1-1-2017

A Scalable Solution For Interactive Video Streaming

Kamal Kayed Nayfeh
Wayne State University,

A SCALABLE SOLUTION FOR INTERACTIVE VIDEO STREAMING

by

KAMAL KAYED NAYFEH

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2017

MAJOR: COMPUTER ENGINEERING

Approved By:

____

Advisor Date

____

____

____

DEDICATION

To my wife Ahlam

To my little ones Omar, Jenna, and Ali

To my parents

ACKNOWLEDGMENTS

This research could not have been possible without the support of many people. I would like to show my sincere appreciation for my adviser Dr. Nabil Sarhan. His guidance and directions were of extreme help during my research. I also would like to thank my committee members Dr. Harpreet Singh, Dr. Lihao Xu, and Dr. Song Jiang for their precious feedback. I would like to extend my gratitude to my family who kept encouraging and supporting me on this track. Especially, my wife who had to put up with me all these years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 Overview

Due to the remarkable growth of online video and social media, video streaming has grown drastically. A wide range of applications were developed to utilize this technology such as Video- on-Demand (VOD), Live Webcasting, Web Conferencing, Distance Learning, Employee Training, Product Announcements, and many more. VOD provides the clients the ability to view a prerecorded media file at any time. Real-time entertainment, including VOD, is the largest traffic category on virtually every network and is expected to continue growing [1]. VOD is expected to replace both broadcast-based TV and DVD rentals at stores.

YouTube and Netflix are two examples of VOD applications. YouTube is currently ranked the second most popular website in the world [2] and has over a billion users, which is about a third of all people on the Internet [3]. The number of people watching YouTube per day is up 40% year over year since March 2014 [3]. More than 86 million members in over 190 countries watch more than 125 million hours per day of videos on Netflix [4]. Netflix's share of peak period downstream traffic from fixed internet connections in the US grew to 35.2% [5].

In True VOD (TVOD), all requests are serviced immediately, but this stringent requirement is hard to meet. Near VOD (NVOD) is the general case because it converges to TVOD when resources become sufficiently high. This dissertation considers the design of NVOD systems.

Since streaming media is resource intensive, media delivery faces a significant scalability challenge. The most common approaches used to address the scalability challenge are Content Distribution Networks [6] and Peer-to-Peer [7]. While the first approach requires maintaining a huge number of geographically distributed servers [8], the second still relies heavily on central servers [9, 10]. Both of these approaches mitigate the scalability problem but do not eliminate it because the fundamental problem is due to unicast delivery [10]. As multicast is highly effective in delivering high-usage content and in handling flash crowds, there has been a growing interest in enabling native multicast to support high-quality on-demand video distribution, IPTV, and other major applications [10, 11, 12]. Moreover, applying multicast delivery schemes in fully owned networks like TV cable networks, dish networks, and universities is totally applicable. We consider the multicast-based approach.

Many resource sharing techniques [13, 14, 15, 16, 17, 18, 19, 20] have been proposed to utilize the multicast facility. Batching [21] accumulates requests for the same video and services them together using multicast streams. Stream merging techniques, such as Patching [22, 15, 23] and Earliest Reachable Merge Target (ERMT) [13, 24], aggregate users into larger groups that share the same multicast streams. Unfortunately, the overwhelming majority of prior studies used simple workload, in which media objects are homogeneous in type, length, and bit rate, and are accessed sequentially from the beginning to the end without interactions (such as Pause, Resume, Jump Backward, and Jump Forward).

More recent characterization studies [25, 26, 27, 28, 29, 30, 31, 32], however, reveal that the actual workload is more complex and dynamic. In particular, the users issue many interactive operations and may access objects from points other than the beginning and may stop before the end. In addition, the user behavior varies with media type, content type, and object length. Only few studies have considered the design of interactive VOD systems, but they assumed that all requests are serviced immediately [14] or did not use stream merging [33, 34, 35, 36, 37].

Cache management was studied either at the server side of VOD systems [33, 34, 35] or at the client side in non-scalable system with no stream merging support [34]. Furthermore, existing clients even those with no multicast support do not allow for cache discontinuity. Any interactive request to outside the local cache causes the client to clear the entire cache.

As VOD systems evolve, users expect better support for advanced interactive functionality, which is difficult to accomplish especially with various content types and access patterns. For a VOD system to be successful, it must be able to handle the user interactive behavior including advanced functions such as bookmarking. Bookmarks are direct links to interesting points within the video. A video segment that is searched and watched repeatedly is called a hotspot and is pointed to by a bookmark. Examples of bookmarks within a sporting content are the start of a match, a foul, or a goal. Within a musical content, a bookmark is placed at the beginning of each piece. Previous studies [38, 39] show that video segments are not watched with the same probability. While some segments are skipped, some skimmed others are repeated and watched again and again. Some video categories may contain bookmarks more than others such as sport games and the educational lectures. Studies [38, 39] show that many users who missed a game are more likely to search for and watch the major events during the game rather than watching the whole game. To the best of our knowledge, the impact of bookmarking on the system performance is not studied in a scalable streaming framework when resource sharing techniques are employed.

1.2 Main Research Objectives

The main objectives of this dissertation can be summarized as follows.

- To design a scalable and interactive Near Video-On-Demand system.
- To develop a sophisticated client-side cache management policy that maximizes the percentage of interactive requests from the local cache without requiring additional resources from the server.
- To study the effect of bookmarking on the system performance and propose enhancements to provide better support for bookmarking.

Next, the main objectives are described in more detail.

1.3 Designing an Interactive Near Video-On-Demand system

In this project, we study the design of interactive NVOD systems that utilize stream merging for scalable delivery. The proposed design enhances customer-perceived quality-of-service (QoS) by servicing requests quickly and ensuring pleasant and smooth playbacks. It also supports user interactions and realistic access patterns with short response times and low rejection probabilities. We generalize the Split and Merge protocol [33] to work with stream merging techniques, such as Patching and ERMT. The design employs a variety of stream types, including B-Streams (full-length multicast streams), P-Streams (multicast streams for supporting Patching), and I-Streams (unicast streams for supporting interactive requests). We introduce a novel I-Stream provisioning policy. This policy determines the I-Stream threshold, which is the maximum I-Stream length (in seconds) allowed to be allocated by the server. It then adjusts the number of I-Streams and B-Streams, according to the current system requirements. By restricting the I-Stream threshold, this policy prevents any request from using an I-Stream for an extended period, thereby allowing other requests to be serviced as soon as possible. Moreover, we propose a modified request scheduling policy to address the unfairness issue in servicing unpopular videos.

1.4 Developing a sophisticated client-side cache management policy

In this project, we introduce a sophisticated client-side cache management policy to maximize the percentage of interactive requests serviced from the client's own cache, which reduces the load on the server. The policy caches data from all streams types including B-Streams, P-Streams, and I-Streams. As the cache becomes full, the policy purges data according to a purging algorithm. We develop three variations of the purging algorithm: Purge Oldest, Purge Furthest, and Adaptive Purge. Purge Oldest removes the oldest data in the cache. Purge Furthest clears the furthest data from the customer's playback point. The Adaptive Purge tries to avoid purging any data that includes the customer's playback point or any of the streams being listened to playback point. To continue to listen to streams or not for paused customers when the cache becomes full is an important decision. The proposed policy can be easily generalized to work with any VOD system, not just when stream merging techniques are utilized.

1.5 Supporting Bookmarking in Scalable and Interactive Video Streaming

In this project, we study the effect of bookmarking on the system performance. Moreover, we develop enhancements to exploit the user behavior and thus improve the customer Quality of Experience (QoE). In Particular, we study the following enhancements.

- Developing a new Purging algorithm, such that it avoids purging any hotspot data that is already cached.
- Reserving server channels to fetch hotspot data for customers not listening to any stream.
- Using multicast channels when they become available to fetch hotspot data.

1.6 Performance Evaluation Methodology

We developed a simulator for both the client-side caches or buffers and the VOD server, supporting various resource sharing and scheduling techniques. The simulator consists of the following main modules: A workload generator, which generates all the events in the system such as session start, session end, and interactive requests according to the realistic workload considered. A resource sharing module to implement the resource sharing techniques such as Batching, Patching, and ERMT. A scheduling module to determine which group of requests to admit to the system. An I-Stream provisioning module to shift resources according to the system requirements at the moment. An interactive requests handling module. A caching module to determine which data to cache and how. A purging module to clear data when the customer cache becomes full. A waiting queue for each video. A blocking queue for all blocked customers. A queuing manager to move customers between the different queues such as waiting and blocking queues. A channel allocation module to reserve the required server resources for the customers.

To make sure that results are stable, we ignore the data during the transient period, during which the queues are filled and the system is not stable. The simulator checks the stability of the results and stop after a steady state analysis with 95% confidence interval is reached.

We evaluate the system performance through extensive simulations using the following metrics: average waiting time, waiting defection probability, cache hit rate, unfairness against unpopular videos, blocking defection probability, average blocking time, blocking probability, playback point deviation, and aggregate delay. We introduce a new metric *Aggregate Delay* to quantify the average delay experienced by a client due to both waiting and blocking during a streaming session. This metric captures the increased customer frustration with subsequent blocking especially if it occurs within a short time from the last blocking. To quantify the cache management policy efficiency, we calculate the cache hit rate, which can be defined as the probability an interactive request is serviced from the cache. We define the minimum amount of data purged at any time as the Purge Block. To quantify the cache fragmentation, we introduce two new metrics: *Average Number of Segments and Average Gap Length.* The Average Number of Segments is the average number of cache segments during a streaming session. While the Average Gap Length is the average distance (gap) measured in seconds between each two consecutive segments in the customer's cache during a streaming session.

CHAPTER 2 BACKGROUND INFORMATION AND RELATED WORK

2.1 Introduction

This chapter presents an overview of previous studies related to NVOD streaming systems. We focus on providing background information about major topics related to this dissertation including resource sharing, request scheduling, support for interactive requests, cache management and bookmarking.

2.2 Resource Sharing

Numerous techniques have been proposed to deal with the scalability challenge, especially in the areas of resource sharing. Batching [21] is one of the earliest resource sharing techniques. It simply services all waiting requests for a video using one full-length multicast stream. Resource sharing techniques utilize the multicast facility and can be classified into client-pull and server-push techniques, depending on whether the channels are allocated on demand or reserved in advance, respectively. The first category includes stream merging techniques Patching [15], and Earliest Reachable Merge Target (ERMT) [13]. Clients usually need to wait before the video playback starts. In contrast, Patching expands the multicast tree dynamically to include new requests. A new request joins the latest full-length stream for the same video and receives the missing portion as a patch. When the playback of the patch is completed, the client continues the playback of the remaining portion using the data received from the full-length stream and buffered locally. When the length of the patch stream exceeds a certain dynamic threshold, delivering a full-length multicast stream becomes more cost-effective than delivering the patch. There are two Patching technique implementations. One implementation starts a new full stream when the patch stream length exceeds a certain limit, called regular window (Wr). The other implementation starts a new full stream when the accumulative patch streams length since the last full stream for a certain video exceeds the video length. We consider the latter since it produces better results for the workload used. Whereas Patching allows a stream to merge only once, ERMT allows streams to merge multiple times, resulting in a hierarchical stream merging tree. A new client or a newly merged group of clients listens to the closest stream it can merge with if no later arrivals can preemptively catch them. The target stream can later be extended to satisfy the needs of the new client (only after the merging stream finishes and merges with the target), and this extension can affect its own merge target. The three resource sharing techniques differ in complexity. Batching is the simplest to implement, since it starts a new full stream for every group of requests to a certain video. It does not allow for stream merging. Patching is next in complexity because it allows only one merge per client and allows only regular streams to be shared. ERMT is the most complex since it allows any number of merges in order to maximize resource sharing. The server should perform frequent calculations to determine the optimal next merge. Hence, selecting the appropriate resource sharing technique is a trade-off between the technique's implementation complexity and the achieved performance. The implementation complexity increases from Batching to Patching to ERMT.

The second category consists of Periodic Broadcasting techniques [40, 41, 42, 43, 44, 45], which divide each media file into multiple segments and broadcast each segment periodically on dedicated server channels. These techniques are cost-performance effective for highly popular content but lead to channel underutilization when the request arrival rate is not sufficiently high.

2.3 Request Scheduling

The server maintains a queue for every video and applies a scheduling policy to select an appropriate queue for service when server resources become available. When a new request arrives, the request is serviced immediately if there is enough server capacity. Otherwise, the request is queued with similar requests for the same video to be serviced later when server resources become available according to a *scheduling policy*. All requests for a certain video can be serviced using one channel, which can be defined as the set of required server resources (network bandwidth, disk I/O bandwidth, etc.) for delivering a multimedia stream. All scheduling policies try to optimize one of the following objectives:

- Minimize the overall client defection probability,
- Minimize the average waiting time, and
- Minimize unfairness.

The defection probability is the likelihood that a new customer leaves the server without being serviced because the waiting time exceeded the user's tolerance. It is the most important metric because it translates to the number of concurrently serviced customers and server throughput. The second and third objectives are indicators of the perceived Quality-of-Service (QoS). It is desirable that servers treat all requests for different videos equally. Unfairness measures the bias of a policy against unpopular videos. There are several studies on scheduling policies [46, 10, 47] for VOD servers. The most popular policies are *First Come First Serve* (FCFS) [46], *Maximum Queue Length* (MQL) [46], and *Maximum Cost First* (MCF) [48, 49]. FCFS selects the queue with the oldest request, and thus it is the fairest policy. MQL, however, tries to maximize the number of requests serviced with one channel by selecting the queue with the largest number of requests. In contrast, MCF selects the queue with the minimum cost in terms of the stream length. MCF-P (P for "Per"), the preferred implementation of MCF, selects the video with the least cost per request. Study [49] shows that MCF-P outperforms other scheduling policies when stream merging is used.

2.4 Support for Interactive Operations

Interactive requests can be supported using dedicated unicast streams, called *I-Streams* [34, 33]. An I-Stream is a dedicated unicast stream used to service only an interactive request. The server utilizes an I-Stream when it cannot service the interactive request by any other way. I-Streams can start from any playback point in the video and can last as long as the customer needs it. Since I-Streams are unicast, they cannot be shared with other requests. The Split and Merge (SAM) protocol [33] was proposed to service interactive requests when using Batching. As soon as a client issues an interactive request, the client is split from the multicast stream, and temporarily assigned an I-Stream to perform the interaction. Once the interaction is complete, the client is merged into an ongoing stream. For a Pause interaction, no I-Stream is required, and the user is merged into an ongoing multicast stream as soon as the user resumes.

2.5 Cache Management

Each customer allocates a certain amount of buffer for caching incoming video stream data. Any interactive request to data within the cache is serviced immediately without requiring additional resources from the server, which reduces the load on both the server and the network. Study [34] proposed a client side cache management policy when Batching is employed. The policy caches video data in a buffer located close to the client, such as in the customers' premise equipment. Interactive requests to data already cached are handled immediately. The buffer is utilized to merge a customer served by an I-Stream back with a B-Stream by fetching frames while the I-Stream is servicing the customer. The policy does not allow for buffer discontinuity. Any interactive request to data outside the buffer clears the entire buffer.

2.6 Bookmarking

Previous studies [38, 39] analyzed a number of video streaming experiments and characterized the user interactive behavior. They have shown that video segments are not watched the same. While some segments are skipped, some skimmed others are repeated and watched again and again. A video segment that is searched and watched repeatedly is called a hotspot and is pointed to by a bookmark. Some video categories may contain bookmarks more than others such as sporting games and educational lectures. Studies [38, 39] show that many users who missed a game are more likely to search for and watch the major events during the game rather than watching the whole game. Study [38] proposed a dynamic bookmark placement to move misplaced bookmarks to the appropriate positions within the video and to pre-fetch content based on predictable user interactive behavior. Study [50] proposed an automated video bookmarking algorithm.

CHAPTER 3 DESIGN OF INTERACTIVE NEAR VIDEO-ON-DEMAND SYSTEMS

3.1 Introduction

Due to the remarkable growth of online video and social media, video streaming has grown drastically. Real-time entertainment, including Video-on-Demand (VOD), is the largest traffic category on virtually every network and is expected to continue growing [1]. In True VOD (TVOD), all requests are serviced immediately, but this stringent requirement is hard to meet. Near VOD (NVOD) is the general case because it converges to TVOD when resources become sufficiently high. This part of the dissertation considers the design of NVOD systems.

Since streaming media is resource intensive, media delivery faces a major scalability challenge. Hence, many resource sharing techniques [13, 14, 15, 16, 17] have been proposed to utilize the multicast facility. Batching [21] accumulates requests for the same video and services them together using multicast streams. Stream merging techniques, such as Patching [15] (and references within) and Earliest Reachable Merge Target (ERMT) [13], aggregate users into larger groups that share the same multicast streams. Unfortunately, the overwhelming majority of prior studies used simple workload, in which media objects are homogeneous in type, length, and bitrate, and are accessed sequentially from the beginning to the end without interactions (such as Pause, Resume, Jump Backward, and Jump Forward). More recent characterization studies [25, 30, 29, 31], however, reveal that the actual workload is more complex and dynamic. In particular, the users issue many interactive operations and may access objects from points other than the beginning and may stop before the end. In addition, the user behavior varies with media type, content type, and object length. Only few studies have considered the design of interactive VOD systems, but they assumed that all requests are serviced immediately [14] or did not use stream merging [34, 33].

We study the design of interactive NVOD systems that utilize stream merging for scalable delivery. The proposed solution enhances customer-perceived quality-of-service (QoS) by servicing requests quickly and ensuring pleasant and smooth playbacks. It also supports user interactions and realistic access patterns with short response times and low rejection probabilities. We generalize the Split and Merge protocol [33] to work with stream merging techniques, such as Patching and ERMT. The solution employs a variety of stream types, including B-Streams (full-length multicast streams), P-Streams (multicast streams for supporting Patching), and I-Streams (unicast streams for supporting interactive requests). The solution includes a novel I-Stream provisioning policy. This policy determines the optimal I-Stream threshold, which is the maximum I-Stream length (in seconds) allowed to be allocated by the server. It then adjusts the number of I-Streams and B-Streams, according to the current system requirement. By restricting the I-Stream threshold, this policy prevents any request from using an I-Stream for an extended period, thereby allowing other requests to be serviced as soon as possible. Moreover, we propose a modified request scheduling policy to address the unfairness issue in servicing unpopular videos. Furthermore, we use a client-side cache management policy to maximize the percentage of interactive requests serviced from the client's own cache, thereby reducing the required server bandwidth.

We evaluate through extensive simulation the effectiveness of the proposed solution under realistic workload and using different resource sharing techniques and scheduling policies. We study the effect of a wide range of system parameters on the clients' waiting and blocking metrics. Furthermore, we develop a new metric, called *Aggregate Delay*, to quantify the average delay experienced by a client due to both waiting and blocking during a streaming session. This metric captures the increased customer frustration with subsequent blocking especially if it occurs within a short time from the last blocking.

The rest of this chapter is organized as follows. Section 3.2 presents the proposed solution. Section 3.3 discusses the performance evaluation methodology. Section 3.4 presents and analyzes the main results. Finally, Section 3.5 concludes with a summary of the results.

3.2 Proposed Solution

3.2.1 Overall System Design

As shown in Figure 3.1, the proposed system consists of a VOD server, clients streaming videos from the server, and a network connecting them. The major components of the VOD server are waiting queues (with one queue being for each video), a blocking queue, a queuing manager, an I-Stream provisioning module, an enhanced SAM protocol, a resource sharing protocol, a waiting request scheduling policy, a blocking request scheduling policy, and streams. The streams are divided into full length multicast streams (B-Streams), multicast patch streams (P-Streams), and unicast interactive streams (I-Streams). B-Streams and P-Streams are used to service waiting customers and can be shared with other requests for only the same video. An I-Stream is a dedicated unicast stream for servicing interactive requests and thus cannot be shared with other requests. The server utilizes the SAM technique to manage I-Streams. The I-Stream is freed once it finishes delivering the requested data.

The customer starts a streaming session by issuing a request to playback a certain video. The server makes the decision whether it can service the request or not based on the number of available server channels. If there are adequate server channels, the server starts streaming the video to the customer using a resource sharing technique (Batching, Patching, or ERMT). Otherwise, the customer's request is placed in the waiting queue for that video. The server applies a waiting scheduling policy (FCFS, MQL, or MCF) to determine which waiting queue to service when streams become available. Waiting customers defect if they stay in the waiting queue for too long. During a streaming session, the customer issues interactive requests to pause the video, resume the video, jump forward, and/or jump backward. The last two are by a certain distance relative to the current playback position. The customer continues to cache from all listening streams during a pause until the cache is full. Since pause and resume requests do not require any additional resources, they are serviced immediately. Jump requests either get serviced immediately (if adequate resources are available) or block. Once a customer's request blocks, the customer stops listening to all streams. The server applies a blocking scheduling policy to service blocked customers when server channels become available. The waiting and blocking scheduling policies are not necessarily the same. Since blocked requests exhibit a different aggregation behavior and the server's objective is to minimize the blocking time, the server applies the FCFS scheduling policy to blocked requests. Like waiting customers, blocking customers defect if they stay in the blocking queue for too long.

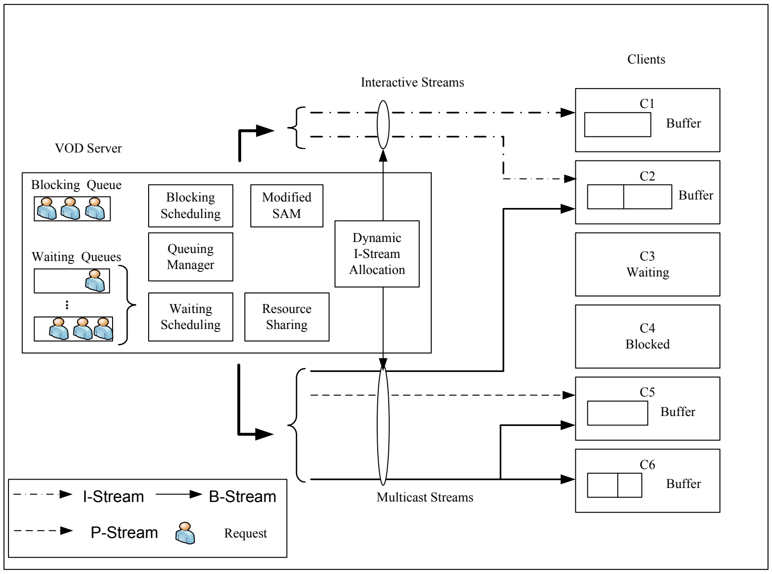


Figure 3.1: Proposed System Design

3.2.2 State Transitions

Figure 3.2 illustrates the customer's state transition diagram. The server keeps track of the customer's state to help decide when and how to service each group of customers with the same state. The customer enters the system when he/she posts the first request to playback a certain video. If there are enough server resources, the customer is serviced immediately and assigned the Play state. Otherwise, the customer is assigned a Waiting state. When the server handles a waiting request, the customer transitions to the Play state. As the customer issues interactive requests, the customer's state might change. A customer in the Play state stays in the same state until the video ends, he/she issues an interactive request, or runs out of buffer and subsequently blocks. A pause request always changes the customer's state to Pause. Similarly, a resume request sets the customer's state to Play. A jump forward/backward is either handled immediately and the customer stays in the Play state, or causes the request to block setting the customer's state to Blocked. When a blocked request is handled, the customer is returned to the Play state.

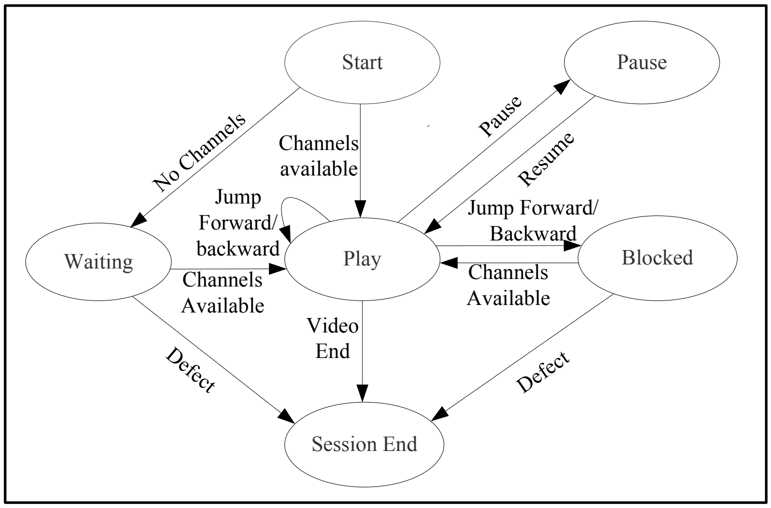


Figure 3.2: Simplified State Transition Diagram

3.2.3 Proposed Support for Interactive Requests

The server keeps track of the customer's state to help decide when and how to service each group of customers with the same state. As shown in the customer's state transition diagram in Figure 3.2, the customer enters the system when he/she posts the first request to playback a certain video. If there are enough server resources, the customer is serviced immediately and assigned the Play state. Otherwise, the customer is assigned a Waiting state. When the server handles a waiting request, the customer transitions to the Play state. As the customer issues interactive requests, the customer's state might change. A customer in the Play state stays in the same state until the video ends, he/she issues an interactive request, or runs out of buffer and subsequently blocks. A pause request always changes the customer's state to Pause. Similarly, a resume request sets the customer's state to Play. A jump forward/backward is either handled immediately and the customer stays in the Play state, or causes the request to block setting the customer's state to Blocked. When a blocked request is handled, the customer is returned to the Play state.

