Understanding and Efficiently Servicing HTTP Streaming Video Workloads

by

James Alexander Summers

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Live and on-demand video streaming has emerged as the most popular application for the Internet. One reason for this success is the pragmatic decision to use HTTP to deliver video content. However, while all web servers are capable of servicing HTTP streaming video workloads, web servers were not originally designed or optimized for video workloads. Web server research has concentrated on requests for small items that exhibit high locality, while video files are much larger and have a popularity distribution with a "long tail" of less popular content. Given the large number of servers needed to service millions of streaming video clients, there are large potential benefits from even small improvements in servicing HTTP streaming video workloads.

To investigate how web server implementations can be improved, we require a benchmark to analyze existing web servers and test alternate implementations, but no such HTTP streaming video benchmark exists. One reason for the lack of a benchmark is that video delivery is undergoing rapid evolution, so we devise a flexible methodology and tools for creating benchmarks that can be readily adapted to changes in HTTP video streaming methods. Using our methodology, we characterize YouTube traffic from early 2011 using several published studies and implement a benchmark to replicate this workload. We then demonstrate that three different widely-used web servers (Apache, nginx and the userver) are all poorly suited to servicing streaming video workloads. We modify the userver to use asynchronous serialized aggressive prefetching (ASAP). Aggressive prefetching uses a single large disk access to service multiple small sequential requests, and serialization prevents the kernel from interleaving disk accesses, which together greatly increase throughput. Using the modified userver, we show that characteristics of the workload and server affect the best prefetch size to use and we provide an algorithm that automatically finds a good prefetch size for a variety of workloads and server configurations.

We conduct our own characterization of an HTTP streaming video workload, using server logs obtained from Netflix. We study this workload because, in 2015, Netflix alone accounted for 37% of peak period North American Internet traffic. Netflix clients employ DASH (Dynamic Adaptive Streaming over HTTP) to switch between different bit rates based on changes in network and server conditions. We introduce the notion of *chains of sequential requests* to represent the spatial locality of workloads and find that even with DASH clients, the majority of bytes are requested sequentially. We characterize rate adaptation by separating sessions into *transient, stable* and *inactive* phases, each with distinct patterns of requests. We find that playback sessions are surprisingly stable; in aggregate, 5% of total session duration is spent in transient phases, 79% in stable and 16% in inactive phases. Finally we evaluate prefetch algorithms that exploit knowledge about workload characteristics by simulating the servicing of the Netflix workload. We show that the workload can be serviced with either 13% lower hard drive utilization or 48% less system memory than a prefetch algorithm that makes no use of workload characteristics.

Acknowledgements

I would like to thank my supervisor, Tim Brecht, for his guidance and detailed feedback on my work. Derek Eager was also deeply involved in my research. I have collaborated with many others while conducting my research and publishing the results, including Bernard Wong, Ben Cassell, Tyler Szepesi and Alex Gutarin.

I thank the members of my PhD committee for thoughtful and thorough reviews: Ken Salem, Paul Ward, and Carey Williamson.

I appreciate the financial support I received in the form of a Go-Bell Scholarship, a David R. Cheriton Scholarship and a scholarship awarded by Netflix. I am also grateful for the Doctoral Thesis Completion Award, largely for the deadline it provided to defend my PhD.

Finally I thank my wife for unwavering support and tolerance, and my parents who encouraged my extended education.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

The popularity of streaming video over the Internet is rapidly increasing. According to Sandvine's December 2015 Global Internet Phenomena Report [83], Netflix and YouTube account for 37.1% and 17.9% of peak fixed-line North American traffic, respectively. As of December 2015, more than 70% of peak downstream traffic is streaming video and audio, compared to 45.7% in September 2010 [82]. Outside of North America, streaming video and audio accounts for about 40% of peak Internet traffic, with higher percentages for countries with access to paid services like Netflix or BBC iPlayer [82].

There have been decades of research and development related to video streaming, but we believe there are two main reasons for the recent rapid increase in popularity: the widespread availability of fast broadband to homes and a new approach to video streaming over the Internet using HTTP (HyperText Transfer Protocol), the underlying protocol for the World Wide Web. In early video-on-demand systems, video was delivered by specialized servers, using video-specific protocols, over dedicated networks, to custom set-top boxes that played the videos for users [66, 36]. This was necessary because of the expense of system resources. System memory and disk storage were very expensive [74], and networks had relatively little bandwidth, so specialized algorithms and protocols were necessary to make it possible to deliver video at all.

As time passed, computer resources became cheaper and more abundant. Increased CPU power made set-top boxes unnecessary. The low cost of system memory enabled clients to buffer large amounts of video, making it possible to deliver video over a best-effort network. Since there is no longer a need for specialized real-time protocols, the practical decision was made to

use HTTP to deliver video, thus leveraging the existing HTTP ecosystem consisting of: routers, caches, CDNs (Content Distribution Networks), and web servers. This has the benefit of lowering the cost of developing video delivery infrastructure. Additionally, using the same protocol for the bulk of Internet traffic makes it easier for video to coexist with traditional HTTP web traffic [12].

However it is unclear how efficiently web servers deliver streaming video because they were not designed with this workload in mind. Prior to the introduction of video streaming, web sites had relatively little data, and it was often possible to store most or all of the data from a web site in the file cache of the web server. Web servers were optimized for the case where most content would be serviced from system memory. In contrast, video workloads cannot typically fit in memory due to the large average size of video files and the tendency for video collections to have long tail popularity distributions [96, 39, 64]. For example, a single web server used to store a fraction of Netflix's catalog may contain more than 200 TB of data on 36 Hard Drives and 6 SSDs [69]. Prior research into designing high-performance web servers by Harji, et al. [41] reveals that, in some cases, web servers should be implemented and tuned differently for workloads that can be serviced entirely from system memory compared to workloads that must be serviced largely from hard drives. For example, Harji et al. experimented with a non-video workload and found that using non-blocking `sendfile` provides higher server throughput when there is little disk I/O, while using the blocking version of `sendfile` provides higher throughput when a web server is disk-bound [42].

Given the large and growing quantity of HTTP-based streaming video traffic, it is essential to study this workload and determine whether or not the HTTP ecosystem is well suited to handling this workload. If it is possible to improve the efficiency of delivering streaming video, the increased capacity could be used to either accommodate future growth with the same hardware, or the current level of demand could be accommodated with less hardware. Much of the existing research in streaming video has concentrated on client implementations and network issues. In contrast, we closely investigate the implementation of the web servers used to deliver HTTP streaming video, about which there has been little research. Our expectation is that web server capacity for streaming video clients can be increased if web servers are specifically designed and tuned to service a highly sequential, disk-bound workload.

The remainder of this chapter is structured as follows. In Section 1.2, we discuss the goals of the thesis. Then we discuss the contributions of the thesis in Section 1.3, and finally summarize this chapter in Section 1.4

1.2 Goals

Our primary goal is to understand HTTP streaming video workloads and, if it is possible, to improve web servers in order to increase the number of HTTP streaming video clients that can be serviced with an acceptable level of quality. To ensure that our results are applicable to production environments, when possible, we will focus on testing actual web server implementations in a representative test environment. We now describe several subgoals:

Understand HTTP Streaming Video Server Workloads

Because we would like our work to be widely applicable, we study the workloads of web servers for the two largest streaming video services: YouTube, which streams short user-generated videos; and Netflix, which streams professionally-produced Movies and TV shows. These services account for 17.9% and 37.1% of peak North American traffic, respectively [83]. We obtain workload information from two different sources: published research papers that characterize HTTP streaming video workloads, and raw data obtained from production web servers. There are many published research studies about aspects of YouTube workloads, but no specific characterizations of the workload on individual web servers. For this reason, we infer the characteristics of a server workload from several papers that describe the implementation of YouTube clients and others that characterize the combined workload of different populations of YouTube clients. There is also a lack of published information about Netflix server workloads, so in this case, we obtained raw server logs under an NDA (Non-Disclosure Agreement) which we analyze to characterize the workload of Netflix servers.

Use Knowledge of Workload to Create a Benchmark

In order to test web servers in a laboratory environment, we create a benchmark to represent clients that are streaming HTTP video. The benchmark should be customizable, with a flexible methodology for creating benchmarks with different specifications. Flexibility is necessary because there are a variety of different video providers with different characteristics that we would like to represent, such as YouTube with mostly short-duration videos in low resolutions, and Netflix with longer videos that are available in higher resolutions. The capabilities of the different client applications available for these services differ, affecting the requests issued by clients [62, 60] and influencing user behaviour [29, 15, 21]. We would like to be able to create benchmarks that reflect these different factors, to experimentally determine the sensitivity of web server implementations to differences between current video services and to anticipated future workloads.

Evaluate and Possibly Improve Existing Web Servers

Using representative benchmarks to understand the operation of existing web servers, we find and evaluate opportunities for increasing efficiency. Some of the questions we are interested in include the best way to tune existing web server implementations, as well as the best way to store video content in the file systems of web servers. This will give us a baseline to use for evaluating potential implementation improvements. We intend to investigate how well web servers exploit the sequential request patterns of streaming video clients. Most existing studies investigate the use of caching as a means to improve efficiency of servicing the most popular content, while we will investigate strategies for efficiently servicing all content, including the less popular content that is prevalent due to the typical "long-tail" nature of video popularity.

1.3 Contributions

This thesis makes contributions in three areas. We develop a flexible methodology for creating streaming video benchmarks to enable the evaluation of web server implementations in a laboratory setting. We contribute to the better understanding of web server workloads for the two most-popular streaming services, YouTube and Netflix. Finally, using benchmarks to evaluate web server implementations, we develop techniques that can significantly increase the throughput, and therefore the capacity, of web servers. We now describe these contributions in detail.

1.3.1 Methodology for Creating Benchmarks

- We develop tools and methodologies to create video streaming benchmarks that can be used in a laboratory to experimentally evaluate the performance of web servers. The benchmark creation tools are flexible so that benchmarks can be created that represent the specifications of the wide variety of streaming video services that currently exist and to accommodate future changes in streaming video workloads.
- We demonstrate the necessity of throttling the network (using `dummynet` [80]) to simulate the variety of network connections common today, including home broadband connections and low-bandwidth cellular networks. We also show that it is necessary to emulate the pacing of requests from simulated clients to represent the important property that clients request data at approximately the same rate as they play it.
- We develop tools for controlling the placement of files on disk to ensure that we can create libraries of content in a repeatable manner, thereby ensuring that experimental results are

repeatable when it is necessary to recreate a catalog. It was necessary to develop the tools because the FreeBSD file system places files at unpredictable locations on disk and throughput can vary by 25% or more for a typical hard drive depending on whether the data is placed on inner or outer tracks.

- We devise an experimental methodology to determine the capacity of a web server: the maximum throughput of client requests that can be serviced with acceptable latency. We demonstrate the benefits of our procedure for warming the server cache and instituting ramp-up and ramp-down periods to ensure that experiments are repeatable and of sufficiently short duration to be practical, yet representative.

This work is described in detail in Chapter 3 and has been published in the proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR 2012) [91].

1.3.2 Creating a YouTube-like Benchmark

- Using existing published research, we derive a specification for the workload of a YouTube server. We then use that specification to create a benchmark using our methodology. We could find no direct characterizations of the workload at YouTube servers, so we infer a server workload specification by combining the information from several papers that examine network traffic from either individual YouTube clients [79, 60, 7], or collections of YouTube clients in different geographical areas [34, 23, 19, 39].

The work deriving the YouTube server workload specification is in Chapter 3 and is also part of the paper presented at the 5th Annual International Systems and Storage Conference (SYSTOR 2012) [91].

1.3.3 Understanding and Improving Web Server Implementations

- Using the YouTube-like benchmark, we examine the performance of three web servers (`Apache` [52], `nginx` [67], and the `userver` [43]). We found that all three exhibit relatively poor throughput. Although streaming video workloads are largely sequential, and hard drives service sequential requests with relatively high throughput, we observe poor hard drive throughput, similar to what might be expected from non-sequential disk accesses. We were surprised to find that the operating system was not able to effectively prefetch and/or schedule the disk requests resulting from this workload. Our work demonstrates the value of the benchmark and being able to run experiments with existing web server implementations in a laboratory setting.

- We determine the best way to store content on disk. Two methods have been used to store content on disk: 1) storing a single video in multiple separate files, used by Apple's Live Streaming [12]; and 2) using a single file for the entire video, used by Microsoft's Smooth Streaming [12]. We show that it is possible to access the disk with significantly higher throughput when using a single file per video because of the reduced amount of file system metadata that must be read.
- We show that relatively simple modifications to the `userver` web server, to perform aggressive prefetching and sequentialized disk accesses, significantly increases throughput. Initial experiments reveal that the modification of the `userver` enables throughput to be doubled when using a prefetch size of 2 MB, but only when the content is stored using a single file per video.
- We use the flexibility of our benchmark methodology to conduct a series of experiments to determine the sensitivity of the prefetch algorithm to different system and workload characteristics, such as: 1) the amount of available system memory, 2) video popularity distribution, 3) video bit rates, and 4) hard drive characteristics. We show that the best prefetch size varies significantly and provides up to 4 times higher throughput than without application-level prefetching, and up to 3 times higher throughput than a prefetch size that is too large.
- Because the benefit from choosing the best prefetch size is large, and because that size may be different for each system or change in workload characteristics, manually tuning the prefetch size requires significant effort. Therefore, we develop an algorithm that dynamically and automatically determines the best prefetch size. We show that our automated algorithm is equally as effective as exhaustive manual tuning.
- Using mathematical analysis, we determine that the overhead of reading content from hard drives is minimized when the prefetch size is proportional to the square root of the file bit rate.

The relatively poor efficiency of existing web servers and the benefits of serialized aggressive prefetching are described in Chapter 3 and have been published in the proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR 2012) [91]. The evaluation of different options for storing content on hard drives is in Chapter 3 and we have published those results in the proceedings of the 22nd International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2012) [92]. The factors that affect the prefetch size and the automated algorithm are presented in Chapter 4. These results have been published in the proceedings of the 7th ACM International Systems and Storage Conference (SYSTOR 2014) [90].

1.3.4 Netflix Server Workload Characterization

- Our analysis and experimentation with our YouTube-like benchmark have been successful, but there are two issues with the benchmark. First, because there was no available literature specifically about the workload of HTTP streaming video servers and because we had no access to production servers, we were required to infer the workload on YouTube servers from information about individual clients and collections of clients. Second, the sources available to us for our work in 2012 were based on studies of YouTube dating 2011 and earlier when YouTube clients rarely (for less than 5% of viewing sessions [34]) changed the bit rate while playing content. YouTube clients switched to using DASH (Dynamic Adaptive Streaming over HTTP [86]) at the end of 2012 [94]. DASH-based clients adapt to changes in available network and system bandwidth by switching the bit rate of content that the clients are playing, and it is unclear how these changes will impact servers. We remedy both of these issues by obtaining log files from two different production Netflix servers. We analyze these log files to characterize more recent server workloads that result from servicing DASH-based clients.
- We introduce two abstractions to aid in our analysis of the Netflix server log files. The log files are very complicated because of the use of DASH by Netflix clients, and because of the large number of different client implementations that exist, so these abstractions are critical tools for understanding the workload. We develop algorithms for forming *chains* of sequential requests as a tool for analyzing the spatial locality of client requests in the workload, regardless of the different methods clients use to issue HTTP requests [62]. We develop methods for recognizing *phases* in the complicated patterns of client requests for content with different bit rates, to aid in understanding the effects of the DASH algorithms.
- One concern with DASH clients is that if they change bit rates often, this will reduce the spatial locality of the workload and make aggressive prefetching less effective. From the analysis of phases, we determine that Netflix clients seldom change bit rates, despite the use of DASH. This is reflected in the distribution of chain lengths. Although there are large numbers of short chains (about 60% are less than or equal to 1 MB), the bulk of content is downloaded in long chains (chains greater than or equal to 10 MB account for about 95% of bytes downloaded). This implies that spatial locality of the Netflix workload is not significantly worse than our YouTube-like benchmark, suggesting that the techniques we developed to improve throughput when servicing our benchmark are likely to be applicable to more modern clients and servers.
- We devise a method for analyzing the measured distribution of chain lengths in order to compare the efficiency of different prefetch algorithms. Using this technique, we find that a relatively large prefetch size enables the Netflix workload to be serviced efficiently, and

that using both smaller and larger prefetch sizes is less efficient, similar to the findings of our experiments with the YouTube-like benchmark. We also find that by using workload-specific characteristics, such as the bit rate of the requested video and the probability that chains will be long or short, we can adjust prefetch sizes to either reduce hard drive utilization by 13% or system memory use by 48%, compared to a prefetch algorithm that does not use workload information.

This work on characterizing and analyzing the Netflix workload appears in Chapter 5 and is published in the proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC 2016) [89].

1.4 Chapter Summary

According to Cisco, Internet video accounted for 64% of global consumer Internet traffic in 2014, and they project that the proportion will rise to 80% by 2019 [27]. Most of this traffic is delivered from web servers, so for this reason, we study the video web server workloads to determine if existing web server implementations can be improved to increase the number of HTTP streaming video clients that can be serviced with the same resources.

In this thesis, we analyze YouTube traffic from 2011 using published research studies to create a specification of a workload and use it to generate a YouTube-like benchmark. We conduct experiments using the benchmark to show that by applying aggressive prefetching, more than double the number of clients can be serviced using the same server hardware. We design and evaluate an automated algorithm that dynamically determines a prefetch size that matches the best manual tuning and increases server capacity by up to 4 times, depending on the workload and server hardware resources. Finally, we obtain server logs from two Netflix servers to compare the workload from clients that implement DASH to the benchmark, which represents non-adaptive clients. We show, by a novel analysis of the chains in the Netflix workload, that the algorithms we developed experimentally using the YouTube-like benchmark are also likely to be effective for Netflix servers.

Chapter 2

Background and Related Work

2.1 Background

In this section, we describe how video is delivered over the Internet. We provide a brief history of how network video delivery has developed from using proprietary networks owned by video providers, to an HTTP-based system that delivers video from existing web servers over the Internet.

In this thesis we will use the term *title* rather than video to refer to the object that a user selects (and a client downloads) from the catalog of a video service. We adopt this term to reduce confusion because titles have both video and audio components and we use the term "video" specifically to refer to a component of a title. Depending on the service, a title could be a user-generated video, a music video, a TV show, or a Movie.

2.1.1 Video Delivery Methods

Video-on-demand systems were first introduced by telecommunication providers and cable companies as an alternative to broadcast TV or renting physical media from video stores [66, 74]. Unlike broadcast TV, subscribers could select a title from a catalog and start viewing whenever they wished, without the inconvenience of leaving their homes to visit a video store. In addition to starting titles whenever they wanted, subscribers could also perform *trick play* operations such as pause, resume, rewind and fast-forward, just as they could with rented physical media.

The earliest video delivery systems were limited by a lack of resources. Memory was expensive (as much as $50 per MB [74]) and disk drives were small, slow and expensive ($3,000

for a drive with a capacity of 9 GB, a 10 MB/s transfer rate and an average seek latency of 15 ms [74]). Designing a video delivery system that was commercially viable was difficult due to the resource constraints, so service providers used proprietary video servers, managed networks (such as ATM), and provided set-top boxes to subscribers to view the content [66, 36]. These systems were *push-based*, with the video server responsible for sending content to the clients, and ensuring that content was transmitted fast enough to avoid gaps in video playback.

As technology improved and became cheaper, it became possible to deliver video with acceptable quality over the unmanaged Internet. Broadband networks with sufficient bandwidth became widely available to consumers, making dedicated video networks unnecessary. Proprietary set-top boxes could be replaced with PCs or cheaper general-purpose client devices with enough processing power to decode video streams and sufficient memory to buffer adequate amounts of data to hide network latencies. In addition to the benefit of being able to watch titles in many more locations, because it was no longer necessary to own a dedicated network to supply titles, it became possible for many new video suppliers to be created. This resulted in new alternative suppliers for movies and TV shows like Hulu, Amazon and Netflix, and also suppliers for new types of video, such as user-generated video sites like YouTube, Yahoo! video and Dailymotion. These new systems are *pull-based*, with the clients responsible for requesting data from the server and timing the requests so that titles can be displayed without pausing or stuttering.

There are several different methods for delivering titles over the Internet. Video streaming technologies either use streaming-specific protocols, such as RTP (Real Time Transport Protocol), RTCP (Real Time Control Protocol) and RTSP (Real Time Streaming Protocol), or simply use the standard HTTP (HyperText Transfer Protocol) [12]. Our work concentrates on HTTP-based video delivery because of its growing popularity and its widespread use including the most popular video suppliers, like YouTube and Netflix.

One method of delivering titles, called *Progressive Download*, is popular for its simplicity. A client requests the entire file containing the title using the standard HTTP protocol, but the client does not wait until the entire file is downloaded before playing the title. Instead, a client starts playing the title once enough data has been transferred, and continues playback at the same time as the remainder of the file is delivered. The limitation of progressive download is that the entire title is requested as a single file, so it can be difficult to efficiently provide some trick play features unless the entire file has been downloaded and stored.

An additional drawback of progressive download is that it can be wasteful of network and server resources. Users usually stop watching a title before the end is reached. For one typical YouTube workload [34], only 20% of users watch to the end of the title and 60% of users watch less than 20% of a title. To avoid downloading content that will not be viewed, *pacing* is used to

reduce the average download rate so that it is just above the bit rate of the content. With pacing, when a user unpredictably ends playback, there is a limited amount of unplayed content that was read from storage and sent over the network. Pacing is a defining feature of HTTP streaming video and can either be implemented on the server or on the clients, as we describe in detail in Section 2.2.2.

To address the shortcomings of progressive download, *streaming* was introduced. In this case, the title is divided into many *segments* of equal playback time, and the client requests a sequence of segments (e.g. using HTTP range requests) rather than a single file. Clients are able to download segments from any portion of the title at any time, making operations like skipping ahead or back in a title easy and efficient to implement. Also, if the title is available in multiple quality levels (requiring playback at different bit rates) the clients can use *rate adaptation* (or *DASH* for Dynamic Adaptive Streaming over HTTP) to compensate for dynamic variations in network bandwidth and server conditions by switching between different versions of the title, choosing the highest-quality version that can be delivered using the available network bandwidth and server resources [87, 53]. This also makes it possible to switch to different servers to hide failures and ensure a good quality of experience for the user.

2.1.2 Video Client Implementation

The difference between push-based and pull-based video delivery systems is whether the clients or servers direct the content delivery system. With a pull-based HTTP video delivery system, the clients are responsible for requesting title content and satisfying the real-time requirements of displaying a title, while the server is simply responsible for responding to client requests as quickly as possible. Because clients operate over the unmanaged Internet, they cannot depend on guaranteed network bandwidth or latencies, so they must maintain a *playout* buffer of title content to be able to compensate for occasional reductions in network bandwidth or delays at the server. It is important for clients to ensure the playout buffer does not under-run (called *rebuffering*), because users are sensitive to annoying delays in displaying video and users will stop viewing if video quality is poor [32].

The details of how a client displays a title differ for specific services such as YouTube [34, 4], Netflix [3], and Hulu [2], but the process generally consists of two phases. First the client acquires a *manifest* that provides information about all the segments that make up a specific title, then the client sends requests to the video server, decodes the title content received, and displays the contents for the user [12].

The manifest file contains information about which server contains the title content and the URL (uniform resource locator) to use to access each segment. The manifest includes informa-

tion about all the different versions of a title, including different resolutions, encoding methods, quality levels and other information [12]. For large video service providers with multiple servers and multiple copies of titles, there is an opportunity to manage demand over the servers by generating manifests specifically tailored to each client.

Viewing Session

After the client receives the manifest, it starts sending requests to the video server and displaying the video for the user as content is downloaded. Figure 2.1 shows the lifetime of a viewing *session* for a typical video service. The bottom rectangles show three requests issued to the server, with delays between the requests. The top rectangle shows playback of the title. The rectangles representing the requests are narrower than corresponding playback areas because content must be downloaded faster than it is played to avoid rebuffering. The delays that are added between requests implement pacing and ensure that the average download rate is close to the bit rate of the title content being played. To start a session, the client establishes a network connection to the server (at time t_0) and requests the first segment in the title, illustrated by the first box on the bottom of the diagram, labelled *fill*. Playback does not start until enough content is downloaded into the playout buffer (at time t_1), which accounts for the delay before playback starts. While playing the title, the client periodically replenishes the playout buffer by requesting subsequent segments. At some unpredictable point, the user will stop viewing and cause the client to terminate the connection. When playback is terminated, the downloaded content depicted by the dashed line is discarded. The amount discarded could have been much larger if pacing had not been used to limit the average download rate.

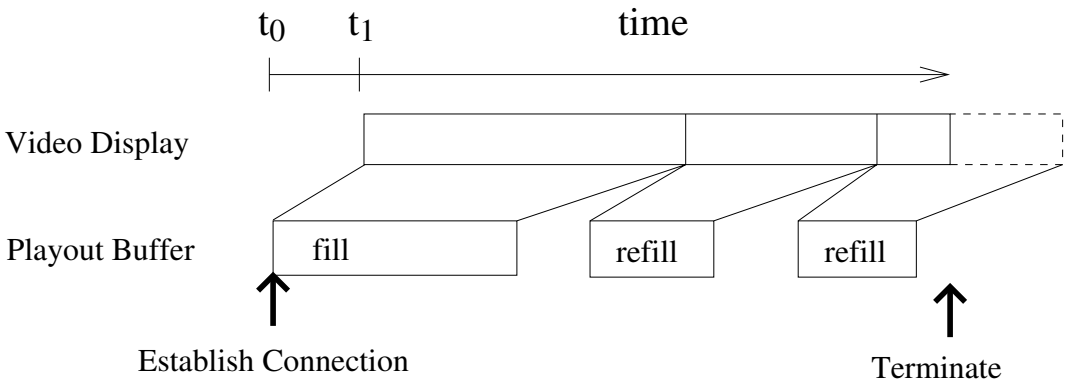


Figure 2.1: Example of a client video session

Not all sessions have this simple form of requesting content sequentially from a single file. Potentially there are user events such as pausing or skipping to a new title location, which cause the client to delay requests or to discard the current contents of the playout buffer and refill it starting at a new point in the title. Also the client may switch files as part of rate adaptation.

2.1.3 Video Server Implementation

Because title content is delivered using the standard HTTP protocol, any web server can be used. However, web servers were not originally designed and tuned for streaming video workloads. Prior to the use of web servers for video streaming, web server workloads could be largely cached in system memory, so web servers may not be effective at servicing disk-bound workloads. There are results that show that the best web server architecture for disk-bound workloads is different from when the workload is largely cached [42], so it is likely that some web server architectures may be better suited to video workloads than more conventional HTTP workloads. In this thesis, we found it necessary to devise the ASAP architecture described in Section 3.7.1.

At a basic level, web servers have a simple design [75]. Figure 2.2 shows the basic steps involved in processing a single viewing sessions from a client. A web server is expected to process hundreds or thousands of these sessions concurrently.

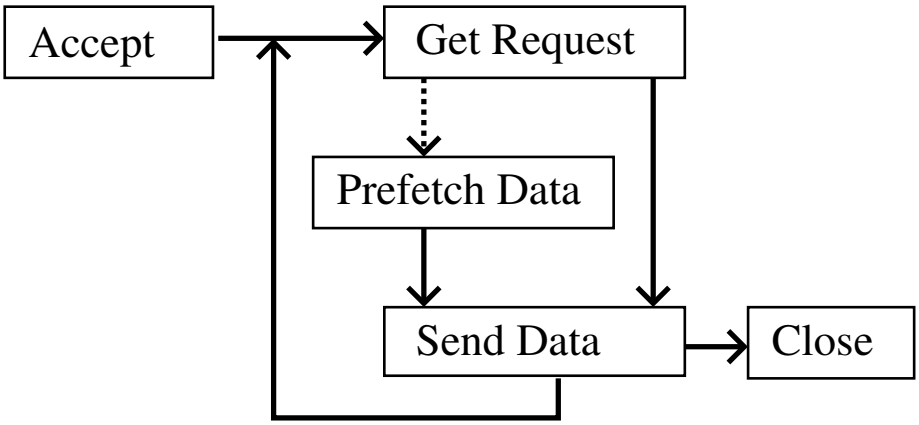


Figure 2.2: Processing an HTTP connection

To start a viewing session, a client establishes one or more TCP connections to use for issuing requests and receiving data in response. After the server accepts the connection, it waits to receive requests from the client. When a request arrives, the server determines if the request can be satisfied out of memory; that is, whether the requested data was read previously in the same or

different session and retained in a web server or filesystem cache. If the requested data is present in memory, the server can proceed to send the data to the client.

If the requested data is not in memory, the server must read the data from a storage device, such as a hard drive or SSD (Solid-State Drive). To minimize overhead [85], the server may read more data from the file system than was actually requested, called *read-ahead* [95], or *prefetch-on-miss* [57]. The server reads extra data in the expectation that it will soon be requested, and by reading the data ahead of time, the server amortizes the overhead of reading over more than one request. Not all servers prefetch only in response to cache misses, which is the reason we use a dotted line between the *Get Request* and *Prefetch Data* boxes in Figure 2.2. If data is always prefetched in advance of it being requested, it is possible to eliminate the latency between receiving a request and sending data in response. This strategy is called *prefetch-on-hit* [57] because a prefetch is triggered when a request can still be serviced using previously prefetched data (i.e., a cache hit), in the expectation that the newly prefetched data will be available just before it is requested. In this thesis, while we are concerned with prefetch latency, our main goal is to ensure that web servers are able to service requests with high throughput, thus servicing as many clients as possible. Our techniques can trade increased latency in exchange for higher throughput because streaming video clients are designed to tolerate the network latencies that are possible when using the HTTP protocol over a best-effort Internet

The server waits for requests and services them as they arrive until the client eventually stops requesting data and the client ends the session by closing connections. The unpredictable timing of when a client will stop requesting data may negate some of the throughput gains from prefetching because storage device throughput is wasted when the server prefetches data that is not subsequently requested by a client. Also, if a server prefetches data too soon, data can be evicted by the memory management algorithm and will need to be read again. Choosing a good prefetch size that minimizes *wasted prefetches* and *evictions* while being large enough to improve efficiency is a difficult problem that is studied in this thesis.

2.1.4 Video Control Plane

Many HTTP streaming video services are very large. Catalogs contain many titles, content is available in many different resolutions and bit rates, and there are large numbers of subscribers. Catalogs are often too large to fit on a single server, even if a single server could provide sufficient throughput for the large number of subscribers. For these reasons, streaming video service providers require large numbers of web servers, either in their own data centres or at external CDNs.

The service provider typically manages the large number of servers and the use of CDNs

with a *video control plane*. One function of the control plane is to determine how to partition the titles in the catalog among the servers and CDNs, and the number of copies to utilize. These decisions may be influenced by the anticipated popularity of the titles, the throughput capabilities of servers, expected server failures, and the geographic location of servers. The other function of the control plane is to direct clients to the best server that contains the content the user wishes to watch, based on the current demands on the servers. A coordinated network control plane may assign clients to different video servers based on which servers are in or out of service, the relative server loads, and the dynamic conditions of the network [59].

The policies of the control plane can have a great effect on throughput. Poor placement of titles can result in overloaded servers, so the control plane logic tries to anticipate demand, then partition the catalog among servers so that the load is balanced. This is a challenging problem that we do not address in this thesis. Instead, we simply recognize that a control plane chooses titles for each server and chooses which clients are directed to each server, and we consider the resulting workload on the server. It is important to note that the workload on an individual server is not necessarily similar to the overall workload, but can be shaped by the policies implemented by the control plane.

2.2 Related Work

Our goal is to understand HTTP streaming video workloads so that we can discover ways to improve the efficiency of the web servers that are used to deliver streaming video. Delivering video using a best-effort network protocol is a difficult problem, and as a result, it is a popular area of research. However, much of this research specifically investigates client implementation issues, network issues, and the design of control planes, rather than server implementation issues. In the absence of specific information about the implementation of web servers for HTTP streaming video, we examine the general characteristics of HTTP streaming video workloads to help understand where web servers might have problems servicing the workload. We also examine research and methods for improving web servers for workloads that are similar to streaming video workloads. In the following sections, we specifically examine the existing research literature in four areas.

Characteristics of Videos and User Viewing Sessions: This is information about what titles are watched and how those titles are watched. This includes characteristics such as title duration and bit rates that determine the workload, and characteristics such as popularity, ratings or title category that allow us to predict which titles will be chosen. An important characteristic of user behaviour is the *viewing ratio* which is the duration of a viewing session divided by the duration

of the title, which has a great effect on the spatial locality of the workload. We are also interested in which parts of titles are watched, to determine if user interest is skewed to certain parts of the video, like the start or end.

Implementation Details for HTTP Streaming Video: In addition to title and session characteristics, server efficiency may be affected by client implementation details. For example, web servers may use different algorithms and kernel syscalls when responding to HTTP range requests, compared to requests for an entire file. We need to know implementation details in addition to title and session characteristics to implement a benchmark that accurately reproduces streaming video traffic.

Existing Benchmarks: We investigate existing tools that can be used to create an HTTP streaming video benchmark. We found studies fell into two categories; workload generators that create workload specifications, and benchmark platforms that can be used to reproduce the specified workloads.

Improving Server Efficiency: Studies that specifically investigate issues with delivering HTTP streaming video typically recommend improvements to the control plane (such as the use of proxy caches or peer to peer video delivery) or recommend changes in client implementations. There is little research that specifically investigates the implementation of web servers for HTTP streaming video workloads. However, we examine a number of more generic studies that describe techniques for improving disk throughput for sequential workloads.

2.2.1 Workload Studies

An HTTP streaming video workload is shaped by the characteristics of titles and sessions. The duration of a title and its bit rate place a ceiling on session duration and spatial locality of requests. There are two broad categories of streaming video services that have been studied: short video services such as YouTube that supply user-generated videos and music videos; and long video services such as Netflix that supply TV shows and movies. In addition to title characteristics, session characteristics such as viewing duration and starting position in a title shape workloads further. The most frequently studied workload is that of YouTube. Chowdhury and Makaroff [26] provide a survey of studies that characterize titles and sessions for YouTube. We describe many different YouTube studies in the following sections, because they each contain different characterizations. We include the year that each YouTube study was conducted because

the YouTube workload has evolved over time. We also describe studies for long video services and short video services other than YouTube.

We organize the different workload studies based on which of four general methods is used to collect the information needed to characterize titles and sessions: 1) Receiving information from instrumented client applications. 2) Capturing streaming video traffic at the edge of networks and analyzing the traces. 3) Acquiring public information about titles via APIs or scraping web pages. 4) Obtaining server logs from service providers. We discuss the workload characteristics that are described in each study and, where appropriate, comment on how we used the information in this thesis.

Instrumenting Clients

A direct approach to determining the behaviour of users is to instrument the client applications that are used to view streaming video. We examined these studies to determine how much of a title is typically watched in a session, which portions of the titles are watched, how often users pause playback or skip to a new title position, and if there are workload characteristics that affect these behaviours.

Dobrian, et al. [32] investigate the effect of video quality on viewing times. We are interested in this information because users are the ultimate judge of the quality of streaming video, so we would like to know which quality metrics are most important to users, so we can use those metrics for evaluating the effectiveness of servers. The authors collect information about 1 million users and 2 million sessions from several popular service providers that are representative of Internet video traffic. They analyze the effect of join time (the delay before the title starts playing), buffering ratio (the percentage of time that playback freezes, waiting to refill the playout buffer), average bit rate and rendering quality to determine which have the greatest effect on viewing times. They find that the buffering ratio has the largest effect; a 1% increase in the buffering ratio decreases viewing time by 1 to 3 minutes. Join time is also negatively correlated with viewing time. As a result, for our experiments in Chapter 4, we model the user behaviour of terminating viewing sessions when there is excessive rebuffering, by aborting sessions when the response time for a request exceeds 10 seconds.

Chen, et al. [22] obtained trace data from 100,000 titles watched in 100 million sessions using clients of the Chinese PPLive VoD service. They specifically analyze the data for title characteristics that can be used to predict how long a user will watch a particular title, as a percentage of title duration. They find that, on average over all sessions, 61% of a title is viewed and the percentage that is viewed depends on the title duration, popularity, and category (e.g. movie, cartoon, sport, news). Notably, they found that users watch the first episode in a series

for a shorter amount of time than subsequent episodes, and that there is an inverse relationship between watching time and title duration. The drawback with this study is that the authors excluded 53% of sessions that involve playback interruptions or user trick play, so the sample they use may not be representative.

Chen, et al. [21] examine viewing ratios of titles, as well as the *browsing* behaviour of users (i.e., a series of short viewing sessions while the user samples titles, followed by a long viewing session that marks the end of browsing). They collected data from instrumented clients for about 540 million sessions from 49 million different users of the Tencent VoD service, which provides both long-duration and short-duration titles. This study contains detailed information about events that occur during sessions: rebuffering events and user skips and pauses. They find the percentage of a title that is viewed is inversely related to popularity. The median viewing ratio for the less popular titles is about 40% of the title, while it is about 70% for the more popular titles. The category of the title (e.g. movie, music video, TV show, or sporting event) also affects the session duration. The logs contain information about rebuffering events, with 80% of sessions having no rebuffering events and 8% having more than 3 rebuffering events. Skips occur in 62% of sessions, there are an average of 6 skips per session, 80% of skips are for intervals that are less than 5 minutes of content, and rebuffering events tend to induce skips. For the Netflix workload in Section 5.4.3, we measured an average of about 2 *transient phases* per session (where transient phases are caused by events such as skips or switching bit rates for rate adaptation), so skips appear to be more frequent for this Tencent workload than for our Netflix workload. We do not currently model browsing for our workload methodology.

These studies show that users behave similarly for all of these different video services. Many sessions (30-60%) last for less than 20% of the title, and only 10-30% of sessions are for more than 90% of content. Unfortunately, we could not use these studies directly to create a workload specification because they are incomplete. They contain only information about user behaviour during sessions, and not the characteristics of the titles (such as popularity or duration) that are watched.

Capturing Traffic from Clients

Although instrumenting clients is an excellent method for obtaining detailed information about user behaviour, researchers do not usually have access to the proprietary information collected from instrumented clients, or have the access necessary to add their own instrumentation to client implementations. As an alternative, a number of researchers collect information by capturing all the raw HTTP requests from clients on a subnetwork, and can infer the behaviour of users by analyzing the raw requests. This analysis can be difficult. For example, the raw HTTP requests

refer to URLs, and it may not be clear which title is stored at a particular URL. If the URL being used by a client changes, there are two potential causes: it may indicate that the user switched titles (and therefore ended one session and started another), or it may indicate that the client changed to accessing a different bit rate version of the same title as part of rate adaptation.

Finamore, et al. [34] collect information about HTTP requests in 2011 from two different types of YouTube clients that use different access mechanisms, named PC-player and Mobile-player. In general, PC-player clients run in browsers on desktop computer and laptop computers, and Mobile-player is used with smart phones and all other devices, including non-mobile devices such as smart TVs that run custom application on iOS or Android operating systems. We found this study to contain the most complete information about streaming video workloads of all the studies we examined. We use the characterizations of titles and sessions from this study for the workload specification in Chapter 3. The authors use `tstat` to classify traffic and generate fine-grained flow-level statistics about viewing sessions. They generate information about sessions, including the amount of time spent viewing videos and the percentage of title content that is downloaded. They find resolution switches are rare; only 5% of PC-player and 0.5% of Mobile-player sessions include a resolution switch, and fewer than 0.3% of PC-player sessions have more than one resolution switch. Because of these low numbers for resolution switches, we do not model adaptation for our workload specification in Section 3.2.2. The authors find that 60% of videos are watched for less than 20% of their duration, and only 20% of users watch the entire video. They also provide some low level details of bandwidth use by clients. The PC-player uses a single persistent TCP connection to download a burst of data at a high transfer rate at the start of a session, then downloads data at a steady lower transfer rate after a few seconds, with the server responsible for pacing the download rate. The mobile-player similarly downloads at a high transfer rate for the first few seconds, then transitions to a mode where each individual segment of data is downloaded using an HTTP range request over a new TCP connection. The mobile-player client is responsible for pacing the download rate and issues requests on approximately a 3 second interval, with about 1 second being required to download the requested segment. The authors do not provide information about the popularity distribution of different titles, so we must obtain that information from other studies.

Gill, et al. [39] provide a comprehensive characterization of the YouTube workload in 2007, with results that are similar to Finamore, et al. [34]. The authors collect information about 626,000 requests for 324,000 unique videos on their University campus and characterize the aggregate average download rate by clients over different periods of time. They find that title popularity has a Zipf distribution with $\alpha = 0.56$, and that fewer than 10% of titles viewed one day are also viewed the next day. Titles are encoded with nearly uniform bit rates with a mean of 394 Kbps and median of 328 Kbps. Based on this information, we use a single fixed bit rate for our workload specification in Section 3.2.1. The main drawback with their work is that although

both title durations and session durations are measured, the authors do not relate the two; whereas Finamore, et al. [34] provide sessions durations as a fraction of title duration, which is a form more convenient for specifying a workload.

