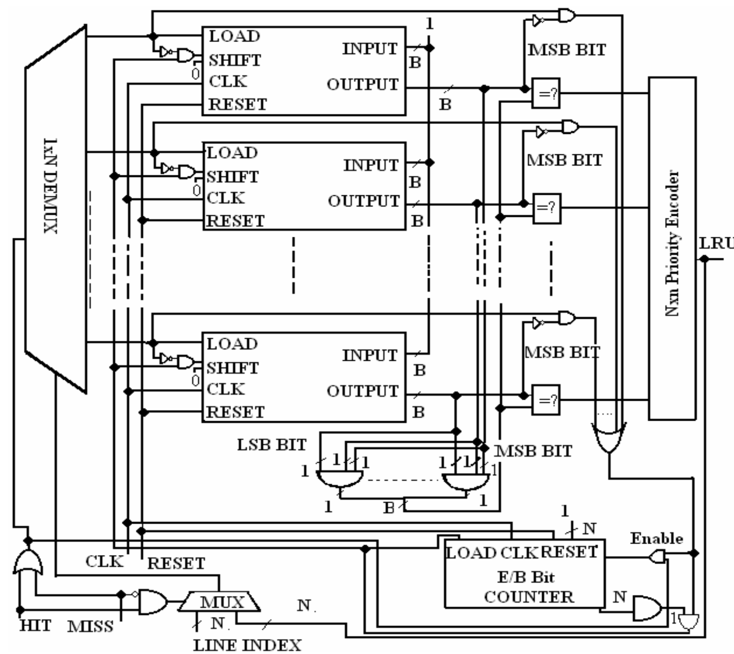


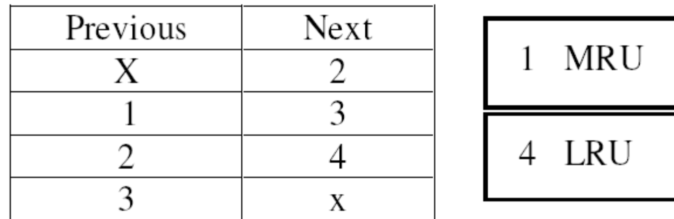
Fig A.6 Phase Implementation

Phase implementation uses a shift register as a basic storage element. A shift value of 1 shift the values in the registers to left, 2 shifts the values in the registers to right and 0 does not shift at all. When there is a left shift the MSB is set to zero. The register is cleared at Reset signal and stores the input value when load signal is high. Along with the N storage elements, the hardware consists, as shown in Fig A.6, of a 1 x N Multiplexer, 1 x N bit Demultiplexer, a priority encoder, N comparators and a E/B-bit counter, as the E-counter. Each storage element corresponds to the row of the Matrix. When there is an access and the value E-counter is not E/B-1 then the E-counter is incremented provided the MSB, active bit of the accessed row is not one. An MSB value, 1 of the row accessed indicates that it is previously accessed and the counter should not be incremented. If the MSB of the accessed row is 0 and the E-counter has reached its maximum value then the rows are left shifted and the counter is set to 1. On every access, all the bits of the corresponding register of the accessed line are set to 1. Multiplexer selects the accessed row, which is the index in case of a hit or the LRU row in case of a miss. A priority encoder indicates the LRU row. Phase implementation is similar to Skewed or Square Matrix implementations but requires more complex logic. Design parameter B has to be chosen carefully. A small value means less storage space but a low accuracy of prediction. A large value means it requires large storage space. Appropriate value of B will help reduce the complexity of the circuit. Accessing a line in Phase implementation takes less time.

A.4.5 LINK-LIST IMPLEMENTATION

In the Link-list implementation, by using a smaller additional space we design logic to determine the LRU line with minimum delay and at the same time update the data structure. The cache lines indexes are mapped to two lists: Previous and Next. The Next register of the cache line maintains the index of the line that was accessed after that cache line and the Previous register of the cache line maintains the index of the line that was accessed before that cache line. The most recently used cache lines are moved to the head of the list and the less recently used lines to the end of the list. LRU register keeps track of the index at the end of the list and the MRU, the index at the head of the list. An arbitrary ordered list can be chosen to initialize all the registers as shown in Fig A.7 and

the line at the end of the list becomes the LRU. When lines are accessed and as they are accessed their order of access is determined in the list. The algorithm used for updating the lists handles the three cases. If the accessed line is LRU then it is made to point to the next of LRU, and if it is the MRU then nothing needs to be done.



Previous	Next
X	2
1	3
2	4
3	x

Fig A.7 Entry in the Previous list and Next list for Link List implementation

But for any other line the previous node is made to point to the next node in the Next list and similarly the Previous list is updated. The accessed line is made the MRU. X is the hardware that determines the Next or the Previous index value of the line index to which this register in the list is mapped. The hardware for a set is as shown in Fig A.8. It has four 1xN $\log_2 N$-bit demultiplexers, four 1xN 1-bit demultiplexers, two Nx1 $\log_2 N$-bit multiplexers, one 2x1 $\log_2 N$-bit multiplexer, N storage elements for the Next list and Previous list each, and two storage elements for LRU and MRU. The demultiplexers select the storage element in the other list and also give the value to be stored in this list. The multiplexer selects the correct storage element to which the data must go and also selects the load of that element. Two pairs of multiplexers are used for updating the list with values from the LRU and MRU storage elements and also from the storage elements in the list itself, simultaneously. The three cases are handled by the load signal to the two lists from the comparators, which compare the LRU and the MRU with the accessed line index. The number of components in the associated logic for this implementation does not increase as the value of N increases, however the size of the components increases. But the delay in determining the LRU is not affected much by the increase in the value of N.

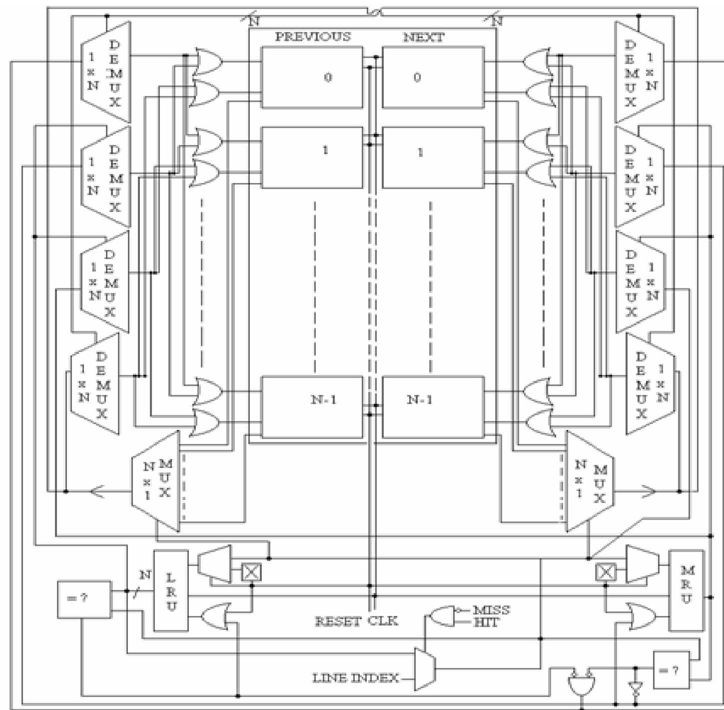


Fig A.8 Link List Implementation

A.3.6 SYSTOLIC ARRAY IMPLEMENTATION

The list for all previous implementations determines the true order of access of the respective cache lines immediately after access but the systolic array, which is an adaptation from [Grossman 2002] does not update the list immediately. This scheme has a Systolic node as shown in Fig A.9 which has both storage and processing capability.

