

2004

## Building Internet caching systems for streaming media delivery

Songqing Chen

*College of William & Mary - Arts & Sciences*

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



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Chen, Songqing, "Building Internet caching systems for streaming media delivery" (2004). *Dissertations, Theses, and Masters Projects*. William & Mary. Paper 1539623444.  
<https://dx.doi.org/doi:10.21220/s2-76n8-5k12>

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

# Building Internet Caching Systems for Streaming Media Delivery

---

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William & Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

---

by

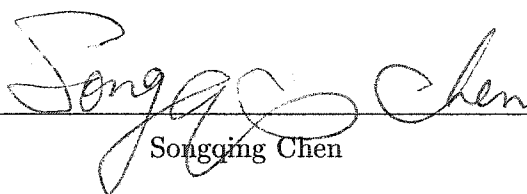
Songqing Chen

2004


## APPROVAL SHEET

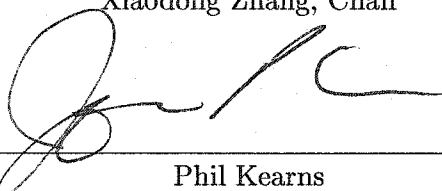
This dissertation is submitted in partial fulfillment of  
the requirements for the degree of

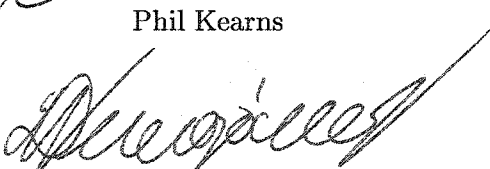
Doctor of Philosophy


  
Songqing Chen


Approved by the Committee, July 2004

  
Xiaodong Zhang, Chair

  
Phil Kearns

  
Dimitrios Nikolopoulos

  
Haining Wang

  
Zhen Xiao  
AT&T Labs - Research

*To my grandparents and parents...*

# Table of Contents

<b>Acknowledgments</b>	<b>ix</b>
<b>List of Tables</b>	<b>xi</b>
<b>List of Figures</b>	<b>xii</b>
<b>Abstract</b>	<b>xv</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Background . . . . .	2
1.2 Dissertation Contributions . . . . .	5
1.3 Dissertation Organization . . . . .	8
<b>2 Segment-based High Quality Streaming Media Proxy Designs</b>	<b>10</b>
2.1 Introduction . . . . .	10
2.2 Related Work . . . . .	14
2.3 Prefetching Methods and Insights into Proxy Jitter . . . . .	16
2.3.1 Look-ahead Window Based Prefetching Method . . . . .	18
2.3.2 Active Prefetching . . . . .	20

2.3.3	Segment-based Proxy Caching and Proxy Jitter Free Strategies . . . .	24
2.3.4	Trade-off Between Low Proxy Jitter and High Byte Hit Ratio . . . .	25
2.4	Byte Hit Ratio vs. Delayed Startup Ratio . . . . .	26
2.4.1	Exponential Segmentation Strategy . . . . .	27
2.4.1.1	Segmentation Method . . . . .	27
2.4.1.2	Admission Policy . . . . .	28
2.4.1.3	Replacement Policy . . . . .	28
2.4.2	Adaptive-Lazy Segmentation Strategy . . . . .	29
2.4.2.1	Aggressive Admission Policy . . . . .	29
2.4.2.2	Lazy Segmentation Method . . . . .	30
2.4.2.3	Two-Phase Iterative Replacement Policy . . . . .	30
2.4.3	Analytical Model . . . . .	31
2.4.4	Performance Objective Analysis . . . . .	34
2.4.4.1	Delayed Start Request Ratio . . . . .	34
2.4.4.2	Byte Hit Ratio . . . . .	34
2.4.5	Performance Bound Analysis . . . . .	39
2.4.5.1	Delayed Start Request Ratio . . . . .	39
2.4.5.2	Byte Hit Ratio . . . . .	41
2.4.6	Analytical Results . . . . .	43
2.4.7	Improved Adaptive-Lazy Segmentation Strategy . . . . .	44
2.5	The Hyper-Proxy System . . . . .	45
2.5.1	Priority-based Admission Policy . . . . .	47
2.5.2	Active Prefetching . . . . .	48

2.5.3	Lazy Segmentation Policy . . . . .	49
2.5.4	Differentiated Replacement Policy . . . . .	50
2.6	Performance Evaluation . . . . .	52
2.6.1	Workload Summary . . . . .	52
2.6.2	Performance Results . . . . .	53
2.7	Summary . . . . .	58

### **3 Implementation and Evaluation of a Segment-based Streaming Media**

<b>Proxy</b>		<b>60</b>
3.1	Introduction . . . . .	60
3.2	Related Work . . . . .	65
3.3	Implementation of Hyper-Proxy . . . . .	66
3.3.1	Streaming Engine . . . . .	66
3.3.2	Local Content Manager and Scheduler . . . . .	69
3.3.3	Segmentation-Enabled Cache Engine . . . . .	71
3.3.4	Fast Data Path . . . . .	73
3.4	Performance Evaluation . . . . .	75
3.4.1	Test Setup . . . . .	75
3.4.2	Evaluation Metrics . . . . .	76
3.4.3	Experimental Results . . . . .	76
3.4.3.1	Full Caching Approach . . . . .	77
3.4.3.2	Effect of Segment Size . . . . .	79
3.4.3.3	Effect of Proximity . . . . .	81

3.4.3.4	Prefetching Effectiveness . . . . .	85
3.4.3.5	Cache Efficiency Study Using a Real Workload . . . . .	86
3.5	Summary . . . . .	89
<b>4</b>	<b>Shared Running Buffers (SRB) Based Proxy Caching Streaming Sessions</b>	<b>91</b>
4.1	Introduction . . . . .	91
4.2	Related Work . . . . .	96
4.3	Shared Running Buffer (SRB) Media Caching Algorithms . . . . .	97
4.3.1	SRB Algorithm: Related Definitions . . . . .	98
4.3.2	SRB Algorithm . . . . .	100
4.3.2.1	SRB Buffer Lifecycle Management . . . . .	101
4.3.2.2	SRB Buffer Dynamic Reclamation . . . . .	105
4.3.2.3	SRB Buffer Replacement Policy . . . . .	108
4.3.3	Patching SRB (PSRB) Algorithm . . . . .	108
4.4	Performance Evaluation . . . . .	110
4.4.1	Evaluation Metrics . . . . .	110
4.4.2	Performance of the WEB Workload . . . . .	111
4.4.3	Performance of the PART Workload . . . . .	115
4.4.4	Performance of the REAL Workload . . . . .	119
4.4.5	Further Analysis on the Client Channel Requirement . . . . .	121
4.5	Summary . . . . .	123
<b>5</b>	<b>Other Related Work</b>	<b>125</b>
5.1	Coordinating P2P System and Proxy for Streaming Media Delivery . . . . .	125



5.2	Detective Browser . . . . .	126
5.3	Cooperatively Shared Proxy Browsers . . . . .	127
5.4	Dynamic Load Sharing With Unknown Memory Demands in Clusters . . . .	128
5.5	Adaptive Memory Allocations in Clusters to Handle Large Data-Intensive Jobs	129
<b>6</b>	<b>Conclusion and Future Work</b>	<b>131</b>
6.1	Conclusion . . . . .	131
6.2	Future Work . . . . .	133
6.2.1	Streaming Based on Proxy-assisted Transcoding . . . . .	134
6.2.2	Cooperative Streaming Proxy to Support Mobile Computers . . . . .	135
6.2.3	Live Streaming Enabled Proxy . . . . .	135
	<b>Bibliography</b>	<b>137</b>

## ACKNOWLEDGMENTS

The past five years at William and Mary has been an exciting and challenging period in my life. On my way to the Ph.D, I have obtained many helps from people around me. I would like to take this opportunity to express my gratitude to them.

I would like to first thank my advisor, Dr. Xiaodong Zhang. He is a role model of hard working, dedication, adopting a rigorous approach, and setting a high standard to research. He has spent a lot of time on discussing the research projects with me and shaping my research visions. He has been a constant source of inspiration whenever I need directions and encouragement. It is with my deepest gratitude that I acknowledge him.

I also want to thank Drs. Phil Kearns, Dimitrios Nikolopoulos, Haining Wang and Zhen Xiao to serve my committee and give me valuable input to my dissertation. I learned a lot from Phil on systems. I also benefit from Dimitrios's knowledgeable background across many areas. I have known Haining and Zhen since 2001 in the ICDCS conference of the year, and we have become collaborators on different projects.

The department of computer science at William and Mary is a great place to pursue my graduate study. I have acquainted many people here. William Bynum and Teresa Long, who have been reviewing many of my manuscripts, have taught me a lot on English writing. They have been always encouraging and rigorous. Alumni and current members of HPCS Lab, Zhao Zhang, Li Xiao, Zhichun Zhu, Xin Chen, Song Jiang, Lei Guo, Shansi Ren, Xiaoning Ding, are always good sources to discuss different problems with when I work here. Graduate students in the department are friendly to work with and have fun together. Vanessa Godwin has always been there to take care many things for us. I feel very happy to spend the past five years in such a nice environment.

During the summers of 2002 and 2003, I worked in HP Labs, Palo Alto, CA. I enjoyed not only the shinning weather there, but also strong intellectual interactions with many researchers, including Drs. Bo Shen, Susie Wee, Yong Yan (a former HPCS member), Zhichen Xu (a former HPCS member), Sumit Roy, Sujoy Basu, Wai-tian Tan, John Apostolopoulos, Ackcorn John, Mitch Trott, and many others.

As an international student, I enjoyed the life in the College and the local Williamsburg community very much. People here are warm-hearted and very helpful. Particularly, I want to thank Dan Caprio and Carolyn Caprio, who brought me into their family five year ago when I arrived here and keep helping me on many things.

Without the support from my family and friends, it would be impossible for me to move forward to this point. The love from my grandparents, my parents, my sister and my brother always encourages me when I am facing difficulties. The understanding and support from many of my friends have been invaluable to my journey to the Ph.D.

Finally, I would like to thank two sources of financial support to my dissertation re-

search: (1) National Science Foundation under grants CCF-9812187, CNS-0098055, and CCF-0129883, and (2) Hewlett-Packard Laboratories under a sponsored research agreement.

# List of Tables

2.1	The notations for prefetching study . . . . .	17
2.2	The items of <i>Hyper-Proxy</i> data structure for each object . . . . .	46
2.3	The workload summary . . . . .	53
3.1	The content and access parameters of real workload . . . . .	87

# List of Figures

1.1	Internet access infrastructures: clients access Web servers through proxies .	3
2.1	Byte hit ratio vs. delayed startup ratio . . . . .	43
2.2	WEB: jitter byte ratio and delayed startup ratio . . . . .	54
2.3	WEB: byte hit ratio . . . . .	55
2.4	PART: jitter byte ratio and delayed startup ratio . . . . .	55
2.5	PART: byte hit ratio . . . . .	56
2.6	REAL: jitter byte ratio and delayed startup ratio . . . . .	57
2.7	REAL: byte hit ratio . . . . .	58
3.1	Organization and protocols used in Hyper-Proxy system . . . . .	63
3.2	Internal design of the Hyper-Proxy system: A client request is divided into $n$ sub-requests with different ranges, $R_s^1$ to $R_s^n$ , requesting different content segments, $D_s^1$ to $D_s^n$ . The Local Content Manager and Scheduler controls when to send the next sub-request. The cache engine returns segment meta data ( $M_s^1$ to $M_s^n$ ) to the Local Content Manager and Scheduler, and caches the segments $D_s^1$ to $D_s^n$ on the disk. . . . .	67

3.3	High level abstraction of an MP4 file: SDP represents the SDP information. V-P represents video data packet while A-P represents audio-data packet. A-H represents audio hint track information, while V-H represents video hint track information. The media data packets are accessed from the pointers of SDP and hint track information in the order. . . . .	69
3.4	Performance of full caching approach: startup latency and miss processing	77
3.5	Performance of full caching approach: handshake . . . . .	78
3.6	Performance study with different segment size . . . . .	80
3.7	Client startup latency for local and remote . . . . .	81
3.8	Time to handle a MISS for local and remote . . . . .	82
3.9	Time to handle a handshake for local and remote . . . . .	83
3.10	Client perceived jitter for local and remote . . . . .	84
3.11	Squid handshake time and client perceived jitter . . . . .	85
3.12	Byte hit ratio and server traffic for segment-based caching strategies . . . .	87
3.13	False prefetch by <i>Window</i> and <i>Half</i> . . . . .	89
4.1	Running buffer and interval caching . . . . .	93
4.2	Greedy patching and grace patching . . . . .	95
4.3	Optimal patching . . . . .	96
4.4	SRB memory allocation: the initial buffer freezes its size (1) . . . . .	102
4.5	SRB memory allocation: the initial buffer freezes its size (2) . . . . .	103
4.6	Data sharing among buffers in SRB algorithm . . . . .	104
4.7	SRB memory reclamation: different situations of session termination . . . .	107

4.8	An example of the PSRB algorithm . . . . .	109
4.9	WEB: server traffic reduction and average client channel with 1GB Memory	112
4.10	WEB: storage requirement (%) with 1GB memory . . . . .	112
4.11	WEB: bandwidth reduction and average client channel with the scale of 1/4	114
4.12	WEB: storage requirement (%) with the scale of 1/4 . . . . .	114
4.13	PART: bandwidth reduction and average client channel requirement with 1GB memory . . . . .	115
4.14	PART: average client storage requirement (%) and client waste (%) with 1GB memory . . . . .	116
4.15	PART: bandwidth reduction and average client channel requirement with the scale of 1/4 . . . . .	117
4.16	PART: average client storage requirement (%) and client waste (%) with the scale of 1/4 . . . . .	117
4.17	REAL: bandwidth reduction and average client channel requirement with 1GB memory . . . . .	118
4.18	REAL: average client storage requirement (%) and client waste (%) with 1GB memory . . . . .	119
4.19	REAL: bandwidth reduction and average client channel requirement with the scale of 1/4 . . . . .	120
4.20	REAL: average client storage requirement (%) and client waste (%) with the scale of 1/4 . . . . .	120
4.21	Client channel requirement CDF: WEB and PART . . . . .	121
4.22	Client channel requirement CDF: REAL . . . . .	122

## ABSTRACT

The proxy has been widely and successfully used to cache the static Web objects fetched by a client so that the subsequent clients requesting the same Web objects can be served directly from the proxy instead of other sources faraway, thus reducing the server's load, the network traffic and the client response time. However, with the dramatic increase of streaming media objects emerging on the Internet, the existing proxy cannot efficiently deliver them due to their large sizes and client real time requirements.

In this dissertation, we design, implement, and evaluate cost-effective and high performance proxy-based Internet caching systems for streaming media delivery. Addressing the conflicting performance objectives for streaming media delivery, we first propose an efficient segment-based streaming media proxy system model. This model has guided us to design a practical streaming proxy, called *Hyper-Proxy*, aiming at delivering the streaming media data to clients with minimum playback jitter and a small startup latency, while achieving high caching performance. Second, we have implemented Hyper-Proxy by leveraging the existing Internet infrastructure. Hyper-Proxy enables the streaming service on the common Web servers. The evaluation of Hyper-Proxy on the global Internet environment and the local network environment shows it can provide satisfying streaming performance to clients while maintaining a good cache performance. Finally, to further improve the streaming delivery efficiency, we propose a group of the *Shared Running Buffers (SRB)* based proxy caching techniques to effectively utilize proxy's memory. SRB algorithms can significantly reduce the media server/proxy's load and network traffic and relieve the bottlenecks of the disk bandwidth and the network bandwidth.

The contributions of this dissertation are threefold: (1) we have studied several critical performance trade-offs and provided insights into Internet media content caching and delivery. Our understanding further leads us to establish an effective streaming system optimization model; (2) we have designed and evaluated several efficient algorithms to support Internet streaming content delivery, including segment caching, segment prefetching, and memory locality exploitation for streaming; (3) having addressed several system challenges, we have successfully implemented a real streaming proxy system and deployed it in a large industrial enterprise.



## Building Internet Caching Systems for Streaming Media Delivery

# Chapter 1

## Introduction

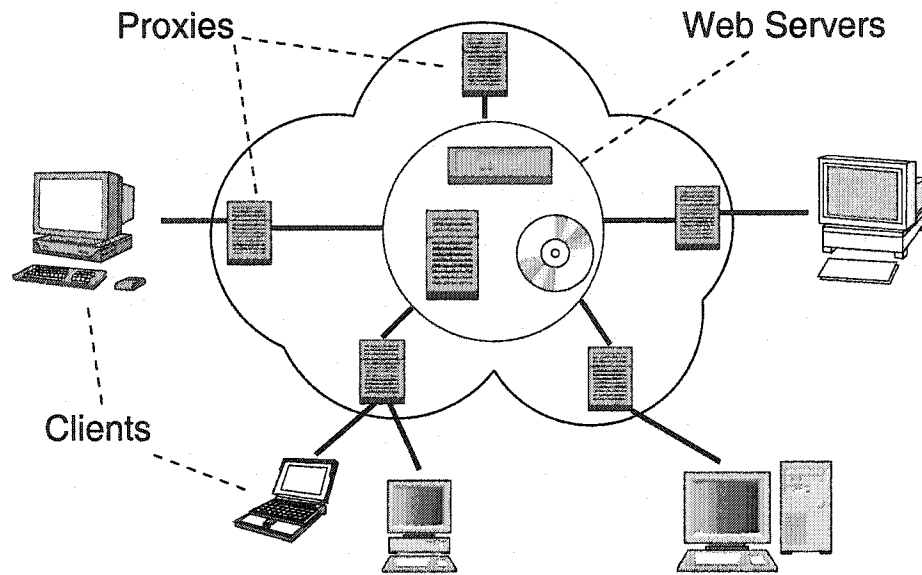
With the emergence of new Internet techniques, the Web contents on the Internet have increased substantially in two aspects: (1) the amount of the Web contents is increasing exponentially; (2) the Web contents are evolving from simple and static text-based pages to the more powerful dynamic contents and multimedia objects. Thus, the content delivery on the Internet becomes increasingly complex and needs effective system support from networking, systems and application software. This dissertation focuses on building cost-effective and high performance Internet caching systems for delivering streaming media contents.

### 1.1 Background

The wide deployment of the Web proxy mainly corresponded to the demand for efficient delivery of the increasing amount of the static Web pages<sup>1</sup>. The proxy can cache the static Web pages delivered by the content server upon a client request, so that the subsequent clients requesting the same pages can be directly served by the data stored in the proxy

---

<sup>1</sup>Proxy servers can also be used as firewalls too.



**Figure 1.1:** Internet access infrastructures: clients access Web servers through proxies

locally, instead of going to other sources faraway (the content server, other peer proxies, etc.), thus reducing the server's load, the network traffic, and the client perceived response time. To efficiently deliver the Web pages, normally the proxy is positioned close to the client<sup>2</sup>. Thus, the basic infrastructure for the content delivery has evolved from the end-to-end system to the server-proxy-client system, as shown on Figure 1.1. In this infrastructure, a client request is firstly received by the proxy, although the existence of proxy can be transparent to clients. If the proxy can serve the request from its local cache, the client can get the response instantly without contacting the content server. Otherwise, the request is forwarded to the content server. The response from the content server can be cached in the proxy to serve the subsequent requests for the same object. Therefore, the proxy caching strategies, other than the techniques for the server and the client, have been heavily studied, mainly on caching the static Web objects to reduce the network traffic and the end-

<sup>2</sup>The proxy positioned close the content server is called reverse proxy.

to-end latency. Proxies, such as Squid [1], Harvest [14] and CERN httpd [78], have been successfully used for caching static Web objects to reduce the Internet traffic and client perceived response time.

However, with more and more complicated contents, especially the streaming media contents, emerging on the Internet continuously, the proxy-based Internet content delivery faces new problem: the benefit of the existing proxy caching is significantly reduced because the proxy is incapable of cost-effectively handling the large amount of Internet streaming media contents.

Streaming media contents have already been widely used in many applications, such as education, entertainment, news, medical surgery cooperation, since its emergence on the Internet. The amount of the streaming media has increased rapidly and is still continue to increase: comparing the workloads of a 2002 study [99] and a 1999 study [116] in a similar campus environment, the portion of network bytes ascribed to audio and video increased by 300% and 400% [52], respectively. Being highly demanded by the society, the streaming media contents bring many new challenges to the existing proxy-based Internet content delivery networks due to the following two characteristics. (1) The size of a streaming media object is usually several orders of magnitudes larger compared to the text-based Web contents. For example, a MPEG2 video of two hours requires about 1 GB of disk space, while a general text-based Web page is in the range of 10 KB. Thus, to entirely cache several streaming media objects as caching the Web pages will quickly exhaust the cache space and result in poor cache efficiency. (2) The demand and requirement of continuous and timely delivery of the streaming media objects are more rigorous than that of the text-based Web pages. In receiving streaming media data, the client always expects free

of jitter playback and a small startup latency. However, existing proxies cannot satisfy these real-time requirements because they only provide best-effort services. Due to these problems, traditional proxies are not able to efficiently cache and deliver the streaming media contents. Currently, the Internet streaming media delivery still relies on the client-server model or the expensive commercial CDNs. In the client-server model, a lot of resources have to be reserved for streaming media delivery, in which even a relatively small number of client requests can overload a media server, causing bottlenecks by demanding high disk bandwidth on the server and high network bandwidth to the clients, or the client has to rely on the “progressive playback” where the quality of streaming is poor with frequently introduced playback jitter. In the extreme case, the client has to download the entire object before playback, where a large startup latency is expected.

## 1.2 Dissertation Contributions

The objectives of this dissertation research are to design, implement, and evaluate a cost-effective and high performance proxy-based Internet caching system for streaming media delivery. We first propose an efficient segment-based streaming media proxy system model. This model serves a foundation for us to further design a practical streaming proxy, called *Hyper-Proxy*, aiming at delivering the streaming media data to clients with minimum playback jitter and a small startup latency, while achieving high cache performance. Second, we implement Hyper-Proxy by leveraging the existing Internet infrastructure. Hyper-Proxy pushes the streaming functions from the server to the proxy, which makes the streaming traffic run on UDP [87] suffer from less data loss and no blocking. The evaluation of Hyper-

Proxy on the global Internet environment connecting Japan and US west coast, and the local network environment shows it can provide satisfying streaming performance to clients while maintaining good cache performance. Finally, to further improve the streaming delivery efficiency, we propose a group of the *Shared Running Buffers (SRB)* based proxy caching techniques to effectively utilize proxy's memory. SRB algorithms can significantly reduce the media server/proxy's load and network traffic and relieve the bottlenecks of the disk bandwidth and the network bandwidth.

The major contributions of this dissertation are summarized as follows.

1. Studying the three major performance objectives for streaming media delivery systems, namely, byte hit ratio in the proxy, startup latency perceived by a client, and client perceived playback jitter, we find that two pairs of conflicting performance objectives (byte hit ratio versus startup latency, and byte hit ratio versus playback jitter) exist in the current segment-based streaming proxy designs. Using heuristic and modeling approaches, we present effective solutions to address these two pairs of conflicting objectives. Comprehensively considering their trade-offs based on the best interests of clients, we propose an efficient segment-based streaming proxy design model, where a streaming proxy guarantees continuous streaming delivery subject to a small startup latency and high byte hit ratio. Guided by this model, we design a streaming proxy, called Hyper-Proxy. Through the evaluations of Hyper-Proxy on synthetic and real workloads, it shows that it can deliver the streaming media data to clients with minimum playback jitter and a small startup latency, while it also achieves good cache performance.

2. We have designed and evaluated several efficient algorithms to support Internet streaming content delivery, including segment caching, segment prefetching, and memory locality exploitation for streaming sessions. We designed a method, called *adaptive-lazy segmentation*, that delays the object segmentation as late as possible so that online client access patterns can be monitored for partitioning the object into smaller segments. We proposed window based prefetching and active prefetching methods to ensure that uncached segments of an object can always be prefetched in time to eliminate the client side playback jitter. The Shared Running Buffers (SRB) based media caching algorithms are designed to efficiently handle streaming media contents in the proxy's memory by exploiting the memory access locality. In SRB algorithms, subsequent client accesses to a same object are served through allocating a new dynamic running buffer and sharing the data in existing dynamic running buffers instantly. Patching SRB further expands the session sharing by utilizing the client side storage. These algorithms have been demonstrated to significantly reduce the media server's load and network traffic, relieve the bottlenecks of the disk bandwidth and the network bandwidth, and improve the client playback quality.
3. The implementation of Hyper-Proxy leverages the existing Internet infrastructure by talking to the content server via HTTP [46] while talking to the client via RTP [101]/RTSP [102]. Thus, it frees the media content server from streaming functions and pushes these functions to the proxy close to clients. Therefore, the traditional Web servers (e.g. Apache) now can provide real streaming service through Hyper-Proxy. In addition, the real streaming traffic on the UDP protocol also suffers from less data

loss and no blocking. We evaluate Hyper-Proxy on the global Internet environment connecting Japan and US west coast and the local network environments (LANs). The results show it can provide satisfactory streaming performance to clients in both environments while maintaining good cache performance.

The significance and potential impacts of this dissertation are as follows.

- Since available Internet streaming services are expensive, most Internet media users take the downloading approach that generates a lot of unnecessary traffic and a large startup latency, or the “progressive playback” approach that always results in frequent playback jitter. Our work on streaming proxy design and implementation will provide low-cost and high performance proxy-based streaming services, which will significantly improve the Internet resource utilization.
- Our optimization model provides quantitative guidances to design an effective proxy-based Internet caching system for streaming delivery, which can be extended for other types of proxy designs, such as caching systems for other sophisticated multimedia contents.
- The Hyper-Proxy has been successfully implemented and deployed in HP Company. We expect its lasting impact in Internet streaming community after its trial stage.

### 1.3 Dissertation Organization

The organization of this dissertation is as follows. The design of Hyper-Proxy is presented in Chapter 2. We describe the Hyper-Proxy implementation and evaluate its performance



in various environments in Chapter 3. SRB based algorithms are presented and evaluated in Chapter 4. We further introduce some related work in Chapter 5. In Chapter 6, we make concluding remarks and briefly present the future work.

## Chapter 2

# Segment-based High Quality Streaming Media Proxy Designs

### 2.1 Introduction

Proxy caching has been widely used to cache static (text/image) objects on the Internet so that subsequent requests to the same objects can be served directly from the proxy without contacting the server. However, the proliferation of Internet multimedia contents makes proxy caching challenging [13, 32, 68, 82, 91, 93, 104, 124]:

1. The size of media objects is usually several orders of magnitudes larger than traditional Web content. For example, a one-hour movie encoded using MPEG4, at desktop resolution, may require more than 1 GByte storage space. This limits the number of objects that can be completely stored in a caching proxy. It also results in large startup latencies if the object is not already cached.
2. Multimedia objects generally have very stringent demands in terms of continuous and timely delivery. This is especially challenging in the current Internet, which only

provides best-effort services.

3. Prior research has observed that most of the media objects are only partially viewed [30, 31]. Using traditional, static Web caching techniques to cache these large objects thus wastes storage and causes unnecessary network traffic.

To solve the problems caused by large-sized media objects, researchers have developed a number of segment-based proxy caching strategies [18, 23, 92, 94, 104, 117] that cache partial segments of media objects instead of their entirety. The existing segment-based proxy caching strategies can be classified into the following two types based on their performance objectives. The first type focuses on the reduction of the client perceived startup latency (denoted by the delayed startup ratio) by always giving a higher priority to caching the beginning segments of media objects based on the observation that clients tend to watch the beginning portions [30, 31]. For example, prefix caching [104, 111] always breaks a media object into a prefix segment and a suffix segment. The proxy caches the prefix segments only so that the cache can preserve prefix segments for more objects. The second type aims at reducing network traffic and the server workload by improving proxy caching efficiency, namely the byte hit ratio. For example, uniform segmentation strategy [94] considers caching of fixed-sized segments of layer-encoded video objects. The exponential segmentation strategy [18, 117] caches segments of media objects in a way that the succeeding segment doubles the size of its preceding one. The most recently proposed adaptive-lazy segmentation strategy [23] can achieve the highest byte hit ratio by delaying the object segmentation as late as possible till some real time access information is collected for this object so that the most popular part of this object can be identified. These proxy-based

caching strategies emphasize on improving the proxy caching efficiency.

However, these segment-based proxy caching strategies cannot automatically ensure continuous streaming delivery to the client. In a segment-based proxy caching system, since only partial segments of objects are cached in the proxy, it is important for the proxy to fetch and relay the uncached segments to the client in time whenever necessary. A delayed fetch of the uncached segments, which we call *proxy jitter*, causes the discontinuous delivery of media content. Proxy jitter aggregates onto the playback jitter at the client side. Once a playback starts, jitter is not only annoying but can also potentially drive the user away from accessing the content. Thus, for the best interests of clients, the highest priority must be given to minimize proxy jitter, and a correct model for media proxy cache design should aim to minimize proxy jitter subject to reducing the delayed startup ratio and increasing the byte hit ratio.

To reduce proxy jitter, one key is to develop prefetching schemes that can timely prefetch uncached segments. Some early work has studied the prefetching of multimedia objects [67, 69, 92, 94]. For layer-encoded objects [92, 94], the prefetching of uncached layered video is done by always maintaining a prefetching window of the cached stream, and identifying and prefetching all the missing data within the prefetching window with a fixed time period (length of  $T$ ) ahead of their playback time. In [67], the proactive prefetch utilizes any partially fetched data due to the connection abortion to improve the network bandwidth utilization. In [69], prefetching is used to prefetch a certain amount of data so that caching is feasible. Unfortunately, little prefetching work has been found to efficiently solve the proxy jitter problem in the context of segment-based streaming proxy caching.

Improving the byte hit ratio increases proxy caching efficiency, while reducing proxy jitter provides clients with a continuous streaming service. Unfortunately, these two objectives conflict with each other. Furthermore, we have also observed that improving the byte hit ratio conflicts with reducing the delayed startup ratio [23]. These three conflicting objectives form two pairs of trade-offs that complicate the design model. No previous work has been found to address the balancing of these trade-offs, which are uniquely important to streaming media proxy systems.

In this chapter, we first propose a look-ahead window based prefetching method and an active prefetching method for the in-time prefetching of uncached segments, which not only gives an effective solution to address the proxy jitter problem, but also provides insights into the trade-off between improving the byte hit ratio and reducing proxy jitter. Second, to effectively addresses the conflicting interests between reducing startup latency and improving byte hit ratio, we build a general model to analyze the performance trade-off between the second pair of conflicting performance objectives and provide an effective approach to balance them. Finally, considering our main objective of minimizing proxy jitter and balancing the two trade-offs, we propose a new streaming proxy system called *Hyper-Proxy* [25] by effectively coordinating both prefetching and segmentation techniques. Hyper-Proxy depends on the HTTP channel for prefetching, while it interfaces with clients in a RTP [101]/RTSP [102] streaming channel. Synthetic and real workloads are used to systematically evaluate the system. The performance results show that the Hyper-Proxy system generates minimum proxy jitter with a low delayed startup ratio and a small decrease of byte hit ratio compared to existing schemes. Our study also indicates that the standard objective of improving the byte hit ratio commonly used in proxy caching for Web

objects is not suitable to streaming media delivery.

The rest of this chapter is organized as follows. Some related work is introduced in Section 2.2. We propose prefetching methods and provide insights into proxy jitter in Section 2.3. The second pair of conflicting interests is addressed in Section 2.4. The Hyper Proxy system is presented in Section 2.5. We evaluate it in Section 2.6 and we make chapter summary in Section 2.7.

## 2.2 Related Work

The proxy caching strategies have been mainly studied in the context of static Web contents and have therefore been the focus of many studies, mainly on caching the static Web pages [11, 12, 16, 39, 41, 44, 49, 54, 72, 73, 75, 113, 114] and dynamic contents [15, 17, 19, 40, 59, 63, 77, 89, 105, 122, 125] to reduce the network traffic and the end-to-end latency. Recently, many characterizations of media workloads [6, 9, 33, 42, 52, 56, 57, 80, 83, 84, 109, 112] have been performed and a lot of streaming media caching systems have been studied in [18, 23, 30, 31, 32, 51, 104, 117]. Some researchers [30, 31] have observed that most of the clients intend to watch the initial parts of the media objects and there are less and less accesses on the later portions. Based on this observation, the segment-based proxy caching strategies are proposed. The segment-based caching strategies cache media objects in segments instead of in full to reduce the user perceived startup latency and to reduce the network traffic to media servers and the disk bandwidth requirement to the media server. Among them, the prefix caching [104] was proposed earlier to segment the media object as prefix segment and suffix segment. Its protocol consideration as well as partial sequence

caching are studied in [32, 51]. More recently, two types of new segmentation strategies had been developed according to how the object is divided. The first is to use uniform sized segments. For example, Rejaie et al [94] considers the caching of fixed sized segments of layer-encoded video objects. In our adaptive-lazy segmentation strategies proposed in [23], each object has itself segmented as late as possible and has a uniform segment length determined according to the client access pattern. The uniform is only to each object while different objects may have different segment lengths. The second is to use exponential sized segments. In this strategy, media objects are segmented in a way that the size of a segment doubles that of its preceding one [117]. The intuition of this strategy is based on the assumption that later segments of media objects are less likely to be accessed. A combining use of these methods can be found in [18], in which the simple constant length and the exponentially increased length are both considered in RCache and Silo.

In a finer granularity for Internet video delivery, some strategies are also proposed based on the video frames. In video staging [124], a portion of bits from the video frames whose size is larger than a predetermined threshold is cut off and prefetched to the proxy to reduce the bandwidth on the server proxy channel. In [92, 93, 94], a similar idea is proposed for caching scalable videos that co-operates with the congestion control mechanism. In [79], the proposed approach attempts to select groups of consecutive frames by the selective caching algorithm, while in [82], the algorithm may select groups of non-consecutive frames for caching in the proxy. The caching problem for layered encoded video is studied in [68]. The context-aware segmentation is studied in [74]. The cache replacement of streaming media is studied in [91, 106].

P2P assisted media streaming has also been studied [35, 58, 64, 85, 96, 108]. In [85, 108],

multicast trees are studied for live media streaming, while P2P streaming schemes for layer-encoded media are proposed in [35, 96]. Work in [58, 64] proposes a Guntella-like unstructured P2P media streaming system and a structured P2P media streaming system.