Zink, et al. [98] captured traces of YouTube traffic at the edge of a campus network over six different time periods in 2007. They measure the popularity of titles and find the traces all have similar Zipf distributions, and about 75% of titles are requested a single time over a 24 hour period. They also measure the distribution of session durations, and find average durations between 75 and 99 seconds for different traces.

These three YouTube workload studies give similar characterizations (where they overlap) despite being collected from different networks and different years, and are therefore the source of most of the information used to create a workload specification for the benchmark in Section 3.2.

In addition to YouTube studies, there are also studies that analyze traces collected at the edge of networks from Netflix clients. Laterman [51] collected traces from both Netflix and Twitch clients over 5 months in 2015 at the University of Calgary. Laterman analyzes 305 million HTTP requests issued by Netflix clients using 14.3 million TCP connections over the collection period to characterize the requests, to analyze the popularity distribution of the titles that were selected, and to show how the volume of requests fluctuates over time. The measured characteristics in this study are similar to the results we obtain in Section 5.2.4. The data collected at the University of Calgary is not representative of the workload of a Netflix server since it combines requests to potentially many different servers. Additionally, the aggregate demand from the University of Calgary is much lower than that for Netflix servers. From our two Netflix server logs (as described in Table 5.1), we determine that a single day of demand for the Netflix catalog server is equal to about 20 days of demand from the University of Calgary and one day of demand for a flash cache server is equal to 50 days of demand from the University of Calgary.

Crawling and Scraping Web Sites

Some researchers are not interested in session characteristics and analyze only information about titles, such as number of views (popularity), user-assigned rating, number of user comments, duration and bit rate. In these cases, it is not necessary to instrument clients or capture client network traffic, but instead researchers can obtain information about titles by crawling and scraping the websites of service providers. Some service providers also offer APIs that supply this information directly. Researchers can observe dynamic changes in title characteristics, if information is periodically sampled. In particular, the evolution of popularity can be characterized by sampling view counts periodically. For this thesis, we are interested in the short-term characteristics

of streaming video workloads and not longer-term characteristics such as the evolution of title popularity, so we are mainly interested in these studies to find detailed information about title characteristics such as duration and bit rates.

Cheng, et al. [24] use a crawler to study YouTube videos. We use their information about the distribution of title durations for our workload specification in Section 3.2.1 because it is comprehensive and clearly presented. A major methodological issue they must solve is that YouTube does not provide a list of all available videos. YouTube provides some short lists of titles, such as "Most Viewed" and "Top Rated", and for each individual title, YouTube web pages include lists of related videos. The authors construct a list of videos to track by starting from a small set of "top rated" lists and then expand that list by visiting the individual web pages for videos in the list to add 20 videos that are related to the original list. They repeat the procedure of visiting pages and adding related videos to a depth of four. After repeating this procedure many times on many different days, the researchers obtained information about more than 2.7 million videos (out of an estimated 43.5 million). Using this information, the authors produce detailed distributions of characteristics of the videos in the list, including a distribution of title durations, title bit rates, and the number of views for each title (representing the popularity of titles). This study provides the most detailed information about title durations and bit rates of the studies we examined, and although the data was collected in 2007, a recent study by Che, et al. [20] found that the title durations in 2013 are largely the same, except that there were more titles longer than 10 minutes because YouTube raised its limit on title duration to 15 minutes.

Cha, et al. [19] investigate the popularity of YouTube titles, as well as Daum UCC, a Korean user generated video site in 2007. They perform a detailed analysis of video popularity, to determine whether the popularity distribution is Zipf, log-normal, or exponential. They discuss the potential mechanisms that could result in the different distributions and determine that the popularity distribution is Zipf-like, with an exponential cutoff in the tail of the distribution that they speculate is caused by a lack of information about available titles. The authors analyze dynamic changes in popularity to determine that caching the top 16% of the most popular videos, augmented by the 10,000 most popular videos from the previous day, would result in an 83% cache hit rate. They also determine that 5% of videos are requested at least once every 10 minutes, and because these are the most popular files, peer-to-peer delivery could reduce the server workload by 41%. These findings could be applied to improve the implementation of control plane algorithms, and therefore could have an indirect effect on the workloads of individual servers.

Mitra, et al. [64] investigate 4 short video services, Dailymotion, Yahoo!, Veoh and Metacafe by collecting information about 1.9 million titles in total, and show the popularity distributions are largely similar to the YouTube workload that is characterized by the three studies we described in the previous section. Mitra, et al. provide two measures of popularity, views since upload and viewing rate (defined as the increase in number of views between two crawls, di-

vided by the interval between crawls). The measured popularity distributions for viewing rates are much more Zipf-like than the distributions for views since upload, which tend to have pronounced cutoffs. The authors argue the viewing rate is more relevant when analyzing caching algorithms because an old title that has many accumulated views may appear to be more popular than a recent video which has not had time to accumulate many views, leading to incorrect caching decisions if the current viewing rate is actually higher for the recent title. This reasoning suggests that viewing rate distributions should be used to specify workloads. For this reason, in Chapter 3, we use a Zipf distribution without cutoff to represent title popularity.

For all of these studies, the authors could measure the popularity of titles and characteristics such as title duration and bit rate using publicly-available information. The main drawback is that the collected lists of videos do not necessarily include all videos available from the different services, so the lists are potentially biased by the method used to construct lists of videos, which in most cases are skewed towards the popular titles. Borghol, et al. discuss biased sampling of YouTube videos and provide a method for unbiased sampling [14]. Another drawback is the lack of information about which servers are used to store the titles, so there is no way to determine which titles are supplied from any particular server. Since there is no information about the workloads of individual servers, we assume the workload of an individual server is similar to the aggregate workload (except for scale) when we create a workload specification in Chapter 3. In Chapter 5, we characterize the workload of an individual Netflix server.

Using Server Logs

The most accurate information about server workloads is contained in server logs and traces, which capture activity from all clients, regardless of the network location of the clients. Service providers usually collect more information than they publicly supply, and it can be difficult to convince service providers to share this proprietary information. Ideally, we would be able to determine the workload of an individual server from these studies, but in all cases, the authors analyze the aggregated logs from all servers and do not provide information about individual servers. Our primary interest in these studies is to obtain more information about sessions, such as session duration, which portions of titles are watched and user actions such as pauses and skips.

Yu, et al. [96] obtained logs containing information about 21 million sessions that occurred over 219 days from all servers of the Chinese PowerInfo VoD service, which supplies titles with long durations. Our primary interest in this study is information about how the shape of the Zipf popularity distribution changes from day to day, and about session durations, to help develop the benchmark in Chapter 3. For each session, the server logs contain a title ID, the session start and

end time, and a code identifying the server that supplied the content. They report changes in the arrival rates of new user sessions over time, the popularity of titles, changes in popularity over the "life span" of a title, and session durations (in minutes). The authors find that 53% of sessions are shorter than 10 minutes. They find that sessions tend to be shorter for more popular titles than less popular titles, which is the converse of the findings of Chen, et al. [21]. As a result, for our workload specification of Section 3.2.2, we choose session durations independently of title popularity. The popularity distribution is Zipf-like with cutoff, and the distribution varies only slightly from day to day; the Zipf α parameter has a mean of 0.8 and standard deviation of 0.07. Because of the relatively small daily variation in the Zipf α parameter, we use a constant Zipf parameter for our workload specification in Chapter 3.

Costa, et al. [29] obtained server logs from two streaming video services in 2004: eTeach, which contains educational content (including lectures) at the University of Wisconsin-Madison, and UOL, a VoD service in Latin America. The title durations are relatively short, 75% of eTeach titles are shorter than 10 minutes and 97% of UOL titles are shorter than 5 minutes. The authors measure the hourly and daily variation in server load, the popularity distribution of content, and session inter-arrival times. They find that the popularity distribution is Zipf and session arrivals are exponentially distributed. As a result, we issue requests using a Poisson process (which produces an exponential distribution) for our benchmark, as described in Section 3.4. They make detailed measurements of sessions: the starting points in the title, intervals between pauses, the duration of pauses, the number of skips, and the skip intervals. They did a thorough analysis of skips and pauses during sessions and found that when the title duration is longer than 15 minutes, sessions include an average of 3 pauses and 2 skips, with skips backward slightly more likely than skips forward. For sessions with multiple skips or pauses, users are likely to repeat the same action (e.g., skip forward repeatedly). Less than 20% of sessions shorter than 5 minutes include skips, so for our short video workload in Section 3.2.2 we use a simple session model that does not include skips. This is the only study we could find that provides information about which portions of titles are viewed during sessions. The distribution of access frequencies for the educational content is roughly uniform for popular files (usually lectures longer than 15 minutes) and skewed towards the start of titles for less popular titles. For entertainment titles, the distribution of access frequencies is skewed towards the start regardless of popularity. We find that the title access pattern for the Netflix workload, shown in Section 5.2.2, are similar to the pattern for popular educational content rather than short entertainment titles.

Summary

Considering all of the studies using different sources of information, we conclude: 1) the popularity of titles usually has a Zipf-like distribution (with an α value likely between 0.5 and 0.8),

possibly with a cutoff in the tail, depending on how the data is collected. 2) Most sessions do not last for the full duration of a title, and a significant number of sessions are very short. 3) For short video services, such as YouTube, users usually start watching from the beginning of the title. For titles with long durations and live video streams, the distribution of starting points varies depending on the content type and access frequencies. They are either uniform, or skewed towards earlier title positions. 4) Skips and pauses appear to be common for sessions with long-duration titles, and uncommon for short-duration titles. This information helps us to determine which characteristics to model for our workload methodology in Chapter 3. In particular, for the workload specification, we decided that all sessions start from the beginning of a title and do not include skips, which is a reasonable simplification for a short video service. The lack of specific information about the workload of individual servers, and the differences between short-duration and long-duration video services, are the main motivations for characterizing the Netflix server workload in Chapter 5.

2.2.2 HTTP Streaming Video Implementation Details

The workload studies in the previous section provide high-level characterizations of the workload, but not much low-level information about how the implementation of clients or servers generate these workload characteristics. In particular, we are interested in how pacing is implemented. Depending on the service, pacing can be implemented either by the clients or the server. We are interested in low level details for two reasons. First, the benchmark of Chapter 3 must represent HTTP streaming video clients accurately, so it must implement one of these methods for pacing. Second, knowing the implementation details of Netflix clients will aid in the interpretation of server logs in Chapter 5, which contain records of individual HTTP requests.

Client-based Pacing Implementations

With client-based pacing, the client is responsible for controlling the download rate from a conventional web server. Begen, et al. [12] provide an overview of implementation issues for streaming clients and describe a number of implementations that perform rate adaptation: 3GPP (3rd Generation Partnership Project), Microsoft Smooth Streaming, and Apple HTTP Live streaming. Their study provides information about request sizes and inter-arrival intervals between requests that are necessary to reproduce client request patterns accurately. This information about a number of specific services enables us to infer common implementation details that we mimic with our workload generator in Section 3.4.

Mansy, et al. [62] study clients for YouTube, Netflix, and Hulu over WiFi and 3G networks and measure inter-arrival rates of requests. This study was published in 2014, after we created our workload specification in Chapter 3, so we use this study primarily to augment our interpretation of the Netflix server logs in Chapter 5 and to understand the evolution of YouTube. The authors describe the on-off behaviour of clients: clients start in a buffering phase where content is downloaded as fast as possible, followed by a steady state phase where clients alternate between an on state when content is being downloaded and an off state when no content is downloaded. They describe two methods of implementing on-off behaviour. For iOS clients of all three streaming services and Android YouTube clients, the on state corresponds to the download of an individual range request and the off state occurs between the completion of one request and issuance of the next request. For Android clients of Netflix and Hulu, the on-off behaviour occurs inherently as part of TCP's flow control mechanism.

The authors also provide information about the number of TCP connections used. Some clients establish a new TCP connection for each request, some use a persistent connection for all requests (starting a new connection when the bit rate changes), and some use a new connection for every few requests. In Section 5.2.4, we perform an analysis of request sizes and TCP connection use for many different Netflix clients including iOS, Android and many others, and we observe the frequent use of parallel TCP connections which were not reported by Mansy, et al.

Server-based Pacing for YouTube

For the studies of the YouTube workload that we describe in Section 2.2.1, the majority of content is delivered with the YouTube servers controlling the pacing. The PC-player described by Finamore, et al. [34] and the player described by Gill, et al. [39] request an entire file at once (potentially starting from a non-zero file offset) and the server sends data at an average rate slightly above the bit rate of the content.

Alcock and Nelson [7] analyze packet header traces in 2011 to provide detailed information about the server-based pacing method used by YouTube, which they call *block sending*. At the start of a session, YouTube servers send an initial burst of 32 seconds of content, followed by the periodic transmission of content in 64 KB blocks at an interval timed to result in an average download rate about equal to the bit rate. They show that this technique causes network congestion and is the cause of 40% of packet loss events for YouTube traffic, resulting in between 1% and 1.5% of YouTube content being retransmitted unnecessarily. Ghobadi, et al. [35] propose a solution to this problem that involves placing an upper bound on the TCP congestion window size based on the bit rate of the content and the round-trip time. The authors (some of whom work for Google) implemented this solution in production servers and reduced retransmissions

by up to 50%, illustrating the value of modifying servers based on the analysis of workload characterizations.

For our benchmark in Chapter 3, the workload specification is based on YouTube because there are more studies that contain information pertinent to constructing a workload specification and because the short title durations support short experiment durations. However, we do not implement server-based pacing, but instead use clients that support range requests for pacing, because it is easier to observe HTTP requests than changes in the TCP receiver window size. Additionally, server-based pacing requires customized servers that keep state information for each client, and it is easier to use client-based pacing with conventional web servers.

2.2.3 Streaming Video Benchmarks

We require an HTTP streaming video benchmark to use for experiments with web servers in a lab environment. However, we could not find a suitable benchmark. In order to create our own benchmark, we survey the literature to find existing benchmarks that could be customized or modified. We also examine HTTP streaming workload generators that generate artificial traces that could be the basis for a benchmark.

Workload Specification Generators

Zink, et al. [98], in a study that characterizes a YouTube workload from traces captured in 2007, explain how to use their measured data to generate a list of sessions. For each session, a title is chosen based on the measured popularity distribution, and the number of bytes of content to download during the session is chosen based on the measured distribution of session lengths from the traces. Using a measured distribution to determine the number of bytes transferred in a session is inflexible because it is not necessarily clear how changing workload parameters such as title duration or the mix of bit rates will change the distribution of session lengths in bytes. In Chapter 3, we compute the number of bytes as the product of the duration of the title, a viewing fraction and a bit rate, where the title duration distribution, viewing fraction distribution and bit rate are all specified independently and could be obtained from different workload studies or based on projections of future workloads.

Abhari and Soraya [1] crawl and scrape the YouTube website in 2008 to collect information about 43,000 titles for use with their workload generator. The workload generator consists of two parts. One part is a server workload generator that specifies the sizes of files required to store the titles in the catalog. The other part makes use of the "related video" lists to simulate a series of session with the same user: either selecting a title from the entire catalog using the

measured popularity distribution, or from the related video list for that title. The drawback of this generator is that session durations are not modelled because session duration information cannot be determined from crawling the YouTube website. We do not model related video lists in our benchmark. The popularity boost from being on a related list is reflected in the overall popularity distribution, so the effect of related video lists is reflected in our benchmark, even though related lists are not explicitly modelled.

Tang, et al. [93] developed the MediSyn streaming workload generator. Their focus is on generating long-term workloads. They allow for new titles to be added to the catalog and changes to the popularity of titles as they age after the time of introduction. There are two steps to workload generation: 1) file property generation, where the duration, bit rate and popularity are assigned from measured distributions; and 2) file access generation where sessions are created by choosing a file based on popularity (accounting for the age of the title) and choosing a session duration. We are interested in conducting experiments that last at most a few hours (because of the need to run multiple experiments for comparisons), where it is reasonable to assume that the catalog contents and title popularity distributions do not change. In Section 3.2, our method for choosing title and session characteristics is similar to MediSyn; but the modelling of long term popularity shifts would be more useful for testing control plane algorithms for placing content on servers.

Benchmark Implementation Platforms

General-purpose workload generators, such as SWORD [9], BenchLab [18], and TCSB [28] can be used to benchmark HTTP-based systems but these generators have not been configured specifically to produce HTTP streaming video. If we were to use one of these benchmark platforms, we would still have to generate a workload specification in the required format and to provide clients. An advantage to some of these benchmarks, like BenchLab, is that they use existing client implementations and potentially eliminate the need to implement a simulated client. The disadvantages are that it may be difficult to reproduce the infrastructure needed to support an existing client (such as a control plane for supplying manifests), and it may be impossible to bypass DRM (digital rights management) to obtain properly encoded files for playback. For our benchmark, we use simulated clients so that we do not need to supply additional infrastructure and so that we do not need to use any specific encoding for our many test files. We use `httperf` [65], an existing HTTP traffic generator; this approach is shared by several other benchmarks, such as StreamGen [61] (which is targeted at distributed applications with complex data flows).

2.2.4 Improving Server Implementations

In order to achieve our goal of improving web server implementations, we examined existing studies that address this issue. Traditionally, the performance of the disk has not been a limiting factor for web server performance. In many cases, the workloads used to evaluate different web servers are not much bigger than the file cache, so there is little disk access and the results may not apply to workloads where that is not true. For example Choi, et al. [25] tested with 5 different traces, with a cache hit rate of 88.3% for one trace and more than 99.5% for the others. Pai, et al. [75] performed experiments over a range of workload sized from 15-150 MB, which is not much larger than the 128 MB of system memory for their experiments. Because streaming video workloads are typically too large to fit in system memory, the long tail of less popular content that is requested must be read from hard drives. Techniques such as disk I/O scheduling and aggressive prefetching are critical to maximize the throughput of reading less popular content from hard drives. Disk performance is not often discussed in research specifically about web servers [42], so we must consider more general research for improving disk performance. Additionally, existing studies usually do not examine a streaming video workload. There are studies that examine streaming workloads or sequential workloads, but they assume that the entire file is always downloaded, which is not true of streaming video workloads.

In the next section, we describe research about the use of prefetching to exploit the spatial locality of a streaming video workload to improve disk throughput, we discuss memory management issues that arise from aggressive prefetching, and we consider how title content should be stored on hard drives.

Prefetching

The main benefit of using prefetching with a streaming video workload is that it reduces the overhead of reading from disk. Hard drives have high sequential throughput, but it takes a relatively large amount of time to move to a new location on disk, due to the time required to move the hard drive head and the rotational latency. The main challenge with prefetching is to decide how much data to prefetch at a time. Considering only the throughput of the hard drive, prefetches should be as large as possible. However, the duration of viewing sessions may not be predictable, since users often stop watching the video before they reach the end, so there is a concern that some prefetched data will never be requested by a client, and therefore the effort spent prefetching may be wasted. Another concern is that if too much data is prefetched, there may be cache pollution or the page replacement algorithm will evict prefetched content before it can be requested.

The requests issued by individual clients may be highly sequential, but when hundreds or thousands of clients access the same server concurrently, the requests from clients are inter-

leaved and requests arrive at the server in a non-sequential, seemingly random order. Recognizing if there are sequential accesses in the server workload, to find opportunities for increasing throughput by prefetching data, can be a difficult problem. Li, et al. [54] survey different definitions of sequentiality that have been used, and discuss three features that differentiate algorithms for detecting sequentiality in workloads: 1) how many interleaved sequences of requests can be recognized, 2) whether strided access is recognized or if requests must be strictly consecutive, and 3) whether there is a limit on the inter-arrival time of requests. In Section 5.3, we devise the *chain* abstraction to represent the sequentiality of the Netflix workload, where we describe our algorithm for recognizing sequentiality to create chains. In terms of the sequentiality features used by Li, et al., our algorithm for creating chains: 1) recognizes any number of concurrent chains, 2) requires that requests be strictly contiguous, and 3) permits at most a 40 second (but adjustable) inter-arrival gap. In addition to the features described by Li, et al., our chain algorithm accepts out-of-order requests, which they do not.

Li, et al. [56] provide a 2-competitive algorithm for determining the prefetch size. They consider two costs of reading data: the time required to transfer the data from a hard drive and the time it takes to reposition a hard drive head between the end of one read and the start of the next (including the seek time, rotational delay and other overhead). They prove that if the prefetch size is chosen so that the transfer time for the data is equal to the average repositioning time, the time required to service the workload will be at most double that of the optimal offline algorithm (that has knowledge of exactly how much data will be requested during each session). This algorithm is not always the best choice, particularly if there is not enough system memory available to store prefetched data. We use this rule in Section 4.5 to choose the initial prefetch size for our automated algorithm, but as the experimental results show, the initial prefetch size is not always the best choice.

Panagiotakis, et al. [76] use a large fixed size for prefetching. They compare different prefetch sizes and show that, when servicing 100 sequential streams, the best prefetch size improves throughput by 4 times compared to not prefetching. Their experiments differ from ours in Chapter 4 because they use an artificial workload of equal length streams rather than a streaming video workload with a wide variety of session lengths. Also, their server only reads data from the hard drive and does not transmit data over the network to clients, which affects the accuracy of their results because as we show in Section 3.6.2, server throughput can be different when servicing clients with limited network bandwidth. They do not provide a method for choosing the best prefetch size, or any of the other parameters for their prefetch algorithm, but instead illustrate the effect of varying those parameters. Our automated algorithm described in Section 4.2, has similarly large benefits as Panagiotakis, et al., in the best case increasing throughput by up to 5.2 times compared to not prefetching.

For large streaming video services, particularly those that implement rate adaptation, clients

request content at many different bit rates. Gill and Bathen [37] analyze a model where there are endless steady streams at different bit rates and prefetched content is stored in an LRU memory buffer, and prove that the prefetch size should be proportional to the bit rate. We discuss these findings in detail in Section 4.3 and present our own analysis using different assumptions than Gill and Bathen. In contrast, we show that the prefetch size should be set proportional to the square root of the bit rate.

Memory Management

A streaming video workload can potentially benefit from both caching and prefetching; with the use of caching to reduce disk requests for the most popular content and prefetching to increase disk throughput to service the long tail of less popular content. Both caching and prefetching require the use of system memory to store content until is requested by clients, and because system memory is limited, it is necessary to balance these two uses. Depending on the memory management algorithm, overly large prefetches can cause either cache pollution (i.e., a decrease in the cache hit rate) or cause prefetched data to be evicted before it can be requested.

Bhatia, et al. [13] provide an algorithm that adjusts the prefetch size so that there are no evictions, essentially setting the prefetch size based on feedback from the memory management algorithm. We found that this algorithm is too restrictive. We include experiments in Section 4.5 where server throughput is higher when a small number of evictions are allowed. We find there is a need to balance the benefit of higher disk throughput against the cost of more evictions.

The problem of integrating prefetching and caching has been widely studied. There are a number of studies that propose algorithms for combining prefetching and caching [78, 17, 50], but these studies consider single-threaded workloads, so it is unclear whether the results hold for web server workloads with hundreds or thousands of concurrent clients. Other authors propose an approach that does not combine the caching and prefetching algorithm, but instead divides system memory into separate pools, for exclusive use by either prefetching or caching. These algorithms, such as DULO [46] and SARC [38], resize the pools to obtain equal marginal benefits from prefetching and caching.

For this thesis, we do not investigate memory management issues, because these algorithms are implemented in the kernel. We expect that the kernel we use has a memory manager that is adequate for our prefetch algorithms and we closely monitor the memory manager for signs of excessive cache pollution or evictions.

Storage for Titles

The increase in throughput from prefetching will be greatly reduced if title content is not stored sequentially on disk. This is an important consideration because the placement of data on a hard drive is determined by the file system. Normally, the kernel does not allow an application to control the placement of data or to easily determine the placement of a file on disk. Because of this, the choice of file system can impact the performance of servers, and the best choice of file system depends on the characteristics of the workload, as demonstrated by Sehgal, et al. [84]. We are able to obtain good performance from the default FreeBSD file system using procedures outlined in Section 3.5, so we do not investigate file systems any further in this thesis.

2.3 Chapter Summary

In this chapter, we describe the methods used to deliver HTTP streaming video. We describe pull-based streaming video client applications, explaining how the client communicates with the control plane to determine which server to use for downloading content. Then we describe how the client maintains a playout buffer of content in case of excessive network latency. Importantly, there is a limit to the amount of content the client places in the playout buffer, after which it refills the buffer at the bit rate of the content as it is watched by the user (called pacing).

We then survey the work related to servicing HTTP streaming video workloads using four categories: characteristics of titles and sessions in the workload, implementation details for HTTP streaming video clients and servers, existing benchmarks, and general methods for improving web server efficiency. We did not find any single study that could provide all details of the workload at an individual server, so we combined several studies to create the workload specification in Section 3.2. We describe the available benchmarks with workloads similar to HTTP streaming video, but found none that adequately reproduce the client behaviours in the studies we examined.

We examine work related to improving server performance in three areas: 1) We survey the literature about sequential prefetching, since a streaming video workload is highly sequential. 2) Since both caching and prefetching are important to servicing streaming workloads, we describe the research on strategies for partitioning system memory between these two uses. 3) We consider potential issues with the file system.

In the next chapter, we make use of the extensive research about YouTube streaming video workloads to create a benchmark that produces a workload that is meant to approximate that seen by a YouTube server.

Chapter 3

Workload Methodology

In this chapter, we create a benchmark for testing web server implementations. We then use the benchmark to experimentally evaluate three different web server implementations and show we can double the throughput of a web server when servicing an HTTP streaming video workload.

There are three specific goals for our web server benchmark. First, it should generate web server loads that are representative of what would be observed at HTTP streaming video web servers in real deployments. Second, the benchmark should measure the performance of web servers by running for a sufficient length of time and by isolating and removing the effects of starting and stopping experiments. Lastly, since we anticipate using benchmarks to test a variety of web servers, including different design and implementation alternatives, we ensure that each experiment does not run for too long, otherwise the benchmark is unlikely to be used.

We use the benchmark to evaluate three different web servers: `Apache`, `nginx` and the `userver`. We determine that they all exhibit unexpectedly poor disk throughput. We then determine that there are two issues that contribute to the poor throughput: storing titles separated into multiple files (called *chunks*) rather than a single file and that the kernel interleaves concurrent disk requests which prevents kernel prefetching algorithms from improving disk throughput. We modify the `userver` web server to perform *Asynchronous Serialized Aggressive Prefetching* (*ASAP*), and show that in concert with storing titles in a single file, this technique doubles server throughput compared to web servers that do not implement ASAP.

Our methodology for creating benchmarks and running experiments was published in the proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR 2012) [91] and appears in Sections 3.1 through 3.4 and 3.6. We published our investigation of whether to store titles in chunks or unchunked in the proceedings of the 22nd International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2012) [92]

which appears in Section 3.5. The experimental results in Section 3.7 are collectively reported in both of these papers.

3.1 Overview of the Methodology

Our primary objective for this chapter is to devise flexible tools and methodologies for constructing HTTP streaming video benchmarks that can be used to understand, compare and improve the performance of web servers. Although the work in this thesis is focused on web server performance, another potential future use for the benchmark is to evaluate the use of proxy caches and CDNs.

Our methodology for generating a workload and benchmark is divided into separate phases:

1. Specify a workload: This requires characterizing a workload by understanding what are believed to be the important observations and parameters (including distributions) required to sufficiently characterize a workload.

2. Construct a workload: This is accomplished by supplying the workload specification to a workload construction program which creates artificial traces (`wsesslogs`) that are used by `httperf` to generate the desired load.

3. Set up the experiment: This phase includes setting up the networking, client, and server environments. This includes populating the server with files that will be requested and setting up `dummynet` on all of the client machines to mimic the desired mix of networks. These steps are performed using information from the workload specification.

4. Run the benchmark: The final phase is to execute the benchmark and collect the performance data.

Figure 3.1 illustrates these different phases. It is important to note that, in this chapter, we generate workloads based on the information we have obtained from several different papers that characterize YouTube video requests. Later, in Chapter 5, we characterize Netflix traffic, which could be used to create a workload specification for use with this methodology.

From our survey of existing workload characterization papers, we discovered a number of common issues in serving HTTP streaming video workloads that we want to capture in our benchmark design.

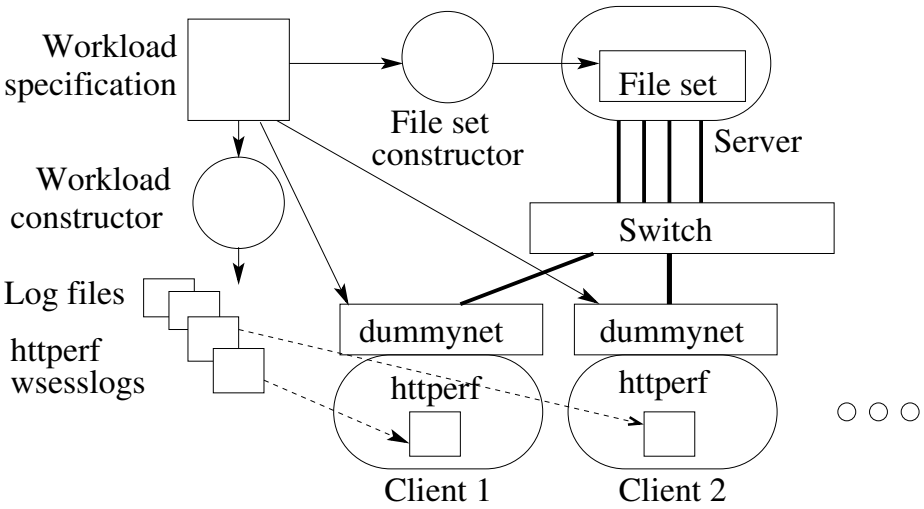


Figure 3.1: Overview of the methodology

- Titles are not always watched to the end, and we want to capture that behaviour. This is done by generating client sessions that ask for an appropriate fraction of a title.
- We expect that many streaming video workloads will be disk intensive, so the methods used to store content in the file system can have a large influence on performance. Very little information is available about the specific methods used to store content on servers, so we have made this portion of the methodology flexible. In particular, we experimentally evaluate whether to store content in multiple files (which we refer to as *chunked*) or in a single file (*unchunked*). We must support two different forms of HTTP requests for these alternatives: HTTP range requests for unchunked titles and normal HTTP GET requests for chunked titles. We are also careful when we create the catalogs of titles used for experiments because the hard drives we use implement zoned bit recording, which means that data on the outer tracks can be accessed with higher throughput than those on inner tracks. We ensure that titles are placed at approximately the same disk positions regardless of how titles are stored.
- We assume that HTTP requests related to searching and browsing for titles occur on a separate machine. This is how large streaming video systems are designed in practice [34, 96] and it simplifies the benchmark.
- Because we obtained measured characteristics of the YouTube workload from existing studies, our benchmark reproduces the actions of older YouTube clients that did not employ adaptive streaming. Fewer than 5% of YouTube sessions involved a resolution switch and a majority of those switches occurred in the first 10 seconds [34], so we do not include resolution switches in our benchmark. We discuss the behaviour of adaptive clients in Chapter 5.

3.2 Workload Specification

Our goal in designing benchmark workloads is to accurately model the request traffic seen by real web servers streaming titles to clients. However, satisfying this goal is not sufficient to ensure benchmark results that are representative of web server performance in real deployments. In real deployments, the server hardware is provisioned to match the actual workload; we instead generate a workload based on the capabilities of the server hardware, such as hard drive capacity. We scale workload characteristics, such as the number of different titles requested, to match the capacity of our experimental server rather than the much larger capacity of real deployments.

There are also a number of pragmatic secondary goals that affect how we design our workloads. For example, although most workloads model current traffic patterns, workloads that are reconfigurable can be used to model anticipated demands and traffic parameters; this can be extremely useful in planning and forecasting future design requirements. Some parameters, such as title durations, are unlikely to be significantly different as a result of technology changes, but values for parameters such as bit rate and client downstream bandwidth may be very different in the future. Therefore, in designing a reconfigurable workload, we explicitly separate parameters we expect will change over time, which enables us to quickly experiment with different workload configurations.

In addition to reconfigurability, another workload design goal is ensuring a reasonable benchmark run time. The goal is to provide rapid feedback that is useful for both web server development and configuration tuning. This led us to design workloads with sessions that are as short as possible without sacrificing their ability to characterize the steady-state performance of the web servers. For example, we found that in our experimental setup, workloads with 7,200 sessions ensure that the web server performance reaches steady-state.

In the following sections, we describe in detail the specification of one example video streaming workload suitable for generating a benchmark.

3.2.1 Title Characteristics

Our title and session characteristics and distributions are drawn primarily from the work of Finamore, et al. [34]. This paper provides low-level details about client sessions by measuring traffic at the edge of the network, and is a source for information regarding YouTube title characteristics and download mechanisms used in 2011.

There is much debate in the literature about the shape of a YouTube title popularity distribution. Some find a close fit with a Zipf distribution [39]. Others find that Gamma or Weibull

distributions fit more closely [24]. For our workload, we use a Zipf distribution because previous work that measures over a short time frame, similar to our target benchmark environment, finds that measurements follow a Zipf distribution [98]. In contrast, measurements that sample over a longer period of time or rely on crawling the YouTube data API [24] to extract viewing information tend to have non-Zipf-like distributions. Cha, et al. discuss these issues [19], and show that when individual clients with Zipf-distributed title preferences are combined, the aggregate distribution curve is not necessarily Zipf-like.

The Zipf distribution requires two parameters; we chose an alpha value of 0.8 [1] and a title population of 10,000 for our benchmark workload. The number of titles was based partly on the capacity of the hard drive in our server, which can hold 10,000 titles with the average size of 13 MB. The catalog size was also chosen to suit our experiment length of 7,200 sessions. This choice of parameters results in about 35% of requests being serviced from the cache for our experiments.

Other choices for the number of titles are possible, and will affect the popularity distribution. Figure 3.2 shows the popularity distributions for both our choice of 10,000 files for the catalog and the distribution if there are 500 files in the catalog. The 10,000 title distribution is nearly a straight line, but the 500 video distribution has a steep cut-off for ranks close to 500. We feel the 10,000 title distribution is more representative of the popularity of YouTube titles selected over a short period of time, with about 67% of titles requested a single time compared to a measured average value of 77% [97].

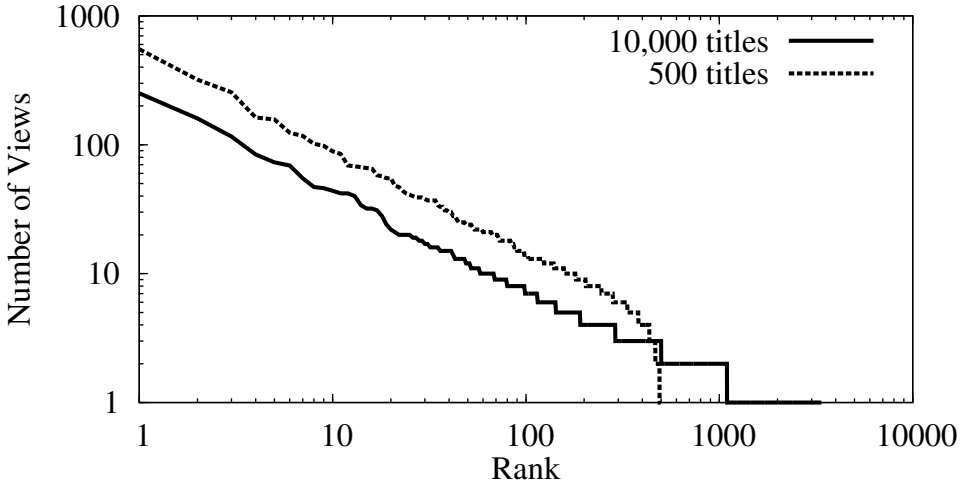


Figure 3.2: Popularity distributions for 7,200 sessions, with different catalog sizes

Another important parameter is the list of durations of the titles in the catalog. YouTube title durations have a complicated distribution; for example there is a peak at 200 seconds (the typical duration of a music video) and another at 10 minutes, a limit that YouTube had imposed on most title durations (this limit has since be increased to 15 minutes, and some users are exempted from duration limits [20]). Some authors specify the duration algorithmically as an aggregation of normal distributions [24]. We use a CDF to represent the distribution of title durations rather than an analytical formula because we believe the distribution is likely to be irregular for any catalog. Figure 3.3 shows the duration distribution used for our workload, from measurements made by Finamore, et al. [34].

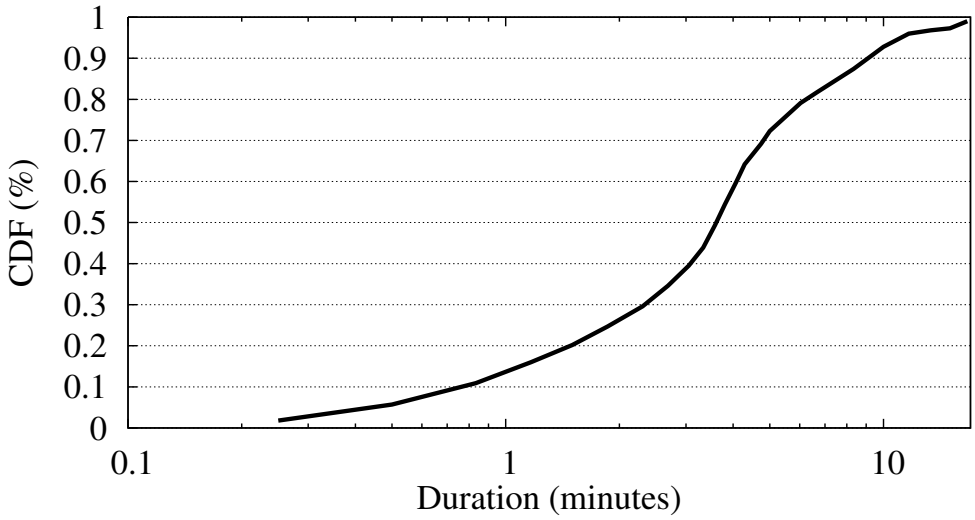


Figure 3.3: Duration of titles

From this distribution, we assign a duration to each title without accounting for the title popularity. One study of YouTube [1] has suggested a very weak correlation between popularity and title duration, with a correlation coefficient of less than 0.1, so we choose duration and title popularity independently. An additional concern with assigning durations is that the most popular titles make up a large proportion of the workload, so the durations assigned to these titles may have a large effect on the proportion of sessions that might be serviced from the cache. We assign the median title duration to the two most popular titles because if the most popular titles are significantly shorter or longer than the median, it will have a big impact on the overall workload.

The final title characteristic we assign is the bit rate. There are many different video encodings and variable bit rates used for YouTube titles, as observed by Finamore, et al. [34], with an

average rate of 394 Kbps [39]. However, a single rate is used for more than 90% of sessions [34], so we use a single fixed bitrate of 419 Kbps, chosen because it represents 10 seconds of content using 0.5 MB of data.

3.2.2 Session Characteristics

Our workload specification must, in addition to providing the characteristics of the titles, also specify how much of each title is downloaded by the clients. From previous studies, we know that most clients do not watch to the end of the titles [34, 96, 22]. Session duration depends on the duration of the title chosen, so Figure 3.4 shows the curve we use to determine what fraction of a title is downloaded during a session, based on measurements by Finamore, et al. [34].

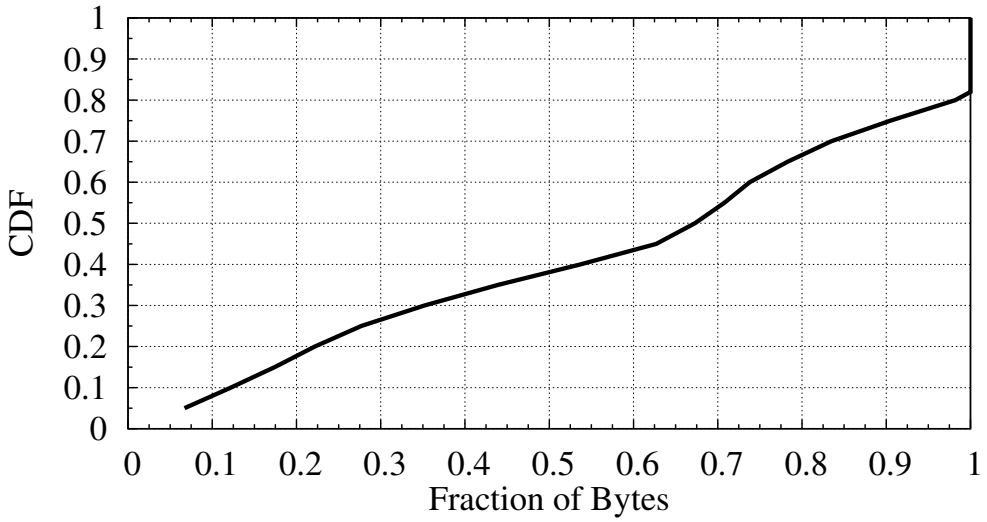


Figure 3.4: Fraction of bytes downloaded during session

We choose a session fraction independently of title properties such as duration or popularity. For some workloads, for example the video on demand system deployed by China Telecom [96], sessions last longer for less popular titles than more popular titles. For other workloads, like Tencent Video [21], the opposite occurs and sessions are shorter for less popular titles. In the absence of specific measurements for YouTube traffic, we believe that choosing the session fraction independent of popularity is a reasonable simplification.