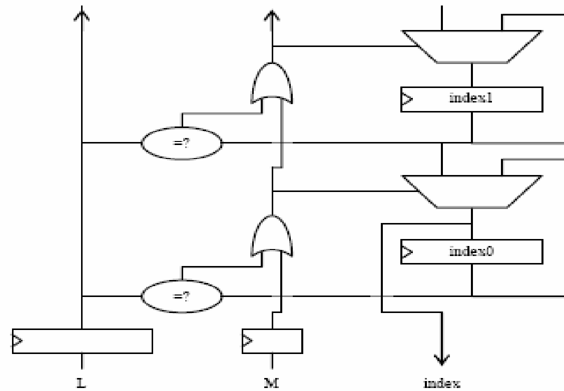


Fig A.9 Systolic Node

Systolic Array implementation for one set, shown in Fig A.10 uses N/2 such nodes to form the list with the last node having the input connected to the output. The LRU is always correct since it is the first node of the list and updated first. Systolic array which uses the concept of link list, sorts the line-index from LRU to MRU. Each node comprises of 3 registers of $\log_2 N$ bit each, one for storing the cache line index L that is being accessed, one for current index that is stored in the node and one for storing the index that would be stored in the register at the end of second clock pulse. For sorting, the cache line index L that is accessed is given to the working set.

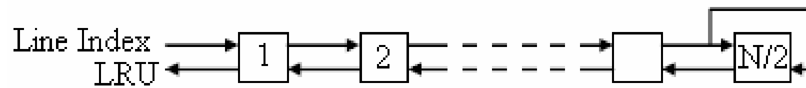


Fig A.10 Systolic Array Implementation

L value passes through each node of the array and is compared with the current indices till the match is found when the Match bit of the node is set to one. When the M bit is set the nodes start copying the value of current index from the adjacent node. The L value passes through all the nodes and finally get settled and deposited in the end node. The forward signal between two adjacent nodes carries L index and M bit and backward signal carries the current index of the node to the backside node when match M bit is set. The last node is wired to itself for L value to get deposit in the Last node, as it is MRU. The list slowly updates over many cycles but maintains the information correctly. When there is a cache miss, the line for replacement is the value in the current index of the first systolic node of the working set and the line index is fed back for updating the access. This reduces the time required for searching the line for replacement. The hardware implementation is accommodated for one cache line access per cycle. This is obtained by the use of two index registers, one $\log_2 N$ bit register for storing current index, one single bit register to store match M, a $\log_2 N$ bit multiplexer, a $\log_2 N$ bit comparator to compare the L value and current index and OR gate. And every node is made to share the L register, which is again $\log_2 N$ bit.

A.5 DISCUSSION OF RESULTS

The Simulations carried out established the functional correctness of the various LRU hardware implementations. Each implementation was simulated with the cache for associativity of 2, 4, 8, 16 and 32. Cache size of 128KB, with line size of 32 bytes, word size 32 bits, using write-back and write allocate policies was the design configuration. The syntheses of different implementations were carried out using the FPGA Advantage 5.2, Leonardo Spectrum Level 3 v2001_1d.45, from Exemplar Logic Inc. The library used for the synthesis is ASIC SCL05u library with ±5 Volt and 300C design parameters.

The graph in Fig. A.11 shows the variation of the number of gates per cache set with the associativity. Fig. A.12 shows the Storage size that the different implementations occupy in the entire cache. The cache considered is a 128KB cache with line size of 32 words. Each word is 4 bytes The design parameter, B for Skewed and Phase implementations used is equal to the associativity when the associativity is less than 16 and equal to 16 for higher associativities. Square Matrix, Skewed Matrix, Counter and Systolic Array show better results and consume less space when the associativity is smaller but at higher associativity Link List, Systolic Array and Skewed Matrices perform better. The graph in Fig. A.13(a) and Fig. A.13(b) shows the growth of the area with increase in Associativity. 2-way Set Associtivity is taken as the reference and the ratio of the number of gates of all associativites with that of 2-way Set Associative is plotted. Fig. A.13 (a) shows the results per cache set and Fig. A.13 (b) shows the result for entire cache. It can be observed that the growth rates are not uniform for various implementations although the growth rates increase for all implementations. The number of gates for one cache line with change of associativity is plotted in the graph of Fig. A.14. It follows the same trend as the Fig. A.11. The response curve in the Fig. A.11 and Fig. A.14 for Phase and Skewed implementations is because after 16-way associativity the design parameter B differs from associativity N. Based on the trends of the size of hardware for associativities ranging from 2 to 32, Fig. A.15 gives the projections for a 128KB cache.

When the associativity is small all the implementations have more or less the same storage to log information but the small difference in area occupied arises due to

difference in the complexity of logic as is observed in Fig A.11 and A.12. The associated logic is the dominating factor determining the area occupied for small associativity. From the graphs we can infer that the Square Matrix method occupies the largest area followed by Skewed Matrix, Counter, Systolic, Phase and Link list implementations. The comparison of Skewed Matrix and Square Matrix clearly indicates that at higher associativity storage space must be the criterion that decides the area required. From Fig A.15 it can be observed that for 128-way set associativity Square Matrix uses 3 times as many gates as Systolic Array, 2.2 times as many gates as Link List, 1.6 times as many gates as Counter implementation. Hence, for high-associative cache the implementations that score well are Link List, Skewed Matrix and Systolic methods, which conserve the storage space. Systolic array has the least area requirement as it has small storage space and also does not employ too much logic to update the list quickly. The counter implementation that has the least storage space for the data requires a large area suggesting the fact that reducing the space alone for high-associativity cache would not provide good results. Skewed and Phase implementations have same characteristics as they use the same storage area for information although the associated logic is different. Link List and Phase implementations have the least growth in area. For link list the size of the components involved increases rather than the number of components and for the Phase implementation the number of components increases but the size remains the same. The LRU implementations that involve smaller storage space with little increase in component size or number of components, show better behavior with increasing associativity. The size of the hardware gives some indications to the delay involved. As the associativity increases the size of different implementations increase indicating that associated delay to retrieve the LRU cache line also increases. The amount of increase in delay for Link List, Systolic and Phase implementations is smaller as the increase in number of gates with increase in associativity, is much slower as compared to other implementations.