## 2.3 Prefetching Methods and Insights into Proxy Jitter

Prefetching schemes can reduce proxy jitter by fetching uncached segments before they are accessed. However, an efficient prefetching method should consider the following two conflicting interests in the proxy. On one hand, proxy jitter occurs if the prefetching of uncached segments is delayed. To avoid jitter, the proxy should prefetch uncached segments as early as possible. On the other hand, aggressive prefetching of uncached segments requires extra network traffic and storage space to temporarily store the prefetched data. Even worse, the client session may terminate before the prefetched segments are accessed. This observation indicates that the proxy should prefetch uncached segments as late as possible. This contradiction requires that the proxy accurately decides when to prefetch which uncached segment in a way to minimize the proxy jitter as well as to minimize the resource usage (network and storage). In this section, we propose a *look-ahead window based prefetching* method and an *active prefetching* method, which jointly consider both objectives. Our subsequent analysis further provides the insights into the conflicting interests between reducing proxy jitter and improving the byte hit ratio.

In our analysis, the following assumptions are made.

- The object has been segmented and is accessed sequentially;
- The bandwidth of the proxy-client link is large enough for the proxy to stream the



content to the client smoothly; and

- Each segment of the object can be fetched from the server (either the origin server or a cooperative one) in a unicast channel.

Since the prefetching is segment based, several related notations used in the analysis are listed in Table 2.1. Note that each media object has its inherent encoding rate, which is the

**Table 2.1:** The notations for prefetching study

$B_s$	the average encoding rate of a certain object segment
$B_t$	the average network bandwidth of the proxy-server link
$k$	the total number of segments of the object
$n$	the number of cached segments of the object
$S_i$	the $i^{th}$ segment of the object
$L_i$	the length of the $i^{th}$ segment
$L_b$	the base segment length of the object, $L_b = L_1$

playback rate. The rate is not a constant in variable bit rate video, but we use  $B_s$  to denote its average value.  $B_t$  may vary dynamically when different segments are accessed. The proxy monitors  $B_t$  by keeping records of the data transmission rate of the most recent prior session with the same server. The transmission rate is calculated by dividing the amount of transferred data by the data transmission duration.

For a requested media object, assume there are  $n$  segments cached in the proxy. The goal is to determine when to schedule the prefetching of the uncached  $(n + 1)^{th}$  segment,  $S_{n+1}$ , so that proxy jitter is avoided. We denote the scheduling point as  $x$ .

Note that prefetching is not necessary when  $B_s \leq B_t$ , so the following discussion is based on when  $B_s > B_t$ .

### 2.3.1 Look-ahead Window Based Prefetching Method

The major action of the look-ahead window based prefetching is to prefetch the succeeding segment if it is not cached when the client starts to access the current one. The window size is thus fixed for the uniformly segmented object and is exponentially increasing for the exponentially segmented object.

When  $B_s > B_t$ , assume the prefetching of the next uncached segment  $S_{n+1}$  starts when the client starts to access the position  $x$  in the current segment  $S_n$ . Thus,  $x$  is the position that determines the starting time of prefetching, called the *prefetching scheduling point*. To denote  $y$  as  $y = L_n - x$  and to guarantee the in-time prefetching of the next uncached segment, we have

$$\frac{y + L_{n+1}}{B_s} \geq \frac{L_{n+1}}{B_t}, \quad (2.1)$$

which means

$$y \geq \frac{L_{n+1} \times (S - T)}{T}. \quad (2.2)$$

Since  $y = L_n - x$ , thus

$$x \leq L_n - \frac{L_{n+1} \times (B_s - B_t)}{B_t}. \quad (2.3)$$

We can calculate the prefetching starting point as the percentage of the current segment by dividing  $x$  by  $L_n$ , which leads to

$$\frac{x}{L_n} \leq 1 - \frac{L_{n+1}}{L_n} \times \left( \frac{B_s}{B_t} - 1 \right). \quad (2.4)$$

Equation (2.4) means to prefetch the next uncached segment when the client has accessed the  $1 - \frac{L_{n+1}}{L_n} \times \left( \frac{B_s}{B_t} - 1 \right)$  portion of current segment. Accordingly, the size of the minimum

buffer size is  $\frac{y}{B_s} \times B_t$ , which is  $L_{n+1} \times (1 - \frac{B_t}{B_s})$ . Once we know the minimum buffer size, we know that in the worst case, the fully buffered prefetched data may not be used by the client, which means the maximum amount of wasted prefetched data,  $W$ , has the same size as the buffer. Thus, we always give the minimum buffer size by the following analysis.

For objects segmented uniformly (e.g. by uniform segmentation or adaptive-lazy segmentation) or exponentially (such as by exponential segmentation), the situations are as follows:

- For the uniformly segmented object, by Equation (2.3), we have  $\frac{x}{L_n} \leq 2 - \frac{B_s}{B_t}$ . It implies that  $B_s$  could not be 2 times larger than  $B_t$ . The minimum size is  $L_1 \times (1 - \frac{B_t}{B_s})$ . The prefetching of the next uncached segment starts when the client has accessed to the  $2 - \frac{B_s}{B_t}$  portion of the current segment.
- For the exponentially segmented object, by Equation (2.3), we have  $\frac{x}{L_n} \leq 3 - 2 \times \frac{B_s}{B_t}$ . It implies that  $B_s$  could not be 1.5 times larger than  $B_t$ . The minimum size is  $L_{n+1} \times (1 - \frac{B_t}{B_s})$ , which increases exponentially. The prefetching of the next uncached segment starts when the client has accessed the  $3 - 2 \times \frac{B_s}{B_t}$  portion of the current segment.

Above analysis shows that this look-ahead window based prefetching method does not work when  $B_s > 1.5B_t$  for the exponential segmentation strategy, and it does not work when  $B_s > 2B_t$  for the uniform segmentation strategy.

In addition, since  $B_s > B_t$ , we have

$$\begin{aligned}
 B_s > B_t &\Rightarrow \frac{B_s}{B_t} > 1 \\
 &\Rightarrow 2 \times \frac{B_s}{B_t} - \frac{B_s}{B_t} > 1 \\
 &\Rightarrow -\frac{B_s}{B_t} > 1 - 2 \times \frac{B_s}{B_t} \\
 &\Rightarrow 2 - \frac{B_s}{B_t} > 3 - 2 \times \frac{B_s}{B_t}. \tag{2.5}
 \end{aligned}$$

The left side of Equation (2.5) represents the prefetching scheduling point for the uniform segmentation strategy, while the right side denotes that for the exponential segmentation strategy. Thus, Equation (2.5) states that the prefetching of the next uncached segment for the exponential segmentation strategy is always earlier than that for the uniform segmentation strategy, causing a higher possibility of wasted resources.

Since the condition of  $B_s > B_t$  is quite common in practice, the look-ahead window based prefetching method has a limited prefetching capability in reducing the proxy jitter. Next, we will address its limit by an active prefetching method.

### 2.3.2 Active Prefetching

If the prefetching is conducted more aggressively, we are able to further reduce proxy jitter, and of course, which will also consume more resources. The basic idea of our second method, active prefetching, is to preload uncached segments as early as when the client starts to access a media object.

We re-define the prefetching scheduling point,  $x$ , as the position in the first  $n$  cached segments (instead of a position in the  $n^{th}$  segment for the look-ahead window based prefetching

method) that is accessed by a client. As soon as this prefetching scheduling point is accessed, the prefetching of  $n + 1^{th}$  segment must start in order to avoid the proxy jitter. Again, the objective of our prefetching is to determine when to prefetch which uncached segment so that proxy jitter is minimized with the minimum amount of resource requirement.

At position  $x$ , the length of the to-be-delivered data from the cache is  $\sum_{i=1}^{i=n} L_i - x$ . To avoid proxy jitter, the time that the proxy takes to prefetch  $S_{n+1}$  must not exceed the time that the proxy takes to deliver the rest of the cached data and the fetched data. That is, the following condition must be satisfied to avoid proxy jitter:

$$\frac{\sum_{i=1}^{i=n} L_i - x + L_{n+1}}{B_s} \geq \frac{L_{n+1}}{B_t}.$$

Therefore, the latest prefetching scheduling point to avoid proxy jitter is:

$$x = \sum_{i=1}^{i=n} L_i - \frac{L_{n+1} \times (B_s - B_t)}{B_t}. \quad (2.6)$$

Refer back to our objectives, when  $x$  is selected as the prefetching scheduling point, the buffer size required for the prefetched data reaches the minimum:

$$\frac{\sum_{i=1}^{i=n} L_i - x}{B_s} \times B_t. \quad (2.7)$$

We now discuss the active prefetching method for objects segmented differently by first determining the prefetching scheduling point and then discussing the prefetching scheme and resource requirements.

- For the uniformly segmented object,  $L_i = L_1$ , based on Equation 2.6, we have the

latest scheduling point  $x$  as

$$x = (n + 1)L_1 - \frac{B_s}{B_t}L_1. \quad (2.8)$$

Equation 2.8 states that if  $n + 1 \geq \frac{B_s}{B_t}$ , in-time prefetching of  $S_{n+1}$  is possible with the minimum required buffer size of

$$L_1 \times \frac{B_s - B_t}{B_s}. \quad (2.9)$$

However, Equation 2.8 also indicates that if  $n + 1 < \frac{B_s}{B_t}$ , in-time prefetching of  $S_{n+1}$  is not possible! Therefore, when  $n + 1 < \frac{B_s}{B_t}$  and the segments between  $n + 1^{th}$  and  $\lceil \frac{B_s}{B_t} \rceil^{th}$  are demanded, proxy jitter is inevitable. To minimize future proxy jitter under this situation, the proxy needs to prefetch the  $\lceil \frac{B_s}{B_t} \rceil^{th}$  segment instead of the  $n + 1^{th}$  segment.

For uniformly segmented objects, active prefetching works as follows:

- $n = 0$ : No segment is cached. Proxy jitter (in this case, startup latency) is inevitable. To avoid future proxy jitter, prefetching of the  $\lceil \frac{B_s}{B_t} \rceil^{th}$  segment is necessary. The minimum buffer size required is  $(1 - \frac{B_t}{B_s})L_1$ .
- $n > 0$  and  $n + 1 < \frac{B_s}{B_t}$ : The proxy starts to prefetch the  $\lceil \frac{B_s}{B_t} \rceil^{th}$  segment once the client starts to access the object. If the segments between  $n + 1^{th}$  and  $\lceil \frac{B_s}{B_t} \rceil^{th}$  are demanded, they are fetched on demand, and proxy jitter is inevitable. The minimum buffer size required is  $(1 - \frac{B_t}{B_s})L_1$ .
- $n > 0$  and  $n + 1 \geq \frac{B_s}{B_t}$ : The prefetching of  $S_{n+1}$  is scheduled when the streaming reaches the position of  $(n + 1 - \frac{B_s}{B_t})L_1$  of the first  $n$  cached segments. Proxy jitter

can be completely eliminated in this case, and the minimum buffer size required is  $(1 - \frac{B_t}{B_s})L_1$ .

- For the exponentially segmented object, active prefetching works as follows. Here, we assume  $B_s \leq 2B_t$ . When  $B_s \geq 2B_t$ , no prefetching of the uncached segments can be in time for the exponentially segmented objects.

- $n = 0$ : No segment is cached. Proxy jitter (in this case, startup latency) is inevitable. To avoid future proxy jitter, the prefetching of the  $\lceil 1 + \log_2(\frac{1}{2 - \frac{B_s}{B_t}}) \rceil^{th}$  segment is necessary once the client starts to access the object. The minimum buffer size required is  $L_1 \times \frac{B_t^2}{2 \times B_s \times B_t - B_s^2}$ .
- $n > 0$  and  $n \leq \log_2(\frac{1}{2 - \frac{B_s}{B_t}})$ : The proxy starts to prefetch the  $\lceil 1 + \log_2(\frac{1}{2 - \frac{B_s}{B_t}}) \rceil^{th}$  segment once the client starts to access this object. Proxy jitter is inevitable when the client accesses data of the  $n + 1^{th}$  segment to the  $\lceil 1 + \log_2(\frac{1}{2 - \frac{B_s}{B_t}}) \rceil^{th}$  segment. The minimum buffer size is  $L_i B_t / B_s$ , where  $i = \lceil 1 + \log_2(\frac{1}{2 - \frac{B_s}{B_t}}) \rceil$ .
- $n > 0$  and  $n > \log_2(\frac{1}{2 - \frac{B_s}{B_t}})$ : The prefetching of the  $n + 1^{th}$  segment starts when the client accesses to the  $1 - \frac{2^n}{2^{n+1} - 1} \times (\frac{B_s}{B_t} - 1)$  portion of the first  $n$  cached segment. The minimum buffer size is  $L_{n+1} \times \frac{B_t}{B_s}$  and increases exponentially for later segments.

Our proposed active prefetching method gives the optimal prefetching scheduling point whenever possible with minimum resource usage. However, under certain conditions the prefetching of uncached segments may still be delayed as our analysis showed, for both uniformly and exponentially segmented objects. Furthermore, the analysis also finds that the uniformly segmented object has advantages over the exponentially

segmented object: it offers enhanced capability for in-time prefetching and the in-time prefetching can always begin in a later stage.

### 2.3.3 Segment-based Proxy Caching and Proxy Jitter Free Strategies

The previous section shows that active prefetching cannot always guarantee continuous media delivery, which is one of the most important objectives for the streaming delivery. However, for any caching strategy, if there are always enough number of segments being cached in the proxy, prefetching of the uncached segments can always be in time. To evaluate this situation, we define *free-of-jitter length* as follows:

- *free-of-jitter length*: the minimum length of data that must be cached in the proxy in order to guarantee the continuous delivery when  $B_s > B_t$ . We denote  $m$  as the number of segments with the aggregated length equal to the free-of-jitter length, called *free-of-jitter segments*.

In-time prefetching must guarantee, in the worst case, that the prefetching of the rest of segments is completed before the delivery of the whole object, that is:

$$\frac{\sum_{i=1}^{i=k} L_i}{B_s} \geq \frac{\sum_{i=1}^{i=k} L_i - \sum_{i=1}^{i=m} L_i}{B_t}. \quad (2.10)$$

This indicates that the following condition must be satisfied to guarantee in-time prefetching:

$$\sum_{i=1}^{i=m} L_i \geq \sum_{i=1}^{i=k} L_i \left(1 - \frac{B_t}{B_s}\right). \quad (2.11)$$



- For the uniformly segmented object, since  $L_b$  is the base segment length, the minimum  $m$  to satisfy the above condition is:

$$m = \lceil \frac{(1 - \frac{B_t}{B_s}) \sum_{i=1}^{i=k} L_i}{L_b} \rceil. \quad (2.12)$$

- For the exponentially segmented objects, since  $L_i = 2L_{i-1}$ , the minimum  $m$  to satisfy the condition is:

$$m = \lceil \log_2(\frac{(1 - \frac{B_t}{B_s}) \sum_{i=1}^{i=k} L_i}{L_b}) \rceil + 1. \quad (2.13)$$

#### 2.3.4 Trade-off Between Low Proxy Jitter and High Byte Hit Ratio

We have calculated the minimum number of segments that must always be cached in the proxy to guarantee a continuous delivery of the streaming media object. Thus we can estimate how much cache space we need to guarantee a proxy-jitter-free delivery. However, in practice, we always have limited cache space and cannot cache all these segments for each object.

In an actual segment-based proxy caching system, popular objects are always cached to reduce network traffic and server load. If an object is popular enough, all its segments can be cached in the proxy, possibly larger than its free-of-jitter length. If an object is not popular enough, some segments may get evicted and only a few of its segments are cached. The aggregated length of these segments may be less than its free-of-jitter length, which causes proxy jitter when the uncached segments are demanded by the client. Given a higher priority in reducing the proxy jitter, the proxy can choose to evict segments of the object whose cached data length is larger than its free-of-jitter length. The released cache space can be used to cache more segments of the object whose cached data length is less than its

free-of-jitter length so that the prefetching of its uncached segment can always be in time. It is possible that segments of popular objects are evicted, which may reduce the byte hit ratio. However, since there are more objects with enough segments cached to avoid delayed prefetching, overall proxy jitter is reduced. From this transition, we can see that the byte hit ratio can be traded for less proxy jitter.

The insights of the conflicting interests between improving the byte hit ratio and reducing proxy jitter have motivated us to revise the principle to design a highly effective proxy caching system, aiming to minimize the proxy jitter.

## 2.4 Byte Hit Ratio vs. Delayed Startup Ratio

From the previous study [23], we have observed that segment-based proxy caching strategies, typically the adaptive-lazy segmentation and exponential segmentation, always perform well in the byte hit ratio, but perform not so well in the delayed startup ratio, or vice versa. This observation leads us to conjecture that there are some conflicting interests between the objectives of improving the byte hit ratio and reducing the delayed startup ratio. We must understand these insights before we can design a correct system according to our design model.

In this section, we formalize the problem and mathematically analyze this trade-off. An analytical model is built to analyze these two representatives: exponential segmentation and adaptive-lazy segmentation for the ideal situation where the objects are cached in the proxy in the order of their popularities. Thus the effect of other factors can be excluded so that we can understand the performance insights. Firstly, we briefly review the two

representative strategies we will examine with the following notations.

- (1)  $T_1$ : the time instance the object is accessed for the first time;
- (2)  $T_r$ : the last reference time of the object. It is equal to  $T_1$  when the object is accessed for the first time;
- (3)  $T_c$ : the current time instance ;

### 2.4.1 Exponential Segmentation Strategy

The exponential segmentation strategy segments each media object exponentially. It then admits the segments of the object according to their relative positions in the object and their caching utilities by the admission policy. The segment replacement uses the LRU policy for the replacement of the beginning segments and always replaces the segment with the least caching utility for the later segments, respectively. More details can be found in [117].

#### 2.4.1.1 Segmentation Method

A media object is divided into multiple equal-sized blocks. Multiple blocks are then grouped into a segment by the proxy. The size of a segment is sensitive to its distance from the beginning of the media object. The number of blocks grouped in segment  $i$  is  $2^{i-1}$ . In general, segment  $i$  is twice as large as segment  $i - 1$ . The purpose of this method is to allow the proxy to quickly discard a big chunk of cached media objects.

### 2.4.1.2 Admission Policy

A two-tiered approach is used for admission control. For a segment with a segment number smaller than a threshold,  $K_{min}$ , it is always eligible for caching. However, for a segment with a segment number equal to or larger than  $K_{min}$ , it is determined to be eligible for caching only if its caching utility is larger than some cached segments also with segment number equal to or larger than  $K_{min}$ . For this purpose, a portion of the cache space is reserved to store the beginning segments only while the remaining of the cache space is used to store the later segments. With such a cache admission control, at least the first  $K_{min}$  segments are stored for any cached objects by reserving a cache portion large enough for the beginning segments.

### 2.4.1.3 Replacement Policy

The caching utility of a segment depends on the reference frequency of an object and the segment distance. It is defined to be the ratio of reference frequency over the segment distance. The reference frequency is estimated as  $\frac{1}{T_c - T_r}$ . As a result, the caching utility of segment  $i$  of an object is defined as  $\frac{1}{(T_c - T_r) \times i}$ . According to the caching utility of the segment, two LRU stacks are maintained for the first  $K_{min}$  segments and the later segments. When an object is requested for the first time, the first  $K_{min}$  segments are always eligible for caching as a unit and the LRU scheme is used to find the replacement, while the later segments are always not cached for the first time. A later segment is eligible to cache only if its caching utility is greater than that of its replacement segment.

### 2.4.2 Adaptive-Lazy Segmentation Strategy

We proposed adaptive-lazy segmentation for streaming media object caching in the proxy by always caching the popular segments depending on the real time client access patterns [23]. In this strategy, each object is fully cached by the aggressive admission policy when it is accessed for the first time. The fully cached object is kept in the cache until it is chosen as an eviction victim by the replacement policy. At which time, the object is segmented using the lazy segmentation method and some segments are evicted by the replacement policy. From then on, the segments of the object are adaptively admitted or adaptively replaced segment by segment. The following additional notations are needed to define this strategy.

- (4)  $L_{sum}$ : the sum of the duration of each access to the object;
- (5)  $n_a$ : the number of accesses to the object;
- (6)  $L_b$ : the length of the base segment;
- (7)  $n_s$ : the number of the cached segments of the object.

Thus, at time instance  $T_c$ , the access frequency  $F$  is denoted as  $\frac{n_a}{T_r - T_1}$ , and the average access duration  $L_{avg}$  is denoted as  $\frac{L_{sum}}{n_a}$ .

#### 2.4.2.1 Aggressive Admission Policy

For any media object, the cache admission is considered aggressively in one of the following procedures whenever the object is accessed. (1) If the object is accessed for the first time, the whole object is subsequently cached regardless of the request's accessing duration. The cache space is allocated through the replacement policy if there is no sufficient space. (2) If the object has been accessed and is fully cached, no cache admission is necessary. (3) If the object has been accessed but it is not fully cached, the proxy aggressively considers to

cache the  $(n_s + 1)$ th segment if  $L_{avg} \geq \frac{1}{2} \times (n_s + 1) \times L_b$ . The inequality indicates that the average access duration is increasing to the extent that the cached  $n_s$  segments can not cover most of the requests while a total of  $n_s + 1$  segments can for a normal distribution. Therefore, the proxy should consider the admission of the next uncached segment.

#### 2.4.2.2 Lazy Segmentation Method

The basic idea of the lazy segmentation method is as follows. If the victim object chosen for replacement turns out to be fully cached, the proxy segments the object in the following way. The average access duration  $L_{avg}$  at that time instance is calculated. It is used as the length of the base segment of this object, that is,  $L_b = L_{avg}$ . Note that the value of  $L_b$  is fixed once it is determined. The object is then segmented uniformly according to  $L_b$ . After that, the first 2 segments are kept in cache, while the rest is evicted by the replacement policy.

#### 2.4.2.3 Two-Phase Iterative Replacement Policy

By defining the utility function as  $\frac{L_{avg}}{T_r - T_l} \times \min\{1, \frac{\frac{T_r - T_l}{n_s}}{T_c - T_r}\}$ , the two-phase iterative replacement policy works as follows. Upon an object admission, if there is not enough cache space, the proxy chooses the object with the smallest utility value at that time as the victim, and the segment of this object is evicted in one of the two phases as follows. (1) First Phase: If the object is fully cached, the object is segmented by the lazy segmentation method. The first 2 segments are kept and the remaining segments are evicted right after the segmentation is completed. Therefore, the portion of the object left in cache is of length  $2 \times L_b$ . Given that  $L_b = L_{avg}$  at this time instance, the cached 2 segments cover a normal distribution in the

access duration. (2) Second Phase: If the object is partially cached, the last cached segment of this object is evicted. The utility value of the object is updated after each replacement and this process repeats iteratively until the required space is found.

### 2.4.3 Analytical Model

Having reviewed the two representative strategies, now we formalize the problem and build a general analytical model for the adaptive-lazy segmentation and exponential segmentation strategies. We assume:

- (1) The popularity of the objects follows a Zipf-like distribution [4, 38, 62], which models the probability set  $p_i$ , where  $p_i = \frac{f_i}{\sum_{i=1}^N f_i}$ , ( $i=1, 2, \dots, N$ ,  $N$  is the total number of objects) and  $f_i = \frac{1}{i^\theta}$ , where  $\theta > 0$  and is the skew factor;
- (2) The request arrival interval process follows Poisson distribution with a mean arrival rate  $\lambda$ . The request arrival interval process to each individual object is independently sampled from the aggregate arrival interval process based on probability set  $p_i$ , where  $\sum_{i=1}^N p_i = 1$ ;
- (3) The clients view the requested objects completely. This is to simplify the analysis and does not affect the conclusion.

These assumptions indicate that the mean arrival rate for each object is:

$$\lambda_i = \lambda p_i = \lambda \times \frac{\frac{1}{i^\theta}}{\sum_{i=1}^N \frac{1}{i^\theta}}.$$

To evaluate the delayed startup ratio, we define the following notation:

- *startup length*: the length of the beginning part of an object. If this portion of the object is cached, no startup delay is perceived by clients when the object is accessed.

We use  $\alpha$  to denote the percentage of the startup length with respect to the full object length<sup>1</sup>.

Other notations used in the discussion are listed below:

- $L_{obj}^i$ : the full length of the  $i^{th}$  object, where  $1 \leq i \leq N$ ;
- $L_{obj}^{ave}$ : the average length of the objects;
- $C$ : the total cache size;
- $\beta$ : the percentage of total cache space reserved for the caching of first  $\alpha$  percent (startup length) of objects;
- $C_{prefix}$ : the size of reserved cache space for caching startup length of objects.  $C_{prefix} = \beta C$ ;
- $C_{rest}$ : the size of the cache space other than the space reserved for caching of startup length of objects.  $C_{rest} = C - C_{prefix} = (1 - \beta)C$ .

We consider the ideal case where the cache space is always allocated to cache the most popular objects. If we sort the objects according to their decreasing popularities, the ideal case indicates that  $C_{prefix}$  is used to cache the segments (within startup length) of the first  $t$  most popular objects.

Thus, ideally, for exponential segmentation, assuming the  $C_{prefix}$  can cache the first  $t$  objects' prefix segments,  $t$  must satisfy the following condition:

$$\sum_{i=1}^{i=t} L_i \times \alpha \leq C_{prefix} \quad \text{and} \quad \sum_{i=1}^{i=t+1} L_i \times \alpha > C_{prefix}. \quad (2.14)$$

---

<sup>1</sup>Note that instead of caching the first  $\alpha$  percent, caching a constant length of the prefix segment for each object leads to the same results.



Ideally, assuming the rest of the cache size  $C_{rest}$  can cache the first  $m$  object's remaining segments,  $m$  must satisfy the following condition:

$$\sum_{i=1}^{i=m} L_i \times (1 - \alpha) \leq C_{rest} \quad \text{and} \quad \sum_{i=1}^{i=m+1} L_i \times (1 - \alpha) > C_{rest}. \quad (2.15)$$

For adaptive-lazy segmentation, no cache space is allocated separately to cache the initial segments of the object, thus, ideally, assuming the whole cache can be used to cache the first  $k$  objects according to the popularity,  $k$  must satisfy the following condition:

$$\sum_{i=1}^{i=k} L_i \leq C \quad \text{and} \quad \sum_{i=1}^{i=k+1} L_i > C. \quad (2.16)$$

To this end, we express the delayed start request ratios for exponential segmentation and adaptive-lazy segmentation as follows:

$$P_{delay-E} = \frac{\sum_{i=t+1}^{i=n} \lambda_i}{\sum_{i=1}^{i=n} \lambda_i} \quad (2.17)$$

and

$$P_{delay-L} = \frac{\sum_{i=k+1}^{i=n} \lambda_i}{\sum_{i=1}^{i=n} \lambda_i}, \quad (2.18)$$

respectively.

Without considering the misses when the object is accessed for the first time, their corresponding byte hit ratios are:

$$P_{hit-E} = 1 - \frac{\sum_{i=t+1}^{i=n} \lambda_i \times L_i \times \alpha + \sum_{i=m+1}^{i=n} \lambda_i \times L_i \times (1 - \alpha)}{\sum_{i=1}^{i=n} \lambda_i \times L_i} \quad (2.19)$$

and

$$P_{hit-L} = 1 - \frac{\sum_{i=k+1}^{i=n} \lambda_i \times L_i}{\sum_{i=1}^{i=n} \lambda_i \times L_i}. \quad (2.20)$$

respectively.

### 2.4.4 Performance Objective Analysis

In order to find the reasons for the unbalanced performance results as we observed, we analyze the performance objectives one by one based on the model we have built. In this section, lazy segmentation is always used to mean the adaptive-lazy segmentation strategy.

#### 2.4.4.1 Delayed Start Request Ratio

Equation 2.17 and Equation 2.18 indicate that the relationship between  $t$  and  $k$  determines which technique performs better in terms of the delayed start request ratio.

Based on Equations 2.14 and 2.16, by comparing

$$\sum_{i=1}^{i=t} L_i \leq \frac{C \times \beta}{\alpha} \quad \text{and} \quad \sum_{i=1}^{i=k} L_i \leq C,$$

we can get that if  $\frac{\beta}{\alpha} > 1$ , it will lead to  $t > k$ . Through Equation 2.17 and Equation 2.18,  $t > k$  means that the exponential segmentation has a better (less) delayed start request ratio. Otherwise,  $t \leq k$ , and lazy segmentation will perform better.

Exponential segmentation always caches beginning segments of all objects, which leads to  $k < t$ . Thus, in terms of the delayed start request ratio, exponential segmentation normally performs better than lazy segmentation.

#### 2.4.4.2 Byte Hit Ratio

Based on Equation 2.19 and Equation 2.20, we can see the relationships among  $t$ ,  $m$ , and  $k$  are deterministic to the byte hit ratio.

We now provide a thorough evaluation of all possible situations as follows.

- $m = t$  :

Equation 2.19 can be written as:

$$P_{hit-E} = 1 - \frac{\sum_{i=t+1}^{i=n} \lambda_i \times L_i}{\sum_{i=1}^{i=n} \lambda_i \times L_i}.$$

Compared with Equation 2.20, the problem once again comes to the relationship of  $t$  and  $k$ . If  $t > k$ , then exponential segmentation performs better. If  $t < k$ , lazy segmentation performs better.

When  $m = t$ , there are  $m$  or  $t$  objects cached by the exponential segmentation totally. Thus, we can get  $t = k$ . Thus, lazy segmentation will perform the same as exponential segmentation in terms of the byte hit ratio.

- $m < t$  :

Equation 2.19 can be written as:

$$P_{hit-E} = 1 - \frac{\sum_{i=t+1}^{i=n} \lambda_i \times L_i + \sum_{i=m+1}^{i=t} \lambda_i \times L_i \times (1 - \alpha)}{\sum_{i=1}^{i=n} \lambda_i \times L_i}.$$

Equation 2.20 can be written as:

$$P_{hit-L} = 1 - \frac{\sum_{i=k+1}^{i=t} \lambda_i \times L_i + \sum_{i=t+1}^{i=n} \lambda_i \times L_i}{\sum_{i=1}^{i=n} \lambda_i \times L_i}.$$

Thus, we must compare  $\sum_{i=m+1}^{i=t} \lambda_i \times L_i \times (1 - \alpha)$  and  $\sum_{i=k+1}^t \lambda_i \times L_i$ . If  $\sum_{i=m+1}^{i=t} \lambda_i \times L_i \times (1 - \alpha) > \sum_{i=k+1}^t \lambda_i \times L_i$ , lazy segmentation performs better. Otherwise, exponential segmentation performs better.

When  $m < t$ , there are  $m$  objects fully cached for exponential segmentation. Thus, for lazy segmentation, there are more objects fully cached and we can get that  $k > m$ .

Therefore, we further analyze the case when  $k > m$  and  $t > m$  as follows.

Since

$$\sum_{i=m+1}^{i=t} \lambda_i \times L_i \times (1 - \alpha) = \left( \sum_{i=m+1}^{i=k} \lambda_i \times L_i + \sum_{i=k+1}^{i=t} \lambda_i \times L_i \right) \times (1 - \alpha)$$

and

$$\sum_{i=k+1}^t \lambda_i \times L_i = \sum_{i=k+1}^t \lambda_i \times L_i \times \alpha + \sum_{i=k+1}^t \lambda_i \times L_i \times (1 - \alpha),$$

we must compare  $\sum_{i=m+1}^{i=k} \lambda_i \times L_i \times (1 - \alpha)$  and  $\sum_{i=k+1}^t \lambda_i \times L_i \times \alpha$ .

From Equations 2.14, 2.15, 2.16, we have

$$t = \frac{C}{L_{ave}^t} \times \frac{\beta}{\alpha}, \quad m = \frac{C}{L_{ave}^m} \times \frac{1-\beta}{1-\alpha}, \quad k = \frac{C}{L_{ave}^k}, \quad (2.21)$$

where  $L_{ave}^t$  denotes the average value of  $L_1$  to  $L_t$ ,  $L_{ave}^m$  denotes the average value of  $L_1$  to  $L_m$ , and  $L_{ave}^k$  denotes the average value of  $L_1$  to  $L_k$ .

Assume objects are of equal length, then  $L_{ave}^t = L_{ave}^m = L_{ave}^k = L_{ave}$ . (Note that this assumption simplifies the condition. However, we have proved it does not affect the conclusion we will draw.) Thus

$$t = k \times \frac{\beta}{\alpha}, \quad m = k \times \frac{1-\beta}{1-\alpha}.$$

Since  $\lambda_i \geq \lambda_{i+1}$ , we get

$$\sum_{i=m+1}^{i=k} \lambda_i \times L_i \times (1 - \alpha) \geq \sum_{i=m+1}^{i=k} \lambda_{k+1} \times L_i \times (1 - \alpha) = \lambda_{k+1} k (\beta - \alpha) L_{ave} \quad (2.22)$$

and

$$\sum_{i=k+1}^t \lambda_i \times L_i \times \alpha \leq \sum_{i=k+1}^t \lambda_{k+1} \times L_i \times \alpha = \lambda_{k+1} k (\beta - \alpha) L_{ave}. \quad (2.23)$$

Based on Equations 2.22 and 2.23, it is clear that exponential segmentation performs worse in byte hit ratio when  $m < t$ . This confirms the performance comparisons in section 2.6.

- $m > t$  :

Equation 2.19 can be written as:

$$P_{hit-E} = 1 - \frac{\sum_{i=t+1}^{i=m} \lambda_i \times L_i \times \alpha + \sum_{i=m+1}^{i=n} \lambda_i \times L_i}{\sum_{i=1}^{i=n} \lambda_i \times L_i}.$$

Equation 2.20 can be written as:

$$P_{hit-L} = 1 - \frac{\sum_{i=k+1}^{i=m} \lambda_i \times L_i + \sum_{i=m+1}^{i=n} \lambda_i \times L_i}{\sum_{i=1}^{i=n} \lambda_i \times L_i}.$$

Thus, we must compare  $\sum_{i=t+1}^{i=m} \lambda_i \times L_i \times \alpha$  and  $\sum_{i=k+1}^m \lambda_i \times L_i$  (Note that if  $k+1 > m$ ,  $\sum_{i=k+1}^m \lambda_i \times L_i$  is defined as  $-\sum_{i=m+1}^k \lambda_i \times L_i$ ). If  $\sum_{i=t+1}^{i=m} \lambda_i \times L_i \times \alpha > \sum_{i=k+1}^m \lambda_i \times L_i$ , exponential segmentation performs worse. Otherwise, lazy segmentation performs worse.