Figure 3.3 illustrates the methods by which the customer's jump requests are serviced.

- The customer utilizes its own cache. Each customer allocates a certain amount of memory for stream merging and for keeping old data for future use. If the target playback point falls inside the customer's cache, the interactive request is serviced from the cache without demanding additional server resources. As will be detailed in Subsection 4.2.2, the cache is organized into segments, with each segment holding a contiguous set of video data.
- The server merges the request with an already existing B-stream for the same video without using any I-Stream. For a successful merge, the target stream playback point should be within a certain tolerance, called playback point deviation tolerance ($DeviationTolerance$) from the target playback point. This tolerance is a measure of how close the requested target playback point to the actual (i.e., achieved) playback point. For example, if the playback point deviation tolerance is 10 seconds, the current playback point is 50 seconds, and the requested jump distance is $+100$ seconds, the actual playback point can be in the range $\{50 + 100 - 10, 50 + 100 + 10\} = \{140, 160\}$ seconds. The system performance improves with this tolerance, but customers appreciate smaller values.
- The server utilizes an available I-Stream for merging the request with an existing B-Stream for the same video. The server tries to minimize the skew between the target playback point and the to-be merged with B-stream playback point in order to reduce the I-Stream length. While an I-Stream is used by a certain customer, the server keeps searching for a suitable B-Stream to merge the customer

with it. Upon a successful merge, the server frees the I-Stream.

The order of the methods mentioned above depends on the interactive request type and the streams currently being used by the customer. When a customer issues a pause request, the customer continues to listen to the stream and buffer data if it is a B-Stream until the buffer is full. However, if the customer is listening to an I-Stream, the customer stops listening to the I-Stream immediately. If the interactive request cannot be serviced, the customer's request is placed in the blocking queue. The server services blocked requests by merging them with other B-Streams or by using I-Streams. Like waiting customers, blocked customers defect when the blocking time exceeds their tolerance.

3.2.4 Proposed Metric: Aggregate Delay

We introduce a new metric, called *Aggregate Delay*, which quantifies the average delay experienced by clients due to both waiting and blocking. The Aggregate Delay can be determined as follows:

$$AggregateDelay = AvgWaitingTime + BlockingWeight \times AvgBlockingTime. \tag{3.1}$$

To reflect the increased customer frustration with subsequent blocking especially if it occurs within a short time of the last blocking, we assign a weight for the blocking *BlockingWeight*, which is the multiplication of the *Occurrence Weight* and *Temporal Weight*. The *Occurrence Weight*, varying from *MinOccurWt* to *MaxOccurWt*, reflects the blocking number within the same session. The first time the customer blocks during the session, the *Occurrence Weight* is set to *MinOccurWt*. The *Occurrence Weight* is set to *MaxOccurWt* when the number of occurrences is equal to or greater than *MaxOccur*, and is linearly interpolated in between. The *Temporal Weight* is to reflect the time difference between two consecutive blocking instances. It varies from *MinTempWt* to *MaxTempWt*. It is set to *MaxTempWt* when the time difference is 0 and to *MinTempWt* when the time difference exceeds a certain time duration, called *TemporalThreshold*. The *TemporalWeight* is linearly interpolated in between. Figure 3.4 (particularly CalcAggrDelay function) illustrates the calculation of the Aggregate Delay.

3.2.5 Proposed I-Stream Provisioning Policy

Since the workload varies with time and hence the system's requirement of I-Streams changes accordingly, we propose an I-Stream provisioning policy. This policy is guided by one of two objectives.

- Minimize the *Aggregate Delay* experienced by the customer, which accounts for both the average waiting time and the average blocking time.
- Minimize the overall customer defection probability, which encompasses waiting defection and blocking defection probabilities. We primarily consider the former objective.

The proposed policy first determines the I-Stream threshold measured in seconds for the system. Subsequently, it responds to changing system requirements by adjusting the number of I-Streams and B-Streams. We generalize the SAM protocol to work with stream merging techniques (Patching and ERMT) and take advantage of the buffer used by these techniques.

The I-Stream threshold is the maximum I-Stream length allowed to be allocated by the server. Since I-Streams are unicast and hence very costly, any interactive request that could be served using an I-Stream must request it for a period less than the threshold. Otherwise, the request will be denied access to I-Streams, even when they are available. The I-Streams are intended to be used for a short period to service interactive requests that could not be serviced by any other way until they could be merged with multicast streams. By restricting the I-Stream length, this policy prevents any request from using I-Streams for an extended period and hence prevents the system performance from degrading.

Subsequently, the system adjusts the number of I-Streams dynamically in response to workload variations. Since the overall server capacity is fixed, this policy works by periodically adjusting the relative ratio of I-Streams and B-Streams in response to workload variations. Any addition to the I-Streams must be accompanied by a subtraction of the same amount to the B-Streams. Figure 3.4 (particularly

IStreamProv function) illustrates the proposed I-Streams provisioning policy. It is executed every specified period, called Adjustment Period or simply AdjustPeriod, which is the time interval between two consecutive adjustments. AdjustPeriod could be kept small to respond quickly to workload variations in rapidly changing workloads, or it could be set to a moderate value in more stable workloads to prevent the server from over-reacting to temporary workload changes. Let us first explain the procedure for adjusting the I-Stream threshold. The average waiting time, blocking time and Aggregate Delay are calculated for the current AdjustPeriod. The system adjusts the threshold in step increments (AdjustStep) every AdjustPeriod and monitors the Aggregate Delay. If the Aggregate Delay becomes better (i.e. smaller) in the current AdjustPeriod compared to the test, the server continues adjusting in the same direction (increase/decrease threshold) as it did in the previous AdjustPeriod. Otherwise, the server reverses the adjustment direction. The AdjustStep has to be big enough to make a difference on the Aggregate Delay every AdjustPeriod and small enough such that it does not cause the threshold to oscillate. We found experimentally that 10 seconds is an appropriate step value. MAXADJUST is the maximum value that could be added or subtracted in one AdjustPeriod. The server limits the adjustment amount each AdjustPeriod to prevent the I-Streams from rapidly oscillating back and forth, which degrades the system performance. The procedure for adjusting the number of I-Streams is similar but employs a parameter, called AdjustFactor, which determines the amount of change in the Aggregate Delay corresponding to the amount of change in the I-Streams. It could be set to a more aggressive value for rapidly changing workloads and it could be kept at a moderate value for steady workloads.

3.2.6 Cache Management

We implement a *cache management* policy to maximize the client cache utilization. The client's cache consists of one or more data segments, with each segment holding a contiguous set of video data. Both future and past data are utilized to service interactive requests. As the cache becomes full, the oldest data in the cache is purged. When two segments overlap, they are merged to form one contiguous segment.

All stream types are used to fill the cache. With Patching, for example, a client simultaneously receives data from both a B-Stream and a P-Stream and caches them in two different segments. Any interactive request for data inside either segment is serviced from the cache without requesting any additional server resources. The two segments keep growing until the P-Stream is finished, at which time they overlap and get merged into one contiguous segment.

The cache management policy is explained and studied in great details in Chapter 4.

3.3 Performance Evaluation

3.3.1 Simulation Platform

We have developed a simulator for both the client-side caches/buffers and the VOD server, supporting various resource sharing and scheduling techniques. The simulator allows for the evaluation of a wide range of system parameters including server capacity, scheduling policy, resource sharing technique, cache size, I-Streams number, I-Stream threshold, request rate, playback deviation tolerance, number of videos, I-Stream provisioning adjust period, and I-Stream provisioning adjust factor. To generate each point, we run the simulation a minimum of 450,000 customers and corresponding interactive requests, accounting for about one million seconds of simulated time. To make sure that results are stable, we ignore the data during the transient period, which is set to 10% of the total simulation time. The simulator checks the stability of the results and stops after a steady state analysis with 95% confidence interval is reached.

3.3.2 Client, Server, and Workload Characteristics

Table 4.1 summarizes the default parameters used. We characterize the waiting and blocking tolerance of customers with a Poisson process with a mean of 30 seconds. We examine the server at different loads by varying server capacities from 294 Gbps to 540 Gbps. Only 10% of the server capacity is reserved for I-Streams. We use MCF-PN scheduling policy for waiting customers and FCFS for blocking customers except when evaluating the effect of scheduling policies. MCF-PN is a new scheduling policy that we introduce and will be defined in Section V.C. By default, the playback deviation point tolerance is kept at 10 seconds. The default client-side cache size dedicated to the streaming application is 200 MB.

3.3.3 Workload Characteristics

The default average arrival rate (λ) is 30 requests per minutes. We utilize the results of [25], which characterizes the workload of a media streaming server using actual access logs. This workload determines the distributions of all requests, including interactive requests, for various videos. It shows a strong relationship between the video file duration and interactive operations and also shows a correlation between consecutive interactive operations. The results of that study are consistent with another study [30], but the latter is less detailed and did not allow for the generation of a full synthetic workload. We study 100 videos of varying lengths and request rates. The workload indicates that 91% of the requested files have average bit-rates of 300-350 kbps. The average bit-rates of the remaining files are in the range of 200-300 kbps. To be more reflective of recent video and future bit-rates, we use 4.9 Mbps for the first group and 4.2 Mbps for the second.

3.3.4 Performance Metrics

We consider the following performance metrics: *waiting defection probability*, *average waiting time*, *blocking defection probability*, *average blocking time*, *blocking probability*, and *unfairness* against unpopular videos, and playback point deviation. The terms "blocking" refers to cases when interactive requests cannot be serviced immediately. The first two blocking-related metrics are analogous to the two waiting-related metrics, but pertain to servicing the interactive requests instead of the initial play-

Table 3.1: Default Parameters Values

Parameter	Default Value(s)
Arrival Rate	30 Requests / minute
Number of Videos	100
Waiting Tolerance Model	Poisson with mean = 30 sec.
Blocking Tolerance Model	Poisson with mean = 30 sec.
Server Capacity	294 - 540 Gbps
Video Bit-rate	$4.2 - 4.9$ Mbps
I-Streams	10% of server capacity
Buffer Size	200 MByte
Playback Point Deviation Tolerance	10 seconds
Waiting Scheduling Policy	MCF-PN
Blocking Scheduling Policy	FCFS

back requests. Blocking probability is the likelihood that an interactive request blocks. The waiting defection probability can be defined as the probability that the customer leaves the server because the waiting time exceeded the customer's tolerance. Similarly, the blocking defection probability can be defined as the probability that the customer leaves the server without finishing watching the video because the customer stayed in the blocking queue longer than its tolerance. Video unfairness quantifies the bias against unpopular videos and is equal to $\sqrt{\sum_{i=1}^{M}(d_i - d)^2/(M-1)}$, where M is the number of videos, d_i is the defection rate of video i, and d is the mean video defection rate. The average waiting time is the average time the customer stays in the waiting queue. Likewise, the average blocking time is the average time the interactive request stays in the blocking queue. The blocking probability is defined as the likelihood of an interactive request to block.

For the Aggregate Delay, we use the following default values: *MinOccurWt=2, MaxOccurWt=4, MaxOccurr=5, MinTempWt=1, MaxTempWt=3, and TemporalThreshold=5 minutes.*

3.4 Result Presentation and Analysis

We analyze the system performance through extensive simulation. We primarily consider the NVOD server model. The NVOD model converges to TVOD when the server capacity becomes sufficiently high. (average waiting time = 0, waiting defection probability= 0, average blocking time = 0, blocking defection probability= 0). We focus on the issues introduced by realistic interactive workloads (blocking metrics) since this is an area that has not been investigated by previous studies.

3.4.1 Impact of I-Stream Threshold

Figure 3.5 illustrates the effectiveness of the I-Stream threshold on the system performance. When the threshold is set to a small value, most requests are denied access to the I-Streams and hence the I-Streams are severely underutilized. As we increase the threshold, the I-Stream utilization increases and hence the blocking metrics improve until we reach a certain threshold (about 150 seconds), and then the blocking metrics start to worsen. As the threshold becomes too high, a small percentage of requests hold the I-Streams for too long preventing other requests from the opportunity of using I-Streams and leading to the degradation of the blocking metrics. We observe this behavior at all server capacities. Since I-Streams are utilized to service interactive requests, the I-Stream threshold does not have a significant impact on the waiting metrics. The Aggregate Delay follows the same trend as the blocking metrics because it accounts for both average waiting and blocking times and the I-Stream threshold has little impact on the waiting metrics.

3.4.2 Effectiveness of the Proposed I-Streams Provisioning Policy

Figures 3.6 and 3.7 illustrate the effectiveness of the proposed I-Stream provisioning policy using Patching at different AdjustPeriod and AdjustFactor values, respectively. The proposed policy outperforms the static allocation of I-Streams in terms of the blocking metrics and the average Aggregate Delay experienced by the client. Up to 45% improvement in the average Aggregate Delay is achieved. At lower server capacity, dynamically adjusting resources shows a little improvement because the system favors one group of customers over another. Thus, either waiting or blocking customers suffer. Moreover, there is a cost associated with the process of switching resources. However, at higher server capacity, which is the preferred range of operation, this policy shows very significant improvements in the system performance. Using this policy with Batching and ERMT techniques exhibits similar results, and thus these results are not shown. The AdjustPeriod and AdjustFactor should be selected based on prior analysis (as that in Figures 3.6 and 3.7). (They can also be adjusted dynamically in a manner similar to the I-Stream threshold and the number of I-Streams in the proposed I-Stream Provisioning policy.)

3.4.3 Impact of the Scheduling Policy

Figure 4.8 compares various scheduling policies. MCF-P outperforms other scheduling policies in terms of the average waiting time and waiting defection probability because it minimizes the cost per request. Since shorter videos are more popular and tend to have shorter patches than longer videos, MCF-P is biased towards shorter popular videos and unfair against longer videos. As longer videos are less popular and receive more interactive requests than shorter videos, they tend to block more often with longer blocking times. MCF-P also outperforms other scheduling policies in terms of blocking metrics because it admits relatively shorter videos requests, which tend to block less with shorter blocking times.

At lower server capacity, MCF-P admits a high percentage of shorter videos, which tend to block less with shorter blocking times. Hence, the blocking metrics are kept relatively small. As the server capacity increases, more resources become available to entertain requests for longer videos. Thus, the blocking metrics worsen with the server capacity. This trend continues until we reach a server capacity where the relative number of requests for longer videos does not increase. After that, the blocking metrics start improving.

The main disadvantage of MCF-P is its unfairness as illustrated in Figure 3.8(c). To overcome the unfairness of MCF-P, we experiment with normalizing the stream length required to service a request by the video length. We call this normalized scheduling policy MCF-PN. MCF-PN takes into consideration the number of requests for each video and the stream length required to service a request divided by the video length. This normalization addressed the fairness issue, while delivering decent performance in terms of waiting, blocking, and Aggregate Delay metrics, as illustrated in Figure 4.8.

3.4.4 Impact of the Client Cache Size

Figure 5.4 demonstrates the effect of client cache size on the waiting and blocking metrics. The main results can be summarized as follows.

- As we increase the client cache size, there is little influence on the waiting metrics. Since the system tries to serve interactive requests from the client's own cache, the client cache plays an important role only after the system serves the client.
- Obviously, increasing the client cache leads to a lower blocking probability and smaller blocking time.
- The system performs well even with only 200 MB of cache, and cache sizes larger than 500 MB provide diminishing returns.
- Devices with available buffers smaller than 100 MB are expected to have relatively high blocking probability (more than 10%) if the server capacity is not increased. Most contemporary client devices, including smart phones, are expected to be served well, considering that not all the cache has to be in the main memory. With higher server capacity, the server can better entertain low-end devices.
- Interestingly, the average waiting time tends to be a little shorter for extremely small client caches. This behavior can be explained as follows. With a very small cache, the blocking defection rate is very high, causing many customers to stop listening to their current streams, thereby leading to many streams with no listeners. Because the server frees all such streams, these streams become available for waiting customers, leading to reduced waiting defection probability and average waiting time.

3.4.5 Impact of the Resource Sharing Techniques

As shown in previous studies, ERMT outperforms other resource sharing techniques in terms of waiting metrics, followed by Patching, as illustrated in Figure 5.9. Since we consider only B-Streams for merging when servicing blocked customers, Batching outperforms the other techniques in terms of the blocking defection and the blocking time because all Batching streams are B-Streams. However, Batching performs poorly in terms of waiting metrics. Patching is the best compromise among all three techniques. It performs well in terms of blocking, waiting, and Aggregate Delay metrics, and it has lower implementation complexity than ERMT.

3.4.6 Impact of Video Bitrate

Figure 3.11 demonstrates the impact of video bitrate on the system performance. At higher video bitrates, the average waiting time follows the same trend as that at lower bitrates, but at higher server capacity. However, the average blocking time and blocking probability are significantly higher at higher video bitrates, when keeping the cache size the same for all video bitrates, as the cache can hold smaller video durations at higher video bitrates. Since the cache plays a significant role in servicing interactive requests, the blocking metrics increase significantly at higher bitrates.

3.5 Conclusions

We have proposed a scalable solution for interactive NVOD systems and have analyzed its performance under realistic workload and using various resource sharing techniques and scheduling policies. The main results can be summarized as follows.

- ERMT outperforms other techniques in terms of waiting metrics. Unlike previous studies, which studied NVOD servers using simple workloads, at low to moderate request rates, the amount of improvement over Patching is insignificant. We conclude that there are fewer opportunities for stream merging in the presence of interactive requests. Given that ERMT is of higher implementation complexity, we recommend using Patching for systems with low to moderate request rates.

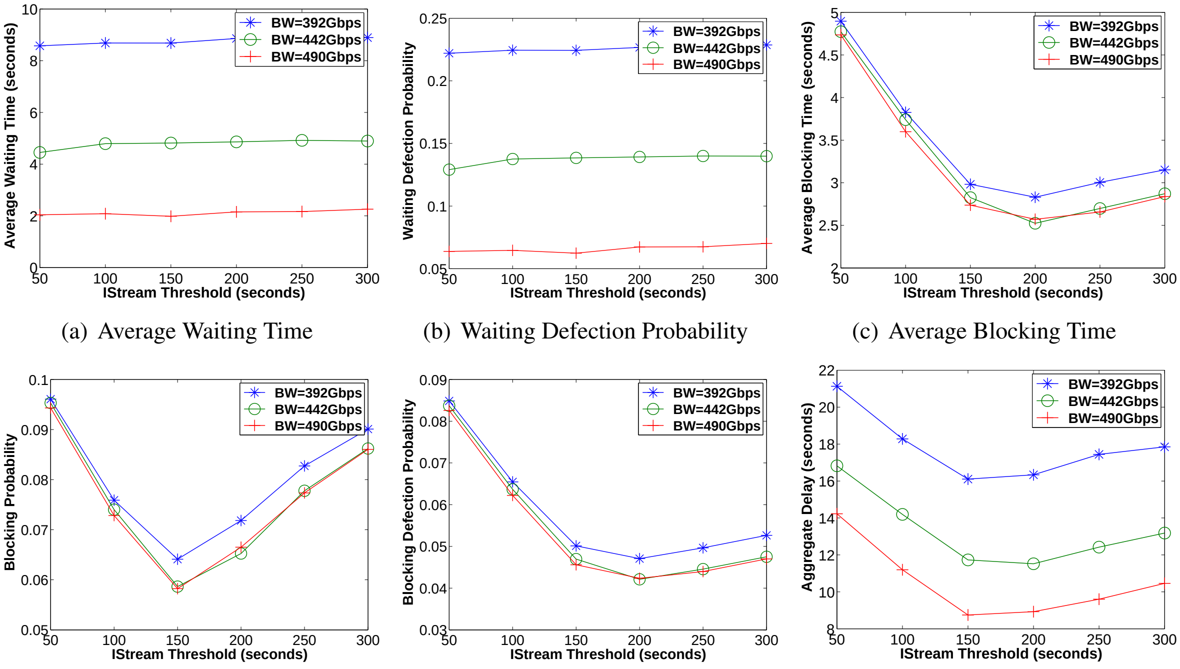
- Blocking is a major issue for any NVOD server. An efficient NVOD server should continuously monitor the system performance and shift resources in order to minimize blocking. The proposed I-Stream provisioning policy determines first the I-Stream threshold and then adjusts the number of I-Streams dynamically based on the current system state. This policy reduces the blocking probability by up to 65% and the blocking defection by up to 50%. Additionally, up to 45% reduction in the average aggregate delay experienced by the client is achieved.
- I-Streams should be constrained in length to give a chance to all interactive requests for using them. Otherwise, a few interactive requests use the I-Streams in an inefficient manner, degrading system performance. The optimum I-Stream length is system dependent.
- Deciding which group of waiting customers to serve has a significant effect on the NVOD server performance. The proposed scheduling policy (called MCF-PN) addresses the unfairness issue of MCF-P while delivering decent performance. We recommend using this policy for its performance and fairness in all systems.
- With efficient cache management, up to 87% of all interactive requests are serviced from the client's own buffer without requiring additional server resources.

```
**ServiceJumpRequest**() {
   //1. Try to service request from client cache
   //Loop through all client cache segments
  **for** $(s = 0; s < NumSegments; s++)\{$
     //System keeps track of segment start and end times in sec.
     //Target is Target Playback point in sec.
     **if** $(Target \leq Seg[s].end$ and $Target \geq Seg[s].start)$ {
        Service request from segment(s) ;
        break ; }  //if
   }  //for
   //2. Try to merge with another B-Stream
   $MinSkew = MAXFLOAT$ ;  //Initialize minimum skew
   $ClosestStream = -1$ ;  //closest to Target Playback point
   //Loop through all B-Streams for same video
   **for** $(i = 0; i < NumStreams; i++)\{$
      $Skew = |Target - Stream[i].Playback|$; //calc. skew
      //Choose the stream with minimum skew
      **if** $(Skew < DeviationTolerance) and (Skew < MinSkew)$ {
         $ClosestStream = i$;  //this is the closest stream
         $MinSkew = Skew$; }  //this is the minimum skew
   } //for
   **if** $(ClosestStream \geq 0)$ {  //Found a stream to merge with
      Client stops listening to current stream;
      Merge client request with ClosestStream;
      **if** $(stream[current].NumListeners == 0)$
         //If Current stream has no listeners, then free it
         free current stream;
      break ; } //if
   //3. Try to service request with an I-Stream
   //Check if there are any available I-Streams
   **if** $(UsedIStreams < AvailableIStreams)\{$
      $MinSkew = MAXFLOAT$ ; $ClosestStream = -1$ ;
      //Loop through all Streams for same video
      **for** $(v = 0; v < NumStreams; v++)\{$
         $Skew = |Target - streams[v].Playback|$ ;  //calc. skew
         **if** $(Skew < MinSkew)$ {
            $ClosestStream = v$; //this is the closest stream
            $MinSkew = Skew$; }  } //for
   } //if
   **if** $(ClosestStream \geq 0)$ {  //Found a stream to merge with
      Client stops listening to current stream;
      Client starts listening to ClosestStream stream;
      Client starts listening to I-Stream;
      **if** $(streams[current].NumListeners == 0)$
         free current stream ;
      break ;} //if
   //4. Block
   **else**{
      Client stops listening to all streams;
      Place request in blocking queue; }
} //ServiceJumpRequest
```