It is inferred that for high-associative cache conservation of space to store data of the schemes is important but the associated logic cannot be totally neglected. In high-associativity cache Link List, Systolic Array and Skewed Matrix are the designs most

suitable for implementations, and also with increase in associativity the Link List, Systolic and Skewed Matrix methods would involve less delay. Although the implementation size for one set grows rapidly for increase in associativity, the similar increase when the entire cache is considered is much less. The results also show that the LRU implementations, which involve smaller storage space with little increase in component size or number of components, show better behavior with increasing associativity. Finally of all the implementations, Systolic and Link List show the best performance.

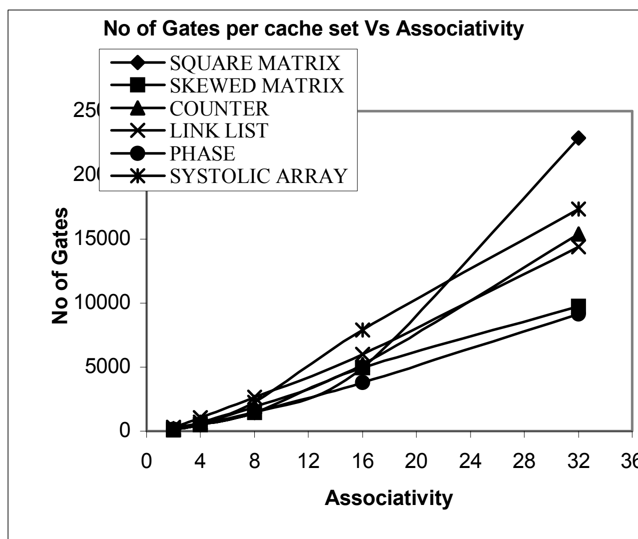


Fig A.11 No. of Gates per cache set vs Associativity

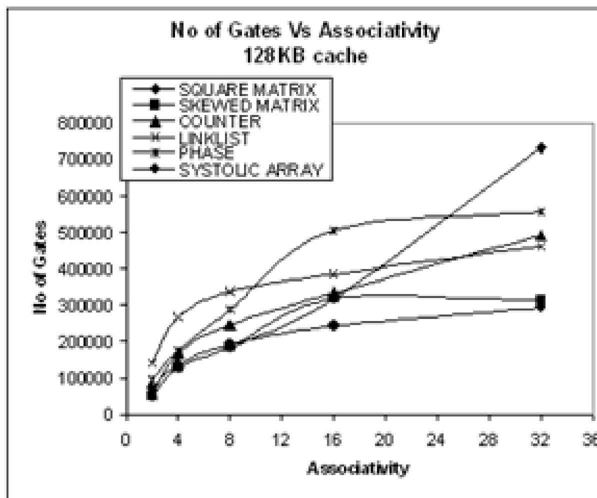


Fig A.12 No. of Gates vs Associativity for 128KB cache

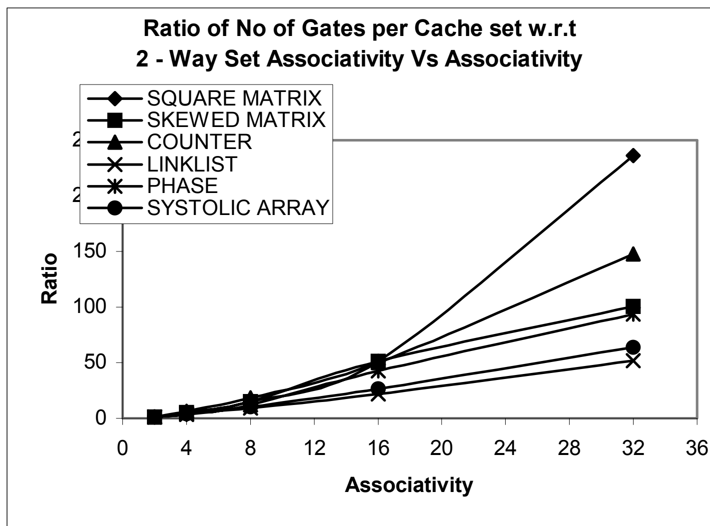


Fig A.13 (a) Ratio of No. of Gates per Cache Set w.r.t 2-way Set Associativity vs Associativity

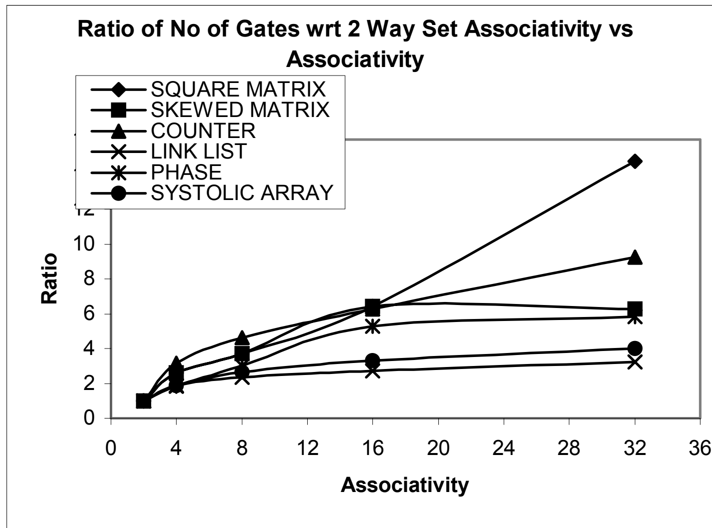


Fig A.13 (b) Ratio of No. of Gates for entire cache w.r.t 2-way Set Associativity vs Associativity