When  $m > t$ , there are only  $t$  objects that are fully cached for exponential segmentation, so it must be  $k > t$ . Thus, the further analysis will be done when  $m > t$  and  $k > t$  as follows.

Since

$$\sum_{i=t+1}^{i=m} \lambda_i \times L_i \times \alpha = \sum_{i=t+1}^{i=k} \lambda_i \times L_i \times \alpha + \sum_{i=k+1}^{i=m} \lambda_i \times L_i \times \alpha$$

and

$$\sum_{i=k+1}^m \lambda_i \times L_i = \sum_{i=k+1}^m \lambda_i \times L_i \times \alpha + \sum_{i=k+1}^m \lambda_i \times L_i \times (1 - \alpha),$$

it comes to compare  $\sum_{i=t+1}^{i=k} \lambda_i \times L_i \times \alpha$  and  $\sum_{i=k+1}^m \lambda_i \times L_i \times (1 - \alpha)$ .

Since  $\lambda_i \geq \lambda_{i+1}$ , with the same assumption as before that objects are of same length, it is easy to get

$$\sum_{i=t+1}^{i=k} \lambda_i \times L_i \times \alpha \geq \sum_{i=t+1}^{i=k} \lambda_{k+1} \times L_i \times \alpha = (\alpha - \beta)k\lambda_{k+1}L_{ave} \quad (2.24)$$

and

$$\sum_{i=k+1}^m \lambda_i \times L_i \times (1 - \alpha) \leq \sum_{i=k+1}^m \lambda_{k+1} \times L_i \times (1 - \alpha) = (\alpha - \beta)k\lambda_{k+1}L_{ave}. \quad (2.25)$$

Based on Equations 2.24 and 2.25, we can get that when  $m > t$ , lazy segmentation performs worse in terms of byte hit ratio under these assumptions. However,  $m > t$  leads to  $k > t$ , recall the analysis conclusion in 2.4.4.1, when  $k > t$ , lazy segmentation will perform better in terms of the delayed start request ratio.

In reality, exponential segmentation always caches all objects' beginning segments, thus,  $m > t$  is always true.

The results of the above analysis show that the performance of segment-based caching strategies is always a trade-off between the byte hit ratio and the delayed start request ratio. They are affected by the relationships of  $t$ ,  $m$ , which are determined by  $\alpha$ ,  $\beta$ ,  $n$  and  $L_{ave}$ . (Note that for the lazy segmentation strategy, in the sense that we do not reserve a part of the cache space for caching the beginning segments of objects,  $\beta = 0$ ; however, if counting the cache space used for caching the beginning segments of objects, a dynamically changing non-zero  $\beta$  is used. For the prefix caching,  $\beta$  is set to 100%.) Based on the analysis results, if  $m$  is decreased, the achieved byte hit ratio is reduced. However, the decrease of  $m$  leads to decrease of  $t$ , which results in a reduced delayed start request ratio. This seems to indicate that we can use the byte hit ratio to trade for delayed start request ratio. Whether this is true or not is critical to if we can find out an effective way to balance these two performance objectives. We will answer these questions heuristically after we derive the performance bounds for each performance objective.

### 2.4.5 Performance Bound Analysis

We have learned that these two performance objectives are always a trade-off between each other. However, how much performance can be optimized is not answered yet. In this section, we will give performance bounds based on the model so that they can guide the performance optimization under certain conditions as our assumptions state. We also assume the objects are of equal length as before. That is,  $L_{ave}^t = L_{ave}^m = L_{ave}^k = L_{ave}$ .

#### 2.4.5.1 Delayed Start Request Ratio

For the exponential segmentation strategy, substituting Equation 2.14 in Equation 2.17, we get

$$P_{delay-E} = \frac{\sum_{i=t+1}^{i=n} \lambda \times \frac{\eta \times \frac{1}{i^\theta}}{\sum_{i=1}^{i=n} \eta \times \frac{1}{i^\theta}}}{\sum_{i=1}^{i=n} \lambda \times \frac{\eta \times \frac{1}{i^\theta}}{\sum_{i=1}^{i=n} \eta \times \frac{1}{i^\theta}}} = \frac{\sum_{i=t+1}^{i=n} \frac{1}{i^\theta}}{\sum_{i=1}^{i=n} \frac{1}{i^\theta}}. \quad (2.26)$$

Carefully using the series theory and integration on Equation 2.26,

- $\theta = 1$

$$P_{delay-E} = \frac{\sum_{i=t+1}^{i=n} \frac{1}{i}}{\sum_{i=1}^{i=n} \frac{1}{i}} \leq \frac{\int_t^n \frac{1}{i} di}{\int_1^{n+1} \frac{1}{i} di} = \frac{\ln n - \ln t}{\ln(n+1)}$$

and

$$P_{delay-E} = \frac{\sum_{i=t+1}^{i=n} \frac{1}{i}}{\sum_{i=1}^{i=n} \frac{1}{i}} \geq \frac{\int_{t+1}^{n+1} \frac{1}{i} di}{1 + \int_1^n \frac{1}{i} di} = \frac{\ln \frac{n+1}{t+1}}{1 + \ln n}.$$

Having  $t = \frac{C}{L_{ave}} \times \frac{\beta}{\alpha}$ ,  $U = \frac{C}{L_{ave}}$ , we have

$$P_{delay-E}^{Max} = \frac{\ln n - \ln U \times \frac{\beta}{\alpha}}{\ln(n+1)} \quad (2.27)$$

and

$$P_{delay-E}^{Min} = \frac{\ln(n+1) - \ln(U \times \frac{\beta}{\alpha} + 1)}{1 + \ln n}. \quad (2.28)$$

For Equation 2.27 and 2.28, the larger the value of  $\beta$ , the smaller the values of  $P_{delay-E}^{Max}$  and  $P_{delay-E}^{Min}$ , and the smaller the value of  $\beta$ , the larger the values of  $P_{delay-E}^{Max}$  and  $P_{delay-E}^{Min}$ .

- $\theta \neq 1$

$$P_{delay-E} = \frac{\sum_{i=t+1}^{i=n} \frac{1}{i^\theta}}{\sum_{i=1}^{i=n} \frac{1}{i^\theta}} \leq \frac{\int_t^n \frac{1}{i^\theta} di}{\int_1^{n+1} \frac{1}{i^\theta} di} = \frac{n^{1-\theta} - t^{1-\theta}}{(n+1)^{1-\theta} - 1},$$

and

$$P_{delay-E} = \frac{\sum_{i=t+1}^{i=n} \frac{1}{i^\theta}}{\sum_{i=1}^{i=n} \frac{1}{i^\theta}} \geq \frac{\int_{t+1}^{n+1} \frac{1}{i^\theta} di}{1 + \int_1^n \frac{1}{i^\theta} di} = \frac{(n+1)^{1-\theta} - (t+1)^{1-\theta}}{n^{1-\theta} - \theta}.$$

Having  $t = \frac{C}{L_{ave}} \times \frac{\beta}{\alpha}$ ,  $U = \frac{C}{L_{ave}}$ , we have

$$P_{delay-E}^{Max} = \frac{n^{1-\theta} - (U \times \frac{\beta}{\alpha})^{1-\theta}}{(n+1)^{1-\theta} - 1} \quad (2.29)$$

and

$$P_{delay-E}^{Min} = \frac{(n+1)^{1-\theta} - (U \times \frac{\beta}{\alpha} + 1)^{1-\theta}}{n^{1-\theta} - \theta}. \quad (2.30)$$

$t^{1-\theta}$  is an *increasing* function when  $0 < \theta < 1$ , and a *non-increasing* function when  $\theta > 1$ . Thus, the larger the value of  $\beta$ , the smaller the values of  $P_{delay-E}^{Max}$  and  $P_{delay-E}^{Min}$ , and the smaller the value of  $\beta$ , the larger the values of  $P_{delay-E}^{Max}$  and  $P_{delay-E}^{Min}$ .

For the lazy segmentation strategy, substituting Equation 2.14 in Equation 2.18, we get

$$P_{delay-L} = \frac{\sum_{i=k+1}^{i=n} \lambda \times \frac{\eta \times \frac{1}{i^\theta}}{\sum_{i=1}^{i=n} \eta \times \frac{1}{i^\theta}}}{\sum_{i=1}^{i=n} \lambda \times \frac{\eta \times \frac{1}{i^\theta}}{\sum_{i=1}^{i=n} \eta \times \frac{1}{i^\theta}}} = \frac{\sum_{i=k+1}^{i=n} \frac{1}{i^\theta}}{\sum_{i=1}^{i=n} \frac{1}{i^\theta}}. \quad (2.31)$$

Carefully using the series theory and integration on Equation 2.31,



- $\theta = 1$

$$P_{delay-L}^{Max} = \frac{\ln n - \ln U}{\ln(n+1)} \quad (2.32)$$

and

$$P_{delay-L}^{Min} = \frac{\ln(n+1) - \ln(U+1)}{1 + \ln n}. \quad (2.33)$$

- $\theta \neq 1$

$$P_{delay-L}^{Max} = \frac{n^{1-\theta} - (U)^{1-\theta}}{(n+1)^{1-\theta} - 1} \quad (2.34)$$

and

$$P_{delay-L}^{Min} = \frac{(n+1)^{1-\theta} - (U+1)^{1-\theta}}{n^{1-\theta} - \theta}. \quad (2.35)$$

Equations 2.27, 2.29 and Equations 2.32, 2.34 give the upper bounds for the exponential segmentation and the lazy segmentation strategies, respectively, with different  $\theta$  conditions. Equations 2.28, 2.30 and Equations 2.33, 2.35 give lower bounds for them in the ideal situation.

#### 2.4.5.2 Byte Hit Ratio

For exponential segmentation, based on Equation 2.19, substituting the  $\lambda_i$  from Equation 2.14, we get

$$P_{hit-E} = 1 - \frac{\alpha \times \sum_{i=t+1}^{i=n} \frac{1}{i^\theta} + (1-\alpha) \times \sum_{i=m+1}^{i=n} \frac{1}{i^\theta}}{\sum_{i=1}^{i=n} \frac{1}{i^\theta}}. \quad (2.36)$$

Carefully using the series theory and integration on Equation 2.36,

- $\theta = 1$

$$P_{hit-E}^{Max} = 1 - \frac{\alpha \times \ln \frac{n+1}{U \times \frac{\theta}{\alpha} + 1} + (1-\alpha) \times \ln \frac{n+1}{U \times \frac{1-\theta}{1-\alpha} + 1}}{1 + \ln n} \quad (2.37)$$

and

$$P_{hit-E}^{Min} = 1 - \frac{\alpha \times \ln \frac{n}{U \times \frac{\beta}{\alpha}} + (1 - \alpha) \times \ln \frac{n}{U \times \frac{1-\beta}{1-\alpha}}}{\ln(n+1)}. \quad (2.38)$$

•  $\theta \neq 1$

$$P_{hit-E}^{Max} = 1 - \left( \frac{(n+1)^{1-\theta} - \alpha \times (\frac{\beta}{\alpha} \times U + 1)^{1-\theta}}{n^{1-\theta} - \theta} - \frac{(1 - \alpha) \times (\frac{(1-\beta)}{(1-\alpha)} \times U + 1)^{1-\theta}}{n^{1-\theta} - \theta} \right) \quad (2.39)$$

and

$$P_{hit-E}^{Min} = 1 - \left( \frac{n^{1-\theta} - \alpha \times (\frac{\beta}{\alpha} \times U)^{1-\theta}}{(n+1)^{1-\theta} - 1} - \frac{(1 - \alpha) \times (\frac{(1-\beta)}{(1-\alpha)} \times U)^{1-\theta}}{(n+1)^{1-\theta} - 1} \right). \quad (2.40)$$

For lazy segmentation, based on Equation 2.20, substituting the  $\lambda_i$  from Equation 2.14, we get

$$P_{hit-L} = 1 - \frac{\sum_{i=1}^{i=n} \frac{1}{i^\theta} - \sum_{i=1}^{i=k} \frac{1}{i^\theta}}{\sum_{i=1}^{i=n} \frac{1}{i^\theta}} = \frac{\sum_{i=1}^{i=k} \frac{1}{i^\theta}}{\sum_{i=1}^{i=n} \frac{1}{i^\theta}} \quad (2.41)$$

Carefully using the series theory and integration on Equation 2.41,

•  $\theta = 1$

$$P_{hit-L}^{Max} = \frac{1 + \ln U}{\ln(n+1)} \quad (2.42)$$

and

$$P_{hit-L}^{Min} = \frac{\ln(U+1)}{1 + \ln n}. \quad (2.43)$$

•  $\theta \neq 1$

$$P_{hit-L}^{Max} = \frac{U^{1-\theta} - \theta}{(n+1)^{1-\theta} - 1} \quad (2.44)$$

and

$$P_{hit-L}^{Min} = \frac{(U+1)^{1-\theta} - 1}{n^{1-\theta} - \theta}. \quad (2.45)$$

Equations 2.37, 2.39 and Equations 2.42, 2.44 give the upper bounds for the exponential segmentation and the lazy segmentation strategies, respectively, with different  $\theta$  conditions. Equations 2.38, 2.40 and Equations 2.43, 2.45 give the lower bounds for the ideal situation.

It is important to note that these upper and lower bounds are based on the model we built for ideal situations. Thus, the upper bounds we found here are valid upper bounds for general situations, while the lower bounds are only valid for ideal situations.

### 2.4.6 Analytical Results

To give an intuition into the dynamic nature of the two performance objectives, an example is given in Figure 2.1 based on Equation 2.29 and Equation 2.39. Here, given a total of 10000 original objects, we assume a cache size 20% of the total object size. Thus,  $U$  is set as 2000 object units. Furthermore,  $\theta$  and  $\alpha$  are set as 0.47 and 5%, respectively. As shown in

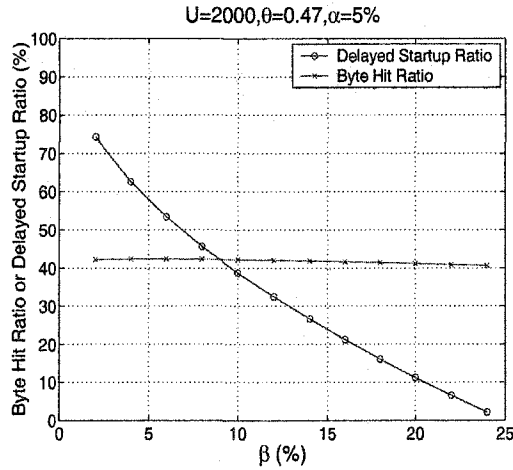


Figure 2.1: Byte hit ratio vs. delayed startup ratio

the figure, the decrease of the byte hit ratio is much slower than the decrease of the delayed startup ratio when  $\beta$  increases. Therefore, we can use a small decrease of byte hit ratio to trade for a significantly large reduction in the delayed startup ratio.

Mathematically, the partial derivative of  $P_{delay}^{Max}$  with respect to  $\beta$  yields  $|\Delta_{Delay}|$  which denotes the change of the delayed startup ratio. The partial derivative of  $P_{hit}^{Max}$  with respect to  $\beta$  yields  $|\Delta_{Hit}|$  which denotes the change of the byte hit ratio. Therefore, we have

$$\frac{|\Delta_{Delay}|}{|\Delta_{Hit}|} = \frac{1}{\alpha} \times \frac{\frac{N^{1-\theta}-\theta}{(N+1)^{1-\theta}-1}}{\left(\frac{1-\beta}{1-\alpha} \times \frac{\alpha}{\beta} + \frac{\alpha}{U\beta}\right)^{-\theta} + \left(\frac{\alpha}{U\beta} + 1\right)^{-\theta}}. \quad (2.46)$$

It can be shown that  $|\Delta_{Delay}|/|\Delta_{Hit}|$  is always greater than 1 when  $\alpha$  and  $\beta$  are less than 50%. For a long but complete derivation, please refer to [26].

The above analysis provides us with a solid basis to restructure the adaptive-lazy segmentation strategy in [23] by giving a higher priority to caching the startup length of objects in the replacement policy. The objective is to effectively address the conflicting interests between improving the byte hit ratio and reducing the delayed startup ratio for the best quality of media delivery. The analysis leads to the following improved replacement policy design.

#### 2.4.7 Improved Adaptive-Lazy Segmentation Strategy

In order to significantly reduce the startup latency with a small decrease of the byte hit ratio as suggested by our previous analysis result, a three-phase iterative replacement policy is re-designed as follows.

Based on a utility function defined similarly as in [23], upon an object admission, if there is not enough cache space, the proxy selects the object with the smallest utility value at that time as the victim, and the segment of this object is evicted in one of the two phases as follows. (1) First Phase: If the object is fully cached, the object is segmented

by the lazy segmentation method [23]. The first 2 segments are kept and the remaining segments are evicted right after the segmentation is completed. (2) Second Phase: If the object is partially cached with more than 1 segment, the last cached segment of this object is evicted. (3) Third Phase: If the victim has only the first segment and is to-be-replaced, then its startup length and the base segment length,  $L_b$ , is compared. If its startup length is less than the base segment length, the startup length is kept and the rest is replaced. Otherwise, it will be totally replaced. The utility value of the object is updated after each replacement and this process repeats iteratively until the required space is found.

This restructured adaptive and lazy segmentation strategy has shown its effectiveness in [26] by well balancing the two performance objectives.

## 2.5 The Hyper-Proxy System

Having the answers to balance the two pairs of conflicting performance objectives in the previous sections, we design a high quality media streaming proxy system, called Hyper Proxy system, following our design model. In our design, for any media object accessed through the proxy, a data structure containing the following items in Table 2.2 is created and maintained. This data structure is called the access log of the object.

For each object, the  $L_{thd}$  is calculated after the segmentation (see section 2.5.3). It is equal to  $\max(\text{startup length}, \text{free-of-jitter length}, 2L_b)$  and its value varies due to the dynamic nature of  $B_s$  and  $B_t$ . In the system, two object lists (*premium list* and *basic list*) are maintained. The *basic list* contains all the objects whose length of cached segments is larger than its  $L_{thd}$  while the *premium list* contains all the objects whose cached data

**Table 2.2:** The items of *Hyper-Proxy* data structure for each object

$T_1$	the time instance the object is firstly accessed
$T_r$	the last reference time of the object
$T_c$	the current time instance
$L_{sum}$	the sum of each access duration to the object
$n_a$	the number of accesses to the object
$L_b$	the length of the base segment
$n$	the number of the cached segments of the object
$FG_{adm}$	the admission flag for admitting segments
$L_{thd}$	the threshold length used in the replacement policy
$L_{avg}$	the average access duration of an object
$F$	the access frequency

length is equal to or less than its  $L_{thd}$ .  $FG_{adm}$  is the flag used to indicate the priority of new segment admission. Items  $L_{avg}$  and  $F$  can be derived from the items above. They are used as measurements of access activities to each object. At time instance  $T_c$ , the access frequency  $F$  is  $n_a/(T_r - T_1)$ , and the average access duration  $L_{avg}$  is  $L_{sum}/n_a$ .

When an object is accessed for the first time, it is fully cached and linked to the *basic list* according to the admission policy. A fully cached object is kept in the cache until it is chosen as an eviction victim according to the replacement policy. At that time, the object is segmented and some of its segments are evicted. The object is also transferred to the *premium list*. Once the object is accessed again, the proxy uses the active prefetching method to determine when to prefetch which uncached segment. Then the segments of the object are adaptively admitted by the admission policy or adaptively replaced by the replacement policy.

We now present the detailed description of four major modules in the Hyper-Proxy caching system.

### 2.5.1 Priority-based Admission Policy

For any media object, cache admission is considered whenever the object is accessed.

- A requested object with no access log indicates that the object is accessed for the first time. The object is then cached in full regardless of the request's accessing duration. The replacement policy (see section 2.5.4) is activated if there is not sufficient space. The victim is selected from objects in the *basic list*, or *premium list* when the *basic list* is empty. In the *premium list*, objects with *PRIORITY* flag are searched if no object with *NON-PRIORITY* flag is in *premium list*. The fully cached object is linked to the *basic list* and an access log is created for the object and the recording of the access history begins. If an access log exists for the object (not the first access to the object), but the log indicates that the object is fully cached, the access log is updated. No other action is necessary.
- If an access log exists for the object, and its  $FG_{adm}$  is *PRIORITY* (see section 2.5.2), the proxy considers the admission of the next uncached segment or segments determined by its *free-of-jitter length*. Whether the segment(s) can be admitted or not depends on if the replacement policy can find a victim or not. Victim selection is limited to objects in the *basic list* or *premium list* with *NON-PRIORITY* flag if *basic list* is empty. Note that for this admission, the system does not need to compare the caching utility value of this object with the victim's, but only to find a victim with the smallest utility value.
- If an access log exists for the object, and its  $FG_{adm}$  is *NON-PRIORITY* (see section 2.5.2), the next uncached segment is considered for admission only if  $L_{avg} \geq$

$(n + 1)L_b/L_{thd}$  (Note  $L_{avg}$  is changing dynamically.). The inequality indicates that the average access duration is increasing to the extent that the cached  $n$  segments cannot cover most of the requests while a total of  $n + 1$  segments can. Whether the next uncached segment is eventually admitted or not depends on whether or not the replacement policy can find a victim whose caching utility is less than this object. The victim selection is limited to the *basic list* only.

After the admission, the object will be transferred to the *basic list* if it is in the *premium list* and its cached data length is larger than its  $L_{thd}$ .

In summary, using the priority-based admission, the object is fully admitted when it is accessed for the first time. Then the admission of this object is considered segment by segment with the higher priority given to the admissions that are necessary for in-time prefetching.

### 2.5.2 Active Prefetching

After the object is segmented and some of its segments are replaced (see section 2.5.4), the object becomes partially cached. Then, upon each subsequent access, active prefetching is activated to determine when to prefetch which segment once the object is accessed according to the following various conditions.

- $n = 0$ : No segment is cached. The prefetching of the  $\lceil \frac{B_s}{B_t} \rceil^{th}$  segment is considered.

The  $FG_{adm}$  of this object is set to be *PRIORITY*.

- $n > 0$  and  $n + 1 < \frac{B_s}{B_t}$ : The proxy starts to prefetch the  $\lceil \frac{B_s}{B_t} \rceil^{th}$  segment once the client starts to access the object. If the segments between  $n + 1^{th}$  and  $\lceil \frac{B_s}{B_t} - 1 \rceil^{th}$



are demanded, proxy jitter is inevitable and the  $FG_{adm}$  of this object is set to be *PRIORITY*.

- $n > 0$  and  $n + 1 \geq \frac{B_s}{B_t}$ : The prefetching of  $n + 1^{th}$  segment starts when the client accesses to the position of  $(n + 1 - \frac{B_s}{B_t})L_b$  of the first  $n$  cached segments. The  $FG_{adm}$  of this object is set to be *NON-PRIORITY*.

Note that  $B_s$  and  $B_t$  are sampled when each segment is accessed. As a result,  $L_{thd}$  is also updated accordingly.

### 2.5.3 Lazy Segmentation Policy

The key of the lazy segmentation strategy is as follows. Once there is no cache space available and replacement is needed, the replacement policy calculates the caching utility of each cached object (see section 2.5.4). Subsequently, the object with the smallest utility value is chosen as the victim if it is not active (no request is accessing it). If the victim object turns out to be fully cached, the proxy segments the object as follows. The average access duration  $L_{avg}$  at current time instance is calculated. It is used as the length of the base segment, that is,  $L_b = L_{avg}$ . Note that the value of  $L_b$  is fixed once it is determined. The object is then segmented uniformly according to  $L_b$ . After that, the first  $\lceil \frac{L_{thd}}{L_b} \rceil$  segments are kept in cache, while the rest are evicted (see section 2.5.4). The number of cached segments,  $n$ , is updated in the access log of the object accordingly. This lazy segmentation scheme allows better determination of  $L_b$ .

### 2.5.4 Differentiated Replacement Policy

The replacement policy is used to re-collect cache space by evicting selected victims. First of all, a utility function is defined below to help the victim selection process by identifying the least valuable object as the victim.

$$\frac{F \times \frac{L_{sum}}{n_a} \times \min(1, \frac{T_r - T_1}{T_c - T_r})}{nL_b}, \quad (2.47)$$

In the above equation, the caching utility value is proportional to

- (1)  $F$ , which estimates the average number of future accesses;
  - (2)  $\frac{L_{sum}}{n_a}$ , which estimates the average duration of future access;
  - (3)  $\min(1, \frac{T_r - T_1}{T_c - T_r})$ , which estimates the possibility of future accesses;
- The system compares the  $T_c - T_r$ , the time interval between now and the most recent access, and the  $\frac{T_r - T_1}{n_a}$ , the average time interval between accesses occurring in the past. If  $T_c - T_r > \frac{T_r - T_1}{n_a}$ , the possibility that a new request arrives soon for this object is small. Otherwise, it is highly possible that a request is coming soon.

and inversely proportional to

- (4)  $nL_b$ , which represents the disk space required.

Corresponding to the different situations of admission, when there is not enough space, the replacement policy selects the victim with the smallest utility value from different lists in the order as designated in section 2.5.1. Then partially cached data of the victim is evicted as follows.

- If the victim is fully cached in the *basic list*, the object is segmented as described in section 2.5.3. The first  $\lceil \frac{L_{thd}}{L_b} \rceil$  segments are kept and the remaining segments are

evicted right after the segmentation is completed. The object is removed from the *basic list* and linked to the *premium list*.

- If the victim is partially cached in the *basic list*, the last cached segment of this object is evicted. After the eviction, the object will be linked to the *premium list* if its cached data length is less than or equal to its  $L_{thd}$ . Note this object can be selected as victim again if no sufficient space is found in this round.

- If the victim is in the *premium list*, the last cached segment of this object is evicted. If no data of this object is cached, it is removed from the *premium list*.

The utility value of the object is updated after each replacement and this process repeats iteratively until the required space is found.

The design of the differentiated replacement policy gives a higher priority for reducing proxy jitter, reduces the erroneous decision of the replacement and gives fair chances to the replaced segments so that they can be cached back into the proxy again by the aggressive admission policy if they become popular again.

Note that after an object is fully evicted, the system still keeps its access log. If not, once the object is occasionally accessed again, it should be fully cached again. Since media objects tend to have diminishing popularities as the time goes on, if the system caches the object in full again, this results in an inefficient use of the cache space. Our design enhances the resource utilization by avoiding this kind of situation. By setting a large enough time-out threshold, the proxy deletes the access logs of unpopular objects eventually.

## 2.6 Performance Evaluation

### 2.6.1 Workload Summary

To evaluate the performance of the Hyper-Proxy system, we conduct extensive simulations based on several workloads. Both synthetic workloads and a real workload extracted from enterprise media server logs are considered. We designed two synthetic workloads. These workloads assume a Zipf-like distribution [4, 38, 62] (Note there are some arguments to this distribution in recent research work [7, 9, 52].) with a skew factor  $\theta$  for the popularity of the media objects and request inter arrival follows the Poisson distribution with a mean interval  $\lambda$ .

The first synthetic workload simulates accesses to media object in the Web environment in which the length of the video varies from short ones to longer ones. We use WEB as the name of this workload. The second simulates the Web accesses where clients accesses to objects are incomplete, that is, a started session terminates before the full media object is delivered. We simulate this behavior by designing a partial viewing workload based on the WEB workload. We use PART as its name. In this workload, 80% of the sessions terminate before 20% of the object is delivered.

For the real workload named as REAL, we use logs from HP Corporate Media Solutions, covering the period from April 1 through April 10, 2001. There is a total of 403 objects, and the unique object size accounts to 20 GB. There is a total of 9000 requests during this period. Our analysis shows that about 83% of the requests only view the objects for less than 10 minutes and more than 56% of the requests only view less than 10% of their requested objects. About 10% of the requests view the whole objects.

Table 2.3 lists some characteristics of these workloads. A detailed analysis of the overall characteristics of the logs from the same servers covering different time periods can be found in the reference [30].

**Table 2.3:** The workload summary

Workload Name	Num of Request	Num of Object	Size (GB)	$\lambda$	$\theta$	Range (minute)	Duration (day)
WEB	15188	400	51	4	0.47	2-120	1
PART	15188	400	51	4	0.47	2-120	1
REAL	9000	403	20	-	-	6 - 131	10

### 2.6.2 Performance Results

In the simulation experiments, the streaming rate of accessed objects is set randomly in the range from half to four times that of the link capacity between the proxy and the server. We use the *jitter byte ratio* to evaluate the quality of the continuous streaming service provided by the proxy system. It is defined as the amount of data that is not prefetched in time by the proxy normalized by the total bytes demanded by the streaming sessions. Delayed prefetching causes potential playback jitter at the client side. A good proxy system should have small jitter byte ratio. The second metric we use is the *delayed startup ratio*, which is the number of requests that are served with a startup latency normalized by the total number of requests. The last metric we use is the *byte hit ratio*, which is the amount of data delivered to the client from the proxy cache normalized by the total bytes the clients demand.

We evaluate these three metrics in three designs of a segment-based proxy caching system. The *Proxy-Hit* represents the adaptive-lazy segmentation based proxy caching

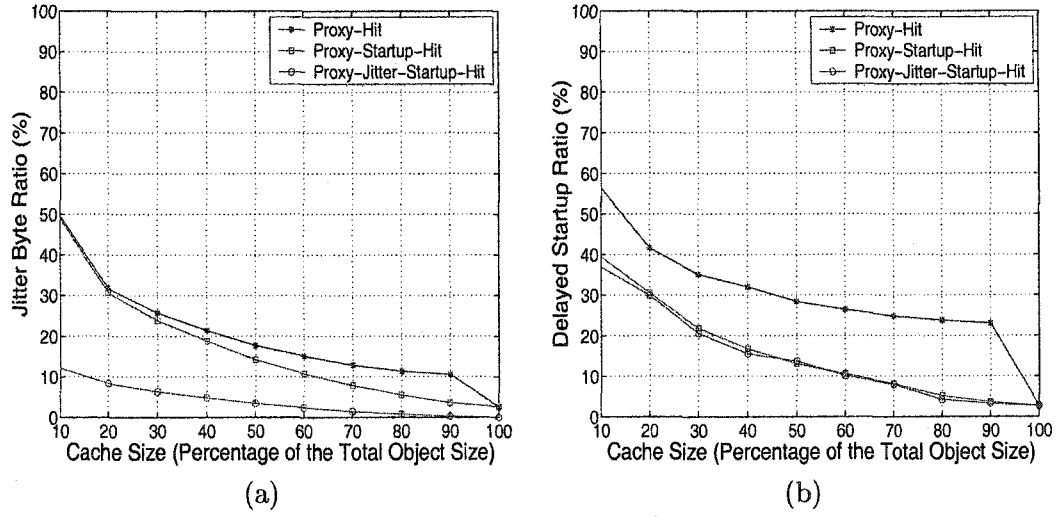


Figure 2.2: WEB: jitter byte ratio and delayed startup ratio

system [23] with active prefetching. This scheme aims at improving the byte hit ratio. The *Proxy-Startup-Hit* represents the improved adaptive-lazy segmentation based proxy caching system with active prefetching. This scheme is designed to reduce the delayed startup ratio subjective to improving the byte hit ratio. The *Proxy-Jitter-Startup-Hit* represents our proposed *Hyper-Proxy* system in this study, aiming at minimizing proxy jitter subjective to minimizing the delayed startup ratio while maintaining a high byte hit ratio.

For the WEB workload, the object encoding rate ranges in the 28Kbps-256Kbps, while the available network bandwidth for its uncached segments prefetching is randomly selected in the range of half to twice of its encoding rate. Figure 2.2(a) shows that Hyper-Proxy always provides the best continuous streaming service to the client while Proxy-Hit system which aims at increasing byte hit ratio, performs worst. Specifically, when cache size is 20% of total object size, Hyper-Proxy reduces proxy jitter by more than 50%.

Figure 2.2(b) shows that Hyper-Proxy achieves the lowest delayed startup ratio. Proxy-

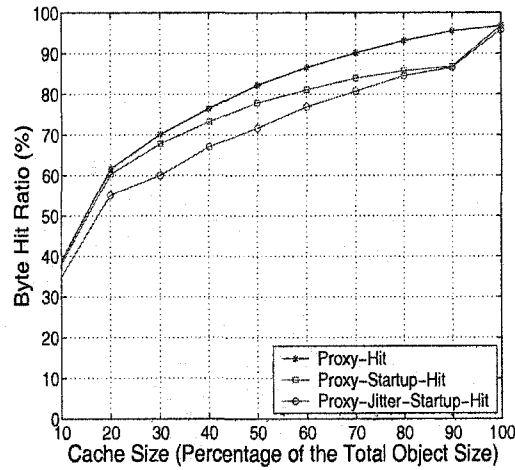


Figure 2.3: WEB: byte hit ratio

Startup-Hit achieves results close to Hyper-Proxy. This is expected as we have analyzed in [26].

Figure 2.3 shows Hyper-Proxy achieves a relatively low byte hit ratio, which indicates a smaller reduction of network traffic. This is the price to pay for less proxy jitter and the smaller delayed startup ratio as shown in Figure 2.2(a) and (b).

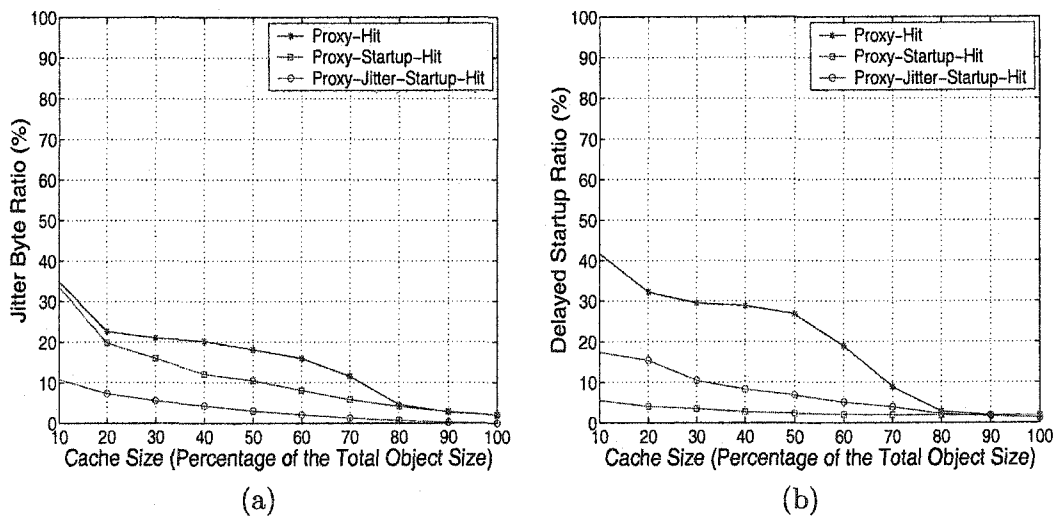


Figure 2.4: PART: jitter byte ratio and delayed startup ratio

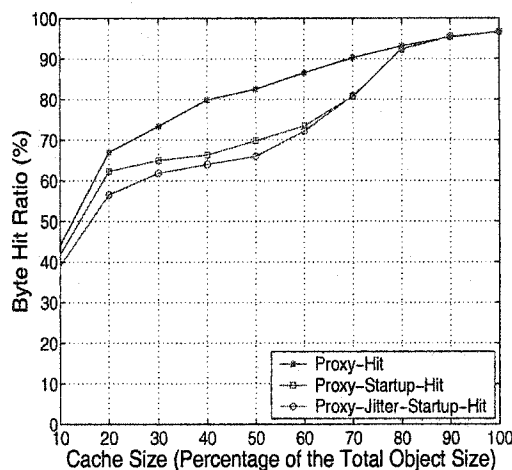


Figure 2.5: PART: byte hit ratio

In PART, the object encoding rate and the available network bandwidth to prefetch its uncached segments are set as in WEB. Similar results are observed for the PART workload as shown in Figure 2.4 and Figure 2.5.