Figure 3.3: Simplified Algorithm for Servicing Jump Forward and Jump Backward Requests

```
IStreamProv() { // Run the I-Stream Provisioning Policy
if $(TransientPeriod)\{$
    //Direction determines to increase or decrease threshold
    $Direction_{Prev} = 1$ ; //Initialize Direction
    CalcAggrDelay() ; //Calculate current Aggregate Delay
    $Diff = AggrDelay_{Curr} - AggrDelay_{Prev}$;
    if $(Diff < 0)$ //Aggregate Delay improved
        $Direction_{Curr} = Direction_{Prev}$; //Keep same Direction
    else //Aggregate delay worsened
        $Direction_{Curr} = -1 * Direction_{Prev}$; //Reverse Direction
    //Calculate I-Stream threshold adjustment amount
    $AdjustAmnt = ceil(Diff) * ADJUSTSTEP$;
    if $(AdjustAmnt > MAXADJUST)$
        $AdjustAmnt = MAXADJUST$; //Cap adjustment amount
    $Threshold = Threshold + Direction_{Curr} * AdjustAmnt$;
    $AggrDelay_{Prev} = AggrDelay_{Curr}$;
    $Direction_{Curr} = Direction_{Prev}$ ;
} //transient period
else { //Adjust number of IStreams after transient period
    $Direction_{Prev} = 1$ ; //Initialize Direction
    CalcAggrDelay() ; //Calculate Aggregate Delay
    $Diff = AggrDelay_{Curr} - AggrDelay_{Prev}$;
    if $(Diff < 0)$ //Aggregate Delay improved
        $Direction_{Curr} = Direction_{Prev}$; //Keep same Direction
    else //Aggregate Delay worsened
        $Direction_{Curr} = -1 * Direction_{Prev}$; //Reverse Direction
    $AdjustAmnt = (Diff/AggrDelay_{Curr}) * AdjustFactor$;
    if $(AdjustAmnt > MAXADJUST)$
        $AdjustAmnt = MAXADJUST$; //Cap adjustment amount
    //Apply change to the number of B-Streams and I-Streams
    $BStreams = BStreams + Direction_{Curr} * AdjustAmnt$;
    $IStreams = IStreams - Direction_{Curr} * AdjustAmnt$;
    $AggrDelay_{Prev} = AggrDelay_{Curr}$;
    $Direction_{Curr} = Direction_{Prev}$ ; } //else
} //IStreamProv
CalcAggrDelay() { // Calculate the Aggregate Delay
    //Accumulate Total Waiting Time (TotWaitingTime) of waiting customers
    for $(c = 0; c < NumWaiting; c++)$
        $TotWaitingTime+ = WaitingTime[c]$; }
    //Loop through all blocking customers
    for $(c = 0; c < NumBlkng; c++)\{$
        //Calculate time difference between current occurrence blocking
        //start time and previous occurrence blocking end time
        if $(Occur[c] > 1)$ //Occurrence number during session
            $TimeDiff = BlockStartTime_{Curr} - BlockEndTime_{Prev}$;
        else
            //If this is the first blocking occurrence, use Playback point
            $TimeDiff = PlayBack[c]$;
        //Calculate Temporal Weight (TempWt)
        $TempWt = MaxTempWt * TimeDiff/TemporalThreshold$;
        if $(TempWt > MaxTempWt)$ $TempWt = MaxTempWt$;
        if $(TempWt < MinTempWt)$ $TempWt = MinTempWt$ ;
        $TempWt = MinTempWt + MaxTempWt - TempWt$;
        //Calculate Occurrence Weight (OccurWt)
        $OccurWt = (MinOccurWt * (Occurr - 1)/(MaxOccurr - 1)$;
        $OccurWt = MinOccurWt + OccurWt$;
        if $(OccurWt > MaxOccurWt)$ $OccurWt = MaxOccurWt$ ;
        //Accumulate Total Blocking Time (TotBlkngTime)
        $TotBlkngTime+ = OccurWt * TempWt * BlkngTime[c]$;
    } //for
    //Calc. Average Waiting Time. NumWaiting is # of Waiting Customers
    $AvgWaitingTime = TotWaitingTime/NumWaiting$;
    //Calc. Average Blocking Time. NumBlkng is # of Blocking Customers
    $AvgBlkngTime = TotBlkngTime/NumBlkng$;
    //Calculate current Aggregate Delay
    $AggrDelay_{Curr} = AvgWaitingTime + AvgBlkngTime$;
} //CalcAggrDelay
```

Figure 3.4: Simplified Algorithm for I-Stream Provisioning



(a) Average Waiting Time

(b) Waiting Defection Probability

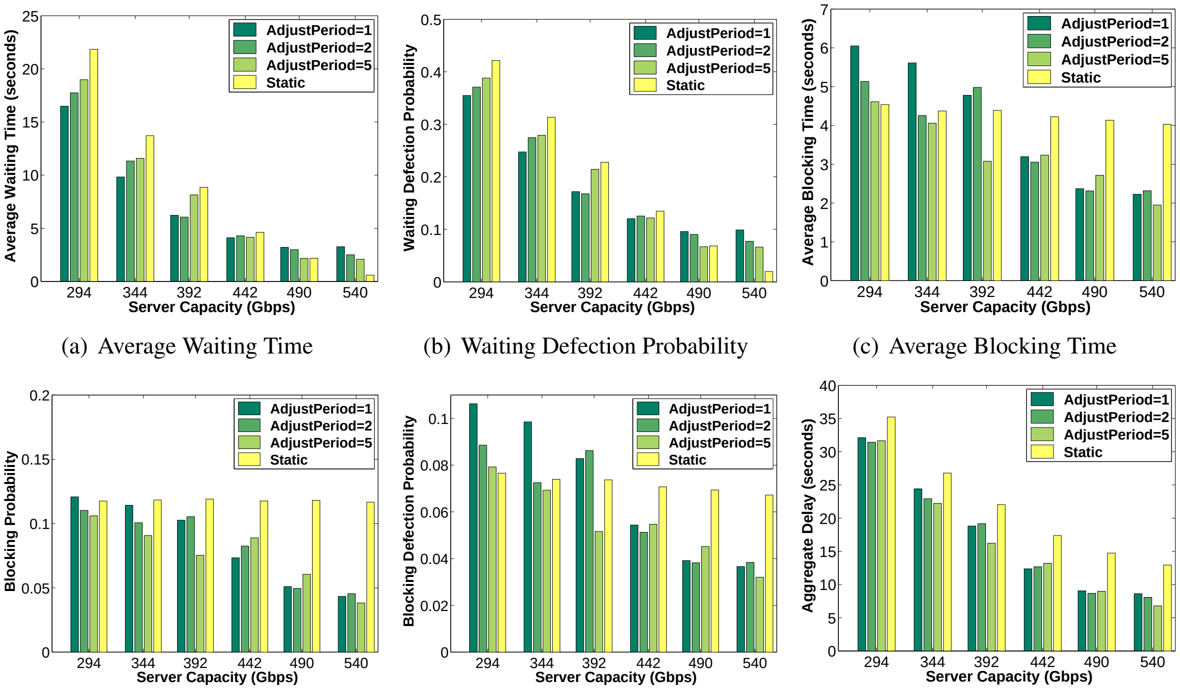
(c) Average Blocking Time

(d) Blocking Probability

(e) Blocking Defection Probability

(f) Aggregate Delay

Figure 3.5: Impact of I-Stream Threshold [BW is the Server Capacity]



(a) Average Waiting Time

(b) Waiting Defection Probability

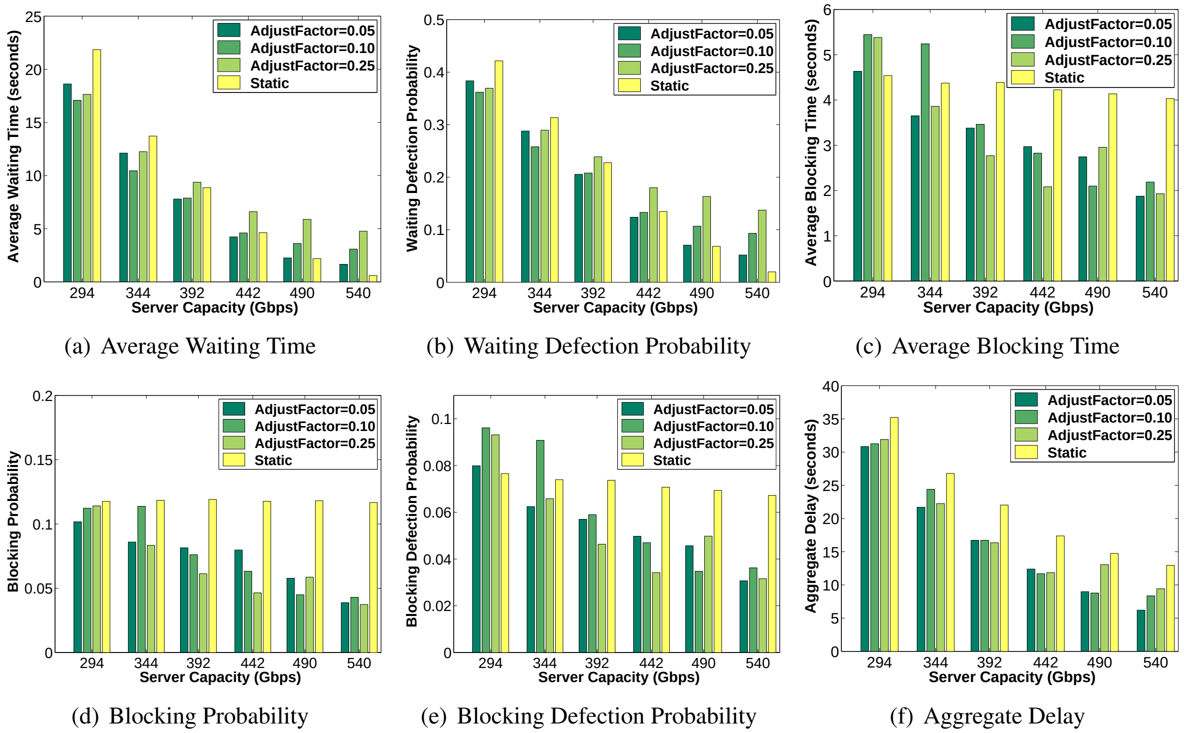
(c) Average Blocking Time

(d) Blocking Probability

(e) Blocking Defection Probability

(f) Aggregate Delay

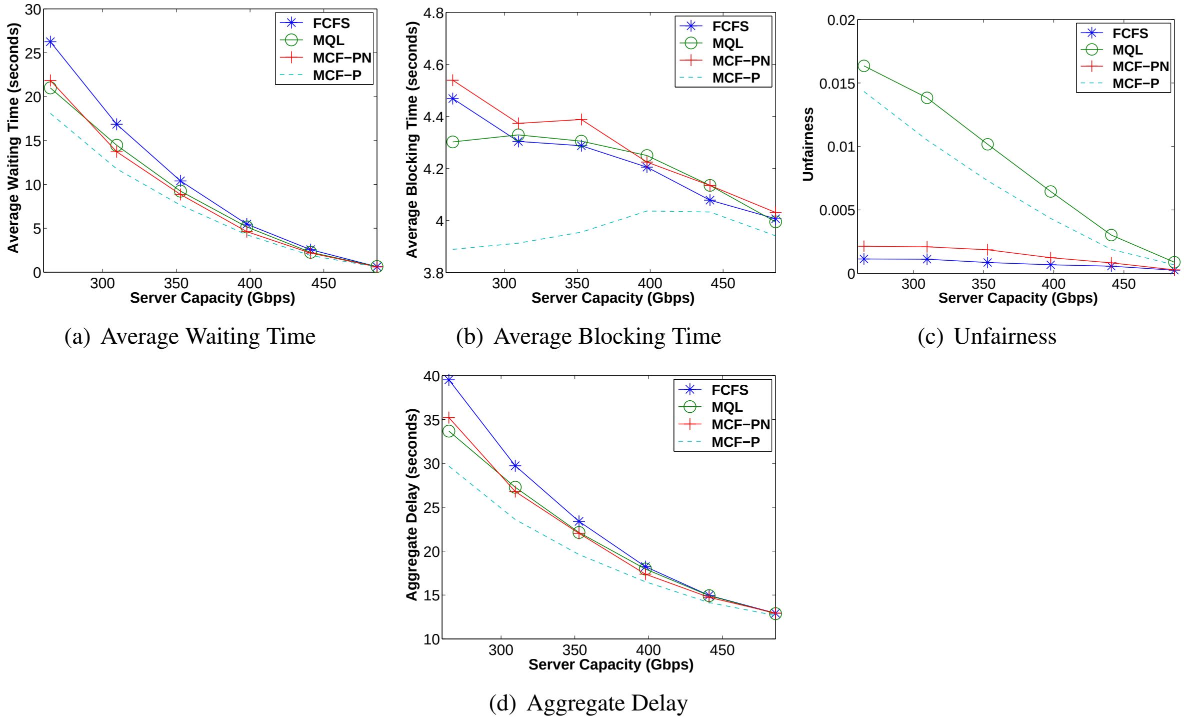
Figure 3.6: Effectiveness of I-Stream Provisioning Policy



(a) Average Waiting Time (b) Waiting Defection Probability (c) Average Blocking Time

(d) Blocking Probability (e) Blocking Defection Probability (f) Aggregate Delay

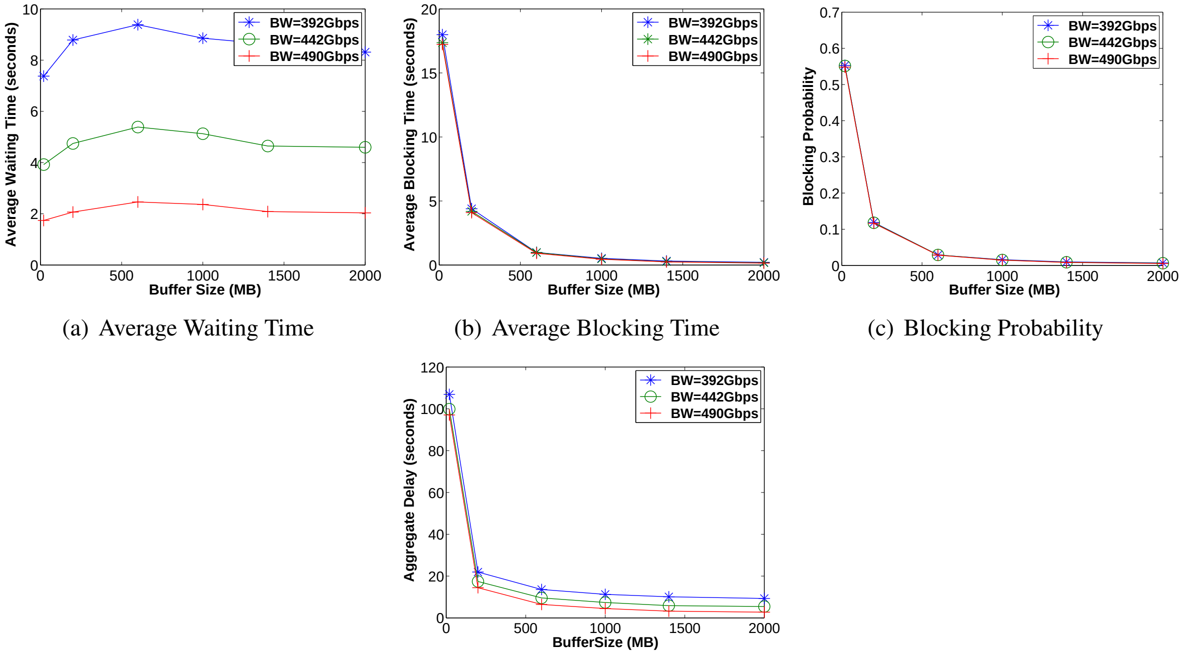
Figure 3.7: Effectiveness of I-Stream Provisioning Policy with Different AdjustFactor Settings



(a) Average Waiting Time (b) Average Blocking Time (c) Unfairness

(d) Aggregate Delay

Figure 3.8: Comparing Effectiveness of Scheduling Policies



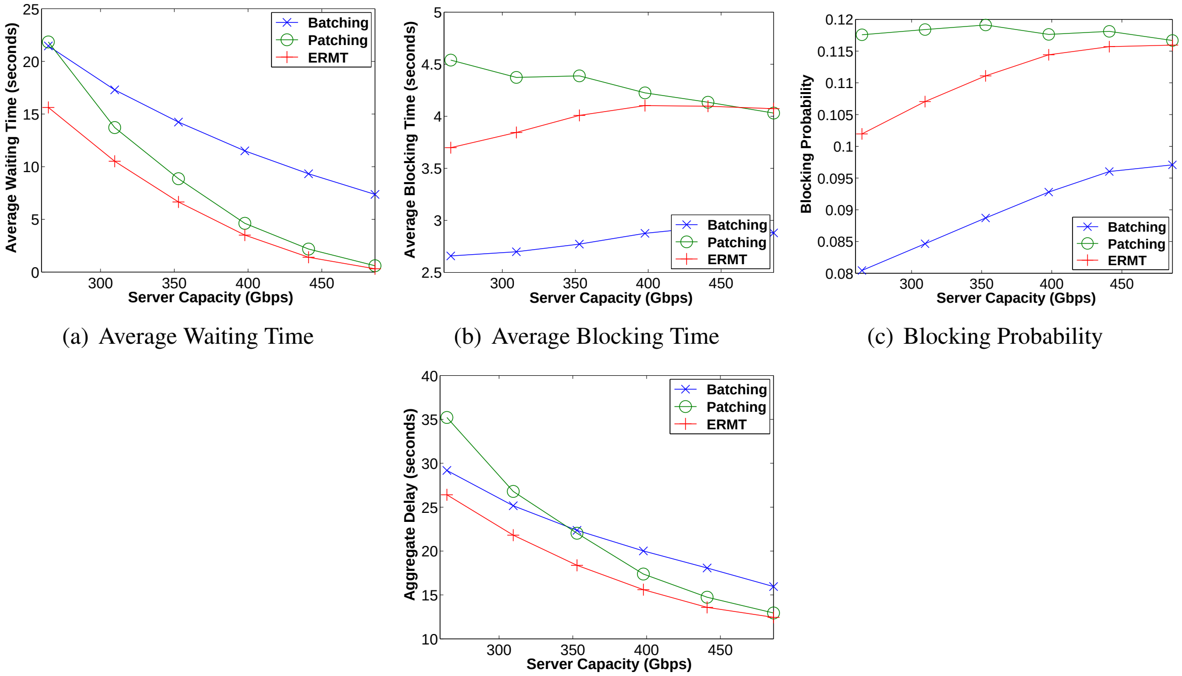
(a) Average Waiting Time

(b) Average Blocking Time

(c) Blocking Probability

(d) Aggregate Delay

Figure 3.9: Impact of Client Cache Size [BW is the server capacity]



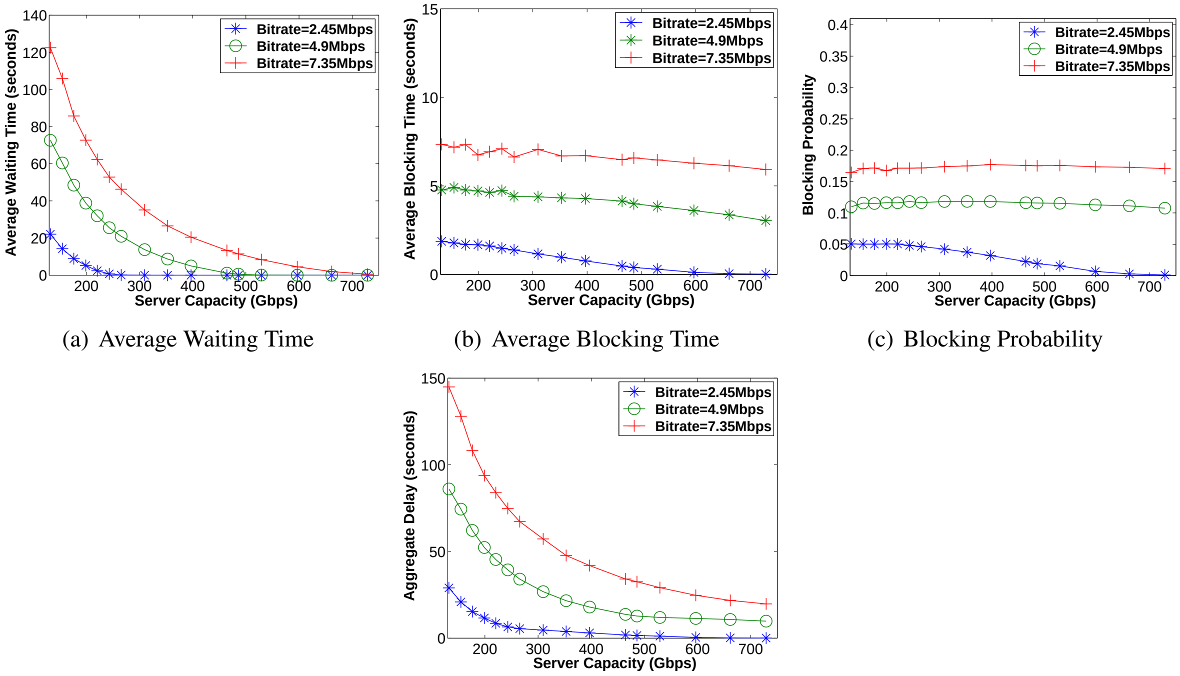
(a) Average Waiting Time

(b) Average Blocking Time

(c) Blocking Probability

(d) Aggregate Delay

Figure 3.10: Comparing Effectiveness of Resource Sharing Techniques



(a) Average Waiting Time

(b) Average Blocking Time

(c) Blocking Probability

(d) Aggregate Delay

Figure 3.11: Impact of Video Bitrate

CHAPTER 4 CLIENT-SIDE CACHE MANAGEMENT FOR VIDEO STREAMING

4.1 Introduction

The design of interactive Video-on-Demand (VOD) systems supporting stream merging is highly challenging. Although most prior studies assumed simple workload, characterization studies [25, 30, 31] show that the workload contains many interactive requests and the customer behavior varies with video length. Further design complications happen in practical systems, which cannot service all requests immediately. This model of operation is referred to as Near VOD (NVOD) and is the general case of True VOD (TVOD). Study [51] is the only study of interactive NVOD with stream merging, but deals only with the server side.

In the first part of the dissertation we considered the design of a VOD server, but in this part we focus on the client. This dissertation is the first that considers the problem of client-side cache management in interactive NVOD system employing stream merging. Cache management was studied either at the server side of VOD systems [33, 34, 35] or at the client side in non-scalable system with no stream merging support [34]. We propose a novel cache management policy, which allows and exploits cache discontinuity. Existing systems even those with no multicast support, such as YouTube, do not allow for such discontinuity. Any interactive request to outside the local cache causes the client to clear the entire cache. The policy caches data from all streams types. As the cache becomes full, the policy purges data according to a purging algorithm. We present three purging algorithms: *Purge Oldest*, *Purge Furthest*, and *Adaptive Purge*. Purge Oldest removes the oldest data in the cache, whereas Purge Furthest clears the furthest data from the client's playback point. In contrast, Adaptive Purge tries to avoid purging any data that includes the customer's playback point or the playback point of any stream that is being listened to by the client. We experiment with another important decision, which is whether pausing users should continue to listen to streams when the cache becomes full.

We evaluate the effectiveness of the proposed cache management policy and purging algorithms under realistic and complex workload through extensive simulations. We analyze many metrics, including waiting and blocking metrics, aggregate delay, cache hit rate, and cache fragmentation. The aggregate delay is the average delay experienced by a client due to both waiting and blocking, incorporating the impact of increased customer frustration with every subsequent blocking during the same session. To quantify cache fragmentation, we introduce two metrics: *average number of segments* and *average gap length between consecutive segments*. The first is the average number of cache segments during a streaming session. Whereas the second is the average distance (in seconds) between each two consecutive segments in the customer's cache during a session. We consider the impact of cache size, purge block size, and server capacity.

The main contributions of this part of the dissertation can be summarized as follows.

- We propose a novel client-side cache management policy that maximizes the percentage of interactive requests from the local cache without requiring additional resources from the server. The proposed policy can be easily generalized to work with any VOD system, not just when stream merging is utilized.
- We introduce three purging algorithms.
- We extensively evaluate the effectiveness of the proposed policy in terms of various metrics under realistic workload, considering several important parameters.

The rest of this chapter is organized as follows. Section 4.2 presents the proposed policy. Section 4.3 discusses the performance evaluation methodology. Section 4.4 presents and analyses the main results. Finally, Section 4.5 concludes with a summary of the results.

4.2 Proposed Solution

4.2.1 Considered System

The system considered is the same as in Chapter 3. Specifically, it consists of a NVOD server, streaming clients, and a network connecting them. The major components of the NVOD server are waiting queues (with one queue for each video), a blocking queue, a queuing manager, a dynamic I-Stream allocation module, a SAM technique, a stream merging protocol, a waiting scheduling policy, a blocking scheduling policy, and streams. The streams are divided into B-Streams, P-Streams, and I-Streams. B-Streams and P-Streams are used to service waiting customers and can be shared with other requests for the same video.