The final needed session characteristic is the session initiation rate. Rather than assign a particular value, we instead vary the average session initiation rate in order to determine web server

performance limits. We could find no measurements of the session initiation rate for YouTube traffic. However, Abhari and Soraya use a Poisson distribution for their YouTube workload generator [1] (without providing a justification), and measured session initiation rates for China Telecom video requests [96] resemble a modified Poisson distribution (modified to reflect lower numbers of arrivals than expected from a true Poisson process). Therefore, we use a Poisson process to initiate sessions.

3.2.3 Client Network Characteristics

Table 3.1 shows the access speeds we use for our clients in this workload. This data represents the speeds of client computers in the United States used to access Akamai servers, and were measured by Akamai in 2011 [5]. Akamai does not provide detailed information about the distribution of access speeds, but instead divides access speeds into ranges and reports the percentage of access speeds in each range, as shown in the *Akamai Share* column. As a result, we represent each category with a single average rate, shown in the *Rates Mbps* column. For our streaming video workload, we disregard the low speed clients reported by Akamai because their access rates are too slow for viewing video and because they represent an insignificant fraction of the total. We chose the specific values in the *Share* column based on the use of 12 hosts to simulate clients.

Category	Akamai Rates	Akamai Share	Rates Mbps	Share
High	Above 5 Mbps	42.0%	10.0	42%
Bband	2 – 5 Mbps	38.0%	3.5	42%
Medium	0.5 – 2 Mbps	18.2%	1.0	17%
Low	Below 0.5 Mbps	1.8%	–	0%

Table 3.1: Client access speeds

We also model network delays between the clients and the server. We do not have information regarding the network delay for typical YouTube clients, so we assign a constant delay of 50 ms to both the forward and reverse paths, which is the approximate time to transmit data from coast-to-coast in North America. We expect that delays in the real world are likely to be different than 50 ms, depending on the geographical location of servers relative to clients and the type of access network used. If a study is produced with more detailed measurements, we can accommodate different delay values within our framework as is the case with all other characteristics.

3.3 YouTube-like Benchmark

Table 3.2 provides a summary of the parameters we use to construct a benchmark that represents YouTube traffic. We give a description of each parameter, its value or distribution, and the source of the measurement. This table is an abstract specification. Once we target an experimental environment, then we can produce a concrete implementation of the abstract session specification that is specific to the test equipment.

Parameter Description	Value used	Source
Video Popularity Distribution	Zipf $\alpha = 0.8$	[1]
Video Count	10,000	
Video Duration Distribution	See Figure 3.3	[34]
Video Bit rate	419 Kbps	[39]
Session Length Distribution	See Figure 3.4	[34]
Session Arrival Process	Poisson	[48, 47]
Session Count	7,200	
Session Chunk Timeout	10 seconds	[12]
Client Network Bandwidth	See Table 3.1	[5]
Client Network Delay	50 ms, one way	
Client Request Size (MB)	0.5 and 2.0	
Client Request Pacing	Yes	
Client Adaptation	None	[34]
Server Storage Method	Chunked	
Server Chunk Size (Time)	10 s and 40 s	
Server Chunk Size (MB)	0.5 and 2.0	
Server Chunk Sequence	By Session	
Server Video Placement	Random	
Server Warming Size	3,500 chunks	
Server Ramp-Up	200 sessions	
Server Ramp-Down	100 sessions	

Table 3.2: Summary of workload specification

An important implementation detail for the benchmark is how to implement pacing so that the average download rate for content is approximately equal to the encoded bit rate of the content. For most YouTube clients, pacing is implemented by special algorithms on YouTube

servers [34, 39]. We prefer that our benchmark be usable without requiring YouTube-specific server modifications, so we implement pacing using a pull-based mechanism based on Apple's HTTP Live Streaming platform [12], that is also similar to the methods used by Netflix clients to download content (described in Section 5.1.3). Titles are divided into segments that represent 10 second intervals of combined audio and video information, and clients download this data in *chunks* consisting of one or more segments. Notionally, clients first download chunks at full speed, representing filling a playout buffer, then they request subsequent chunks at an average rate that is equal to the bit rate of the content, representing pacing.

There are two ways to implement client chunking; the clients can use HTTP range requests to download chunks from a single title file, or the titles can be divided into chunks and stored in separate files that are requested by the clients in their entirety. We utilize both methods in experiments, depending on whether content is stored in a single file or in multiple files. In Section 3.7.2 we evaluate three alternatives for storing 10 second segments of content: a) 0.5 MB files that contain a single segment, b) 2.0 MB files that contain 4 segments, and c) single files that contain all segments. In addition to experimenting with different storage options, we also evaluate the use of different client request sizes in Section 3.7.3, either 0.5 MB or 2.0 MB requests. For the experiments with different storage sizes and request sizes, we use six different workload specifications to create benchmarks.

3.3.1 Experimental Environment

The equipment and environment we use to conduct our experiments were selected to ensure that network and processor resources are not a limiting factor in the experiments. We use 12 client machines and one server. All client machines run Ubuntu 10.04.2 LTS with a Linux 2.6.32-30 kernel. All systems have had the number of open file descriptors permitted per user increased to 65,535. Eight clients have dual 2.4 GHz Xeon processors and the other four have dual 2.8 GHz Xeon processors. All clients have 1 GB of memory and four 1 Gbps NICs. The clients are connected to the server with multiple 1 Gbps switches each containing 24 ports.

The server machine is an HP DL380 G5 with two Intel E5400 2.8 GHz processors that each include 4 cores. The system contains 8 GB of RAM, three 146 GB 10,000 RPM 2.5 inch SAS disks and three Intel Pro/1000 network cards with four 1 Gbps ports each, for a total of up to 12 Gbps of bandwidth. The server runs FreeBSD 8.0-RELEASE. The data files used in all experiments are on a separate disk from the operating system. We intentionally avoid using Linux on the server because of serious performance bugs involving the cache algorithm, previously discovered when using sendfile [40]. Additionally, at least one popular streaming video service, Netflix, uses FreeBSD on their servers [69].

On the clients, we use a version of `httperf` [65] that was locally modified to support new features of `wsesslog` and to track statistics for every requested chunk. We use `dummynet` [80], which comes with Ubuntu, to emulate networks with different access speeds.

We use a number of different web servers. Most experiments use version 0.8.0 of the `userver`, which has been previously shown to perform well [16, 77] and is easy for us to modify. We also use `Apache` version 2.2.21 and version 1.0.9 of `nginx`. The default configuration parameters for `Apache` are not well suited to servicing video. It closes persistent client connections if a new request isn't received within 5 seconds of the previous request and also after 100 requests have been received. We modified these and other configuration parameters in `Apache` and similar parameters in the other servers to obtain the best performance, evaluating different potential parameters with a series of experiments, similar to the procedure used by Arlitt and Williamson [10].

3.4 Client Configuration

Our experiments utilize 12 client hosts to generate hundreds to thousands of concurrent streaming video sessions. Each client host is on its own gigabit subnet and we use `dummynet` to impose bandwidth limits and add delay to each session. Overhead from `dummynet` limits each client computer to a maximum of approximately 600 Mbps of throughput, or 7,200 Mbps aggregate bandwidth over all clients.

We approximate the specification in Table 3.1 by configuring `dummynet` to allow 10 Mbps of bandwidth on 5 of the clients, 3.5 Mbps on 5 of the clients, and 1 Mbps on the remaining two clients. Statistics are collected separately for each client, so this configuration makes it easy to generate statistics for individual rates. We use `dummynet` to delay both incoming and outgoing packets by 50 ms to simulate network latencies and tuned the client and server TCP parameters to handle the larger bandwidth-delay product introduced by the delay.

Our workload generator creates a trace file (called a `wsesslog`) for each client host that specifies a sequence of HTTP requests for entire files or ranges within files. An instance of `httperf` running on each host uses the `wsesslog` file to issue HTTP requests to the web server. Figure 3.5 shows a small example of a `wsesslog` that contains requests for several titles. The comments in the figure explain the requests.

Each title is requested in a sequence of chunks using a persistent HTTP connection called a session. Sessions are initiated using a Poisson process, so the duration between session initiations is independent with a common exponential distribution. New sessions are started independently, simulating the access pattern of many concurrent streaming video viewers. Normally, `httperf`

requests the next chunk in a session as soon as the previous chunk is completely received, but if a pacing delay is specified, a request will not be sent until the specified pacing time has elapsed from the start of the previous request. This is used to emulate video player buffering and/or users pausing a title.

```
# Session 1: 4 chunks stored in separate files, with pacing
#            after the first two requests
vid01/secs-0-9 timeout=10
vid01/secs-10-19 timeout=10
vid01/secs-20-29 timeout=10 pacing=10
vid01/secs-30-39 timeout=10 pacing=10

# Session 2: 3 chunks with range requests within the same file,
#            with no pacing
vid02 range=0-524287 timeout=10
vid02 range=524288-1048575 timeout=10
vid02 range=1048575-1572863 timeout=10

# Session 3: bit rate change - 2 chunks from separate files
vid03/high/secs-0-9 timeout=10
vid03/med/secs-10-19 timeout=10

# Session 4: pause/rewind/skip forward
vid04/secs-0-9 timeout=10
vid04/secs-10-19 timeout=10 pacing=60 # pause for 60 seconds
vid04/secs-20-29 timeout=10
vid04/secs-0-9 timeout=10             # skip back to start
vid04/secs-100-109 timeout=10         # skip forward
```

Figure 3.5: A small example of an `httperf` `wsesslog`

We also specify a timeout for each request in the `wsesslog` file, and if the request is not completely serviced before the timeout elapses, `httperf` terminates the session. This loosely approximates a user becoming unsatisfied with the response or video quality and ending the session. We use the failure count as a primary indication of whether the web server is overloaded. The throughput figures are also affected by timeouts because only completed requests

are included in our throughput measurements.

In our benchmark, the first 3 chunks of each session are requested without pacing delays, simulating the filling of a client device's playout buffer; and subsequent chunks are paced so they are requested at a rate of one chunk every 10 seconds. Each request is given a 10 second timeout, to represent the requirement that data must be received before it can be played back.

Table 3.3 contains summary statistics that characterize our workloads for the two different sizes of client requests. Both are constructed using the specification in Table 3.2 and differ only in the request size. We do not need to include separate statistics for the three alternative storage methods because those methods affect only the format of requests that are issued; the size and timing of requests is the same for all three storage methods. The first four values in Table 3.3 refer to statistics derived solely from the abstract specification, and are the same for both workloads. The remaining values differ because session durations are rounded up to a multiple of the chunk size.

Description	0.5 MB	2.0 MB
unique titles	3,366	3,366
single-session titles	67.5 %	67.5 %
average title duration	258.7 s	258.7 s
average title size	12.9 MB	12.9 MB
average session time	146.3 s	150.6 s
average requests per session	14.628	3.766
unique file chunks requested	60,004	16,318
total file chunks requested	105,323	27,188
number of chunks viewed once	44,478	12,293

Table 3.3: Characteristics of constructed workloads

3.5 Server Configuration

Any web server is capable of servicing the requests made by the `httperf` clients, which are simple static file requests. Files that represent the titles must be created *a priori* on the server's file system. We use two different methods for generating files to represent the titles, depending on whether the clients are configured to use range-requests or chunk requests. For chunk requests, we create many thousands of chunk-size files in the same directory, in numerical order,

starting from a newly-installed file system. Each title in the specification is assigned a consecutive sequence of chunks. Sessions are represented by sequential requests through as many of the chunks as necessary to equal the session duration. For range-requests, we create a single file to represent each title, with the size of a file equal to the duration of the title multiplied by the bit rate of the title. Sessions are represented by a sequence of contiguous range requests.

Since there are two methods for storing titles in files, one of our goals is to compare the alternative methods to determine the effect on performance. To permit a fair comparison, care is taken to ensure that the same number of bytes of data are being requested whether the title is stored in chunks or not. Additionally, all of the data associated with each title is as close to the same location as possible on disk, irrespective of the size and number of chunks used to store the title. This is required because the throughput of disk reads can be significantly impacted by the location of the files being read.

In the following sections, we describe a procedure for creating different file sets and for confirming that titles are stored at comparable disk locations. We developed and tested this procedure on servers using FreeBSD (versions 8.0 and 9.0) with the native UFS file system. It is likely that different file systems or operating systems will use different algorithms for file placement, so different algorithms and tools will have to be developed for other operating systems.

3.5.1 Determining File Placement

File system implementation details are hidden behind a layer of abstraction. Applications are able to create directories and files within a hierarchy of directories, but cannot control where files are physically placed on disk. The kernel is responsible for file placement, and it is difficult for applications to even determine where the files are placed, let alone control the placement.

We determine the physical location of each file on disk using `dtrace` and the Unix `wc` utility. `dtrace` is a framework that allows us to insert probes into the kernel to monitor when specific kernel functions are called, and to record the arguments to those functions. While `dtrace` is monitoring the kernel, we run a script that uses `wc` to read every byte in every file in the file set. We use `dtrace` to collect information about all calls to the internal kernel functions `open`, `close`, and `bufstrategy`. We capture the names of files from the `open` call, and track the `close` calls to determine which files are associated with `bufstrategy` calls. The arguments to `bufstrategy` provide the logical block addresses (LBA) where the files are stored on disk.

After collecting the LBAs accessed for each file, we post-process the `dtrace` logs to compute the average LBA for each file, including inodes. If the files represent full titles, the computed average LBA is used to represent the location of the title. If titles are represented by multiple

chunks, we average the locations of the chunks to represent the location of the title. Title locations calculated in this manner can then be used to determine whether a title is stored at the same disk locations regardless of the method used to store content in files.

3.5.2 File Set Generation

Our goal is to be able to directly and fairly compare the performance of titles stored with different granularities and to examine the impact that decision has on web server performance. As a result, we develop a methodology to control where files are placed on disk so that we can use the same locations on the same disk to store different file sets (i.e., chunked and unchunked).

We use three different file sets: one using a 0.5 MB chunk size, one using a 2.0 MB chunk size, and one that stores titles unchunked. Because title durations may not be exact multiples of each of the chunk sizes, we pad the file sizes (with data that is never requested) to ensure that a title occupies the same amount of space on disk, regardless of the chunk size. This helps to ensure that for each chunk size examined, the same title data can be placed in approximately the same location on disk.

We create a file set by starting with a freshly created file system, then writing all file chunks in a single directory. We have observed that when starting from an empty directory, files tend to be written to disk in sequential order starting from the location of the directory entry. We influence where files are placed on disk by controlling the order that files are written. When a title consists of multiple chunks, we create the chunks consecutively on disk, but we create titles in a randomized order so there is no particular relationship between the location of a title and the number of times it is viewed. Using the same creation order for the different file sets, all chunks for the same title will be stored contiguously on disk, and at very close to the same physical locations for each of the different file sets. Unfortunately, this simple procedure does not produce repeatable results because the FreeBSD file system does not place directories in the same location each time the file system is recreated. Since files are placed at the same position as the containing directory, it follows that files will be placed at unpredictable locations on disk.

We work around this problem by creating a large number of directories (in this case 500), while using `dtrace` to determine the location of each directory. We then create the file set in the directory with the lowest LBA, so that the file set will be created at low LBAs.

This procedure for creating file sets is expected to place files at the fastest locations on disk, with the chunks that comprise a title placed consecutively and with minimal file fragmentation. This layout permits significant performance optimizations that might not be possible with file sets that are heavily fragmented. We expect this layout could be achieved in most commercial

environments where title deletions are relatively uncommon, so it is a reasonable and consistent basis for comparing file sets.

3.5.3 File Set Locations

Figure 3.6 shows the average locations of each title when the file sets are created using our procedure. This figure shows that, for the most part, the files are placed in sequential order and titles created earlier have lower average locations. For each file set, only about 1.2% of the files are outliers, calculated as files that are out of sequence by more than 1% of the maximum LBA (i.e. $3x10^6$ LBA). Comparing the locations of files in different file sets (excluding the outliers) shows the average divergence in location is quite low, at approximately $7x10^5$ LBA (0.23% of the maximum LBA). Therefore, we believe titles created using our procedure are placed at comparable locations on disk.

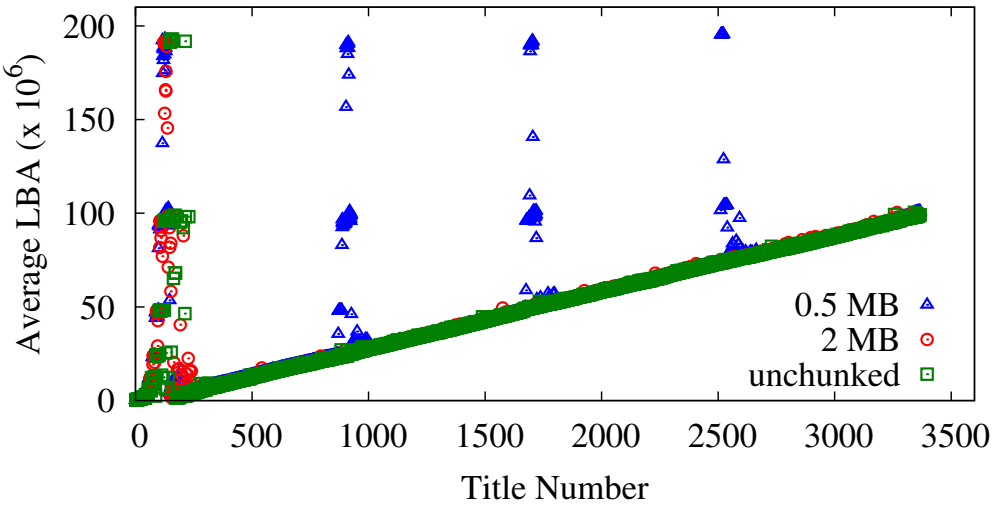


Figure 3.6: Video locations at low block numbers

3.5.4 Potential File System Performance

We use two different file sets, one created at low LBAs as shown in Figures 3.6 and an equivalent file set created at high LBAs, to conduct experiments to determine the potential throughput that

can be obtained while reading those title files. We use `wc` to read all the chunks used for all titles in the file set. The chunks making up a particular title are read in sequential order but the titles are chosen in a random order. We repeat the experiments 15 times for each file set while using `iostat` to measure average disk throughput. We calculate 95% confidence intervals using a t-distribution for the results for each file set, which are shown in Table 3.4.

File Set	Mean (MB/s)	95% c.i.
Low Unchunked	94.9	0.062
Low 2 MB Chunks	57.3	0.023
Low 0.5 MB Chunks	34.8	0.12
High Unchunked	77.7	0.057
High 2 MB Chunks	50.0	0.016
High 0.5 MB Chunks	31.9	0.085

Table 3.4: Average throughput using `wc`

Throughput is 10 to 25% higher when the file sets are placed at low LBAs on disk (i.e. on outer tracks) compared to high LBAs. These results demonstrate that placement has a significant effect on access speed and further illustrates the importance of placement when conducting a fair comparison between file chunk sizes. For consistency, we use the file sets with low LBAs for all other experiments in this thesis.

The results also show there is a significant difference caused by the choice of chunk size; the larger the chunk size, the higher the throughput. The following sections explore whether the throughput differences that occur when using `wc` also occur when a web server accesses the file set.

3.6 Running Web Server Experiments

For our web server experiments, we measure the aggregate throughput of the server when servicing workloads at different request rates. We use these measurements to produce graphs, such as Figure 3.8, that show the aggregate throughput in MB/s from requests that were completely serviced prior to the timeout expiry. The benchmark workload is generated in an open loop fashion, with the session initiation rate independent of responses from the server. However, when timeouts occur, it affects the workload because the remaining scheduled requests in a session are

skipped after a timeout occurs. When the server is not overloaded, we expect the throughput to be equal to the request rate multiplied by the size of chunks requested.

The methodology for running experiments and collecting measurements has a significant effect on the results. In this section we describe some of the methodological issues related to running experiments. We explain how we ensure that experiments reach steady-state in a reasonable amount of time. We then demonstrate the importance of two features of our methodology: using `dummynet` to throttle network connections, and the pacing of requests. Finally, we discuss the execution time and repeatability of our experiments.

3.6.1 Steady-state Behaviour

We include in our measurements only those sessions that are serviced completely while the web server is operating at a steady-state (i.e., when the rate of session initiations is equal to the rate of session completions). At the start of an experiment, the rate of session completions is very low because the paced sessions in our workload last an average of 146 seconds. Because of this, we don't measure sessions during a ramp-up period. Similarly, when we stop initiating new sessions at the end of the experiment, any sessions that are still active should not be included in the measurement because the server is no longer at steady state. We apply a ramp-down period at the end of an experiment to account for this, and we do not count sessions that are initiated too close to the end of the experiment.

An additional consideration at the start of an experiment is the state of the cache. For repeatable results, we must ensure that the cache is in the same state at the beginning of every experiment. It is most practical to start with an empty cache, but a web server will not reach full performance until the cache is full, which can take a considerable amount of time.

Figure 3.7 is an example of the curves we use to evaluate the progress of an experiment. The curves show the length of time it takes for each individual chunk to be serviced, in the order they are requested over the course of an experiment. The *no warming* curve shows the response times when no cache warming is performed, in which case the response times are longer and more variable before 4000 chunks have been requested than after. The *warm 3500 chunks* and *warm 14000 chunks* curves show the results when the cache is pre-warmed with the most popular chunks before the start of the experiment, which results in lower and more stable response times at the start of experiments. Given the results in Figure 3.7, we modified `httperf` to ignore requests that time out during a 3,200 request ramp-up period at the start of experiments and for a 1,600 request ramp-down period at the end of experiments.

We use a script to start the clients, the server, and the tools we use to monitor the progress of the benchmark. `vmstat` is used to monitor CPU utilization. `iostat` monitors disk usage

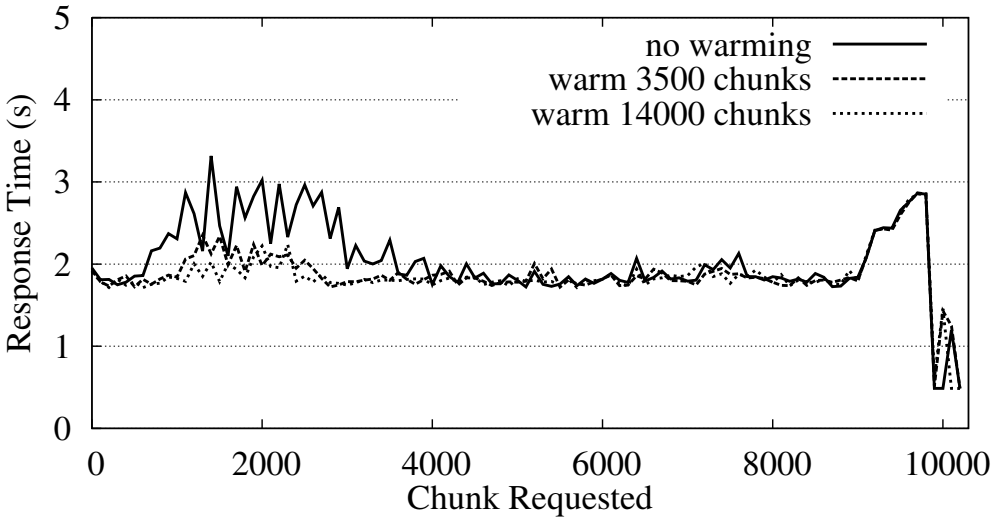


Figure 3.7: Cache warming techniques

which includes bandwidth, transaction size, transaction times, queue lengths, and the fraction of time the disk is busy. At the end of an experiment, our script combines the most important information from `httperf`, the server, and the monitoring tools into a single report. In particular, the total amount of data read from disk, sent by the server and received by the client are all recorded.

3.6.2 Bandwidth-Limited Clients

The use of `dummynet` enables us to simulate network characteristics such as available bandwidth and latency. We conduct experiments to demonstrate the importance of simulating representative client access networks. In one case, we configure the `userver` to use up to a maximum of 100 processes and 20,000 simultaneous connections. In the other case, we configure the `userver` to use only 1 process and 1 connection in order to serialize the servicing of requests. We run two experiments comparing these configurations using the workload with 0.5 MB chunks. The clients do not use pacing because with a single connection configuration, the server throughput will be bounded by the pacing rate. The first experiment does not use `dummynet` (*unthrottled*) and the second uses `dummynet` to model different client networks (*throttled*).

Figure 3.8 shows the results of these two experiments. When client bandwidth is unthrottled, there is only a small difference between using a single connection and using multiple connec-

tions. However, when client bandwidth is throttled, a single connection provides unacceptably low throughput. This demonstrates the importance of simulating realistic client network speeds, to avoid proposing methods that are acceptable in a lab environment but would fail in the real world.

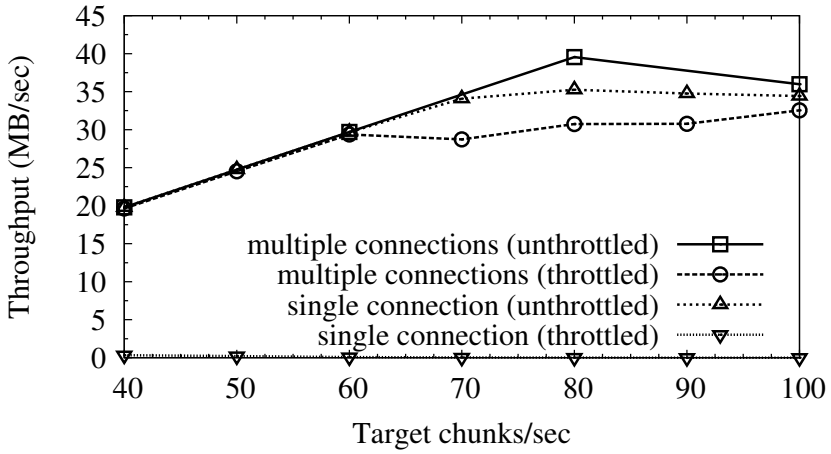


Figure 3.8: Using `dummynet` to model client networks

3.6.3 Effect of Pacing

The video players on some devices, especially those with limited memory capacity like smartphones and tablet devices, will limit the amount of title content stored on the device at any point in time. Playback begins by first buffering a reasonable amount of data to play the title without having to rebuffer (i.e., stop playback while waiting for title content to be delivered) and then requesting more content when buffer space becomes available. As described previously, this behaviour is mimicked in our workloads by using the pacing functionality we have added to `httperf`.

However, video players on some devices have significant amounts of memory and utilize a technique known as *progressive download* [12]. In this case, a client device downloads title content as quickly as it can be delivered, either by requesting the entire file that contains the title, or if the title is stored in chunks, by requesting the next chunk as soon as the previous chunk arrives. In this case, clients will request content far in advance of when it will be played back.

An interesting question is whether or not such behaviour by the clients (issuing paced versus non paced requests) affects the overall throughput of the server. To examine this issue we create

a new workload that has no pacing delays to compare to a workload with pacing. We can create workloads with specified mixes of clients issuing paced versus non paced requests, but we consider here the extreme case in which none of the clients pace their requests and are only limited by the speed of the server and their network connection.

Figure 3.9 shows the results of this experiment. The lines in the graph that are labeled *pacing* and *no pacing* show differences in throughput depending on whether pacing is part of the workload. These results would have been difficult to predict *a priori*, and demonstrate the necessity of including pacing in the benchmark.

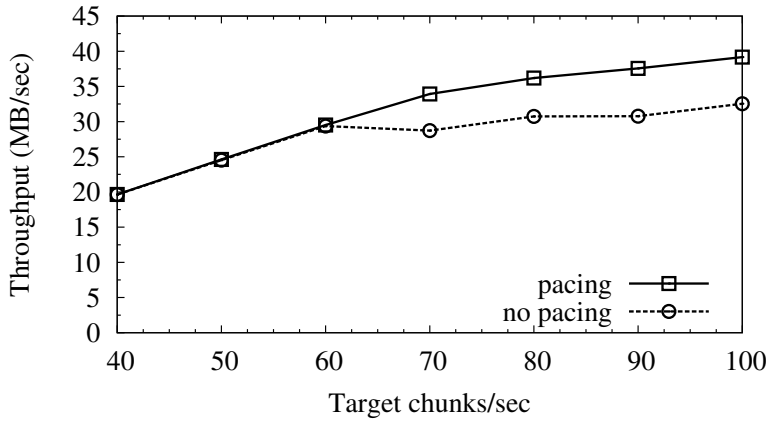


Figure 3.9: Effect of pacing on throughput

3.6.4 Duration and Repeatability

One of our stated goals is to produce a benchmark that completes in a reasonable amount of time. We believe that the execution times are sufficiently long to reach a steady state yet not so long to prohibit their use. For the graphs in this chapter, an experiment for one data point lasts 30 – 65 minutes (depending on the target request rate) with about 5 hours needed to generate one line on a graph (e.g., Figure 3.9).

The length of the experiments and the exclusion of ramp-up and ramp-down phases when gathering performance metrics helps in obtaining repeatable results. We tested the stability of the experiments by repeating experiments 10 times at selected rates. This also enabled us to compute 95% confidence intervals for the failure rate percentages. Table 3.5 contains measured statistics for a number of experiments that are described in this chapter. The confidence intervals

for throughput are small, particularly when there are no failures during the experiment, so experiments are repeatable whether or not the server is overloaded. We omit confidence intervals from our figures because the range is typically smaller than the symbols used to mark data points and it makes the graphs easier to read.

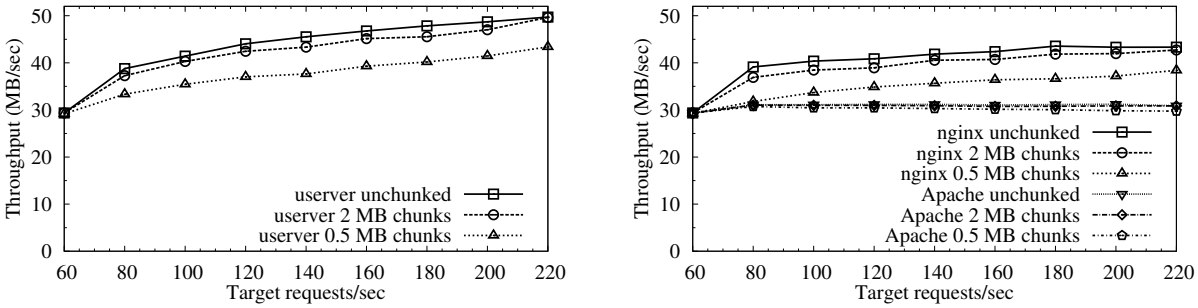
Request Rate	Tput Mean (MB/s)	Tput CI (MB/s)	Failure Mean (%)	Failure CI (%)
Figures 3.12a and 3.12b, ASAP `userver`				
20	39.2	0.01	0.0	0.00
40	68.4	0.09	5.7	0.13
Figures 3.12a and 3.12b, vanilla `userver`				
20	33.1	0.12	14.1	0.95
40	33.3	0.61	52.5	1.68
Figures 3.11a and 3.11b, vanilla `userver`				
70	33.8	0.06	2.3	0.17
100	38.9	0.15	19.6	0.66

Table 3.5: Throughput and confidence intervals for some runs of the `userver`

3.7 Baseline Server Performance

We use our HTTP streaming video benchmark to evaluate the performance of three different web servers: `nginx`, `Apache` and the `userver`. The purpose of the experiments in this section is to obtain baseline performance results for these web servers, and to compare different potential methods for storing titles. Specifically, we evaluate workloads where clients request 0.5 MB segments at a time, for 3 different on-disk chunk sizes (0.5 MB, 2 MB, and unchunked), comparing the throughput of these web server configurations at different target request rates. Figure 3.10a shows the results for the `userver` and Figure 3.10b shows the results for `nginx` and `Apache`. From these results, we see that the `userver` and `nginx` perform similarly, with `Apache` generally trailing in performance. The file chunk size has only a modest impact on throughput for all three web servers, with the largest performance increase occurring when changing from 0.5 MB to 2 MB chunks.

Given the results using `wc` in Table 3.4, we were surprised by the small difference in performance from increasing the chunk size for the web servers. These performance results led us to examine the contributions of the disk in isolation, as the throughput results in Figures 3.10a and 3.10b combine the throughput of both the cache and disk. We used `iostat` to measure



(a) Throughput of the `userver`

(b) `Apache` and `nginx` Throughput

Figure 3.10

the disk throughput alone using a workload with a target request rate of 80 requests/sec. The results were similar for both `nginx` and the `userver`: 26.5 MB/s, 25.7 MB/s and 22.6 MB/s for unchunked, 2 MB, and 0.5 MB file sets, respectively. These throughput values are far below the peak disk throughput in the top half of Table 3.4 of 94.9, 57.3 and 34.8 MB/s, which were determined by using `wc` to read the same file sets. These results suggest that neither the `userver` nor `nginx` are efficiently reading from disk, and that the chunk size has a small impact on disk read performance for these servers.

3.7.1 Implementing Asynchronous Serialized Aggressive Prefetching

We had originally expected that the operating system could, without additional hints or modifications, significantly leverage the larger chunks to improve disk performance. However, the similar levels of throughput observed in Figures 3.10a and 3.10b when using different chunk sizes might indicate that there are inefficiencies with the implementation of prefetching and disk scheduling in the kernel. We hypothesized that an application-level disk scheduler and prefetcher might help the web servers take advantage of larger chunk sizes and achieve higher throughput.

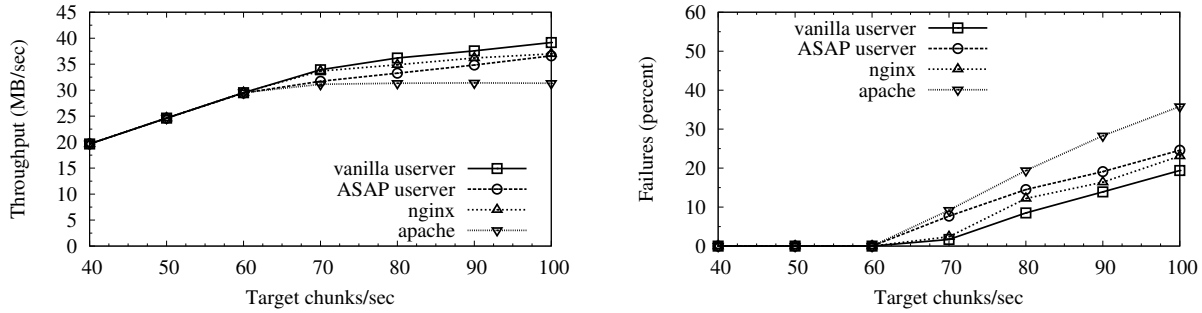
Therefore, we implement a version of the `userver` with two modifications. First we modify the `userver` to perform aggressive prefetching: when it is necessary to read from the hard drive, the `userver` always requests a large amount of data (e.g., 2 MB) rather than the smaller amount (e.g., 0.5 MB) requested by the client. A second modification to serialize access to the hard drive was also necessary because FreeBSD interleaves prefetch operations from concurrent threads, which has the effect of dividing the large prefetches into numerous small disk accesses, negating the benefits of aggressive prefetching.

We modify the architecture of the `userver` to enable application-controlled prefetching by introducing a single helper thread which is solely responsible for accessing the hard drive. For servers with multiple hard drives, a helper thread is used for each drive. We use the `SF_NODISKIO` option in the FreeBSD implementation of `sendfile` that causes the system call to return an `EBUSY` error code rather than blocking to read from disk [81]. When this occurs, we send a message to a helper thread which reads a portion of the file and signals the main thread of the `userver` after the data is read. The helper thread uses a FIFO queue and services requests sequentially. It prefetches a configurable amount of data prior to servicing each request. Although we bypass the kernel disk scheduling algorithm completely by issuing read requests singly in FIFO order, the performance gain from avoiding interleaved disk accesses is much greater than the small potential gain from allowing the kernel to schedule read requests. However, if it is necessary to maximize hard drive throughput, it is possible to implement a more sophisticated scheduling algorithm for the request queues in the helper threads. In the experiments and graphs that follow, we refer to the version of the `userver` that is modified to implement *Asynchronous Serialized Aggressive Prefetching* as the *ASAP* `userver` and the unmodified version as the *vanilla* `userver`.

We also made additional modifications to the `userver` that allow us to specify all of the files that comprise each title. This information can be used to prefetch multiple consecutive chunks of the same title when the desired prefetch amount is larger than a single chunk. This is done for comparison purposes rather than as something we would expect a server to implement. It permits us to study the throughput of the server when files are stored in different sized chunks, while prefetching the same amount of data.

3.7.2 Effect of Chunk Size

We now repeat the baseline server experiments using the *ASAP* version of the `userver` and compare the results to the vanilla versions of the three web servers. Figure 3.11a shows the throughput of all four different servers using the 0.5 MB chunk file set. For these experiments, we vary the target request rate between 40 and 100 chunks/s (20 to 50 MB/s). When the request rate exceeds the capacity of the server, some requests will not be received within the timeout interval, and those sessions will be terminated before they can be completely serviced. Figure 3.11b shows the percentage of sessions that could not be completely serviced for each target load. These results show that all four server configurations provide similar performance. The failure rates at 70 chunks/sec are lower for `nginx` and the vanilla `userver` than `Apache` and the ASAP `userver`. The difference is larger than the 95% confidence intervals, so these server configurations are somewhat better at servicing the workload when using the 0.5 MB file set.



(a) Aggregate throughput with 0.5 MB chunk size

(b) Missed deadlines, 0.5 MB chunks

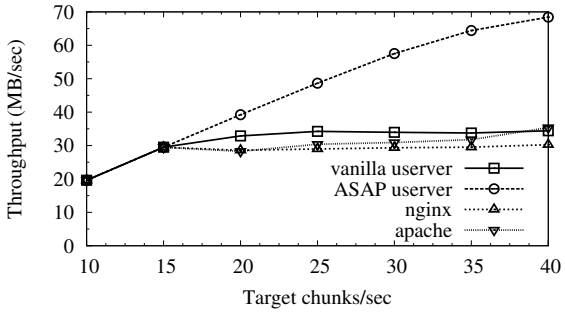
Figure 3.11

Table 3.6 shows the results from monitoring the disk performance during the experiments at a request rate of 70 chunks/sec (35 MB/s). The average times for read transactions are lower for `nginx` and the vanilla `userver`, which may explain why the performance of those servers is slightly better. The ASAP `userver` performs worse than two of the other servers with the 0.5 MB chunk workload because the serialization of disk accesses by the web server prevents disk I/O scheduling in the kernel. We are also interested in establishing an approximate upper limit on disk throughput for this file set. Our examination involves running a simple experiment using `wc` to sequentially read the exact same file chunks that are requested as part of the benchmark. This experiment is similar to the one described in Section 3.5.4, except that rather than accessing all files on disk a single time, we instead access the same subset of files as the benchmark, including multiple accesses of the same file. The results of this experiment are labeled *wc* in Table 3.6, and the average throughput using `wc` is 33% higher than the throughput of any web server.

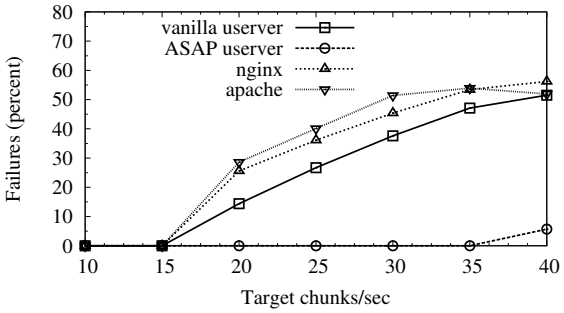
Web server	Avg. Time Per Read (ms)	Avg. Read Size (KB)	Avg. Tput MB/s	Avg. Disk Utilization
Vanilla userver	2.9	72.9	24.3	100%
ASAP userver	3.7	84.7	21.7	97%
Apache	3.1	71.3	22.1	99%
Nginx	2.8	71.1	24.7	100%
wc	2.3	84.0	37.2	91%

Table 3.6: Disk performance, 0.5 MB chunks at 70 req/s

Interestingly, the serialization of disk accesses is beneficial for the 2.0 MB file set. Figure 3.12a shows the aggregate throughput and Figure 3.12b shows the session failure rate when the chunk size is 2.0 MB. With the 2.0 MB chunk size, the ASAP `userver` reaches a failure-free throughput of 35 chunks/sec, 2.33 times the failure-free throughput of the other servers. Table 3.7 shows that there is much higher disk throughput with prefetching, both because of low average read times, and because the average read size is large. The disk throughput of the ASAP `userver` is only 13% lower than the performance of `wc` with 2.0 MB chunks The 2.0 MB workload uses the same abstract workload specification as the 0.5 MB workload, but we cannot compare the throughput measurements because the file sets are different and performance will vary because the session durations are slightly different in the two workloads.