As shown on Figure 2.4, when cache size is 20% of total object size, Hyper-Proxy reduces proxy jitter by 50% by giving up less than 5 percentage points in the byte hit ratio. Figure 2.4(b) shows that Proxy-Startup-Hit achieves the best performance in reducing the delayed startup ratio. The result is expected since this scheme is specifically designed to prioritize reducing the delayed startup ratio. On the other hand, since Hyper-Proxy proactively prevents proxy jitter by keeping more segments, more cache space is used for segments that may not be requested due to early termination. This in turn makes Hyper-Proxy perform not well in reducing the delayed startup ratio.

Not surprisingly, Figure 2.5 the Hyper-Proxy achieves the lowest byte hit ratio when comparing with Proxy-Hit and Proxy-Startup-Hit.

In a more realistic setup, we use the REAL workload to evaluate performance. The encoding rate for an object in REAL is the same as recorded in the log, while we take



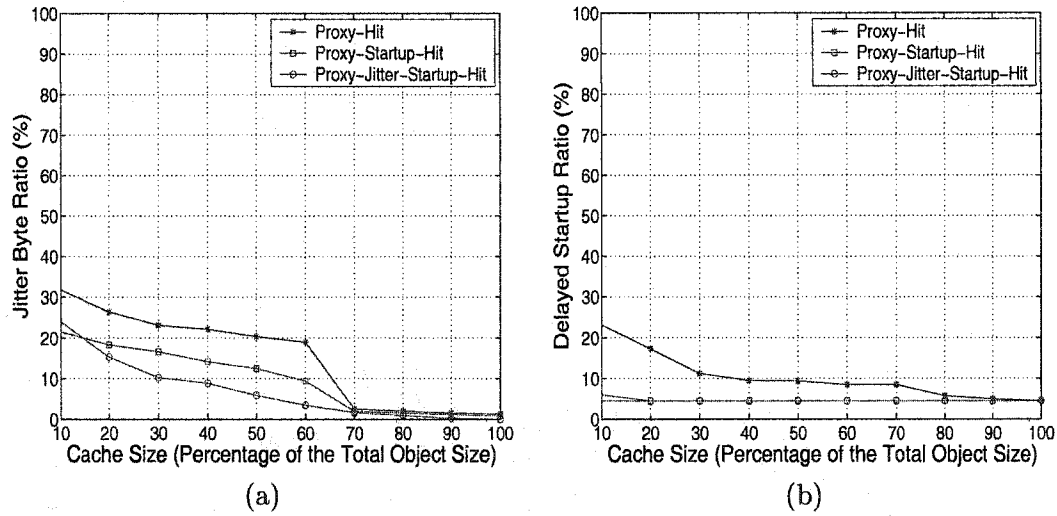


Figure 2.6: REAL: jitter byte ratio and delayed startup ratio

the client connection link bandwidth as the available bandwidth for its uncached segment prefetching.

As shown in Figure 2.6 and Figure 2.7, Hyper Proxy performs best in reducing proxy jitter and delayed startup. The performance degradation in byte hit ratio is also acceptable. As shown on Figure 2.7, the byte hit ratio achieved by Proxy-Startup-Hit is larger than that achieved by Proxy-Hit when the available cache size is greater than 40% of the total object size. This is because the cache size is large enough to cache the startup lengths of the objects. For this evaluation, it is also interesting that the byte hit ratio achieved by the Proxy-Hit system is not as high as without considering bandwidths. Studying different situations, we find that in our simulation the available bandwidth of the proxy-server link is typically much smaller than the object encoding rate, causing a large number of byte misses in Proxy-Hit due to request busty, which however would not have happened without considering the bandwidth constrain.

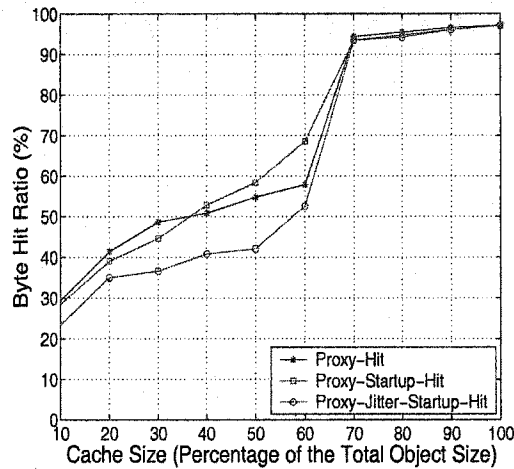


Figure 2.7: REAL: byte hit ratio

## 2.7 Summary

Proxy has been successfully used for caching text-based content. Using proxy to support media delivery is cost-effective, but challenging due to the nature of large media sizes and the low-latency and continuous streaming demand. Most existing studies target at improving the byte hit ratio that is commonly used in standard proxy caching. However, this is not the major concern for streaming media delivery, because it does not guarantee the continuous media delivery when the to-be-viewed object segments are not cached in the proxy, which causes proxy jitter. Our contributions in this study are as follows:

- We have presented an optimization model to guide designs of highly effective media proxy caches and ensure a high delivery quality to the clients, which aims at minimizing proxy jitter subject to reducing the startup latency and increasing the byte hit ratio.
- We have provided insights into the model by analyzing two pairs of conflicting interests and trade-offs inherent in this model.

- We have proposed to build a new media proxy caching system called Hyper-Proxy. This system addresses the interests from the perspectives of both clients and Internet resource management with a high priority given to the clients. We have shown that the Hyper-Proxy system minimizes the amount of proxy jitter with a low delayed startup ratio and acceptable low network traffic compared with other existing caching schemes.

## Chapter 3

# Implementation and Evaluation of a Segment-based Streaming Media Proxy

### 3.1 Introduction

The delivery of diverse streaming media contents on IP networks in a cost effective manner, while maintaining high quality, is challenging but highly desirable for many applications. In a Web service environment, a continuous streaming session (often with a duration of minutes or hours, compared to milliseconds or seconds for traditional Web pages) keeps consuming network bandwidth and disk bandwidth on the hosting server. Multiple concurrent streaming sessions can easily exhaust the available network bandwidth and overload the media content server. Placing multimedia objects closer to clients is an effective solution that will relieve the network bottleneck and reduce the load on the media content server.

Research efforts have been made to extend existing proxy cache methods of static Web pages to the case of streaming media objects. Streaming media objects have some features

that makes caching promising: the objects are generally static and do not change for a long time. Moreover, they show some degree of locality of reference. However, as stated in Chapter 2, proxy caching of multimedia objects is also challenging due to the typical large size and the low-latency and continuous streaming demand of media objects.

To handle these problems, several *partial* caching methods have been proposed, which divide media objects into smaller units, more feasible for caching. There are two types of partial caching approaches according to the object segmentation directions. The first divides objects in the time domain [94, 104, 117], which we call *segment-based* approaches. The second is to divide objects in the media quality domain [34, 70, 92, 93].

Although some algorithmic solutions and prototypes are available, today the practical usage and deployment of such systems are rare. Mocha [95] and QBIX [100] are prototype systems that divide media objects along the quality domain. Mocha is based on layered encoded streams, while QBIX tries to leverage MPEG4 and MPEG7 standards to do quality adaptation. However, they have not been widely deployed since they require extensive support from Internet Service Providers. For example, for Mocha, there are almost no layered-encoded streams provided online today. QBIX requires an online transcoding proxy, and does not work for videos in formats other than MPEG4 and MPEG7. Moreover, the quality of the media objects served in these systems is not controlled by the client, but by the service provider. Thus, they may not be client friendly.

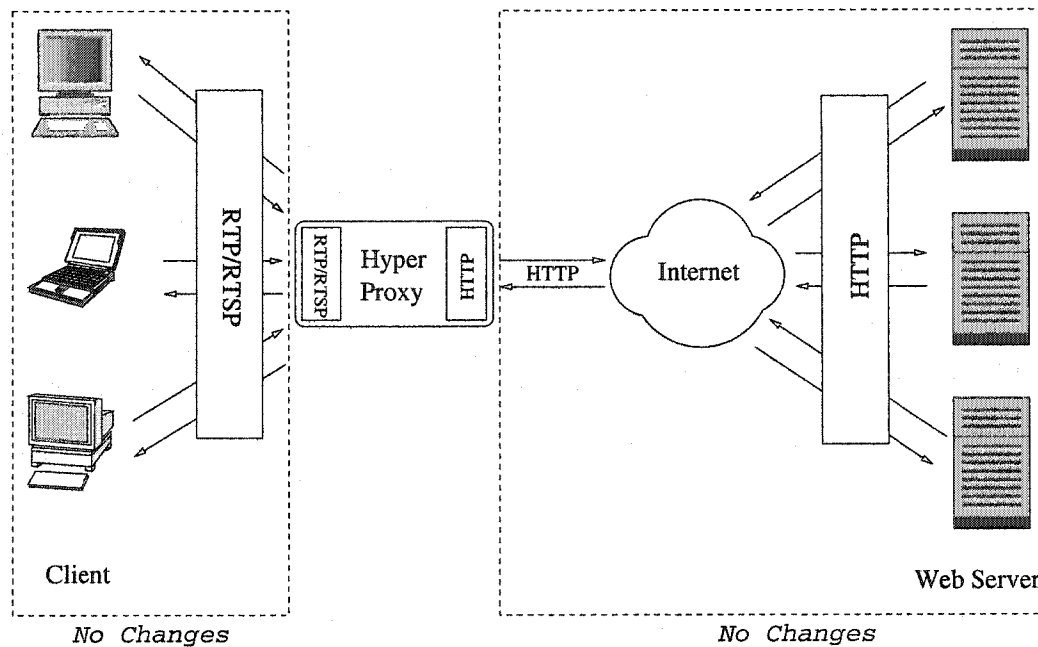
When dividing media objects in the time domain (a segment-based approach), the aforementioned problems do not exist. The media with the original quality can always be served to the client. However, there are a number of technical problems. First, multimedia objects are stored in container files, such as MP4 [71]. The file contains both audio and video

tracks. In addition, it also contains indices to audio and video media packets, and may contain hint tracks with meta information. The flexibility of positioning these elements in the container file makes media aware segmentation difficult for the proxy.

Second, media content is usually streamed using the RTP protocol, running on top of UDP. In practice, UDP traffic is likely to be blocked by firewalls at the client side due to security considerations. Also, Internet wide UDP-based communication raises reliability and fairness concerns. UDP packets are often subject to dropping at intermediate routers and switches. On the other hand, large amounts of unregulated UDP traffic unfairly throttles TCP traffic [61]. These concerns make it difficult to deploy the system based on UDP connecting the proxy and the server [13].

Finally, after the object is segmented, the coordination between the caching of discrete object segments and the streaming of continuous media data is challenging. For example, although different online prefetching algorithms have been proposed to provide continuous streaming to clients, few measurement results in Internet streaming have been reported. Precise prefetching techniques [24] can provide continuous streaming with maximum resource utilization. However, system support is needed to accurately estimate the available bandwidth of the proxy – content-server link at runtime.

We have designed and implemented a segment-based proxy, named *Hyper-Proxy*, to address these problems. It leverages existing Internet infrastructure and is able to serve and cache media objects in time-domain segments. As shown in Figure 3.1, the deployment of Hyper-Proxy does not require modifications on either the server side or the client side. This design takes advantage of the prevalence of HTTP, and eliminates most of the concerns about UDP based communications, especially when the proxy is placed inside the firewall.



**Figure 3.1:** Organization and protocols used in Hyper-Proxy system

The Hyper-Proxy system uses a segment-aware file I/O system that enables automatic segmentation and intelligent prefetching techniques to guarantee continuous streaming. This allows Hyper-Proxy to transparently handle the complexity of media formats and to support continuous delivery demands. It has the following merits:

1. Hyper-Proxy handles client requests for streaming media objects via the standard RTSP [102] and RTP [101] protocols. It communicates with the content-server using the HTTP protocol. This design allows a regular Web server to serve streaming content, as well as regular Web documents. Thus, the existing Internet infrastructure is fully leveraged without any extra support.
2. A client request is processed and divided into multiple sub-requests. Each sub-request asks for only a small part of the whole media object. The sequence of sub-requests is stopped whenever the client terminates its session, which subsequently terminates the

data transfer. This design introduces a low startup latency while providing efficient bandwidth utilization.

3. Prefetching techniques are implemented to assist high quality continuous streaming. Based on dynamically detected available bandwidths of the proxy-server link, active prefetching techniques are used to dynamically prefetch the data likely to be accessed by the client.
4. The data contained in each segment is stored as a distinct object. The existing popularity based replacement policy is leveraged from the traditional Web proxy, and applied on these segments. It is a global, segment-based replacement policy instead of a media object-based one, which enables better utilization of the cache space.

Actual implementation of Hyper-Proxy is evaluated under various conditions. The effect of different segment sizes on streaming performance is evaluated and compared with the performance of a full object caching approach. Different proxy-to-server network proximity and available bandwidth scenarios are also considered. We tested its cache performance based on an actual workload. Our extensive experimental results show that Hyper-Proxy consistently provides high quality streaming delivery to clients, with reduced startup latency and more efficient cache utilization.

The rest of this chapter is organized as follows. We review related work in Section 3.2. We present the design and implementation of Hyper-Proxy in Section 3.3. We evaluate the system performance through extensive experiments in Section 3.4. We summarize the chapter in Section 3.5.



## 3.2 Related Work

The research on proxy caching of streaming media content has received much attention lately. Early efforts, e.g. Middleman [5], which has studied cluster of proxies for streaming media delivery, have considered little on one important feature of streaming media accessing. It is found that continuous media objects such as video or music clips are often partially accessed. Based on this observation, *partial* caching approaches have been proposed to reduce the cache space requirement. The basic strategy is to cache segments of objects that are divided in the viewing time domain. Typical examples include prefix caching [104], uniform segmentation [94], exponential segmentation [117]. Prefix caching always caches the prefix of the objects to minimize the startup latency. The optimal prefix length can be calculated according to [111]. Its protocol consideration as well as partial sequence caching were studied in [51]. In uniform segmentation, objects are cached in uniform-size segments, while in exponential segmentation, the segment size doubles along the viewing direction. Considering limited resources available from a single cache, the Rcache [18] considers the usage of multiple proxies, focusing on the memory and disk utilization. These strategies focus on protocol design or benefit analysis based on artificial workloads. Recently, authors in [123] proposed a flexible and scalable proxy testbed to support a wide and extensible set of next-generation proxy streaming services. Our work is based on the uniform segmentation caching strategy with the focus on real system implementation and evaluation of the system in real networking environments using real workloads.

The partial caching strategy can be extended to the quality domain. Layered caching techniques [92, 93] have demonstrated efficient usage of cache space by considering different

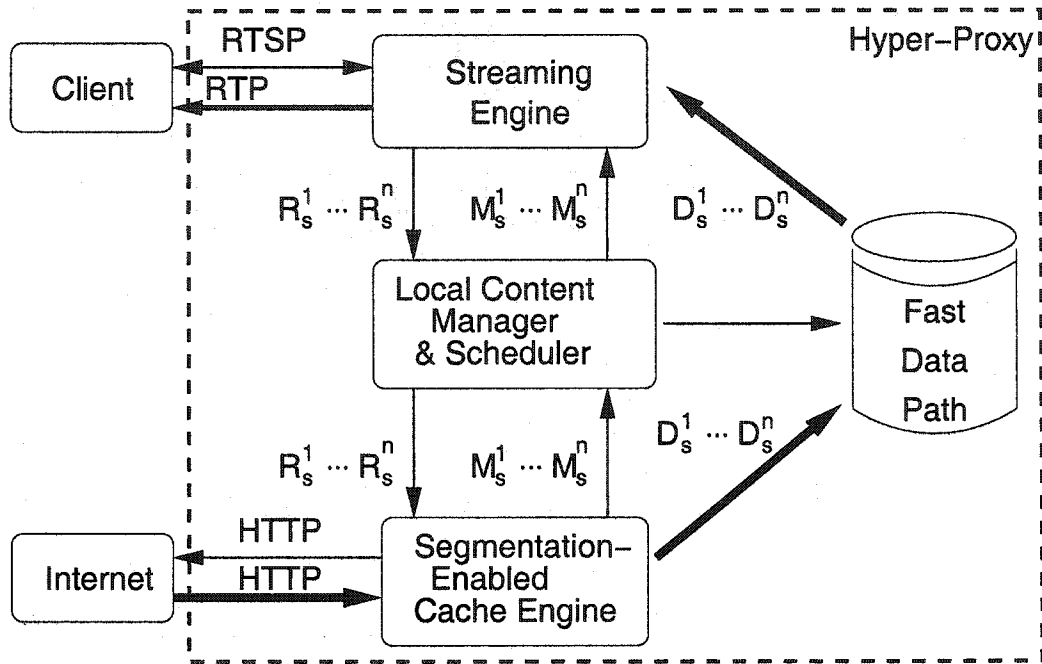
QoS characteristics of client devices or connectivities. A comparison with multiple version caching is studied in [70] while a model of layered-encoded object distribution is studied in [68]. In [79], the proposed approach attempts to select groups of consecutive frames by the selective caching algorithm, while in [82], the algorithm may select groups of non-consecutive frames for caching in the proxy. A different idea is proposed in video staging [124], in which a portion of bits from the video frames whose size is larger than a predetermined threshold is cut off and prefetched to the proxy a priori to reduce the bandwidth on the server proxy channel. Recently, a fine grained, network aware and media adaptive rate control scheme is used in caching of scalable streaming content [76]. Most of partial caching schemes in quality domain require layered encoded objects or additional support from the proxy or client. The work presented in this chapter does not have these requirements.

### 3.3 Implementation of Hyper-Proxy

Figure 3.2 shows the architecture of a Hyper-Proxy, as well as its request handling. The Hyper-Proxy is composed of four main components: a streaming engine that interfaces with the client, a segmentation-enabled cache engine that interfaces with content servers, a Local Content Manager and Scheduler (LCMS) module that coordinates the streaming engine and the cache engine, and a high speed disk that provides a fast data-path via the local file system.

#### 3.3.1 Streaming Engine

The streaming engine is a multi-threaded media server. It is responsible for providing an interface to the client, which is described in detail in [97]. As shown in Figure 3.2, it receives



**Figure 3.2:** Internal design of the Hyper-Proxy system: A client request is divided into  $n$  sub-requests with different ranges,  $R_s^1$  to  $R_s^n$ , requesting different content segments,  $D_s^1$  to  $D_s^n$ . The Local Content Manager and Scheduler controls when to send the next sub-request. The cache engine returns segment meta data ( $M_s^1$  to  $M_s^n$ ) to the Local Content Manager and Scheduler, and caches the segments  $D_s^1$  to  $D_s^n$  on the disk.

a client request for a RTSP URL and converts it to multiple segment requests,  $R_s^1 \dots R_s^n$ , that are sent to the LCMS. It uses the meta-data information,  $M_s^1 \dots M_s^n$ , returned by the cache engine through the LCMS to access the raw data segments on the disk.

A typical client request for `foo.mp4` is as follows.

```
RTSP://streaming-proxy:port1//content-server:port2/foo.mp4
```

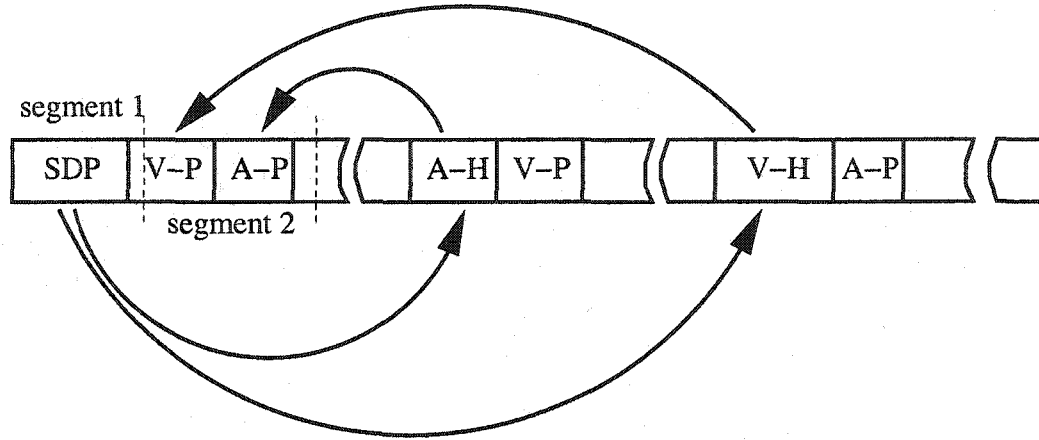
In this URL, RTSP denotes the protocol used. The second “//” is used to specify the content server and the URL. In the streaming engine, such a client request is normally processed with the following four messages sent to the proxy in the order.

- **DESCRIBE:** First, a DESCRIBE message is sent. The DESCRIBE retrieves the description of the media object identified by the URL. The description contains the

information about the presentation format, the static and temporal properties, etc. as specified in Session Description Protocol (SDP) [55].

- **SETUP**: After the SDP information of the requested object is fetched and processed correctly, the SETUP message is sent to notify the proxy to allocate corresponding resources for a stream and to create a RTSP session for the request. Note that the RTSP session for a stream includes both audio and video streams.
- **PLAY**: If the SETUP is successful, the PLAY message notifies the proxy to start data transmission for a stream on the channels allocated via SETUP.
- **TEARDOWN**: When the streaming is complete or the client terminates the request, the TEARDOWN message is sent. Basically, it frees resources associated with the stream and the RTSP session is eliminated.

As shown in Figure 3.2, the streaming engine reads data segments,  $D_s^1 \dots D_s^n$ , from the disk to serve clients after the **PLAY** is received and processed. However, there is a problem: a randomly chosen segment length breaks the object into pieces, thus creating segments that are likely to include an incomplete media packet as shown on Figure 3.3, where a MP4 file is divided into incomplete pieces. If this incomplete packet is sent to the client, the client player would have to use error concealment or it may crash. One solution to this problem is to always segment the object on a packet boundary, which requires Hyper-Proxy to have packet boundary knowledge *before* segmentation can be done. This information could be obtained by parsing the complete media file, or by using a hint track, if available. However, the hint track data can be dispersed through the media file, so in either case, the whole file may have to be downloaded. A better solution is to allow



**Figure 3.3:** High level abstraction of an MP4 file: SDP represents the SDP information. V-P represents video data packet while A-P represents audio-data packet. A-H represents audio hint track information, while V-H represents video hint track information. The media data packets are accessed from the pointers of SDP and hint track information in the order.

random segment boundaries, but to always feed a complete data packet to the client. In the Hyper-Proxy system, a segment-aware file I/O system is implemented to support this requirement. It automatically requests the appropriate segment when reading or seeking beyond the boundaries of the current segment. The LCMS tries to ensure that the next segment is always available in the cache.

### 3.3.2 Local Content Manager and Scheduler

The Local Content Manager and Scheduler (LCMS) coordinates the streaming engine and the segmentation-enabled cache engine. It converts the sub-requests, e.g.,  $R_s^1 \dots R_s^n$ , to corresponding HTTP requests (with Range headers) and forwards them to the cache engine. It returns the appropriate cache meta-data  $M_s^1 \dots M_s^n$  from the proxy replies to the streaming engine. More importantly, the LCMS schedules segment prefetching. Prefetching is necessary because segment-based proxy caching is a partial caching solution, in which only a part of the object is cached in the proxy while a client may access an object to a

segment which is not cached in the system. To guarantee continuous media delivery, each segment should be available locally before the streaming engine tries to read and stream to the client. Otherwise, the client can experience playback jitter.

We have implemented multiple segment based prefetching modes and provided analytical models in [24].

In this study, the following modes are implemented and experimentally evaluated:

- *OnDemand*: In this mode, no prefetching is implemented. The succeeding segment is fetched when it is needed by the streaming engine. This mode is simple and works fine when the available bandwidth of HTTP channel is large enough. Otherwise, streaming can be interrupted due to the delay in fetching the next segment from the server. Some of these effects can be partially hidden by providing buffering in the streaming engine.
- *Window*: In this mode, the sub-request for the next uncached segment is always issued when the client starts to access the current one. Thus it provides aggressive prefetching with a look-ahead window size of one segment.
- *Half*: Intuitively, the window size is adjustable. We also implemented a *Half* mode, in which the sub-request for the next uncached segment is issued after the server has reached the middle of the current one. Thus, in this mode, the window size is half a segment length.
- *Active*: Active prefetching is implemented to dynamically decide when to prefetch an uncached segment according to the real-time bandwidths. It is the most precise online prefetching technique according to [24] and is implemented with the aid of

Packet CAPture (PCAP) library [2]. With the API provided by PCAP, we periodically estimate the available network bandwidth between the Hyper-Proxy and the content-server. The prefetch schedule is then computed using the media encoding rate extracted from the header of the media file, which corresponds to the desired data transmission rate between Hyper-Proxy and the Client.

### 3.3.3 Segmentation-Enabled Cache Engine

The segmentation-enabled cache engine handles the sub-requests from the LCMS. In a case of a cache MISS, the cache engine gets the data for the sub-request from the content-server (or other peering proxies). The cache stores data  $D_s^n$  (data for segment  $n$ ) on the disk, as well as constructing and sending a reply with meta data  $M_s^n$  only to the LCMS. The meta data includes the name and the location of the file containing the data for this sub-request on the local disk. In a case of a cache HIT, the cache directly constructs and sends the  $M_s^n$  meta-data to the LCMS.

Currently, Hyper-Proxy uses a modified version of Squid2.3 (STABLE4) as the cache engine. Segmentation support is provided through the **Range** header in HTTP requests. Squid identifies objects in its cache using the MD5 hash of the request URL. Hence, in the original version of Squid, different ranges of a URL would have the same MD5 keys, and HTTP requests that include the **Range** header would be considered non-cachable. To make these requests cachable, our segmentation-enabled version re-writes the URL internally. For example, a request for:

`http://www.foo.com/bar.mp4`

with

`Range=123-890`

can be re-written as:

`http://www.foo.com/bar.mp4_123_890.`

This guarantees that different ranges of the same object generate different MD5 keys. This mechanism enables the caching of different segments of a media object.

The re-written URL is used internally in the proxy to identify different range requests for the same object. If the corresponding segment is not cached, the request is forwarded to the content-server (or peering proxies). However, the content-server does not contain an object named as `bar.mp4_123_890`, but only `bar.mp4`. Thus, the request restoring is necessary here. Since the restored request is an HTTP request, the content-server can be a standard server, such as `httpd` [78]. Note we use special methods in the proxy for streaming segment fetching as described in section 3.3.4. Thus, the method with this URL will be replaced to standard `GET` (Note if the range request is forwarded to peers, the method does not need the change.). Thus, the request will be re-written as follows:

`GET www.foo.com/bar.mp4 HTTP/1.0`

`range=123-890`

and sent to the content-server.

Since the re-writing of the URL provides the opportunity to cache the data for different segments of the same object, segment caching is enforced by saving the partial data on disk without violating the HTTP protocol. In the implementation, an HTTP reply status of `PARTIAL_CONTENT` (206) indicates the reply corresponds to a range request. If the reply is checked to match the range request for a segment, it is stored locally, and an additional status (`HTTP_PARTIAL_CONTENT_OK`) is added to the reply to the client. This status can only



be used by the methods (PREFETCH, LOCATEFILE, and LOCK) designed for streaming as to be stated in the next section.

Popularity based replacement policy has been found to be the most efficient for the multimedia object caching. Hyper-Proxy leverages the existing popularity based replacement policy in Squid.

Additionally, cooperative proxies have been used for caching static Web objects. It is even more desirable for caching large streaming media objects. Hyper-Proxy also leverages the existing cooperative functions in Squid. When requesting segments from neighboring caches, the internally re-written URL is restored to the original version, with the **Range** header added. This allows Hyper-Proxy to interact with regular Web-proxies without streaming capability, as well as other streaming-enabled Squid proxies.

### 3.3.4 Fast Data Path

In the early days, storage systems for media systems have been studied in [10, 107, 115]. In our system, the shared local file-system provides a fast data path between segmentation-enabled cache engine and the streaming engine. Traditionally, Squid transfers incoming data to an HTTP client over a network. For large media data files, it is more efficient to directly share the part of file system used as a data cache by Squid. In the Hyper-Proxy system, a set of new methods, PREFETCH, LOCATEFILE and LOCK, was added to Squid for this purpose:

1. PREFETCH is implemented as a non-blocking version of the HTTP GET method.

Whenever a segment is required, a request with a PREFETCH method and the corresponding **Range** header is sent to the proxy. The proxy checks if the requested segment

is cached or not. If it is cached, a HIT is returned. Otherwise, a MISS is returned and the corresponding request is re-written as a HTTP GET and forwarded to the content-server or the peer simultaneously. The proxy will store the reply containing the requested segment data on its local disk for future requests.

2. LOCATEFILE is implemented as a blocking method. The LCMS only invokes this method after a PREFETCH request returns a HIT. It returns the file location of the requested segment in the cache file structure maintained by Squid. It blocks until the entire data for a range request has been written to disk.
3. LOCK is used before the streaming engine starts to stream a segment to the client. Since the segment is cached and the cache is managed by Squid, the replacement policy in Squid automatically starts the replacement when the available cache space is below some threshold. It does not know whether or not the to-be-replaced segment is being used by the streaming engine. Thus, before reading the data of a segment for streaming, the LCMS issues a request with a LOCK method. This ensures that the to-be-read file will not be a candidate for eviction. After segment access is complete, the LOCK is released.

The non-blocking PREFETCH method and the blocking LOCATEFILE method effectively split the original, blocking GET method into a two-phase protocol. This is critical to the system performance when Hyper-Proxy needs to handle a large number of concurrent requests or when the segment size is large. Multiple PREFETCH methods for different segments can be issued without locking up the LCMS. The design of LOCK provides a tool to coordinate the streaming engine and cache engine.

### 3.4 Performance Evaluation

In this section, we describe the test setup and evaluation metrics that we use in experiments. We then present detailed experimental results, including a full caching approach to provide baseline values. Four case studies of Hyper-Proxy are presented then.

#### 3.4.1 Test Setup

We run tests in real network settings using actual implementation of the content-server, Hyper-Proxy, and media client. We use Apache Web Server (version 2.0.45 with HTTP 1.1) as the content-server. It is hosted on a HP Netserver lp1000r, with a 1 GHz Pentium III Linux PC platform. The Hyper-Proxy system runs on a HP workstation x4000 with two dual 2 GHz Pentium III Xeon Linux PC, with 1 GB memory. The media client used for the experiments is a dummy loader that logs incoming RTP and RTSP packets.

For all tests, the network connection between Hyper-Proxy and the client machine is a switched 100 Mbps Ethernet. For network conditions to the content-server, three settings are used, namely *local*, *remote* and *controlled* environments. The local environment is set up with both the content-server and the Hyper-Proxy system connected via a switched 100 Mbps Ethernet within HP Labs (Palo Alto, CA USA). The remote environment is constructed with the Hyper-Proxy system and content-server at trans-Pacific sites, where Hyper-Proxy is in HP Labs while the content-server is located in Takaido, Japan. The bottleneck bandwidth for the transoceanic link is approximately 10 Mbps. To study the effectiveness of the four prefetching methods in different network settings, we also construct a controlled environment in which the link capacity between the proxy and the content-

server can vary. We use traffic control support in the Linux kernel via the *tc(8)* [3] utility to establish bottleneck bandwidths.

### 3.4.2 Evaluation Metrics

We evaluate two end-to-end statistics that are especially relevant to streaming media delivery, client perceived startup latency and client perceived jitter. Startup latency is measured as the interval between the client sending a request for a media stream (the RTSP DESCRIBE method), and the arrival of the first media packet. The client jitter is the average of the values in the Receiver Report RTCP messages, calculated based on the algorithm in Appendix A.8 of RFC 1889 [101]. It represents the statistical variance of the media packet inter-arrival time.

To better evaluate different prefetching methods, we also record two statistics specific to Hyper-Proxy. We instrument the proxy system to measure time spent in handling each segment request. It is measured as the interval between the time when a segment is requested and the time the location of the segment in the cache storage is returned to the proxy. Note that every request for a segment results in a Squid handshake (to check whether it is in cache), while an uncached segment causes an HTTP transfer from the content-server. This measurement reveals whether the proxy can fetch segments in time in the middle of streaming sessions.

### 3.4.3 Experimental Results

We first perform experiments using a full object caching approach. The results provide a basis for comparisons with our segment-based approach. Further, to study the performance

of our Hyper-Proxy, we conduct experiments in four different aspects. We first consider the effects of using different segment sizes in Hyper-Proxy, when the number of concurrent clients increases. Then we evaluate the performance difference when the content-server sits at different network distances from Hyper-Proxy. We further evaluate the effectiveness of each prefetching method under different Hyper-Proxy – content-server link bandwidth capacities. For each of these experiments, the cache size is set large enough to store all the fetched content, and each client accesses a unique object. Moreover, the clients play clips in their entirety. Since there is no segment re-use across clients, this represents the worst case behavior for a cache engine. We finally validate our results with a trace driven experiment, using real enterprise access patterns. These traces include multiple clients accessing the same clip, and clients that do early termination.