The customer starts a streaming session by issuing a request to playback a certain video. The server decides whether to service the request based on the availability of server channels. A channel is the set of required server resources for delivering a video stream. The number of such channels is called *server capacity*. If adequate channels are available, the request is delivered using a stream merging technique (Patching or ERMT); otherwise, the request is placed in the waiting queue for that video. The server applies a waiting scheduling policy to determine which waiting queue to service when streams become available. The most popular policies are *First Come First Serve* (FCFS) [46], *Maximum Queue Length* (MQL) [46], and *Maximum Cost First* (MCF) [49] and references within. FCFS selects the queue with oldest request and is thus the fairest, whereas MQL tries to maximize the number of requests that can be serviced with one channel by selecting the queue with the largest number of requests. In contrast, MCF selects the queue with the minimum cost in terms of stream length.

During a streaming session, the customer issues interactive requests to pause the video, jump forward, or backward by a certain amount relative to the current playback position. Interactive requests are serviced in the manner discussed in Subsection 3.2.3. The server applies a blocking scheduling policy to service blocked customers when server channels become available. The waiting and blocking scheduling policies are not necessarily the same. Waiting and blocked customers defect when the waiting/blocking time exceeds their tolerance.

4.2.2 Cache Management

The main contribution of this part of the dissertation is to propose an intelligent client-side cache management policy to maximize the cache utilization for servicing interactive requests by allowing and exploiting cache discontinuity. The client's cache consists of one or more segments of data. Each segment is a contiguous set of video data. The data kept in the cache can be future data for stream merging techniques and past data the customer watched already. All stream types including full-length B-Streams, Patch Streams (P-Streams), and Interactive Streams (I-Streams) are used to fill the cache. Both future and past data are utilized to service interactive requests. When two segments overlap, they are merged to form one segment.

A cache hit happens when an interactive request is serviced from the cache. Hence, the *cache hit rate* is defined as the probability that the interactive request is serviced from the cache. To assess the cache discontinuity, we introduce two new metrics: the *average number of segments* and *the average gap length*. The first is the customer's average number of segments during a streaming session, whereas the second is the average length (in seconds) between each two consecutive segments during a customer's session. Figure 4.1 shows an example of a customer's cache for a system employing Patching. The cache consists of two segments, which are currently being filled from two different streams. The customer is listening to two streams at time *i*: one patch stream (Stream1) delivering the current data, and a full-length stream (Stream2) delivering future data. The customer's playback point is the same as that of Stream1. At time *i+1*, both segments have cached data. The two segments overlap at time *i+2*, and thus get merged into one bigger segment encompassing the data in both. At the same time, Stream1

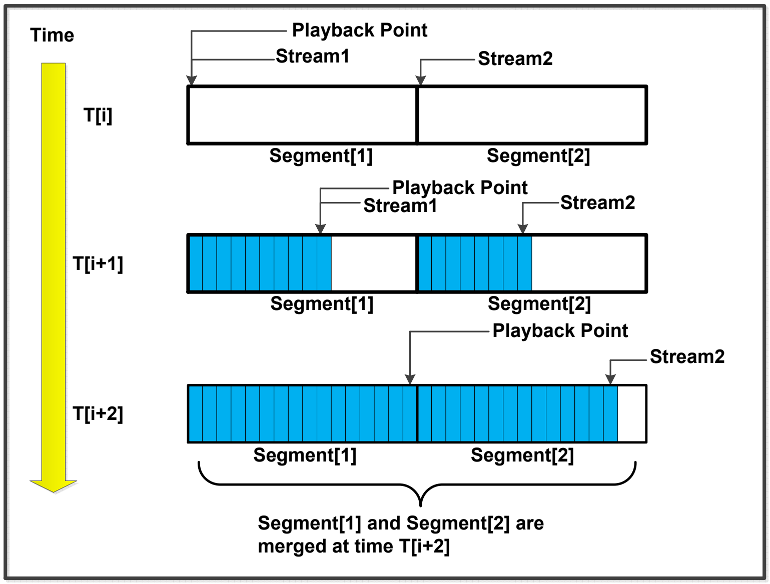


Figure 4.1: Clarification of Cache Structure: Example 1

finishes delivering all its data. Any interactive request changes the customer's playback point, and thus it becomes different from the stream(s) that it is listening to, if it is not already so. Figure 4.2 shows another example of a customer's cache for a system employing Patching. The customer's cache consists of four segments. Two are being filled from the two streams being listened to (Stream1 and Stream2). The other two are past and future data kept for servicing interactive requests. Any interactive request changes the customer's playback point. Hence, it becomes different from one of the streams playback point. Figure 4.3 illustrates the cache management policy for customers in the Pause and Play states. Since Jump requests are either serviced immediately or cause the customer to block, the caching algorithm does not deal with jump requests and has no Jump state. Jump requests, however, cause a shift in the customer's playback point within the cache.

When the cache becomes full, data is purged according to a *purging algorithm*. To avoid fragmenting the cache, the data is either purged from the segment tail (i.e., the oldest data in the segment) or head (i.e., the most recent data). The data is purged in the unit of *Purge Block*, which we define as the minimum amount of data that can be purged at one time. We present three purging algorithms *Purge*

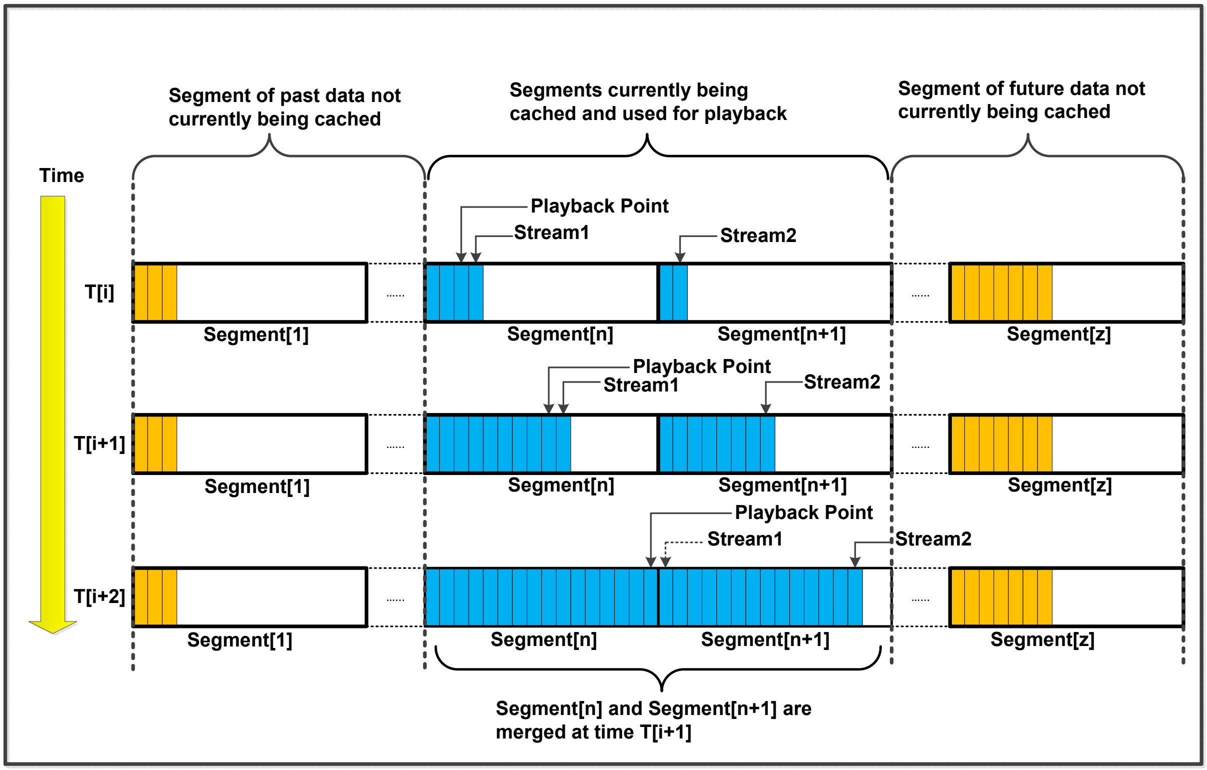

Segment of past data not currently being cached
Segments currently being cached and used for playback
Segment of future data not currently being cached
Time
T[i]
T[i+1]
T[i+2]
Playback Point
Stream1
Stream2
Segment[1]
Segment[n]
Segment[n+1]
Segment[z]
Segment[n] and Segment[n+1] are merged at time T[i+1]

Figure 4.2: Clarification of Cache Structure: Example 2

Oldest, *Purge Furthest*, and *Adaptive Purge*. Purge Oldest purges the tail of the oldest segment in the customer's cache, whereas Purge Furthest purges the furthest data from the customer's playback point. The furthest data could be the segment's tail or head. In contrast, Adaptive Purge tries to avoid purging data that includes the customer's playback point or the playback point of any stream that is currently being listened to by the client.

Figure 4.4 illustrates how the values of three conditions are computed in three scenarios. In the first scenario, Segment A does not include the client's playback point or the playback point of any stream that is being listened to by the client. Hence, the three conditions are set to PURGE_BOTH. Segment B has the customer playback point and Stream1 playback point at the head of the segment. Hence, both Condition 1 and Condition 2 are set to PURGE_TAIL. Condition 3 is set to PURGE_BOTH because Stream2 playback point is not within Segment B. The algorithm loops through all segments

```
//Handle all paused customers
HandlePause(Customer) {
  //Is customer listening to I_Stream?
  if ($Customer.Stream == I\_STREAM$) {
    //Stop listening to I-Stream and stop caching
    StopListeningToIStream(Customer);
    StopCaching(Customer); }
  //Is cache less than max allowed size?
  if ($CacheSize \geq MAX\_CACHE\_SIZE$) {
      //Should we continue to listen to streams?
      if ($C2L == Yes$) {
        KeepListeningToStream(Customer);
        KeepCaching(Customer); }
      else {
        //Stop caching and stop listening to stream
        StopCaching(Customer);
        StopListeningToStream(Customer); } }
  AdjustCache(Customer); } //HandlePause
HandlePlay(Customer) { //Handle all playing customers
    AdjustCache(Customer); } //HandlePlay
AdjustCache(Customer) {
  //Is customer listening to first stream?
  if ($Customer.Stream1 == Yes$) {
    //Advance the cache data in segment 1
    AdvanceCache(Customer, segment1); } //if
  //Is customer listening to second stream?
  if ($Customer.Stream2 == Yes$) {
    //Advance the cache data in segment 2
    AdvanceCache(Customer, segment2); } //if
  for ($s = 0; s < NumSegments; s++$){
    if ($Segment[s] overlaps Segment[s+1]$) {
      Merge(Segment[s], Segment[s+1]); } }
  PurgeAlg(Customer); } //AdjustCache
```

Figure 4.3: Simplified Algorithm for Cache Management

starting from oldest. If all the aforementioned conditions allow for purging of the segment tail, that tail is purged; otherwise, the algorithm purges the furthest data from the customer's playback point. If the purged data is enough to keep the cache size smaller than the maximum size, the algorithm stops; otherwise, it moves to the next segment.

We introduce and analyze a new option, called *Continue-to-Listen* (C2L), specifying whether the pausing user continues to listen to all multicast streams and to purge data according to the purging algorithm or stops listening to all streams when the cache gets full.

4.3 Performance Evaluation Methodology

We developed a simulator for an interactive NVOD system, including both the clients with caches and the server. The system supports various stream merging and scheduling techniques. The simulator

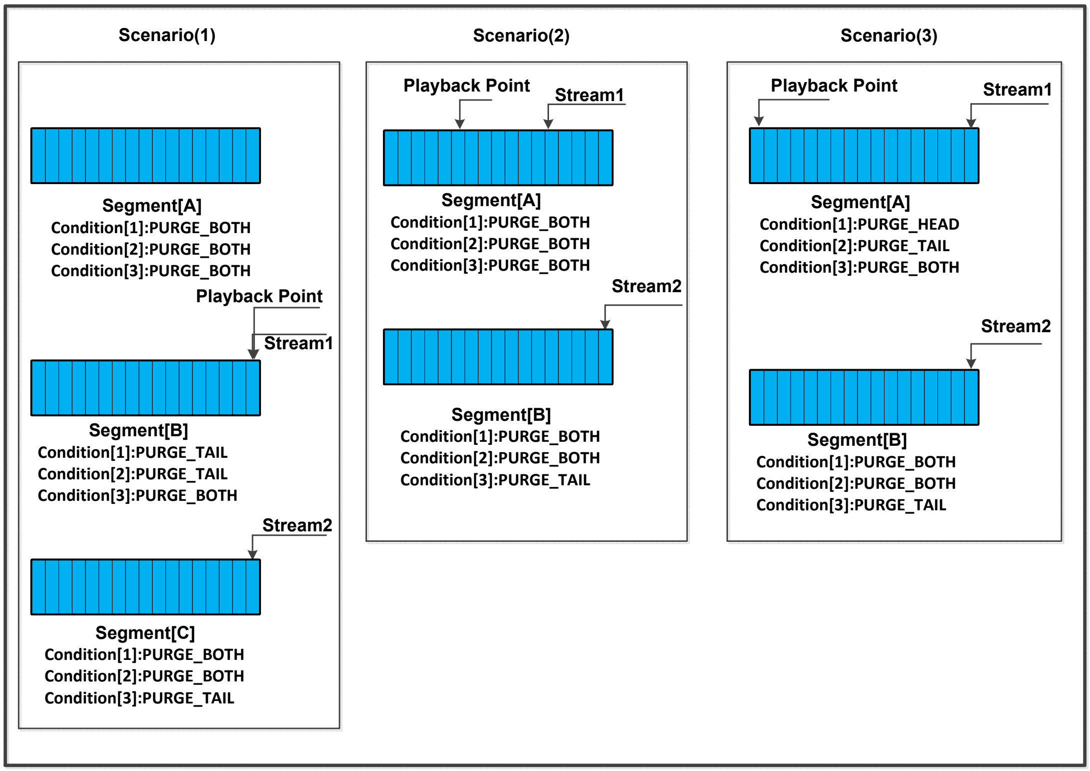


Figure 4.4: Illustration of Adaptive Purge Algorithm

stops after a steady state analysis with 95% confidence interval is reached. We experimented with a wide range of system parameters, but only the main results are shown.

4.3.1 Client and Server Characteristics

Table 4.1 summarizes the default parameters used. We characterize the waiting and blocking tolerance of customers as Poisson with a mean of 30 seconds. We examine the server at different loads by varying server capacities from 196 Gbps to 320 Gbps. Only 10% of the server capacity is reserved for I-Streams [51]. We use the FCFS scheduling policy for waiting and blocking customers to avoid any issues of unfairness. The playback deviation point tolerance is kept at 10 seconds. The default client side cache size dedicated to the video streaming application is kept at 50 MB except when evaluating the cache size.

4.3.2 Workload Characteristics

We utilize the results of [25], which characterizes the workload of a media streaming server using actual access logs. This workload determines the distributions of all requests, including interactive requests, for various videos. It shows a strong relationship between the video file duration and interactive operations and also shows a correlation between consecutive interactive operations. The results of that study are consistent with another study [30], but the latter is less detailed and did not allow for the generation of a full synthetic workload. The average arrival rate (λ) is 30 requests per minutes. We study 100 videos of varying lengths and request rates. The workload indicates that 91% of the requested files have average bit-rates of $300 - 350$ Kbps. The average bitrates of the remaining files are in the range of $200 - 300$ Kbps. To be more reflective of recent video bitrates, we use 2.1 Mbps for the first group and 2.45 Mbps for the latter.

Table 4.1: Default Parameters Values

Parameter	Default Value(s)
Arrival Rate	30 Requests / minute
Number of Videos	100
Waiting Tolerance Model	Poisson with mean = 30 sec.
Blocking Tolerance Model	Poisson with mean = 30 sec.
Server Capacity	196 - 320 Gbps
Video Bit-rate	$2.1 - 2.45$ Mbps
I-Streams	10% of server capacity
Cache Size	50 MByte
Playback Point Deviation Tolerance	10 seconds

4.3.3 Performance Metrics

We consider the following **performance metrics**: *average waiting time*, *waiting defection probability*, unfairness against unpopular videos, playback point deviation, *average blocking time*, *blocking defection probability*, *blocking probability*, and *aggregate delay*. The waiting defection probability is defined as the probability that the customer leaves the server because the waiting time exceeded the customer's tolerance. Likewise, the blocking defection probability is defined as the probability that the customer leaves the server without finishing watching the video because the customer stayed in the blocking queue longer than its tolerance. The average waiting time is the average time the customer stays in the waiting queue. Similarly, the average blocking time is the average time the interactive request stays in the blocking queue. The blocking probability is defined as the likelihood of an interactive request to block. The playback point deviation is a measure of how close the requested target playback point to the actual playback point. The aggregate delay is the average delay experienced by a client due to both waiting and blocking. To reflect the increased customer frustration, the weight is increased for every subsequent blocking during the same session [51].

4.4 Result Presentation and Analysis

We analyze the system performance through extensive simulation. We primarily consider the NVOD server model. The NVOD model converges to TVOD when the server capacity becomes sufficiently high. We focus on the issues introduced by realistic interactive workloads (blocking metrics) since this is an area that has not been investigated by previous studies.

4.4.1 Impact of Purging Algorithm

Figure 5.5 illustrates the effect of the purging algorithm on the system performance when C2L is not considered. The Adaptive Purge and Purge Oldest algorithms significantly outperform Purge Furthest in terms of blocking metrics. Since the Adaptive Purge defaults to the purging the oldest data first, the performance of the Adaptive Purge is close to the Purge Oldest. The purging algorithm affects the cache of already admitted customers. Hence, there is no significant difference between the purging algorithms in terms of waiting metrics.

The cache hit rate decreases with server capacity. Since the system tries first to merge the interactive request with another multicast (B-Stream or P-Stream) for customers not listening to any stream and there is a higher likelihood the system would find a multicast stream at higher server capacity, the merge percentage for interactive requests increases with server capacity. Hence, the cache hit rate decreases.

The cache fragmentation also increases with the server capacity as shown by the average number of segments and the average gap length. As the server capacity increases, a higher percentage of interactive requests are serviced by merging with other streams. The playback points of these streams are more likely to be outside the client cache. Since the client caches data from all streams, the probability of caching more segments increases. The cache hit rate is higher for Adaptive Purge and Purge Oldest than Purge Furthest at all server capacities because Purge Furthest is more likely to purge future data that will be needed in the future. Purge Oldest and Adaptive Purge, however, avoid purging future data. Although not shown, the Aggregate Delay improves with server capacity because it incorporates both waiting and blocking times and both improve with increased server capacity.

Figure 5.7 illustrates the performance of the purging algorithms when C2L is employed. Adaptive Purge outperforms Purge Oldest and Purge Furthest in terms of the average blocking time. Both Adaptive Purge and Purge Oldest outperform Purge Furthest in terms of average waiting time. Both the cache hit rate and the average number of segments do not change significantly with the server capacity for all purging algorithms. Figure 4.8 illustrates the impact of purging algorithm and C2L. The Adaptive Purge and the Purge Oldest consider the C2L approach, while Purge Furthest does not. The Adaptive Purge outperforms the other two in terms of blocking metrics, followed by Purge Oldest. When the customers continue to listen to streams, they keep caching the most recent data, which helps in servicing interactive requests, which tend to reference data close to the customer's playback point. Purge Furthest outperforms Adaptive Purge and Purge Oldest in terms of waiting metrics. As the cache gets full for pausing customers, they stop listening to streams and there is a higher probability these streams have no listeners. Since the system frees all such streams, they become available for waiting customers. Hence, the waiting metrics improve significantly. The cache hit rate of the Adaptive Purge is the highest because it avoids purging any data that includes the customer's playback point. The aggregate delay shows an interesting trend. At the lower server capacity, the Purge Furthest outperforms the other two purging algorithms. However, at higher server capacity, which is the preferred region of operation, the Purge Oldest and Adaptive Purge outperform the Purge Furthest algorithm.

4.4.2 Impact of Cache Size

Figure 5.6 illustrates the impact of the cache size on the waiting and blocking metrics. In the figure, Adaptive Purge uses C2L, whereas Purge Oldest and Purge Furthest do not. Both waiting and blocking metrics improve with the cache size for all purging algorithms. The cache hit rate improves with cache size until it reaches a certain limit beyond which no significant improvement can be achieved due to diminishing returns. Hence, the blocking metrics improve. The waiting metrics improve because the system can serve longer patches, which can fit in the larger cache. Hence, the relative percentage of P-Streams to B-Streams increases with cache size. Consequently, the system utilizes the smaller more efficient P-Streams than the more expensive B-Streams. We notice that Adaptive Purge with C2L significantly outperforms the other two algorithms in terms of blocking metrics because when the pausing customer continues to listen to all streams, there are more future data in the cache. The future data becomes very useful when the customer resumes playback, thereby improving the cache hit rate as shown in Figure 4.9(f). At a smaller cache size, Purge Oldest and Purge Furthest outperform Adaptive Purge in the waiting metrics. At a larger cache size, however, Adaptive Purge outperforms the other two algorithms.

4.4.3 Impact of Scheduling Policy

Figure 4.10 illustrates the impact of scheduling algorithms. MCF-P outperforms the other two scheduling algorithms in terms of waiting metrics because it minimizes the cost per request. Since shorter videos are more popular and have shorter patches, MCF-P is biased toward shorter videos. MCF-P outperforms the other two in terms of blocking metrics also. At lower server capacity, MCF-P admits a high percentage of shorter videos, which tend to block less with shorter blocking times. Hence, the blocking metrics are kept relatively small. As the server capacity increases, more resources become available to entertain requests for longer videos. Thus, the blocking metrics worsen with the server capacity. This trend continues until we reach a server capacity where the relative number of requests for longer videos does not increase. After that, the blocking metrics start improving. The main disadvantage of MCF-P is its unfairness as illustrated in Figure 4.10(f). To overcome the unfairness of MCF-P, we experiment with normalizing the stream length required to service a request by the video length. We call this normalized scheduling policy MCF-PN. MCF-PN takes into consideration the number of requests for each video and the stream length required to service a request divided by the video length. This normalization did not address the unfairness issue satisfactory.

4.4.4 Impact of Purge Block Size

Finally, Figure 5.4 illustrates the impact of the purge block size on the system performance. The blocking metrics keep improving with purge block size until the size reaches a certain value (about 50 seconds), after which it starts to worsen. This behavior is explained by the fact that the average number of cache segments per customer keeps decreasing until it reaches the block size of 50 seconds, after which it starts to increase. The aggregate delay does not change much until a certain purge block size (about 60 seconds), after which it starts to increase with purge block size.

4.5 Conclusions

The main results can be summarized as follows.

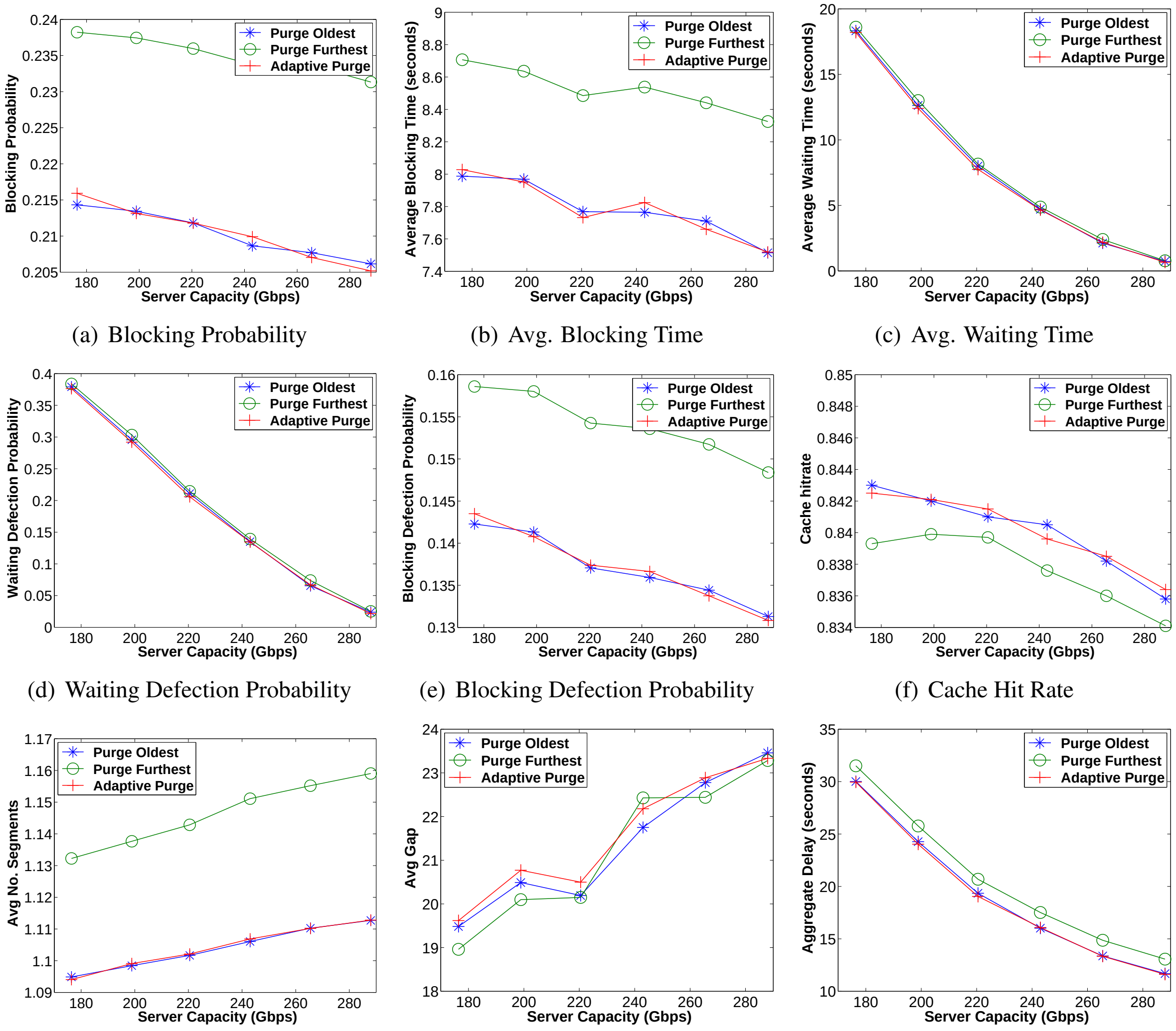
- Using our proposed cache management policy, up to 92% of all interactive requests are serviced from the client's own cache without requiring additional server resources.
- The overwhelming majority of jump backward and resume interactive requests are serviced from the cache. The jump forward interactive requests cache hit rate is significantly lower than the other two

interactive requests because some requested data is future data that is never cached.