(a) Aggregate throughput with 2.0 MB chunk size



(b) Missed deadlines, 2.0 MB chunks

Figure 3.12

Web server	Avg. Time Per Read (ms)	Avg. Read Size (KB)	Avg. Tput MB/s	Avg. Disk Utilization
Vanilla userver	3.6	88.1	23.9	100%
ASAP userver	2.3	112.4	42.7	90%
Apache	2.2	39.3	17.2	100%
Nginx	4.3	87.4	19.9	100%
wc	2.0	112.0	48.8	90%

Table 3.7: Disk performance, 2.0 MB chunks at 35 req/s

Finally, in Figure 3.13 we show the throughput that can be achieved with the ASAP `userver` when titles are stored unchunked, compared to using 0.5 and 2 MB chunks. For the experiments with the unchunked and 2 MB chunk file sets, the `userver` was configured with a prefetch size of 2 MB. In the case of the 0.5 MB chunk size experiment, the files are too small for 2 MB prefetches, so instead we prefetch four consecutive chunks from the same title, so all experiments read 2 MB when accessing the hard drive.

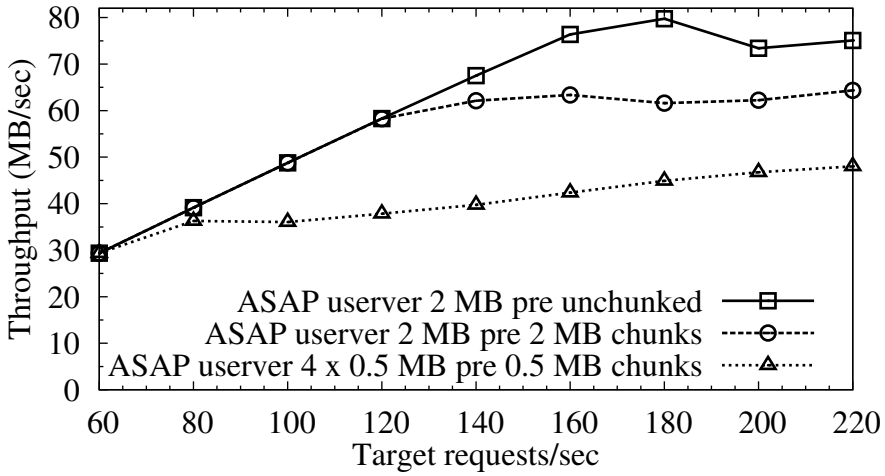


Figure 3.13: Throughput using a 2 MB prefetch size

These results show that, when using sequentialized reads and aggressive prefetching, the size of chunks used to store titles has a large effect on server throughput. Server throughput is lowest with 0.5 MB chunks, it is improved by approximately 20 MB/s with 2 MB chunks, and is improved by an additional 20 MB/s with unchunked files. This shows that prefetching data from chunks performs significantly worse than prefetching from an unchunked title, even when prefetching the same amount.

3.7.3 Effect of Request Size

In the previous section, we showed that server throughput can be greatly increased by using aggressive 2 MB prefetches to read data from the hard drive. This increase is accomplished without modifying the client log files, which contain requests for 0.5 MB of data. We performed experiments to determine if we could achieve equivalent performance as the ASAP `userver` by changing the client log files to contain bigger requests, accomplished by simply generating a

new workload with a different request size. Figure 3.14 shows the results of experiments using a workload with 2 MB requests, which represent 40 seconds of video content. We chose a size of 2 MB to equal the prefetch size we have been using, not because we know of any HTTP streaming video implementation that uses this request size; most implementations use shorter request sizes that optimize network performance [53]. We cannot compare the results of these experiments directly to the results in Figure 3.13 because the different request size changes the workload. For example, there is an average of 15.4 requests per session using the 0.5 MB request size compared to an average of 4.3 requests per session with 2 MB requests, so the target requests per second are significantly different for the two workloads. Comparing only the experiments in Figure 3.14 that use a 2 MB request size, it is clear that increasing the request size does not have the same effect as prefetching.

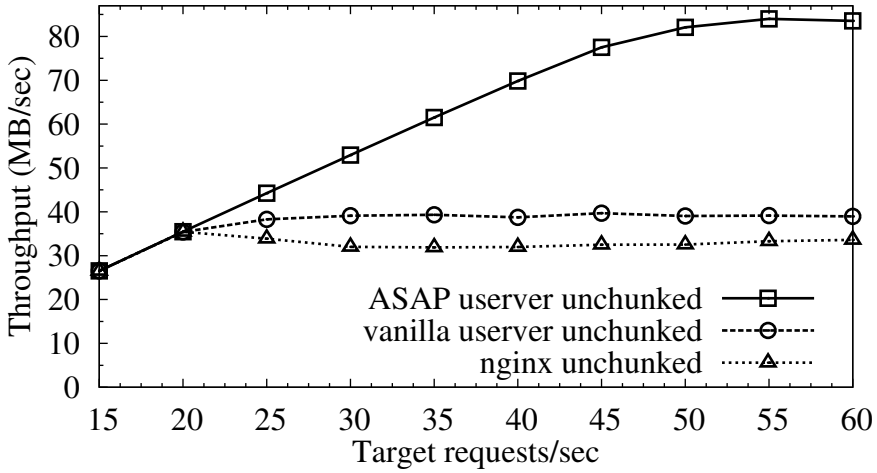


Figure 3.14: Throughput servicing 2 MB requests

3.8 Chapter Summary

In this chapter, we present our methodology for generating an HTTP streaming video benchmark in a lab environment. We create a workload specification using the published information available about YouTube at the time. Our methodology allows us to update the specification to represent other HTTP streaming video workloads, such as the Netflix information provided in Chapter 5. We describe how we use the specification to control a modified version of `httperf` and `dummynet` to generate a benchmark consisting of hundreds or thousands of concurrent

sessions that can be used to test server implementations. We demonstrate the importance of a number of features of the methodology: measuring performance only during steady-state operation, limiting network bandwidth to represent WAN networks used by client devices, and pacing client requests. We also show that experiments are very repeatable.

An important part of servicing the benchmark workload is populating the file system of the server with the files that will be requested. Because there are many possible ways to store title content in files, we develop a careful procedure that ensures we can accurately compare alternative approaches. We find that storing the entire title in a single file enables maximum throughput, but a web server must also employ serialized aggressive prefetching to take advantage of titles stored this way. We investigate three different web servers and determine that they cannot take advantage of unchunked titles in our experimental environment, while a modified version of the `userver` can. Figure 3.15 shows that throughput is nearly doubled when the `userver` is modified to prefetch 2 MB at a time from an unchunked file set.

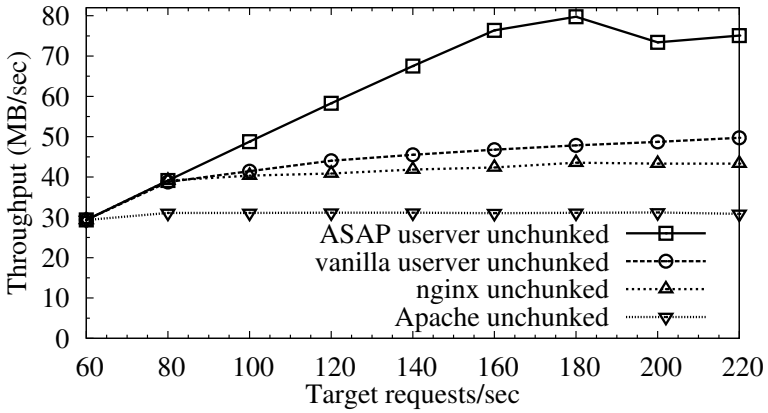


Figure 3.15: Throughput using an unchunked file set

We have shown the benefit of using 2 MB prefetches to service HTTP streaming video workloads, but we have not examined other prefetch sizes. In the next chapter, we show that 2 MB is not necessarily the prefetch size that maximizes throughput, and we design, implement and experimentally evaluate an algorithm that will automatically and dynamically find a good prefetch size.

Chapter 4

Selecting a Prefetch Size

In the previous chapter, we created a benchmark and used it to show that the throughput of the `userver` could be greatly increased by the use of aggressive prefetching. We started by choosing a rather arbitrary 2 MB prefetch size and demonstrated that we could more than double throughput. One goal of this chapter is to better understand how to choose a prefetch size and to determine whether it is possible to increase throughput by more than a factor of two. To investigate this question, we varied a number of different workload and system characteristics and performed experiments to determine the best prefetch size to use depending on these factors: 1) the bit rates of content, 2) the amount of system memory available, 3) the popularity distribution of titles, and 4) the performance characteristics of the hard drive. We found that the best prefetch size varied significantly, between 2 and 12 MB, depending on how we changed the workload and server hardware. In the best case, we find that throughput can be increased by a factor of 3.6 when servicing the YouTube-like benchmark using an ASAP version of the `userver`, compared to a vanilla version. If the bit rate of the content is higher, throughput can be increased by a factor of 5.2.

It requires exhaustive testing to experimentally determine the best prefetch size, and it is unreasonable to expect system administrators to determine the best prefetch size using time-consuming experiments; particularly since the procedure would have to be repeated when the workload or system hardware changes. Therefore, we develop an algorithm that is able to dynamically adjust the prefetch size to choose a good prefetch size automatically. We demonstrate that the results of the automated algorithm are similar to the best results found using a manual procedure, under all the different conditions we tested.

Finally, we consider the problem of how to choose a prefetch size when the workload contains content with mixed bit rates. For our YouTube-like benchmark, we use a single bit rate (based on

the published information about YouTube traffic in 2011), but for more recent workloads, such as the Netflix workload characterized in Chapter 5, content is requested at many different bit rates. We use a novel mathematical analysis to determine that the prefetch sizes should be proportional to the square root of the bit rate of the content. We conduct experiments with a mixed bit rate workload to show that throughput is higher when prefetch sizes are scaled based on the square root of the bit rates than 4 other alternatives we tried.

In the next section, we start our discussion by demonstrating the need to use different prefetch sizes to service different workloads.

4.1 Motivation

In this section, we show the results of experiments that demonstrate the need to use different prefetch sizes for workloads with different characteristics. We use two different benchmarks for these experiments: the benchmark we developed in the previous chapter, and one where the content has been changed to represent a higher bit rate. We have changed the way we report experimental results in this chapter because of the large number of experiments that we conduct to compare potential prefetch sizes. Rather than using a curve to report the throughput achieved for a series of experiments using different request rates (e.g. Figures 3.12a and 3.12b), we instead report the maximum throughput that could be achieved without timeouts. We discuss the changes to the experiment procedure in detail in Section 4.4.

Figure 4.1 shows the results obtained while servicing the two different benchmarks. The *SD* (standard definition) results are for experiments using the 419 Kbps files of the benchmark used previously, and the *HD* (high definition) results are for the new benchmark with files that represent a 2,095 Kbps bit rate (5 times the SD bit rate). The individual bars represent the throughput obtained using the unmodified *vanilla* `userver` (labeled "V") and using the ASAP `userver` with different fixed prefetch sizes (labeled 2, 4, 6, 8, 10 and 12 to denote the prefetch size used in MB). The throughput obtained from the disk (Disk Tput) and the actual throughput observed by all clients (Actual Tput) are also shown.

These results reveal that there is a significant performance difference between the vanilla `userver` and the ASAP `userver` using the best prefetch size, with a factor of 2.5 improvement with SD files (prefetch size 4 MB), and a factor of 3.4 improvement with HD files (prefetch size 8 MB). They also show that the 2 MB prefetch size used in the previous chapter may be significantly undersized for titles with HD bit rates. Additionally, these results demonstrate that although larger prefetch sizes can be used to increase disk throughput, at some point the benefits become minimal or non-existent. Furthermore, the increase in disk throughput does not necessarily translate into improved client throughput (Actual Tput). When prefetch sizes grow too

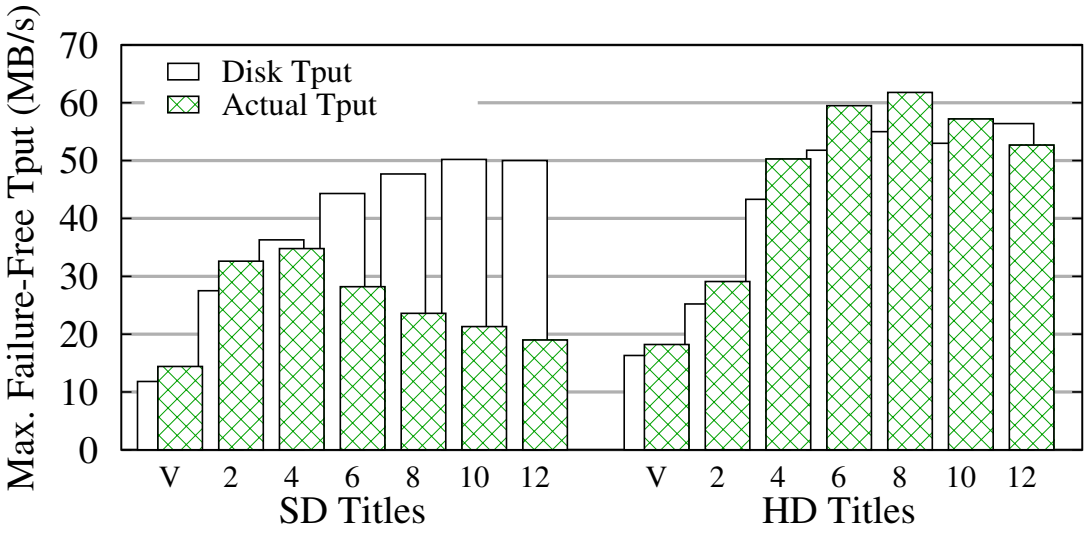


Figure 4.1: Throughput versus prefetch size

large, we observe reduced server throughput. However, what is too large for one workload and system configuration (e.g., 8 MB for SD files in this figure), may be the best prefetch size for another (8 MB for HD), motivating the need for an automated tuning algorithm.

Considering the possible consequences of prefetch sizes that are too big allows one to obtain insight into the types of information that would be useful for an automated prefetching algorithm. *Cache evictions* occur when prefetched data evicts data from memory that was retrieved for a previous client request, before it can be sent in response to a new client request for the same title. This results in a re-read of the data from disk that would have otherwise been unnecessary. Prefetched data can also be evicted from memory before it is requested by a client. This phenomena, which we refer to as a *prefetch eviction*, results in the same data being prefetched more than once (our current implementation only initiates prefetches for on demand requests that cannot be serviced from the file system cache). Data that is prefetched but never requested can be considered a *wasted prefetch*. This occurs for sessions that do not reach the end of the title (the majority of sessions), and the system prefetches beyond the data requested.

As seen in Figure 4.1, for SD files prefetch sizes larger than 4 MB reduce total server throughput. To provide some insights into the impact of different prefetch sizes, Table 4.1 presents some rough measurements, gathered during the execution of the SD experiments, using columns with the following meanings: *Size*: prefetch size; *Disk/Requested*: ratio of the total bytes read from disk versus the total bytes requested (values less than one indicate cache hits and values greater than one indicate that re-reads were required); *Cached/Requested*: ratio of bytes read from the file

system cache versus bytes requested[1]; *Evicted/Requested*: ratio of bytes evicted versus bytes requested; *Wasted/Requested*: ratio of bytes wasted versus bytes requested. The amount of wasted prefetch bytes may be slightly inaccurate when the same file is requested during more than one session, as it is difficult in this case to determine if it was truly a wasted prefetch.

Size (MB)	Disk / Requested	Cached / Requested	Evicted / Requested	Wasted / Requested
V	0.85	0.15	0.00	0.00
2	0.90	0.14	0.00	0.03
4	1.10	0.08	0.10	0.08
6	1.62	0.02	0.52	0.12
8	2.04	-0.02	0.86	0.16
10	2.36	-0.06	1.12	0.18
12	2.63	-0.09	1.34	0.20

Table 4.1: Extra data read due to prefetching (SD titles)

As seen in Table 4.1, once prefetch sizes are too large (in this case greater than 4 MB) the system does a lot of "extra work" to service requests. Significantly more data must be read than is being requested because it must either be re-read due to file cache or prefetch evictions or because the system has already prefetched data beyond the end of a session. Although cache evictions, prefetch evictions, and wasted prefetches serve as the limiting factors to the performance of increasingly aggressive prefetching, in this experiment the prefetch evictions (Evicted/Requested) cause the most harm. For example, with a prefetch size of 6 MB the total number of bytes that have been prefetched and evicted (and therefore need to be re-read from disk) is greater than half of the total number of bytes requested. As a result, our automated algorithm tries to avoid excessive prefetch evictions and the results shown in our evaluations in Section 4.5 include information about prefetch eviction rates.

We have shown that there is a delicate balance between prefetch sizes that are large enough to support high disk throughput, and sizes that are too large resulting in adverse consequences such as eviction of useful data from memory. With this balance in mind, we describe the design of our automated algorithm.

[1] Unfortunately file cache hit information is not available from the OS so we calculate an approximation using the number of requested bytes minus the bytes read from disk and the number of evicted and wasted bytes. Some values are negative because of the inaccuracies in wasted prefetches.

4.2 Automatic Prefetch Sizing

Our automated algorithm relies on an underlying prefetcher that performs aggressive prefetching. The ASAP version of the `userver`, which performs prefetching at the application layer, is one example of a web server that can be controlled by the automated algorithm, but other web servers that use different mechanisms for aggressive prefetch could also be used with this algorithm. Prefetches for all files of the same bit rate (the handling of different bit rates is described in Section 4.3) are done using the same prefetch size p except at the end of a file, where only the remainder of the file is read. The role of the automated sizing algorithm is to monitor the state of the server and use this feedback to periodically adjust p to improve the throughput of the system.

In the remainder of this section, we describe the algorithm we use to adapt the prefetch size, discuss the need to adjust prefetch sizes slowly, and demonstrate the operation of the automated algorithm using two example experiments.

4.2.1 Algorithm for Adjusting Prefetch Size

Our automated algorithm uses a gradient-descent method. While the web server is running, we continually calculate a score S, which represents the amount of work done, and the algorithm periodically adjusts the prefetch size either larger or smaller depending on which adjustment is expected to reduce the score. Past work suggests using file cache misses to represent the effort required by the server [17, 37, 13]. However, our objective is to maximize throughput to the client, which involves both file cache misses and disk transfer times. Minimizing cache misses irrespective of disk transfer times can result in poor overall throughput. Therefore, we define our score as the product of both the time to read from disk and the file cache miss ratio.

Figure 4.2 illustrates the trade-off between cache misses and disk throughput for different choices of prefetch size, with a prefetch size of "0" representing the results of using the vanilla version of the web server and relying on the operating system (FreeBSD) mechanisms to obtain good throughput. The left y-axis is used to show the number of milliseconds per transaction (*mspt*), which measures the time to read 128 KB from disk, and the right y-axis is used to show the *file cache miss ratio*, which measures the ratio of data read from disk to the total data requested (note that ratios greater than 1 are possible when data must be read into memory multiple times due to evictions). From this figure we can see that for some prefetch sizes, accepting a small increase in the cache miss ratio can greatly reduce the disk transfer time, and potentially improve overall throughput to the clients.

Our automated algorithm starts with an initial prefetch size and continually measures and tries to minimize the score while the server is running. At regular intervals, the algorithm compares

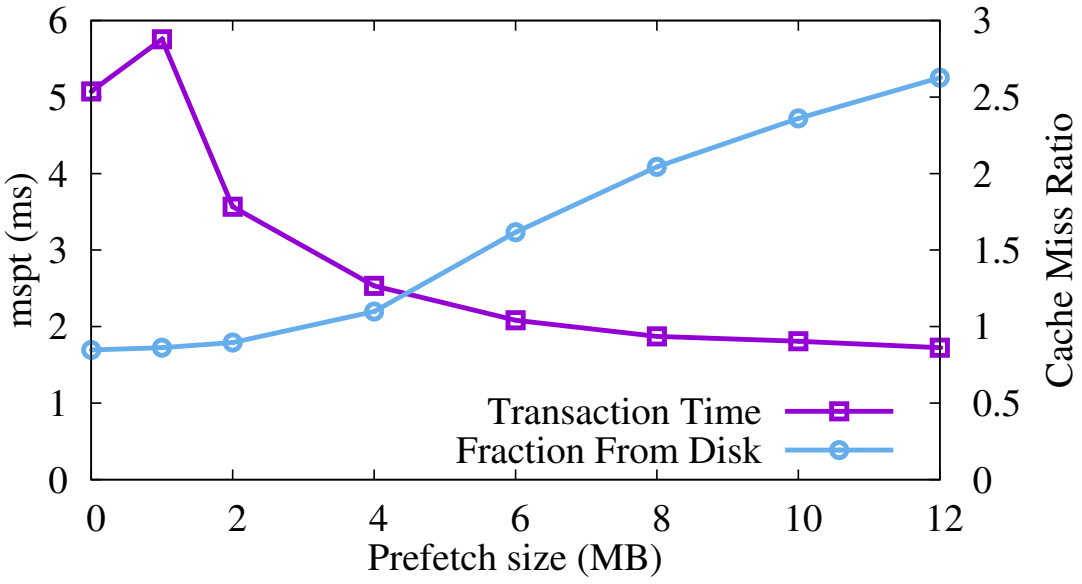


Figure 4.2: Transaction times versus prefetch size

the score at the current prefetch size (p) to the scores that were previously measured for the next larger (p_l) and smaller (p_s) prefetch sizes. If either p_l or p_s has a lower score than p, the prefetch size is adjusted towards that size. If the score has not yet been measured for p_l and the measured score for p_s is higher than the score for p, we try the unmeasured prefetch size, p_l (and vice versa if p_s has not been measured). If neither p_l nor p_s have been measured we try p_l. As will be discussed in Section 4.2.2, the prefetch size must be changed gradually. Therefore, we slowly change p until it is equal to the desired prefetch size.

One of the limitations of gradient descent methods is that they are prone to finding local minima if random fluctuations in the workload cause the score to be unusually large. To solve this issue, we deflate the past scores when the algorithm reaches a minimum. The gradual deflation of past scores will eventually cause the algorithm to retry a previously rejected prefetch size and recompute the gradient using the newly measured score. If the algorithm was at a local minimum, the new gradient will be lower, and the algorithm will continue to descend towards the global minimum.

While the gradient descent method is suitable under most conditions, it is not effective when the system is under either very light or heavy memory pressure. In the case that the server is under light memory pressure, increasing the prefetch size will improve the disk transfer time, but it may not cause a corresponding decrease in the cache miss ratio. The result is an oversized

p that will cause the server to perform very poorly when the load increases. Therefore, we turn off the automated algorithm when the load is deemed too light. Notice that there is no harm in turning off the automated algorithm because the server is under loaded and is therefore able to successfully service all of its clients.

The opposite extreme is when the server is under heavy memory pressure. In this scenario, the gradient descent method is incapable of changing the prefetch size quickly enough to avoid client timeouts. We handle this situation by quickly changing the prefetch size to a new value calculated using the *prefetch eviction time* (the average amount of time that prefetched data is resident in memory before it is evicted). By knowing the client request rate, we can calculate the time t before the prefetched data will be accessed. Therefore, if t is greater than the prefetch eviction time, then the prefetched data must be re-read from disk when the client makes its request. By reducing the prefetch size, we reduce memory pressure, which in turn increases the time that data remains in memory before being evicted, thereby avoiding re-reading the data from disk.

This algorithm is controlled with the following parameters (the actual values used while conducting our experimental evaluation are provided in Section 4.5):

start_size This is the initial prefetch size used when starting the server. In a production system, we would set this value to the size in use at the time the server was last stopped or shut down.

adjust_interval The score and other system performance information is collected over this interval, then used to adjust the prefetch size. This interval must be sufficiently long to average out short-term variations in demand, but short enough that the algorithm will converge on the best prefetch size before the server is overloaded.

step_size This is the amount by which the prefetch size is increased or decreased.

score_deflation_factor This is a deflation factor used to reduce the value of the scores stored by the algorithm when it reaches a local minimum.

busy_threshold This sets a minimum threshold for when to apply the adaptation algorithm in terms of how busy the prefetcher is. This threshold is used to avoid adjusting the prefetch size when the server is under a light load.

evict_threshold This threshold specifies the amount of prefetch evictions that are necessary before we use the prefetch eviction time to set the prefetch size.

4.2.2 Slowly Adjusting Prefetch Size

There is a serious practical limitation with respect to implementing an automated algorithm for prefetch sizing: changing the prefetch size can result in server overload if not handled properly. The problem arises when there are large numbers of titles with the same bit rate, as is the case for our workloads. To keep up with each client, the server prefetches data at the title bit rate. Therefore, the interval between prefetches is equal to the prefetch size divided by the bit rate. We use the same prefetch size for all clients, so data for all clients must be prefetched within the prefetch interval.

This can be a problem when we increase the prefetch size. For example, assume we are currently using a prefetch size that results in a 60 second interval and we increase the prefetch size and the result is an 80 second interval. After changing to the new size, every client will require a prefetch within 60 seconds (prefetches are issued on demand). However, a larger amount of data will be prefetched for each client. If the system was close to overload when prefetching the smaller amount, it will likely overload while issuing larger prefetches over the same interval. However, once the larger amount of data has been prefetched for each client; subsequent client requests will be spread over an 80 second interval and will not overload the server.

A similar problem occurs when the prefetch size is reduced. Suppose clients are prefetching in a 60 second interval and the prefetch size is decreased, resulting in a 40 second interval. For the first 40 seconds after the change, we will prefetch data using the smaller prefetch size, then between 40 and 60 seconds we will prefetch data for both the remaining clients that are prefetched in the 60 second interval as well as the clients who have converted to the new size. This period of doubling the number of prefetches can also cause the server to overload.

The solution to both of these problems is to change the prefetch size gradually, which corresponds to a gradual change in the interval between prefetches. Slowly adapting the prefetch size over time limits the additional demand on the disk for the first requests after the prefetch size changes (in particular when it increases). Furthermore, this gradually changes the prefetch interval, which ensures that prefetches using the new size do not concentrate in the same manner as they would if the prefetch interval changed rapidly.

By adapting slowly, the automated algorithm avoids server overload conditions that could otherwise occur. Despite the fact that it changes gradually, however, it remains effective at converging towards effective prefetch rates, as will be seen in Section 4.5.

4.2.3 Prefetch Algorithm in Action

We present the results of two different experiments to demonstrate the operation of the automated algorithm and to motivate some of its features. These experiments show that the algorithm can adapt regardless of whether the starting prefetch size is higher or lower than the best size. They also show the utility of the score deflation feature and of using the prefetch eviction time to set the prefetch size.

Figure 4.3 shows the operation of the algorithm during the execution of an experiment using HD files. The figure shows the throughput of data delivered to the client, the value of the score, and the prefetch sizes chosen by the algorithm over time. The throughput is plotted using the right y-axis, while both the value of the score and the prefetch size are plotted using the left y-axis (which, for clarity, has only a single axis labelling, for prefetch size). To show how the algorithm dynamically adapts, the request rate builds over the experiment, starting at 80% of the maximum load and increasing in 5% steps over 2,700 seconds. It remains at the maximum load for 1,200 seconds, until the ramp-down period at the end of the experiment when existing sessions end and no new sessions are initiated.

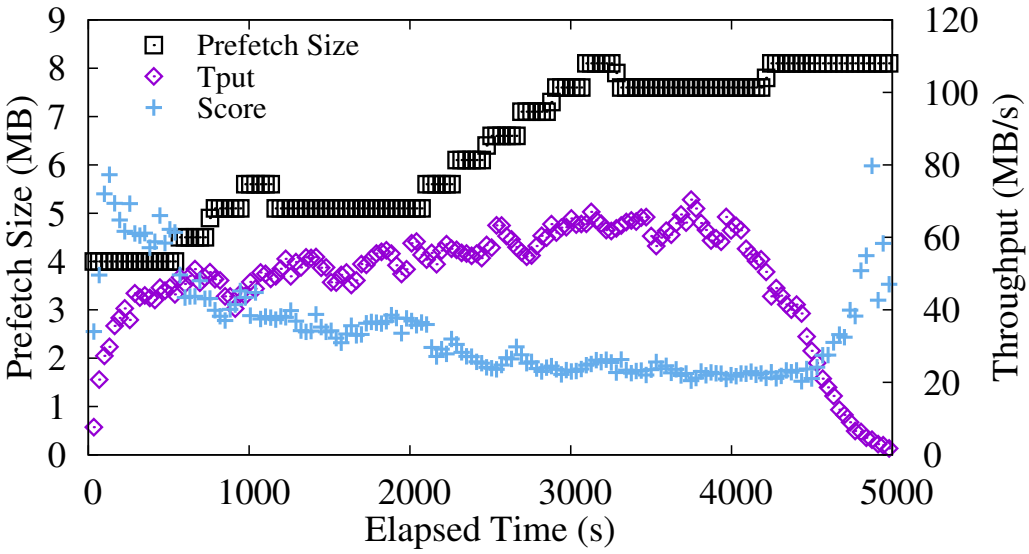


Figure 4.3: Dynamic prefetch size adjustments using HD files

The algorithm starts with a prefetch size of 4 MB, which is smaller than the best prefetch size for this workload. The prefetch size remains at 4 MB until the system experiences enough load, which begins to occur at around the 500 second mark. The algorithm increases the prefetch size

in response to lower scores until the algorithm reaches a local minimum at about 1,000 seconds. The stored scores are gradually deflated until the algorithm retries the higher prefetch size and measures a lower score. The algorithm continues to increase the prefetch size until it settles at 7.5 MB, where it stays until the experiment enters the ramp-down period. As the request rate drops at the end of the experiment, the algorithm responds by increasing the prefetch size to 8 MB, just before the load drops below the `busy_threshold` value and the algorithm no longer adjusts the prefetch size.

Figure 4.4 shows an example of an experiment where the prefetch eviction time is used to quickly adjust the prefetch size. This experiment uses SD files and starts with a prefetch size of 8 MB, a size that worked well with the HD files in the previous experiment, but is too large for this workload. The y-axis on the right is used to show workload throughput in MB/s as well as the average prefetch eviction times in seconds. Initially, the large prefetch sizes are effective because the system load is relatively light. After about 500 seconds have elapsed with the increasing workload, the large prefetches begin to cause prefetch evictions. The algorithm uses the average prefetch eviction time of about 60 seconds and the average bit rate of 0.05 MB/s to calculate a new prefetch size estimate of 3 MB. After the prefetch size reaches 3 MB, the prefetch size is close to the best value, and the gradient descent algorithm reacts to the rise and eventual fall of the request rate during the remainder of the experiment.

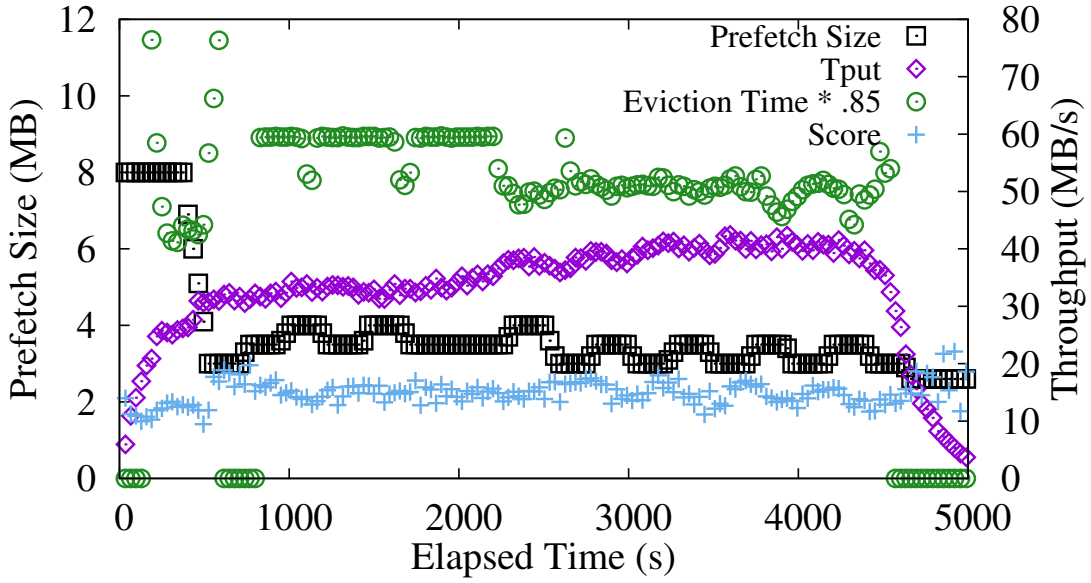


Figure 4.4: Dynamic prefetch size adjustments using SD files

These experiments demonstrate that the algorithm is able to converge to a good prefetch size even if it starts at a prefetch size significantly higher or lower than the well-performing alternative. They also demonstrate the ability of the algorithm to adapt well with different workloads.

4.3 Handling Multiple Bit Rates

For most experiments conducted to this point, we have considered environments in which only standard definition (SD) bit rate files are requested. This is because YouTube workload characterization studies found that although different bit rates were available, about 99% of requests were for SD titles [34] and because to our knowledge there do not exist workload characterization studies that can be used to construct representative benchmarks that include mixed bit rates (this problem is addressed in Chapter 5).

However, we have performed some experiments using HD bit rate titles (in section 4.1), and we find that the best prefetch size can be different for different bit rate titles. We would like to determine how we should choose a prefetch size, based on the bit rate of the title, for workloads where clients request content with a number of different bit rates. Our analysis in this section is not a substitute for the automated algorithm we have previously described. For the multi-bitrate experiment described in Section 4.5.4, we use the automated algorithm to select a prefetch size for one of the bit rates in the workload and scale the prefetch size used for the other bit rates based on this analysis.

The most relevant existing study is a paper by Gill and Bathen [37]. They prove that the optimal solution, which provides the highest aggregate throughput, is to use prefetch sizes that are proportional to the bit rates b_i of the individual streams i. For their proof, they assume that the workload consists of multiple steady sequential streams (that do not end) and that the cache used to store prefetched data is maintained using an LRU (Least-Recently Used) page replacement algorithm. First they prove that with an LRU-based cache, prefetched data is evicted from memory after an interval L that depends only on the size of the cache and the aggregate bit rate of the streams. Then they prove that because prefetched data will remain in memory for L seconds, and because prefetching more data always increases throughput (which is true for endless streams), it is optimal to choose a prefetch size that ensures the interval between prefetches is equal to L. This implies that the prefetch size should be equal to $L \cdot b_i$; in other words, the prefetch size is optimal when it is proportional to the bit rate of the stream.

There are two critical assumptions in Gill and Bathen's proof. The first is that streams are endless, which is not true of streaming video workloads where streams not only end, but they may end at unpredictable times. Second, their analysis assumes that an LRU cache is used to

store prefetched data, but this is not necessarily the best choice compared to many other page replacement algorithms that have been proposed [72, 73, 17, 49, 31, 46]. So we consider the problem of determining the best prefetch sizes if we assume streams are finite, and do not assume a particular page replacement algorithm is used.

Analyzing Finite Streams

First, we consider the implications of assuming streams will end at unpredictable times. We assume that there are n streams of client requests, with different request rates b_i. We will service each of those streams with a prefetch size p_i that may be different for each stream. Video streams are finite and may have unpredictable durations (see Section 3.2.2), so we define D as the expected duration of a stream. When a stream ends, there may be some amount of wasted prefetched data that has not been requested by clients because data was prefetched beyond the point at which requests are made. Our intuition is that doubling the prefetch size will double the waste, so we assume that waste is a linear function of the prefetch size: $w \cdot p_i$. This intuition is supported by Table 4.1, which shows that the amount of wasted prefetched data is approximately a linear function of the prefetch size.

To service the finite streams, we must prefetch all the requested data from disk, and there are also two additional overhead costs that we must minimize. 1) The cost of repositioning the disk head between prefetches (e.g., the seek time, rotational latency and other processing overhead), which we represent as C_p, in terms of milliseconds per prefetch. 2) The cost of prefetching data that is never requested by clients, which we represent as C_w in terms of the number of milliseconds used to prefetch each byte of data that is wasted. We use average values for C_p and C_w, even though the cost of repositioning the disk head will vary depending on timing and disk I/O scheduling, and the cost of reading a byte of wasted data varies depending on the placement of data on disk.

We now quantify the average cost per second of these two overheads. The number of prefetches per second for each individual stream is b_i/p_i, so the cost of those prefetches is $C_p \cdot b_i/p_i$. The cost of wasted prefetches when each stream ends is $C_w \cdot w \cdot p_i$, so we amortize the one-time cost over the expected duration of the stream to calculate the cost of wasted prefetches per second as $(C_w \cdot w \cdot p_i)/D$. Therefore, to minimize the overhead of prefetching, we must solve the following problem:

$$\text{minimize} \sum_i (C_p \frac{b_i}{p_i} + C_w \frac{wp_i}{D}) \tag{4.1}$$

Constraint on Memory Use

An additional constraint on the problem of choosing the best prefetch sizes for each bit rate is that there is a limited amount of memory available for caching prefetched content. Rather than assuming a particular page replacement algorithm, we simply assume that prefetched pages are not evicted before they are requested by clients. Since all the prefetched pages must fit in memory of size M, our constraint on memory use is:

$$\sum_i p_i = M \tag{4.2}$$

We use the method of Lagrange multipliers to minimize the cost of prefetching, subject to the memory constraint:

$$\text{minimize } \Lambda(p_i, \lambda) = \sum_i (C_p \frac{b_i}{p_i} + C_w \frac{w p_i}{D}) + \lambda(\sum_i p_i - M) \tag{4.3}$$

where λ is the Lagrange multiplier. This equation is minimized when the partial derivatives, with respect to p_i, are zero:

$$\forall i, \frac{C_p b_i}{p_i^2} = C_w \frac{w}{D} + \lambda \qquad \Longrightarrow \qquad p_i = \sqrt{\frac{C_p b_i}{C_w \frac{w}{D} + \lambda}} \tag{4.4}$$

We do not use this equation directly to compute prefetch sizes because it is difficult to quantify some of the constants. Instead we note that all factors in this equation other than b_i and p_i are constants, and therefore prefetch sizes that are proportional to the square root of the bit rate will minimize the overhead costs of prefetching, and thereby maximize the aggregate throughput of client requests. In practice, we would use the automated algorithm to choose a prefetch size for a designated bit rate, then scale the prefetch sizes for other bit rates in proportion to the ratio of the square root of the bit rate divided by the square root of the designated bit rate.

Both our theoretical analysis and that of Gill and Bathen make assumptions that do not necessarily hold for real-world streaming video workloads. For this reason, we compare these alternatives for choosing prefetch sizes, either proportionally to the bit rate or to the square root of the bit rate, in two ways. First, in Section 4.5.4, we conduct experiments to determine which rule yields better results for the YouTube-like benchmark and the memory management algorithm implemented by the FreeBSD kernel (which we believe is LRU-based). Second, in Section 5.6.3, we

use simulation to compare different prefetch algorithms for servicing the workload of a Netflix server.

Now that we have described a mechanism for handling mixed bit rates and an automated algorithm for choosing prefetch sizes, we conduct a number of experiments to evaluate the effectiveness of the algorithms.

4.4 Changes to Experiments

After the preliminary experiments described in the previous chapter, we made three changes for the experiments in this chapter. First, we change the hard drives we use in the server. In the previous chapter, we used a server-class SAS hard drive, which is unlikely to be used for a production system because it is very expensive for its relatively small capacity. In its place, we use two cheaper and larger SATA hard drives that are more likely to be used for a production video server (such as Netflix servers [69]). Second, we populate the hard drives with two different sets of files, representing titles with two different bit rates. We made this change for convenience, so we could test a multi-bitrate workload, and so that we do not have to repopulate the hard drive to test different bit rate titles. Finally, as previously mentioned in Section 4.1, we report results only for the highest request rate that could be supported without timeouts. We will discuss these changes in detail in the sections that follow.