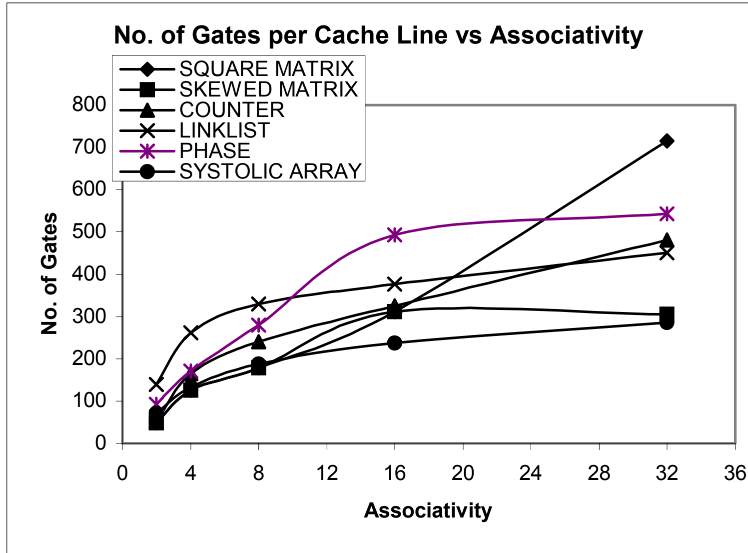


Fig A.14 No. of Gates per Cache Line vs Associativity

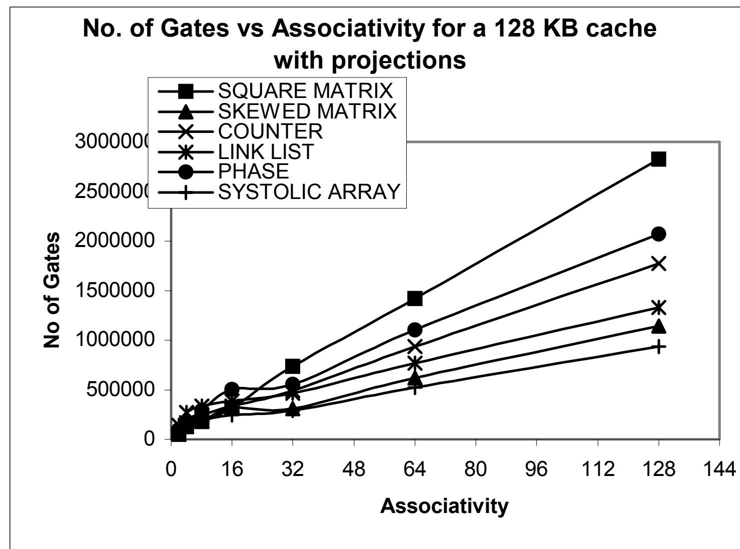


Fig A.15 No. of Gates vs Associativity for a 128 KB cache

A.6 SUMMARY

This discussed the implementation of the least recent used replacement policy for caches with high-associtivity. High-associativity cache with LRU as replacement policy is a good configuration for reducing miss rate in the cache design and enriching the performance in many applications, high-end servers, workstation and modern processors. Implementing LRU policy in hardware for high associativity is difficult. Implementation objectives are identified and various designs, namely Square Matrix, Skewed Matrix, Counter, Phase, Link List and Systolic Array are implemented and the results are analyzed. It is inferred that for high-associativity, conservation of space to store data of the scheme is important but the associated logic cannot be totally neglected. In high-associativity cache, Link List, Systolic Array and Skewed Matrix are the designs most suitable for implementation. Also with increase in associativity the Link List, Systolic and Skewed Matrix would involve less delay. Although the implementation complexity for a set grows rapidly with increase in associativity, the growth is much less when considered for the entire cache. The results also show that the LRU implementations, which involve smaller storage space with little increase in component size or number of components, show better behavior with increasing associativity. Finally of all the implementations, Systolic and Link List shows better results.

REFERENCES

[Abrams 1995] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams and Edward A. Fox, “Caching Proxies: Limitations and Potentials”, *Proceedings of the 4th International World Wide Web Conference,* Boston, MA, 1995, pp 119-133.

[Aggarwal 1999] Charu Aggarwal, Jeol L Wolf and Philip S Yu, “Caching on the World Wide Web”, *IEEE Transaction on Knowledge and Data Engineering*, vol. 11, no.1, 1999, pp 94-107.

[Ahuja 2002] Sadhana Ahuja, Tao Wu and Sudhir Dixit, “Cache On Demand”, *IEEE International Conference on Multimedia & Expo,* Switzerland, Aug 2002.

[Ailamaki 2000] Anastassia Ailamaki, “Architecture-Conscious Database Systems” Ph.D. dissertation, Univ. of Wisconsin, Madison, Department of Computer Sciences, Wisconsin, 2000.

[Al-Zoubi 2004] Hussein Al-Zoubi, Aleksandar Milenkovic and Milena Milenkovic, “Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite” *Proceedings of 42nd annual ACM South East Regional Conference*, Huntsville, Alabama, USA, April 2004, pp 267-272.

[Arlitt 1996] M. F. Arlitt and C. L. Williamson, “Web Server Workload Characterization: The search for invariants,” *Proceedings of SIGMETRICS 96,* Philadelphia, PA, 1996, pp. 126-137.

[Avinoam 2000] Avinoam N Eden, Brain W Joh and Trevor Mudge, “Web Latency Redution via Client-Side Prefetching”, *Proceedings of IEEE International Symposium on Performance Analysis of Systems & Software*, Austin, Texas, 2000, pp 193-200.

[Balamash 2004] Abudallah Balamash and Marwan Krunz, "An Overview of Web Caching Replacement Algorithms", *IEEE Communications: Surveys and Tutorials*, vol. 6, No. 2. 2004, pp 44-56.

[Bahn 2002] Hyokyung Bahn, Kern Koh, Sam H. Noh, Sang Lyul, "Efficient Replacement of Nonuniform Objects in Web Caches", *IEEE Computer*, vol. 35, No. 6, June 2002, pp 65-73.

[Baquero 1995] C. Baquero, V. Fonte, F. Moura, and R. Oliveira, "MobiScape: WWW Browsing under Disconnected and Semi-Connected Operation", *Proceedings of First Portuguese WWW National Conference*, Braga, Portugal, July 1995.

[Barbara 1994] D. Barbara and T. Imielinski, "Sleepers and Workaholics: Caching in Mobile Distributed Environments," *Proceedings of 1994 ACM-SIGMOD Int'l Conf. Management of Data*, June 1994, pp. 1-12.

[Barish 2000] Greg Barish and Katia Obraczka, "World Wide Web Caching: Trends and Techniques", *IEEE Communications Magazine*, vol 38, no 5, 2000, pp 178-184.

[Belady 1966] Belady, L. A., "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal,* vol. 5, no. 2, 1966, pp. 78-101.

[Bhatti 2000] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. "Integrating user perceived quality into Web server design" *Proceedings of the 9th International World Wide Web conference on Computer networks: The International Journal of Computer and Telecommunications Networking*, Amsterdam, The Netherlands, 2000, pp 1-16.

[Bhide 2002] Manish Bhide, Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham and Prashant J Shenoy, "Adaptive Push-Pull: Disseminating Dynamic Web Data", *IEEE Transactions On Computers*, vol 51, no. 6, 2002, pp 652-668.

[Brady 1986] James T. Brady. “A theory of productivity in the creative process”, *IEEE Computer Graphics and Applications*, 6(5), 1986, pp 25-34.

[Breslau 1999] Lee Breslau, Pei Cao, Li Fan, Graham Phillips and Scott Shenker, “Web Caching and Zipf-Like Distributions: Evidence and Implications,” *Proc. IEEE INFOCOM Conf.*, 1999, pp. 126–134.

[Brewington 2000] Brian Brewington and George Cybenko, **“**How Dynamic is the Web?” *Proceedings of the 9th International World Wide Web conference on Computer networks: The International Journal of Computer and Telecommunications Networking*, Amsterdam, The Netherlands 2000, pp: 257 - 276

[Calzarossa 2003] M. Calzarossa and G. Valli. “A Fuzzy Algorithm for Web Caching”, *Proceedings of SPECTS 03*, SCS Press, 2003.

[Cao 1997] P. Cao and S Irani, “Cost-aware WWW proxy caching algorithms” *Proceedings of USENIX Symposium Internet Technologies and Systems*, Monterey, CA, 1997, pp 193-206.

[Cao 1998] P. Cao, J. Zhang, and K. Beach, “Active Cache: Caching Dynamic Contents (Objects) on the Web," *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, The Lake District, England, Sept. 1998, pp. 373-388.