- Determining which data to purge when the cache becomes full has a significant impact on the cache hit rate and blocking metrics. Purging the oldest data improves cache hit rate and blocking metrics. To further enhance the performance, we experimented with an adaptive purging algorithm. Interestingly, examining the additional conditions does not provide considerable improvements over purging the oldest data in specific cases.
- Choosing the cache purge block size has a significant impact on the blocking and waiting metrics. The optimal block size is system dependent.
- Increasing the cache size improves both waiting and blocking metrics until the cache size reaches a certain limit, beyond which, no significant improvement can be achieved due to diminishing returns.
- Another important decision is whether pausing customers continue to listen (C2L) to streams when the cache becomes full. Enabling C2L is highly desirable, especially for sufficiently large caches.

```
AdaptivePurge(Customer) {
  int Cond1, Cond2, Cond3;
  $CacheSize = CalcCacheSize(Customer)$;
  if $((CacheSize - MAX\_CACHE\_SIZE) \geq 0)$ {
    for $(s = 0; s < NumSegments; s++)\{$
      //Is customer's Playback Point inside segment tail or head
      if $(CustPlay \leq Seg[s].start)$ and $(CustPlay \geq Seg[s].end)$
        $Cond1 = PURGE\_BOTH$;
      else if $(CustPlay \leq Seg[s].start)$
        $Cond1 = PURGE\_TAIL$;
      else if $(CustPlay \geq Seg[s].end)$
        $Cond1 = PURGE\_HEAD$;
      //Is First Stream Playback Point inside segment tail or head
      if $(Strm1Play \leq Seg[s].start)$ and $(Strm1Play \geq Seg[s].end)$
        $Cond2 = PURGE\_BOTH$;
      else if $(Strm1Play \leq Seg[s].start)$
        $Cond2 = PURGE\_TAIL$;
      else if $(Strm1Play \geq Seg[s].end)$
        $Cond2 = PURGE\_HEAD$;
      //Is Second Stream Play Point inside segment tail or head
      if $(Strm2Play \leq Seg[s].start)$ and $(Strm2Play \geq Seg[s].end)$
        $Cond3 = PURGE\_BOTH$;
      else if $(Strm2Play \leq Seg[s].start)$
        $Cond3 = PURGE\_TAIL$;
      else if $(Strm2Play \geq Seg[s].end)$
        $Cond3 = PURGE\_HEAD$; } }
  CalcPurgeData(Customer);
} //AdaptivePurge
CalcPurgeData(Customer) {
  for $(s = 0; s < NumSegments; s++)\{$
    //Find which segment to purge
    $PurgeSeg = GetOldestSegment(Customer)$
    //Determine how much to purge
    $Extra = CacheSize - MAX\_CACHE\_SIZE$;
    $PurgeData = PURGE\_BLK * ceil(Extra/PURGE\_BLK)$;
    //Purge the segment tail
    Purge(Customer,PURGE_TAIL, PurgeData, PurgeSeg); } //for
  //Calculate the new cache size
  $CacheSize = CalcCacheSize(Customer)$;
  //Is cache size within limit
  if $((CacheSize - MAX\_CACHE\_SIZE) \leq 0)$ return;
  for $(s = 0; s < NumSegments; s++)\{$
    //Find the furthest segment from the customer playback
    //And whether to purge the segment head or tail
    $PurgeSeg = GetFurthestSegment(Customer, HeadTail)$
    //Determine how much data to purge
    $Extra = PURGE\_BLK * ceil(Extra/PURGE\_BLK)$;
    //Purge the segment
    DoPurge(Customer,HeadTail, Extra, PurgeSeg);
  $CacheSize = CalcCacheSize(Customer)$;
  if $((CacheSize - MAX\_CACHE\_SIZE) \leq 0)$ return; } }
DoPurge(Customer, HeadTail, Extra, s) {
  //Purge the segment head or tail?
  if $(HeadTail == PURGE\_TAIL)$ and $(Cond1 == PURGE\_TAIL)$
  and $(Cond2 == PURGE\_TAIL)$ and $(Cond3 == PURGE\_TAIL)$ {
    //Purge from the segment tail
    if $(Seg[s].Size \geq Extra)$
      $Seg[s].start = Seg[s].start + Extra$
    else
      Remove(Seg[s]); } //if
  if $(HeadTail == PURGE\_HEAD) and (Cond1 == PURGE\_HEAD)$
  and $(Cond2 == PURGE\_HEAD)$ and $(Cond3 == PURGE\_HEAD)$ {
    //Purge from the segment head
    if $(Seg[s].Size \geq Extra)$
      $Seg[s].start = Seg[s].end - Extra$
    else
      Remove(Seg[s]); }} //DoPurge
```

Figure 4.5: Simplified Algorithm for Adaptive Purging Algorithm



(a) Blocking Probability

(b) Avg. Blocking Time

(c) Avg. Waiting Time

(d) Waiting Defection Probability

(e) Blocking Defection Probability

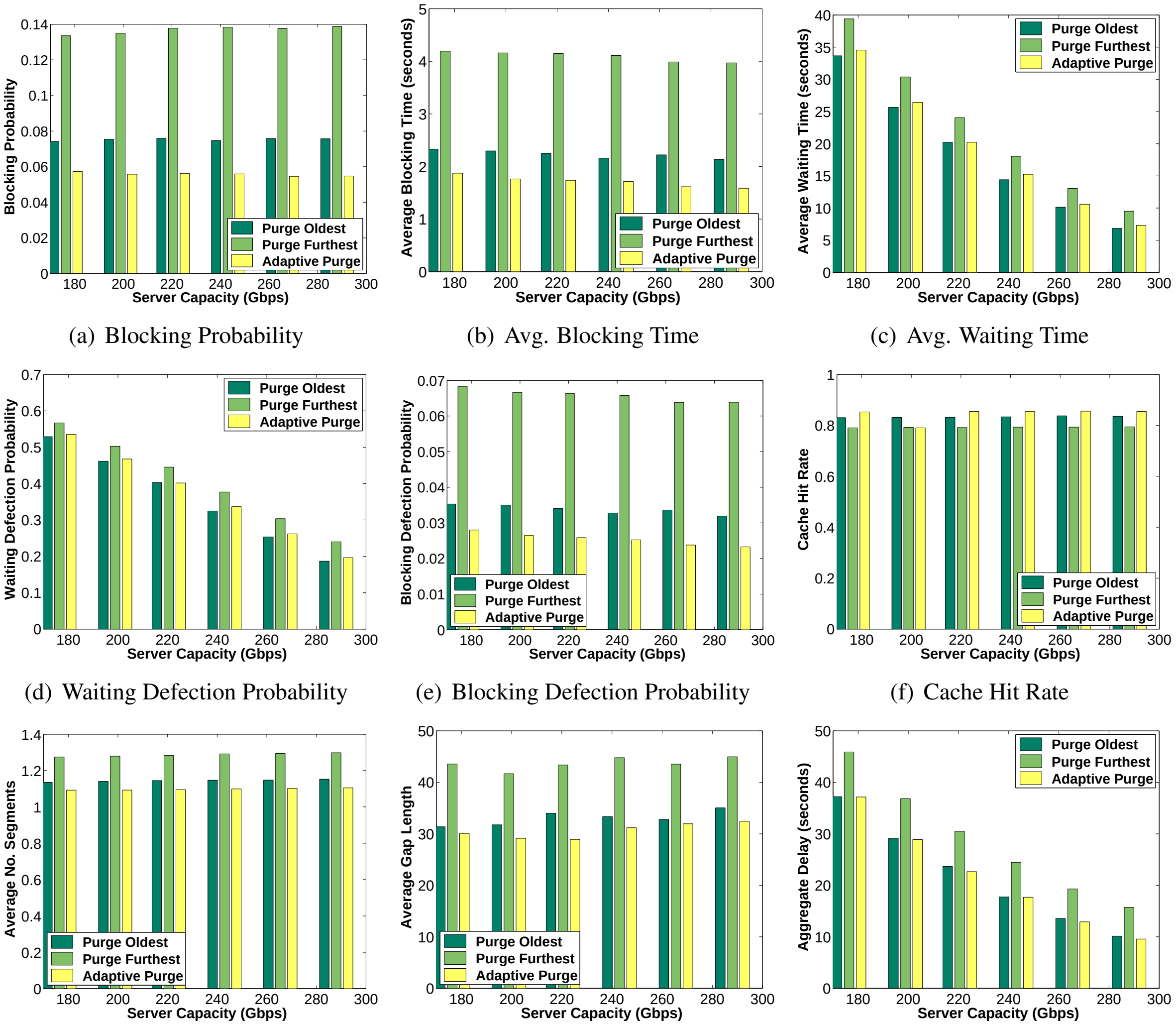
(f) Cache Hit Rate

(g) Avg. No. Segments

(h) Avg. Gap Length (seconds)

(i) Aggregate Delay (seconds)

Figure 4.6: Impact of the Purging Algorithm without C2L



(a) Blocking Probability

(b) Avg. Blocking Time

(c) Avg. Waiting Time

(d) Waiting Defection Probability

(e) Blocking Defection Probability

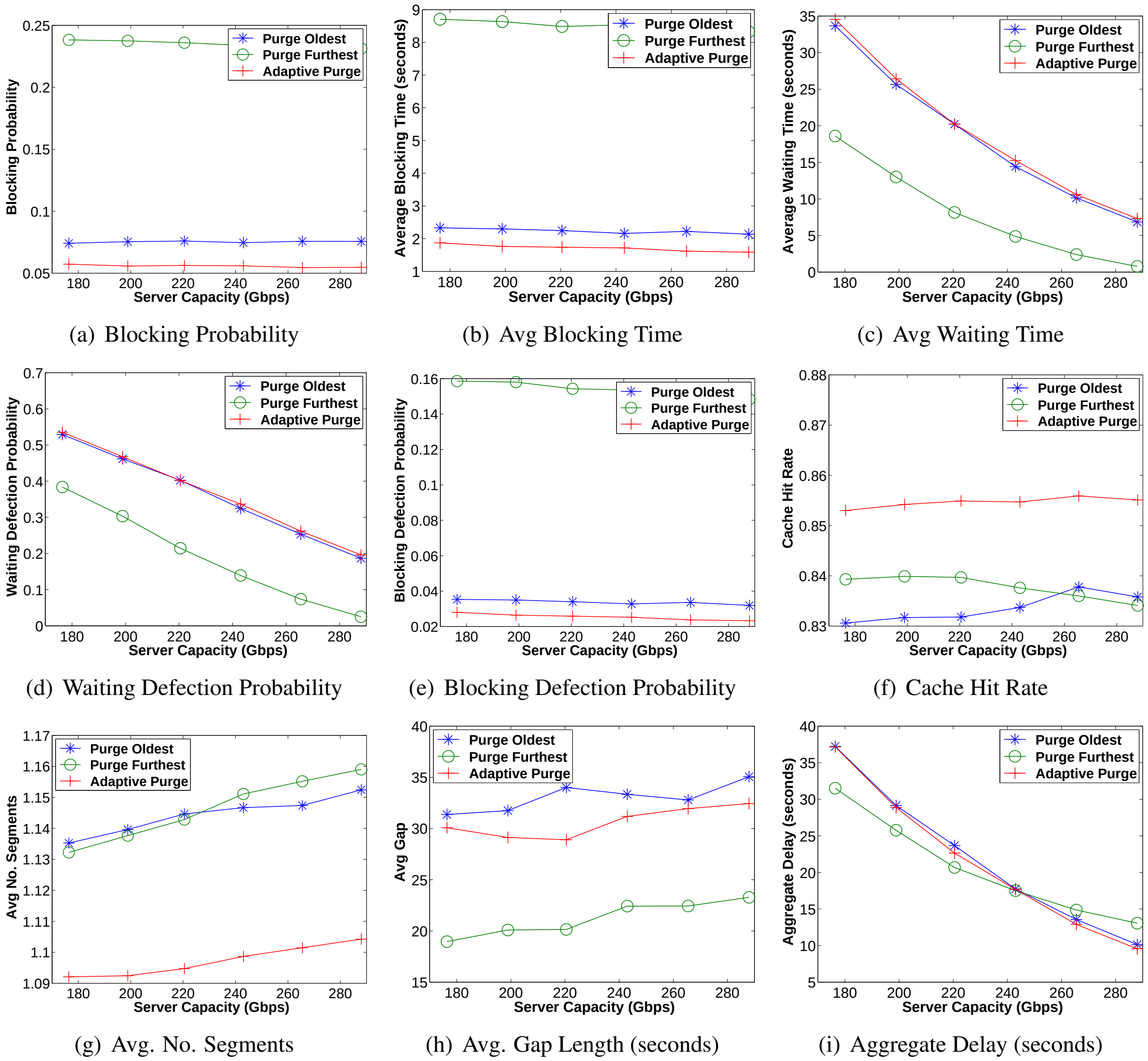
(f) Cache Hit Rate

(g) Avg. No. Segments

(h) Avg. Gap Length (seconds)

(i) Aggregate Delay (seconds)

Figure 4.7: Impact of the Purging Algorithm with C2L



(a) Blocking Probability

(b) Avg Blocking Time

(c) Avg Waiting Time

(d) Waiting Defection Probability

(e) Blocking Defection Probability

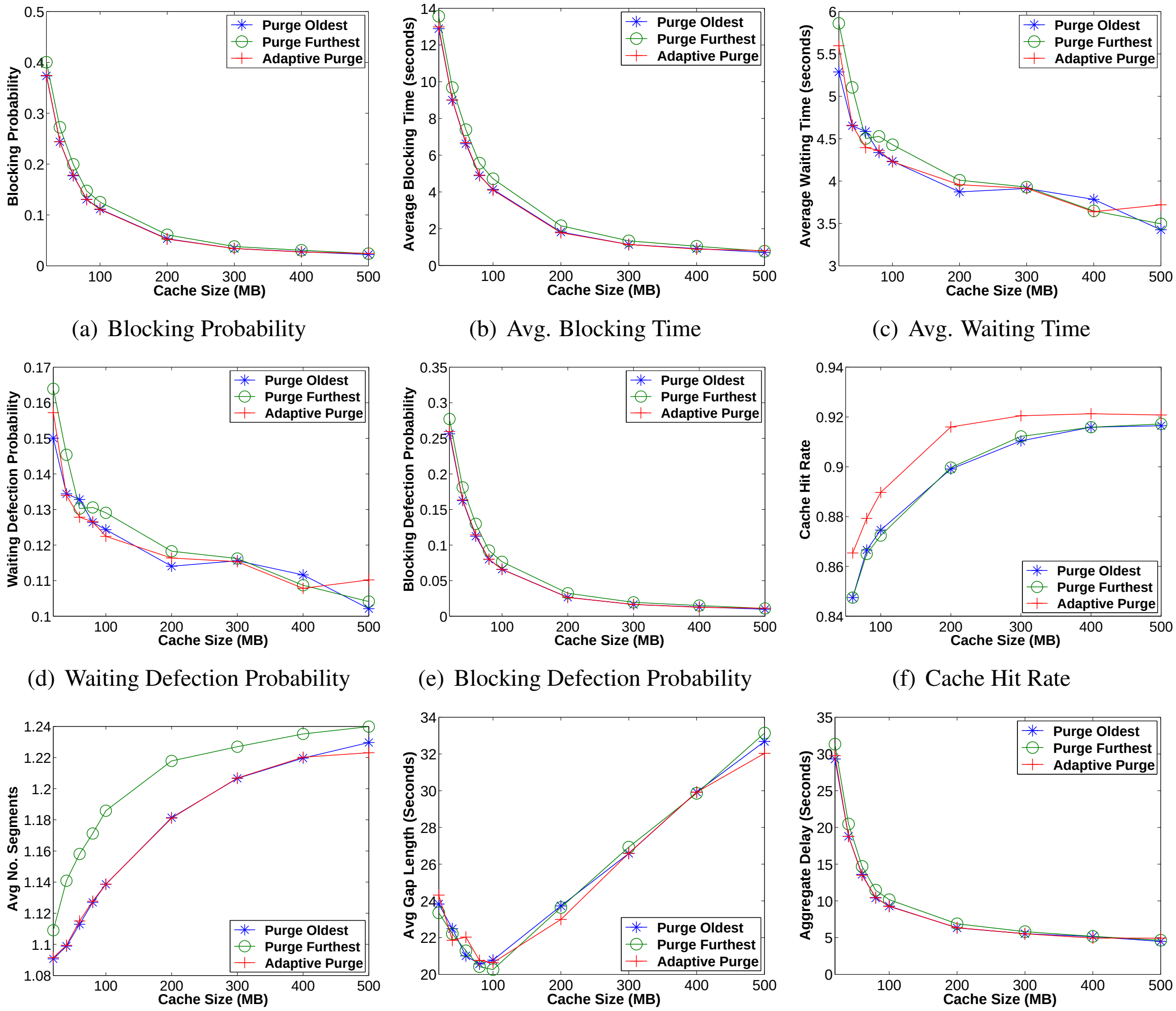
(f) Cache Hit Rate

(g) Avg. No. Segments

(h) Avg. Gap Length (seconds)

(i) Aggregate Delay (seconds)

Figure 4.8: Impact of the Purging Algorithm: Purge and C2L



(a) Blocking Probability

(b) Avg. Blocking Time

(c) Avg. Waiting Time

(d) Waiting Defection Probability

(e) Blocking Defection Probability

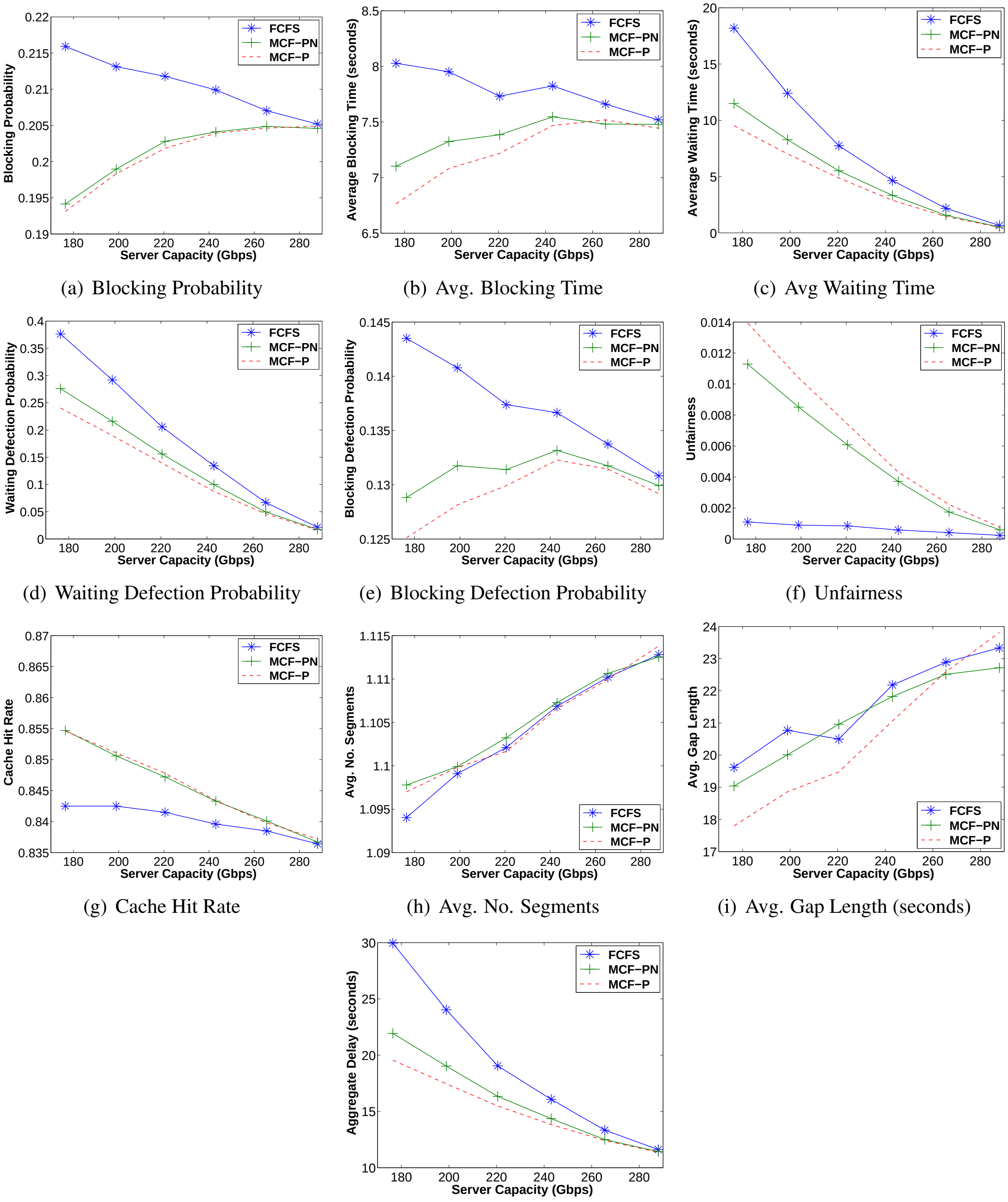
(f) Cache Hit Rate

(g) Avg. No. Segments

(h) Avg. Gap Length (seconds)

(i) Aggregate Delay (seconds)

Figure 4.9: Impact of Cache Size [Adaptive Purge with C2L]



(a) Blocking Probability

(b) Avg. Blocking Time

(c) Avg Waiting Time

(d) Waiting Defection Probability

(e) Blocking Defection Probability

(f) Unfairness

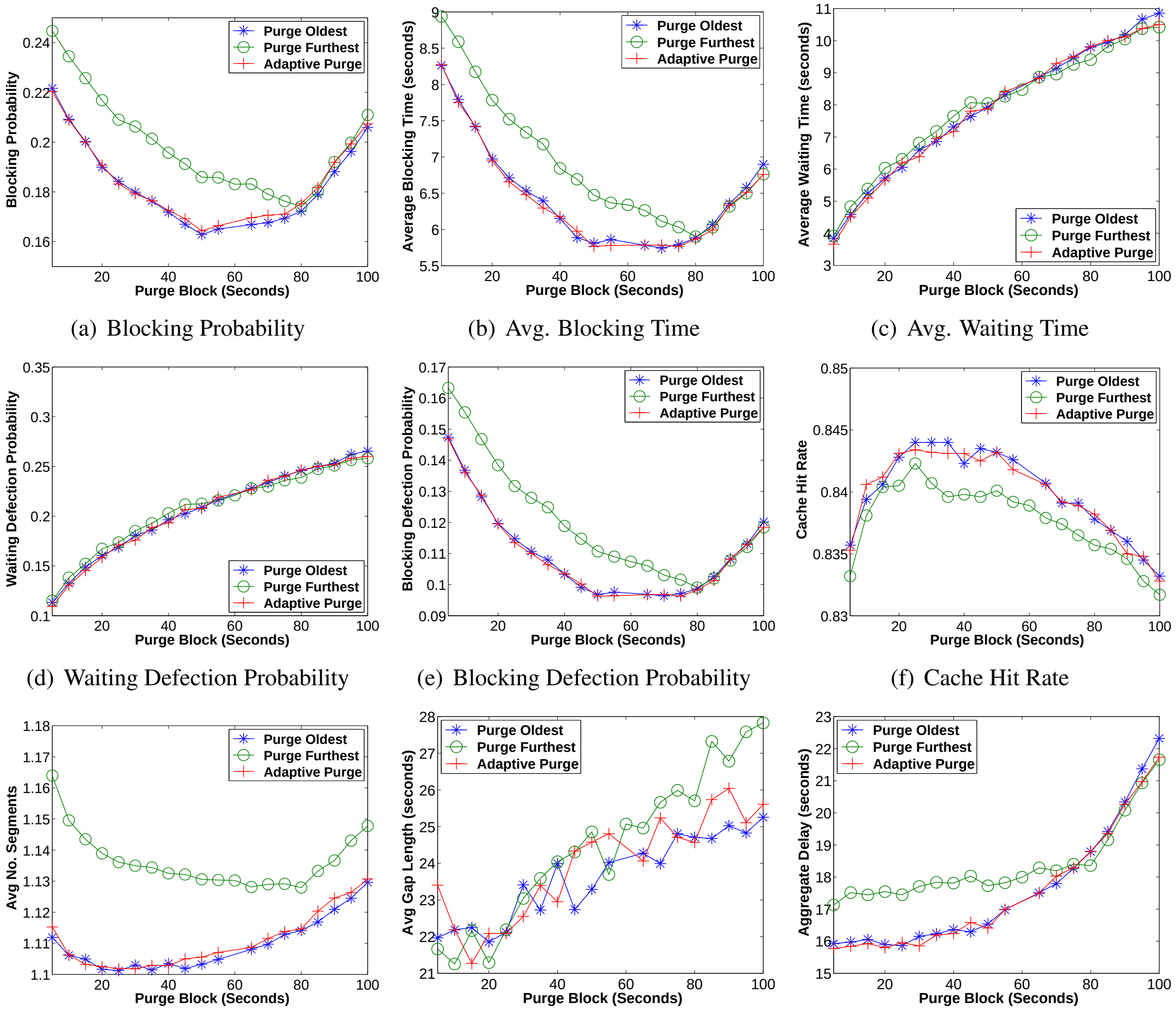
(g) Cache Hit Rate

(h) Avg. No. Segments

(i) Avg. Gap Length (seconds)

(j) Aggregate Delay (seconds)

Figure 4.10: Comparing Effectiveness of Scheduling Policies



(a) Blocking Probability

(b) Avg. Blocking Time

(c) Avg. Waiting Time

(d) Waiting Defection Probability

(e) Blocking Defection Probability

(f) Cache Hit Rate

(g) Avg. No. Segments

(h) Avg. Gap Length (seconds)

(i) Aggregate Delay (seconds)

Figure 4.11: Impact of Purge Block Size

CHAPTER 5 ANALYZING BOOKMARKING IN SCALABLE VIDEO STREAMING

5.1 Introduction

As streaming online videos becomes more and more popular, the need for more sophisticated interactive requests grows. One type of interactive request that is gaining a lot of momentum recently is the ability to jump to interesting points (bookmarks) within the video and start playing from there. Previous studies [38, 39] analyzed a number of video streaming experiments and characterized the user interactive behavior. They have shown that video segments are not watched at the same probability. While some segments are skipped, some skimmed, others are repeated and watched again and again. A video segment that is searched and watched repeatedly is called a *hotspot* and is pointed to by a bookmark. Some videos' categories may contain bookmarks more than others such as sporting games and educational lectures. Studies [38, 39] show that many users, who missed a game, are more likely to search for and watch the major events during the game rather than watching the whole game video. Study [38] proposed a dynamic bookmark placement to move misplaced bookmarks to the appropriate positions and to pre-fetch content based on predictable user interactive behavior.