4.4.1 Server Configuration

The server hardware and software used for experiments in this chapter are similar to those in the previous chapter (described in section 3.3.1). We updated the kernel to FreeBSD 9.1-RELEASE, re-tuned the kernel and web server settings, and observed that the change in kernel version had little effect on experimental results. We continue to use a fast 10,000 RPM SAS drive to store operating system files and applications, but we no longer use an SAS drive to store title content and instead use two different SATA drives. The hard drive used to store content for most experiments is a 1.0 TB 5,400 RPM Western Digital Red drive (model WD10EFRX), chosen because it is advertised to be energy efficient while providing high throughput. We also use a faster 1.0 TB 7,200 RPM Seagate drive for comparison (model ST 1000DM003).

The server contains 32 GB of RAM, but for experiments, we configure the kernel to recognize a smaller amount, typically 4 GB for most experiments. We use a small amount of system memory so that our results are relevant for production servers which typically contain multiple hard drives. Our experiments use a single hard drive, but we expect the results to scale with the

number of hard drives, if we provide the same amount of system memory per drive. For example, we expect that our results using 4 GB of system memory would be applicable to a production server with 36 hard drives and 144 GB of system memory. Since servers are typically limited to a few hundred GB of system memory, we restrict our use of system memory to at most 8 GB, which would be scaled to 288 GB for a production server with 36 hard drives.

We use the ASAP version of the `userver` web server for experiments as well as the original "vanilla" version of the `userver` for comparison purposes. The ASAP version of the `userver` has been further modified to collect extra statistics, which is possible because we implement prefetching at the application layer. We monitor data that is evicted before it is used (*prefetch eviction*), the time between prefetching and eviction (*eviction time*), and data that is prefetched but never requested (*wasted prefetches*). These and other metrics are used in our automated prefetching algorithm, described in Section 4.2.

4.4.2 Workload Characteristics

We populate the hard drives on the server with files that represent two different fixed bit rates: 419 Kbps, to represent the common bit rate for YouTube as used in the previous chapter (called *SD*), and 2,095 Kbps to represent higher definition titles (called *HD*). We chose a bit rate of 2,095 Kbps because it is close to the average bit rate reported for the most common high bit rate format requested by YouTube clients [34]. For these two types of files, ten seconds of title content is represented by 0.5 MB and 2.5 MB of data, respectively. Title content is stored unchunked, with one file per title, and clients issue HTTP range requests for segments of title content.

We approximately fill the 1 TB hard drives on the server by creating 20,000 SD files and 8,000 HD files, which are inter-mixed so that both types of file are present in all zones on a drive. Titles are stored so that there is no particular placement due to popularity.

Titles have a Zipf popularity distribution with $\alpha = 0.8$ for all experiments except where otherwise noted. The title duration distribution is the same as used in the previous chapter, where the average duration of a title is 267 seconds. Therefore, SD and HD files have an average size of 13 MB and 66 MB, respectively. The average duration of a session is 162 seconds. The title duration and viewing duration distributions are the same for HD and SD files.

4.4.3 Experiment Procedure

The goal of the experiments in this chapter is to determine which prefetch size enables the highest throughput, for particular workload characteristics and server resources. To determine this, we

perform a number of experiments for each candidate prefetch size, varying the client request rate and observing when the server is overloaded. To find the highest request rate that can be serviced, we increase the aggregate request rate in 2 MB/s increments, until the rate is high enough that timeouts are detected. Each experiment takes 30-50 minutes, so it takes several hours to determine the highest throughput supported by each prefetch size. We normally investigate 6-10 different prefetch sizes to determine the best prefetch size for a given configuration.

Our criterion for determining whether a given aggregate request rate can be serviced is that the response time does not exceed a 10 second timeout threshold for any request in the benchmark workload. This is the *maximum failure-free rate*, and is a very strict test. Real-world client implementations typically buffer more than 10 seconds worth of title content, so users will not experience rebuffering after such a short delay. Also, users will typically accept more than one rebuffering delay. We use the strict criteria to ensure consistency across experiments since the same data is requested for all successful experiments, but failed sessions do not all request the same content.

For the SD experiments, the clients request 92 GB worth of data in 12,000 viewing sessions with the number of concurrent sessions peaking at 650. When using a Zipf α parameter of 0.8, 69% of titles are requested a single time and the average number of views per title is 2.1.

For the HD experiments, the clients request 148 GB worth of data in 4,000 sessions with a peak of 250 concurrent sessions. With $\alpha = 0.8$, 72% of titles are requested only once, and the average number of views per title is 1.9.

4.5 Experimental Evaluation

In the following sections, we present results from a suite of experiments for two reasons. First, there is a wide variety of potential deployment scenarios, with different server hardware and workload characteristics, and we would like to understand the factors that impact the best choice for prefetch size. Second, we would like to test the effectiveness of the automated algorithm for different deployment scenarios. We investigate two different server configuration choices: the amount of system memory and the performance characteristics of the hard drives used to store content. We also investigate the effect of the title popularity distribution, by generating workloads with different α values. Finally, we evaluate different options for handling multi-bitrate workloads.

For each variation in workload or hardware configuration, we perform a number of experiments to determine the maximum failure-free throughput the `userver` can achieve for a range of fixed prefetch sizes (representing a manual tuning). We compare those results to the automated

algorithm (labeled "A" in the graphs in this chapter), as well as the vanilla web server (labeled "V"). In all cases, we present the average throughput of the `userver` (Actual Tput), the hard disk (Disk Tput), and prefetch evictions (Evictions Tput).

For the automated algorithm, in addition to the parameters shown in Table 4.2, we also choose a starting prefetch size, which we determined based on work done by Li et al. [58]. Their analysis shows that the prefetch size should be set so that the time required to transfer the data from disk is equal to the average seek time. Based on that analysis we experimentally determined that the WD Red drive should use a starting prefetch size of 3 and 4 MB for the SD and HD workloads, respectively. Similarly, for the Seagate drive, we use a starting prefetch size of 2 and 3 MB for the SD and HD workloads, respectively. While other starting sizes are possible, these were chosen using previously established best practices and because good estimates for the starting size help to control the duration of experiments (recall that adjustments in prefetch sizes must be done gradually over time). Note that in many cases these starting sizes are not good choices and that despite having to slowly make prefetch size adjustments, our algorithm does converge on sizes that are appropriate for the system and workload.

Parameter	Value
adjust_interval	180 seconds
step_size	0.5 MB
score_deflation_factor	5%
busy_threshold	60%
evict_threshold	5%

Table 4.2: Automated algorithm parameters

4.5.1 Effect of System Memory

We performed experiments with three different amounts of system memory by changing the `hw.physmem` kernel parameter for 2 GB, 4 GB and 8 GB of physical memory. The SD workload, as shown in Figure 4.5, is quite sensitive to the amount of system memory. Disk throughput generally increases with prefetch size regardless of the amount of system memory, but if there is too little system memory available to store the prefetched data, the extra disk throughput results in evictions rather than actual throughput. The throughput when using the automated algorithm

(labeled "A") is within 5% of the throughput when using the best fixed prefetch size, demonstrating that the performance with the automated algorithm is comparable to that of the best hand tuned value. The throughput when using the unmodified vanilla version of the `userver` (labelled "V") is lower than when using the best prefetch sizes of 2 or 4 MB.

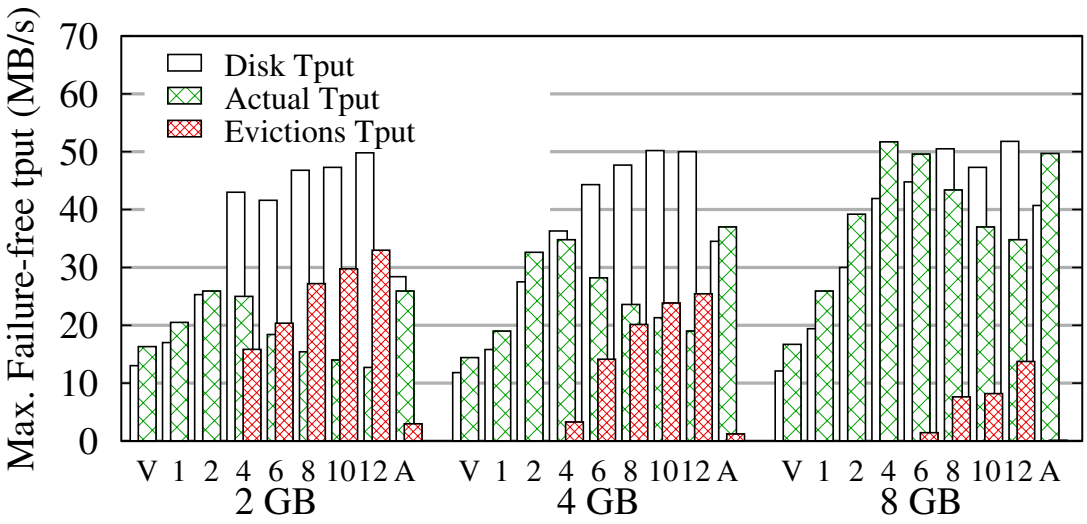


Figure 4.5: Comparing prefetching techniques for SD files while varying system memory

These results also provide insight into reasons why increasing system memory improves throughput. The benefits of caching are seen when actual throughput is greater than disk throughput. For example, with a 2 MB prefetch size, increasing system memory from 2 GB to 8 GB results in a 54% increase in actual throughput without a significant change in disk throughput. A larger amount of system memory also enables the use of larger prefetch sizes. When using 8 GB of system memory, throughput increases by an additional 35% when we increase the prefetch size from 2 MB to 4 MB.

We repeated the experiments using an HD workload and obtained the results shown in Figure 4.6. Compared to the SD workload, when using the same prefetch size, the eviction throughput is lower for all memory configurations This is because the bit rate of an HD title is 5 times higher than an SD title, so only $1/5$ as many HD clients can be serviced for a given throughput. Less system memory is required to store prefetched data for the fewer (HD) clients, because the amount of memory needed to store prefetched data is equal to the number of concurrent clients multiplied by the prefetch size. As a result, there are fewer evictions for a given prefetch size. Additionally, the improvements in throughput with larger system memory sizes are mainly due to

increasing the number of cache hits. Going from 2 GB to 8 GB of system memory, when using a prefetch size of 6 MB, there is a 28% improvement in actual throughput with no significant gain in disk throughput. Then, using 8 GB of system memory, there is an additional 15% throughput gain by increasing the prefetch size from 6 MB to 12 MB.

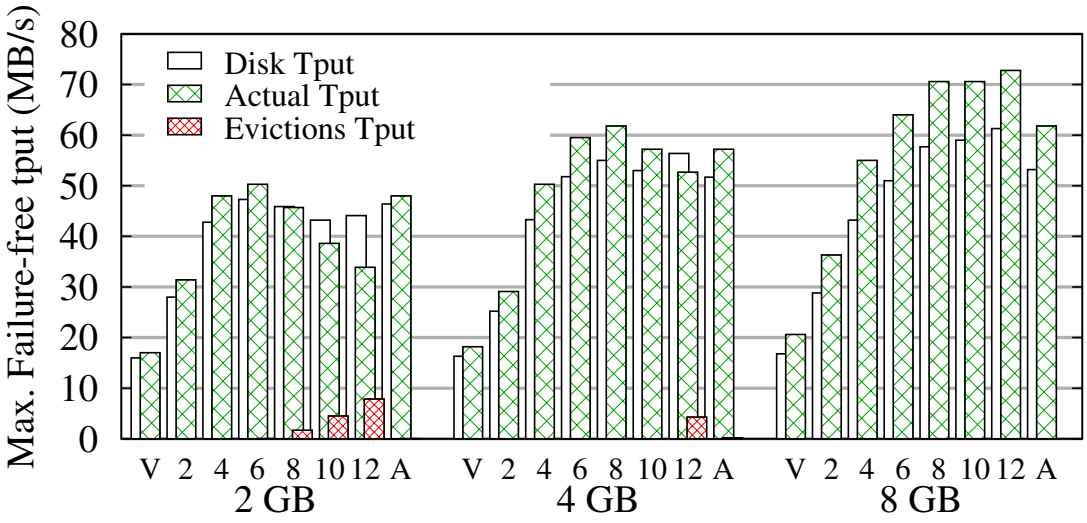


Figure 4.6: Comparing prefetching techniques for HD files while varying system memory

For most of the test configurations, the actual throughput of the automated algorithm is similar to using the best fixed prefetch size. In the worst case, when using 8 GB of system memory, throughput is 16% lower. This is because the starting prefetch size of 4 MB is a poor estimate compared to the best fixed prefetch size of 12 MB, and the experiment is too short to allow the adaptive algorithm to converge on a better prefetch size.

4.5.2 Effect of Popularity Distribution

The distribution of the workload directly impacts the effectiveness of caching. Changing the α parameter of the Zipf popularity distribution affects the number of requests for the most popular files. Workloads with higher α values more frequently request popular files and should benefit more from the file system cache.

Although prefetching is not directly affected by the popularity distribution, caching and prefetching compete for system memory. In order to determine if the best prefetch size is sen-

sitive to the popularity distribution, we generated two additional workloads: one with $\alpha = 0.6$ and the other with $\alpha = 1$. These are compared with the standard workload that uses $\alpha = 0.8$. All three workloads use the same SD file set, and files have the same popularity rank in all the distributions. This ensures that, to the extent possible, the same titles are requested in each workload.

Figure 4.7 shows the maximum failure-free rates that could be achieved across the different workloads, using 4 GB of system memory. These results demonstrate that the actual throughput increases with larger α values, across all prefetch sizes. In contrast, there is little or no change in disk throughput and eviction throughput across the prefetch sizes. For these experiments, the workload distribution has little impact on prefetching so the best throughput is achieved using a 4 MB prefetch size, regardless of the α parameter. The increase in actual throughput with larger values of α is caused by an increase in cache hits.

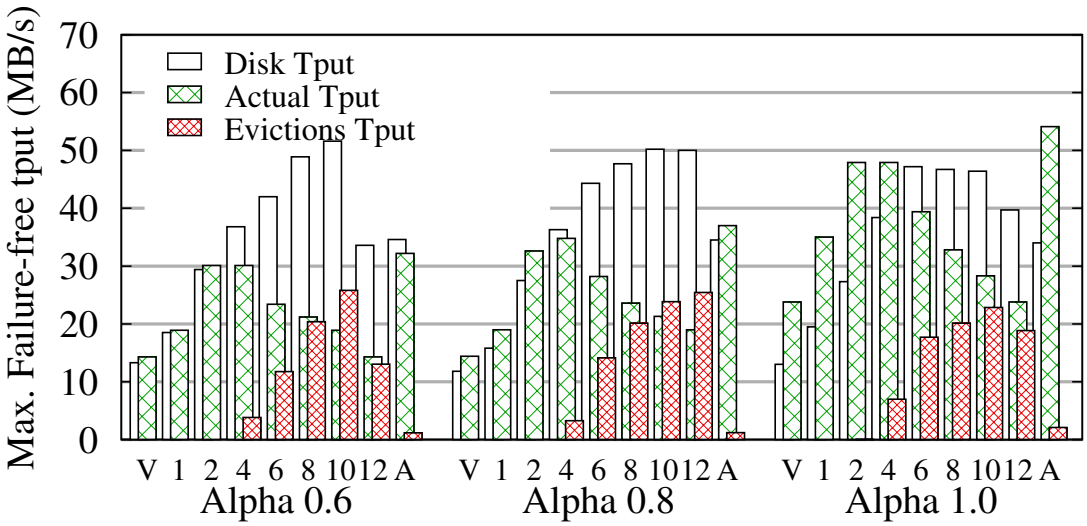


Figure 4.7: Different popularity distributions (α), with SD files and 4 GB of memory

Again, in this case, the automated algorithm works well with these workloads. With the $\alpha = 1$ workload, the automated algorithm provides 14% higher throughput compared to the best fixed prefetch size. The automated algorithm was able to achieve higher throughput because it converged on a prefetch size of 3 MB, which is not one of the fixed prefetch sizes that was evaluated in our manual tuning process.

4.5.3 Effect of Hard Drive Characteristics

The experiments presented so far use a 1.0 TB 5,400 RPM Western Digital Red drive. As a point of comparison, we repeat the SD and HD experiments of Section 4.1 using a 1.0 TB 7,200 RPM Seagate drive with lower rotational and seek latencies. Both drives have the same capacity and we carefully populate the drives with files of the same size in the same locations using the procedure in Section 3.5.2 to ensure results obtained using these drives can be directly compared.

The differences in the speeds of these disks are reflected in Figure 4.8, which shows the results of prefetch size experiments using 4 GB of system memory. The results for the Red drive were previously shown in Figure 4.1 and are included here for convenience.

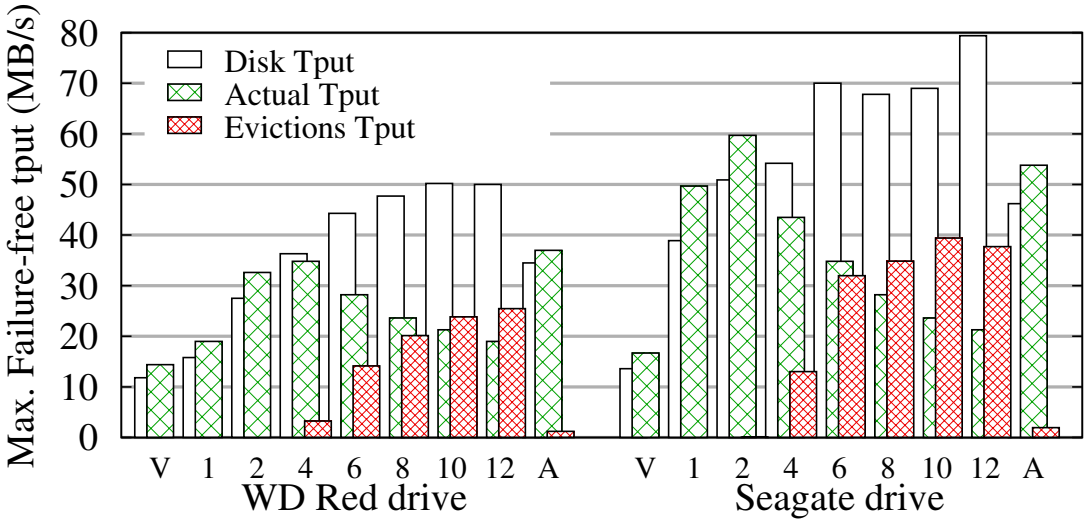


Figure 4.8: Comparing prefetching techniques on different disks with SD files and 4 GB memory

Due to the higher transfer rates and shorter seek times, the Seagate drive is able to achieve higher throughput when using small prefetch sizes. In addition, the higher throughput of the Seagate drive allows the `userver` to support about twice as many concurrent clients. The tradeoff is that, by doubling the number of concurrent clients, the memory required to store all of the prefetched data is also doubled. The increased memory pressure causes more evictions when using the Seagate drive. The differences between the drives are smaller when using the HD workload, shown in Figure 4.9. When servicing HD files, there are fewer concurrent clients, which reduces memory pressure and the eviction rates when compared with the SD case.

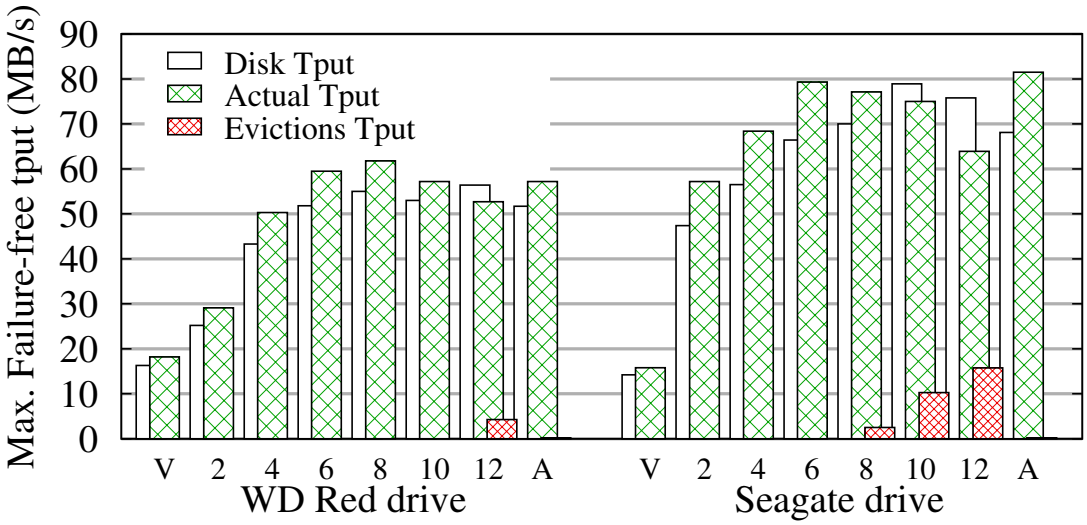


Figure 4.9: Comparing prefetching techniques on different disks with HD files and 4 GB memory

When comparing the improvements due to asynchronous serialized aggressive prefetching with the Vanilla case, the throughput increases are larger when using the Seagate drive than when using the WD Red Drive. For the WD Red drive the improvements are as large as a factor of 2.4 and 3.4 for the SD and HD workloads, respectively. However, for the Seagate drives the improvements are as large as a factor of 3.6 and 5.2 for the SD and HD workloads, respectively. The throughput using the automated algorithm is close to the throughput using the best manually-determined prefetch size, ranging from 11% lower for the SD workload on the Seagate drive to 7% higher for the SD workload on the Red drive.

4.5.4 Effect of Multi-Bitrate Workloads

The workloads in the experiments prior to this point have used a single bit rate for all files. In Section 4.3, we devised a method for handling workloads with multiple bit rates, and we now test our method using a workload that contains 50% HD files and 50% SD files.

Some of the results of the experiments we have conducted with this multiple bit rate workload are shown in Figure 4.10. For these experiments we use either a fixed 6 MB prefetch size (which provides the highest throughput) or we use the automated algorithm to compute a prefetch size for the HD files, then scale that prefetch size to be used for SD files. The labels just below the x-axis (.1, .2, .45, .75 and 1) specify the different scaling factors we applied in different experiments. The other labels on the x-axis (6 MB, A, and V) show the prefetch algorithm used,

either a fixed 6 MB prefetch size, the automated algorithm ("A") or the vanilla `userver` ("V"), which does not use a scaling factor.

The scaling factors of key interest are 1, 0.45 and 0.2. A factor of 1 means that the same prefetch size is used for SD files as HD files. A factor of 0.45 is calculated using the rule we developed in Section 4.3. That is, the prefetch size should be proportional to the square root of the bit rate. Since HD files have a 5 times higher bit rate than SD files, by our analysis, the scale factor for SD files should be $\sqrt{1/5} = 0.45$. Therefore, when the HD file prefetch size is 6 MB the SD prefetch size is 2.7 MB. We include a scaling factor of 0.2 because previous work by Gill and Bathen [37] determined that it is optimal (under specific conditions) to prefetch an amount proportional to the bit rate, and the SD bit rate is 20% of the HD bit rate. In addition, 0.1 and 0.75 are included to examine the sensitivity of the results to the scaling factor. Note that the best prefetch size for this workload is close to that for the purely HD workload. This is not surprising because the throughput required to service the HD requests dominates the throughput requirements for the SD requests; although there are equal numbers of clients requesting SD and HD files, HD files account for $5/6$ of the total throughput and SD files are $1/6$ of the total throughput.

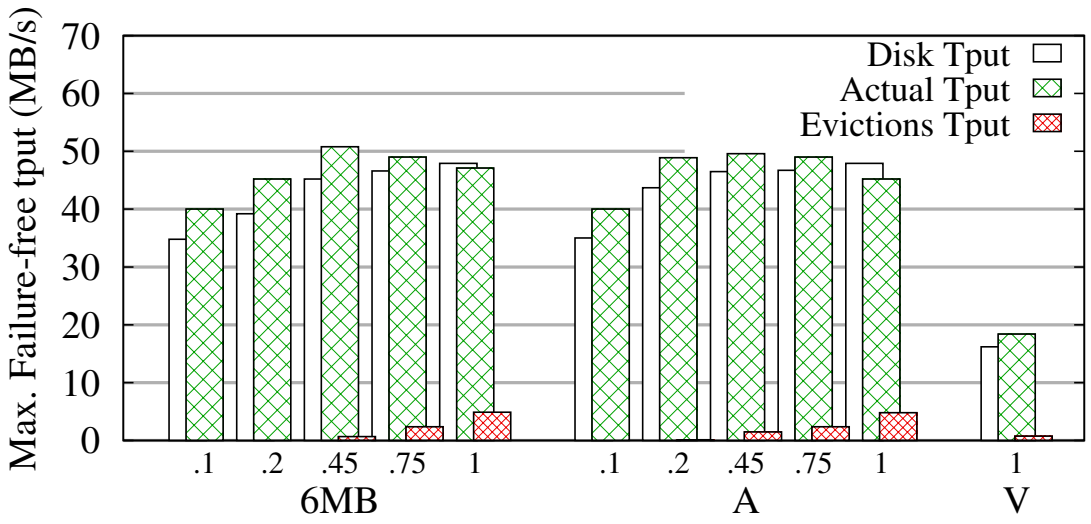


Figure 4.10: Varying scaling factors and the prefetching technique while using 50% SD files and 50% HD files on WD Red drive.

We observe that the differences in throughput are fairly small across the different scaling factors, for this workload and system configuration. This indicates that throughput is not very sensitive to the scaling factor when servicing this workload. However, the throughput obtained

with the scaling factor of 0.45 is as good or better than that achieved with the other scaling factors. In future work, it would be interesting to determine if there are circumstances under which the differences are larger. Finally, we note that the automated algorithm performs very well and is again comparable to the best fixed prefetch sizes for this workload.

4.6 Discussion

Our results show the importance of having sufficient server memory to support a large prefetch size. For example, with the scenarios considered in Figure 4.5, increasing the server memory from 2 to 8 GB enables a doubling of the throughput. Although these results are for a single disk system, appropriate scaling by the number of disks can yield insight applicable to the common deployment scenario in which an HTTP streaming video server is configured with many disks.

It might be possible to reduce the amount of system memory required by improving the memory management algorithms and reducing prefetch evictions. For our experiments, we use the existing FreeBSD memory management algorithms, but other studies have shown benefits can be obtained by treating prefetched memory specially [55, 13, 37] and by using streaming-specific caching algorithms such as interval caching [30]. In future work, we plan to investigate these techniques, which can be applied in concert with our automated algorithm.

Another possible area of future work concerns stream-specific adjustments to prefetch size. Our results show the potential benefit of using a prefetch size scaling factor based on the bit rate of the title being streamed. Considering additional characteristics might yield further benefits. For example, using a smaller prefetch size for new streams could be beneficial when there is a relatively high rate of termination by the user early in the playback of the title, as is often observed in practice [34, 11]. One might also take into account the title or user identity, for example prior work has observed that some users are "serial" early-quitters [11].

4.7 Chapter Summary

We have shown that the prefetch size that maximizes throughput varies with changes to workload and server hardware characteristics. We performed a large number of experiments to determine that the prefetch size is sensitive to the bit rate of title content, the popularity distribution of titles, the amount of system memory, and the performance characteristics of hard drives. Because it is very time-consuming to manually conduct the large number of experiments necessary to determine the best prefetch size, we developed an algorithm that continually monitors both the cache

hit rate and transfer times for hard drive transactions in order to dynamically and automatically choose the best prefetch size. We successfully applied the automated algorithm to all our experiment scenarios, thereby demonstrating that the algorithm is likely to be useful for a wide variety of potential production installations without requiring system administrators to select a prefetch size manually.

We also used a mathematical analysis of mixed bit rate workloads to show that the overhead of prefetching is minimized when the prefetch size is proportional to the square root of the bit rate. We validated our analysis by comparing it to other alternatives experimentally. We will make further use of this result in the next chapter, where we characterize the workload of Netflix servers as well as evaluating and improving prefetch algorithms for servicing that multi-bitrate workload.

Chapter 5

Netflix Server Workload

In Chapter 3 we created a benchmark that reproduces a YouTube-like workload in a laboratory environment. We then used that benchmark to conduct experiments to develop and evaluate different prefetch algorithms in Chapter 4. This approach allowed us to develop and test the ASAP architecture and an automated prefetch sizing algorithm, which together significantly improve web server throughput.

However, there are limitations to this work. First, YouTube is mainly a service for titles with short durations: predominantly user-generated videos and professionally-produced music videos. The characteristics of YouTube network traffic may differ from those of video services like Netflix, Amazon, HBO, Hulu, Apple and others, which stream long format, professionally produced content like TV shows and movies. Second, existing workload characterizations have typically been constructed from traces observed at the edge of a client network [34, 98, 39] and as a result, the traces contain only a subset of demand for content servers, forcing us to infer the total demand on a server. Finally, most existing workload studies were conducted for services that did not implement rate adaptation (typically called DASH, for Dynamic Adaptive Streaming over HTTP) to provide high quality video by adjusting the bit rate in reaction to changes in network and server conditions. Studies that have examined DASH have analyzed client implementations [45, 62, 79, 60] or the use of network bandwidth [33, 44, 63, 3, 6], rather than the impact of DASH on the servers. Therefore, in this chapter, we conduct a characterization of a second HTTP streaming video workload that does not have these limitations.

We were able to obtain anonymized log files (HTTP request traces) from two different production Netflix web servers. Netflix has over 81 million global subscribers [71] and supports an extremely diverse and representative set of client devices that implement DASH over broadband, DSL, WiFi and cellular connections. The log files capture all the requests to these servers, re-

gardless of the locations of the client devices. The log files provide detailed information about requests, and also identify individual viewing sessions, so we can reliably identify all of the HTTP requests made by a particular client device while a particular title is viewed. Additionally, we have obtained information about the nominal bit rates of the files that are stored on the server, which enables us to better understand the impact of rate adaptation on the server. This is an unprecedented view of the workload of a production server from what is, as of 2016, the largest source of HTTP streaming video. Characterizing this type of workload is an important step to understanding and optimizing the performance of the servers used to support the growing number of streaming video services.

In this chapter, we provide background information about how client devices interact with Netflix servers. We then analyze the server logs to characterize the titles in the catalog, viewing sessions, and information about client requests, which are markedly different than the specification of a YouTube-like workload in Chapter 3. Our goal is to understand the impact of rate adaptation on servers, in particular how the spatial locality of the workload is affected. We develop and use a *chain* abstraction to characterize the spatial locality of requests for the same title, and a *phase* abstraction to characterize the operation of the many different clients that access the Netflix service. We apply the knowledge we gain by characterizing the server workload to propose workload-specific prefetch algorithms. We analyze those algorithms and show that by using workload-specific characteristics, such as the probability that chains will be long or short, we can adjust prefetch sizes to reduce hard drive utilization and consume less system memory compared to a prefetch algorithm that does not use workload information. We expect that the techniques we use can be repeated for similar HTTP streaming video services with DASH clients, which represent the future of how TV shows and movies will be viewed.

5.1 Background

Netflix is a widely popular Internet service for streaming TV shows and movies (collectively called *titles*). In the past, Netflix made extensive use of CDNs such as Akamai, Limelight and Level 3 [3] but the rapid growth of its popularity has led it to create and manage its own CDN, starting in 2012 [70].

The `netflix.com` site is served from the Amazon AWS cloud in geographically relevant regions. However, audio and video content is serviced using high-capacity web servers or clusters of such servers, placed in Internet exchange sites around the world. In addition, ISPs may also utilize one or more Netflix-supplied servers inside their data centre, to reduce inter-ISP traffic [68]. Together these servers can be thought of as comprising the Netflix CDN.

The Netflix CDN does not operate like a traditional pull-based CDN. Nightly, during off-peak hours (called a *fill period*), the Netflix control plane predicts which titles are likely to be requested during the next 24 hour period and directs each individual content server to remove and add titles according to those predictions. Then, to playback a title selected by a user, a client device acquires a manifest from the control server that specifies which content servers should be accessed by the client and provides URLs for the files containing the different bit rates for the selected title. Clients strive to use the highest quality video and audio supported by the network and available content servers. They select a primary server for playback and for the most part continue to use that server unless it experiences low throughput, errors, timeouts or rebuffering events while playing at an already low bit rate.

5.1.1 Netflix Servers

Netflix servers are Open Connect Appliances (OCAs) [69] which run FreeBSD 10.0 and `nginx`. There are different hardware configurations that are continually evolving. We focus on a log file from a *Catalog* (or *Storage*) server which contains 36 hard drives of 3 TB and 6 SSDs of 512 GB. We also examine a second log file from a *flash cache* (or *offload*) server, which contains 14 SSDs of 1 TB each. A single storage server has too little capacity to store the entire Netflix catalog (about 2 Petabytes of data) so it is part of a cluster of 20 servers. An offload cluster contains copies of the most popular content from the catalog cluster. Typically both cluster types are deployed in Internet exchange sites.

The server logs obtained from Netflix contain information about every HTTP request that is received by the servers. Each log entry contains the URL of the file being read (which is anonymized), the offset and size of the request, a timestamp for the completion time of the request (with 1 second accuracy), the time required at the server to service the request, and the number of bytes sent to the client. Each log entry also specifies which playback device type was used and includes an anonymized viewing session identifier. Normally session identifiers are not included in web server logs and appropriately discerning sessions in such logs can be difficult because HTTP requests do not require an application-layer session. Subscribers are not identified in the log files, so we cannot tell which sessions involve the same subscriber, preventing us from analyzing user behaviours such as binge watching.

The request data provided in the server logs are in terms of bytes, which is difficult to interpret when files are available in many different bit rates and because variable bit rate (VBR) encodings are used. In order to convert byte values into quantities that are meaningful in the context of titles and which can be used to compare requests with different bit rates, we obtained information about all the files present on the server. For each file, we have the nominal bit rate of the file, the size

of the file in bytes, and the identity of the hard drive or SSD on which the file is stored. When necessary, we convert a file offset in bytes into a *nominal title time* by dividing the byte offset by the nominal bit rate. This is an approximation because the average bit rate of VBR-encoded content fluctuates over time and is not necessarily equal to the nominal bit rate at a given file offset. As a result, there may be variation in title-relative calculations, which is reduced by computing averages over many requests.

5.1.2 Data Collected

Table 5.1 provides statistics about the contents of the catalog server and flash cache log files. The catalog server log was collected over 24 hours, in March of 2014 and the flash cache log was collected over 23 hours in May of 2014.

Statistic	Catalog Server	Flash Cache
Total Data Sent	30.8 TB	75.9 TB
Average Throughput	2.9 Gbps	7.5 Gbps
Peak Throughput	5.5 Gbps	12.9 Gbps
Number of Sessions	126,064	284,986
Number of Unique Titles	9,793	1,170
Number of Unique Files	102,386	30,606
Number of TCP Connections	1,725,983	4,379,498
Number of HTTP Requests	64,993,469	192,814,827

Table 5.1: Summary of server log files contents

Figure 5.1 shows the aggregate throughput for each of the two servers, calculated by averaging the bytes serviced in 5 minute intervals. Requests are logged after servicing the request. As a result, the end of each log file will be missing requests that were issued but not completed before the end of the log period. To simplify the handling of these cases, we ignore sessions that start in the last hour of a log period. From 06:00 until 08:00 the Flash Cache is in the fill period. During this time it is adjusting its content rather than servicing client requests. For both servers, the peak throughput is about double the average throughput and as one would expect, the Flash Cache server is servicing considerably more traffic. Note that each server is capable of servicing substantially more traffic if required.

We compared several characteristics of the workloads for the two different servers and found they were very similar. The key exception is the popularity distribution of titles (shown in Sec-

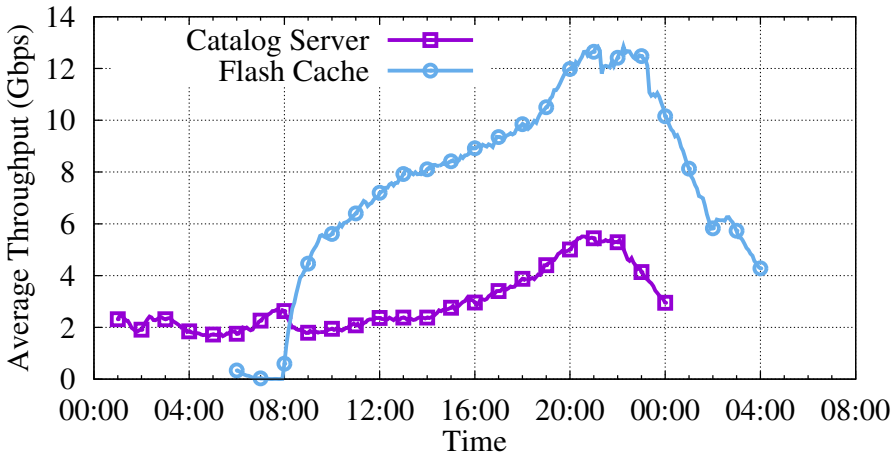


Figure 5.1: Catalog and Flash Cache throughput

tion 5.2.1), which is explained by the different roles for these servers. For all other characteristics, we only show results for the catalog server.

5.1.3 Netflix Clients

Netflix supports many different types of client devices, including consumer electronic devices like Blu-ray players and televisions, desktop computers, laptops, Android and iOS mobile devices, and game consoles. Audio and video content are encoded separately, at many different variable bit rates. Some clients require the content to be stored in separate audio and video files to allow the selection of video bit rate independently from the audio bit rate, and to allow audio playback in different languages. Other clients require audio and video content to be combined in the same file. The server log files include references to 5 different *audio* bit rates, 14 different *video* bit rates and 8 different *combined* bit rates.

Title content is divided into 2 second intervals called *segments*. An entire segment is required for decoding and playback, thus clients change bit rates at segment boundaries. Because segments vary in size, there is a table at the start of each file that specifies the offsets of all segments. When a session starts, the client downloads segment offsets and content from multiple files with different bit rates, then selects a starting bit rate that can be supported by available network bandwidth. Clients continue to download segments sequentially from the same file unless network or server conditions change (which may result in switching to a different bit rate) or a user event occurs (e.g., stopping or skipping to a new title position). Clients that are in a steady

state limit the number of segments they download ahead of the playback point to avoid waste when a user event occurs (i.e. the clients implement pacing). There is no simple relationship between segments and requests; some clients download multiple segments with a single request, while others use multiple concurrent HTTP requests to obtain segments in multiple parts.

Netflix clients issue HTTP GET requests using two different formats. Some clients fully specify the block of data they are requesting by providing the offsets of the first and last bytes, called *chunk* requests (or *range requests*). Alternately, clients can specify only the offset of the first byte, and the server will send data until the client terminates the TCP connection (or the file ends), called *open-ended* requests. Clients do not necessarily use only one of these request formats; often different request formats are used for audio and video content.

We provide more information about the requests issued by Netflix clients when we show two example viewing sessions in Section 5.2.3, and we provide measured statistics of request characteristics in Section 5.2.4. The wide variety of different client implementations is the major motivation for the chain and phase abstractions introduced in Sections 5.3 and 5.4.

5.2 Netflix Workload Characteristics

We use the methodology of Chapter 3 as a framework for characterizing the Netflix workload represented by the server traces. We first consider the fundamental aspects of streaming video workloads: the characteristics of the titles in the catalog, and how titles are accessed during viewing sessions. We then characterize the requests issued by Netflix clients, which are much more complicated and varied than are used for YouTube [34, 8].

5.2.1 Catalog Contents

The contents of the catalog of titles stored on a server dictate some of the basic characteristics of the workload. For example, the duration of titles affects the maximum duration of viewing sessions, the bit rates used for encoding titles affect the size of requests, and the popularity of files affects the temporal locality of the workload. In the following sections, we characterize the subset of the Netflix catalog that was requested by clients from the titles available on the catalog server.

Distribution of Bit Rate Choices

Netflix titles are encoded in many different formats and bit rates, both to accommodate a wide variety of devices and network access methods and to provide alternatives for rate adaptation algorithms [62]. In all cases, a single file is used to hold all content for a particular title and rate. Figure 5.2 shows the popularity of all bit rates requested by clients. The bit rate labels are not in strict numerical order, they are grouped by content type (e.g. audio, video, and combined). The graph shows popularity in terms of both the proportion of total bytes requested and the proportion of total nominal title time.

The two lowest bit rates are very popular in terms of nominal title time because they contain audio content, which is available in only a few encodings. The most popular video bit rates in terms of title time are the 1,750 and 3,000 Kbps versions (11% and 6%, respectively), but in terms of the volume of data, 25% of the total bytes are requested from files with bit rates of 5,800 Kbps. It is interesting that so many sessions have enough bandwidth for high bit rate content.