[Chang 2001] Ray-I Chang, Shin-Hung Chang, and Yen-Jen Oyang, “OC: An Optimal Cache Algorithm for Video Staging”, *Proceedings of International Conference on Networking,* August 2002

[Chang 2002] Shin-Hung Chang, Ray-I Chang, Jan-Ming Ho and Yen-Jen Oyang, “An Effective Approach to Video Staging in Streaming Applications”, *Proceedings of IEEE Global Telecommunications Conference,* November 2002, Vol 2, pp. 1733-1737.

[Chankhunthod 1996] A. Chankhunthod, P. Danzig, and C. Neerdaels, "A hierarchical internet object cache," *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, California, Jan. 1996, pp. 153--163.

[Clark 2001] Lawrence T. Clark, Eric J.Hoffman, Jay Miller, Manish Biyani, Yuyun Liao, Stephen Strazdus, Michael Morrow, Kimberley E.Velarde and Mark A.Yarc, "An Embedded 32b microprocessor core for low power and high performance applications," *IEEE Journal of Solid State Circuits*, vol. 36, no. 11, 2001, pp 1599-1608.

[Cohen 1998] E Cohen, B Krishnamurthy, and J Rexford, "Evaluating server-assisted cache replacement in the Web", *Proceedings of the 6th European Symposium on Algorithms*. Lecture Notes in Computer Science, vol. 1461, Springer-Verlag, Germany, 1998, pp 307–319.

[Creres 1998] Ramn Creres, Fred Douglis, Anja Feldmann, Gideon Glass, Michael Rabinovich, "Web Proxy Caching: The devil is in the details", *Workshop on Internet Server Performance*, Madison, WI, June 1998.

[Davison 1999] B D Davison, "Adaptive Web Prefetching," *Position Paper in Proceedings of the 2nd workshop on Adaptive Systems and User Modelling on the World Wide Web*, Toronto, May 1999.

[Davison 2001]B D Davison, "A Web Caching Primer," *IEEE Internet Computing*, 5, July 2001. pp 38-45.

[Davison 2002] B D Davison, "The Design and Evaluation of Web Prefetching and Caching Techniques", *Ph.D. dissertation*. Department of Computer Science, Rutgers University, New Brunswick, NJ. October 2002.

[Deville 1992] Yannick Deville and J. Gobert, "A class of replacement policies for medium and high-associativity structures," *ACM SIGARCH Computer Architecture News*, vol. 20, no.1, 1992, pp. 55-64.

[Douglis 1997] F. Douglis, A. Haro, and M. Rabinovitch, "HPP: HTML macro-preprocessing to support dynamic document caching", *Proceedings of the Symposium on Internetworking Systems and Technologies*, USENIX, December 1997, pp 83-94.

[Eden 2000] Avinoam N Eden, Brain W Joh and Trevor Mudge, "Web Latency Redution via Client-Side Prefetching", *Proceedings of IEEE International Symposium on Performance Analysis of Systems & Software*, Austin, Texas, 2000, pp 193-200.

[Excalibur 2002] "Excalibur Device Overview", Data Sheet, May 2002
http://www.altera.com/literature/ds/ds_arm.pdf

[Fan 2000] Li Fan, Pei Cao, Jussara Almeida and Andrei Z. Broder, "Summary cache: A Scalable Wide-Area Web Cache Sharing Protocol", *IEEE / ACM Transactions on Networking*, vol 8, no. 3, 2000, pp 281-293.

[Forman 1994] G H Forman and J Zahorjan, "The Challenges of Mobile Computing", *IEEE Computer*, 27 (6), 1994, pp 38-47.

[Foxwell 1998]Harry J. Foxwell and Daniel A Menasce, "Prefetching Results of Web Searches", *Proceedings of International Computer Measurement Group Conference*, Anaheim, CA, 1998, pp 602-609.

[Goldberg 2004] David E Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning", Pearson Ed., New Delhi, 2004.

[Gopalan 2002] Kartik Gopalan and Tzi-cker Chiueh, “Improving Route Lookup Performance Using Network Processor Cache", *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Baltimore, Maryland, November 2002, pp 1-10.

[Grossman 2002] J.P. Grossman, “A Systolic Array for Implementing LRU Replacement”, Project Aries Technical Memo ARIES-TM-18, AI Lab, M.I.T., Cambridge, MA, March 13, 2002.

[Han 1998] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret and J. Rubas,“Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing,” *IEEE Personal Communications*, vol. 5, no. 6, 1998, pp. 8-17.

[Hennessy 2003] John L Hennessy and David Patterson, “Computer Architecture: A Quantitaive Approach”, Morgan Kaufmann, 3rd ed., 2003.

[Housel 1996] Barron C. Housel and David B. Lindquist, “WebExpress: A system for optimizing Web browsing in a wireless environment”, *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, November 1996, pp 108-116

[Hwang 1993] K. Hwang, “Advanced Computer Architecture: Parallelism, Scalability and Programmability”, New York, McGraw-Hill Book Co., 1993.

[Irani 1997] S. Irani. “Page replacement with multi-size pages and applications to web caching”, *Proceedings for the 29th Symposium on Theory of Computing*, 1997, pp 701-710.

[Jerkins 2003] Lawrence Jerkins and R Radhika, “Multimedia Proxy caching for video Streaming Applications”, *National Conference on Networking*, IIT Chennai, 2003.

[Jiang 1998] Z. Jiang and L. Kleinrock, “Web prefetching in a mobile environment,” *IEEE Personal Communications*, vol. 5, no. 5, 1998, pp.25-34.

[Jin 2000a] S. Jin and A. Bestavros, “Sources and Characteristics of Web Temporal Locality,” *Proceedings of IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computing and Telecommunication Systems*, San Francisco, CA, 2000, pp. 28–35.

[Jin 2000b] S. Jin and A. Bestavros, “Popularity-Aware Greedy-dual Size Web Proxy Caching Algorithms,” *Proceedings of IEEE International Conference on Distributed Computing Systems*, Taiwan, 2000, pp.254–261.

[Jing 1999] J. Jing, A. Helal, and A. Elmagarmid, “Client-Server Computing in Mobile Environments,” *ACM Computing Surveys*, vol. 31, no. 2, 1999. pp. 117-157.

[Jouppi 1990] Norman P. Jouppi, “Improving Direct mapped Cache Performance by the addition of a small fully-associative cache and prefetch buffers”, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, Washington, 1990, pp 364-373.

[Katsaros 2004] D Katsaros and Y Manolopoulos, “Caching in Web memory hierarchies”, *Proceedings of ACM Symposium on Applied Computing*, Nicosia, Cyprus, 2004. pp 1109-1113.

[Kosko 1994] Bart Kosko, *Neural Networks and Fuzzy Systems*, Prentice Hall of India, New Delhi, 1994.

[Lee 2002] Sung-Ju Lee, Wei-Ying Ma, and Bo Shen, “An Interactive Video Delivery and Caching System Using Video Summarization”, *Computer Communications*, vol. 25, no. 4, Mar. 2002, pp. 424—435.