We evaluate the system performance under a highly interactive realistic workload supporting bookmarking through extensive simulations. We analyze many metrics, including waiting and blocking metrics, aggregate delay and cache hit rate. To quantify the customers perceived quality of experience, we introduce a new metric Quality of Experience (QoE), which we will explain in detail in Section 5.3.

The main contributions of this part of the dissertation can be summarized as follows.

- We evaluate the system performance when a highly interactive workload with bookmarks is considered.
- We introduce three enhancements to improve the QoE perceived by the customer.
- We extensively evaluate the effectiveness of the enhancements in terms of various metrics under

realistic workload, considering several important parameters.

The rest of this chapter is organized as follows. Section 5.2 presents the. Section 5.3 discusses the performance evaluation methodology. Section 5.4 presents and analyses the main results. Finally, Section 5.5 concludes with a summary of the results.

5.2 Proposed Solution

5.2.1 Considered System

The system considered is the same as in Chapter 3. It consists of a streaming server, clients, and a network connecting them. The major components of the NVOD server are waiting queues (with one queue for each video), a blocking queue, a queuing manager, a dynamic I-Stream allocation module, a SAM technique, a stream merging protocol, a waiting scheduling policy, a blocking scheduling policy, and streams. The streams are divided into B-Streams, P-Streams, and I-Streams. B-Streams and P-Streams are used to service waiting customers and can be shared with other requests for the same video.

The customer starts a streaming session by issuing a request to playback a certain video. The server decides whether to service the request based on the availability of server channels. A channel is the set of required server resources (network bandwidth, disk I/O bandwidth, etc.) for delivering a video stream. The number of such channels is called *server capacity*. If adequate channels are available, the request is delivered using a stream merging technique; otherwise, the request is placed in the waiting queue for that video. The server applies a waiting scheduling policy to determine which waiting queue to service when streams become available. The most popular policies are *First Come First Serve* (FCFS) [46], *Maximum Queue Length* (MQL) [46], and *Maximum Cost First* (MCF) [49] and references within. FCFS selects the queue with oldest request and is thus the fairest, whereas MQL tries to maximize the number of requests that can be serviced with one channel by selecting the queue with the largest number of requests. In contrast, MCF selects the queue with the minimum cost in terms of stream length.

During a streaming session, the customer issues interactive requests to pause the video, jump forward, or backward by a certain amount relative to the current playback position. The server applies a blocking scheduling policy to service blocked customers when server channels become available. The waiting and blocking scheduling policies are not necessarily the same. Waiting and blocked customers defect when the waiting/blocking time exceeds their tolerance.

We developed three enhancements to the system to exploit user behavior in order to improve the customer perceived QoE. The following sections explain each one of them in detail.

5.2.2 No Bookmark Purging Algorithm

Figure 5.1 illustrates the proposed purging algorithm (NoBookmarkPurging). The algorithm avoids purging any hotspot data that is already cached. When the cache gets full, the purging algorithm clears the segment tail or the segment head. The algorithm avoids purging the data in the middle of a segment, such that it does not increase the cache fragmentation. The purging algorithm tries to purge the oldest data first. However, if it cannot, it purges the segment head. A new condition is added to check whether any hotspot data is included in the to-be purged data. If it is not, then the purging algorithm clears the data. Otherwise, it moves on to the next segment. The process loops through all segments until enough data is purged. If the system cannot find any segment without any hotspot data to purge, then it purges the least amount of hotspot data of the oldest segment.

Figure 5.3 shows an example of a customer's cache. All three scenarios show the customer cache consists of three segments. The segments are sorted chronologically from the segment holding the oldest data to the segment holding the latest data. Hence, segments A and C hold the oldest and the most recent streamed data respectively. The segment's tail holds the oldest in the segment. Hotspots are contained in each segment. Scenario 1 shows that segment A tail does not include any hotspot data. Hence, the proposed algorithm can purge the tail. Scenario 2 shows that segment A tail includes a hotspot data.

Hence, the proposed algorithm purges segment's A head, which does not include any hotspot data. Scenario 3 shows that both the head and tail of segment A contain hotspot data. Hence, the proposed algorithm avoids purging both and moves to segment B. Since segment's B tail does not contain hotspot data, the proposed algorithm purges its tail.

```
NoBookmarkPurge(Customer)
{
//Loop through all segments starting from oldest
  for ($segment = 0; segment < NumSegments; segment++$)
  {
    //Check if any hotspot is within segment tail
    if ($IsHotspot(Customer, segment, TAIL) == 0$)
      PurgeData(Customer, segment, TAIL);
    else if ($IsHotspot(Customer, segment, HEAD) == 0$)
        PurgeData(Customer, segment, HEAD);
     //Is cache less than max allowed size?
     if ($CacheSize \leq MAX\_CACHE\_SIZE$)
        return;
     else
        continue; //Try next segment
    } //for
  if ($CacheSize \geq MAX\_CACHE\_SIZE$)
     PurgeOldest(Customer, segment, TAIL);
} //NoBookmarkPurge
IsHotspot(Customer, Segment, TAIL_HEAD)
{
  for ($Bookmark = 0; Bookmark < NumBookmarks; Bookmark++$)
     if (TAIL_HEAD == TAIL)
     {
        if ($(Bookmark > Segment.Start) and (Bookmark < Segment.Start + PurgeData)$)
           return(1);
        else
           return(0);
     }
     if (TAIL_HEAD == HEAD)
     {
        if ($(Bookmark < Segment.End) and (Bookmark > Segment.End - PurgeData)$)
           return(1);
        else
           return(0);
     } //if
  } //for
} //IsHotspot
```

Figure 5.1: Simplified Algorithm for NoBookmark Purging

5.2.3 Channel Reservation

We reserve server channels to cache hotspot data to increase the probability that future interactive requests are serviced from the customer's cache. Since the overall server capacity is fixed, any increase in the reserved channels must be accompanied by an equal decrease in the server capacity available for

```
//Reduce Server Capacity by Reserved Channels
$BW = BW - ReservedChannels$
**ReserveChannels()**
{
  **if** $(ReservedChannels \geq 1)$
  {
    //Loop through all customers that are will be served this time
    **for** $(Customer = 0; Customer < NumCustomers; Customer++)$
      FetchHotspots(Customer);
    $ReservedChannels = ReservedChannels - 1$
  } //if
} //ReserveChannels
**FetchHotspots(Customer)**
{
  //Loop through all bookmarks in the customer's video
  **for** $(Bookarmk = 0; Bookmark < NumBookmarks; Bookmark++)$
    CacheHotspot(Customer, Bookmark);
} //FetchHotspots
**CacheHotspot(Customer, Bookmark)**
{
  //Loop through all customer's segments
  **for** $(segment = 0; segment < NumSegments; Segment++)$
  {
  //Find the first unused segment in the customer's cache
    **if** $(Segment.Start \geq 0)$
      continue;
    **else**
      break;
  }
  $Segment.Start = Bookmark$
  $Segment.End = Bookmark + HOTSPOT\_LENGTH$
  //Merge cache segments if they overlap
  MergeSegments(Customer);
} //CacheHotspot
```

Figure 5.2: Simplified Algorithm for Reserving Channels

waiting customers. Since the hotpot data could overlap with already cached data in the customer's cache, the system checks for overlapping segments and merges them.

5.2.4 Fetching Hotspot Data

We use multicast channels when they become available to fetch hotspot data for customers not listening to any stream. When channels become available, the system starts streaming all hotspot data for the video to fetch hotspot data. Since multicast channels are used, all customers listening to the same video cache the hotspot data from the same channel at the same time.

5.3 Performance Evaluation Methodology

We developed a simulator for an interactive NVOD system, including both the clients with caches and the server. The system supports various stream merging and scheduling techniques. The simulator

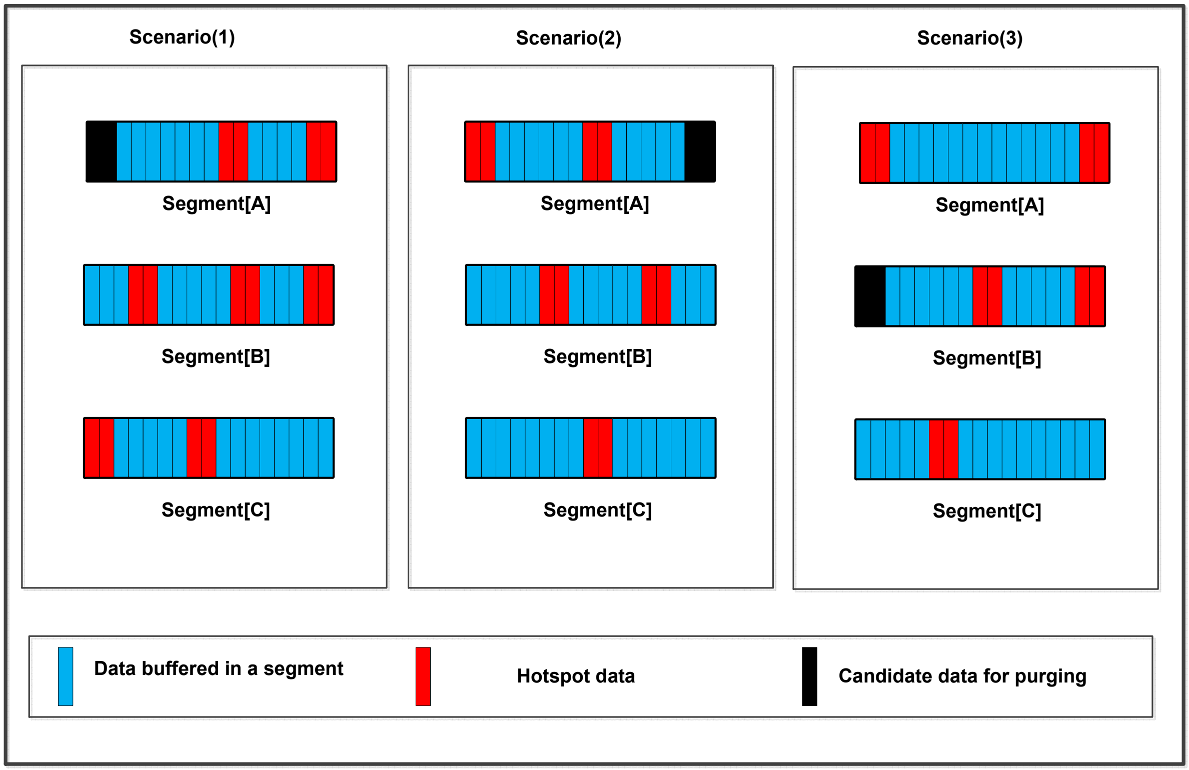


Figure 5.3: Clarification of the No Bookmark Purging Algorithm

stops after a steady state analysis with 95% confidence interval is reached. We experimented with a wide range of system parameters, but only the main results are shown.

5.3.1 Client and Server Characteristics

Table 5.1 summarizes the default parameters used. We characterize the waiting and blocking tolerance of customers as Poisson with a mean of 30 seconds. We examine the server at different loads by varying server capacities from 1000 Gbps to 1250 Gbps. Only 2% of the server capacity is reserved for I-Streams. We use the MCF-PN scheduling policy for waiting customers except when evaluating scheduling policies and FCFS for blocking customers to avoid any issues of unfairness. The playback deviation point tolerance is kept at 10 seconds. The default client side cache size dedicated to the video streaming application is kept at 1 GB except when evaluating the cache size. Table 5.2 shows the distribution of interactive requests.

Table 5.1: Default Parameters Values

Parameter	Default Value(s)
Arrival Rate	30 Requests / hour
Number of Videos	100
Waiting Tolerance Model	Poisson with mean = 30 sec.
Blocking Tolerance Model	Poisson with mean = 30 sec.
Server Capacity	1000 - 1250 Gbps
Video Bit-rate	1 Mbps
I-Streams	2% of server capacity
Cache Size	1 GByte
Playback Point Deviation Tolerance	10 seconds

Table 5.2: Distribution of Interactive Requests

Action	Percentage	Mean	Standard Deviation Per Session
Back 10 s	4.5	0.59	0.34
Back 30 s	0.95	0.12	0.83
Back 60 s	2.22	0.29	1.90
Forward 10 s	10.79	1.41	8.61
Forward 30 s	2.62	0.34	2.93
Forward 60 s	11.51	1.50	7.38
Seek-bar	14.37	1.88	7.39
Bookmarks	21.90	2.62	2.63
Pause	17.18	2.24	7.65
Resume	13.69	1.82	6.80

5.3.2 Workload Characteristics

We utilize the results of [38], which characterizes a highly interactive workload with bookmarking. The average arrival rate (λ) is 30 requests per hour. We study 100 videos of varying lengths. The video length varies from 45 minutes to 4 hours with an average of 2.5 hours. The workload indicates that requested files have an average bit-rates of 1 Mbps.

5.3.3 Performance Metrics

We evaluated the system performance using the following metrics: Average waiting time, average blocking time, waiting defection probability, blocking defection probability, blocking probability, unfairness, cache hit rate, and aggregate delay. Furthermore, we propose a new metric to measure the customer perceived QoE. Since aggregate delay encompasses both waiting and blocking times, we mapped it to the QoE metric. If a customer defects due to waiting or blocking, then his QoE is considered Bad. We mapped the aggregate delay to zones according to Table 5.3

Table 5.3: QoE Mapping

Aggregate Delay	QoE
0 - 10 s	Excellent
10 - 20 s	Good
20 - 30 s	Fair
30 - 40 s	Poor
40 - 50 s	Bad

5.4 Result Presentation and Analysis

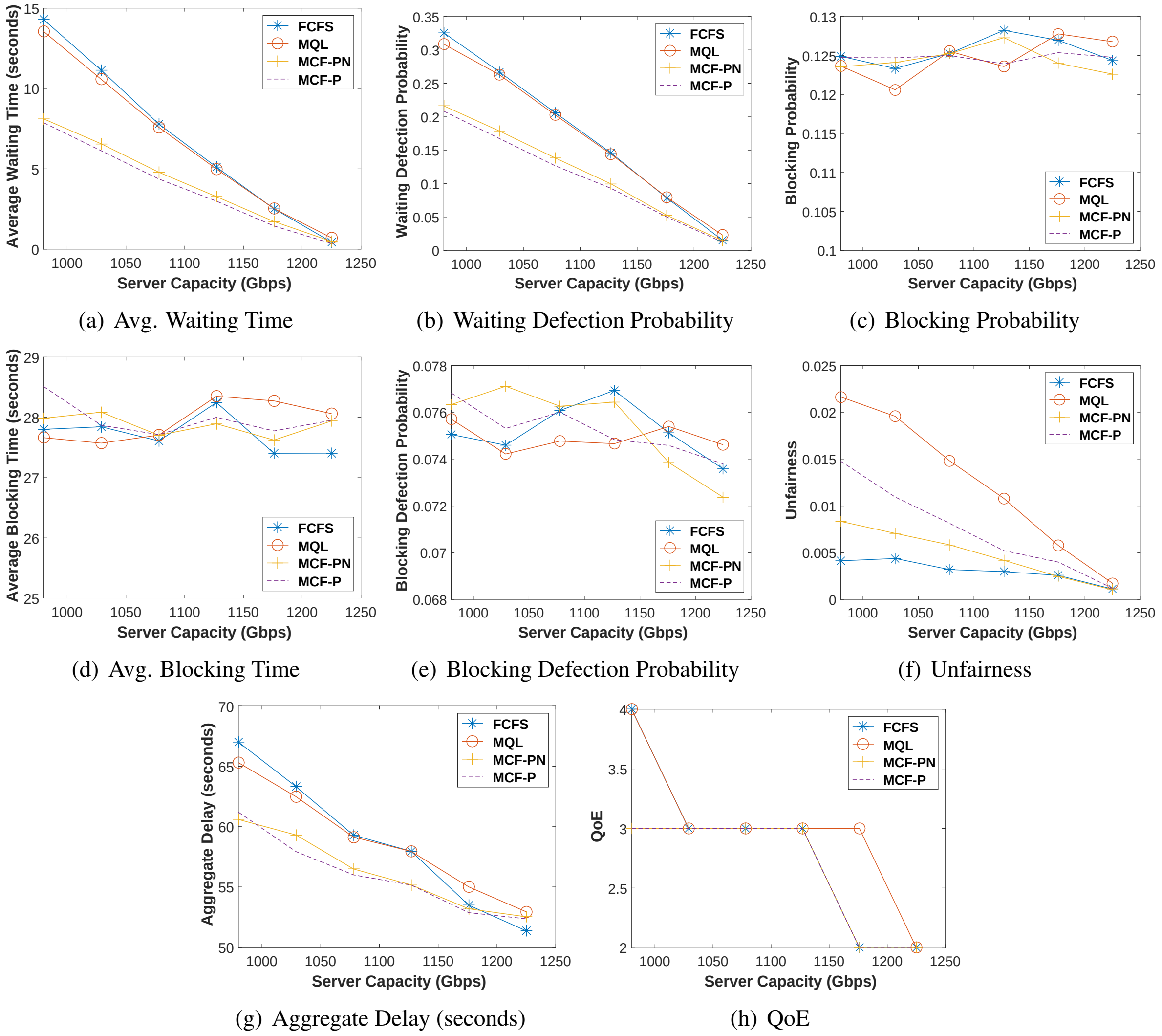
We studied the impact of bookmarking on NVOD systems through extensive simulations when stream merging techniques such as Patching and various scheduling policies are employed.

5.4.1 Impact of Scheduling Policy

Figure 5.4 compares the effectiveness of different scheduling policies. MCF-P outperforms the other scheduling policies in terms of waiting metrics. Since the scheduling policy affects already admitted customers into the system, the blocking metrics are not impacted by the scheduling policy. MCF-PN, which is the normalized version of MCF-P, is fairer than MCF-P and very close in performance to MCF-P. Hence we use it for all subsequent sections.

5.4.2 Impact of Request Rate

Figure 5.5 illustrates the impact of the request rate on the system performance. As the request rate increases, the average waiting time and waiting defection probability increase because there are more customers waiting to be serviced while the server capacity is kept the same. The server can't keep up with the incoming requests. The blocking metrics are not affected by the increase of request rate. As we increase the server capacity, the waiting metrics improve significantly because the server can admit more customers. However the blocking metrics improve slightly. Since the Aggregate delay encompasses both waiting and blocking metrics and both metrics improve with server capacity, we notice that the



(a) Avg. Waiting Time

(b) Waiting Defection Probability

(c) Blocking Probability

(d) Avg. Blocking Time

(e) Blocking Defection Probability

(f) Unfairness

(g) Aggregate Delay (seconds)

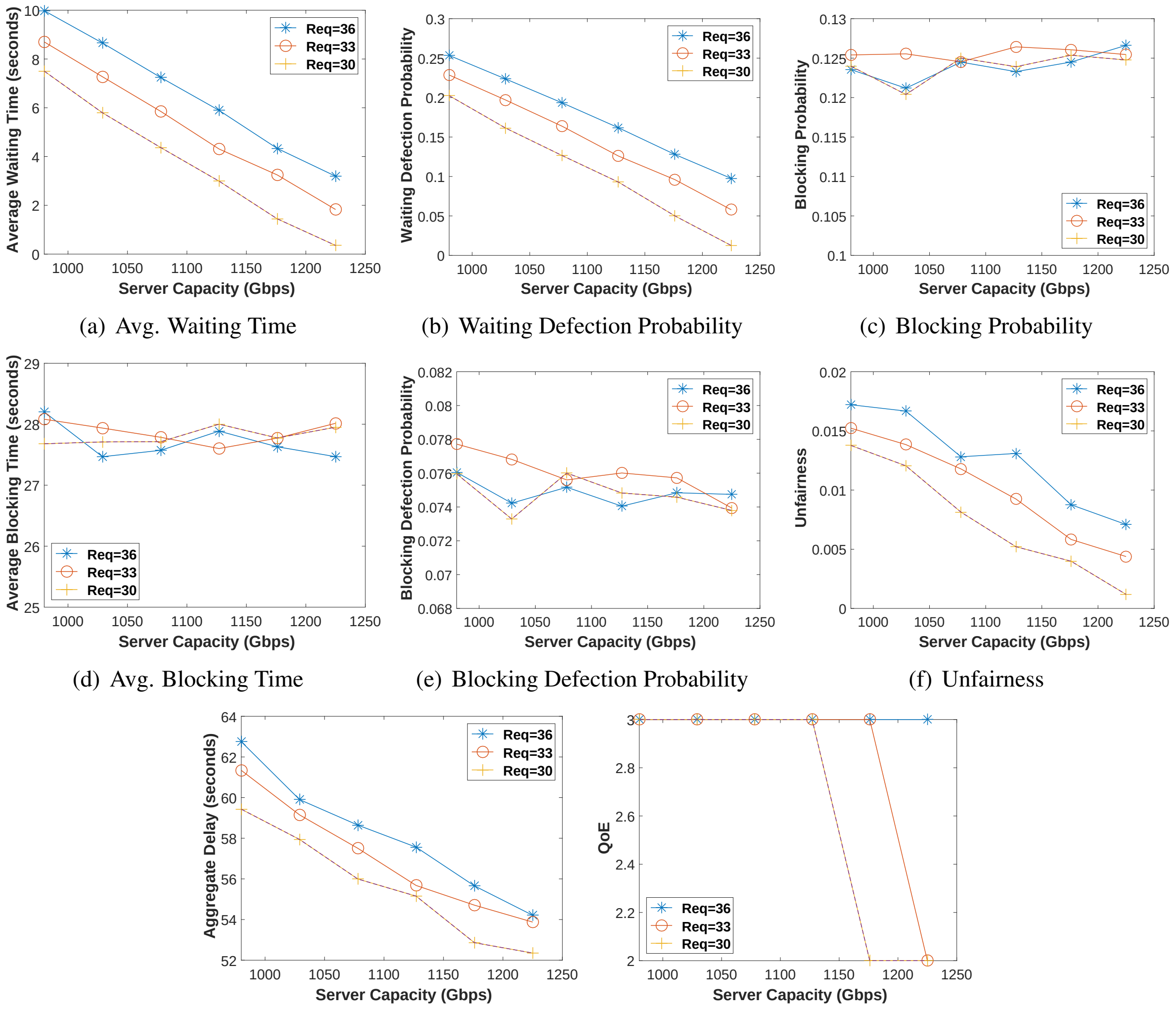
(h) QoE

Figure 5.4: Comparing Effectiveness of Scheduling Policies

Aggregate Delay improves significantly with server capacity. The Aggregate Delay degrades with the increase of the request rate.