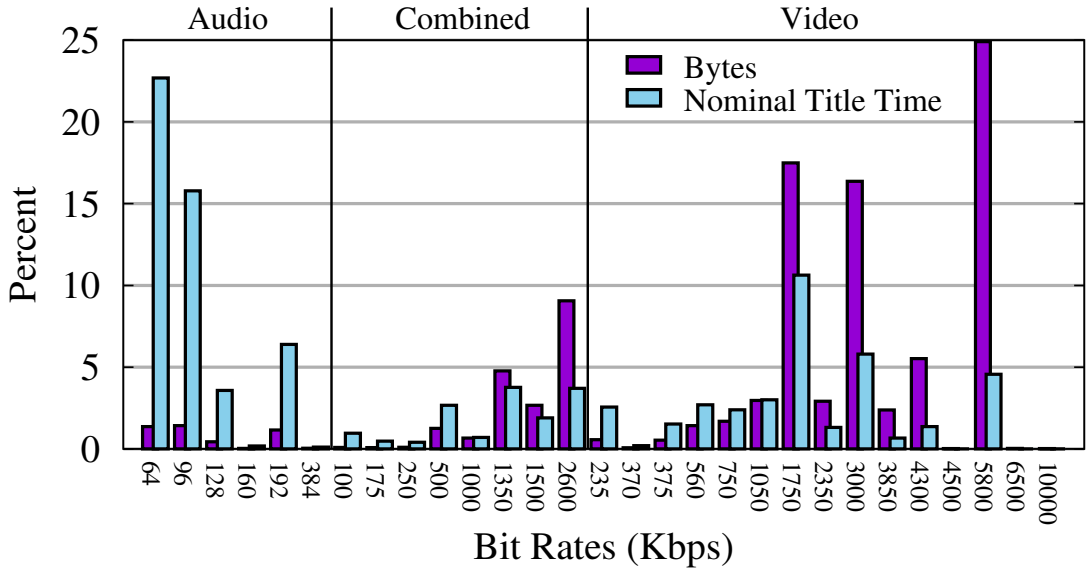


Figure 5.2: Bit rates chosen by clients

Distribution of Title Lengths

Figure 5.3 shows the distribution of durations for titles in the catalog that were requested by users in the catalog server trace. There are peaks at 22 and 43 minutes, corresponding to the durations of TV shows, and a broad peak at about 90 minutes, corresponding to movies. This variety of title durations is dramatically different than the title durations for short user-generated video services like YouTube. For YouTube (according to a recent survey [20]), more than 96% of titles are shorter than 10 minutes, while more than 99% of titles requested from the Netflix catalog server are longer than 10 minutes.

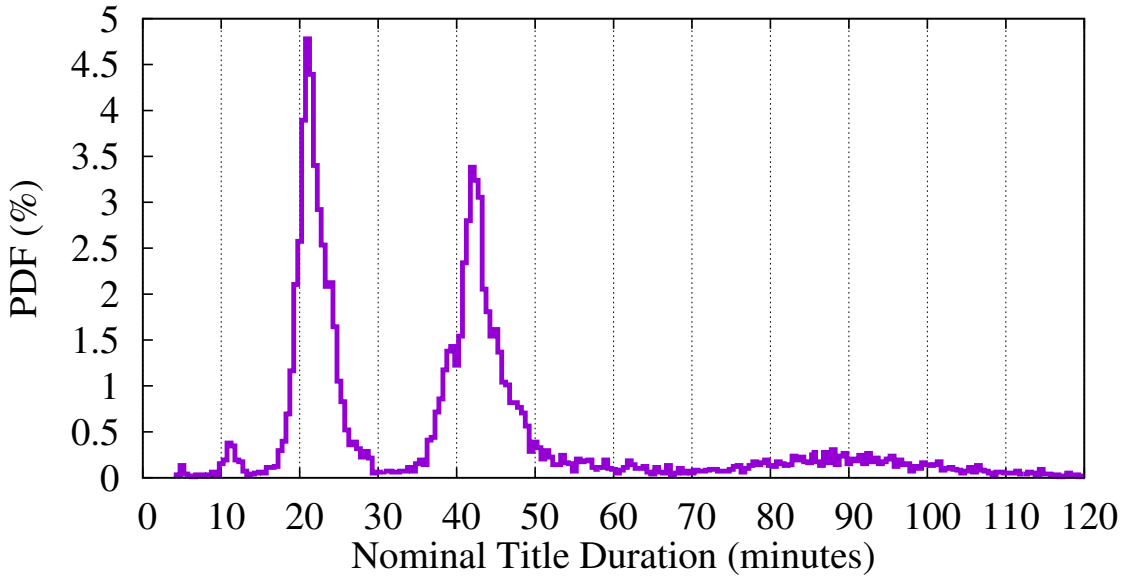


Figure 5.3: Duration of titles viewed

Popularity Distribution of Titles

Figure 5.4 provides two measures of popularity: the number of times that each distinct title was selected by users; and because titles are encoded in multiple bit rates, the number of times that individual files were requested. Note that log scales are used for both axes. We include measurements for both the catalog server and flash cache to show the differences between the workloads. The flash cache popularity distribution is Zipf-like with a cutoff, and it closely resembles the popularity distribution observed in workload studies for YouTube [24, 19], Yahoo! Video [48],

and PowerInfo [96]. For the catalog server curves, there is an unusual peak in the number of sessions that access the 20 most popular titles and 200 most popular files. This is a large divergence from the Zipf-like popularity distribution observed for the flash cache (and the other cited workloads).

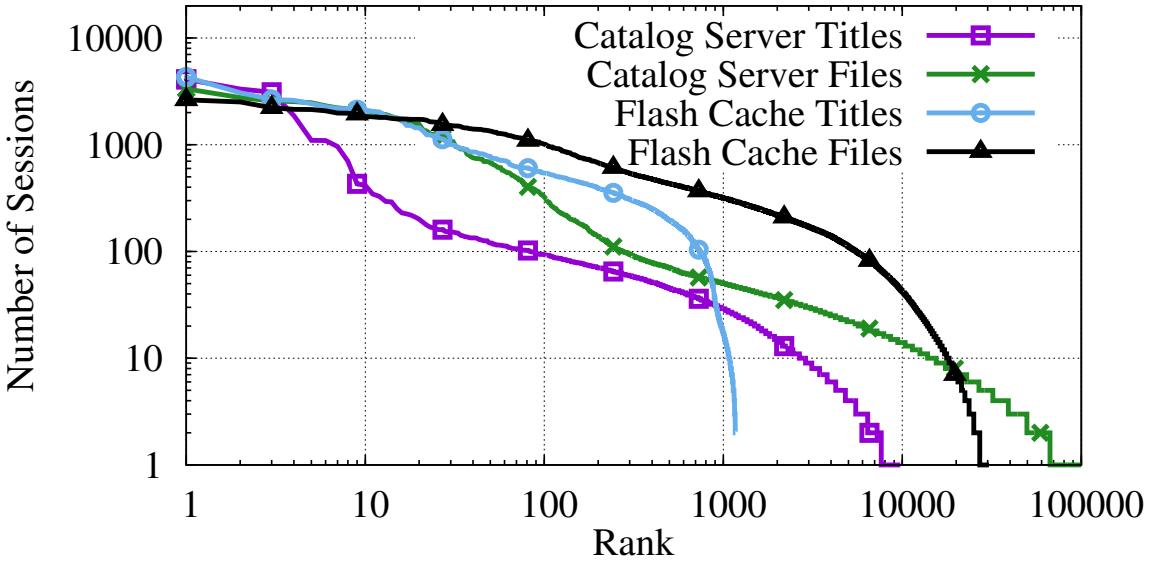


Figure 5.4: Number of times titles or files were viewed

5.2.2 Viewing Sessions

Viewing sessions start with a user selecting a particular title and include all the user actions (such as pausing or skipping position in the title) and client actions (such as switching bit rates) that occur while the client is playing the title. Sessions are identified in the server logs, but we do not have direct information about user and client actions, we infer those actions by observing the relative file offsets of requests and the intervals between request arrivals. In this section, we investigate the main characteristics of viewing sessions including: the duration of sessions, which parts of the titles are accessed during sessions, where sessions start and end within the title, and at which positions playback is paused.

Fraction of Title Downloaded

Since the duration of a title has a large influence on the duration of a viewing session for that title, we express session durations as a percentage of the title duration. There are some complications in computing this percentage because viewing sessions often involve requests for data from many different files with different bit rates, due to rate adaptation. We compute the percentage of title duration downloaded at each individual bit rate, then add those percentages together to get the overall percentage. We compute fractions separately for audio, video and combined files because clients often access them concurrently.

Figure 5.5 shows the cumulative percentage of sessions that are longer than a given percentage of a title. About 10% of sessions download more than 100% of the title, which is a result of repeated downloads of the same title content; so it appears that clients likely do not cache much content after it has been played. Clients typically download a greater percentage of the title for audio content than video content (e.g., about 30% download more than 100%). We find that for about half of the sessions, clients download 20% or less of the video content for a title. These results are remarkably similar to those reported for other services offering TV shows and movies [21], and for short user-generated video services like YouTube [34];

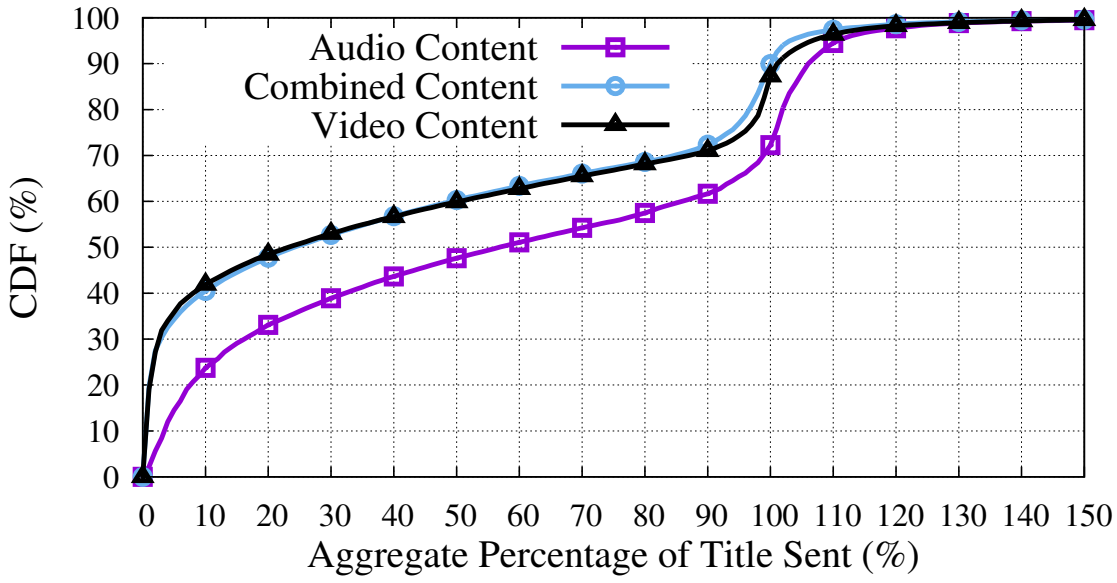


Figure 5.5: Fraction of files downloaded in a session

Portions of Titles Viewed

Having determined how much of a title is typically downloaded, we are interested in determining which parts of a title are typically accessed. To generate this data, we first convert the start and end offsets of each request into title-relative percentages by dividing the offsets by the size of the file. We then use bins of size 1% to record which portions of the title are requested. We use percentages to normalize the results for different title durations, and to accommodate files with different bit rates. Figure 5.6 shows that the first 1 percent of the title (i.e. the first 1% of at least one file) is accessed in 85% of all sessions, very few sessions access the end, and the number of sessions that access the title between 10% and 90% is remarkably uniform. As is the case in Figure 5.5, more audio content is accessed than video content during a session.

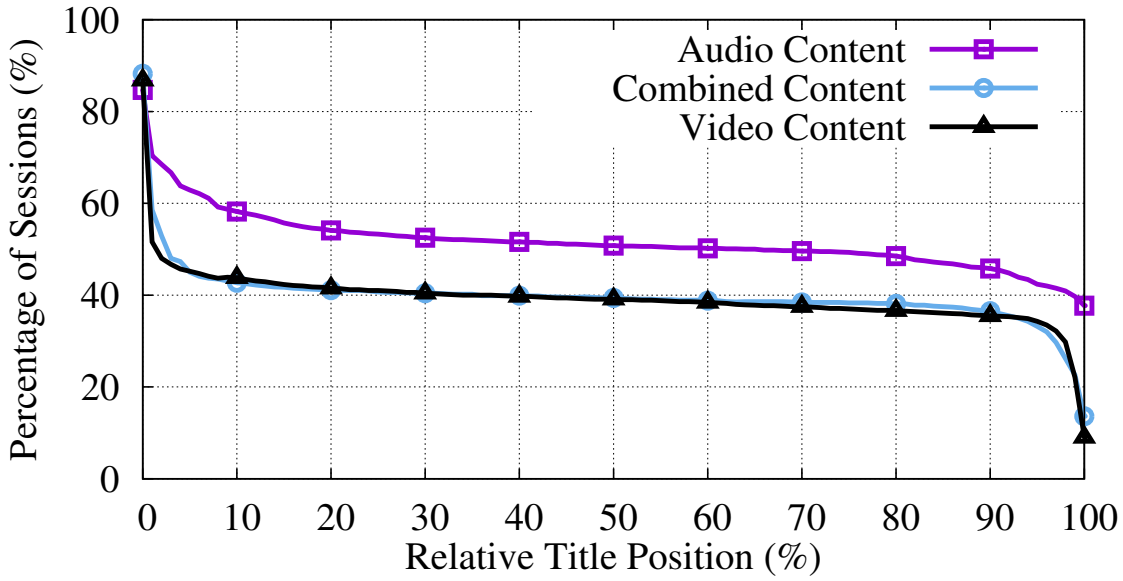


Figure 5.6: Portion of titles accessed during sessions

We are unable to determine whether this characteristic of the Netflix workload, that most parts of titles are accessed with similar frequency, is unusual. Most studies have not investigated this issue and instead assume sessions always start at the beginning of a title [22, 34, 98]. We are aware of only one other system, an educational media site [29], with similar uniform access frequencies.

Figure 5.7 shows information about the typical positions where sessions start and end, and the positions where pauses occur. To compute the start position of a session, we find the offset of the earliest request and divide it by the size of the file. We compute the end position similarly, by using the final offset of the request that is received last. To find pauses, we look for requests that are for adjacent portions of a file, but the interval between the requests is more than 40 seconds. We divide sessions into 1% bins to count the number of sessions that start, end or pause at that relative title position, and report the results as percentages relative to the total number of sessions. About 36% of sessions start in the first 1% of the title, 60% start within the first 5% of the title, and about 45% of all sessions reach the last 5% of the title. Outside of the first and last 5% of a title, pauses are about equally likely to occur at any relative position in a title. We have no information about the identity of the user for a session because that information is not available in the server logs, but the close match between the number of sessions starting and ending at the same point in the title may indicate that users resume sessions from where they stop, which might indicate a long-term pause.

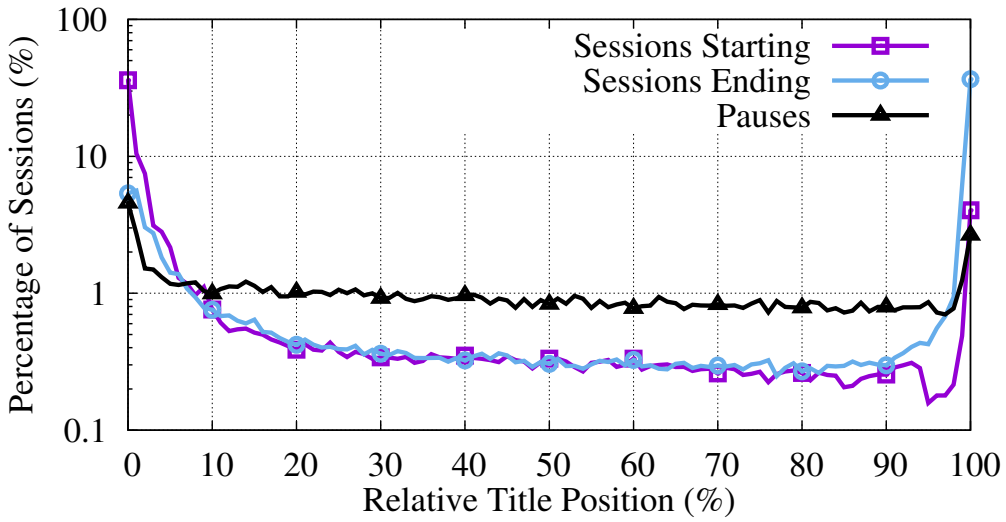


Figure 5.7: Relative start and end of sessions and pauses.

5.2.3 Example Sessions

To illustrate client behaviour during a session, we now present and describe two individual examples of sessions.

Figure 5.8 shows a session that lasts for about 32 minutes, consisting of requests for about 22 minutes of title content. The top half of Figure 5.8 uses rectangles to represent each request. The x-coordinate of the bottom left corner of each rectangle indicates the elapsed time at which the request is issued and the y-coordinate shows the position within the title of the first byte of the request. The x-coordinate of the top right corner indicates the time at which the request is completed and the y-coordinate identifies the position within the title of the last byte of the request. A tall and narrow rectangle indicates that a large request was serviced quickly and a short and wide rectangle denotes a small request that was serviced slowly. The y-axis values (position in the title by minutes) are an approximation, since the content is encoded using variable bit rates. We convert file offsets to title positions by dividing the byte offset of a request by the file size, then multiplying the resulting fraction by the title duration.

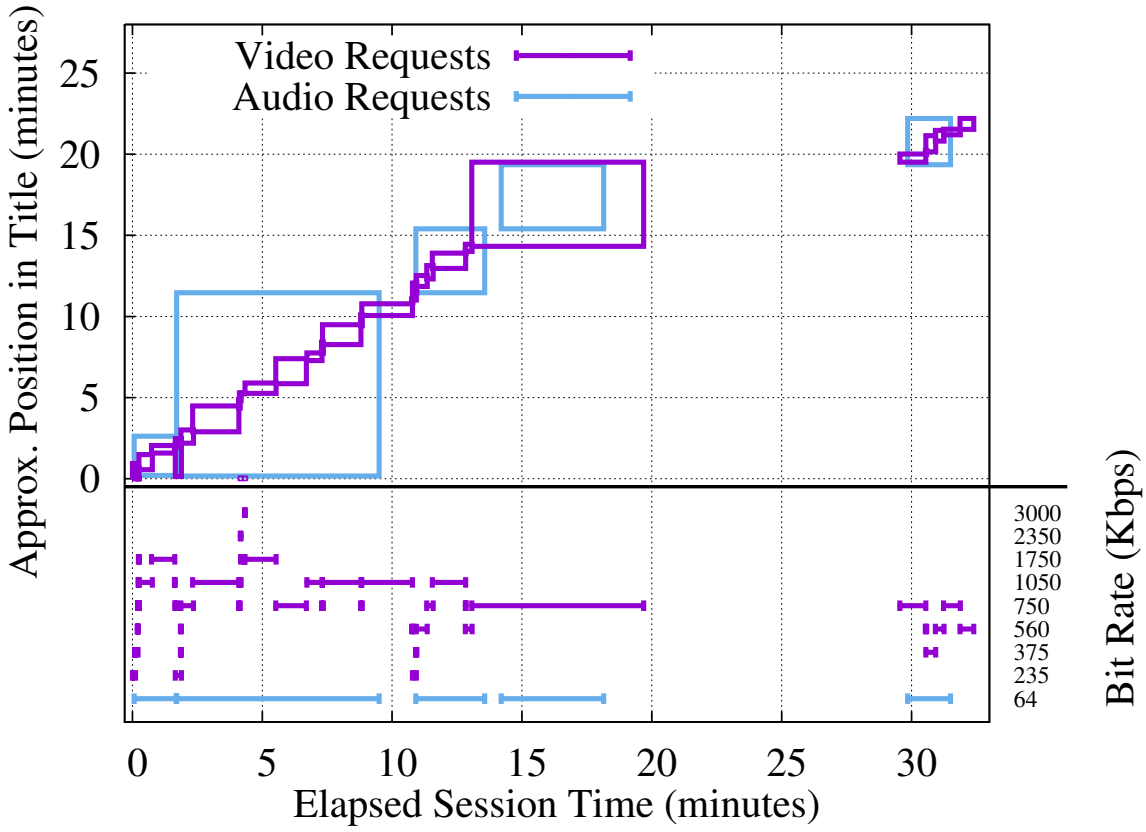


Figure 5.8: Requests issued during a session

The bottom half of Figure 5.8 shows the different bit rate files that are accessed during the

session, with the bit rates labelled on the right. Each interval (between two vertical lines) depicted in the bottom half of the figure corresponds to a rectangle in the top half.

There are a total of 51 open-ended requests in this session. Audio and video content is obtained from separate files and only a single audio bit rate is accessed. There is a period from about the 19 to 29 minute marks where there are no requests, likely indicating that the user paused playback. There are periods when a large number of files with different video bit rates are accessed over a short period, such as at times 0, 2, 4 and 11 minutes. In these cases 6, 6, 5 and 5 video bit rates are accessed, respectively. These unstable periods reflect the actions of the DASH algorithm, either downloading segment offset tables or performing rate adaptation. There are also stable periods when only a single bit rate is accessed, for an extended period from 13 to 20 minutes, as well as many shorter periods.

Figure 5.9 shows the first 1.5 minutes of a different example session. This zoomed-in view illustrates the use of chunk requests, and the spacing of the requests (the slope formed by the series of rectangles) reveals important details about request timing. The initial unstable period lasts for about 0.1 minutes, then the client accesses a single video bit rate for the remaining time. From 0.1 to 0.5 minutes, the client downloads about 2 minutes of title content in 0.4 minutes of elapsed time, so content is downloaded about 5 times faster than the bit rate of the content, indicating a period of time when the client is filling its playback buffer. After 0.5 minutes, 1 minute of content is downloaded in 1 minute of elapsed time, which indicates that pacing is occurring. These patterns of requests and inter-arrival timings are important characteristics of client implementations.

Figure 5.9 also reveals details about how chunk requests are issued. After 0.75 minutes of session time, requests are issued in clusters that are separated by time gaps, illustrating the method used for pacing chunk requests. Pacing occurs naturally due to TCP flow control for open-ended requests [62]. Requests in the clusters overlap in elapsed session time (e.g., at the 1 minute mark in the top portion) due to the use of parallel TCP connections. These concurrent requests are occasionally processed out of order, with examples of this just after 0.75 minutes of elapsed time and just before the 1 minute mark. In these cases, there are requests that are higher in the graph (indicating a later title time) while starting further to the left (indicating an earlier elapsed time).

From these examples, it is clear that DASH clients have complicated patterns of requests, and that different client implementations may use substantially different methods for downloading content.

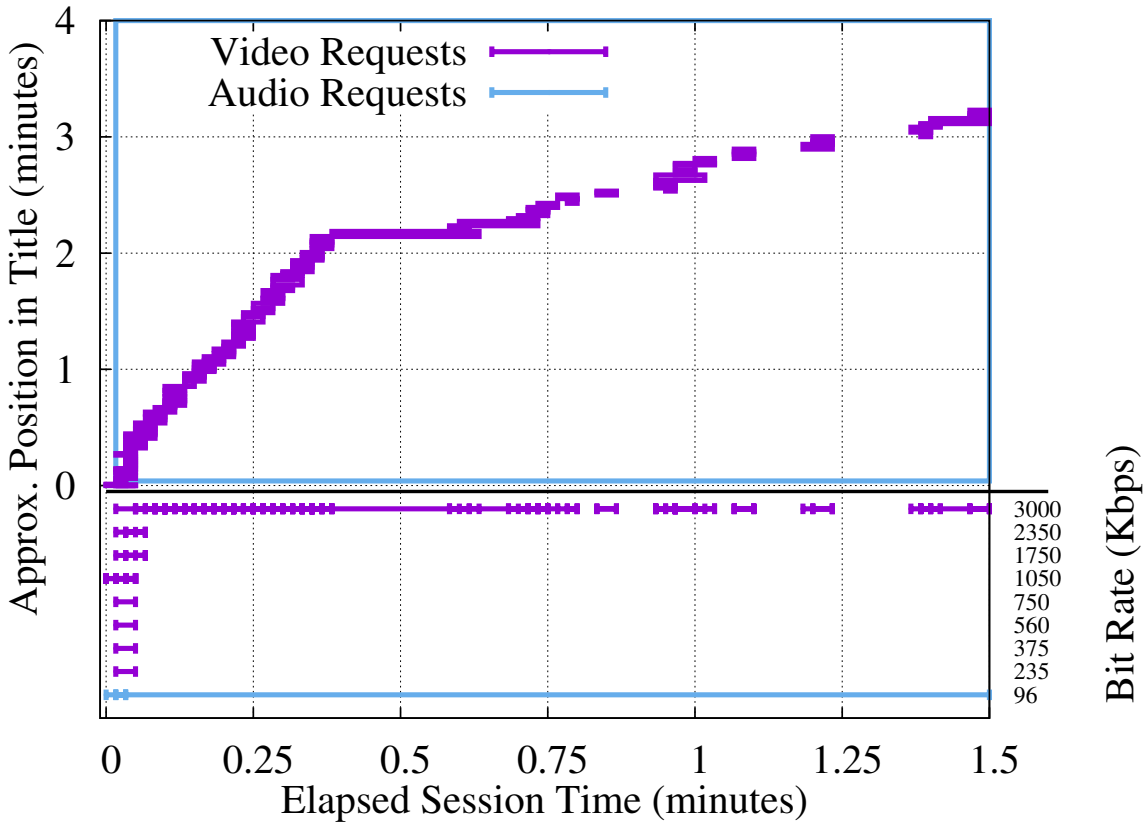


Figure 5.9: Details of session startup

5.2.4 Request Statistics

The variation in the format and sizes of requests issued by Netflix clients reflects the wide variety of client devices. Table 5.2 categorizes requests based on the type of content, for both open-ended and chunk requests. Audio accounts for about 5% of the total bytes requested. Only 1.3% of all requests are open-ended, but they account for 1/3 of the total bytes requested. Clients are more likely to use open-ended requests for audio and combined content, but more likely to use chunk requests for video content. Clients often use different request types for audio and non-audio content, and for about 10% of sessions, clients alternate between open-ended and chunk requests.

There is significant variation in request sizes, even after considering the different available bit rates. The number of bytes downloaded with open-ended requests is extremely variable, and

Type	Requests	%	GB	%
Open Audio	208,045	0.3	1,095.7	3.5
Open Video	445,728	0.7	5,828.1	18.5
Open Combined	166,720	0.3	4,033.0	12.8
Open Total	820,493	1.3	10,956.8	34.8
Chunk Audio	2,654,422	4.1	299.2	0.9
Chunk Video	53,758,669	82.3	18,411.5	58.4
Chunk Combined	8,102,517	12.4	1,832.2	5.8
Chunk Total	64,515,608	98.7	20,542.9	65.2
Total	65,336,101	100.0	31,499.6	100.0

Table 5.2: Prevalence of request types

depends more on session events than client implementations, so we examine only the chunk requests in Figure 5.10. The figure shows the average lengths of chunk requests, both in bytes (left axis), and in nominal title time (right axis). For most audio and combined content, the average amount requested is more than 2 seconds in terms of title time; this is caused by clients requesting more than one 2 second segment at a time. For video content requests (bit rates 235-10,000), the average title time is between 0.4 and 1.9 seconds because some clients divide a segment into multiple parts and download the parts in parallel, while others download multiple segments in a single request, and some do both.

Because parallel downloading has a large effect on the request size, we measure how frequently clients issue parallel requests for the same file. Table 5.3 shows, for each file in each session, the percentage of files that are downloaded using parallel connections at some point in a session. We compute these numbers by finding the maximum number of concurrent requests to the same file during the same second, which is the granularity of the timestamps. More than half of files are downloaded in parallel during a session.

Number of Connections	1	2	3	4	5	≥ 6
Percent of Files	41.9	18.0	33.0	6.0	0.8	0.3

Table 5.3: Per-file use of parallel downloads

Next, we introduce an abstraction that provides a unified view of requests that is independent of how different clients issue requests.

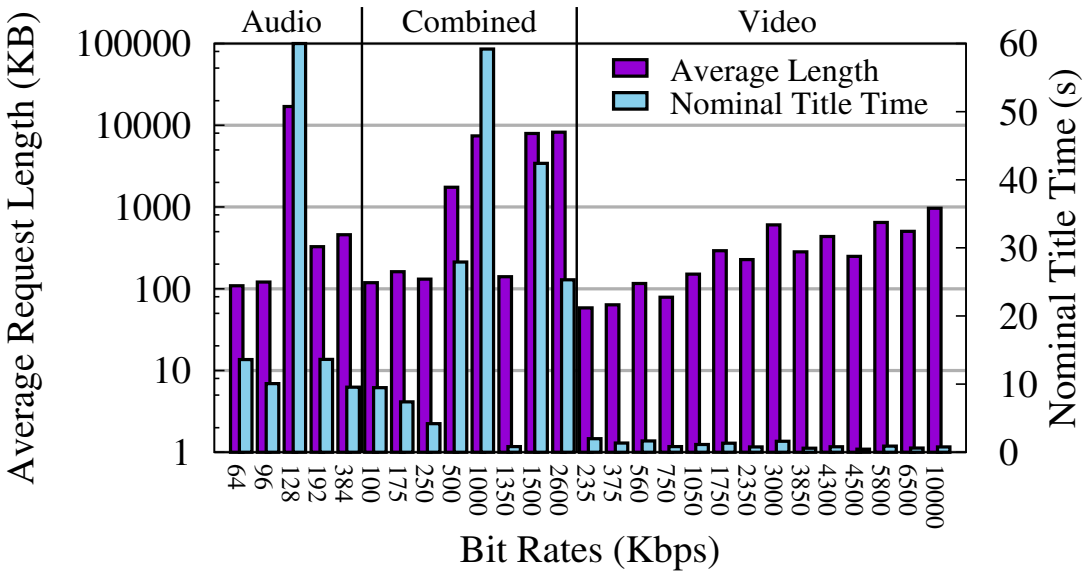


Figure 5.10: Average sizes of chunk requests

5.3 Chains

In this section we analyze the spatial locality of the server workload. Although requests from individual clients are commonly viewed as being highly-sequential, trick play operations, and especially rate adaptation (which causes requests to be issued for different files) can disrupt sequentiality. Determining the degree of spatial locality is important since it can be used to understand whether or not aggressive prefetching is likely to be as effective on this workload, as it has been on other workloads [91] [90]. Additionally, it may be possible to use workload characteristics to tailor the web server to better handle this particular workload. In Section 5.6, we use simulation to evaluate a prefetching algorithm that makes use of our workload characterization to make better decisions when prefetching.

To study spatial locality, we introduce a *chain* abstraction that represents a contiguous sequence of requests to the same file from the same client. Unlike prior characterizations of sequential access [54], a chain can include requests that were received out-of-order, on different parallel TCP connections used by the same client. We analyze the spatial locality of the server workload by examining characteristics of the chain lengths, such as the overall chain length distribution. Our simulations in Section 5.6 also employ the chain abstraction, as a higher-level workload representation than individual requests.

Our algorithm for finding chains of requests is simple in principle: find sequences of contiguous requests for content from the same file during the same session. The algorithm uses two passes. First, we iterate through each request in a session to determine if the end offset of the request is directly adjacent to the start offset of another request for the same file. To handle potentially out-of-order requests, we recognize adjacent requests regardless of the relative order in which they were received, as long as the adjacent requests are received within 40 seconds of each other. We chose this limit after analyzing the distribution of time gaps and finding that the longest commonly occurring gap due to pacing is 32 seconds, so a limit of 40 seconds encompasses pacing gaps while preserving gaps caused by client inactivity. For the second pass, the algorithm combines adjacent requests into the longest chains possible.

We applied the chain-formation algorithm to the Netflix workload and found about 2.3 million chains in the 65 million requests. Table 5.4 provides statistics about the chains. The "%" column specifies the percentage of the total number of chains of each type and *NTT/chain* specifies the average lengths of chains in seconds of nominal title time. The table shows that chains rarely consist of more than one open-ended request, in contrast to chains of chunk requests with 41 requests on average. The average sizes of video and combined chains are similar for both chunk and open-ended requests, indicating that the spatial locality of non-audio chains is similar regardless of the way HTTP requests are made.

Chain Type	%	Reqs /chain	MB /chain	NTT /chain
Open Audio	7.2	1.3	6.8	534.1
Open Video	17.3	1.1	15.1	63.7
Open Combined	7.0	1.0	25.5	148.5
Open Total	31.5	1.2	15.5	189.8
Chunk Audio	10.7	10.8	1.3	134.7
Chunk Video	54.6	43.0	15.1	51.6
Chunk Combined	3.2	109.5	25.5	147.6
Chunk Total	68.5	41.1	13.4	69.1
Grand Total	100.0	21.5	14.1	107.1

Table 5.4: Chain statistics

5.3.1 Lengths of Chains

We now characterize the lengths of the chains found. Figure 5.11 provides two different cumulative distributions of the length of chains, ordered by bytes, from shortest to longest. The curve labelled *Chains* shows the percentage of chains that are shorter than a given length, and the *Bytes Requested* curve shows the cumulative percentage of total bytes that are downloaded in chains. More than 60% of the chains are 10^6 bytes (1 MB) or shorter and only about 15% of chains are longer than 10^7 bytes (10 MB), so the majority of chains are relatively short. However, most of the content is downloaded in long chains. More than 90% of the total bytes are downloaded in chains longer than 10 MB and fewer than 2% are downloaded in chains shorter than 1 MB. These results suggest that despite servicing DASH clients that access many different bit rate files, most bytes will be requested in long chains with high spatial locality.

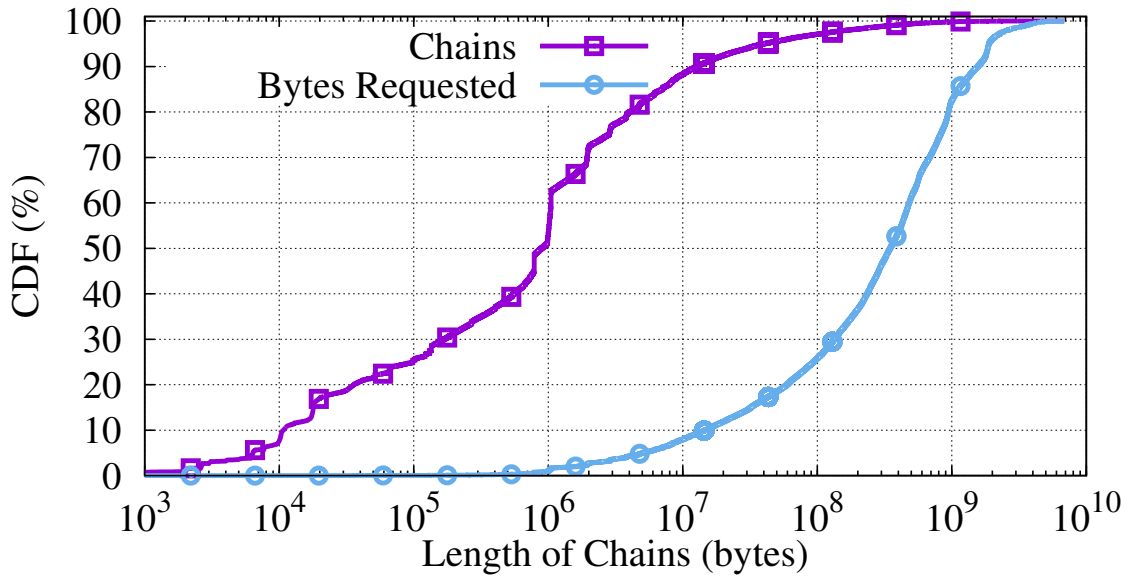


Figure 5.11: Percentage of chains ordered by chain size

5.3.2 Chains Starting at Offset Zero

Clients must download segment offset tables before requesting content from files so that playback can be started from any position in the title. This supports the implementation of DASH algorithms as well as user actions such as skipping backward and forward in the title. About

24% of chains start from an offset of zero, where the segment offsets are stored, so these chains are a significant subset of the workload.

Figure 5.12 shows a CDF of the lengths of all chains that start from a file offset of zero and therefore contain segment offset information, compared to the chains with non-zero offsets. We divide the chains that start at an offset of zero into two categories, depending on whether the chain consists of open-ended or chunk requests. For chains of open-ended requests that start at zero, 84% are exactly 768 KB in length (likely due to the size of socket buffers used on the server), and only 0.4% are shorter. For chains of chunk requests that start at an offset of zero, the majority of chains are very short; more than 49% are shorter than 16 KB and 98% are shorter than 128 KB. The remaining chains that start at an offset greater than zero, of either open-ended or chunk requests, are longer. Only 33% of chains starting from a non-zero offset are shorter than 768 KB in length. Chains that start at an offset of zero are easy to recognize and tend to be much shorter than chains with non-zero starting offsets. We evaluate two prefetch algorithms that make use of these properties in section 5.6.

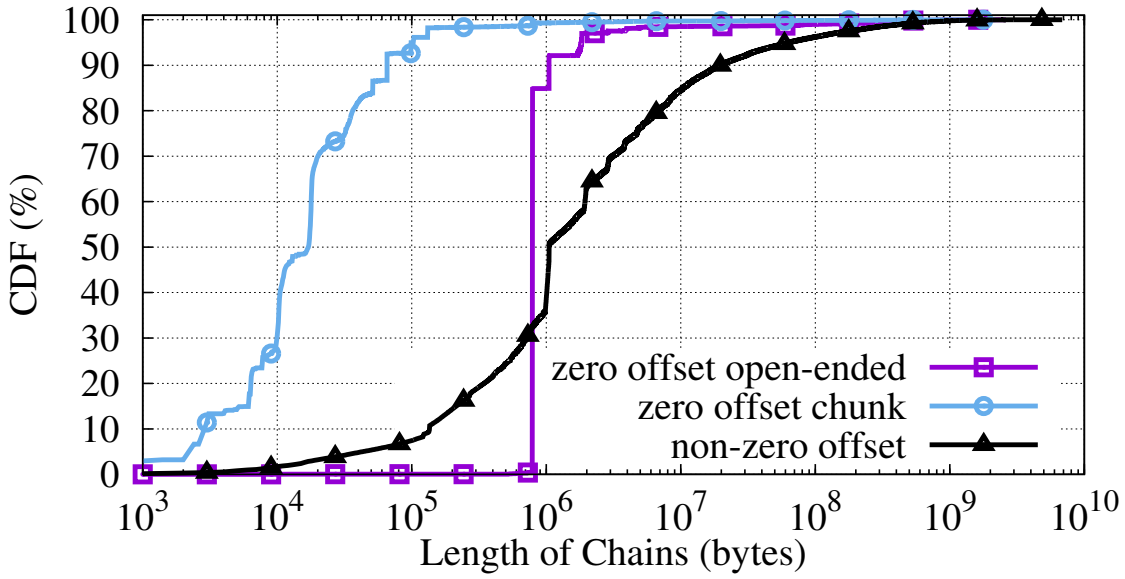


Figure 5.12: CDF of lengths of chains with different start offsets

5.3.3 Chain Survival Distances

We now analyze the chain length distribution to determine whether it can be described by simple equations. Figure 5.13 is a complementary cumulative distribution function (CCDF) that is generated from the measured chain lengths. This figure contains the same information as Figure 5.11, but shows the percentage of all chains that are longer than a given length as opposed to the percentage of chains that are shorter than a given length. For example, Figure 5.13 shows that about 10% of chains are longer than 20 MB ($2\text{x}10^7$ bytes) and about 1% are longer than 450 MB. We display the data using log scales on both axes in order to find potential power-law relationships in the data, which will appear as straight segments on the curve.