### 3.4.3.1 Full Caching Approach

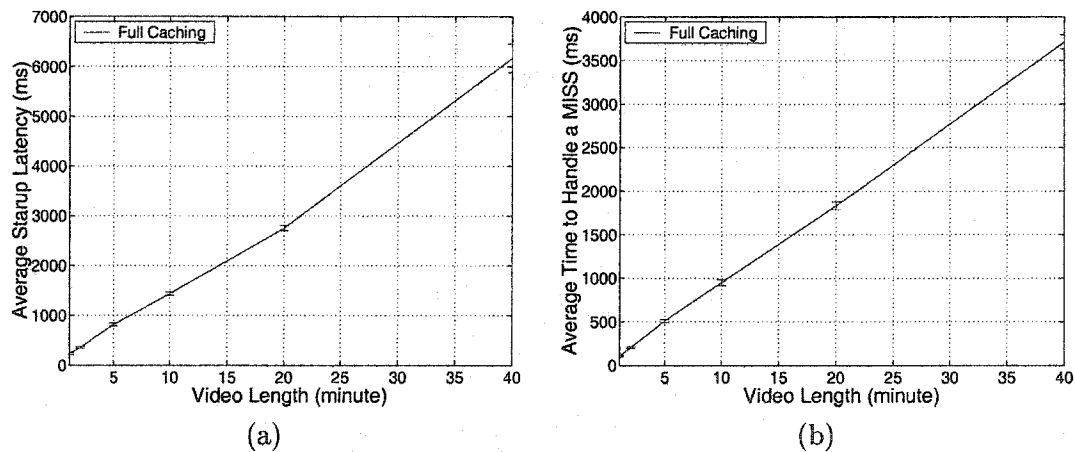
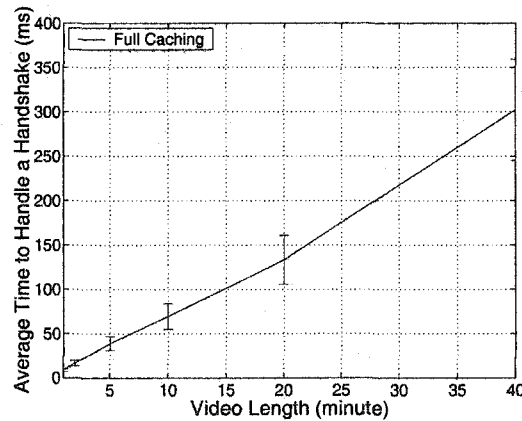


Figure 3.4: Performance of full caching approach: startup latency and miss processing

In the full caching approach, media objects are not segmented, but fetched in their

entirety. In this experiment the client, Hyper-Proxy and content- server are located in the same local network. Experiments are performed on different video clips of length 1, 2, 5, 10, 20, 40 minutes, encoded at 112 Kbps. The cache size is set large enough so that there are no capacity misses, hence no replacement is necessary. The results are averaged over 10 runs.

Figure 3.4(a) shows that the startup latency perceived by the client, as expected, increases linearly with the video size. Similar trends are reflected in performance in terms of the average time to handle a miss as shown on Figure 3.4(b).



**Figure 3.5:** Performance of full caching approach: handshake

Figure 3.5 shows that the handshake time in the proxy for the full caching approach also increases linearly with the video size. Note it is substantially smaller than the corresponding miss process time. In the full proxy caching approach, each media object is only fetched once from the content-server at the beginning, after that all requests hit in the proxy, thus the client perceived jitter is very negligible in this situation.

### 3.4.3.2 Effect of Segment Size

In the previous section, where each object is accessed and stored in entirety, we have learned that the performance of the full caching approach depends on the media object length and degrades almost linearly with the increase of the object length. In Hyper-Proxy, objects are segmented and managed as smaller units. The next experiment tests the effect of segment size on the Hyper-Proxy performance.

In this set of experiments, increasing number of concurrent clients request unique media objects. The media objects are all copies of the same piece of content (the 2 minute video clip) with different names. This effectively disables the file buffer cache in the Operating System. Moreover, the media data served to each client is identical, which allows us to present the data as averages across each client session. The client request inter-arrival interval is 1 second. The Squid cache file system is re-initialized before each experiment. We evaluate the performance by running tests with Hyper-Proxy using different segment sizes for segment-based caching. These experiments are carried out in the local environment using the *OnDemand* mode. Thus, there is no explicit prefetching, so we can isolate the effect of the segment size.

Figure 3.6 (a) shows the client perceived startup latency when the segment size varies from 100 KB to 500 KB and when the segment size is large enough to include the entire object. Clearly, the startup latency increases when the base segment size increases since Hyper-Proxy waits until the first segment is fetched from the content-server before starting streaming to the client. It is also expected that the startup latency increases when the number of concurrent clients increases, since this puts a load on the streaming server.

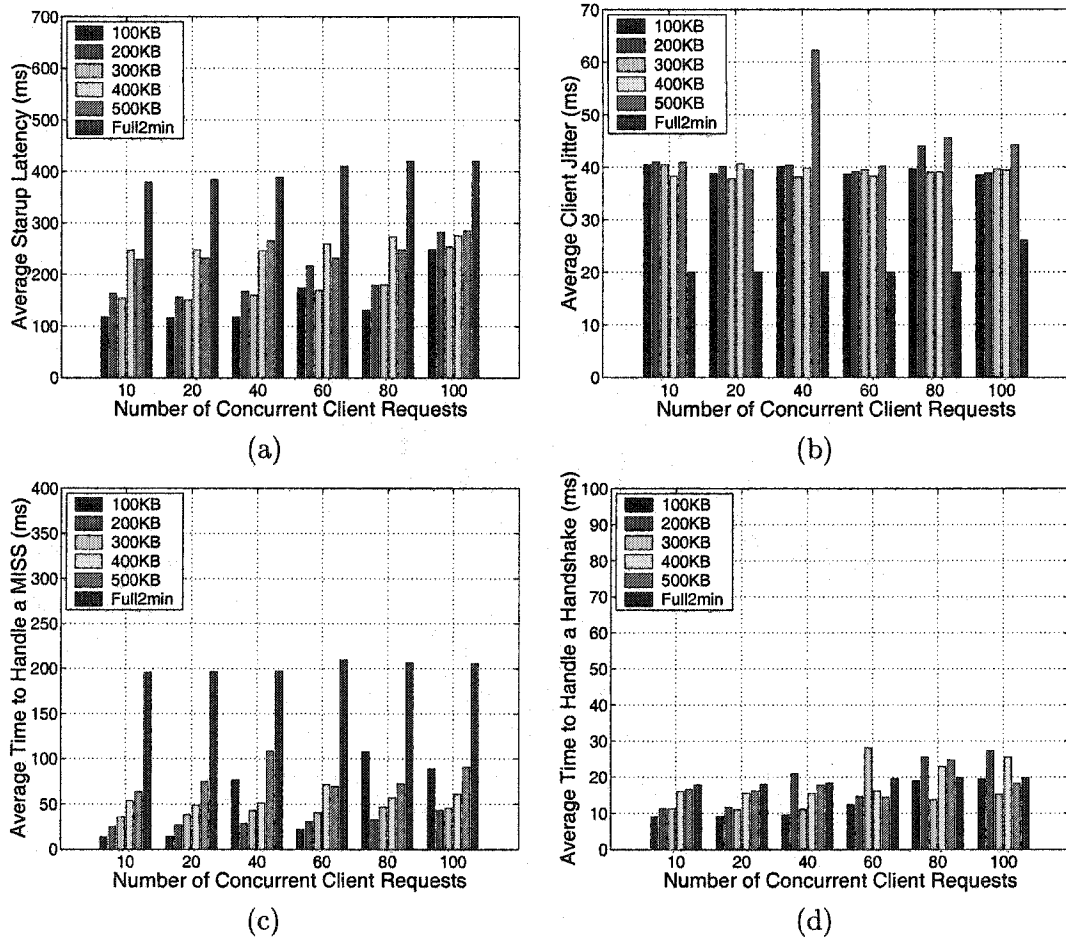


Figure 3.6: Performance study with different segment size

Compared to the startup latency when the entire object is fetched as one segment, the startup latency in Hyper-Proxy is only about 30% to 60%. It is also found that when segment size increases beyond 300 KB, the client perceived startup latency increases faster, while the effect is less pronounced when the segment size varies in the range of 100 KB to 300 KB. The startup latency is proportionally larger for clients in the remote environment.

Figure 3.6 (b) shows the client perceived jitter. It is obvious that jitter is the smallest when segment size is large enough to include the entire object. Otherwise, additional jitter



may be perceived due to the on-demand nature of segment based fetching by Hyper-Proxy. We show in Figure 3.6 (c) and (d) the average time consumed for the proxy to handle a MISS and a Squid handshake, respectively. It is clear that the average consumed time to handle both a MISS and a handshake increases with the segment size. Comparing Figure 3.6 (a) and (c), we note that the client perceived startup latency is usually larger than the time to handle a cache MISS. This is because the startup latency includes the time to setup the streaming session in addition to the time of fetching the first segment.

This set of experiments shows that Hyper-Proxy outperforms the full caching approach in terms of the client perceived startup latency and the average time to handle a miss, while it causes comparable amount of playback jitter even without prefetching support.

### 3.4.3.3 Effect of Proximity

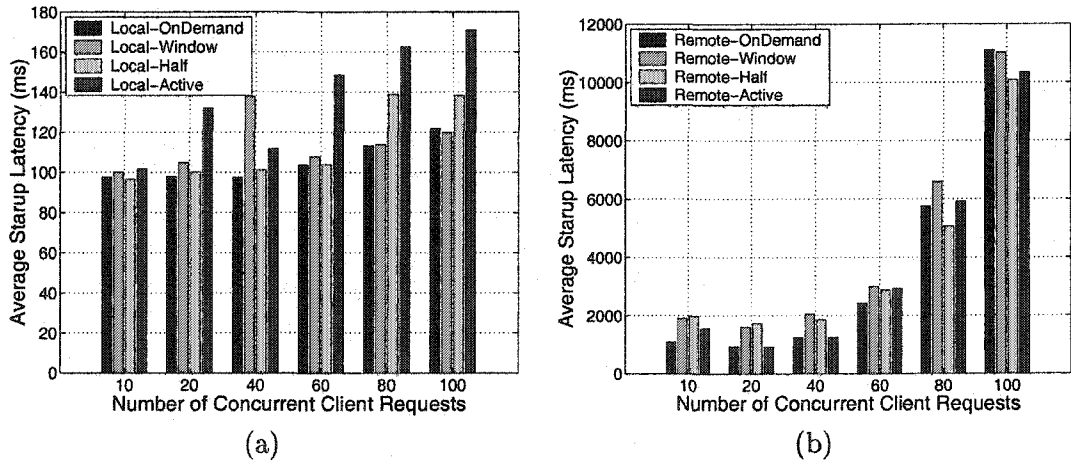


Figure 3.7: Client startup latency for local and remote

Another factor that affects the scalability of Hyper-Proxy is the proximity: the distance between the content-server and Hyper-Proxy. We evaluate the performance by running

tests with the content-server located in the local environment as well as in the remote environment. Segment size of 100 KB is used for this set of experiments. For fairness we run the *Local-OnDemand* again with others. Its results slightly differ from those in the previous subsection.

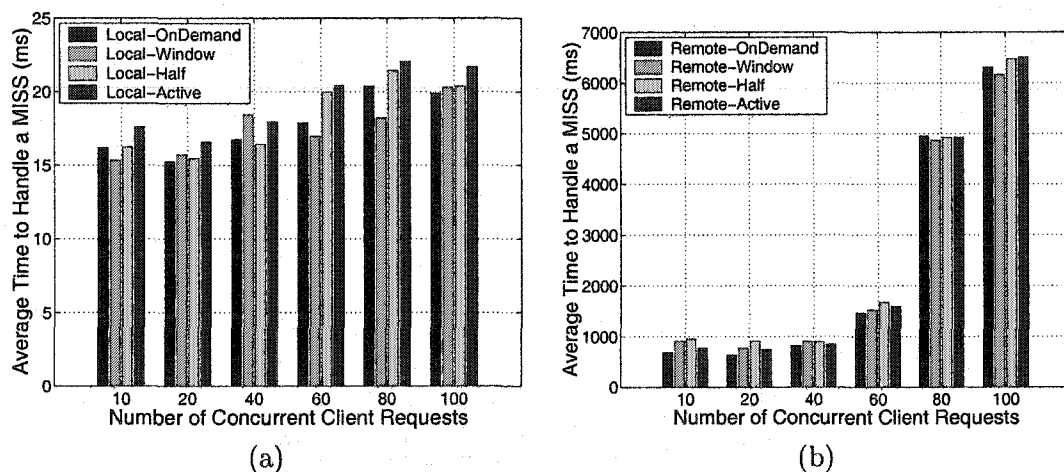


Figure 3.8: Time to handle a MISS for local and remote

Figure 3.7 (a) shows the startup latency for local accesses, while Figure 3.7 (b) shows this metric for remote accesses. In the local case, it varies from 96 ms to 169 ms, while for remote accesses, the startup latency is much larger, with a much bigger dynamic range, from 2 s to 11 s. The startup latency in both environments shows only a small variation across different prefetching methods. This is an intuitive result, since the value would be dominated by the access time for the *first* segment accessed. It is also seen that the startup latency generally increases when there are more concurrent requests. The results indicate that more concurrent requests can be served in local networks, and that more concurrent requests can lead to a longer startup latency in wide area networks. This figure also shows that our design and implementation of Hyper-Proxy can support the delivery of media

objects with reasonable startup latency in both intranet and Internet environments.

Another important aspect of streaming media delivery is whether the proxy can provide rigorous continuous streaming. Figures 3.8 (a) and (b) show the average time that the proxy consumes to process a cache MISS in each environment. The average time to handle a MISS in the local testing environment is less than 23 ms. In a wide area network, the average consumed time can reach 6.5 seconds. The results justify that prefetching for content from remote content servers is necessary, since such a large delay may potentially cause playback jitter at the client side.

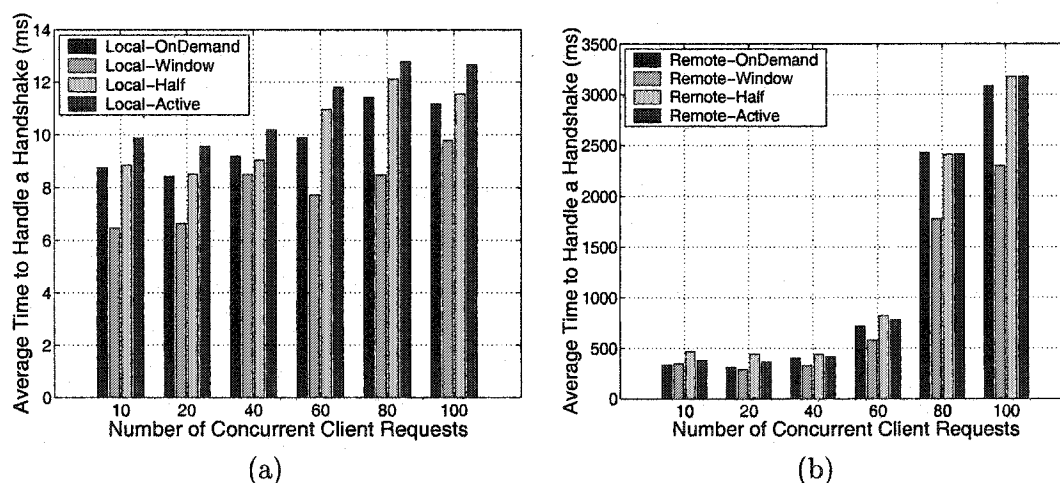


Figure 3.9: Time to handle a handshake for local and remote

Figures 3.9 (a) and (b) show the average time for the proxy to handle a Squid handshake, whether a HIT or a MISS, for the local and remote environment, respectively. A good prefetching method will have a higher percentage of HIT cases, which leads to a correspondingly smaller average time. Comparing Figure 3.9 with Figure 3.8, we can see that a handshake consumes much less time than a MISS on average. Note that *OnDemand* does not do any prefetching. It shows some HIT cases, since the file format parser might request

the same segment multiple times, e.g., first for parsing the hint track, and then for reading media data. As shown in the remote case of Figure 3.9, *OnDemand* always consumes more Squid handshake time and other prefetching methods reduce the Squid handshake time somewhat. It seems that a simple *Window* mode performs the best in this set of tests. We have shown in [25] that *Active* should perform best if an accurate real-time measurement of the proxy-server link bandwidth is in place. Our current implementation of *Active* may have been limited by PCAP's capability.

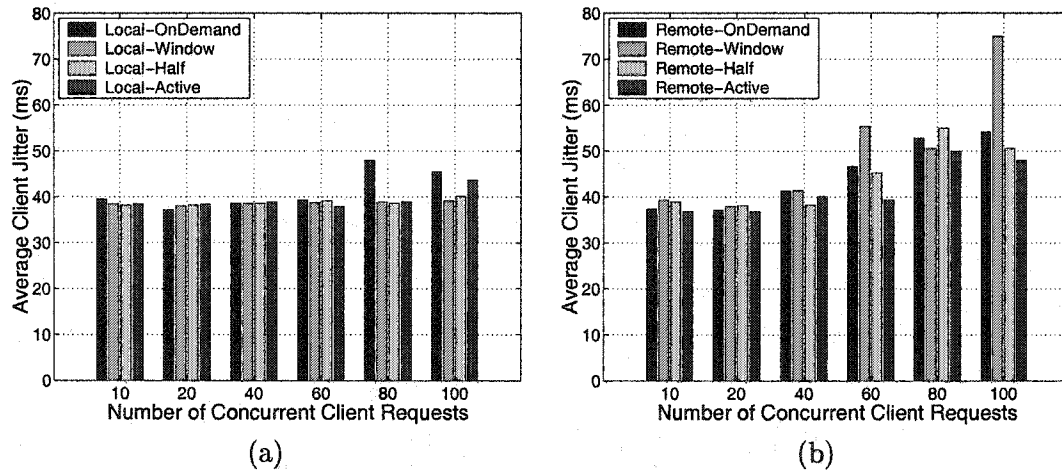


Figure 3.10: Client perceived jitter for local and remote

Figures 3.10 (a) and (b) show the client perceived jitter in both local and remote environments. In both cases, the absolute client perceived jitter is small, which indicates that our Hyper-Proxy can successfully serve a large number of clients with rigorous continuous streaming demand. Note that the client jitter tends to increase when more concurrent requests are served, especially in the remote environment. This indicates that accurate prefetching is very important especially when the Hyper-Proxy – content-server link bandwidth resource becomes scarce. *Active* prefetching achieves better performance as shown in

the remote case.

#### 3.4.3.4 Prefetching Effectiveness

The preceding experiments have evaluated the system performance in local and remote network settings. To further study the effectiveness of the different prefetching methods in different network settings, we test the system in a controlled environment, as described in Section 3.4.1. For each bandwidth setting, a video clip with an encoding rate of 75 Kbps is served from the content-server to the client through Hyper-Proxy. We collect the Squid handshake time and client jitter statistics for each prefetching method. Note that all prefetching methods except *Active* schedule prefetching regardless of the link bandwidth (in this test, the bottleneck link bandwidth).

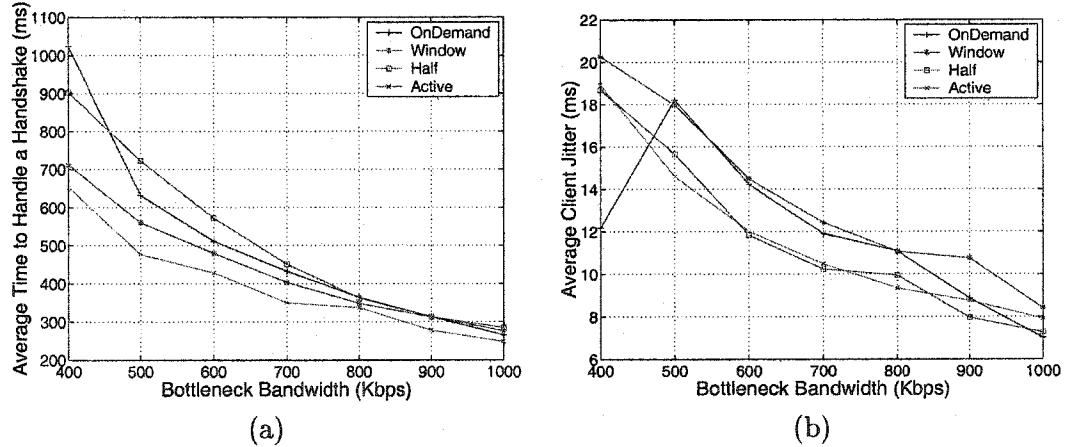


Figure 3.11: Squid handshake time and client perceived jitter

As shown in Figure 3.11 (a) and (b), both the Squid handshake time and the client perceived jitter decrease when the bandwidth increases. Note that the Squid handshake time here is generally much longer than the result in the proximity study since the bottleneck

link bandwidth is much smaller. The *Active* prefetching can be seen to have the shortest Squid handshake time, especially when the link bandwidth is low. The differences in client perceived jitter are less obvious although *Active* and *Window* methods perform better with a low bottleneck link bandwidth.

### 3.4.3.5 Cache Efficiency Study Using a Real Workload

Even though we use segmentation-enabled Squid as the cache engine in our Hyper-Proxy, it is important to evaluate its performance on cache efficiency in conjunction with prefetching methods. For this purpose, we use a trace extracted from real enterprise media server logs to drive a 24-hour run of the actual system in the local environment. These traces include clients that access the same clip, and clients that terminate a clip prematurely, or start playing a clip from the middle. Thus we would expect better caching behavior, but also wasted bandwidth consumed due to segments that are pre-fetched and never used.

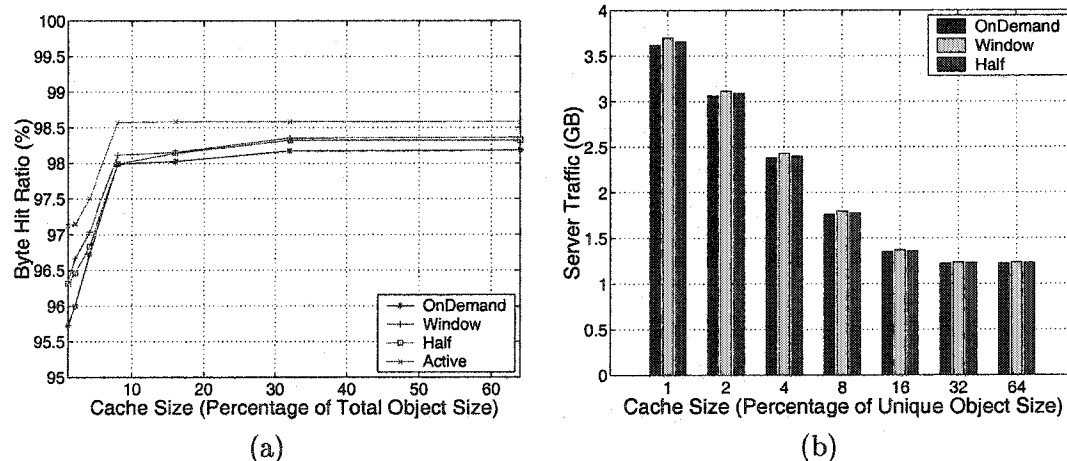
The trace contains 16,238 requests with the access duration varies from 1 to 50 minutes. In these 16,238 requests, 92% of them are demanding the same video clip, most of which are with premature terminations. Thus, the caching performance is expected to be very high. We select such a workload because we also want to test whether the system can survive a large number of concurrent requests in a long time period. There are a total of 70.775 GB data accessed. The unique object size amounts to 5.358 GB. Based on the file length and streaming rate of the objects requested, we have created matching video clips in MP4 file format. Specifically, a content pool is created as follows using the parameters as shown in Table 3.1. As shown on the table, video objects are created with 6 bitrate (28 - 256 Kbps) versions, each with a maximum of 7 possible file length (1 - 100 minute). Each created

object is replicated multiple times so that they can be accessed as unique objects.

**Table 3.1:** The content and access parameters of real workload

Rate (Kbps)	File length (minute)	Max access duration (minute)
28	1, 10, 20, 50	1
56	50	12
112	1, 2, 5, 10, 20, 50	14
156	1, 20, 50	14
180	2, 5, 10, 20, 50, 100	50
256	1, 2, 5, 10, 20, 50, 100	25

Based on this real enterprise media access trace, we first run the trace through the Hyper-Proxy system to evaluate the caching performance in terms of the byte hit ratio. Then we use a cache simulator to evaluate these segment-based caching strategies with various prefetching methods and different segment sizes. We use simulator since some metrics, such as false prefetch, is very difficult to measure in the real runs.



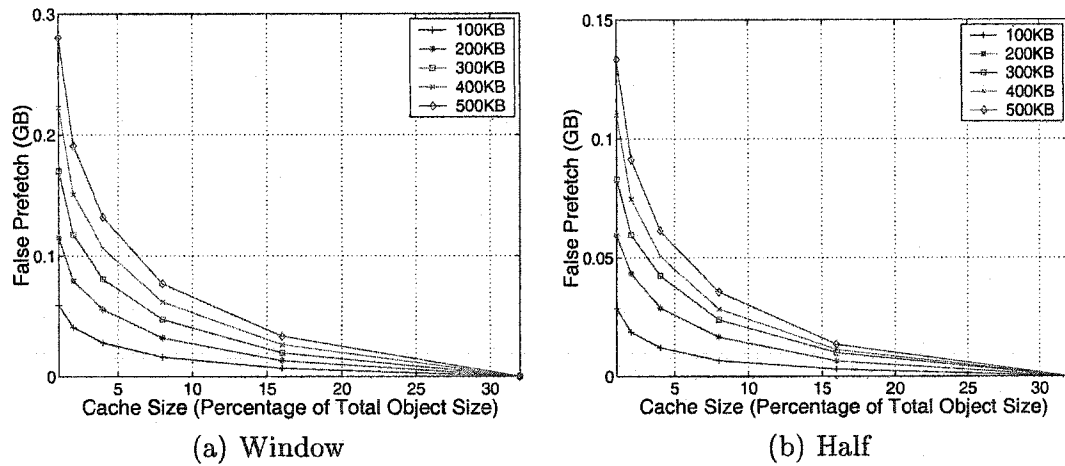
**Figure 3.12:** Byte hit ratio and server traffic for segment-based caching strategies

Figure 3.12 (a) shows the byte hit ratio achieved by Hyper-Proxy with different prefetching methods with the increase of the cache size when the segment size is set to 100 KB. It

is expected that *Active* achieves the best byte hit ratio since active prefetching can always prefetch uncached segments in time. The byte hit ratio is very high even with a small cache space. This is due to the high access locality in this 24 hour trace as we have mentioned. In addition, some segments, such as segments containing hint track information, are repeatedly accessed in segment-based caching approaches to increase the byte hit ratio. Except for the repeatedly accessed segments, some prefetched segments may not actually be used by the clients due to early terminations, thus the byte hit ratio cannot precisely reflect the server traffic reduction. Figure 3.12 (b) shows the corresponding traffic reduction based on simulations for segment-based caching with different prefetching methods when the segment size is of 100 KB. Figure 3.12 (b) shows that *OnDemand* generates the least server traffic since it does not do any prefetch; *Half* and *Window* methods, with increased aggressiveness in prefetching, generate more and more server traffic. This is a small penalty Hyper-Proxy pays to improve continuous streaming of media content. *Active* prefetching is not simulated here since it is difficult to simulate a dynamic estimation of the channel bandwidth between content-server and Hyper-Proxy. The server traffic generated by *Active* prefetching would depend on the time-varying nature of the channel bandwidth.

However, not all prefetched segments will be used by the clients. We define false prefetch as the size of the segments that are prefetched and cached but have never been streamed to clients before they are evicted. Figure 3.13 shows for this 24 hour trace, *Half* method produces about 50% of the false prefetches compared to *Window*. Thus, for real traces, the *Window* method is too aggressive, since many clients terminate clips early (i.e., before accessing half of a segment). Also, since the prefetching granularity is segment, smaller segment size produces less false prefetch. We study these in detail in [98].



Figure 3.13: False prefetch by *Window* and *Half*

### 3.5 Summary

Recent years have seen a large amount of research work in segment-based proxy caching for streaming media delivery. However, its implementation and deployment are hindered by several factors. One is the complexity of media file formats. The other concern comes from UDP, which is the base for RTP, the base for real streaming. Additionally, system support is demanded to guarantee continuous streaming. We have designed, implemented, and tested that it is possible to push the streaming capability to the edge of the network and couple it with a caching proxy to efficiently serve a large number of clients. This design fundamentally frees the content provider from serving constraints. Specifically, our contributions are:

- i. We have designed and implemented a segment-based caching proxy that supports streaming of multimedia content with rigorous latency and continuity constraints.
- ii. The design and implementation leverage the existing Internet infrastructure. The

content-server needs only to be a simple Web server, yet its contents are served through Hyper-Proxy in a scalable and efficient fashion.

iii. We have thoroughly evaluated different prefetching methods which are closely coupled with the segment-based caching. We have shown that segment-based access inherently reduces the client perceived startup latency and various prefetching methods can provide continuous streaming in various network conditions.

iv. We have tested the full system with real network conditions and with a real workload.

We believe this is the first work of this kind.

Currently, the Hyper-Proxy system is deployed at a site for large scale deployment tests and at many sites of a large enterprise for one-year trial stage.

## Chapter 4

# Shared Running Buffers (SRB) Based Proxy Caching Streaming Sessions

### 4.1 Introduction

Currently the basic infrastructure of the Internet content delivery network is a server-proxy-client system. In this system, the server delivers the content to the client through a proxy. The proxy can choose to cache the object so that subsequent requests to the same object can be served directly from the proxy without contacting the server. But the high performance and high quality delivery demand of the vast amount of streaming media contents presents several challenges to this infrastructure since the existing proxy cannot efficiently delivery streaming media data due to two facts. On one hand, the size of a media object is normally much larger than that of a Web object, making it infeasible to cache the entirety of the media object. On the other hand, the delivery of streaming media objects is constrained by the real-time requirements from clients as stated in previous chapters. Without the

proxy caching assisting the streaming media delivery, a lot of resources have to be reserved for delivering the streaming media data to a client. In practice, even a relatively small number of clients can overload a media server, creating bottlenecks by demanding high disk bandwidth on the server and high network bandwidth to the clients.

To address these challenges, researchers have proposed different methods to utilize the available resources in the proxy for caching streaming media objects. The caching approaches [32, 43, 51, 65, 74, 81, 94, 104, 117] always utilize the storage (mainly the disk) in the proxy to cache either a prefix or a certain number of segments of a media object.

More recently, researchers have paid attention to the temporal caching of media objects in the proxy memory. With the falling prices of memories, the magnitude of Gbytes memory of a server/proxy is not uncommon. So to temporarily cache the media data in the server/proxy memory while delivering to the client so that the later requests could benefit from the cached data in the memory is feasible. Similar to the proxy caching based on disk, to cache the entire media object in the memory is unrealistic, while randomly caching a fraction of the media objects statically is not useful because normally the media objects are requested sequentially. So when and how to cache the media objects, when and how to allocate/reclaim the memory buffers so that the media data can be delivered to as much as possible from the limited memory space to as many clients as possible become an interesting and challenging problem. There was some work discussing this problem. The fixed-sized running buffer caching [13] and the interval caching [36, 37] are two major running (memory) buffer based caching techniques. The basic working principle of the running buffer based caching technique is as follows: when a request arrives, a fixed-sized buffer of a predetermined length is allocated to cache the media data fetched by the proxy, hoping

that closely followed requests could reuse the data in the memory instead of obtaining it from other sources (the disk, the origin server or other caches). In contrast, the interval caching technique uses a different approach. It considers two requests immediately followed as a request pair, and incrementally orders the arrival intervals of all request pairs (the arrival interval of a request pair is defined as the difference in their arrival times). In the memory allocation, the interval caching gives preference to smaller intervals, expecting more requests can be served for a given amount of memory. Figure 4.1 illustrates basic ideas of the running buffer caching and the interval caching techniques.

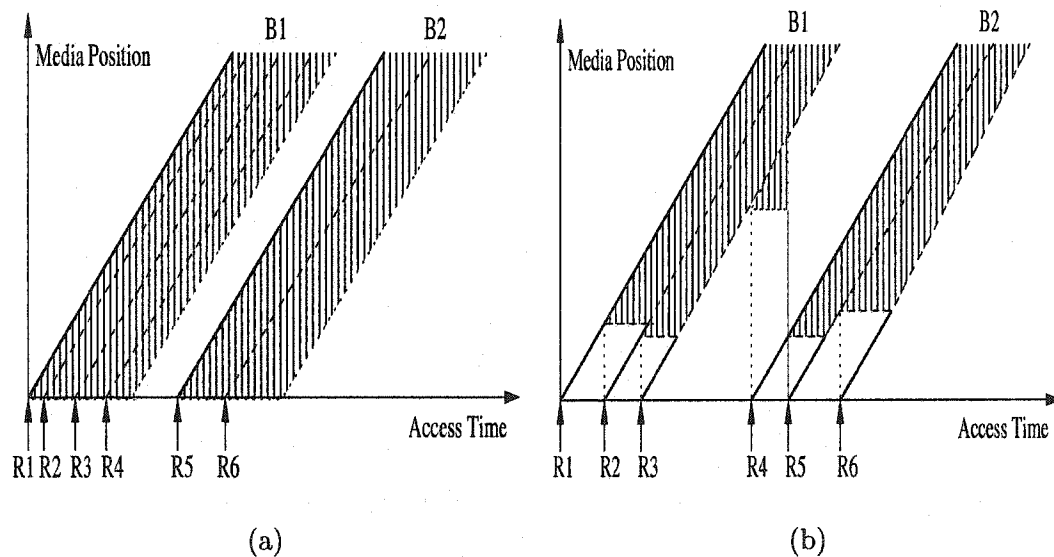


Figure 4.1: Running buffer and interval caching

In Figure 4.1, the *Media Position* indicates a time position in a streaming media where the media object is delivered to the client. The solid slope represents a delivery session. In Figure 4.1 (a), a fixed-sized buffer  $B1$  is allocated upon the arrival of the request  $R1$ . Subsequently, requests  $R2$ ,  $R3$ ,  $R4$  are served by the data cached in this buffer since they arrive in time. The request  $R5$  does not arrive in time, so a new buffer  $B2$  of the same

length is allocated, which benefits the request  $R_6$ .