5.4.3 *Impact of Cache Size*

Figure 5.6 illustrates the impact of the cache size on the system. The waiting metrics improve with the increase of cache size because the system makes sure there is enough cache to hold a patch before it starts a new one. Otherwise, it starts a full length stream. Hence, as the cache size increases, a larger percentage of the streams is patch streams. The improvement continues till the cache size reaches about



(a) Avg. Waiting Time

(b) Waiting Defection Probability

(c) Blocking Probability

(d) Avg. Blocking Time

(e) Blocking Defection Probability

(f) Unfairness

(g) Aggregate Delay (seconds)

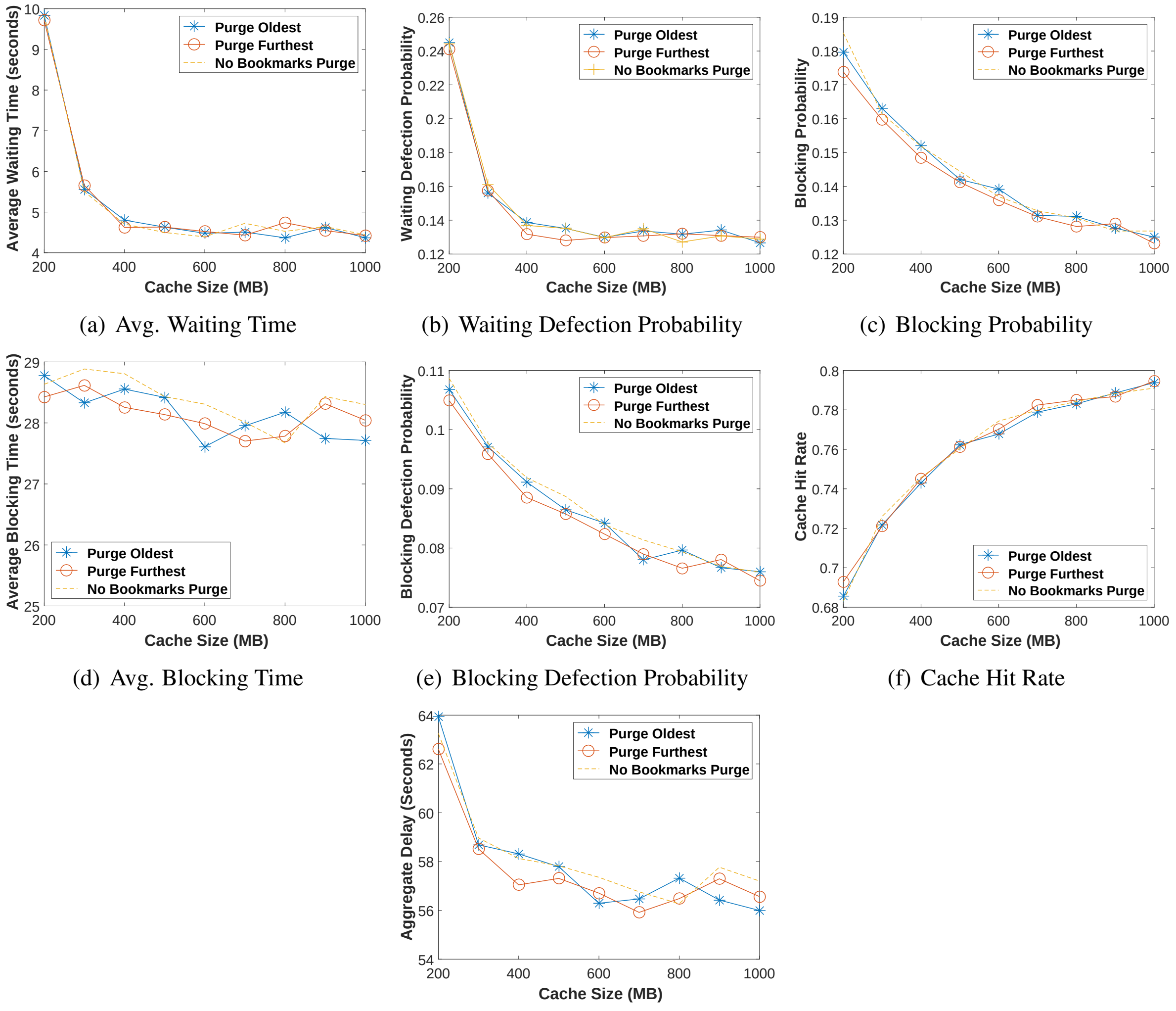
(h) QoE

Figure 5.5: Impact of Request Rate

400 MByte. Blocking metrics also improve with cache size because a larger percentage of interactive requests could be serviced from the cache as shown in the 5.6(f). Since both waiting and blocking metrics improve with cache size, the Aggregate Delay improves.

5.4.4 Impact of Purging Algorithm

Figures 5.6 and 5.7 illustrate the impact of the purging algorithm on the system performance. We introduced a new purging algorithm (NoBookmarks purging), which avoids purging hotspot data if they already exist in the cache. The purging algorithm does not make a significant impact on waiting cus-



(a) Avg. Waiting Time

(b) Waiting Defection Probability

(c) Blocking Probability

(d) Avg. Blocking Time

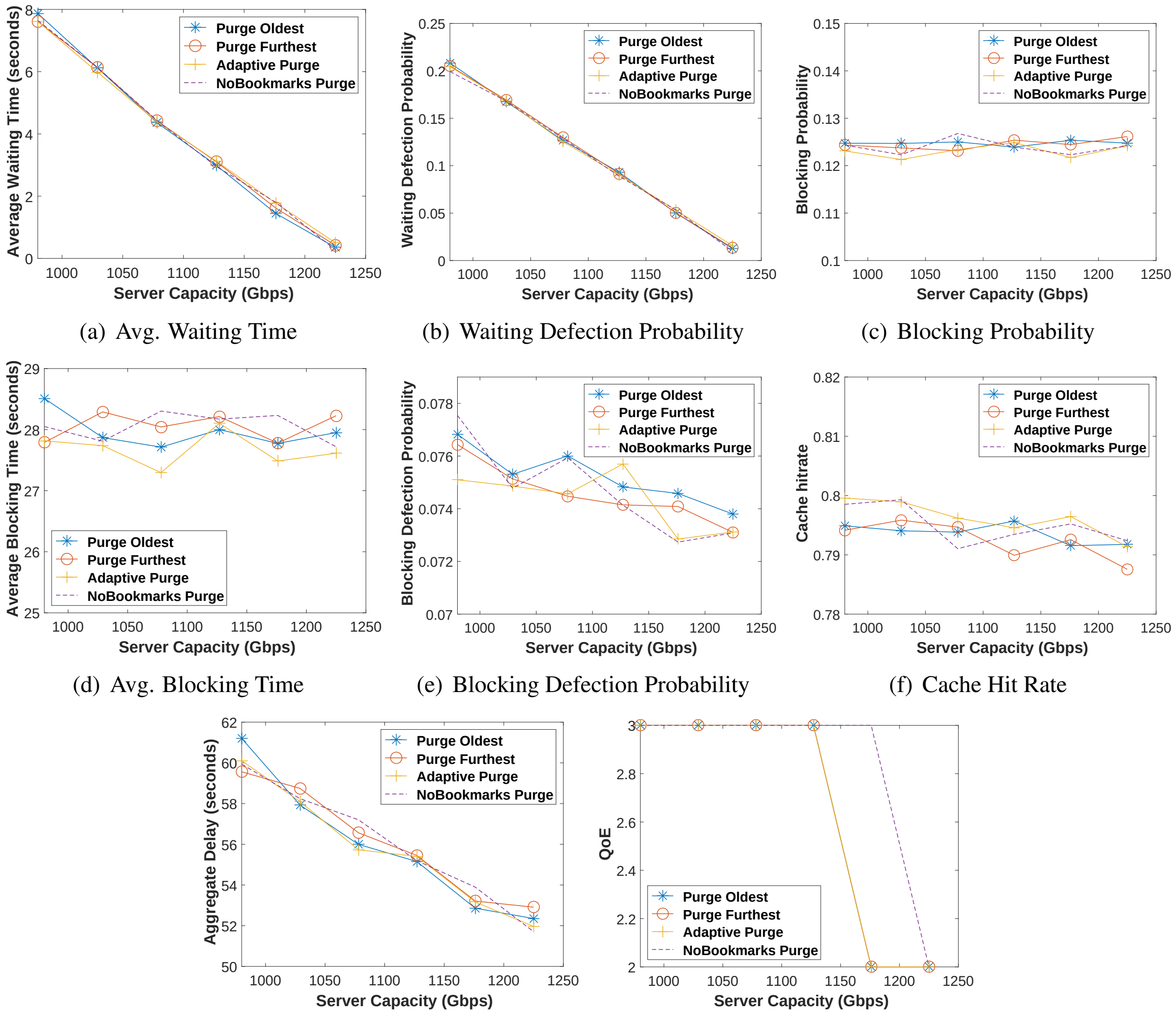
(e) Blocking Defection Probability

(f) Cache Hit Rate

(g) Aggregate Delay (seconds)

Figure 5.6: Impact of Cache Size

tomers because it purges the cache of already admitted customers. We notice that the new purging algorithm does not make a significant difference on blocking metrics also. Since most of the jump to bookmarks are long jump forwards, the likelihood of a hotspot to be already cached is very small, which minimizes the impact of the newly suggested algorithm. We conclude that the proposed algorithm would have a bigger impact in workloads where a high percentage of the jump to bookmarks are to points closer to the playback point, which tend to be inside the cache. Furthermore, since the system caches past data already watched, the proposed algorithm would be more effective if a significant percentage of the jump



(a) Avg. Waiting Time (b) Waiting Defection Probability (c) Blocking Probability

(d) Avg. Blocking Time (e) Blocking Defection Probability (f) Cache Hit Rate

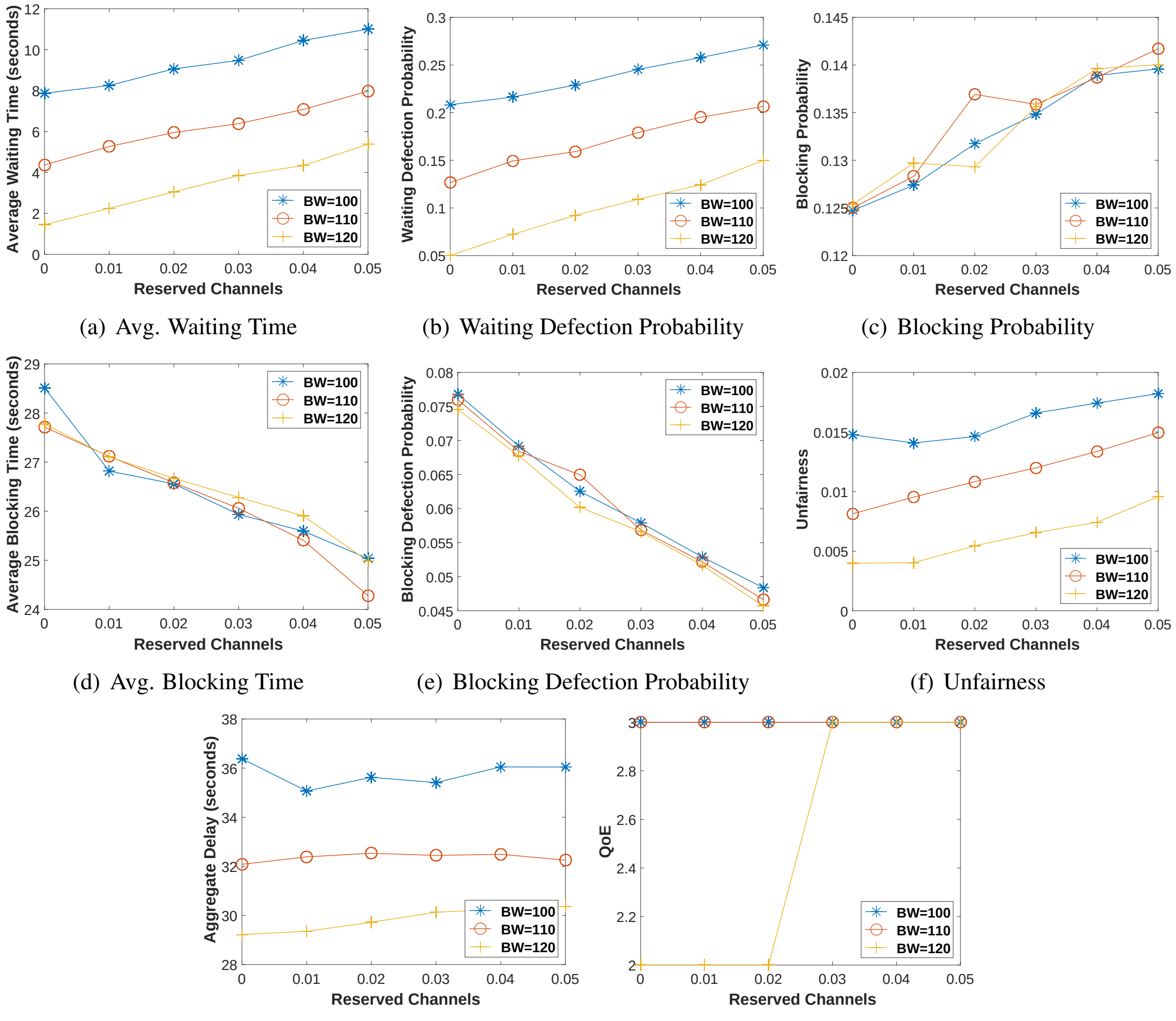
(g) Aggregate Delay (seconds) (h) QoE

Figure 5.7: Impact of the Purging Algorithm

to bookmarks are jump backwards.

5.4.5 Impact of Reserved Channels

Figure 5.8 illustrates the impact of reserving channels to cache hotspot data on the system performance. The average blocking time and blocking defection decrease as the percentage of the server capacity dedicated for streaming hotspot data is increased. Since the overall server capacity is fixed, as we increase the percentage of reserved channels, the available server capacity for waiting customers decreases. Hence, the waiting metrics worsen with the increase of reserved channels. Subsequently, the



(a) Avg. Waiting Time (b) Waiting Defection Probability (c) Blocking Probability

(d) Avg. Blocking Time (e) Blocking Defection Probability (f) Unfairness

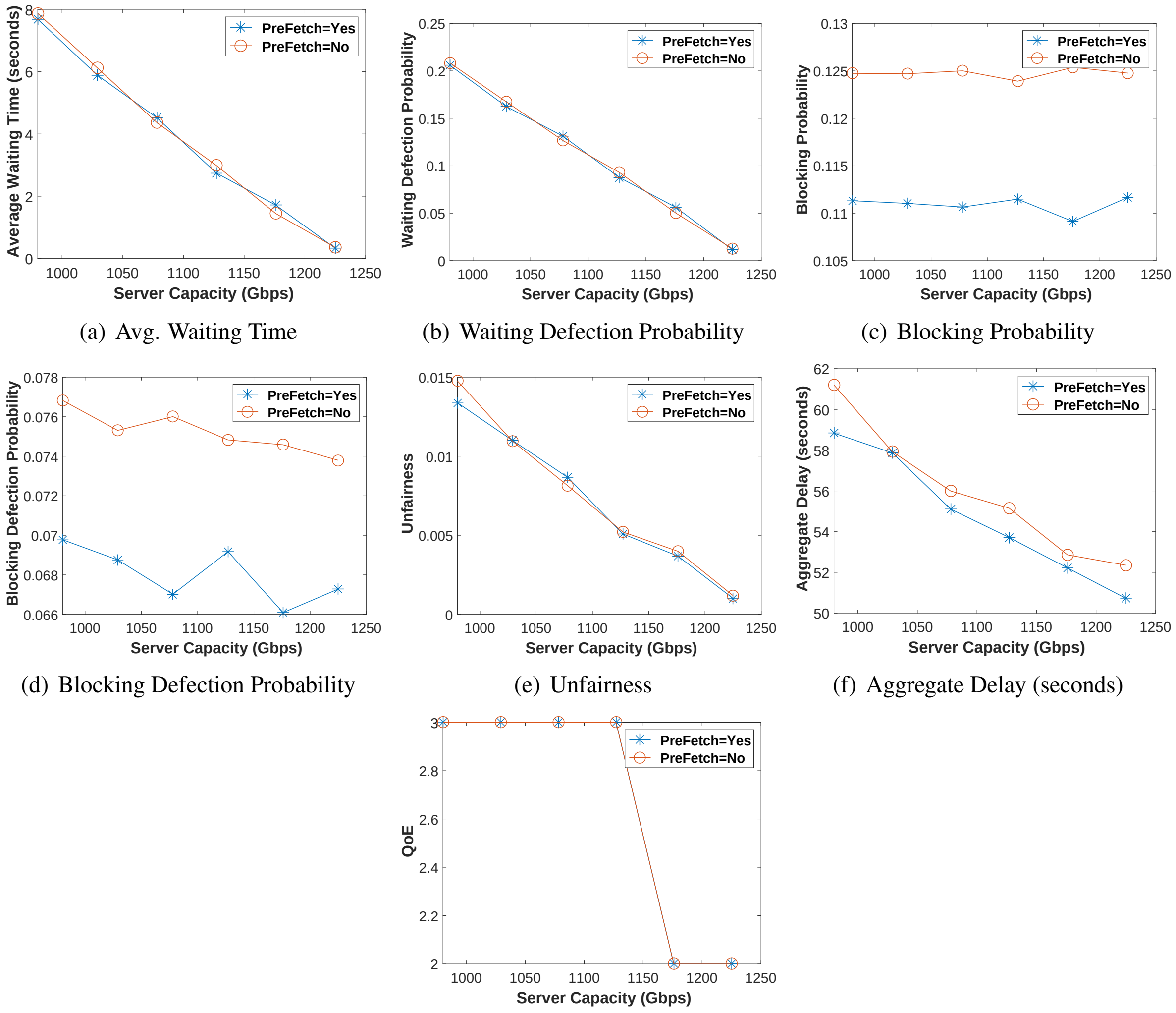
(g) Aggregate Delay (seconds) (h) QoE

Figure 5.8: Impact of Reserved Channels

Aggregate Delay does not show significant improvement.

5.4.6 Impact of Fetching Hotspot Data

Figures 5.9, 5.10, and 5.11 illustrate the impact of fetching hotspot data on the system performance when PurgeOldest, PurgeFurthest, and NoBookmarks Purge algorithms are considered respectively. All three figures show improvement in blocking metrics and Aggregate Delay.



(a) Avg. Waiting Time

(b) Waiting Defection Probability

(c) Blocking Probability

(d) Blocking Defection Probability

(e) Unfairness

(f) Aggregate Delay (seconds)

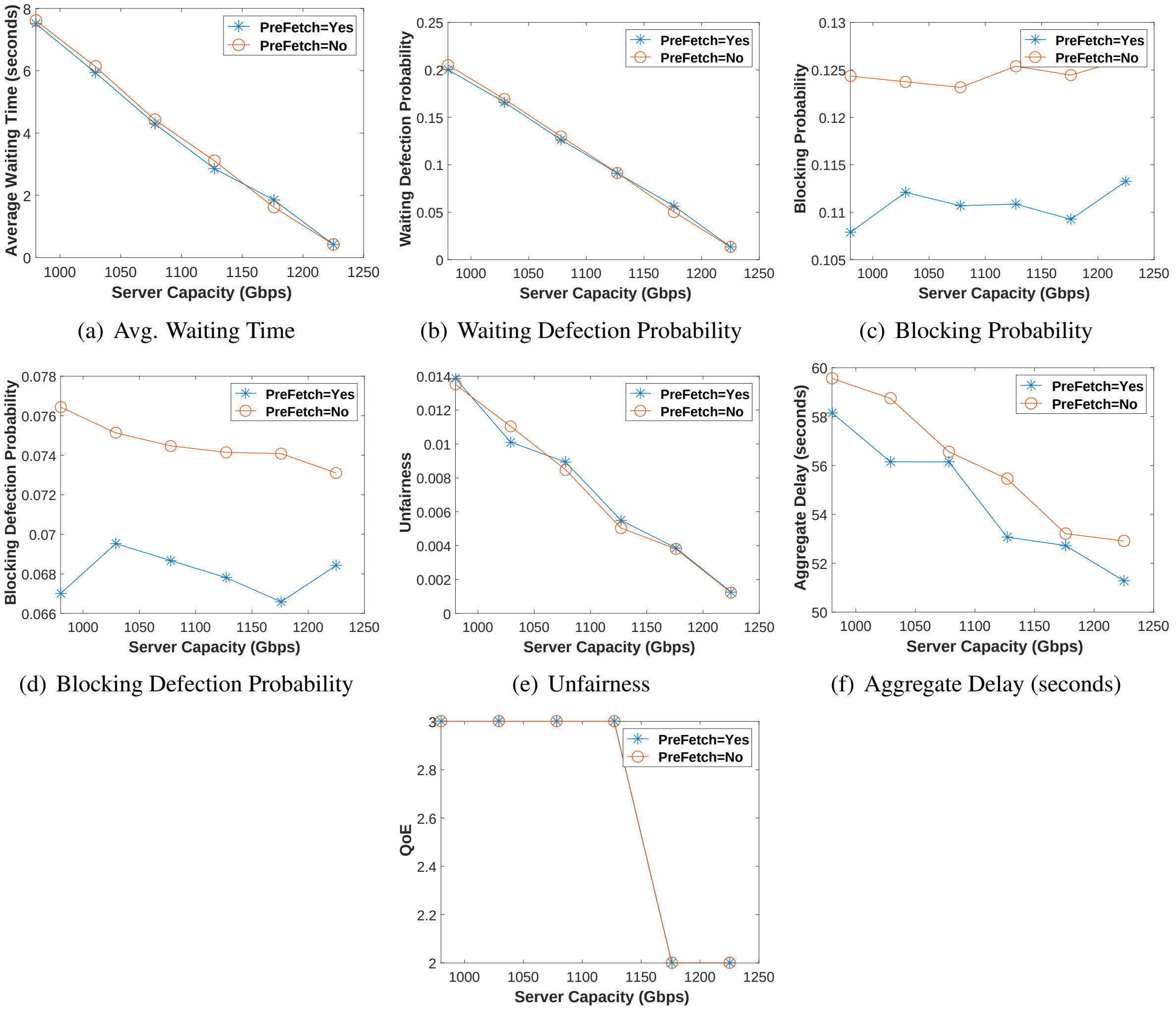
(g) QoE

Figure 5.9: Impact of Pre-fetching [Purge Oldest]

5.5 Conclusions

We have evaluated a highly interactive NVOD systems and have analyzed its performance under realistic workload including bookmarking and using various resource sharing techniques and scheduling policies. The main results can be summarized as follows.