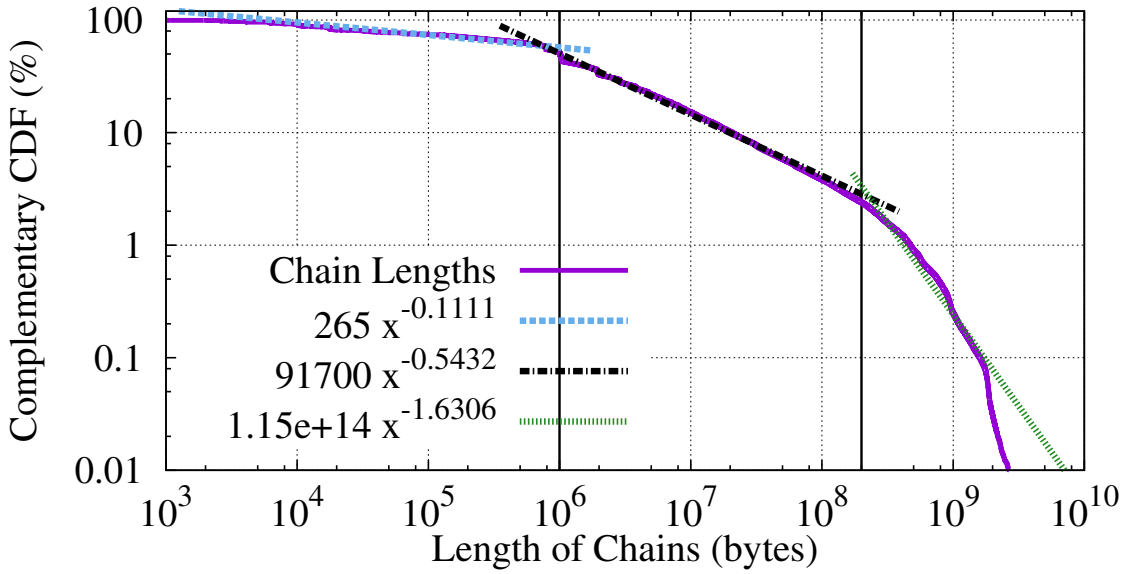


Figure 5.13: Cumulative number of longer chains

We observe that the curve appears very straight for chain lengths between 1 MB and 200 MB, so we divide the chain lengths into three segments: shorter than 1 MB, between 1 MB and 200 MB, and longer than 200 MB, and fit power-law equations to each of those segments. We compute the survival function $S(x)$ for a chain of length x in bytes from the set of all chain lengths X as follows:

$$S(x) = \text{Prob}(X > x) = Ax^{-c} \tag{5.1}$$

The three power-law equations are shown in Figure 5.13. For each equation, the values for A and c were calculated using a linear fit of the logarithms of the data values.

These equations can be used to compute a conditional probability for how long chains will survive. We would like to determine the probability that a chain that is longer than x will have total length at least $x+d$. We call d the *survival distance* for a given chain length and probability.

We compute an expected survival distance d, given a particular probability P that a chain of length at least x will have total length longer than $x + d$ as follows:

$$P = \frac{\text{Prob}(X > x + d)}{\text{Prob}(X > x)} = \frac{A(x + d)^{-c}}{Ax^{-c}} \tag{5.2}$$

which can be solved for d:

$$d = (P^{-1/c} - 1)x \tag{5.3}$$

To use this equation, we can select a probability target, for example, $P = 0.45$, and compute that $d = 3.35x$ for the range 1 MB to 200 MB. So for any chain that reaches a length between 1 MB and 60 MB (= 200 MB/ 3.35), there is a 45% chance the chain will grow to 3.35 times that length.

Having defined and characterized chains, we use them in the next section to understand how the DASH algorithms affect chain lengths. We also use chains in Section 5.6 both to implement a prefetch algorithm and to compare different prefetch algorithms for servicing the workload.

5.4 Phases

While examining traces of individual sessions, such as those in Figures 5.8 and 5.9, and others not included in this thesis, we found that clients seem to exhibit patterns of requests. For example, the bottom part of both Figures 5.8 and 5.9, show that each session starts by issuing requests for multiple files (each containing a different bit rate encoding). This pattern of issuing requests for multiple files for a short period of time occurs in many other sessions we have examined, in addition to the examples in Figure 5.8. We also noticed patterns where clients either access content sequentially from a single bit rate (i.e., a single file), or they do not access any files (i.e. playback is paused). Our goal in this section is to try to understand if such patterns are common across sessions and to understand the impact of these patterns on the sequentiality of requests.

Our characterization provides insights into client behaviour including the use of rate adaptation and helps explain the observed chain length distribution.

We first characterize phases by examining patterns of activity for chunk requests. We analyze chunk requests because their short duration provides fine-grained information about average download rates, compared to open-ended requests. We show how chain lengths can be used to recognize transient phases, then we show how the average download rate of chunk requests varies during the stable phase. Finally we show that the average transfer characteristics of chains of both open-ended and chunk requests are similar, so our findings specifically about chunk requests are applicable to both types of chains.

5.4.1 Request Patterns During Phases

In our model, we identify three different *phases*, where clients issue requests with characteristic patterns during each phase. In the following list, we informally describe the patterns of requests that identify each phase, as well as the actions of the client during the phase.

Transient: The client issues requests for a number of different bit rate files in a short period of time. For most clients this occurs at the start of a session, when there is a change in network or server bandwidth, or after the user changes to a different playback position. This pattern of requests for different files over a short period of time reflects the operation of the rate adaptation algorithm, when the client downloads segment offset tables and content from many different bit rate files.

Stable: The client retrieves content sequentially from the same file because the bit rate being used is stable. At some points in time (e.g., at the beginning of a session) the client retrieves content as quickly as possible. The client operates in this mode when it must fill its playout buffer after a transient phase. As a result, we call this mode of operation *filling mode* . Once the playout buffer contains enough data, requests are paced to arrive at the server so the average download rate is approximately equal to the bit rate of the file. We refer to this mode of operation as *pacing mode* . Clients use different mechanisms for pacing requests, depending on whether they are issuing open-ended or chunk requests.

Inactive: The client temporarily stops issuing requests for content from files of any bit rate. After this phase, the client usually enters the transient phase.

5.4.2 Phases at the Start of Sessions

In this section, we analyze non-audio chunk requests issued at the beginning of sessions, where we assume that clients start in a transient phase, followed by a stable phase. We validate this assumption in Section 5.4.3, after we develop a method for recognizing phases, by showing that 97% of sessions start in a transient phase.

Figure 5.14 shows average request characteristics calculated by aggregating all non-audio chunk requests in the workload. The values are generated in 1 second intervals, relative to the start of each session. We compute the four measurements for each second of each session, then calculate averages by totaling the measurements over each second and dividing by the number of sessions that have not yet ended during that second. The left axis shows the number of concurrent connections and files, and the right axis shows the arrival interval and request duration. The curves in Figure 5.14 are smoothed because there are few long sessions. We average data values in bins equal to 1% of the elapsed session time for all the graphs in this section.

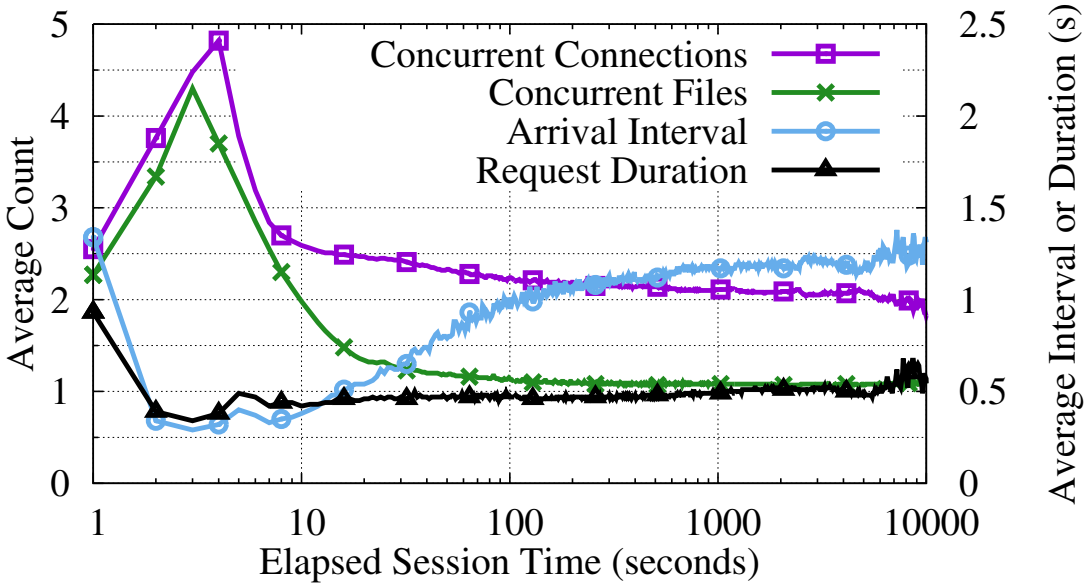


Figure 5.14: Request activity aggregated over all sessions

The *Concurrent Files* curve in Figure 5.14, which shows the number of unique files that were requested during each second of elapsed time, can be used to characterize the first transient phase, when a number of files with different bit rates are accessed in a short amount of time. The

number of concurrent files peaks after 3 seconds, then declines steadily; which indicates that the first transient phase is less than 20 seconds for most sessions.

From 1 second to 10 seconds the *Concurrent Connections* curve and *Concurrent Files* curve are quite similar. This indicates that, on average, a single connection is used to request files during this time. Afterwards, clients use an average of about two TCP connections to access each file.

The *Request Duration* curve is fairly level, indicating that the average request size remains constant regardless of phase. The *Arrival Interval* curve is calculated by subtracting the arrival time of the next request from the arrival time of the current request (regardless of which parallel TCP connections are used), and will be equal to 0 if the requests arrive during the same second. The average time between arrivals gradually increases during the period from 10 to 200 seconds because an increasing proportion of clients transition from the transient phase to the stable phase and then from filling mode (when requests are issued as quickly as possible) to pacing mode (when requests are issued at the same rate as content is consumed). After about 200 seconds, almost all clients are in pacing mode, so the average arrival interval remains nearly constant.

We have now characterized the pattern of chunk requests for the phases that occur at the start of sessions. In the following sections, we provide algorithms for recognizing phases whenever they occur during a session, which are also applicable to sessions with open-ended requests.

5.4.3 Transient Phases

During the transient phase, many different bit rate files are accessed in quick succession in order to download segment offset tables and video content from many different bit rate files, which results in a cluster of relatively short chains. During the stable phase, all content is requested sequentially from the same file which causes relatively long chains. The clusters of short requests that occur during transient phases have significantly different patterns depending on the client implementation and the action that triggered the transient phase. Figure 5.8 shows some examples of different patterns of requests for transient phases. Because of this wide variety of patterns, we use a robust and simple algorithm to recognize phases.

We define *short* chains as those with a duration of less than or equal to 40 seconds, and *long* chains have a duration longer than 40 seconds. We then find clusters of short chains that have less than a 10 second gap between the end of one short chain and the start of another. We define the discrete clusters of short chains as representing transient phases. The long chains identify stable phases. We determined the 40 second and 10 second threshold values experimentally. We searched for values that would result in roughly an equal number of short clusters and long chains, because we expect that a transient phase will typically be followed by a stable phase.

Figure 5.15 shows, for each second of elapsed time, the percentage of active sessions that are starting a cluster of short chains and the percentage of sessions that are starting a long chain. Over 97% of sessions start with a transient phase, and approximately equal numbers of transient phases and long chains start after 100 seconds of elapsed time. Using our chosen threshold values, the occurrence of transient phases and the other phases meets our expectations.

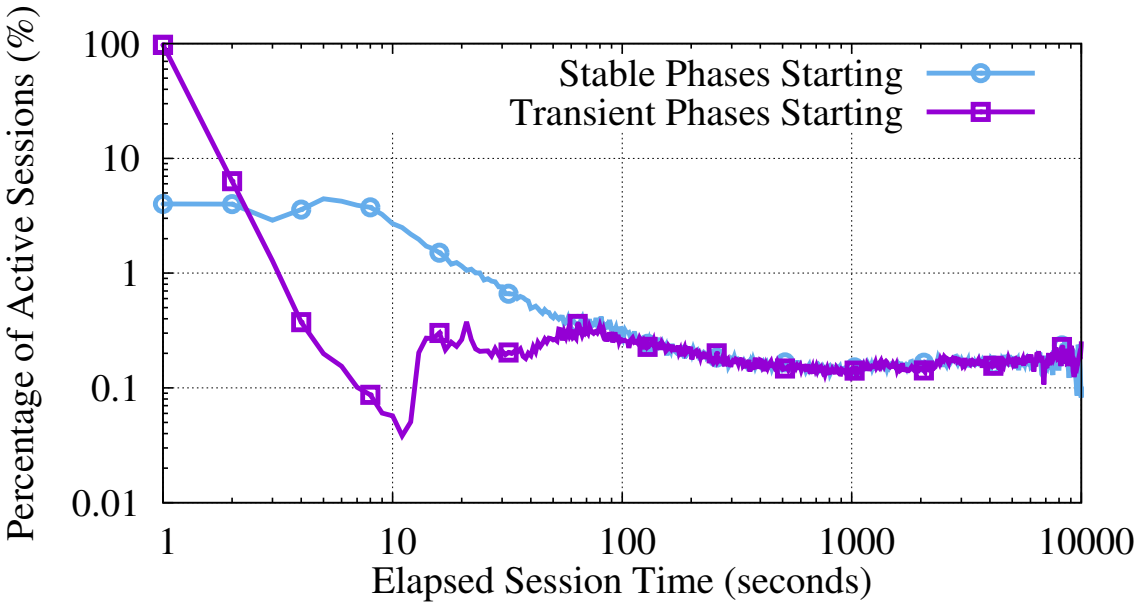


Figure 5.15: Start times of transient phases

Transient Phase Bit Rates: First Transient Phase

Having determined when transient phases occur, we now characterize which bit rates are accessed together during transient phases. We provide information about the average number of bit rates accessed at the start of sessions in Figure 5.14, but there is no information about which specific bit rates are accessed. This is information that would be useful when constructing a benchmark. During the transient phases, Netflix clients systematically request content at different bit rates in order to determine the highest video bit rate that can currently be supported by the network and server. Additionally, segment offset tables at the start of files must be read once before requesting any content, so the first transient phase usually contains more requests for more bit rates than later transient phases. After the transient phase, the most suitable bit rate is used to download the bulk of content during the stable phase that follows the transient phase.

For example, Figure 5.16 shows a detailed view of the bit rates accessed during the first 12 minutes of the example session shown in Figure 5.8. During the first 0.75 minutes, the client accesses 6 different video bit rates between 235 Kbps and 1,750 Kbps (downloading at 1,050 Kbps for 0.5 minutes) before choosing 1,750 Kbps for the first stable phase (i.e., the first chain with a duration longer than 40 seconds).

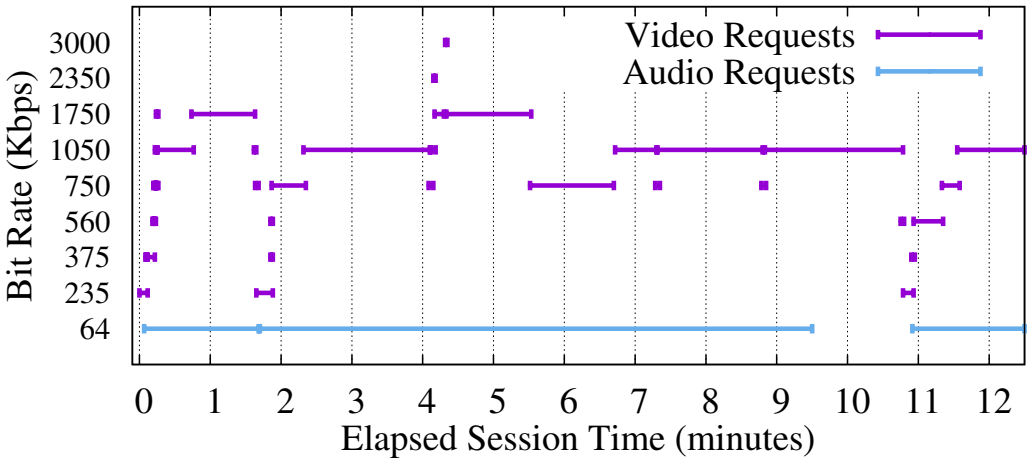


Figure 5.16: Detailed view of bit rates accessed at start of example session from Figure 5.8

We wish to examine the relationship, over all sessions, between the average amount downloaded at different bit rates during the transient phase and the bit rate chosen for the stable phase that follows the transient phase. We start by characterizing the bit rates accessed during the first transient phase, based on the bit rate used for the first stable phase. About 33% of sessions are shorter than 40 seconds and cannot have a stable phase by definition, 56% of sessions have a stable phase, and for the remaining 11%, either the session duration is too short or the network connection is too unstable for a stable phase to form. Figure 5.17 shows the bit rates accessed during the first transient phase averaged over all sessions. For each stable phase bit rate (given on the x-axis), the figure shows the average percentage of content (in terms of nominal title time) that is downloaded during the first transient phase at each different video or combined bit rate (given on the y-axis). In other words, each column is a histogram of the percentage of content downloaded at each bit rate (i.e. each row) during the transient phase that precedes the stable phase with the given bit rate. Percentages are calculated for each stable phase bit rate separately, so the percentage downloaded during the transient phase at all video and combined bit rates total 100% for each stable phase bit rate. Circles are used to show values between 5% and 100%, and the circle in the legend represents 10%. The area of each circle is proportional to the percentage

of the total content downloaded at that bit rate during the transient phase. The circles are too small below 5%, so crosses are used to show percentages between 0.5% and 5%. Percentages below 0.5% are considered to be negligible, so they are not shown.

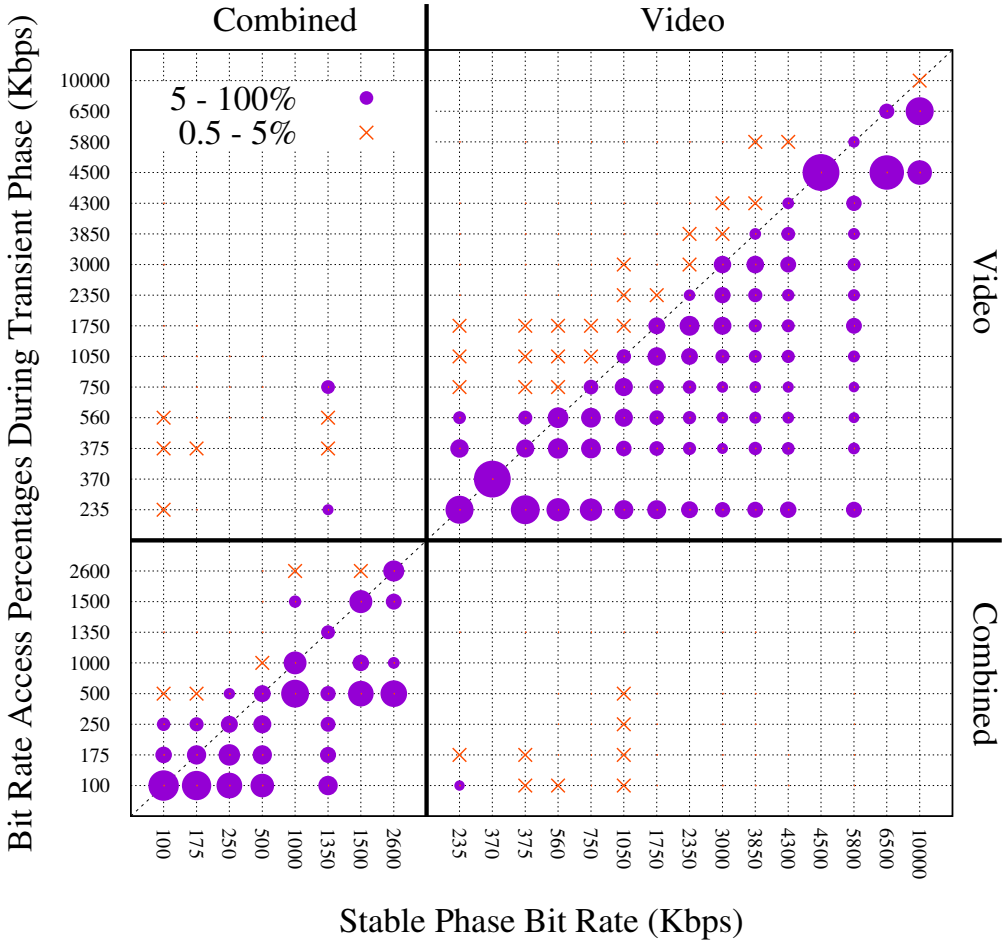


Figure 5.17: Associated bit rates: First transient phase for sessions with a stable phase. The x-axis specifies the bit rate used for the first stable phase. The y-axis specifies the bit rates accessed during the first transient phase, for each individual stable phase bit rate.

We provide a brief example of how this chart is read, using the example session shown in Figure 5.16. During the first transient phase of the example session, the client issues requests for video bit rates between 235 Kbps and 1,750 Kbps, then starts a stable phase at 1,750 Kbps. Comparing the example session of Figure 5.16 to the average of all sessions that have a stable phase

bit rate of 1,750 Kbps, the example session appears to be typical. In the 1750 column of Figure 5.17, only a small percentage of content is downloaded at video bit rates above 1,750 Kbps, and the percentage of content downloaded at bit rates between 235 Kbps and 1,750 Kbps is about the same (the circles are of similar size), which is similar to the pattern of bit rate accesses of the example session (except for the large amount of content downloaded at a bit rate of 1,050 Kbps).

We make some general observations about Figure 5.17. The pattern of examining a number of lower rates before settling on a stable phase bit rate largely holds for the other stable phase video bit rates. However, there are exceptions: 0.5% of sessions use a 370 Kbps bit rate and never access any other bit rate, and the group of sessions using stable phase bit rates of 4,500, 6,500 and 10,000 Kbps (0.05% of sessions) do not access bit rates lower than 4,500 Kbps. In these cases, the atypical patterns could be caused by differences in client implementations, or it is possible that the results are an artifact of the small sample size. We also observe that similar amounts are read at each bit rate during the first transient phase and that very little content is downloaded at rates higher than the stable phase bit rate.

Transient Phase Bit Rates: Successor Transient Phases

Many sessions have more than one transient phase. For example, in Figure 5.16, there are 5 transient phases that follow the first (called *successor* transient phases), at approximately 2,4,7,9 and 11 minutes. Some of the successor transient phases (those at 2 and 11 minutes) are similar to the first transient phase in that they request data for a large number of bit rates that are lower than the stable phase bit rate, while the other successor transient phases are different because the lowest bit rates are not accessed. To determine if there are generally differences between the first transient phase and successor transient phases, we analyze the 150,290 successor transient phases in the workload. We generated three charts, for the cases where the bit rate of the stable phase increases (29%), decreases (17%) or remains the same (54%) relative to the preceding stable phase bit rate. Figure 5.18 shows the chart for successor transient phases when the stable phase bit rate decreases. The other two charts are quite similar, so we do not show them. The patterns of bit rate accesses show that on average, compared to the first transient phase, there is a bias towards accessing bit rates slightly above and below the stable phase bit rate during successor transient phases, and a weaker tendency to access the lowest bit rates. This is possibly because clients have more knowledge of network conditions after the first transient phase and therefore have less need to probe the lowest bit rates, or fewer bit rates are accessed because segment headers are downloaded during the first transient phase and do not need to be downloaded again.

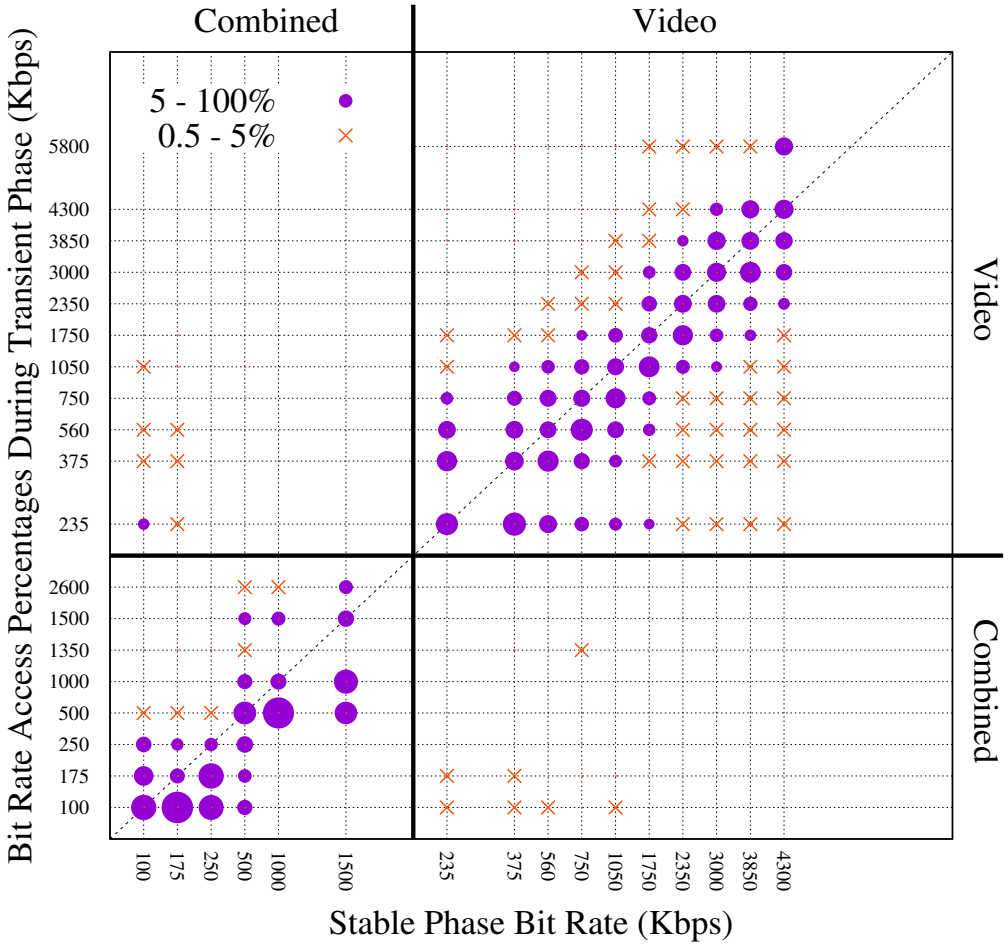


Figure 5.18: Associated bit rates: Successor transient phases. Includes only transient phases where the following stable phase bit rate is lower than preceding stable phase bit rate.

We have now explained how the length of chains (using a threshold of 40 seconds) can be used to identify phases. We have also characterized the timing of transient phases and the bit rates that are accessed during transient phases. We next investigate the timing of requests during the stable phase, which changes between the filling and pacing modes.

5.4.4 Stable Phases

Chains not belonging to a transient phase make up stable phases. That is, each chain longer than 40 seconds comprises a stable phase. The average duration of a stable phase is 8.5 minutes.

This provides an estimate of the average interval between events, which include skips, pauses, or ending the session by the user, as well as the operation of the rate adaptation algorithm by client devices.

We use two different methods to characterize changes in the download rate during a stable phase, which will enable us to identify the transition from filling mode to pacing mode. The first method is only valid for chains of chunk requests. We compute the average download rates to directly show the point where the chain transitions from high download rates to lower download rates. The second method characterizes download rates indirectly, but can be used for chains of both open-ended and chunk requests.

Download Rates

Figure 5.19 shows the average number of bytes downloaded in requests, categorized by the length of chain that contains the request, during each second of elapsed session time. The average download rate is the total number of bytes requested during each second divided by the number of sessions that are active during that second. The *Long Chains* curve has an early peak signifying the filling mode, followed by a decrease to a largely constant rate after 200 seconds. From this curve, it appears that almost all sessions are in pacing mode after 200 seconds, in accord with Figure 5.14. The apparent decrease in the download rate for long chains after 1,000 seconds is caused by an increasing number of sessions that are in inactive phases (as shown in Figure 5.21).

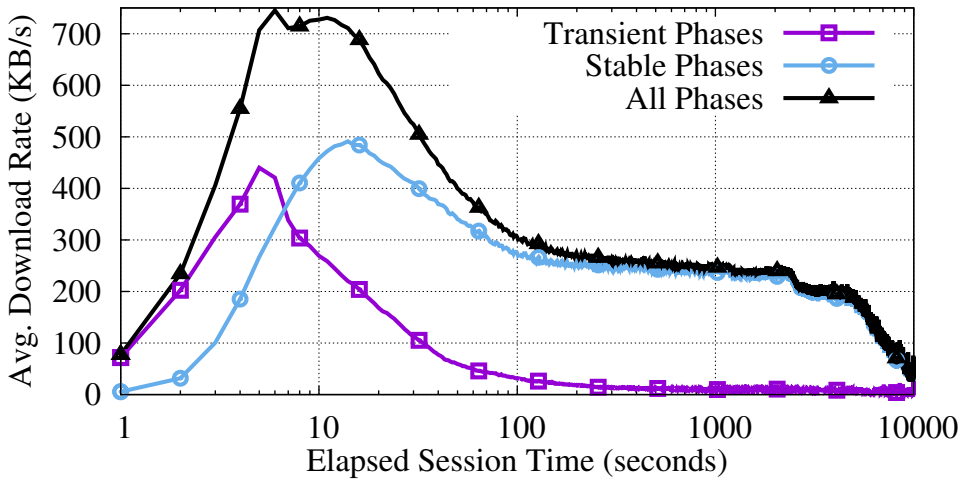


Figure 5.19: Aggregate download rate for requests

Figure 5.19 also shows a large variation in total download rates by elapsed session time. The average download rate of all requests is about 3 times higher at the start of the session (between 6 and 11 seconds) than it is after 200 seconds. This indicates that typical clients use much less bandwidth in pacing mode than is available during the first transient phase.

Transfer Ratios

For open-ended chains, we cannot directly observe changes to the transfer rate because they occur at the TCP level, via TCP flow control [62], which is not recorded in the server traces. However, we can use an indirect method to show that the transfer characteristics of chains of open-ended and chunk requests are similar, and therefore conclude that the characteristics of filling mode are the same for both types of requests. We use the property that the filling mode is limited in duration, so the longer the chain, the higher the proportion of time spent in the pacing mode. Since the ratio of content downloaded in a chain to the duration of the chain (the *transfer ratio*) is equal to 1 while in pacing mode and greater than 1 while in filling mode, we expect that the longer a chain, the closer the transfer ratio will be to 1.

Figure 5.20 shows the average transfer ratios for chains with the same elapsed time, calculated separately for chains of open-ended and chunk requests. The transfer ratio is much larger than 1.0 for short chains, particularly chains of open-ended requests, where the maximum ratio of 14.5 is for chains with 2 second duration. The transfer ratio declines quite gradually, indicating that the filling mode is very long for some sessions. The calculated ratios for chains of open-ended and chunk requests are nearly identical for durations longer than 12 seconds. This is strong evidence that, although we cannot directly measure them, the patterns of changes of transfer rates for open-ended requests are similar to the patterns shown in Figure 5.19 for long chains of chunk requests.

5.4.5 Inactive Phases

To detect inactive phases, we simply find sufficiently long periods of time when no requests are issued. Figure 5.21 shows the percentage of sessions (that have not yet ended) that are in an inactive phase during each second of elapsed time. A session is in an inactive phase if it issues no requests for any file for a period of at least 40 seconds. We chose the threshold value of 40 seconds to match the 40 second threshold for inter-arrival gaps for chains (as described in Section 5.3). Inactive phases are common; at least 10% of sessions are in an inactive phase after

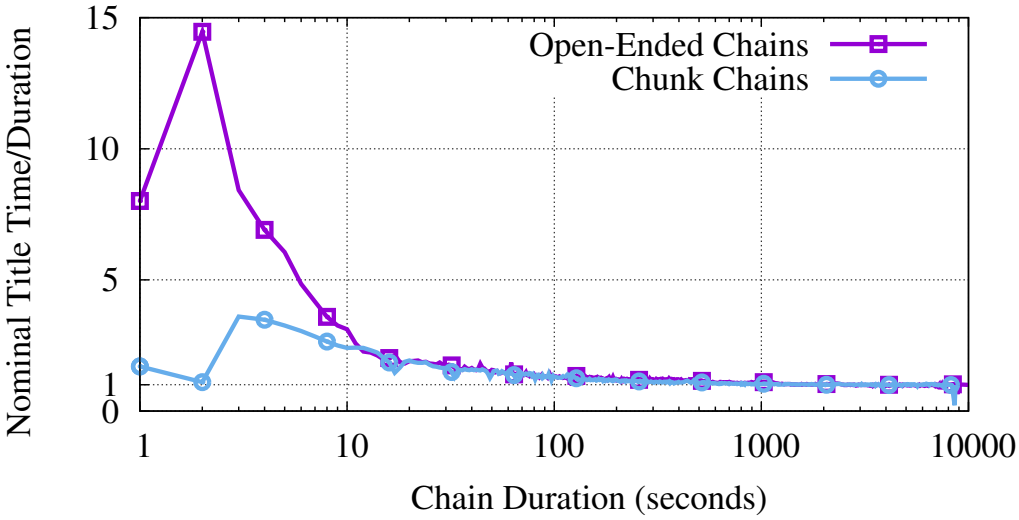


Figure 5.20: Ratio of play time to chain duration

the first few seconds, and the percentage increases rapidly after 5,000 seconds. This indicates that most long sessions are caused by inactive phases.

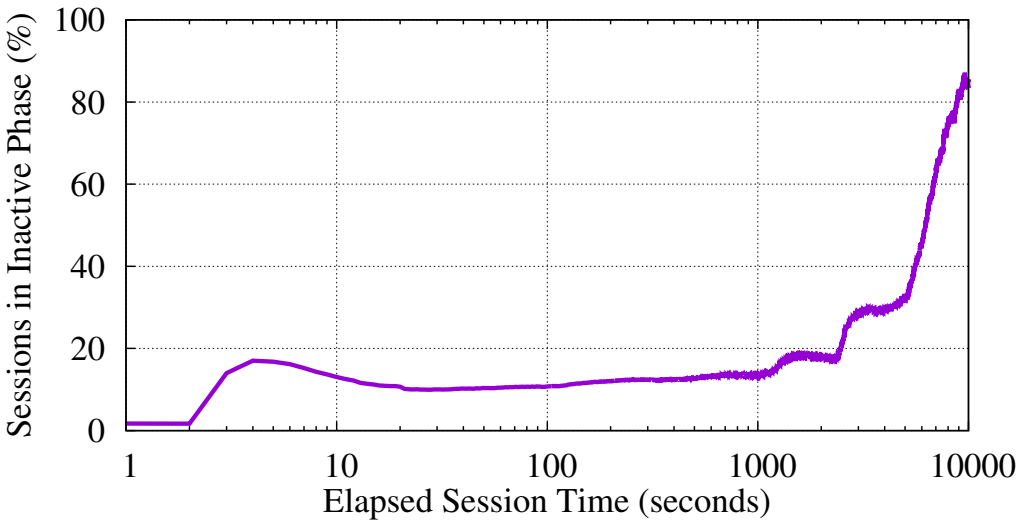


Figure 5.21: Percentage of sessions in inactive phases at different times

5.4.6 Impact on Sequentiality

Now that we have defined and examined the different phases, we consider the proportion of time spent in each phase for this workload. Across all sessions, transient phases account for 5.2% of the time, stable phases 79.1%, and inactive phases 16.4%. These numbers add up to 100.7% because there is a small amount of overlap between the transient and stable phases for some clients. With respect to bytes transferred, 7.6% of the total number of bytes are downloaded in transient phase chains, and 92.4% in stable phase chains. The proportion of bytes downloaded in the transient phase is higher than the proportion of time because clients do not use pacing during the transient phase.

Understanding these phases helps to explain the distribution of chain lengths in Figure 5.11. Many different bit rate files are accessed during transient phases, but not much content is read from each file, resulting in a large number of short chains. Clients spend a large proportion of time in stable phases, and the stable phases last a long time (8.5 minutes on average), so there are relatively few long chains that account for a large proportion of the total bytes requested. We make use of these chain length characteristics in Section 5.6, when we analyze potential prefetch algorithms that make use of these characteristics.

5.5 Creating a Workload Specification

In order to demonstrate the utility and completeness of our workload characterization, we consider the problem of creating a workload specification using the information we have presented. As described in Chapter 3, the workload specification could be used to construct a Netflix-like benchmark in the future.

Table 5.5 lists the major workload parameters for the Netflix workload, which are represented by the figures listed in the *source* column. This table has a similar format as Table 3.2, but because we do not know the capacity of the target server and experiment environment, we omit those hardware-dependent parameters.

Because the sessions and request patterns for the Netflix workload are more complicated than the YouTube-like benchmark, future work would involve modifying the workload generator and `httperf` to accommodate the differences.

We have much more information about sessions and requests in the Netflix workload than we did for the YouTube-like workload specification. Therefore, the workload constructor would need to be extended to implement the following functions:

Parameter Description	Source
Video Popularity Distribution	Figure 5.4
Video Duration Distribution	Figure 5.3
Video Bit Rates	Figure 5.2
Session Length Distribution	Figure 5.5 (and 5.7)
Session Arrivals	Figure 5.1
Client Request Size (MB)	VBR, Figure 5.10
Client Request Pacing	Figure 5.19
Client Adaptation	Figure 5.15 (and 5.17)
Server Storage Method	Single File
Server Chunk Size (Time)	2 second segments

Table 5.5: Summary of Netflix storage server workload specification

- Generate a set of sessions that are representative of the Netflix workload. Tasks include choosing a title, a starting point in the title, the duration of the session, the time at which bit rate adaptations occur, and the timing of any user events such as skips and pauses. We characterize these parameters in Sections 5.2.1, 5.2.2 and 5.4.4.
- Generate phases based on the session events, then derive chains to represent the phases. We characterize these factors in Section 5.4.
- Convert the chains into individual requests using the bit rate, duration and title position of the chain. We must choose an HTTP format for the requests: open-ended or chunk, and whether or not to use parallel connections, as detailed in Section 5.2.4.

To be able to generate open-ended chains that are paced, it may be necessary to make further modifications to `httperf`, in addition to those described in Section 3.4. However, since the transfer characteristics for chains of open-ended requests and chunk requests are similar, it might be sufficiently accurate to use chunk requests for all chains.

After these changes, it would be possible to test web server implementations using a Netflix-like benchmark, including the simulation of rate adaptation. A Netflix-like benchmark could also be adjusted to examine future changes in workload characteristics (e.g., the use of HTTPS).

5.6 Evaluation of Workload-specific Prefetch Algorithms

In this section, we demonstrate the utility of our workload characterization by using it to develop workload-specific prefetch algorithms. We use a simple simulation model of a web server to carry out a first-cut performance evaluation of the algorithms, in comparison to a baseline prefetch algorithm that makes no use of workload characteristics. In the future, the insights we gain can be used as a starting point for modified server implementations, which can then be evaluated experimentally.

5.6.1 Prefetch Algorithms

We describe five algorithms for choosing a prefetch size: the baseline algorithm that is used for most of the experiments in Chapter 4, two algorithms that make use of the bit rate of files as discussed in Section 4.3, and two algorithms that make use of the characteristics of chains in the Netflix workload.

Fixed: The baseline algorithm uses a single fixed prefetch size and it requires no workload information.

Proportional: This algorithm uses a fixed prefetch size that is proportional to the bit rate of the file being accessed.

Square-Root: This algorithm uses a fixed prefetch size that is proportional to the square root of the bit rate.

First-Grow: This algorithm makes use of workload chain characteristics in two ways. The algorithm uses one of three specifically-determined prefetch sizes for the *first* prefetch in a chain, depending on the type of chain and its starting offset in the file. Second, it *grows* the size of subsequent prefetches by a multiplier, based on the power-law relationship we derived from the chain lengths in Section 5.3.1, until the prefetch size reaches a maximum.

The *First-Grow* algorithm uses a relatively small prefetch size at the start of chains because the majority of chains are short. Using a relatively small first prefetch size will reduce the amount of content that is prefetched but not subsequently requested by clients for short chains. Additionally, chains that start from an offset of zero (where the segment offset table is stored) are often very short, as shown in Figure 5.12. We divide chains into three categories: chains of open-ended requests that start at zero, chains of chunk requests that start at zero, and the remaining

chains. To take advantage of the different chain length distributions, we perform a separate cost-benefit analysis for each category and determine the best first prefetch sizes. For this analysis, the benefit is the amount of data that is requested by clients at the start of each chain, up to the prefetch size. The cost has two components: prefetching data that will not be requested because the chain is shorter than the prefetch size, and performing seeks for the chains that are longer than the prefetch size. The prefetch sizes that provide the lowest cost-benefit ratios are listed in Table 5.6.

Offset of Chain Start	Request Type	Size (bytes)
Starts at zero	Open-Ended	1,049,212
Starts at zero	Chunk	131,073
Starts later than zero	Either Type	2,936,037

Table 5.6: Size for first prefetch in a chain

The *First-Grow* algorithm also gradually increases the prefetch size to take advantage of the characteristics of longer chains derived in Section 5.3.3. Specifically, we multiply the current chain length by 3.35 to compute a prefetch size, which is calculated to provide a 45% survival rate, based on Equation 5.3. We tried a range of different survival rates and determined that 45% provided the lowest disk utilization for the workload.

First-Grow-SR: This algorithm is the same as *First-Grow*, except the prefetch size grows to a maximum that is proportional to the square root of the file bit rate.

5.6.2 Evaluation Methodology

We apply a trace-based simulation, using the chains that are computed from the Netflix storage server log, to compare the different prefetch algorithms. For our analysis, we track the usage of two important resources: the hard drives that store content, and the system memory that holds prefetched content. We determine resource usage by applying a prefetch algorithm to each chain independently. Given the length of a chain in bytes, as well as the start and end times of the chain duration, we simulate the timing and size of the prefetch operations that would be required to service each chain. We maintain a global record, divided into 60 second intervals over the elapsed time of the logs, that aggregates resource usage from individual chains to determine total utilization over time. After processing all chains, we find the maximum system memory consumption and the maximum utilization of a hard drive that occurs during an interval.