In Figure 4.1 (b), upon the arrival of  $R_2$ , an interval is formed between  $R_1$  and  $R_2$ , and a buffer of the interval size is allocated to cache data read by  $R_1$  from now on. So the request  $R_2$  only needs to read the first part of data from the proxy/server while receiving the rest data from the buffer. The situation changes until the arrival of request  $R_5$ , where we assume the interval between  $R_4$  and  $R_5$  is smaller than that between  $R_3$  and  $R_4$ . Since the interval between  $R_4$  and  $R_5$  is smaller than that between  $R_3$  and  $R_4$ , the buffer allocated for the interval between  $R_3$  and  $R_4$  is deallocated, and the space is re-allocated to the new interval between  $R_4$  and  $R_5$ .

However, the running buffer caching does not take consideration of user access patterns, resulting in the inefficient usage of the memory resource. For example, in Figure 4.1 (a), the size of buffer  $B_1$  is larger than the amount needed to serve the requests of  $R_1$  through  $R_4$ , the size of buffer  $B_2$  is larger than the amount needed to serve the request  $R_5$  and the request  $R_6$ . The interval caching approach does consider the client access pattern in the buffer allocation. However, it shares another limit with the running buffer caching approach: data sharing among different buffers has not been considered. For example,  $B_2$  in Figure 4.1 (b) does not need to run to the end of the media if the data cached in buffer  $B_1$  are shared by the later requests.

In this chapter, we propose a new memory-based caching algorithm for streaming media objects using *Shared Running Buffers (SRB)*. In this algorithm, the memory space on the proxy is allocated adaptively by considering the client access patterns and the requested media objects themselves. More importantly, the data cached in different running buffers are fully shared. When requests are terminated, the algorithm efficiently reclaims the idle

memory space and does near-optimal buffer replacement at runtime.

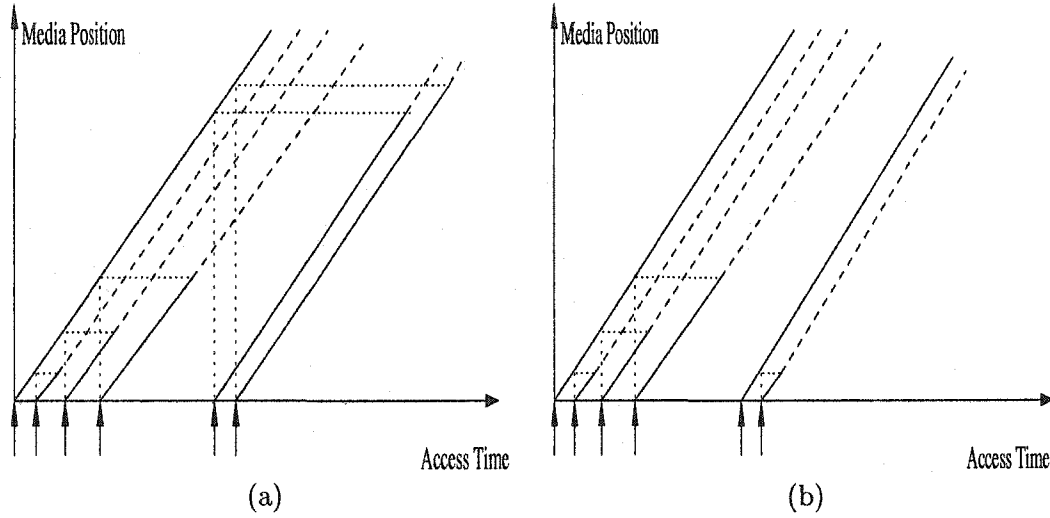


Figure 4.2: Greedy patching and grace patching

By further looking into the patching approaches that were heavily studied in the VOD environment [38, 48], we found that patching algorithms, such as the greedy patching, the grace patching and the optimal patching [60], take advantage of the client-side storage resource to buffer data in multiple channels without delay. The greedy patching as shown on Figure 4.2 (a) always patches to the existing complete stream while the grace patching restarts a new complete stream at some appropriate points as shown on Figure 4.2 (b). Furthermore, the optimal patching [103] considers how to re-use the limited storage on the client side to receive as many data as possible while listening to as many channels as possible. Figure 4.3 shows the basic idea of the optimal patching. Motivated by this, we propose another enhanced media delivering algorithm: Patching SRB (PSRB), which further improves the performance of the media data delivery.

Finally, we evaluated our algorithms through an intense set of simulations based on both

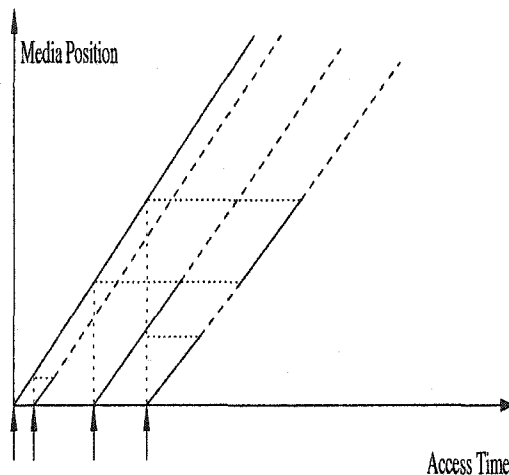


Figure 4.3: Optimal patching

synthetic workloads and a real access trace of an enterprise media server. The simulation results show that the performance of our algorithms is superior comparing with previous solutions. More details are presented in the [21].

The rest of this chapter is organized as follows. Some related work is introduced in Section 4.2. Section 4.3 describes the memory-based SRB algorithms we proposed. Simulation based performance evaluation results are presented in Section 4.4. We then summarize this study in Section 4.5.

## 4.2 Related Work

The delivery of a streaming media object takes time to complete. We call this delivery process a streaming session. Session sharing is possible among sessions that overlap with each other. In addition, proxy buffering can be used. Batching is a special type for session sharing [38, 47]. In this approach, requests are grouped and served simultaneously via



multicast [48]. Therefore, requests arrived earlier have to wait. Hence, some delays have to be introduced to the early arrived requests. Session sharing and proxy buffering for streaming media delivery have also been studied with other techniques, such as multicast, broadcasting, and proxy caching. In [111], the batching, patching and streaming merging are combined with proxy caching. Multicast [121] is considered with caching in [90] and [50]. Staggered multicast is proposed in [45]. A circular buffer is used in [20], while in [22], a set of existing techniques are evaluated and the running buffer is efficiently utilized together with patching for efficiently delivering the media content. A number of different broadcasting strategies have also been proposed to be used separately or together with other techniques, such as skyscraper broadcasting [62], pyramid broadcasting (PB) [110] and permutation-based pyramid broadcasting (PPB) [8], harmonic broadcasting [66] and its variant [86], variable bandwidth broadcasting (VBB) [88].

### 4.3 Shared Running Buffer (SRB) Media Caching Algorithms

It has been shown that two current memory caching approaches of the media objects: the running buffer caching and the interval caching approaches, do not take effective use of the limited memory resource.

Motivated by the limits of the current memory buffering approaches, we design a Shared Running Buffer (SRB) based caching algorithm, briefly called SRB algorithm, for the streaming media with the aim to maximize the effective utilization of the memory. In this section, with the introduction of several new concepts, we first describe our basic SRB media caching algorithm in detail. Then, we characterize the SRB media caching algorithm

and present an extension to the SRB: Patching SRB (PSRB).

#### 4.3.1 SRB Algorithm: Related Definitions

The algorithm first considers buffer allocation in a time span  $T$  starting from the first request. We denote  $R_i^j$  as the  $j$ -th request to media object  $i$ , and  $T_i^j$  as the arrival time of this request. Assume that there is  $n$  request arrivals within the time  $T$  and  $R_i^n$  is the last request arrived in  $T$ . For the convenience of representation without losing precision,  $T_i^1$  is normalized to 0 and  $T_i^j$  (where  $1 < j \leq n$ ) is a time relative to  $T_i^1$ . Based on the above, the following concepts are defined to capture the characteristics of the user request pattern.

1. *Interval Series*: An interval is defined as the difference in time between two consecutive request arrivals. We denote  $I_i^j$  as the  $j$ -th interval for object  $i$ . An *Interval Series* consists of a group of intervals. Within the time  $T$ , if  $n = 1$ , the interval  $I_i^1$  is defined as  $\infty$ ; otherwise, it is defined as:

$$I_i^j = \begin{cases} T_i^{j+1} - T_i^j, & \text{if } 1 \leq j < n \\ T - T_i^n, & \text{if } j = n. \end{cases} \quad (4.1)$$

Since  $I_i^n$  represents the time interval between the last request arrival and the end of the investigating time period, it is called as the *Waiting Time*.

2. *Average Request Arrival Interval (ARAI)*: The *ARAI* is defined as  $\sum_{k=1}^{n-1} I_i^k / (n - 1)$  when  $n > 1$ . *ARAI* does not exist when  $n = 1$  since it indicates only one request arrival within time frame  $T$  and thus we set it as  $\infty$ .

For the buffer management, three buffer states and three timing concepts are defined as follows, respectively.

1. *Construction State & Start-Time*: when an initial buffer is allocated upon the arrival of a request, the buffer is filled while the request is being served, expecting that the data cached in the buffer could serve the closely followed requests for the same object. The size of the buffer may be adjusted to cache less or more data before it is frozen. Before the freezing happens, the buffer is in the *Construction State*.

Thus, the *Start-Time* of a buffer  $B_i^j$ , the  $j$ -th buffer allocated for object  $i$ , is defined as the arrival time of the last request before the buffer is frozen. We use  $S_i^j$  to denote the *Start-Time* of the buffer  $B_i^j$ . The requests arriving in a buffer's *Construction State* are called as the *resident requests* of this buffer and the buffer is called as the *resident buffer* of these requests.

2. *Running State & Running-Distance*: after the buffer freezes its size it will serve as a running window of a streaming session and moves along with the streaming session. Therefore, the state of the buffer is called the *Running State*.

The *Running-Distance* of a buffer is defined as the distance in terms of time between a running buffer and its preceding running buffer. We use  $D_i^j$  to denote the *Running-Distance* of  $B_i^j$ . Note that for the first buffer allocated to an object  $i$ ,  $D_i^1$  is equal to the length of object  $i$ :  $L_i$ . Here, we assume a complete viewing scenario initially. Since we are encouraging the sharing among the buffers, the buffer  $B_i^j$  needs only to run to the end point of  $B_i^{j-1}$ . Mathematically, we have:

$$D_i^j = \begin{cases} L_i, & \text{if } j = 1 \\ S_i^j - S_i^{j-1}, & \text{if } j > 1. \end{cases} \quad (4.2)$$

3. *Idle State & End-Time*: when the running window reaches the end of the streaming

session, the buffer enters the *Idle State*, which is a transient state that allows the buffer to be reclaimed.

The *End-Time* of a buffer is defined as the time when a buffer enters *idle state* and is ready to be reclaimed. The *End-Time* of the buffer  $B_i^j$ , denoted as  $E_i^j$  is defined as:

$$E_i^j = \begin{cases} S_i^j + L_i, & \text{if } j = 1 \\ T_i^{\text{latest}} + D_i^j, & \text{if } j > 1. \end{cases} \quad (4.3)$$

$T_i^{\text{latest}}$  denotes the arrival time of the most recent request to the object  $i$ . Here, the  $T_i^{\text{latest}}$  dynamically changes with the coming of new requests and so does the  $E_i^j$ . The detailed updating procedure of  $E_i^j$  is described in the following section.

#### 4.3.2 SRB Algorithm

With these related definitions, the SRB algorithm works as follows: for an incoming request for the object  $i$ , the SRB algorithm works as follows:

1. If the latest running buffer of the object  $i$  is caching the prefix of the object  $i$ , the request will be served directly from all the existing running buffers of the objects.
2. Otherwise,
  - (a) if there is enough memory, a new running buffer of a predetermined size  $T$  is allocated. The request will be served from the new running buffer and all existing running buffers of the object  $i$ .
  - (b) if there is no enough memory, the SRB buffer replacement algorithm (see 4.3.2.3) is called to either re-allocate an existing running buffer to the request or serve this request without caching.

3. Update the *End-Times* of all existing buffers of the object  $i$  based on Equation 3.

During the process of the SRB algorithm, parts of a running buffer could be dynamically reclaimed as described in 4.3.2.2 due to the request termination and the buffer is dynamically managed based on the user access pattern through a lifecycle of three states as described in 4.3.2.1.

#### 4.3.2.1 SRB Buffer Lifecycle Management

Initially, a running buffer is allocated with a predetermined size of  $T$ . Starting from the *Construction State*, it then adjusts its size by going through a three-state lifecycle management process as described in the following.

- Case 1: the buffer is in the *Construction State*. The proxy makes a decision at the end of  $T$  as follows.
  - If  $ARAI = \infty$ , which indicates that there is only one request arrival so far, the initial buffer enters the *Idle State* (case 3) immediately. For this request, the proxy will serve as a bypass server, i.e., the content is passed to the client without caching. This scheme gives preference to more frequently requested objects in the memory allocation. Figure 4.4 (a) illustrates this situation. The shadow indicates the allocated initial buffer, which is reclaimed at  $T$ .
  - If  $I_n > ARAI$  ( $I_n$  is the waiting time), the initial buffer is shrunk to the extent that the most recent request can be served from the buffer. Subsequently, the buffer enters the *Running State* (case 2). This running buffer will serve as a shifting window and run to the end. Figure 4.4 (b) illustrates an example. Part

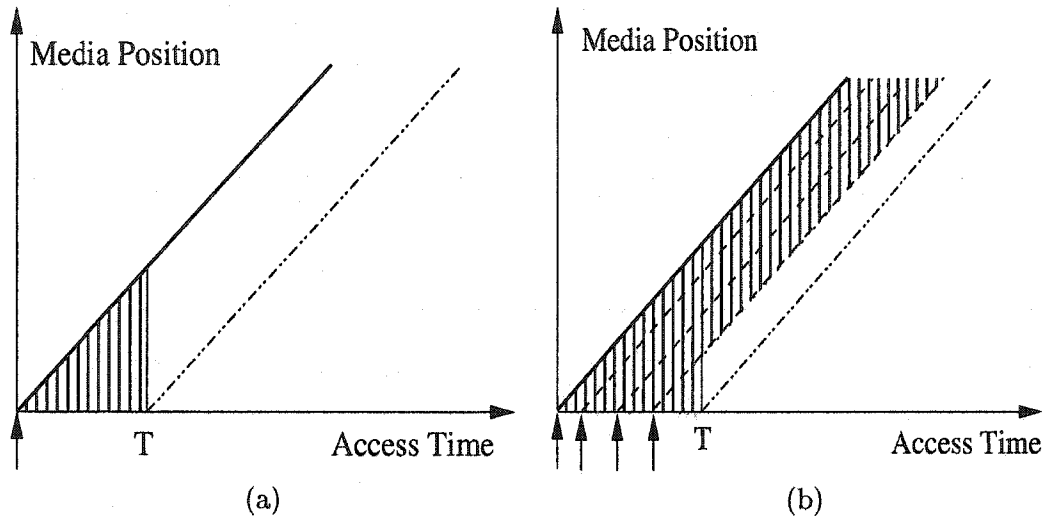


Figure 4.4: SRB memory allocation: the initial buffer freezes its size (1)

of the initial buffer is reclaimed at the end of  $T$ . This scheme performs well for periodically arrived request groups.

- If  $I_n \leq ARAI$ , the initial buffer maintains the construction state and continues to grow to the length of  $T'$ , where  $T' = T - I_n + ARAI$ , expecting that a new request arrives very soon. At  $T'$ , the  $ARAI'$  and  $I'_n$  are recalculated and the above procedure repeats. Eventually, when the request to the object becomes less frequent, the buffer will freeze its size and enter the *Running State* (case 2). In the extreme case, the full length of the media object is cached in the buffer. In this case, the buffer also freezes and enters the running state (a static running). For most frequently accessed objects, this scheme ensures that the requests to these objects are served from the proxy directly. Figure 4.5 (a) illustrates this situation. The initial buffer has been extended beyond the size of  $T$  for the first time.

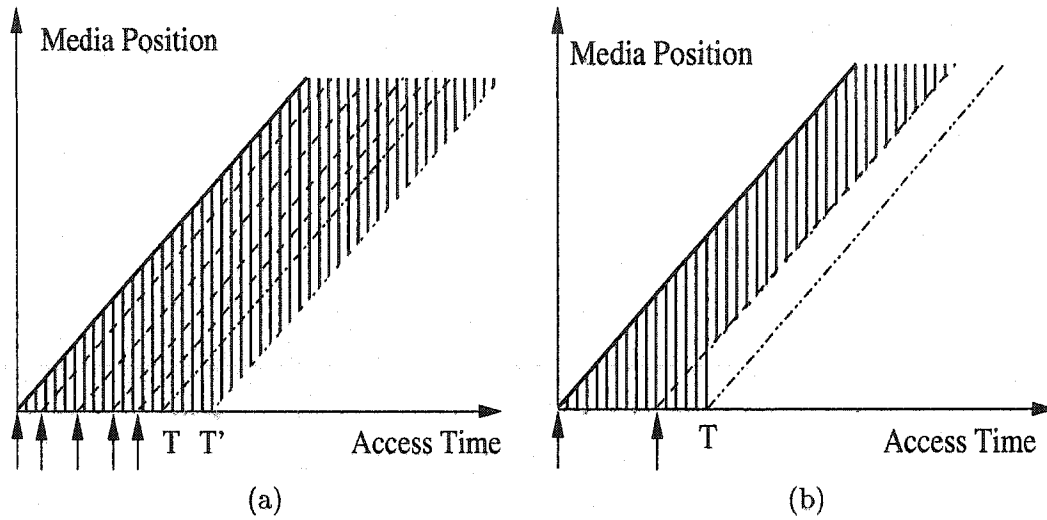


Figure 4.5: SRB memory allocation: the initial buffer freezes its size (2)

The buffer expansion is bounded by the available memory in the proxy. When the available memory is exhausted, the buffer freezes its size and enters the running state regardless of future request arrivals.

- If  $T > ARAI > T_h$  ( $T_h$  is a threshold specified by the client, say  $T/2$ ), the initial buffer is shrunk to the extent that the most recent request can be served from the buffer. Subsequently, the buffer enters the *Running State* (case 2). This running buffer will serve as a shifting window and run to the end. Figure 4.5 (b) illustrates an example. Part of the initial buffer is reclaimed at the end of  $T$ . This deals with scattered requests for an object. The idea of a threshold is not a must for our algorithm, but for the better resource utilization.

- Case 2: the buffer is in the *Running State*. After a buffer enters the running state, it has run away from the beginning of the media object and subsequently arrived requests can not be served completely from the running buffer. In this case, a new

buffer of an initial size  $T$  is allocated and subsequent requests are served from the new buffer as well as the first running buffer.

In addition, the *End-Time* of the new running buffer needs to be determined and the *End-Times* of its preceding running buffers  $E_i^{j-1}, \dots, E_i^2$  need to be modified according to the arrival time of the latest request, as shown in Equation 4.3.

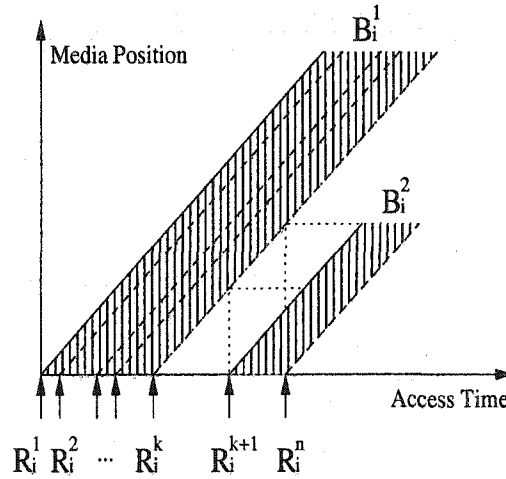


Figure 4.6: Data sharing among buffers in SRB algorithm

Figure 4.6 illustrates the maximal data sharing in the SRB algorithm. Here, since we consider the request and delivery of one object. The requests  $R_i^n$  to  $R_i^{k+1}$  are receiving data simultaneously from  $B_i^1$  and  $B_i^2$ . Late requests could receive data from all existing preceding running buffers. **Note that except for the first buffer, other buffers do not have to run to the end of the object.** When the buffer runs to its *End-Time*, it enters the *Idle State* (case 3).

- Case 3: the buffer is in the *Idle State*. When a buffer enters the *Idle State*, it is ready for reclamation.



In the above algorithm, the time span  $T$  (which is the initial buffer size) is determined based on the object length. Typically, a *Scale* factor (say,  $1/2$  to  $1/32$ ) of the origin length is used. To prevent an extremely large or small buffer size, the buffer size is bounded by an upper bound: *High-Bound* and a lower bound: *Low-Bound*. It can be adjusted by the streaming rate to allow the initial buffer to cache a reasonable length (e.g., 1 minute to 10 minutes) of media data.

The algorithm requires the client be able to listen to multiple channels at the same time: once a request is posted, it should be able to receive data from all the ongoing running buffers of that object simultaneously.

#### 4.3.2.2 SRB Buffer Dynamic Reclamation

The memory reclamation in a running buffer is triggered in two different types of session terminations: complete session termination and premature session termination. In the complete session termination, a session terminates only when the delivery of the whole media object is completed, which only happens when the buffer is in the *Running State*. In this case, assume that  $R_i^1$  is the first request being served by a running buffer. When  $R_i^1$  reaches the end of the media object, the following two scenarios happen for the *resident buffer* of  $R_i^1$ :

1. If the resident buffer is the only running buffer for the media object, the resident buffer enters the idle state. In this state, the buffer maintains its content until all the resident requests reach the end of the session. On that time, the buffer is released.
2. If the resident buffer is not the only running buffer, that is, there are succeeding

running buffers, the buffer enters the idle state and maintains its content until its end-time. Note that the end-time may be updated by succeeding running buffers.

In the premature session termination, the thing is much more complicated. Here, the request arriving later may terminate earlier, which can happen when a buffer is in the *Construction State* or the *Running State*. Considering a group of consecutive requests  $R_i^1$  to  $R_i^n$  that are being served by a running buffer, the session for one of the requests, say  $R_i^j$ , where  $1 < j < n$ , terminates before everyone else. The situation is handled as follows.

1. If  $R_i^j$  is served from the middle of its resident buffer, that is, there are preceding and succeeding requests served from the same running buffer, the resident buffer maintains its current state and the request  $R_i^j$  gets deleted from all its associated running buffers. Figure 4.7 (a) and (a') show the buffer situation before and after  $R_i^j$  is terminated, respectively.
2. If  $R_i^j$  is served from the head of its resident buffer, the request is deleted from all of its associated running buffers. The resident buffer enters the idle state for a time period of  $I$ . During this time period, the content within the buffer is moved from  $R_i^{j+1}$  to the head. At the end of the time period  $I$ , the buffer space from the tail to the last served request is released and the buffer enters the running state again. The figures of (b) and (b)' in Figure 4.7 show this situation.
3. If  $R_i^j$  is served at the tail of a running buffer, two scenarios are further considered.
  - After deleting the  $R_i^j$  from the request list of its resident buffer, if the request list is not empty, then do nothing. Otherwise, the algorithm can choose to shrink

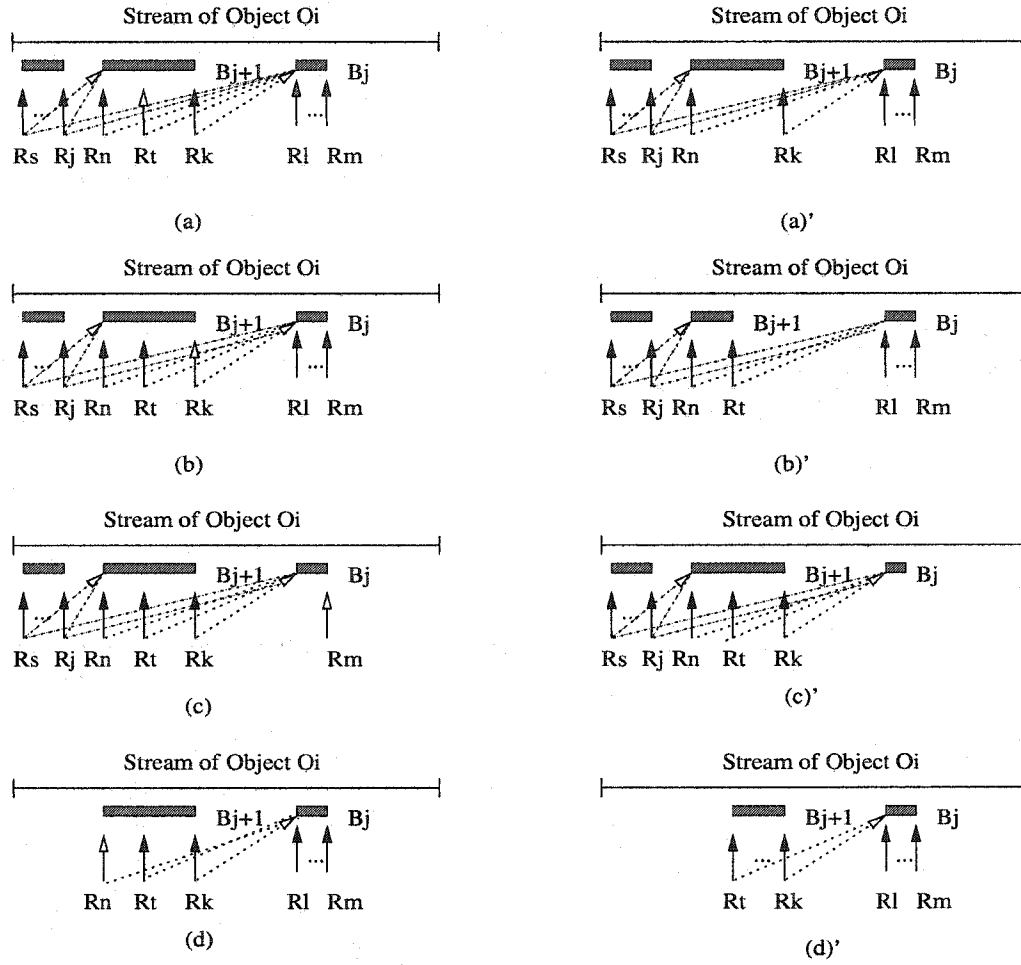


Figure 4.7: SRB memory reclamation: different situations of session termination

the buffer to the extent that  $R_i^{j-1}$  can still be served from the buffer (assuming  $R_i^{j-1}$  is a resident request of the same buffer). In this case, the *End-Times* of the succeeding running buffers need to be adjusted. The figures of (c) and (c)' in Figure 4.7 illustrate this situation.

- If  $R_i^j$  is at the tail of the last running buffer, as shown in Figure 4.7 (d), the buffer will be shrunk to the extent that  $R_i^{j-1}$  is just served from the buffer.  $R_i^j$  is deleted from the request list. Subsequently, the buffers run as shown in Figure 4.7 (d').

### 4.3.2.3 SRB Buffer Replacement Policy

The replacement policy is important in the sense that the available memory is still scarce compared to the size of video objects. So to efficiently use the limited resources is critical to achieve the best performance gain. In this section, we propose popularity based replacement policies for the SRB media caching algorithm. The basic idea is described as follows:

- When a request arrives while there is no available memory, all the objects that have on-going streams in the memory are ordered according to their popularities calculated in a certain past time period. If the object being demanded has a higher popularity than the least popular object in the memory, then the latest running buffer of the least popular object will be deallocated, and the space is re-allocated to the new request. Those requests without running buffers do not buffer their data at all. In this case, theoretically, they are assumed to have no memory consumption.

We have precisely analyzed our popularity based replacement policies by both the modeling and the simulation in [21].

### 4.3.3 Patching SRB (PSRB) Algorithm

Since the proxy has a finite amount of memory space, it is possible that the proxy serves as a bypass sever to transiently cache concurrent sessions. In the SRB algorithm, concurrent sessions are not shared, which may lead to excessive proxy load when there are huge number of requests to different objects. Motivated by this, the SRB algorithm is extended to a Patching SRB (PSRB) algorithm which enables the sharing of concurrent sessions.

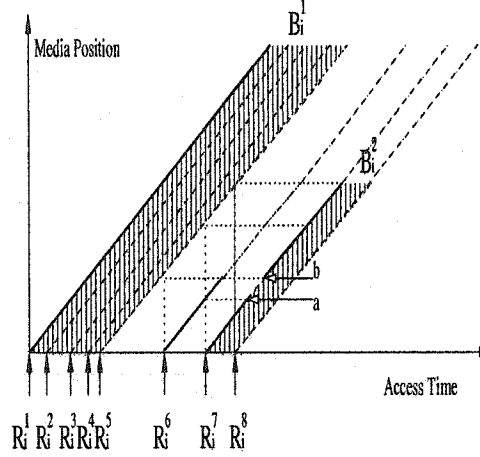


Figure 4.8: An example of the PSRB algorithm

Figure 4.8 illustrates a PSRB scenario. The first running buffer  $B_i^1$  has been formed for requests  $R_i^1$  to  $R_i^5$ . No buffer is running for  $R_i^6$  since it does not have a close neighboring request. However, a patching session has been started to retrieve the absent prefix in  $B_i^1$  from the content server. At this time, request  $R_i^6$  is served from both the patching session and  $B_i^1$  until the missing prefix is patched. Then,  $R_i^6$  is served from  $B_i^1$  only (the solid line for  $R_i^6$  ends).

When  $R_i^7$  and  $R_i^8$  arrive and form the second running buffer  $B_i^2$ , they are served from  $B_i^1$  and  $B_i^2$  as described in the SRB algorithm. In addition, they also receive data from the patching session initiated for  $R_i^6$ . Note that the patching session for  $R_i^6$  is transient, or we can think of it as a running buffer session with zero buffer size. As evident from the figure, the filling of  $B_i^2$  does not cause server traffic between position  $a$  and  $b$  (no solid line between  $a$  and  $b$ ) since  $B_i^2$  is filled from the patching session for  $R_i^6$ . Sharing the patching session further reduces the the number of server accesses for  $R_i^7$  and  $R_i^8$ . In general, the PSRB algorithm is a combination of the SRB algorithm with the optimal patching algorithm proposed in [103]. By taking advantage of the client-side storage, PSRB tries to maximize

the data sharing among concurrent sessions in order to minimize the traffic in the content server.

## 4.4 Performance Evaluation

To evaluate the performance of the proposed algorithms and to compare them with prior solutions, we conduct event-driven simulations based on the same synthetic and real workloads we used in section 2.6 of Chapter 2. The *Low-Bound* and *High-Bound* for the initial buffer size are 2 Mbytes and 16 Mbytes in simulations.

### 4.4.1 Evaluation Metrics

We have implemented an event-driven simulator to model a proxy's memory caching behaviors. Since the *object hit ratio* or *hit ratio* is not suitable for evaluating the caching performance of the streaming media, we use the *server traffic reduction* (shown as *bandwidth reduction* in the figures) to evaluate the performance of the proposed caching algorithms. If the algorithms are employed on a server, this parameter indicates disk I/O traffic reduction.

Using SRB or PSRB algorithms, a client needs to listen to multiple channels for the maximal sharing of the cached data in the proxy's memory. We measure the traffic between the proxy and the client in terms of the *average client channel requirement*. This is an averaged number of channels the clients are listening to during the sessions. Since the clients are listening to earlier on-going sessions, storage is needed at the client to buffer the data before its presentation. We use the *average client storage requirement* in percentage of the full size of the media object to indicate the storage requirement on the client side.

The effectiveness of the algorithms is studied by simulating different *scale* factors for the allocation of the initial buffer size and varying memory cache capacities. The streaming rate is assumed to be constant for simplicity. The simulations are conducted on HP workstation x4000, with 1 GHz CPU and 1 GB memory.

For each simulation, we compare a set of seven algorithms in three groups. The first group contains buffering schemes which include the running buffer caching and the interval caching. The second group contains patching algorithms, specifically the greedy patching, the grace patching and the optimal patching. The third group contains the two sharing running buffer algorithms proposed in this study.

In the following subsections, we present the simulation results. We consider complete viewing scenario for streaming media caching in the Web environment and the partial viewing scenario in Web environment first. The evaluation results on a real workload are presented then.

#### 4.4.2 Performance of the WEB Workload

First, we evaluate the caching performance with respect to the initial buffer size. With a fixed memory capacity of 1GB, the initial buffer size varies from 1 to 1/32 of the length of the media object. For each *scale* factor, the initial buffer of different sizes is allocated if the length of the media object is different. The *server traffic reduction*, the *average client channel requirement* and the *average client storage requirement* are recorded in the simulation. The results are plotted in Figure 4.9.

Figure 4.9 (a) shows the server traffic reduction achieved by each algorithm. Notice that PSRB achieves the best reduction and SRB achieves the next best reduction after

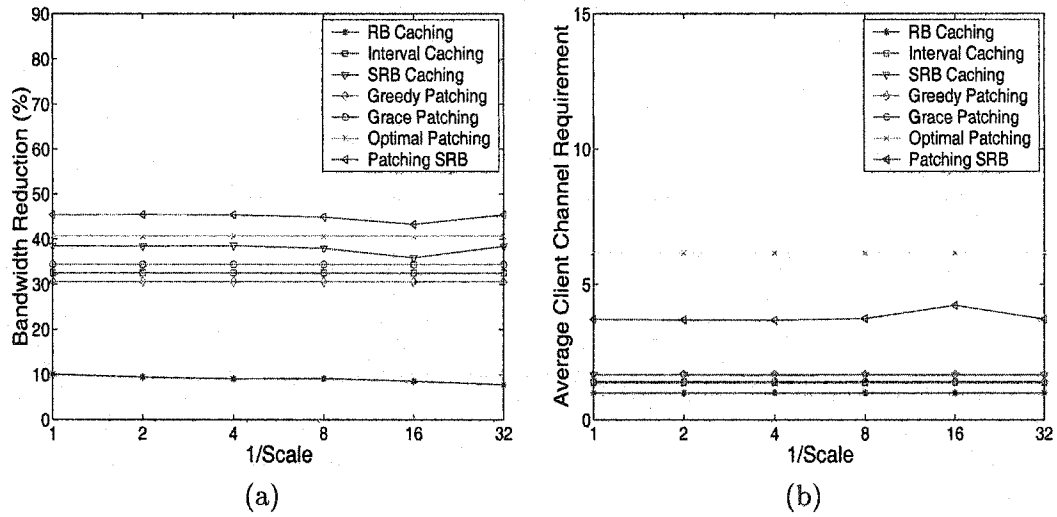


Figure 4.9: WEB: server traffic reduction and average client channel with 1GB Memory

the optimal patching. RB caching achieves the least amount of reduction. As expected, the performance of the three patching algorithms does not depend on the *scale* factor for allocating the initial buffer. Neither does that of the interval caching since the interval caching allocates buffers based on access intervals.