[Maltzahn 1997] Carlos Maltzahn, Kathy Richardson and Dirk Grunwald, "Performance issues of Enterprise-level Web Proxies", *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modelling of Computer Systems,* June 1997, pp 13-23.

[Markatos 1998] E. P. Markatos and C. E. Chronaki, "A Top-10 Approach to Prefetching on the Web," *Proceedings of INET 1998*, Jul. 1998

[Mattson 1996] R.L. Mattson, J. Gecsei, D.R. Slcitz and I.L. Traiger, "Evaluation techniques for storage hierarchies," *IBM system journal*, 1996, pp. 169-193.

[Michel 1998] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson, "Adaptive Web Caching: Towards a New Global Caching Architecture" *Computer Networks and ISDN Systems,* vol 30, 22-23, 1998, pp 2169-2177.

[Motwani 1995] R. Motwani and P Raghavan, "Randomized Algorithms", Cambridge Univ. Press, Cambridge, UK, 1995.

[Nanopoulos 2003] A Nanopoulos, D Katsaros and Y Manolopoulos, "A data mining algorithm for generalized Web prefetching", *IEEE Transaction on Knowledge and Data Engineering*, 15(5), 2003, pp 1155-1169.

[Papermaster 1998] Mark Papermaster, Robert Dinkjian, Michael Mayfield,Peter Lenk, Bill Ciarfella, Frank O'Connell and Raymond DuPont , "POWER3: Next Generation 64-bit PowerPC Processor Design", White Paper, IBM Corporation, 1998. *http://www-.ibm.com/ servers/eserver/pseries/hardware/whitepapers/power3wp.pdf*

[Patterson 2005] David A. Patterson and John L. Hennessy, "Computer Organization and Design", 3rd ed., Morgan Kaufman, New Delhi, 2005.

[Parker 1998] Tammy Parker. "Mobile Wireless Internet Technology Faces Hurdles," *Computer*, vol. 31, no. 3, 1998, pp. 12-14.

[Pasquale 2002] J. Pasquale, E. Hung, T. Newhouse, J. Steinberg and N. Ramabhadran, “Improving Wireless Access to the Internet By Extending the Client/Server Model”, *Proceedings of European Wireless Conference,* Florence, Italy, Feb. 2002, pp. 670-676.

[Pitkow 1994] J Pitkow and M Recker, “A simple yet robust caching algorithm based on dynamic access patterns”, *Proceedings of the 2nd International World Wide Web Conference*, Chicago, 1994, pp 1039-1046.

[Podlipnig 2003] Stefan Podlipnig and Laszlo Boszormenyi, “A Survey of Web Cache Replacement Strategies”, *ACM Computing Surveys*, vol 35, no. 4, 2003, pp 374-398.

[Povey 1997] D. Povey and J. Harrison, “A Distributed Internet Cache", *Proceedings of the 20th Autralasian Computer Science Conference*, Sydney, Australia, February 1997, pp. 175-184.

[Psounis 2002] K Psounis and B Prabhakar, “Efficient Randomized Web-Cache Replacement Schemes Using Samples From Past Eviction Times”, *IEEE/ACM Transactions on Networking,* vol 10, No. 4, August 2002, pp 441-455.

[Rabinovich 1998] M. Rabinovich, J. Chase, and S. Gadde, “Not all hits are created equal: cooperative proxy caching over a wide-area network”, *Computer Networks And ISDN Systems*, 30, 22-23, 1998, pp. 2253-2259.

[Rabinovich 2002] Michael Rabinovich and Oliver Spatscheck, “Web Caching and Replication”, Addison-Wesley, 2002.

[Rajasekaran 2003] S. Rajasekaran and G A Vijayalakshmi Pai, " Neural Networks, Fuzzy Logic, and Genetic Algorithms: Synthesis and Applications", Prentice Hall of India, New Delhi, 2003.

[Rejaie 1999] Reza Rejaie, Mark Handley, Haobo Yu and Deborah Estrin, "Proxy Caching Mechanism for Multimedia Playback Streams in the Internet", *Proceedings of 4th International Web Caching Workshop*, San Deigo, CA, April, 1999.

[Rejaie 2000] R. Rejaie, H. Yu, M. Handley, and D. Estrin., "Multimedia Proxy Caching Mechanism for Quality Adaptive Streaming Applications in the Internet", *Proceedings of the IEEE INFOCOM*, Tel-Aviv, Isreal, Mar. 2000.

[RFC 2186] D.Wessels and K. Claffy, Internet cache protocol (IPC), version 2, RFC 2186.

[Roast 1998] Chris Roast. "Designing for delay in interactive information retrieval", *Interacting with Computers*, 10(1), 1998, pp 87-104.

[SA110 2000] "SA-110 Microprocessor,Technical Reference Manual," December 2000. http://renan.org/ARM/doc/27805802.pdf

[Saha 2001] Subhasis Saha, Mark Jamtgaard and John Villasenor. "Bringing the Wireless Internet to Mobile Devices", *Computer*, vol. 34, no. 6, 2001, pp. 54-58.

[Satyanarayanan 1996] M Satyanarayanan, "Fundamental Challenges in Mobile Computing", *Proceedings of the Fifteenth Annual ACM symposium on Principles of Distributed Computing*, Pennsylvania, 1996, pp 1-7

[Scheuermann 1997] Peter Scheuermann, Junho Shim and Radek Vingralek, "A Case for Delay-Conscious Caching of Web Documents",*Computer Networks and ISDN Systems*, vol 29, no. 8--13, 1997, pp 997-1005.

[Sen 1999] S. Sen, J. Rexford, and D. Towsley, "Proxy prefix caching for multimedia streams", *Proceedings of IEEE Infocom*, 1999, pp 1310-1319.

[Shankarnarayanan 2002] N K Shankarnarayanan, Anupam Rasotgi and Z Jiang, "Performance of a Wireless Data Network with Mixed Interactive User Workloads", *Proceedings of IEEE International Conference on Communications*, NewYork, May 2002.

[Silberschatz 2001] A. Silberschatz and P. Galvin, Operating System Concepts, 6th Ed. Addison-Wesley, 2001

[Smith 1982] A .J Smith, "Cache Memories," *ACM Computing surveys*, vol. 14, no. 3, 1982, pp. 473-500.

[Smith 1985] J. E. Smith and J. R. Goodman," Instruction cache replacement policies and organizations," *IEEE Transactions on Computers*, vol. C-34, no. 3, 1985, pp. 234–241.

[Sugumar 1993] Sugumar R.A and Abraham S.G, "Efficient simulation of caches under optimal replacement with application to miss characterization," *Proceedings of the ACM SIGMETRICS conference on Measurement and modeling of computer system*, May1993, pp. 24-35.

[Squid 1998] http://www.squid-cache.org/

[Tan 2001] Kian Lee Tan, Jun Cai and Beng Chen Ooi, "An Evaluation of Cache Invalidation Strategies in Wireless Environments", *IEEE Transactions on Parallel and*

Distributed Systems, vol 12, no. 8, 2001, pp 789-807.