- The used scheduling policy impacts the system's performance significantly. While MCF-P outperforms the other scheduling policies, it is an unfair policy. Using MCF-PN (proposed in Chapter 3) proves to be fairer and provides very close performance to MCF-P. Hence, we recommend using it



(a) Avg. Waiting Time

(b) Waiting Defection Probability

(c) Blocking Probability

(d) Blocking Defection Probability

(e) Unfairness

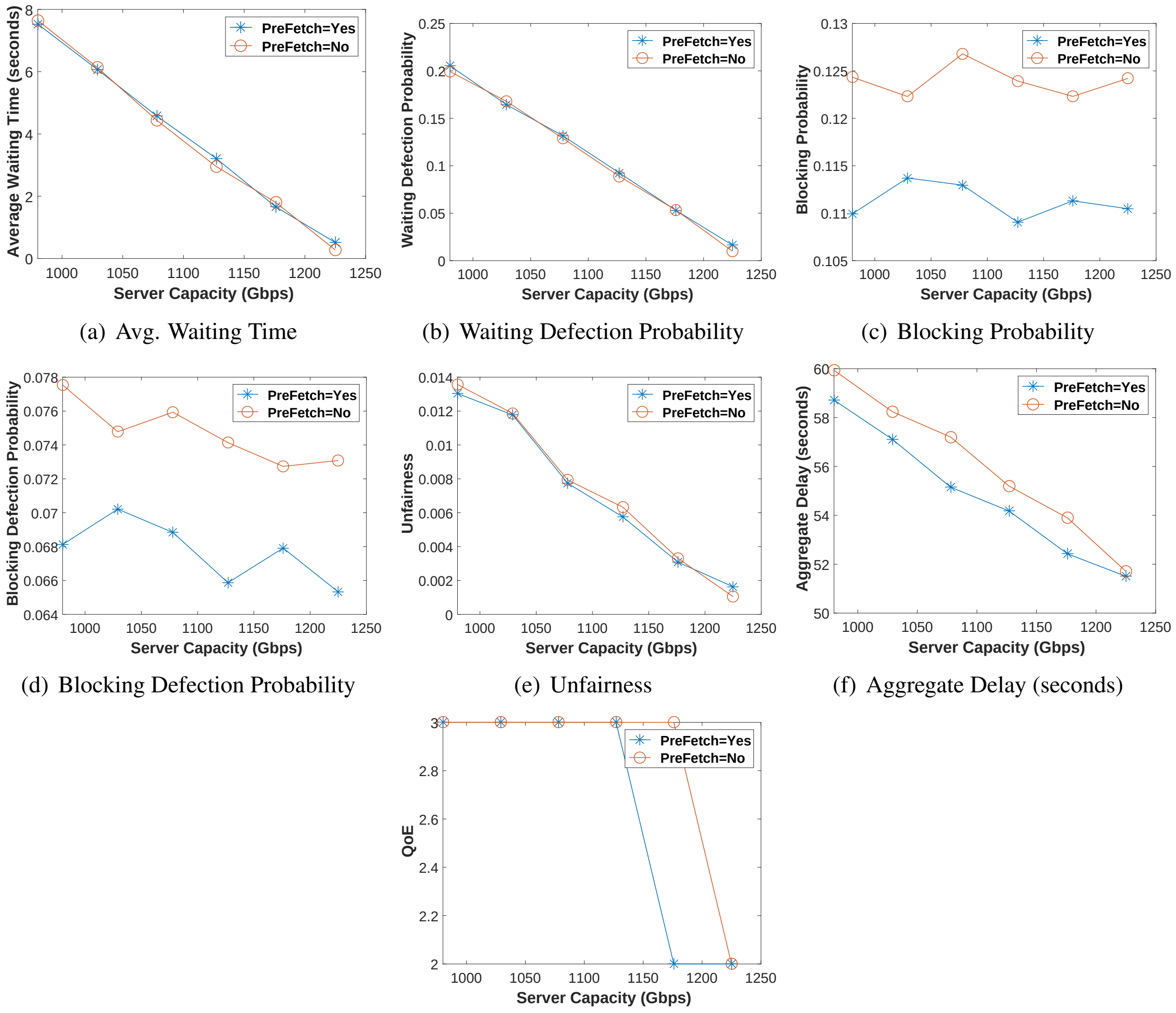
(f) Aggregate Delay (seconds)

(g) QoE

Figure 5.10: Impact of Pre-fetching [Purge Furthest]

for all systems.

- Increasing the customer's cache size improves both waiting and blocking metrics up to a certain size, beyond which, no significant improvement could be achieved due to diminishing returns.
- Blocking is major issue for any NVOD system. An efficient NVOD server should try to minimize blocking using different approaches.
- We experimented with a new purging algorithm (NoBookmark Purging) to avoid purging hotspot data already cached in the customer's cache. The proposed algorithm did not provide significant



(a) Avg. Waiting Time

(b) Waiting Defection Probability

(c) Blocking Probability

(d) Blocking Defection Probability

(e) Unfairness

(f) Aggregate Delay (seconds)

(g) QoE

Figure 5.11: Impact of Pre-fetching [NoBookmarks Purge]

improvement over existing ones because the majority jump to bookmarks in the considered workload are long jump forwards. This data tends not to be in the customer's cache. The proposed algorithm would be more effective in workloads where the majority of jump to bookmarks is either jump backward or short jump forward.

- Reserving multicast channels for caching hotspot data improves blocking metrics. However, there is a tradeoff between improving blocking and waiting metrics. Waiting metrics suffer as the percentage of reserved channels is increased.

- Fetching hotspot data for customers not listening to any stream reduces blocking probability by 13% and the blocking defection by 12%.

CHAPTER 6 SUMMARY AND FUTURE WORK

6.1 Introduction

In this chapter, we summarize the work that has been presented in the dissertation and list the related publications.

6.2 Summary

We developed an overall solution for interactive NVOD systems, where limited resources prevent the system from servicing all customers' requests. The interactive nature of recent workloads complicates matters further. Interactive requests are resource demanding. We analyze the system performance under a realistic workload using different stream merging techniques and scheduling policies. We consider a wide range of system parameters and studies their impact on the waiting and blocking metrics. In order to improve waiting customers experience, we propose a new scheduling policy for waiting customers that is fairer and delivers a descent performance.

ERMT outperforms other stream merging techniques in terms of waiting metrics. At low to moderate request rates, the amount of improvement over Patching is insignificant. Given that ERMT is of higher implementation complexity, we recommend using Patching for systems with low to moderate request rates.

Blocking is a major issue in any interactive NVOD systems and we propose a few techniques to minimize it. Particularly, we study the maximum I-Stream length (Threshold) that should be allowed in order to prevent a few requests from using the expensive I-Streams for a prolonged period of time, which starves other requests from a chance of using this valuable resource. Using a reasonable Threshold proves effective in improving blocking metrics. Moreover, we introduce an I-Stream provisioning policy to dynamically shift resources based on the system requirements at the time. The proposed policy proves to be highly effective in improving the overall system performance.

To account for both average waiting time and average blocking time, we introduce a new metric (Aggregate Delay).

We propose a novel client-side cache management policy for interactive NVOD systems. We utilize the customer's cache to service most interactive requests and reduce the load on the server. Using our proposed cache management policy, up to 92% of all interactive requests are serviced from the client's own cache without requiring additional server resources. We find that the overwhelming majority of jump backward and resume interactive requests are serviced from the cache. The jump forward interactive request cache hit rate is significantly lower than the other two interactive requests because some requested data is future data that is never cached. We propose three purging algorithms (PurgeOldest, PurgeFurthest, and AdaptivePurge) to clear data when the cache gets full. We study the impact of the purge block, which is the least amount of data to be cleared, on the system performance. We show that the cache purge block size has a significant impact on the blocking and waiting metrics. The optimal block size is system dependent. We study the impact of the cache size on the system performance. Increasing the cache size improves both waiting and blocking metrics until the cache size reaches a certain limit, beyond which, no significant improvement can be achieved due to diminishing returns. Another important decision is whether pausing customers continue to listen (C2L) to streams when the cache becomes full. Enabling C2L is highly desirable, especially for sufficiently large caches.

Finally, we study the effect of bookmarking on the system performance. A video segment that is searched and watched repeatedly is called a hotspot and is pointed to by a bookmark. We introduce three enhancements to improve system performance for workloads with bookmarking. Specifically, we propose a new purging algorithm to avoid purging hotspot data if it is already cached. The proposed algorithm did not provide significant improvement over existing ones because the majority jump to bookmarks in the considered workload are long jump forwards. This data tends not to be in the customer's cache. The proposed algorithm would be more effective in workloads where the majority of jump to bookmarks is either jump backward or short jump forward. Additionally, we reserve multicast channels to fetch hotspot data. This technique improves blocking metrics at the expense of waiting metrics. Furthermore, we fetch hotspot data for customers not listening to any stream. This technique improves blocking metrics without negatively impacting waiting metrics.

6.3 Future Work

Several tracks can be pursued from this dissertation. First, an array of scheduling policies for blocked customers could be proposed to minimize the time the customer spends in the blocking queue, which should improve the blocking metrics. The performance and unfairness of the blocking scheduling policy should be evaluated. Second, the impact of different workloads on the overall system performance can be studied. Particularly, interactive NOVD systems with high to very high request rate. Also, several techniques could be proposed to handle flash crowds. Third, a new generation of workload-aware cache management policies could be developed, where the cache management policy is optimized for a certain workload. Additionally, prediction algorithms for the cache segments that will be accessed next could be proposed and evaluated in terms of cache hit rate and blocking metrics. Several techniques for cache fragmentation reduction could be introduced and studied. Fourth, combining periodic broadcasting of hotspot data with stream merging techniques could be studied to improve the customer's experience in workloads with bookmarks. The effect of the network delay and congestion could be studied and evaluated on the system performance. What techniques should be used or proposed to handle network delay and congestion is still an open research topic. Finally, interactive NVOD systems with heterogeneous clients with different cache sizes and network bandwidth could be considered and their effect on the overall system performance can be studied.

6.4 List of Publications

- **Kamal Nayfeh** and Nabil Sarhan, Design and Analysis of Scalable and Interactive Near Video-on-Demand Systems, in Proceedings of *IEEE International Conference on Multimedia and Expo (ICME)*, San Jose, USA, July 2013, pp. 1-6.
- **Kamal Nayfeh** and Nabil J. Sarhan, A Scalable Solution for Interactive Near Video-on-Demand Systems, *IEEE Transactions on Circuits and Systems for Video Technology*. vol. 26, no. 10, pp. 1907-1916, Oct. 2016.
- **Kamal Nayfeh** and Nabil J. Sarhan, Client-Side Cache Management Policy for Scalable and Interactive Video Streaming. *IEEE International Conference on Multimedia and Expo (ICME)*, Seattle, USA, July 2016, pp. 1-6.

BIBLIOGRAPHY

[1] Sandvine-Incorporated, "Global Internet phenomena report," in *Sandvine*, 1H 2014, p. 1.

[2] Alexa, "`http://www.alexa.com/topsites/`".

[3] YouTube, "`http://www.youtube.com/t/press_statistics/`".

[4] Netflix, "`http://ir.netflix.com/`".

[5] Statista, "`http://www.statista.com/topics/842/netflix/`".

[6] Athula Balachandran, Vyas Sekar, Aditya Akella, and Srinivasan Seshan, "Analyzing the potential benefits of cdn augmentation strategies for internet video workloads," in *Proceedings of the Internet Measurement Conference*, Barcelona, Spain, 2013, IMC '13, pp. 43–56, ACM.

[7] Zhijie Shen, Jun Luo, Roger Zimmermann, and Athanasios Vasilakos, "Peer-to-peer media streaming: Insights and new developments," in *Proceedings of the IEEE 99(12)*, 2011, pp. 2089–2109.

[8] George Pallis and Athena Vakali, "Insight and perspectives for content delivery networks," *Communications of the ACM*, vol. 49, pp. 101–106, January 2006.

[9] Baochun Li Chuan Wu and Shuqiao Zhao, "Diagnosing network-wide p2p live streaming inefficiencies," in *Proceedings of IEEE INFOCOM*, April 2009, pp. 2731–2735.

[10] Vaneet Aggarwal, Robert Caldebank, Vijay Gopalakrishnan, Rittwik Jana, K. Ramakrishnan, and Fang Yu, "The effectiveness of intelligent scheduling for multicast video-on-demand," in *Proceedings of ACM Multimedia*, Beijing, China, 2009, pp. 421–430.

[11] K. Almeroth, "Multicast help wanted: From where and how much?," *Keynote Speech, Workshop on Peer-to-Peer Multicasting, Consumer Communications and Networking Conference*, January 2007.

[12] S. Ratnasamy and A. Ermolinskiy, "Revisiting ip multicast," in *Proceedings of ACM SIGCOMM Computer Communication Review*, Pisa, Italy, September 2006, vol. 36, pp. 15–26.

[13] Derek L. Eager, Mary K. Vernon, and John Zahorjan, “Bandwidth skimming: A technique for cost-effective Video-on-Demand,” in *Proceedings of Multimedia Computing and Networking Conference (MMCN)*, San Jose, California, USA, January 2000, pp. 206–215.

[14] Marcus Rocha, Marcelo Maia, Italo Cunha, Jussara Almeida, and Sergio Campos, “Scalable media streaming to interactive users,” in *Proceedings of ACM Multimedia*, Singapore, November 2005, pp. 966–975.

[15] Sung Soo Moon, Kyung Tae Kim, Seong Woo Lee, Hee Yong Youn, Ohyoung Song, and Ohyoung Song, “An efficient vod scheme combining fast broadcasting with patching.,” in *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, Busan, Korea, 2011, pp. 189–194.

[16] Wanfei Chi, Yongping Xiong, and Jian Ma, “Feature analysis and performance evaluation of streaming media scheduling algorithms in patching algorithm family,” in *Proceedings of International Conference on Computer Science and Network Technology (ICCSNT)*, Harbin, China, December 2011, vol. 4, pp. 2332–2335.

[17] Bashar Qudah and Nabil Sarhan, “Workload-aware resource sharing and cache management for scalable video streaming,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 3, pp. 386–396, March 2009.

[18] D. L. Eager, M. K. Vernon, and J. Zahorjan, “Optimal and efficient merging schedules for Video-on-Demand servers,” in *Proceedings of ACM Multimedia*, Orlando, Florida, USA, October 1999, pp. 199–202.

[19] Huadong Ma, G. Kang Shin, and Weibiao Wu, “Best-effort patching for multicast true VoD service,” *Multimedia Tools Appl.*, vol. 26, no. 1, pp. 101–122, 2005.

[20] Ying Cai and Kien A. Hua, “An efficient bandwidth-sharing technique for true video on demand

systems," in *Proceedings of ACM Multimedia*, Orlando, Florida, USA, October 1999, pp. 211–214.

[21] Charu C. Aggarwal, Joel L. Wolf, and Philip S. Yu, "The maximum factor queue length batching scheme for Video-on-Demand systems," *IEEE Transactions on Computers*, vol. 50, no. 2, pp. 97–110, February 2001.

[22] Kien A. Hua, Ying Cai, and Simon Sheu, "Patching: A multicast technique for true Video-on-Demand services," in *Proceedings of ACM Multimedia*, Bristol, United Kingdom, 1998, pp. 191–200.

[23] S. W. Carter and D. D. E. Long, "Improving Video-on-Demand server efficiency through stream tapping," in *the International Conference on Computer Communication and Networks (ICCCN)*, Las Vegas, NV, USA, September 1997, pp. 200–207.

[24] D. L. Eager, M. K. Vernon, and J. Zahorjan, "Minimizing bandwidth requirements for on-demand data delivery," *IEEE Trans. on Knowledge and Data Engineering*, vol. 13, no. 5, pp. 742–757, September 2001.

[25] Cristiano Costa, Italo Cunha, Alex Borges, Claudiney Ramos, Marcus Rocha, Jussara Almeida, and Berthier Ribeiro-Neto, "Analyzing client interactivity in streaming media," in *Proceedings of The World Wide Web Conference*, New York, USA, May 2004, pp. 534–543.

[26] S. Acharya, B. Smith, and P. Parnes, "Characterizing user access to videos on the world wide web," in *Proceedings of Multimedia Computing and Networking Conference (MMCN)*, San Jose, California, USA, January 2000, pp. 130–14.

[27] Eveline Veloso, Virg Almeida, Jr. Wagner Meira, Azer Bestavros, and Shudong Jin, "A hierarchical characterization of a live streaming media workload," *IEEE/ACM Transactions on Networking*, vol. 14, no. 1, pp. 133–146, 2006.

[28] Fan Qiu and Yi Cui, "An analysis of user behavior in online video streaming," in *Proceedings*

of ACM International Workshop on Very-Large-Scale Multimedia Corpus, Mining and Retrieval, Firenze, Italy, October 2010, pp. 49–54.

[29] Fabrcio Benevenuto, Adriano M. Pereira, Tiago Rodrigues, Virglio A. F. Almeida, Jussara M. Almeida, and Marcos Andr Gonalves, "Characterization and analysis of user profiles in online video sharing systems.," *Journal of Information and Data Management (JIDM)*, vol. 1, no. 2, pp. 261–276, 2010.

[30] Joonho Choi, A. Reaz, and B. Mukherjee, "A survey of user behavior in vod service and bandwidth-saving multicast streaming schemes," *Communications Surveys Tutorials, IEEE*, vol. 14, no. 1, pp. 156–169, February 2012.

[31] Florin Dobrian, Asad Awan, Dilip Joseph, Aditya Ganjam, Jibin Zhan, Vyas Sekar, Ion Stoica, and Hui Zhang, "Understanding the impact of video quality on user engagement," *Communications of the ACM*, vol. 56, no. 3, pp. 91–99, March 2013.

[32] Ludmila Cherkasova and Minaxi Gupta, "Characterizing locality, evolution, and life span of accesses in enterprise media server workloads," in *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video (NOSSDAV)*, Miami, Florida, USA, May 2002, pp. 33–42.

[33] Wanjium Liao and Victor O. K. Li, "The split and merge (SAM) protocol for interactive video-on-demand systems," in *Proceedings of the 16th Annual Joint Conference of the IEEE Computer and Communications Societies*, Kobe, Japan, April 1997, pp. 1349–1356.

[34] Emmanuel L. Abram Profeta and Kang G. Shin, "Providing unrestricted VCR functions in multicast video-on-demand servers," in *Proceedings of IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, Austin, Texas, USA, July 1998, pp. 66–75.

[35] Shahid Akhtar, Andre Beck, and Ivica Rimac, "Hifi: A hierarchical filtering algorithm for caching

of online video," in *Proceedings of the 23rd Annual ACM on Multimedia Conference*, Brisbane, Australia, 2015, MM '15, pp. 421–430, ACM.

[36] Konstantinos Poularakis and Leandros Tassiulas, "Optimal cooperative content placement algorithms in hierarchical cache topologies," in *Proceedings of Conference of Information Sciences and Systems (CISS)*, Princeton, NJ, USA, 2012, IEEE, pp. 1–6.

[37] Sem Borst, Varun Gupt, and Anwar Walid, "Distributed caching algorithms for content distribution networks," in *Proceedings of IEEE INFOCOM*, San Diego, California, USA, 2010, IEEE, pp. 1–9.

[38] Andrew Brampton, Andrew MacQuire, Michael Fry, Idris A. Rai, Nicholas J.P. Race, and Laurent Mathy, "Characterising and exploiting workloads of highly interactive video-on-demand," *Multimedia Systems*, vol. 15, no. 1, pp. 3–17, 2009.

[39] Andrew Brampton, Andrew MacQuire, Idris Rai, Nicholas JP Race, Laurent Mathy, and Michael Fry, "Characterising user interactivity for sports video-on-demand," in *Proceedings of Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Urbana, Illinois, USA, 2007.

[40] K. A. Hua and S. Sheu, "Skyscraper broadcasting: A new broadcasting scheme for metropolitan Video-on-Demand system," in *Proceedings of ACM SIGCOMM*, Cannes, France, September 1997, pp. 89–100.

[41] L. Juhn and L. Tseng, "Harmonic broadcasting for Video-on-Demand service," *IEEE Transactions on Broadcasting*, vol. 43, no. 3, pp. 268–271, September 1997.

[42] J.-F. Pâris, S. W. Carter, and D. D. E. Long, "Efficient broadcasting protocols for video on demand," in *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Montreal, Canada, July 1998, pp. 127–132.

[43] Cheng Huang, Ramaprabhu Janakiraman, and Lihao Xu, "Loss-resilient on-demand media stream-

ing using priority encoding," in *Proceedings of ACM Multimedia*, New York, NY, USA, October 2004, pp. 152–159.

[44] P. Gill, L. Shi, A. Mahanti, Z. Li, and D. Eager, "Scalable on-demand media streaming for heterogeneous clients," *ACM Transactions on Multimedia Computing, Communications, and Applications (ACM TOMCCAP)*, vol. 5, no. 1, pp. 1–24, October 2008.

[45] Lixin Gao, James F. Kurose, and Donald F. Towsley, "Efficient schemes for broadcasting popular videos," *Multimedia Systems*, vol. 8, no. 4, pp. 284–294, 2002.

[46] Asit Dan, Dinkar Sitaram, and Perwez Shahabuddin, "Scheduling policies for an on-demand video server with batching," in *Proceedings of ACM Multimedia*, San Francisco, California, USA, October 1994, pp. 391–398.

[47] A. Bar-Noy, G. Goshi, R.E. Ladner, and K. Tam, "Comparison of stream merging algorithms for Media-on-Demand," *Multimedia Systems Journal*, vol. 9, pp. 211–223, 2004.

[48] Nabil J. Sarhan and Bashar Qudah, "Efficient cost-based scheduling for scalable media streaming," in *Proceedings of Multimedia Computing and Networking Conference (MMCN)*, San Jose, California, USA, January 2007, pp. 327–334.

[49] Nabil J. Sarhan, Mohammad A. Alsmirat, and Musab Al-Hadrusi, "Waiting-time prediction in scalable on-demand video streaming," *ACM Transactions on Multimedia Computing, Communications, and Applications (ACM TOMCCAP)*, vol. 6, no. 2, pp. 1–24, March 2010.

[50] Yang Zhao, Ye Tian, and Yong Liu, "Extracting viewer interests for automated bookmarking in video-on-demand services," *Frontiers of Computer Science*, vol. 9, no. 3, pp. 415–430, 2015.

[51] Kamal Nayfeh and Nabil Sarhan, "Design and analysis of scalable and interactive near video-on-demand systems," in *Proceedings of IEEE International Conference on Multimedia and Expo (ICME)*, San Jose, California, USA, July 2013, pp. 1–6.

ABSTRACT

A SCALABLE SOLUTION FOR INTERACTIVE VIDEO STREAMING

by

KAMAL KAYED NAYFEH

May 2017

Adviser: Dr. Nabil Sarhan

Major: Computer Engineering

Degree: Doctor of Philosophy

This dissertation presents an overall solution for interactive Near Video On Demand (NVOD) systems, where limited server and network resources prevent the system from servicing all customers' requests. The interactive nature of recent workloads complicates matters further. Interactive requests require additional resources to be handled. This dissertation analyzes the system performance under a realistic workload using different stream merging techniques and scheduling policies. It considers a wide range of system parameters and studies their impact on the waiting and blocking metrics. In order to improve waiting customers experience, we propose a new scheduling policy for waiting customers that is fairer and delivers a descent performance.

As blocking is a major issue in interactive NVOD systems, we propose a few techniques to minimize it. In particular, we study the maximum Interactive Stream (I-Stream) length (Threshold) that should be allowed in order to prevent a few requests from using the expensive I-Streams for a prolonged period of time. Using a reasonable I-Stream threshold is very effective in improving blocking metrics. Moreover, we introduce an I-Stream provisioning policy to dynamically shift resources based on the system requirements at the time. The proposed policy is highly effective in improving the overall system performance. To account for both average waiting time and average blocking time, we introduce a new metric

(Aggregate Delay) .

We study the client-side cache management policy. We utilize the customer's cache to service most interactive requests, which reduces the load on the server. We propose three purging algorithms to clear data when the cache gets full. Purge Oldest removes the oldest data in the cache, whereas Purge Furthest clears the furthest data from the client's playback point. In contrast, Adaptive Purge tries to avoid purging any data that includes the customer's playback point or the playback point of any stream that is being listened to by the client. Additionally, we study the impact of the purge block, which is the least amount of data to be cleared, on the system performance.

Finally, we study the effect of bookmarking on the system performance. A video segment that is searched and watched repeatedly is called a hotspot and is pointed to by a bookmark. We introduce three enhancements to effectively support bookmarking. Specifically, we propose a new purging algorithm to avoid purging hotspot data if it is already cached. On top of that, we fetch hotspot data for customers not listening to any stream. Furthermore, we reserve multicast channels to fetch hotspot data.

AUTOBIOGRAPHICAL STATEMENT

Kamal Nayfeh received his B.S. degree from Jordan University of Science and Technology in 1991 and M.Sc. degree from Wayne State University in 1998. He is the software team supervisor at the Emissions and Fuel Economy Certification lab at Fiat Chrysler Automobiles (FCA) since 11/2015. Kamal was the software team leader at the same lab since 1/2012. Kamal worked as a software engineer at Chrysler LLC since 1998. Kamal is a Ph.D. candidate in the Multimedia Computing and Networking Research Lab. His research interests are Video on Demand Servers, streaming multimedia, cache management, and Interactive Video Operations.