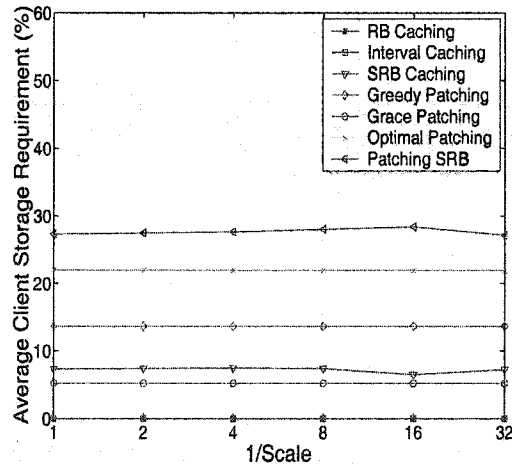


Figure 4.10: WEB: storage requirement (%) with 1GB memory

For the running buffer schemes, we observe some variation in performance with respect to the *scale* factor. In general, the variations are limited. To a certain extent, the performance



gain of the *bandwidth reduction* is a trade-off between the number of running buffers and the sizes of running buffers. A larger buffer implies that more requests can be served from the proxy buffer. However, a larger buffer also indicates that less memory space is left for other requests. This in turn leads to a larger number of server accesses since there is no available memory. On the other hand, a smaller buffer may serve a smaller number of requests but it leaves more memory space for the system to allocate for other requests.

Figure 4.9 (b) and Figure 4.10 show the average channel and storage requirement on the client. Notice that the optimal patching achieves a better server traffic reduction by paying the penalty of imposing the biggest number of client channels required. Comparatively, PSRB and SRB require 30 to 60% fewer client channels while achieving a better or closer server traffic reduction ratio.

PSRB allows the session sharing even when memory space is not available. It is therefore expected that PSRB achieves the highest rate of server traffic rate reduction. In the meantime, it also requires the largest client side storage. On the other hand, SRB achieves about 6% less traffic reduction, but the requirement on client channel and storage is significantly lower.

We now investigate the performance of different algorithms with respect to various memory capacities on the proxy. In this simulation, we use a fixed *scale* factor of  $1/4$  for the initial buffer size.

Figure 4.11 (a) shows an unchanged traffic reduction rate for the three patching algorithms. This is expected since no proxy memory resource is utilized in the patching. On the other hand, all the other algorithms investigated achieve higher traffic reduction rates when the memory capacity increases. It is important to note that the proposed SRB algorithms

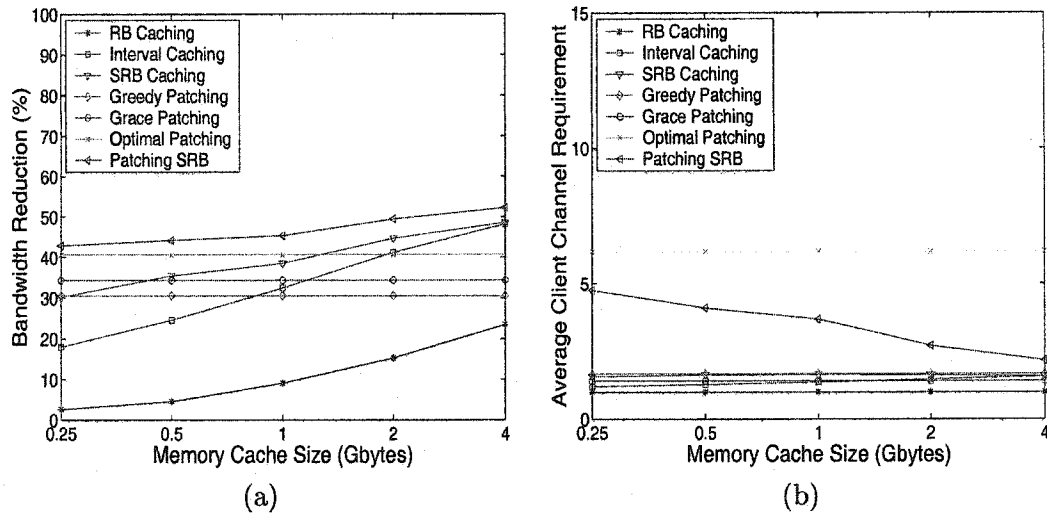


Figure 4.11: WEB: bandwidth reduction and average client channel with the scale of 1/4

achieve a better traffic reduction ratio than the interval caching algorithm and the running buffer algorithm.

In Figure 4.11 (b), the client channel requirement decreases for the PSRB algorithm when the memory capacity increases. This is again expected since more clients are served from the proxy buffer instead of proxy patching session.

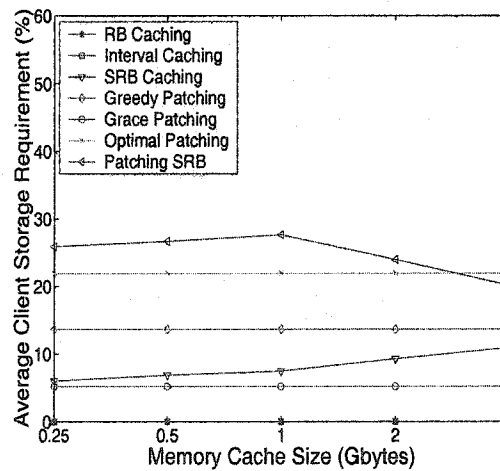


Figure 4.12: WEB: storage requirement (%) with the scale of 1/4

When the cache capacity reaches 4GB, PSRB only requires 30% of the client channel

needed for the optimal patching scheme. PSRB also requires less client storage at this point as indicated in Figure 4.12. And yet, PSRB achieves more than 10 percentage points of traffic reduction comparing to the optimal patching scheme. For the SRB algorithm, it generally achieves the second best traffic reduction with even less penalty on client channel and storage requirements.

#### 4.4.3 Performance of the PART Workload

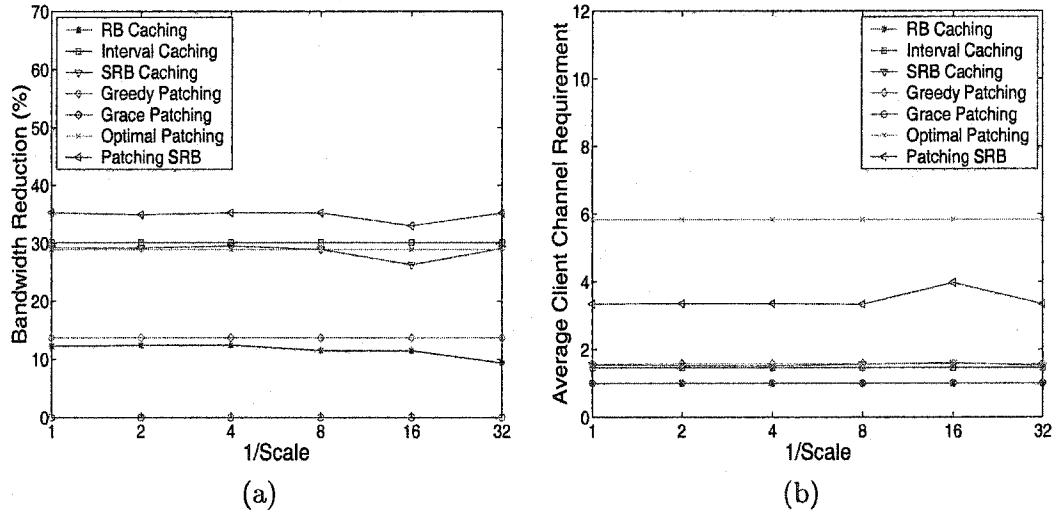


Figure 4.13: PART: bandwidth reduction and average client channel requirement with 1GB memory

In streaming media delivery over the Internet, it is possible that some clients terminate the session after watching for a while from the beginning of the media object. It is important to evaluate the performance of the proposed algorithm under this situation. Using the PART workload, we perform the same simulations and evaluate the same set of parameters. Figure 4.13 (a) shows similar characteristics as that in Figure 4.9. PSRB and SRB still achieve better traffic reduction. The conclusion holds that PSRB uses 60% of the client channel to achieve 5 percentage point better traffic reduction compared with the optimal

patching.

In the event that a session terminates before it reaches the end of the requested media object, it is possible that the client has already downloaded future part of the media stream which is no longer needed. To characterize this wasted delivery from the proxy to the client, we record *average client waste*. It is the percentage of wasted bytes versus the total prefetched data. Figure 4.14 shows the client waste statistic. Note that for PART and REAL workloads, since both contain premature session terminations, the prefetched data which is not used in the presentation are not counted as bytes hit in the calculation of the *server traffic reduction*.

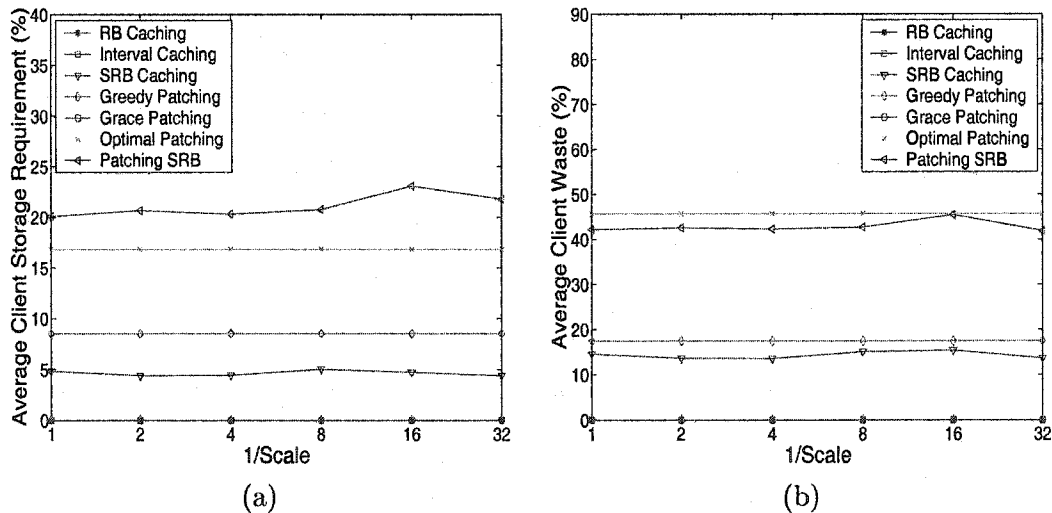


Figure 4.14: PART: average client storage requirement (%) and client waste (%) with 1GB memory

As shown in Figure 4.14 (b), PSRB and SRB have about 42% and 15% of prefetched data wasted comparing with 0% for interval caching. Since the wasted bytes are not counted as hit, this leads to the lowered traffic reduction rate for PSRB and SRB comparing to that of interval cache. From another perspective, interval caching does not promote sharing among buffers, hence the client listens to one channel only and there is no buffering of

future data. Thus, there is no waste in proxy-to-client delivery in the event of premature session termination.

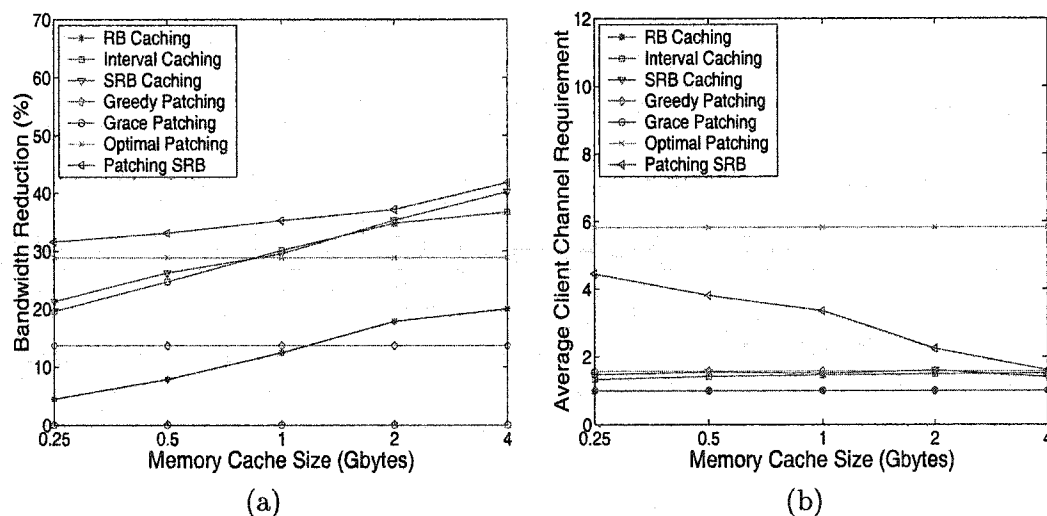


Figure 4.15: PART: bandwidth reduction and average client channel requirement with the scale of  $1/4$

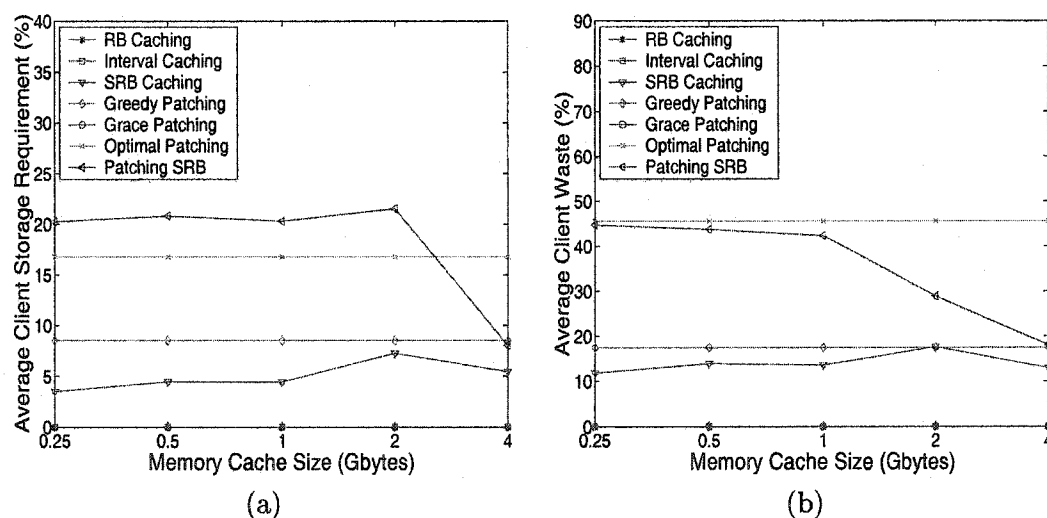


Figure 4.16: PART: average client storage requirement (%) and client waste (%) with the scale of  $1/4$

We now again start investigation of the caching performance with a fixed *scale* factor for the initial buffer size in Figure 4.15. Comparing with Figure 4.9 (a), the distances between

the traffic reduction curves between PSRB, SRB and interval caching become much smaller in general. This reinforces the observation above that PSRB and SRB may lead to more wasted bytes in the partial viewing cases. In addition, the grace patching achieving almost no traffic reduction shows its incapability in dealing with the partial viewing situation. The reason might be that the new session started by the grace patching, which is supposed to be a complete session, often terminates when 20% of the media object is delivered. Since the duration of the session is short, it is less likely that a new request to the same media object is received.

In Figure 4.15 (b), PSRB demonstrates monotonic decreasing of average client channel requirement when memory capacity increases. This is due to the fact that there is a fewer number of zero-sized running buffers with increasing proxy memory capacity. Similarly, as shown in Figure 4.16, the client storage requirement and average client waste also decrease since a fewer number of patching is required.

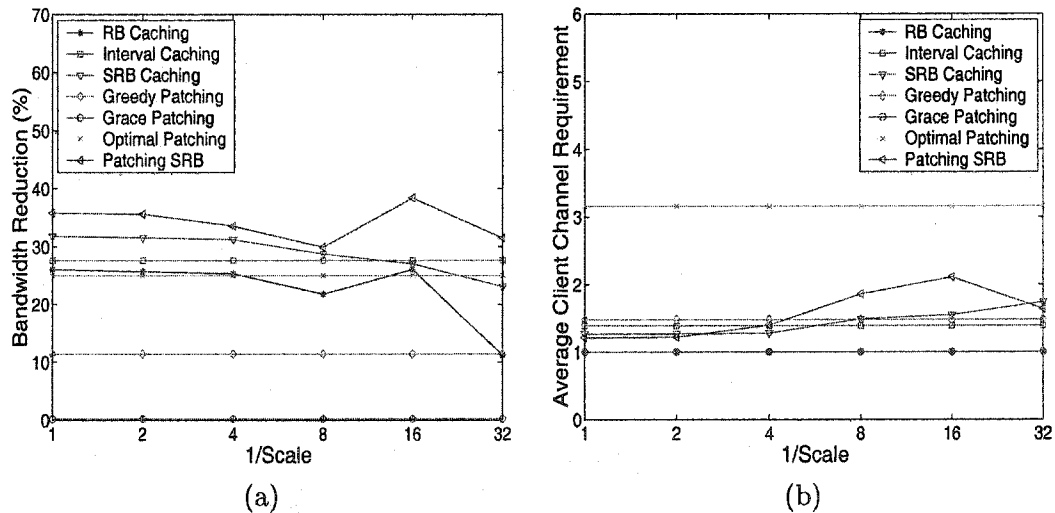


Figure 4.17: REAL: bandwidth reduction and average client channel requirement with 1GB memory

#### 4.4.4 Performance of the REAL Workload

Based on a real video delivering workload captured from corporate intranet, the same simulations are conducted to evaluate the caching performance. We start first by evaluating the caching performance versus varying *scale* factor for the initial buffer size.

Comparing Figure 4.17 (a) with Figure 4.9 (a), it is clear that changing *scale* factor has a much more significant impact on the behavior of the proposed SRB and PSRB algorithms for REAL. This could be due to the burst nature of the accesses logged in the workload. To a certain degree, this result indicates the effectiveness of the adaptive buffer allocation scheme we proposed in the algorithms.

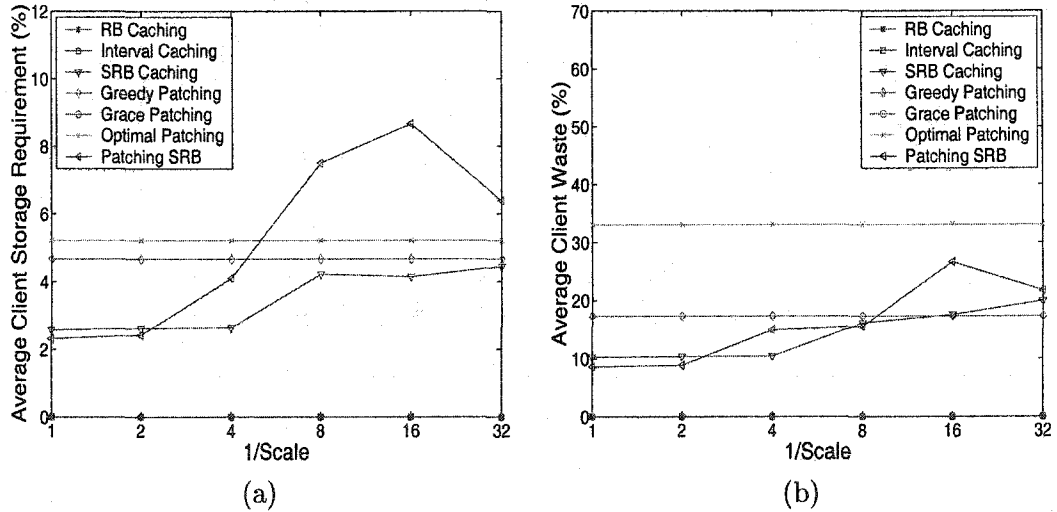
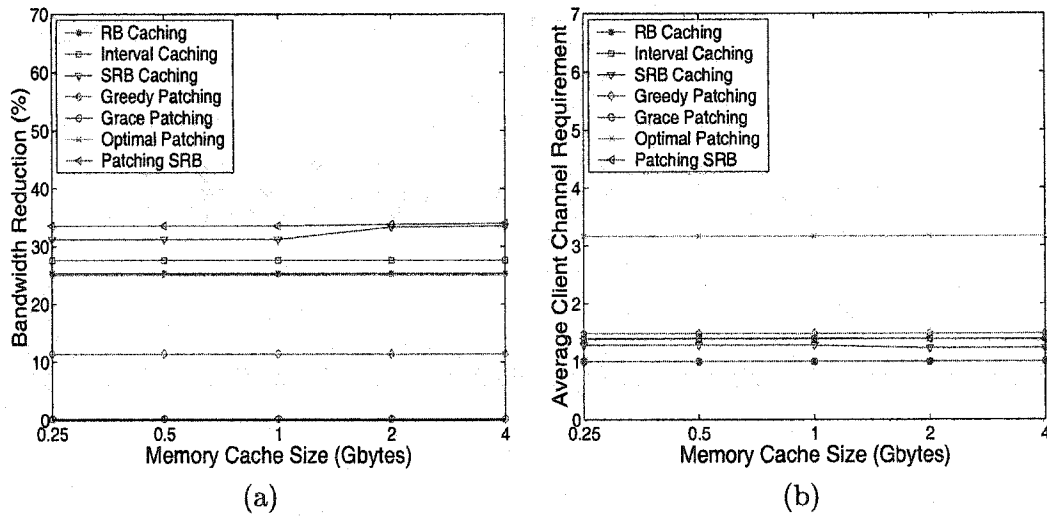


Figure 4.18: REAL: average client storage requirement (%) and client waste (%) with 1GB memory

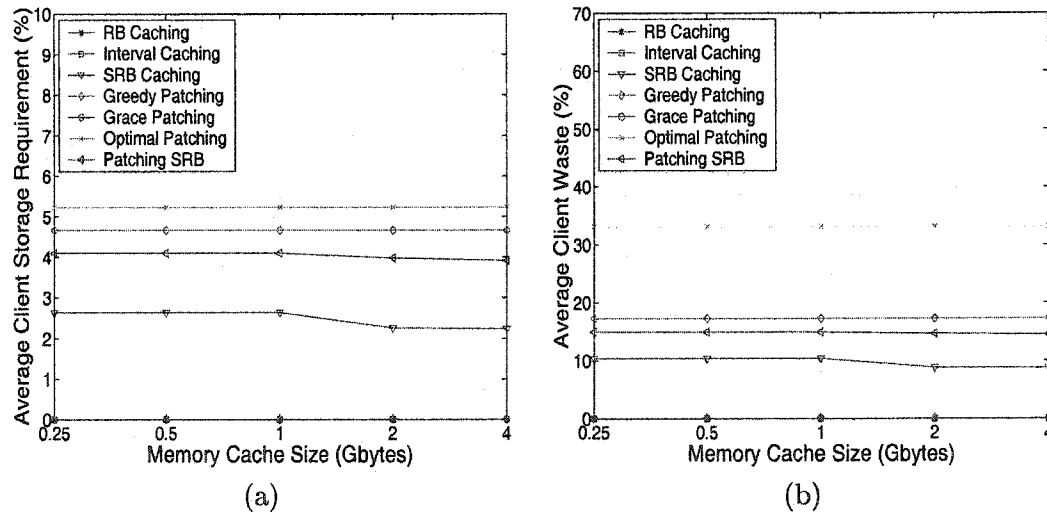
Setting the initial buffer size as  $1/4$  of the requested media objects, we again evaluate the caching performance with increasing amount proxy memory available. Figure 4.19 and 4.20 show the server traffic reduction and the client side statistics.

Compared with the simulation results obtained with synthetic workloads, Figure 4.19 (a) shows a flat gain when memory capacity increases. It seems to indicate that memory



**Figure 4.19:** REAL: bandwidth reduction and average client channel requirement with the scale of  $1/4$

capacity is less of a factor. Once again, the burst nature of the request arrival may play a role here. In addition, the volume of the burst may also be low which leads to the fact that limited amount of memory space suffices the sharing of sessions.



**Figure 4.20:** REAL: average client storage requirement (%) and client waste (%) with the scale of  $1/4$

The simulation results for the real workload provide the following understanding for the



studying of caching of streaming media: **Contrary to the intuition that caching of streaming media requires large memory space, intelligent allocation of the available memory resource is probably more important than the memory resource itself.** This is also the motivation of the proposed SRB and PSRB algorithms. More details on the performance analysis can be referred in the [21].

#### 4.4.5 Further Analysis on the Client Channel Requirement

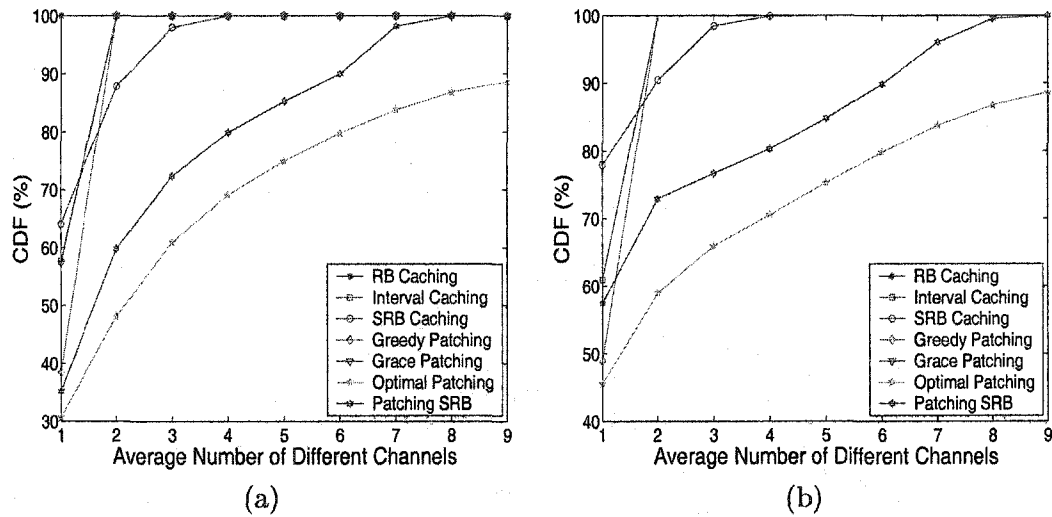


Figure 4.21: Client channel requirement CDF: WEB and PART

The performance analysis in the previous section indicates that SRB and PSRB algorithm achieve superior server traffic reduction by utilizing the memory resource on the proxy and sufficient bandwidth resource between the proxy and the clients. In most cases, the proxy streams data from multiple buffers to the client through multiple channels. To have a better understanding on the client channel requirement, we collect additional statistics that illustrates the distribution of the number of client channels required. Figure 4.21 and 4.22 show the CDF of the client channel requirement for simulations on the workloads. In

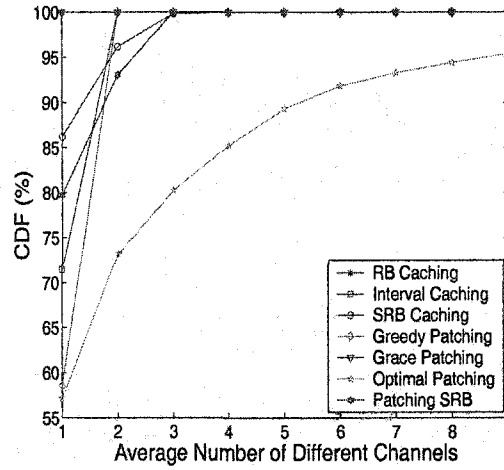


Figure 4.22: Client channel requirement CDF: REAL

these simulations, the proxy has 1GB memory capacity and the *scale* factor for the initial buffer size is fixed at  $1/4$ .

For simple running buffer caching, since no session sharing is happening, only one channel is required for clients. Greedy and grace patching algorithms need at most 2 client channels. For the WEB workload, approximately 60% of greedy patching sessions and 40% of grace patching sessions require only one client channel. Interval caching also requires at most 2 client channels with 78% of the sessions requiring only one channel.

Optimal patching needs the largest number of client channels. It is not surprising since requests arrive later always try to patch to as many earlier on going sessions as possible. For all four workloads, the number of client channel required could exceed nine.

For the proposed SRB and PSRB algorithms, the number of the required client channel often falls in between the optimal patching and the group of algorithms containing greedy, grace patching and interval caching. Note further that for SRB algorithm, very few sessions require more than 3 client channels with around 98% of session requires no more than 2. The statistics shown for in the REAL workload as in Figure 4.22 verifies further that 94% of

the PSRB session only needs no more than 2 client channels. On the other hand, more than 10% of the optimal patching sessions needs 3 or more client channels. Referring back to Figure 4.19 (a), it is clear that SRB and PSRB achieve better server traffic reduction than the optimal caching. This analysis enhances the advantages of the proposed algorithms.

In addition, these observations are useful when limited bandwidth resource is available between the proxy and client. In this case, the proxy system can choose to execute a session sharing algorithm which achieves better caching performance without exceeding the capacity of link between the proxy and clients.

## 4.5 Summary

In this chapter, we proposed two new algorithms for caching of streaming media objects by utilizing the proxy memory. Shared Running Buffers (SRB) caching algorithm is proposed to dynamically cache media objects in the proxy memory during delivery. Patching SRB(PSRB) algorithm is proposed to further enhance the memory utilization in the proxy. The adaptiveness of the two algorithms are analyzed and exploited. Simulations based on both synthetic and real work load are conducted. The simulation results show the efficiency of the proposed algorithms. Both algorithms require the client be capable of listening to multiple channels at the same time. Comparing with previous solution which also require multiple client channels, the proposed algorithm achieves better server traffic reduction with less or similar load on the link between the proxy and the client.

SRB algorithms have been evaluated without considering the effect of the disk based proxy caching. The proposed algorithms can also be applied to any streaming server to

reduce the disk bandwidth requirement. Now we are considering to combine the memory based caching and disk based caching together. We are also investigating the performance of the algorithms when the client side storage is limited and the streaming rate is varying. As SRB algorithms can further improve the streaming delivery performance when applying in the streaming proxy, the implementation of these algorithms in our Hyper-Proxy system is under investigation.

## Chapter 5

# Other Related Work

Besides the studies performed on the Internet caching systems for delivering streaming media objects, I have also looked into the P2P assisted proxy-based streaming, Internet caching problems related to the static objects and dynamic objects, as well as the problems in the cluster computing. Following is a brief summary for each of them.

### 5.1 Coordinating P2P System and Proxy for Streaming Media Delivery

As the demand of delivering streaming media content in the Internet has become increasingly high for scientific, educational, and commercial applications, three representative technologies have been developed for this purpose, each of which has its merits and serious limitations. Infrastructure-based CDNs with dedicated network bandwidths and powerful media replicas can provide high quality streaming services but at a high cost. Server-based proxies are cost-effective but not scalable due to the limited proxy capacity and its centralized control. Client-based P2P networks are scalable but do not guarantee high quality streaming service due to the transient nature of peers. To address these limitations, we propose a novel

and efficient design of a scalable and reliable media proxy system supported by P2P networks. This system is called *PROP*, abbreviated from our technical theme of “collaborating and coordinating PROxy and its P2P clients”. Our objective is to address both scalability and reliability issues of streaming media delivery in a cost-effective way. In the PROP system, the clients’ machines in an intranet are self-organized into a structured P2P system to provide a large media storage and to actively participate in the streaming media delivery, where the proxy is also embedded as an important member to ensure quality of streaming service. The coordination and collaboration in the system are efficiently conducted by our P2P management structure and replacement policies. We have comparatively evaluated our system by trace-driven simulations with synthetic workloads and with a real workload extracted from the media server logs of an enterprise network. The results show that our design significantly improves the quality of media streaming and the system scalability [53].

## 5.2 Detective Browser

The amount of dynamic Web contents and secured e-commerce transactions has been dramatically increasing [19, 105, 122] on the Internet where proxies between clients and Web servers are commonly used for the purpose of sharing commonly accessed data and reducing the Internet traffic. A significant and unnecessary Web access delay is caused by the overhead in the proxy to process two types of requests, namely dynamic Web content requests and secured transaction requests, not only increasing response time, but also raising some security concerns. Conducting experiments on Squid proxy 2.3STABLE4, we have quantified the unnecessary processing overhead to show their significant impact on increased

client access response times. We have also analyzed the technical difficulties in eliminating or reducing the processing overhead and the security loopholes based on the existing proxy structure. In order to address these performance and security concerns, we propose a simple but effective technique from the client side that adds a detector interfacing with a browser. With this detector, a standard browser, such as the Netscape/Mozilla, will have simple detective and scheduling functions, called *Detective Browser*. Upon an Internet request from a client, the Detective Browser can immediately determine whether the requested content is dynamic or secured. If so, the browser will bypass the proxy and forward the request directly to the Web server; otherwise, the request will be processed through the proxy. We implemented a Detective Browser prototype in Mozilla version 0.9.7, and tested its functionality and effectiveness. Since we simply move the necessary detective functions from a proxy to a browser, the Detective Browser introduces little overhead to Internet accessing, and our software can be patched to existing browsers easily [29].

### 5.3 Cooperatively Shared Proxy Browsers

Proxy hit ratios tend to decrease as the demand and supply of Web contents are becoming more diverse. By case studies, we quantitatively confirm this trend, and observe significant document duplications among a proxy and its client browsers' caches. One reason behind this trend is that the client/server Web caching model does not support direct resource sharing among clients, causing the Web contents and the network bandwidths among clients being relatively under-utilized. To address these limits and improve Web caching performance, we have extensively enhanced and deployed our browsers-aware framework,

a peer-to-peer Web caching management scheme. We make the browsers and their proxy share the contents to exploit the neglected but rich data locality in browsers, and reduces document duplications among the proxy and browsers' caches to effectively utilize the Web contents and the network bandwidth among clients. The objective of our scheme is to improve the scalability of proxy-based caching both in the number of connected clients and in the diversity of Web documents. We show that building such a caching system with considerations of sharing contents among clients, minimizing document duplications, and achieving data integrity and communication anonymity, is not only feasible but also highly effective [120].