[Tiwari 1998] Renu Tewari, Harrick M., Vin, Asit and Dinkar Sitaramy, “Resource-based Caching for Web Servers”, *Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking*, San Jose, Vol. 3310, 1998, pp 191-204.

[Tiwari 1999] Renu Tewari, M. Dahlin, H. Vin, and J. Kay, "Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet", *Proceedings of the 19th International Conference on Distributed Computing Systems*, Austin, TX, June 1999

[Traces 1995] http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html

[Vakali 1999a] Athena Vakali, “ A Web-based evolutionary model for Internet Data Caching”, *Proceedings of 10th International Workshop on Database & Expert Systems Applications*, Italy, Sept 1999 pp 650-654

[Vakali 1999b] A. Vakali, “ An Evolutionary scheme for Web Replication and Caching”, *Proceedings of the 4th International Web caching Workshop, Work in Progress Session*, San Diego, California, Mar-Apr 1999.

[Valloppillil 1998] Vinod Valloppillil and Keith W. Ross. Cache Array Routing Protocol v1.0, INTERNET DRAFT, 1998.

[Video 1995] www3.informatik.uni-wuerzburg.de/MPEG/traces/.

[Wang 1996] Z. Wang and J. Crowcroft. “Prefetching in World Wide Web”, *Proceedings of IEEE Global Internet*, 1996, pp 28-32.

[Wang 1999] J. Wang, “A Survey of Web Caching Schemes for the Internet,” *ACM Computer Communication Review*, vol. 29, no. 5, 1999, pp. 36–46.

[Wessels 1998] Duane Wessels and K Claffy, “ICP and Squid Web Cache”, *IEEE Journal on Selected Areas in Communication*, vol 16, no. 3, 1998, pp 345-357.

[Wills 1999] Craig E. Wills and Mikhail Mikhailov, “Towards a better understanding of web resources and server responses for improved caching” *Proceedings of the Eighth International World Wide Web Conference*, Toronto, Canada, 1999, pp 1231-1243.

[Wolisz 2000a] Adam Wolisz, “Wireless Internet Architectures: Selected Issues”, *Personal Wireless Communications*, Poland, Kluwer, September 2000, pp 1-16.

[Wolisz 2000b] Adam Wolisz, C Hoene, B Rathke and M Schlager, “Proxies, Active Networks, Re-configurable Terminals: The Cornerstones of Future Wireless Internet”, *Proceedings. of IST Mobile Communications Summit*, Galway, Ireland, October 2000, pp. 795-803

[Wong 2000] Wayne A. Wong and Jean-Loup Baer, “Modified LRU policies for improving second-level cache behavior,” *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA)*, Toulouse, France, January 2000, pp 49–60.

[Wong 2001] K. Y. Wong, S. Y. Hui and K. H. Yeung, "An Adaptive Network Prefetch Scheme in a Transcoding Environment", *Proceedings of 5th World Multiconference on Systemics, Cybernetics, and Informatics*, vol. XII, Orlando, U.S.A, July 2001, pp. 229-233.

[Wooster 1997] R. P. Wooster and M. Abrams, "*Proxy caching that estimates page load delays*," Computer Networks and ISDN Systems, vol. 29, no. 8-13, 1997 pp. 977—986.

[Williams 1996] S Williams, M Abrams, C R Standridge, G Abdulla and E A Fox, “Removal policies in network caches forWorld-WideWeb documents”, *Proceedings of ACM SIGCOMM*. ACM Press, New York, NY, 1996, pp 293–305.

[Wu 2001] K.L.Wu, P.S.Yu, and J.L.Wolf, “Segment-based proxy caching of multimedia Streams”, *Proceedings of the l0th International Conference on World Wide Web*, Hong Kong, 2001, pp 36-44.

[Yeung 2003] K. H. Yeung, S. Y. Hui, and K. Y. Wong, "An Intelligent Proxy Server for Mobile Communication Systems", *SSGRR Conference*, Italy, Aug. 2003.

[Young 1994] N. Young “The k-server dual and loose competitiveness for paging” *Algorithmica*, vol. 11, no.6, 1994, pp 525-541.

[Zhang 2000] Zhi-Li Zhang, Yuewei Wang, David H. C. Du, and Dongli Su, “ Video Staging: A Proxy-Server-Based Approach to End-to-End Video Delivery over Wide-Area-Networks”, *IEEE/ACM Transaction on Multimedia*, vol. 8, no. 4, 2000, pp 429-442.

[ZhangC 1997] Chenxi Zhang, Xiaodong Zhang, and Yong Yan, “Two Fast and High-Associativity Cache Schemes,” *IEEE Micro Magazine*, vol. 17, no. 5, 1997, pp. 40-49.

[ZhangM 2000] Michael Zhang and Krste Asanavio, “Highly-Associative Caches for Low-Power Processors”, *Kool Chips Workshop, 33rd International Symposium On Microarchitecture*, Monterey, California, December 2000.

[Zona 1999] “The economic impacts of unacceptable Web site download speeds”, White paper, Zona Research, 1999. Available from: www.ebtwg.org/wp_**download**speed.doc

List of Publications

Journal Papers

1. T S B Sudarshan, G Raghurama, "Fuzzy Logic Approach for Replacement Policy in Web Caching", International Journal of Systemics, Cybernetics and Informatics (Communicated)

2. T.S.B. Sudarshan, Rahil Abbas Mir, Vijayalakshmi S, G Raghurama, "LRU Hardware Implementations for High Associativity Caching of Uniformed Objects", IET Transaction of Digital Electronics and Computers. (Communicated)

3. Ganesh T S, M.T.Fredrick, T S B Sudarshan, A K Somani, "Hash Chip: A Shared-Resource Multi-Hash Function Processor Architecture on FPGA", Integration, The VLSI Journal, Volume 40, Issue 1, January 2007, pp 11-19.

4. Biju Raveendran, T S B Sudarshan, and S Gurunarayanan, "Cache Memory Design with Late Replacements for Embedded Systems", International Journal of Lateral Computing, (Accepted, To Appear)

5. T S B Sudarshan, Rahil Abbas Mir, Vijayalakshmi S "DRIL-A Flexible Architecture for Blowfish Encryption using Dynamic Reconfiguration, Replication, Innerloop pipelining, Loop-folding techniques", Lecture Notes in Computer Science, Advances in Computer Systems Architecture, Volume 3740, 2005, pp 625 – 639.

Conference Papers Related to Thesis

6. T S B Sudarshan, G Raghurama, " Dual Stage Victim Cache based approach for Web Caching", Proceedings of National Conference on Signal Processing, Intelligent Systems & Networking Bangalore, India, December 2003.

7. T S B Sudarshan, J Sahidhar, G Raghurama, "Multimedia Proxy Caching Algorithms for Streaming Objects.", International Conference on Recent Trends and New Directions of Research in Cybernetics & Systems Theory, IASST, Guwahati, India, January 2004.

8. T S B Sudarshan, A Ganesh, G Rahurama, "Genetic Algorithm Based Approach For Replacement Policy in Web Caching", Proceedings of International Conference on Systemics, Cybernetics and Informatics, Hyderabad, India, January 2005, pp 803-806.