To determine the consumption of system memory, we calculate the amount of prefetched data and the time that it is resident in memory. Notionally, when a chain starts, a memory buffer equal to the first prefetch size is allocated, which is reused and potentially resized for any subsequent prefetches. The prefetch buffer is freed 40 seconds after the chain ends. This 40 second interval matches our criteria for forming chains, so this delay in deallocating a prefetch buffer represents the actions of a memory management algorithm that keeps prefetched data in memory until a chain is known to end. We make the simplifying assumptions that prefetched data will not be evicted prematurely, and also that there is sufficient system memory for prefetch buffers, to avoid simulating a memory management algorithm.

To track hard drive utilization, we consider two separate operations: transferring data from disk and repositioning the disk head between prefetch operations. We calculate hard drive utilization as a fraction of maximum capacity. Given the maximum transfer rate t_{max} and the maximum seek rate s_{max}, we compute transfer utilization as t/t_{max} and seek utilization as s/s_{max}. We add these utilization fractions together (since a hard drive cannot seek and transfer data at the same time) to determine total disk utilization. If the calculated utilization of a hard drive is more than 100%, then the prefetch algorithm is infeasible because it would overload a hard drive.

To obtain values for s_{max} and t_{max}, we use benchmark results for a Hitachi Deskstar 5K3000, which has been used by Netflix in the past [69]. We found measured transfer rates for two benchmarks: 125.5 MB/s for 2 MB sequential transfer reads and 48.4 MB/s for 2 MB random transfer reads [88]. From the sequential benchmark, where no seeks are required, it follows that $t_{max} = 125.5$ MB/s. From the random benchmark, $t = 48.4$ and $s = 24.2$, since there is 1 seek per 2 MB read, and we calculate s_{max} as follows:

$$\frac{t}{t_{max}} + \frac{s}{s_{max}} = 1 \quad \Longrightarrow \quad s_{max} = s\frac{t_{max}}{t_{max} - t} \tag{5.4}$$

So $s_{max} = 24.2 \cdot 125.5/(125.5 - 48.4) = 39.4$.

5.6.3 Evaluation Results

We first evaluate the algorithms that make use of information about the bit rate of the content being requested, in comparison to the baseline algorithm which uses a single fixed prefetch size. Our earlier mathematical analysis of the best way to service mixed bit rate workloads in Section 4.3 indicates that we should choose a prefetch size that is proportional to the square root of the bit rate of the content. We confirmed the analysis experimentally using the YouTube-like

benchmark in Section 4.5.4, and we now repeat the evaluation analytically, using the *Square-Root* and *Proportional* algorithms, to determine if using prefetch sizes proportional to the square root of the bit rate is also the best option for the Netflix workload.

Figure 5.22 shows the usage of two different resources: the utilization of the hard drive with the highest load (labelled *Util* and using the left axis), and the maximum amount of system memory consumed (labelled *Mem* and using the right axis). Each data point is the result of analyzing one of the prefetch algorithms using a prefetch size parameter shown on the x-axis. The parameters have different meanings for each algorithm. For the *Fixed* algorithm, the parameter is the single size used for prefetching. For the *Proportional* and *Square-Root* algorithms, the x-axis shows the prefetch size used for a 2,000 Kbps bit rate and the other prefetch sizes are scaled proportionally to the bit rate, or proportionally to the square root of the bit rate, respectively.

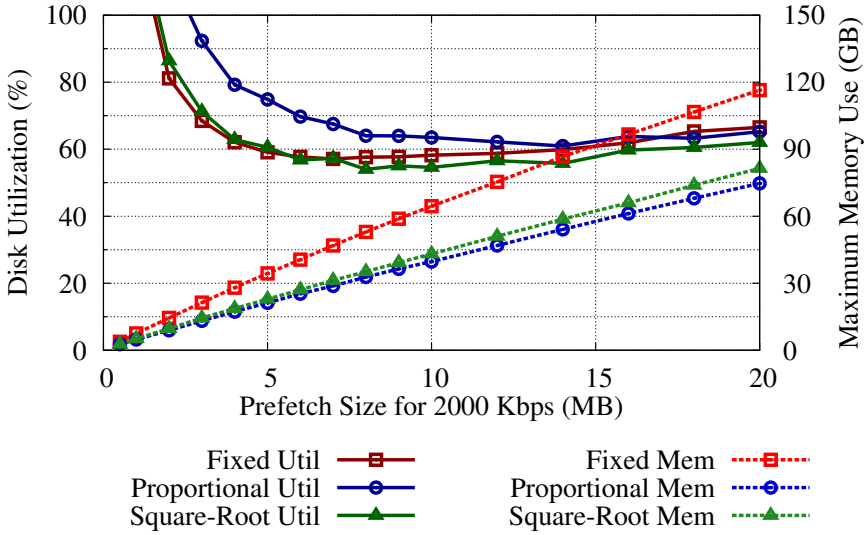


Figure 5.22: Comparison of bit-rate-based algorithms

The baseline *Fixed* algorithm provides its best hard drive utilization of 57% using a prefetch size of 7 MB, with memory consumption of 47 GB. Compared to the baseline, memory consumption is lower for both *Proportional* and *Square-Root*, using 29 GB and 31 GB respectively for the same nominal 7 MB prefetch size. However, the best hard drive utilization is higher than the baseline for *Proportional* while it is 5% lower than the baseline for *Square-Root*, with utilization of 54% when using a nominal 8 MB prefetch size. These results are similar to our experiments with the YouTube-like benchmark, shown in Figure 4.10, where the square-root rule provides the highest throughput.

We now consider algorithms that make use of chain characteristics. Figure 5.23 shows the results when we compare the *First-Grow* and *First-Grow-SR* algorithms to the baseline *Fixed* algorithm. The x-axis parameter specifies: the single prefetch size for *Fixed*, the maximum prefetch size for *First-Grow* algorithm, and for *First-Grow-SR*, the maximum prefetch size used for the 2,000 Kbps bit rate with the other maximum prefetch sizes scaled proportionally to the square root of the bit rate. The *Fixed* curve is provided for reference and also appears in Figure 5.22.

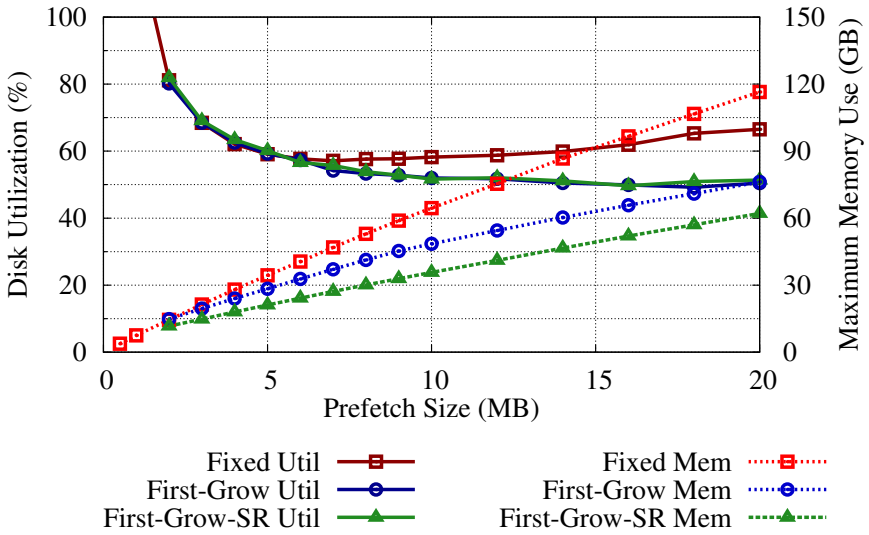


Figure 5.23: Comparison of chain-length-based algorithms

Compared to the baseline *Fixed* algorithm, the *First-Grow* algorithm reduces hard drive utilization and consumes less system memory because the initial prefetch sizes used by the *First-Grow* algorithm are small compared to the final prefetch size, which is equal to the single prefetch size always used by the *Fixed* algorithm. Using smaller initial prefetch sizes also reduces wasted prefetches for short chains, which is the reason that hard drive utilization is reduced. The *First-Grow-SR* algorithm has nearly identical hard drive utilization as *First-Grow* while reducing memory consumption further, because using a maximum prefetch size that is proportional to the square root of the bit rate makes the most efficient use of limited system memory (as shown in Section 4.3).

In summary, the best choice of prefetch algorithm depends on the extra workload information that is available to the server. If there is no workload-specific information available, we can use the baseline *Fixed* algorithm with a prefetch size of 7 MB, which results in disk utilization of

57% and memory consumption of 47 GB. If there is workload information available, then we can reduce disk utilization or memory consumption or both by using one of the following algorithms:

Square-Root: If only bit rate information is available, *Square-Root* reduces memory consumption by 42% while matching the 57% hard drive utilization of the *Fixed* algorithm.

First-Grow: If only chain information is available, *First-Grow* reduces hard drive utilization by 5% with the same 47 GB memory consumption as *Fixed*.

First-Grow-SR: If both bit rate and chain information is available, *First-Grow-SR* reduces memory consumption by 48% while matching the 57% hard drive utilization of *Fixed*. Alternatively, hard drive utilization can be lowered by 10% when matching the 47 GB memory consumption of *Fixed*.

The lowest hard drive utilization of all the algorithms we evaluated is 50% for *First-Grow-SR* with a prefetch parameter of 16 MB, which is a 13% reduction compared to the disk utilization of *Fixed* with a prefetch size of 7 MB. The *First-Grow-SR* algorithm consumes 52 GB of system memory to achieve 50% disk utilization, which is 11% more memory than *Fixed*. These results show that the efficiency of servicing the Netflix workload, either in terms of disk utilization or memory consumption, can be improved by making use of workload characteristics.

5.7 Chapter Summary

In this chapter, we analyze anonymized web server log files to characterize the workloads of Netflix streaming video servers. Because of the complexity and variety of methods that Netflix clients use to request content, we introduce chains to represent sequential requests for content from the same file, regardless of the different forms that requests may take. We then utilize chains to help identify transient, stable and inactive phases in the client implementations. We observe that playback sessions are quite stable. On average, the first transient phase lasts for only 1% of the session duration, there are only four transient phases per session, and they account for only 4% of total session duration (equivalent to 1 minute for a 22 minute title or 2 minutes of a 44 minute title). We find that despite there being large numbers of short chains due to activities such as rate adaptation (60% are shorter than 1 MB), the vast majority of content is downloaded in long chains (95% of content is downloaded in chains larger than 10 MB).

We also use the chains to analyze potential prefetch sizing algorithms for use with the Netflix workload. We analyze a simple algorithm that uses a single fixed prefetch size and requires no workload information as a baseline. We then test algorithms that make use of the bit rate of

the content to select a prefetch size and confirm that setting the prefetch size proportional to the square root of the bit rate results in lower resource utilization than setting the prefetch size proportional to the bit rate, confirming the mixed bit rate analysis in Section 4.3. We also show that utilization can be further reduced by making use of specific chain characteristics, such as the starting offset of chains and chain survival distances. to implement prefetch algorithms.

We find that despite the differences between the Netflix workload and YouTube-like benchmark workload, particularly for session characteristics and the use of DASH, both HTTP streaming video workloads can be more efficiently serviced by aggressively prefetching content that is expected to be requested by clients.

Chapter 6

Conclusions and Future Work

We now conclude our investigation of HTTP streaming video workloads as well as our efforts to improve the implementation of web servers to deliver this workload. In this chapter, we first briefly summarize our research and highlight our contributions. Then we discuss directions for future research. Finally, we present concluding remarks and discuss the potential impact of our work.

6.1 Summary and Contributions

The goal of our research is to understand HTTP streaming video workloads and to potentially use that knowledge to increase the throughput of video web servers. In this thesis, we investigate two HTTP streaming video workloads from YouTube and Netflix. Between them, these two services account for more than half of peak fixed-line Internet traffic in North America [83]. The catalogs offered by these services are huge (for example, the Netflix catalog is more than 2 PB in size), and user interest in titles has a long-tail distribution. For these reasons, relatively cheap high-capacity hard drives are typically used to store the long tail of the catalog that is viewed infrequently. Web servers should be able to achieve high throughput when servicing these streaming video workloads using hard drives because the requests issued by the clients are highly sequential, and hard drives provide their highest throughput when accessed sequentially.

Given our understanding of HTTP streaming workloads and the performance characteristics of hard drives, it is surprising that our experiments reveal that popular web servers have low aggregate throughput when servicing streaming video workloads. As we discover, this is because hundreds or thousands of streaming video clients issue relatively small requests to a web

server concurrently, and as a result, the requests issued from clients are interleaved and appear as though they are random reads to the server. Therefore, our approach to improving web server implementations is to devise algorithms that recognize and exploit the high spatial locality of HTTP streaming video workloads. That is, to modify the web server so that it transforms the random workload it receives into a more sequential workload when reading content from a hard drive.

6.1.1 Workload Methodology and YouTube-like Benchmark

To test web server implementations, we require a benchmark to generate representative HTTP streaming video workloads, but prior to our work in this thesis, such benchmarks did not exist. To remedy this situation, in Chapter 3, we introduce a methodology for creating benchmarks from workload specifications. However, all the workload characterizations available at the time that the methodology was created describe a workload that consists of requests issued to multiple servers, and not the workload of an individual server, which is the information we need to specify the workload for our methodology. Therefore, we combine the information from several studies that characterize the workload of the YouTube service to infer the workload specification for an individual server. Then, using that workload specification, we construct a YouTube-like benchmark that is used to conduct experiments on web servers.

In Chapter 4, we conduct carefully-designed experiments to reveal that two widely used web servers (`Apache` and `nginx`) and a high performance research web server (the `userver`) have throughput that is much lower than expected from the rated sequential throughput of the hard drive used in the server. We improve the throughput of the `userver` by implementing aggressive prefetching. This is not a simple task because the kernel interleaves concurrent disk I/O and thereby limits the size of requests issued to the hard drive. Therefore, we devise the ASAP architecture (Asynchronous Serialized Aggressive Prefetching) for web servers, which uses a single helper thread per disk to perform prefetches, thereby serializing access to each disk and preventing the kernel from interleaving I/O to each disk. We find that implementing the ASAP architecture on its own is not sufficient to improve throughput, we find it is also necessary to ensure that title content is stored in a single file (as demonstrated in Section 3.7.2), rather than the alternative of dividing content into chunks that are stored in multiple files. After modifying the `userver` to implement ASAP, and storing each title as a single file, we find that using a prefetch size of 2 MB more than doubles throughput compared to an unmodified "vanilla" `userver`, `Apache`, and `nginx` web servers.

6.1.2 Determining Prefetch Sizes

A prefetch size of 2 MB increases throughput, but it is unclear if 2 MB is the best prefetch size. In Chapter 4, we study the problem of choosing the best prefetch size. We conduct a systematic and time-consuming series of experiments, using a series of different prefetch sizes, to determine the highest failure-free throughput that is possible for each prefetch size. We show that the choice of the best prefetch size depends on: 1) the amount of system memory, 2) the popularity distribution of the titles requested, and 3) the characteristics of the hard drive.

Because the best prefetch size depends on so many different factors, that can potentially change, we devise an algorithm that dynamically and automatically adjusts the prefetch size. Our algorithm monitors both cache hits and the time required to service disk transactions in order to balance between maximizing the throughput of disk I/O and ensuring that the prefetched data is not evicted before it is used. The automated algorithm is able to choose prefetch sizes that are equivalent to the best found using our manual procedure. By combining the automated algorithm and the ASAP architecture, it is possible to improve `userver` throughput by up to 5.2 times compared to the vanilla `userver` (see Section 4.5.3), without requiring changes to the kernel and without the need for system administrators to perform tedious and time-consuming manual tuning.

Prefetch Sizes for Mixed Bit Rate Streams

The automated prefetch algorithm uses a single prefetch size (which is changed dynamically) to service client requests. This is a reasonable strategy when the same bit rate is used for all content, but for many streaming video services, content is viewed and requested at multiple different bit rates because viewers use different devices and access video services over a variety of networks. Using a single prefetch size results in more frequent prefetching for high bit rate content than for low bit rate content, which is not necessarily optimal. We used Lagrange analysis to show that for a workload consisting of finite streams with a variety of bit rates, when there is a limited amount of system memory available for storing prefetched data, then the prefetch sizes should be proportional to the square root of the title bit rate. To verify this analysis, we conduct experiments to compare our method to other alternatives, including using the same prefetch size regardless of bit rate, and using prefetch sizes that are proportional to the video bit rate (suggested by Gill and Bathen [37]). We find that our algorithm of using prefetch sizes that are proportional to the square root of the bit rate obtains the highest throughput.

6.1.3 Characterize Netflix Server Workload

Our workload characterization of an individual Netflix server is, to our knowledge, the first characterization of an individual web server used to service an HTTP streaming video workload. We study this workload because Netflix accounts for a large fraction of total Internet traffic and because the Netflix workload has significantly different characteristics than YouTube. In particular, information about rate adaptation was not available at the time we created the YouTube specification and benchmark. We are interested in whether aggressive prefetching will increase the capacity of Netflix servers, and whether a more detailed understanding of the specific characteristics of the Netflix workload could be used to improve prefetching.

In Chapter 5, we analyze the server log files obtained from two production Netflix servers to characterize the Netflix workload, including the characteristics needed to create a workload specification that could be used in the future to construct a Netflix-like benchmark. To analyze the wide variety of different requests made by the many types of client devices supported by Netflix, we introduce two abstractions: *chains* of sequential requests, and *phases* of client behaviour,

The chain abstraction is helpful for analyzing the spatial locality of the workload, and we find two specific characteristics of chains we can use to predict the length of chains: 1) chains that start at a file offset of 0 tend to be short, and 2) there is a power-law relationship between a given chain length and the percentage of chains in the workload that are longer than that given length. We propose prefetch algorithms that exploit these characteristics.

Algorithms for Choosing Prefetch Sizes Based on Workload Characteristics

We analyze five different prefetch algorithms that make use of workload characteristics such as the bit rate of the content being requested and the offset of the first request in a chain. We compare these algorithms by simulating the servicing of all the chains in the workload to compute the utilization percentage of the hard drives and the consumption of system memory. We show that prefetch algorithms that make use of our findings about servicing mixed bit rate workloads and workload-specific chain characteristics have lower disk utilization and lower memory consumption than an algorithm that has no information about the workload and uses a fixed prefetch size.

6.2 Future Work

We now discuss avenues for future research. In the following sections, we describe the additional effort required to experimentally evaluate service algorithms for the Netflix workload. We also

discuss potential ideas for improving the implementation of web servers for HTTP streaming video workloads.

6.2.1 Constructing a New Benchmark

In Section 5.5, we describe the necessary steps to create a Netflix-like workload specification, to then be used to create a Netflix-like benchmark. Our experiments with the YouTube-like benchmark were helpful for discovering important implementation issues that we had to address before our prefetch algorithm would work as expected. The patterns of requests in the Netflix workload are considerably more complicated than those used for the YouTube-like benchmark. It is possible that characteristics of those requests, such as the difference between open-ended requests and chunk requests, the use of parallel TCP connections, and the potential for out-of-order requests will need to be accommodated in web server implementations. One consideration for conducting experiments with a Netflix-like benchmark is that Netflix servers contain many hard drives, so the equipment used to simulate client demand will have to be scaled-up to utilize the capacity of multiple hard drives.

6.2.2 Testing Prefetch Algorithms with New Benchmark

After a Netflix-like benchmark has been created, it would be possible to implement and evaluate the workload-specific prefetch algorithms, such as *square-root* and *first-grow*, that are described in Section 5.6.1. Either the `userver` or another web server such as `nginx` could be modified to implement the alternative algorithms. One reason for implementing and testing the prefetch algorithms is to validate the chain-based simulation we use to evaluate algorithms in Section 5.6.3. The simulations take much less time to run than experiments, and if the results from experiments are sufficiently similar to the results from the simulation, it will be possible to rapidly evaluate many alternate prefetch algorithms and variations in server hardware (e.g., different types of storage devices) using simulation.

Additionally, the ASAP architecture serializes access to storage devices and therefore bypasses the disk I/O scheduling that is normally performed by the kernel. The current implementation of ASAP implements a FCFS (first come, first serve) scheduling algorithm for prefetches, and it may be possible to decrease hard drive seek times by reordering prefetch requests at the application level.

6.2.3 Multiple disks

Because of a lack of published information about how title content is distributed over multiple hard drives in large-scale production web servers, for our experiments, we evaluated a web server with only a single hard drive. However, the catalogs offered by streaming video services are often very large, making it necessary to configure streaming video servers with the maximum amount of storage capacity possible. Servers can also contain different types of storage devices. For example, Netflix storage servers contain 36 hard drives and 6 SSDs. In order to make the best use of a server with multiple, possibly heterogeneous hard drives and SSDs, we would investigate the following issues:

Provisioning Given the anticipated workload on the server, we need to determine the number and types of storage devices that are necessary to store the titles. Servers must be provisioned with enough storage devices that aggregate throughput is high enough to satisfy the maximum aggregate demand from clients (which is limited to the total Internet bandwidth of the server). Servers must also be provisioned with enough storage capacity. Hard drives have the lowest cost per TB, but may not provide sufficiently high throughput. SSDs have higher throughput but are more expensive, and the new NVMe (Non-Volatile Memory express) interface enables even higher throughput from flash devices, at an added cost. Finally, servers must be provisioned with enough system memory to support the large prefetch sizes necessary to obtain high throughput from hard drives.

File Placement Once the storage devices are provisioned, files need to be placed, based on characteristics of the titles and of the storage devices. The titles that are popular, but not popular enough to be cached in system memory should be placed where throughput is the highest: on NVMe or SSDs, or at low block addresses on hard drives, where the transfer rate is higher (e.g., see Table 3.4). It may also be beneficial to migrate titles between different storage devices or servers when popularity changes or when the catalog contents are changed.

Load Balancing Files are placed based on anticipated demand, and if that demand is not estimated correctly, some drives may become hot spots. This may affect the clients that are downloading content from the hot spot hard drive, and may result in a lower quality of experience for users because either the client devices will have to switch to lower bit rate content which results in playback with lower quality, or the client devices will be forced to freeze playback and rebuffer. One method to reduce the load for a hot spot drive is to copy files that are in high demand onto other storage devices with excess capacity, from the very busy hard drive to an underutilized hard drive, or to faster NVMe, SSDs, or system memory.

6.2.4 Investigate Memory Management

In this thesis, we have chosen to implement changes at the web server application level to ensure the portability of our algorithms. We rely on the default kernel algorithms for caching content and managing system memory. The memory manager must balance the amount of system memory allocated between caching and prefetching. Memory can be used to increase hard drive throughput via prefetching, and to reduce the amount of data read from hard drives by caching content. The page replacement algorithms in the kernel should be designed to avoid both cache pollution and evicting prefetched data before it is requested.

In our experiments in Section 4.5.2, we show that the benefits of caching depend on the popularity distribution of titles; the cache hit rate was about 3% for α = 0.6, 15% for α = 0.8, and 42% for α = 1.0. For the Netflix workload popularity distributions in Section 5.2.1, the α values are approximately 0.6 and 0.5, making it unlikely that the existing approach used by the kernel for caching will be very effective for these Netflix servers. However, it is possible that a different cache algorithm, like *interval caching* [31] might provide a higher cache hit rate.

6.3 Concluding Remarks

We analyze the characteristics of two different HTTP streaming workloads, and show that we can greatly increase aggregate throughput for both workloads by using aggressive prefetching. We show that throughput could be further improved by making use of workload-specific characteristics such as the bit rate of the content being requested. Although the implementation was done in the `userver`, our work could be easily used in other servers, such as `nginx`.

We believe the results of this thesis will have significant and immediate impact. Netflix is currently studying these findings and is considering adopting some of the strategies we have devised into their `nginx` web servers. Specifically, Netflix is investigating: 1) the use of larger prefetch sizes, 2) the use of different prefetch sizes for different bit rates (e.g., setting prefetch sizes proportional to the square root of the bit rate), and 3) the use of information about the client's progress to potentially moderate aggressive prefetching when clients are in a transient phase. We believe the ideas and tools in our thesis can be adapted for use with other web servers and operating systems, and can be used to increase the capacity of HTTP streaming video servers which are the source of the majority of Internet traffic.

References

[1] A. Abhari and M. Soraya. Workload generation for YouTube. *Multimedia Tools and Applications*, 46(1):91–118, 2010.

[2] V. K. Adhikari, Y. Guo, F. Hao, V. Hilt, Z. L. Zhang, M. Varvello, and M. Steiner. Measurement study of Netflix, Hulu, and a tale of three cdns. *IEEE/ACM Transactions on Networking*, 23(6), Dec 2015.

[3] Vijay Kumar Adhikari, Yang Guo, Fang Hao, Matteo Varvello, Volker Hilt, Moritz Steiner, and Zhi-Li Zhang. Unreeling Netflix: Understanding and improving multi-CDN movie delivery. In *Proc. IEEE INFOCOM*, 2012.

[4] Vijay Kumar Adhikari, Sourabh Jain, Yingying Chen, and Zhi-Li Zhang. Vivisecting YouTube: An active measurement study. In *INFOCOM, 2012 Proceedings IEEE*, 2012.

[5] Akamai Corporation. *The State of the Internet*, Q2, 2011. http://www.akamai.com/dl/-whitepapers/akamai_soti_q211.pdf.

[6] Saamer Akhshabi, Ali C. Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proceedings of the second annual ACM conference on Multimedia systems*, 2011.

[7] Shane Alcock and Richard Nelson. Application flow control in YouTube video streams. *SIGCOMM Comput. Commun. Rev.*, 2011.

[8] Pablo Ameigeiras, Juan J. Ramos-Munoz, Jorge Navarro-Ortiz, and J.M. Lopez-Soler. Analysis and modelling of youtube traffic. *Transactions on Emerging Telecommunications Technologies*, 23(4).

[9] K. S. Anderson, J. P. Bigus, E. Bouillet, P. Dube, N. Halim, Z. Liu, and D. Pendarakis. Sword: Scalable and flexible workload generator for distributed data processing systems. In *Proc. Winter Simulation Conference*, 2006.

[10] Martin Arlitt and Carey Williamson. Understanding web server configuration issues. *Software: Practice and Experience*, 34(2), 2004.

[11] Athula Balachandran, Vyas Sekar, Aditya Akella, and Srinivasan Seshan. Analyzing the potential benefits of CDN augmentation strategies for Internet video workloads. In *Proc. ACM IMC*, 2013.

[12] Ali C. Begen, Tankut Akgul, and Mark Baugher. Watching video over the web: Part 1: Streaming protocols. *IEEE Internet Computing*, 15(2):54–63, 2011.

[13] S. Bhatia, E. Varki, and A. Merchant. Sequential prefetch cache sizing for maximal hit rate. In *Proc. MASCOTS*, 2010.

[14] Youmna Borghol, Siddharth Mitra, Sebastien Ardon, Niklas Carlsson, Derek Eager, and Anirban Mahanti. Characterizing and modelling popularity of user-generated videos. *Performance Evaluation*, 68(11), 2011.

[15] Andrew Brampton, Andrew MacQuire, Michael Fry, IdrisA. Rai, NicholasJ.P. Race, and Laurent Mathy. Characterising and exploiting workloads of highly interactive video-on-demand. *Multimedia Systems*, 2009.

[16] T. Brecht, D. Pariag, and L. Gammo. accept()able strategies for improving web server performance. In *Proc. USENIX Annual Technical Conference*, 2004.

[17] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.*, 14, 1996.

[18] Emmanuel Cecchet, Veena Udayabhanu, Timothy Wood, and Prashant Shenoy. Benchlab: an open testbed for realistic benchmarking of web applications. In *Proc. USENIX WebApps*, 2011.

[19] M. Cha, H. Kwak, P. Rodriguez, Y. Y Ahnt, and S. Moon. I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system. In *Proc. ACM IMC*, 2007.

[20] X. Che, B. Ip, and L. Lin. A survey of current youtube video characteristics. *IEEE MultiMedia*, 22(2), Apr 2015.

[21] Liang Chen, Yipeng Zhou, and Dah Ming Chiu. Video browsing - A study of user behavior in online VoD services. In *Proc. International Conference on Computer Communications and Networks (ICCCN)*, 2013.

[22] Yishuai Chen, Baoxian Zhang, Yong Liu, and Wei Zhu. Measurement and modeling of video watching time in a large-scale Internet video-on-demand system. *IEEE Transactions on Multimedia*, 15(8), 2013.

[23] X. Cheng, C. Dale, and J. Liu. Statistics and social network of YouTube videos. In *IEEE International Workshop on Quality of Service, IWQoS*, pages 229–238, 2008.

[24] Xu Cheng. Understanding the characteristics of Internet short video sharing: YouTube as a case study. In *Proc. ACM IMC*, 2007.

[25] Gyu Sang Choi, Jin-Ha Kim, Deniz Ersoz, and Chita R. Das. A multi-threaded pipelined web server architecture for SMP/SoC machines. In *Proceedings of the 14th international conference on World Wide Web*, 2005.

[26] Shaiful Alam Chowdhury and Dwight Makaroff. Characterizing videos and users in YouTube: A survey. In *Proc. Seventh International Conference on Broadband, Wireless*

Computing, Communication and Applications, 2012.

[27] Cisco Inc. Cisco visual networking index: Forecast and methodology, 2014-2019, 2014. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next--generation-network/white_paper_c11-481360.pdf.

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. ACM SoCC*, 2010.

[29] Cristiano P. Costa, Italo S. Cunha, Alex Borges, Claudiney V. Ramos, Marcus M. Rocha, Jussara M. Almeida, and Berthier Ribeiro-Neto. Analyzing client interactivity in streaming media. In *Proc. 13th International Conference on World Wide Web*, 2004.

[30] A. Dan, D. M. Dias, R. Mukherjee, D. Sitaram, and R. Tewari. Buffering and caching in large-scale video servers. In *Proc. of the 40th IEEE Computer Society International Conference*, 1995.

[31] Asit Dan and Dinkar Sitaram. A generalized interval caching policy for mixed interactive and long video workloads. In *Readings in Multimedia Computing and Networking*. 2001.

[32] 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. In *Proc. ACM SIGCOMM*, 2011.

[33] Jeffrey Erman, Alexandre Gerber, K. K. Ramadrishnan, Subhabrata Sen, and Oliver Spatscheck. Over the top video: The gorilla in cellular networks. In *Proc. SIGCOMM Internet Measurement Conference*, 2011.

[34] A. Finamore, M. Mellia, M. Munafo, R. Torres, and S.G. Rao. YouTube everywhere: Impact of device and infrastructure synergies on user experience. In *Proc. ACM IMC*, 2011.

[35] Monia Ghobadi, Yuchung Cheng, Ankur Jain, and Matt Mathis. Trickle: Rate limiting YouTube video streaming. In *Proc. USENIX ATC*, 2012.

[36] Debasish Ghose and Hyoung Joong Kim. Scheduling video streams in video-on-demand systems: A survey. *Multimedia Tools Appl.*, 2000.

[37] Binny S. Gill and Luis Angel D. Bathen. Optimal multistream sequential prefetching in a shared cache. *Trans. Storage*, 3(3), 2007.

[38] Binny S. Gill and Dharmendra S. Modha. SARC: sequential prefetching in adaptive replacement cache. In *Proc. USENIX ATEC*, 2005.

[39] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. YouTube traffic characterization: A view from the edge. In *Proc. ACM IMC*, 2007.

[40] Ashif Harji, Peter Buhr, and Tim Brecht. Our troubles with Linux and why you should care. In *Proc. 2nd ACM SIGOPS Asia-Pacific Workshop on Systems*, 2011.

[41] Ashif Harji, Peter Buhr, and Tim Brecht. Comparing high-performance multi-core web-server architecture. In *Proc. SYSTOR*, 2012.

[42] Ashif S. Harji. *Performance Comparison of Uniprocessor and Multiprocessor Web Server*

Architectures. PhD thesis, University of Waterloo, 2010. http://uwspace.uwaterloo.ca/-bitstream/10012/5040/1/Harji_thesis.pdf.

[43] HP Labs. The userver home page, 2003. Available at http://hpl.hp.com/research/linux/-userver.

[44] Te-Yuan Huang, Nikhil Handigol, Brandon Heller, Nick McKeown, and Ramesh Johari. Confused, timid, and unstable: Picking a video streaming rate is hard. In *Proc. ACM Conference on Internet Measurement Conference*, 2012.

[45] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proc. SIGCOMM'14*, 2014.

[46] Song Jiang. DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *USENIX FAST*, 2005.

[47] Shudong Jin and Azer Bestavros. GISMO: A generator of internet streaming media objects and workloads. *ACM SIGMETRICS Perf. Eval. Rev.*, 29(3):2–10, 2001.

[48] X. Kang, H. Zhang, G. Jiang, H. Chen, K. Yoshihira, and X. Meng. Measurement, modeling, and analysis of Internet video sharing site workload: A case study. In *Proc. IEEE ICWS*, 2008.

[49] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proc. OSDI*, 2000.

[50] Tracy Kimbrel, Andrew Tomkins, R. Hugo, Patterson Brian, and Bershad Pei Cao. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proc. OSDI*, 1996.

[51] Michel Laterman. Netflix and twitch traffic characterization. Master's thesis, University of Calgary, 2015.

[52] B. Laurie and P. Laurie. *Apache: The Definitive Guide, 2nd Edition*. O'Reilly, February 1999.

[53] Stefan Lederer, Christopher Müller, and Christian Timmerer. Dynamic adaptive streaming over HTTP dataset. In *Proc. MMSys*, 2012.

[54] Cheng Li, Philip Shilane, Fred Douglis, Darren Sawyer, and Hyong Shim. Assert(!defined(sequential i/o)). In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[55] Chuanpeng Li and Kai Shen. Managing prefetch memory for data-intensive online servers. In *Proc. USENIX FAST*, 2005.

[56] Chuanpeng Li, Kai Shen, and Athanasios E. Papathanasiou. Competitive prefetching for concurrent sequential I/O. In *Proc. EuroSys '07*, 2007.

[57] Mingju Li, Elizabeth Varki, Swapnil Bhatia, and Arif Merchant. TaP: Table-based prefetching for storage caches. In *Proc. USENIX FAST '08*, 2008.

[58] W. Li, W. B. Zheng, and X. H. Guan. Application controlled caching for web servers.

Enterp. Inf. Syst., 1(2), 2007.

[59] Xi Liu, Florin Dobrian, Henry Milner, Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. A case for a coordinated Internet video control plane. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, 2012.

[60] Yao Liu, Qi Wei, Lei Guo, Bo Shen, Songqing Chen, and Yingjie Lan. Investigating redundant Internet video streaming traffic on iOS devices: Causes and solutions. *IEEE Transactions on Multimedia*, 2014.

[61] M. Mansour, M. Wolf, and K. Schwan. Streamgen: A workload generation tool for distributed information flow applications. In *Proc. ICPP*, 2004.

[62] Ahmed Mansy, Mostafa Ammar, Jaideep Chandrashekar, and Anmol Sheth. Characterizing client behavior of commercial mobile video streaming services. In *Proc. Workshop on Mobile Video Delivery*, MoViD'14, 2013.

[63] J. Martin, Yunhui Fu, N. Wourms, and T. Shaw. Characterizing Netflix bandwidth consumption. In *Consumer Communications and Networking Conference (CCNC), 2013 IEEE*, 2013.

[64] Siddharth Mitra, Mayank Agrawal, Amit Yadav, Niklas Carlsson, Derek Eager, and Anirban Mahanti. Characterizing web-based video sharing workloads. *ACM Trans. Web*, 2011.

[65] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *Proc. 1st Workshop on Internet Server Performance*, 1988.

[66] AntoineN. Mourad. Issues in the design of a storage server for video-on-demand. *Multimedia Systems*, 4(2), 1996.

[67] Clément Nedelcu. *Nginx HTTP server second edition*. Packt Publishing Ltd, 2013.

[68] Netflix. ISP partnership options. https://openconnect.itp.netflix.com/deliveryOptions/-index.html.

[69] Netflix. Open connect appliance hardware. https://openconnect.itp.netflix.com/hardware/-index.html.

[70] Netflix. Netflix content delivery summit keynote. May 2013. Available at http://blog.streamingmedia.com/wp-content/uploads/2014/02/2013CDNSummit-Keynote-Netflix.pdf.

[71] Netflix. Q1 16 letter to shareholders. April 2016. Available at http://ir.netflix.com/-results.cfm.

[72] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 1993.

[73] B. Ozden, R. Rastogi, and A. Silberschatz. Buffer replacement algorithms for multimedia storage systems. In *Proc. Multimedia Computing and Systems*, 1996.

[74] Banu Ozden, Rajeev Rastogi, and Avi Silberschatz. On the design of a low-cost video-on-demand storage system. *MULTIMEDIA SYSTEMS*, 1996.

[75] V.S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proc. USENIX*, 1999.

[76] G. Panagiotakis, M.D. Flouris, and A. Bilas. Reducing disk I/O performance sensitivity for large numbers of sequential streams. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*, 2009.

[77] D. Pariag, T. Brecht, A. Harji, P. Buhr, and A. Shukla. Comparing the performance of web server architectures. In *Proc. ACM EuroSys*, 2007.

[78] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. SOSP '95*, 1995.

[79] Ashwin Rao, Arnaud Legout, Yeon-sup Lim, Don Towsley, Chadi Barakat, and Walid Dabbous. Network characteristics of video streaming traffic. In *Proc. Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, 2011.

[80] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, 1997.

[81] Yaoping Ruan and Vivek S. Pai. Understanding and addressing blocking-induced network server latency. In *Proc. USENIX Annual Technical Conference*, 2006.

[82] Sandvine Inc. Global Internet phenomena report – spring 2011, 2011.

[83] Sandvine Inc. Global Internet phenomena – Africa, Middle East & North America, Dec 2015. https://www.sandvine.com/downloads/general/global-internet-phenomena/2015/global-internet-phenomena-africa-middle-east-and-north-america.pdf.

[84] Priya Sehgal, Vasily Tarasov, and Erez Zadok. Evaluating performance and energy in file system server workloads. In *Proc. FAST'10*, 2010.

[85] Elizabeth AM Shriver, Christopher Small, and Keith A Smith. Why does file system prefetching work? In *Proc. USENIX ATC*, 1999.

[86] I. Sodagar. The MPEG-DASH standard for multimedia streaming over the Internet. *MultiMedia, IEEE*, 18(4), April 2011.

[87] Thomas Stockhammer. Dynamic adaptive streaming over HTTP –: standards and design principles. In *Proc. MMSys*, 2011.

[88] Storage Review. Hitachi deskstar 5k4000 review. http://www.storagereview.com/-hitachi_deskstar_5k4000_review.

[89] Jim Summers, Tim Brecht, Derek Eager, and Alex Gutarin. Characterizing the workload of a Netflix streaming video server. In *Proc. IISWC*, 2016, in press.

[90] Jim Summers, Tim Brecht, Derek Eager, Tyler Szepesi, Ben Cassell, and Bernard Wong. Automated control of aggressive prefetching for HTTP streaming video servers. In *Proc. SYSTOR*, 2014.

[91] Jim Summers, Tim Brecht, Derek Eager, and Bernard Wong. Methodologies for generating HTTP streaming video workloads to evaluate web server performance. In *Proc. SYSTOR*, 2012.

[92] Jim Summers, Tim Brecht, Derek Eager, and Bernard Wong. To chunk or not to chunk: Implications for HTTP streaming video server performance. In *Proc. ACM NOSSDAV*, 2012.

[93] W. Tang, Y. Fu, L. Cherkasova, and A. Vahdat. Medisyn: A synthetic streaming media service workload generator. In *Proc. ACM NOSSDAV*, 2003.

[94] Rajeev Tiwari. Mpeg-dash support in YouTube, 2013.

[95] Fengguang Wu. Sequential file prefetching in Linux. *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*, pages 218–261, 2009.

[96] Hongliang Yu, Dongdong Zheng, Ben Y. Zhao, and Weimin Zheng. Understanding user behavior in large-scale video-on-demand systems. In *Proc. ACM EuroSys*, 2006.

[97] M. Zink, K. Suh, Y. Gu, and J. Kurose. Watch global, cache local: YouTube network traffic at a campus network - measurements and implications. In *Proceedings of SPIE - The International Society for Optical Engineering*, volume 6818, 2008.

[98] M. Zink, K. Suh, Y. Gu, and J. Kurose. Characteristics of YouTube network traffic at a campus network - measurements, models, and implications. *Computer Networks*, 53(4), 2009.