## 5.4 Dynamic Load Sharing With Unknown Memory Demands in Clusters

A compute farm is a pool of clustered workstations to provide high performance computing services for CPU-intensive, memory-intensive, and I/O-active jobs in a batch mode. Existing load sharing schemes with memory considerations assume jobs' memory demand sizes are known in advance or predictable based on clients' hints. This assumption can greatly simplify the design and implementation of load sharing schemes, but is not desirable in practice. In order to address this concern, we present three new results and contributions in this study. (1) Conducting Linux kernel instrumentation, we have collected different types of workload execution traces to quantitatively characterize job interactions, and modeled page fault behavior as a function of the overloaded memory sizes and the amount of jobs' I/O activities. (2) Based on experimental results and collected dynamic system infor-



mation, we have built a simulation model which accurately emulates the memory system operations and job migrations with virtual memory considerations. (3) We have proposed a memory-centric load sharing scheme and its variations to effectively process dynamic memory allocation demands, aiming at minimizing execution time of each individual job by dynamically migrating and remotely submitting jobs to eliminate or reduce page faults and to reduce the queuing time for CPU services. Conducting trace-driven simulations, we have examined these load sharing policies to show their effectiveness. Results of this study have been published in [27, 118].

## 5.5 Adaptive Memory Allocations in Clusters to Handle Large Data-Intensive Jobs

In a cluster system with dynamic load sharing support, a job submission or migration to a workstation is determined by the availability of CPU and memory resources of the workstation at the time. In such a system, a small number of running jobs with unexpectedly large memory allocation requirements may significantly increase the queuing delay times of the rest of jobs with normal memory requirements, slowing down executions of individual jobs and decreasing the system throughput. We call this phenomenon as the *job blocking* problem because the big jobs block the execution pace of majority jobs in the cluster. Since the memory demand of jobs may not be known in advance and may change dynamically, the possibility of unsuitable job submissions/migrations to cause the blocking problem is high, and the existing load sharing schemes are unable to effectively handle this problem. We propose a software method incorporating with dynamic load sharing, which adaptively

reserves a small set of workstations through virtual cluster reconfiguration to provide special services to the jobs demanding large memory allocations. This policy implies the principle of shortest-remaining-processing-time policy. As soon as the blocking problem is resolved by the reconfiguration, the system will adaptively switch back to the normal load sharing state. We present three contributions in this study. (1) We quantitatively present the conditions to cause the job blocking problem. (2) We present the adaptive software method in a dynamic load sharing system. We show that the adaptive process causes little additional overhead. (3) Conducting trace-driven simulations, we show that our method can effectively improve the cluster computing performance by quickly resolving the job blocking problem. The effectiveness and performance insights are also analytically verified. Results of this study have been published in [28, 119].

## Chapter 6

# Conclusion and Future Work

### 6.1 Conclusion

The Internet technologies have greatly changed our life in the recent years, where all kinds of contents are available through the widely deployed Web sites. The contents on Web sites have evolved from the simple text-based Web pages to complicated dynamic contents and streaming media objects. Their applications in many areas, such as education, medical treatment, and entertainment, demand cost-effective and high performance delivery strategies. The research of efficient Internet content delivery strategies has been the focus of many studies from both industry and academia.

In the early stage, the proxy caching approach has been very successfully used for delivering static text-based contents by storing them locally closer to the client after a client access, so that subsequent client requests for a same object can be directly served from the proxy instead of the server. With the proliferation of streaming media objects on the Internet today, the traditional proxy caching approach faces new challenges due to the following two facts. The first is that the size of a media object is generally much larger than a text-based object. Thus, caching the entire media objects as caching static Web

objects can quickly exhaust the proxy cache space, making it infeasible. The second is that the client requesting streaming media objects always demands continuous streaming delivery and a small startup latency. Occasional delays of the data transferring over the Internet may be acceptable for text-based Web browsing, however, the streaming media data transferring delay always results in playback jitter at the client side. The jitter not only is annoying, but also can drive clients away from the streaming service. A large startup latency have the same effect to clients.

In this dissertation, we built cost-effective and high performance proxy-based Internet caching systems for delivering streaming media objects, with minimum playback jitter and a small perceived startup latency at the client side. At the same time, it can achieve reasonable good cache performance so that the Internet traffic and the disk bandwidth requirement to the media server are reduced.

We first examined the performance objectives of the existing streaming proxy designs, and found that conflicting performance objectives exist in current schemes. Through heuristic and modeling approaches, we provided effective solutions to balance these conflicting performance objectives. By comprehensively considering the objectives from the client point of view, we proposed a streaming proxy design model: a streaming proxy should provide continuous streaming delivery to the client subject to a small startup latency and high byte hit ratio. Guided by this model, we designed Hyper-Proxy. Our evaluation based on synthetic and real workloads shows that Hyper-Proxy can deliver streaming media data to clients with minimum playback jitter and small startup latency, while it also achieves good cache performance. As far as we know, this is the first system considering all three performance objectives comprehensively.

Second, we implemented Hyper-Proxy and evaluated it in the global Internet environment and the local network environment. The implementation leverages the existing Internet infrastructure: Hyper-Proxy talks to the media content server via HTTP while it talks to the client via RTP/RTSP. Thus, the streaming functions are pushed from the streaming server to the proxy close to the client. Therefore, the traditional Web servers, such as Apache, can now provide real streaming service through Hyper-Proxy. The evaluation with the Hyper-Proxy and content server located in the LAN and connected between Japan and USA demonstrated it can provide satisfying streaming results to clients while maintaining good cache performance. To the best of our knowledge, this is the first system of this kind. Now Hyper-Proxy has been deployed in a large industrial environment for trials.

Finally, considering the client access locality in the memory of the proxy when subsequent clients request the same object successively, we proposed a group of the Shared Running Buffers (SRB) based techniques to exploit this locality. In SRB based techniques, streaming data are not only shared among subsequently arrived sessions served by a running buffer, but also among different running buffers. Even more, the Patching SRB (PSRB) further utilizes the client side storage to increase the number of possibly shared sessions. Our evaluation shows that SRB based techniques further reduce the media server's load, the amount of the network traffic and the client perceived response time.

## 6.2 Future Work

With Hyper-Proxy, now the new proxy-based Internet caching systems can cache the streaming media as well as static Web pages on the Internet, the proxy-assisted content delivery

to all kinds of devices (such as PDAs, wireless phones, etc.) becomes necessary and promising. Proxy-assisted transcoding only shadows a light on this problem. Provided that each proxy only has a limited computing power, cache space and memory, how it can deal with the mobility is important and how cooperative proxies work together to provide such kind of services will attract more attention from academia and industry. With the availability of streaming proxy to support the on-demand streaming, how the existing proxy provides support to live streaming is another interesting direction. We think to research and provide solutions to these problems has great potentials. Therefore, I will continue the research on the following directions in the future.

### 6.2.1 Streaming Based on Proxy-assisted Transcoding

Today a client can use PDA, cell-phone or other mobile devices to browse the Internet besides the desktop or laptop computers. The usage of these devices complicates the Internet streaming media delivery problem because a media object that is appropriate to a computer may not be appropriate to a PDA. They have different screen sizes and different color depths. The object has to be customized for different client devices. Thus, the media delivery network must be able to distinct and adapt to different client devices by conveying an appropriate version of a media object to a corresponding client. The problem is how. There are different solutions. One is to store multiple versions of an object statically, which we call as offline-transcoding. That is, to prepare different versions for all kinds of devices before the streaming is available. It consumes a huge amount of storage to store all versions of an object. The other is online-transcoding. That is to do transcoding and delivering simultaneously. This approach demands a large amount of CPU cycles on the fly. Except

for these two approaches, we are looking for if there is any other approach and whether it is possible to develop some new approaches based on media segments.

### 6.2.2 Cooperative Streaming Proxy to Support Mobile Computers

The usage of mobile-computers not only incurs the aforementioned transcoding problem, but also incurs the mobility problem. When a client holding a PDA or cell-phone reads the video-based news, the client may be in a moving train (a typical practice in big cosmopolitan cities, such as Tokyo, Hong Kong), or may walk on the street. Thus, the media delivery network should provide a nomadic streaming service. This implies one streaming proxy is not capable of providing such a continuous streaming. The cooperation among multiple proxies is a must. However, how different proxies cooperate among themselves is not an easy problem, since a continuous streaming service must be guaranteed. The hand-off between the proxy and the client is not only expensive, but also can cause the interrupted services. This problem is related to a lot of research issues I am interested in.

### 6.2.3 Live Streaming Enabled Proxy

Currently, we are working on the streaming proxy for delivering on-demand streaming media objects. On the Internet, there is another stream type, called live stream. Video conferencing is such an example. To transport live events to clients, live streaming has to be relied on. This is not a new problem, as a lot of multicast research has been done and is still undergoing. However, whether it can work smoothly with a segment-based proxy is not clear yet. The previous transcoding and mobility problems are also related to this issue, since a client may use a PDA to watch the Oscar's Annual Academy Awards, while

walking on the street. For this application, the offline-transcoding does not work. How to do online-transcoding for live events streaming cost-effectively remains open.



# Bibliography

- [1] <http://www.squid-cache.org/>.
- [2] <http://www.tcpdump.org/>.
- [3] Linux advanced routing & traffic control. <http://lartc.org/>.
- [4] Video store magazine. <http://www.videostoremag.com>, March 2000.
- [5] S. ACHARYA AND B. SMITH. Middleman: A video caching proxy server. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Chapel Hill, NC, June 2000.
- [6] S. ACHARYA, B. SMITH, AND P. PARNES. Characterizing user access to videos on the World Wide Web. In *Proceedings of ACM/SPIE Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 1998.
- [7] S. ACHARYA, B. SMITH, AND P. PARNES. Characterizing user access to videos on the World Wide Web. In *Proceedings of ACM/SPIE Conference on Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 2000.
- [8] C. C. AGGARWAL, J.L. WOLF, AND P.S. YU. A permutation-based pyramid broadcasting scheme for video-on-demand systems. In *Proceedings of the International Conference on Multimedia Computing and Systems*, Hiroshima, Japan, June 1996.
- [9] J. M. ALMEIDA, J. KRUEGER, D. L. EAGER, AND M. K. VERNON. Analysis of educational media server workloads. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Port Jefferson, NY, June 2001.
- [10] D. P. ANDERSON, Y. OSAWA, AND R. GOVINDAN. A file system for continuous media. In *ACM Transactions on Computer Systems (TOCS)*, volume 10 (4), November 1992.
- [11] M. ARLITT AND C. WILLIAMSON. Web server workload characterization. In *Proceedings of ACM SIGMETRICS International Conference on Measurement and Modelling of Computer Systems (SIGMETRICS)*, Philadelphia, PA, May 1996.
- [12] L. BENT, M. RABINOVICH, G. M. VOELKER, AND Z. XIAO. Characterization of a large Web site population with implications for content delivery. In *Proceedings of WWW*, New York City, NY, May 2004.

- [13] E. BOMMAIAH, K. GUO, M. HOFMANN, AND S. PAUL. Design and implementation of a caching system for streaming media over the Internet. In *Proceedings of IEEE Real Time Technology and Applications Symposium (RTAS)*, Washington, DC, May 2000.
- [14] C.M. BOWMAN, P.B. DANZIG, D.R. HARDY, U. MANBER, M.F. SCHWARTZ, AND D.P. WESSELS. Harvest: A scalable, customizable discovery and access system. In *Tech. Re. CU-CS-732-94*, University of Colorado, Boulder, CO, 1994.
- [15] K. S. CANDAN, W. LI, Q. LUO, W. HSIUNG, AND D. AGRAWAL. Enabling dynamic content caching for database-driven Web sites. In *Proceedings of ACM SIGMOD*, Santa Barbara, CA, May 2001.
- [16] P. CAO AND S. IRANI. Cost-aware www proxy caching algorithms. In *Proceedings of USENIX Symposium on Internet Technology and Systems (USITS)*, Monterey, CA, December 1997.
- [17] P. CAO, J. ZHANG, AND K. BEACH. Active cache: Caching dynamic contents on the Web. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, The Lake District, England, September 1998.
- [18] Y. CHAE, K. GUO, M. BUDDHIKOT, S. SURJ, AND E. ZEGURA. Silo, rainbow, and caching token: Schemes for scalable fault tolerant stream caching. In *IEEE Journal on Selected Areas in Communications*, September 2002.
- [19] J. CHALLENGER, A. IYENGAR, AND P. DANTZIG. A scalable system for consistently caching dynamic Web data. In *Proceedings of IEEE INFOCOM*, New York City, NY, March 1999.
- [20] S.-H. G. CHAN AND F. A. TOBAGI. Distributed server architectures for networked video services. In *IEEE/ACM Transactions on Networking*, volume 9(2), April 2001.
- [21] S. CHEN, B. SHEN, S. BASU, AND Y. YAN. SRB: The Shared Running Buffer based proxy caching of streaming sessions. Technical report, Hewlett-Packard Laboratories, 2003.
- [22] S. CHEN, B. SHEN, S. BASU, Y. YAN, AND X. ZHANG. SRB: Shared Running Buffers in proxy to exploit memory locality of multiple streaming sessions. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, Tokyo, Japan, March 2004.
- [23] S. CHEN, B. SHEN, S. WEE, AND X. ZHANG. Adaptive and lazy segmentation based proxy caching for streaming media delivery. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Monterey, CA, June 2003.
- [24] S. CHEN, B. SHEN, S. WEE, AND X. ZHANG. Streaming flow analyses for prefetching in segment-based proxy caching strategies to improve media delivery quality. In

- Proceedings of the International Workshop on Web Content Caching and Distribution (WCW)*, Hawthorne, NY, September 2003.
- [25] S. CHEN, B. SHEN, S. WEE, AND X. ZHANG. Designs of high quality streaming proxy systems. In *Proceedings of IEEE INFOCOM*, Hong Kong, China, March 2004.
  - [26] S. CHEN, B. SHEN, S. WEE, AND X. ZHANG. Investigating performance insights of segment-based proxy caching of streaming media strategies. In *Proceedings of ACM/SPIE Conference on Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 2004.
  - [27] S. CHEN, L. XIAO, AND X. ZHANG. Dynamic load sharing with unknown memory demand of jobs in clustered compute farms. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, AZ, April 2001.
  - [28] S. CHEN, L. XIAO, AND X. ZHANG. Adaptive and virtual reconfigurations for effective dynamic resource allocations in cluster systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.
  - [29] S. CHEN AND X. ZHANG. Detective browsers: A software technique to improve Web access performance and security. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution (WCW)*, Boulder, CO, August 2002.
  - [30] L. CHERKASOVA AND M. GUPTA. Characterizing locality, evolution, and life span of accesses in enterprise media server workloads. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Miami, FL, May 2002.
  - [31] M. CHESIRE, A. WOLMAN, G. VOELKER, AND H. LEVY. Measurement and analysis of a streaming media workload. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, CA, March 2001.
  - [32] M. Y. CHIU AND K. H. YEUNG. Partial video sequence caching scheme for vod systems with heterogeneous clients. In *Proceedings of the International Conference on Data Engineering (ICDE)*, Birmingham, United Kingdom, April 1997.
  - [33] C. COSTA, I. CUNHA, A. BORGES, C. RAMOS, M. ROCHA, J. ALMEIDA, AND B. RIBEIRO-NETO. Analyzing client interactivity in streaming media. In *Proceedings of WWW*, New York City, NY, May 2004.
  - [34] P. D. CUETOS, D. SAPARILLA, AND K. W. ROSS. Adaptive streaming of stored video in a tcp-friendly context: Multiple versions or multiple layers? In *Proceedings of Packet Video Workshop*, Kyongju, Korea, April 2001.
  - [35] Y. CUI AND K. NAHRSTEDT. Layered peer-to-peer streaming. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Monterey, CA, June 2003.

- [36] A. DAN AND D. SITARAM. Buffer management policy for an on-demand video server. In *IBM Research Report 19347*, 1993.
- [37] A. DAN AND D. SITARAM. A generalized interval caching policy for mixed interactive and long video workloads. In *Proceedings of ACM/SPIE Conference on Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 1996.
- [38] A. DAN, D. SITARAM, AND P. SHAHABUDDIN. Scheduling policies for an on-demand video server with batching. In *Proceedings of ACM Multimedia*, San Francisco, CA, October 1994.
- [39] F. DOUGLIS, A. FELDMANN, B. KRISHNAMURTHY, AND J. MOGUL. Rate of change and other metrics: A live study of the World Wide Web. In *Proceedings of USENIX Symposium on Internet Technology and Systems (USITS)*, Monterey, CA, December 1997.
- [40] F. DOUGLIS, A. HARO, AND M. RABINOVICH. Hpp: Html macropreprocessing to support dynamic document caching. In *Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, December 1997.
- [41] B. M. DUSKA, D. MARWOOD, AND M. J. FEELEY. The measured access characteristics of World-Wide-Web client proxy caches. In *Proceedings of USENIX Symposium on Internet Technology and Systems (USITS)*, Monterey, CA, December 1997.
- [42] D. EAGER, M. VERNON, AND J. ZAHORJAN. Bandwidth skimming: A technique for cost-effective video-on-demand. In *Proceedings of ACM/SPIE Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 2000.
- [43] D. L. EAGER, M. C. FERRIS, AND M. K. VERNON. Optimized regional caching for on-demand data delivery. In *Proceedings of ACM/SPIE Conference on Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 1999.
- [44] L. FAN, P. CAO, J. ALMEIDA, AND A. Z. BRODER. Summary cache: A scalable wide-area Web cache sharing protocol. In *Proceedings of the ACM SIGCOMM*, Vancouver, Canada, September 1998.
- [45] Z. FEI, M. H. AMMAR, I. KAMEL, AND S. MUKHERJEE. Providing interactive functions for staggered multicast near video-on-demand systems. In *Proceedings of the International Conference on Multimedia Computing and Systems*, Florence, Italy, June 1999.
- [46] R. FIELDING, J. GETTYS, J. MOGUL, H. FRYSTYK, L. MASINTER, P. LEACH, AND T. BERNERS-LEE. Hypertext transfer protocol – http/1.1. <http://www.faqs.org/rfcs/rfc2616.html>, June 1999.
- [47] N.L.F FONSECA AND R.A eFACANHA. The look-ahead-maximize-batch batching policy. In *IEEE Transactions on Multimedia*, volume 4 (1), March 2002.

- [48] L. GAO AND D. TOWSLEY. Supplying instantaneous video-on-demand services using controlled multicast. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, Florence, Italy, June 1999.
- [49] S. GRIBBLE AND E. BREWER. System design issues for Internet middleware service: Deduction from a large client trace. In *Proceedings of USENIX Symposium on Internet Technology and Systems (USITS)*, Monterey, CA, December 1997.
- [50] C. GRIWODZ, M. ZINK, M. LIEPERT, G. ON, AND R. STEINMETZ. Multicast for savings in cache-based video distribution. In *Proceedings of ACM/SPIE Conference on Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 2000.
- [51] S. GRUBER, J. REXFORD, AND A. BASSO. Protocol considerations for a prefix-caching proxy for multimedia streams. In *Computer Network*, volume 33(1-6), pages 657–668, June 2000.
- [52] K. P. GUMMADI, R. J. DUNN, S. SAROIU, S. D. GRIBBLE, H. M. LEVY, AND J. ZAHORJAN. Measurement, modeling and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.
- [53] L. GUO, S. CHEN, S. REN, X. CHEN, AND S. JIANG. Prop: A scalable and reliable p2p assisted proxy streaming system. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, Tokyo, Japan, March 2004.
- [54] J. GWERTZMAN AND M. SELTZER. World-Wide-Web cache consistency. In *Proceedings of USENIX Annual Technical Conference*, San Diego, CA, January 1996.
- [55] M. HANDLEY AND V. JACOBSEN. SDP: Session Description Protocol. RFC 2327, April 1998.
- [56] H. HAREL, V. VELLANKI, A. CHERVENAK, G. ABOWD, AND U. RAMACHANDRAN. Workload of a media-enhanced classroom server. In *Proceedings of 2nd Annual Workshop on Workload Characterization*, Austin, TX, October 1999.
- [57] L. HE, J. GRUDIN, AND A. GUPTA. Designing presentations for on-demand viewing. In *Proceedings of ACM Conference on Computer Supported Cooperative Work*, Philadelphia, PA, December 2000.
- [58] M. HEFEEDA, A. HABIB, B. BOTEV, D. XU, AND B. BHARGAVA. Promise: A peer-to-peer media streaming system. In *Proceedings of the ACM Multimedia*, Berkeley, CA, November 2003.
- [59] V. HOLMEDAHL, B. SMITH, AND T. YANG. Cooperative caching of dynamic content on a distributed Web server. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, Chicago, IL, July 1998.
- [60] K. A. HUA, Y. CAI, AND S. SHEU. Patching : A multicast technique for true video-on-demand services. In *Proceedings of ACM Multimedia*, Bristol, United Kingdom, September 1998.

- [61] L. HUANG, U. HORN, F. HARTUNG, AND M. KAMPMANN. Proxy-based tcp-friendly streaming over mobile networks. In *Proceedings of the Fifth International Workshop on Wireless Mobile Multimedia*, Atlanta, GA, September 2002.
- [62] K.A. HUS AND S. SHEU. Skyscraper broadcasting: a new broadcasting scheme for metropolitan video-on-demand systems. In *Proceedings of ACM SIGCOMM*, Cannes, France, September 1997.
- [63] A. IYENGAR AND J. CHALLENGER. Improving Web server performance by caching dynamic data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, December 1997.
- [64] X. JIANG, Y. DONG, D. XU, AND B. BHARGAVA. Gnustream: A p2p media streaming system prototype. In *Proceedings of the 4th International Conference on Multimedia and Expo (ICME)*, Baltimore, MD, July 2003.
- [65] S. JIN, A. BESTAVROS, AND A. IYENGER. Accelerating Internet streaming media delivery using network-aware partial caching. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.
- [66] L. JUHN AND L. TSENG. Harmonic broadcasting for video-on-demand service. In *IEEE transactions on Broadcasting*, volume 43(3), Sept. 1997.
- [67] J. JUNG, D. LEE, AND K. CHON. Proactive Web caching with cumulative prefetching for large multimedia data. In *Proceedings of WWW*, Amsterdam, Netherland, May 2000.
- [68] J. KANGASHARJU, F. HARTANTO, M. REISSLEIN, AND K. W. ROSS. Distributing layered encoded video through caches. In *Proceedings of IEEE INFOCOM*, Anchorage, AK, April 2001.
- [69] J. I. KHAN AND Q. TAO. Partial prefetch for faster surfing in composite hypermedia. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, CA, March 2001.
- [70] T. KIM AND M. H. AMMAR. A comparison of layering and stream replication video multicast schemes. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Port Jefferson, NY, June 2001.
- [71] R. KOENEN. Overview of the mpeg-4 version 1 standard. <http://wwwam.hhi.de/mpeg-video/standards/mpeg-4.htm>, March 2001.
- [72] B. KRISHNAMURTHY AND C. E. ELLIS. Study of piggyback cache validation for proxy caches in the World Wide Web. In *Proceedings of USENIX Symposium on Internet Technology and Systems (USITS)*, Monterey, CA, December 1997.
- [73] T. M. KROEGER, D. D. E. LONG, AND J. C. MOGUL. Exploring the bounds of Web latency reduction from caching and prefetching. In *Proceedings of USENIX*

- Symposium on Internet Technology and Systems (USITS)*, Monterey, CA, December 1997.
- [74] S. LEE, W. MA, AND B. SHEN. An interactive video delivery and caching system using video summarization. In *Computer Communications*, volume 25, pages 424–435, March 2002.
  - [75] C. LIU AND P. CAO. Maintaining strong cache consistency for the World-Wide-Web. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS)*, Baltimore, MD, May 1997.
  - [76] J. LIU, X. CHU, AND J. XU. Proxy cache management for fine-grained scalable video streaming. In *Proceedings of IEEE INFOCOM*, Hong Kong, China, March 2004.
  - [77] Q. LUO, R. KRISHNAMURTHY, Y. LI, P. CAO, AND J. F. NAUGHTON. Active query caching for database Web servers. In *Proceedings of the 3rd International Workshop on the Web and Databases*, Madison, WI, June 2000.
  - [78] A. LUOTONEN, H. FRYSTYK NIELSEN, AND T. BERNERS-LEE. Cern [httpd](http://www.w3.org/Daemon/Status.html). <http://www.w3.org/Daemon/Status.html>.
  - [79] W.H. MA AND H.C. DU. Reducing bandwidth requirement for delivering video over wide area networks with proxy server. In *Proceedings of International Conferences on Multimedia and Expo (ICME)*, New York City, NY, July 2000.
  - [80] A. MENA AND J. HEIDEMANN. An empirical study of real audio traffic. In *Proceedings of IEEE INFOCOM*, Tel-Aviv, Israel, March 2000.
  - [81] Z. MIAO AND A. ORTEGA. Proxy caching for efficient video services over the Internet. In *Proceedings of Packet Video Workshop*, New York City, NY, April 1999.
  - [82] Z. MIAO AND A. ORTEGA. Scalable proxy caching of video under storage constraints. In *IEEE Journal on Selected Areas in Communications*, September 2002.
  - [83] J. NICHOLS, M. CLAYPOOL, R. KINICKI, AND M. LI. Measurements of the congestion responsiveness of windows streaming media. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, County Cork, Ireland, June 2004.
  - [84] J. PADHYE AND J. KUROSE. An empirical study of client interactions with a continuous media courseware server. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Cambridge, United Kingdom, July 1998.
  - [85] V. PADMANABHAN, H. WANG, P. CHOU, AND K. SRIPANIDKULCHAI. Distributing streaming media content using cooperative networking. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Miami, FL, May 2002.

- [86] J.F. PARIS, S.W. CARTER, AND D.D.E. LONG. Efficient broadcasting protocols for video on demand. In *Proceedings of the Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Montreal, Canada, July 1998.
- [87] J. POSTEL. User datagram protocol. <http://www.faqs.org/rfcs/rfc768.html>, August 1980.
- [88] J.-F. PRIS AND D. D. E. LONG. A variable bandwidth broadcasting protocol for video-on-demand. In *Proceedings of ACM/SPIE Conference on Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 2003.
- [89] M. RABINOVICH, Z. XIAO, F. DOUGLIS, AND C. KALMANEK. Moving edge side includes to the real edge – the clients. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, WA, March 2003.
- [90] S. RAMESH, I. RHEE, AND K. GUO. Multicast with cache (mcache): An adaptive zero-delay video-on-demand service. In *Proceedings of IEEE INFOCOM*, Anchorage, AK, April 2001.
- [91] M. REISSLEIN, F. HARTANTO, AND K. W. ROSS. Interactive video streaming with proxy servers. In *Proceedings of the First International Workshop on Intelligent Multimedia Computing and Networking*, Atlantic City, NJ, February 2000.
- [92] R. REJAIE, M. HANDELY H. YU, AND D. ESTRIN. Multimedia proxy caching mechanism for quality adaptive streaming applications in the Internet. In *Proceedings of IEEE INFOCOM*, Tel-Aviv, Israel, March 2000.
- [93] R. REJAIE, M. HANDELY, AND D. ESTRIN. Quality adaptation for congestion controlled video playback over the Internet. In *Proceedings of ACM SIGCOMM*, Cambridge, MA, September 1999.
- [94] R. REJAIE, M. HANDLEY, H. YU, AND D. ESTRIN. Proxy caching mechanism for multimedia playback streams in the Internet. In *Proceedings of International Web Caching Workshop (WCW)*, San Diego, CA, March 1999.
- [95] R. REJAIE AND J. KANGASHARJU. Mocha: A quality adaptive multimedia proxy cache for Internet streaming. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Port Jefferson, New York, June 2001.
- [96] R. REJAIE AND A. ORTEGA. Pals: Peer-to-peer adaptive layered streaming. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Monterey, CA, June 2003.
- [97] S. ROY, J. ANKCORN, AND S. WEE. Architecture of a modular streaming media server for content delivery networks. In *Proceedings of IEEE International Conference on Multimedia and Expo (ICME)*, Baltimore, MD, July 2003.



- [98] S. ROY, B. SHEN, S. CHEN, AND X. ZHANG. Empirical study of a segment-based streaming proxy in an enterprise environment. In *Proceedings of the International Workshop on Web Content Caching and Distribution (WCW)*, Beijing, China, October 2004.
- [99] S. SAROIU, K. P. GUMMADI, R. J. DUNN, S. D. GRIBBLE, AND H. M. LEVY. An analysis of Internet content delivery systems. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.
- [100] P. SCHOJER, L. BOSZORMENYI, H. HELLWAGNER, B. PENZ, AND S. PODLIPNIG. Architecture of a quality based intelligent proxy (qbix) for mpeg-4 videos. In *Proceedings of WWW*, Budapest, Hungary, May 2003.
- [101] H. SCHULZRINNE, S. CASNER, R. FREDERICK, AND V. JACOBSON. Rtp: A transport protocol for real-time applications. <http://www.ietf.org/rfc/rfc1889.txt>, January 1996.
- [102] H. SCHULZRINNE, A. RAO, AND R. LANPHIER. Real time streaming protocol (rtsp). <http://www.ietf.org/rfc/rfc2326.txt>, April 1998.
- [103] S. SEN, L. GAO, J. REXFORD, AND D. TOWSLEY. Optimal patching schemes for efficient multimedia streaming. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Basking Ridge, NJ, June 1999.
- [104] S. SEN, J. REXFORD, AND D. TOWSLEY. Proxy prefix caching for multimedia streams. In *Proceedings of IEEE INFOCOM*, New York City, NY, March 1999.
- [105] B. SMITH, A. ACHARYA, T. YANG, AND H. ZHU. Exploiting result equivalence in caching dynamic Web content. In *Proceedings of Second USENIX Symposium on Internet Technologies and Systems (USITS)*, Boulder, CO, October 1999.
- [106] R. TEWARI, A. DAN H. VIN, AND D. SITARAM. Resource-based caching for Web servers. In *Proceedings ACM/SPIE Conference on Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 1998.
- [107] F. TOBAGI, J. PANG, R. BAIRD, AND M. GANG. Streaming raid: A disk array management system for video files. In *Proceedings of the first ACM international conference on Multimedia*, Anaheim, CA, August 1993.
- [108] D. TRAN, K. HUA, AND T. DO. Zigzag: An efficient peer-to-peer scheme for media streaming. In *Proceedings of IEEE INFOCOM*, San Francisco, CA, April 2003.
- [109] E. VELOSO, V. ALMEIDA, W. MEIRA, A. BESTRAVOS, AND S. JIN. A hierarchical characterization of a live streaming media workload. In *IEEE/ACM Transactions on Networking*, September 2004.
- [110] S. VISWANATHAN AND T. IMIELINSKI. Metropolitan area video-on-demand service using pyramid broadcasting. In *Multimedia Systems*, volume 4(4), August 1996.

- [111] B. WANG, S. SEN, M. ADLER, AND D. TOWSLEY. Proxy-based distribution of streaming video over unicast/multicast connections. In *Proceedings of IEEE INFOCOM*, New York City, NY, June 2002.
- [112] Y. WANG, M. CLAYPOOL, AND Z. ZUO. An empirical study of realvideo performance across the Internet. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop (IMW)*, San Francisco, CA, November 2001.
- [113] S. WILLIAMS, M. ABRAMS, C.R. STANBRIDGE, G. ABDULLA, AND E.A. FOX. Removal policies in network caches for World-Wide-Web documents. In *Proceedings of the ACM SIGCOMM*, Stanford University, CA, August 1996.
- [114] C. E. WILLS AND M. MIKHAILOV. Towards a better understanding of Web resources and server responses for improved caching. In *Proceedings of WWW*, Toronto, Canada, May 1999.
- [115] J. L. WOLF, P. S. YU, AND H. SHACHNAI. Disk load balancing for video-on-demand systems. In *ACM Transaction on Multimedia Systems*, volume 5 (6), December 1997.
- [116] A. WOLMAN, G. VOELKER, N. SHARMA, N. CARDWELL, M. BROWN, T. LANDRAY, D. PINNEL, A. KARLIN, AND H. LEVY. Organization-based analysis of Web-object sharing and caching. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS)*, Boulder, CO, October 1999.
- [117] K. WU, P. S. YU, AND J. WOLF. Segment-based proxy caching of multimedia streams. In *Proceedings of WWW*, Hong Kong, China, May 2001.
- [118] L. XIAO, S. CHEN, AND X. ZHANG. Dynamic cluster resource allocations for jobs with known and unknown memory demands. In *IEEE Transactions on Parallel and Distributed Systems*, volume 13 (3), 2002.
- [119] L. XIAO, S. CHEN, AND X. ZHANG. Adaptive memory allocations in clusters to handle unexpectedly large data-intensive jobs. In *IEEE Transactions on Parallel and Distributed Systems*, volume 15 (7), 2004.
- [120] L. XIAO, X. ZHANG, A. ANDRZEJAK, AND S. CHEN. Building a large and efficient hybrid peer-to-peer Internet caching system. In *IEEE Transactions on Knowledge and Data Engineering*, volume 16 (6), 2004.
- [121] Z. XIAO AND K. P. BIRMAN. Providing efficient, robust error recovery through randomization. In *Proceedings of the International Workshop on Applied Reliable Group Communication (Jointly held with the 21st International Conference on Distributed Computing Systems)*, Phoenix, AZ, April 2001.
- [122] J. YIN, L. ALVISI, M. DAHLIN, AND A. IYENGAR. Engineering server-driven consistency for large scale dynamic Web services. In *Proceedings of WWW*, Hongkong, China, May 2001.

- [123] X. ZHANG, M. BRADSHAW, Y. GUO, B. WANG, J. KUROSE, P. SHENOY, AND D. TOWSLEY. Amps: A flexible, scalable proxy testbed for implementing streaming services. Technical report, Department of Computer Science, University of Massachusetts, Amherst, MA, 2004.
- [124] Z.L. ZHANG, Y. WANG, D.H.C. DU, AND D. SU. Video staging: A proxy-server based approach to end-to-end video delivery over wide-area networks. In *IEEE Transactions on Networking*, volume 8, August 2000.
- [125] H. ZHU AND T. YANG. Class-based cache management for dynamic Web content. In *Proceedings of IEEE INFOCOM*, Anchorage, AK, April 2001.

## VITA

### Songqing Chen

Songqing Chen was born in Yancheng, Jiangsu Province, China. He obtained his BS and MS in computer science from Huazhong University of Science and Technology, Wuhan, China in 1997 and 1999, respectively. Since August 1999, he started to pursue his Ph.D. in computer science at the College of William and Mary, Williamsburg, Virginia. His research interests include distributed computing and Internet systems. He is a student member of IEEE and ACM.