9. T S B Sudarshan, Rahil Abbas Mir, Vijayalakshmi S "Highly Efficient LRU Implementations for High Associativity Cache Memory", Proceedings of 12th IEEE International Conference on Advanced Computing and Communications, Ahemdabad, Gujarat, Allied Publishers Pvt. Ltd., India, December 2004, pp 87-95.

10. T S B Sudarshan, Pavankiran, Swetha Krishnan and G Raghurama, "Fuzzy Logic Approach for Replacement Policy in Web Caching", Proceedings of 2nd IEEE Indian International Conference on Artificial Intelligence, Pune, December 2005, .ISBN: 0-9727412-1-6, pp 2308-2319.

11. T S B Sudarshan, A Ganesh, G Raghurama, "Caching and Replacement of Streaming Objects Based on a Popularity Function", Proceedings of Third IASTED International Conference on Communications and Computer Networks, Marina Del Rey, California, USA, ACTA Press, October 2005, pp 208-212.

12. T S B Sudarshan, Ganesh Ananthanarayanan, G Raghurama, "Genetic Algorithm Based Approach for Replacement Policy in Multimedia Web Caching", (Accepted) 4th International Conference on Communications, Internet and Information Technology, Cambridge, USA, November 2005.

Other Conference Papers During Thesis Period

13. T S B Sudarshan, Ganesh T S, "Hardware Architectures for Message Padding in Cryptographic Hash Primitives", Proceedings of 8th IEEE Workshop on Progress in VLSI Design & Test 2004, Mysore, August 2004, pp 136-144.

14. Ganesh T S, T S B Sudarshan, N K Srinivasan, K Jaypal "Pre-Silicon Prototype of a Unified Hardware Architecture for Cryptographic Message Detection Codes", Proceedings of IEEE 2004 International Conference on Field Programmable Technology, Brisbane, Australia, Dec 2004, pp 324-326.

15. Ganesh T S, T S B Sudarshan "ASIC Implementation of a Unified Hardware Architecture for Non-Key Based Cryptographic Hash Primitives", Proceedings of IEEE International Conference on Information Technology Coding and Computing, Las Vegas, Nevada, USA, Published by Nova Science, New York, USA, Vol-I, April 2005, pp 580-585.

16. T S B Sudarshan, Rahil Abbas Mir, Vijayalakshmi S "Dynamic Reconfigurable Architecture for Blowfish Algorithm Using Inner-Loop Pipelining, Loop-folding Technique", Proceedings of IEEE Asia and South Pacific International Conference on Embedded SoCs, IISc, Bangalore, July 2005. (Available on CD)

17. T S B Sudarshan, Rahil Abbas Mir, Vijayalakshmi S "DRIL-A Flexible Architecture for Blowfish Encryption using Dynamic Reconfiguration, Replication, Innerloop

pipelining, Loop-folding techniques", Proceedings of 10th Asia Pacific Computer Systems Architecture Conference, Nanyang Technological University, Singapore, October 2005, pp 625-639.

18. Suresh Sharma, T S B Sudarshan, "Design of an Efficient Architecture For Advanced Encryption Standard Algorithm Using Systolic Structures", Web Proceedings of IEEE International Conference of High Performance Computing, Goa, December 2005. http://www.hipc.org/hipc2005/hipc2005posters.html

19. Ravikumar L, T S B Sudarshan and Bharat Deshpande, "Matching Proximity of Scheduling Algorithms for Grid Computing", Proceedings of Third International Conference on Systemics, Cybernetics and Informatics, Hyderabad, Jan 2006.

20. Ninad B Kothari, T S B Sudarshan, Shipra Bhal, Tejesh E C and S Gurunarayanan, "Design of an Efficient Low-Power AES Engine for Zigbee Systems Progress in VLSI Design & Test, ISBN 81-88901-24-5, Elite Publishing House Pvt. Ltd., Goa, August 2006, pp 264-272

21. Biju Raveendran, T S B Sudarshan, and S Gurunarayanan, "Cache Memory Design with Late Replacements for Embedded Systems", Proceedings of International Conference on Embedded Systems, Mobile Communication and Computing, Bangalore, August 2006, pp 76-90.

22. Biju Raveendran, T S B Sudarshan, S Gurunarayanan, "Selection Placement Data Cache for Low Energy Embedded System", Proceedings of 14th IEEE International Conference On Advanced Computing (ADCOM), NITK, Surathkal, December 2006, pp 473-476. IEEE CATALOG No: 06EX1537, ISBN No: 1-4244-0715-X, LIBRARY OF CONGRESS 2006934023

23. Ninad B Kothari, T S B Sudarshan, S Gurunarayanan, S Chandrashekhar "SOC Design of a Low Power Wireless Sensor Network Node for Zigbee Systems", Proceedings of 14th IEEE International Conference On Advanced Computing (ADCOM), NITK, Surathkal, December 2006, pp 462-466. IEEE CATALOG No: 06EX1537, ISBN No: 1-4244-0715-X, LIBRARY OF CONGRESS 2006934023

24. Amarnath Bhadrashetty, Mandar Raje, T S B Sudarshan, "Enhancement of Advanced Encryption Standard with CBC Mode and Building a Prototype for Secured Communication", Proceedings of 4th International Conference on Systemics, Cybernetics and Informatics, Vol 1, Jan 2007, pp 563-568. (Best Paper & Best Presentation award).

25. Biju Raveendran, T S B Sudarshan, Avinash Patil, Komal Randive, S Gurunarayanan, "Enhancement Predictive Placement Scheme in Set-Associative Cache for Energy Efficient Embedded Systems", Proceedings of 4th IEEE International Conference on Information Technology: New Generations, Las Vegas, Nevada, USA, (Published by the IEEE Computer Society), April 2-4 2007 (Accepted)

BRIEF BIOGRAPHY OF CANDIDATE

T S B Sudarshan is working as Assistant Professor in Computer Science & Information Systems Group in Birla Institute of Technology and Science, Pilani from January 1998. Prior to this he also worked as a faculty member in Bangalore University from 1993 to 1998. He obtained his Bachelors of Engineering (Electrical & Electronics) from Bangalore University, and Masters of Engineering (Systems), from Birla Institute of Technology, Mesra, Ranchi in 1989 & 1993 respectively. He is a life member of Indian Society for Technical Education and Member of IEEE Computer Society. His research interests are Web Caching, Caching for Streaming Media Objects for Fixed and Mobile Networks, Computer Architecture, Crypto-Hardware Design, Memory Design for Embedded Systems.

BRIEF BIOGRAPHY OF SUPERVISOR

Prof. G Raghurama is Professor in Electronics and Instrumentation Engineering Group and Deputy Director (Academic) in Birla Institute of Technology and Science, Pilani. He obtained Masters in Science (Physics) from IIT, Chennai, and Ph.D. (Physics) from the Indian Institute of Science, Bangalore in 1980 and 1986 respectively. He was a member of Technical Advisory Board of Cradle Technologies, USA. He has several publications in National and International Journals. His research interests are in the area of Telecommunications, Networking and Network Management.