Methodologies for Generating HTTP Streaming Video Workloads to Evaluate Web Server Performance

Jim Summers, Tim Brecht
University of Waterloo
jasummer,brecht@cs.uwaterloo.ca

Derek Eager
University of Saskatchewan
eager@cs.usask.ca

Bernard Wong
University of Waterloo
bernard@cs.uwaterloo.ca

Abstract

Recent increases in live and on-demand video streaming have dramatically changed the Internet landscape. In North America, Netflix alone accounts for 28% of all and 33% of peak downstream Internet traffic on fixed access links, with further rapid growth expected [26]. This increase in streaming traffic coincides with the steady adoption of HTTP for use in video streaming. Many streaming video providers, such as Apple, Adobe, Akamai, Netflix and Microsoft, now use HTTP to stream content [5]. Therefore, it is critical that we understand the impact of this emerging workload on web servers. Unlike other web content, a recent study [13] of streaming video shows that even small infrequent latency spikes, manifested as buffering related pauses, can result in shorter viewing times especially during live broadcasts. Unfortunately, no appropriate benchmarks exist to evaluate web servers under HTTP video streaming workloads.

In this paper, we devise tools and methodologies for generating workloads and benchmarks for video streaming systems. We describe the difficulties encountered in trying to utilize existing workload characterization studies, motivate the need for workloads, and create example benchmarks. We use these benchmarks to examine the performance of three existing web servers (`Apache`, `nginx`, and `userver`). We find that simple modifications to `userver` provide promising and significant benefits on some representative streaming workloads. While these results warrant additional investigation, they demonstrate the need for and value of HTTP video streaming benchmarks in web server development.

1. Introduction

Many modern video streaming services have eschewed the previously dominant streaming protocols, such as Real-time Transport Protocol (RTP) and Real-Time Streaming Protocol (RTSP), in favour of having simple clients that request chunks (a few seconds of video) via HTTP over TCP/IP from standard, stateless web servers. This technique is being used by Apple, Adobe, Akamai, Netflix, Microsoft, and many others [5]. The switch to HTTP fundamentally changes the role of a streaming video server; rather than have servers *push* the data to the clients, the clients instead *pull* data from the servers using HTTP requests.

The dominance of HTTP stems from the following advantages: HTTP is simple and stateless; it can easily traverse firewalls (since it uses TCP); and it can leverage the existing ubiquitous infrastructure such as web servers, caches, and Content Distribution Networks (CDNs). We refer to this infrastructure as the HTTP ecosystem. In spite of these advantages, there are performance uncertainties in using HTTP for video streaming due to the lack of video streaming benchmarks targeting the HTTP ecosystem. While we believe that it will be important to develop and use benchmarks for all aspects of the HTTP ecosystem, in this paper we start by focusing on web servers, as they are the most central and performance critical component in the ecosystem.

This work was motivated by the absence of existing benchmarks that can be used to evaluate new techniques for improving web server performance under video streaming workloads. While many of the techniques we were considering were effective on micro-benchmarks, the lack of existing benchmarks prevented us from conducting meaningful comparisons with other servers and from understanding whether or not the benefits existed under representative workloads.

Our goal in this paper is to take the necessary first steps to develop tools and methodologies for generating and running benchmarks for modern HTTP-based streaming video services. Our contributions in this paper are:

- We develop tools and methodologies to create video streaming benchmarks that can be used to evaluate the performance of web servers. We describe our modifications to `httperf` [22], a general benchmarking tool, to generate traffic according to our workloads and ensure quality of service requirements are met. We also describe how we use `dummynet` [24] to simulate the variety of connec-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SYSTOR'12, June 4–6, 2012, Haifa, Israel.
Copyright © 2012 ACM 978-1-4503-1448-0/12/06...$10.00

tions common today, including home broadband connections and low-bandwidth high-delay wireless networks.
- We incorporate results from several papers on workload characterization of modern Internet video services in our workload generator. This ensures that our generated workloads and benchmarks exercise web servers in the same manner as users would in a real deployment.
- We examine the performance of three web servers (`Apache`, `nginx`, and `userver`). We find that relatively simple modifications to `userver` can provide significant benefits for some video streaming workloads but are ineffectual for others. These results motivate the need for further studies using a variety of realistic video streaming workloads to ensure web servers are prepared for current and future video streaming demands.

2. Background and Related Work

Existing web server workloads and benchmarks, such as SPECweb2009 [29], reflect traffic characteristics that were prevalent in the past and differ significantly from HTTP video traffic. Web server research has concentrated on serving requests for items that are primarily small and exhibit lots of locality [7, 8, 23, 27, 28]. In contrast, video files are large and, while there is some locality, there is a long tail of content that is viewed only a small number of times [16]. Although the existing HTTP ecosystem can service videos and other large files, users often experience significant delays while waiting for video data to be delivered. This is especially apparent when watching high-quality video. A recent study [13] shows that even occasional pauses lead to shorter video viewing times, especially during live events. Also, clients rarely watch an entire video, typically terminating a connection before reaching the end; 60% of YouTube videos are watched for less than 20% of their duration [14].

Previous work has also largely ignored the growing number of devices accessing Internet services over low-bandwidth, high-delay networks. Instead experiments are conducted in laboratories using high-speed (gigabit) local area networks [7, 23, 27, 28]. It is therefore essential that we develop benchmarks for HTTP-based video servers that can be used in laboratory settings but emulate the wide variety of devices and networks used to view video content.

Existing video workload generators are unsuitable for our purposes because they generate workloads for analysis and simulation rather than for creating traffic to exercise a web server [1, 18, 31, 34]. Although these workloads could be used as part of a benchmark [18], it would require significant re-engineering of the workload generators.

General-purpose workload generators [4, 9, 12] can be used to benchmark HTTP-based systems but these generators have not been configured to produce HTTP streaming video. In order to use one of these generators, we would have to implement a new client module, and such extensions are non-trivial to implement. Therefore, we instead use `httperf` [22], an existing HTTP traffic generator; this approach is shared by several other benchmarks [6, 21].

There are many sources of real-world video information available for use in workload generation. These include papers describing workload generators, which typically include measurements as sources for their workload distributions, but the major source of information is papers that focus on characterizing measured video traffic. Many different types of videos and delivery systems have been studied: user generated video sites like YouTube [1, 10, 11, 15, 16, 34] and Yahoo! video [19], video on demand sites [32], and corporate video websites [31]. All of these papers measure and characterize the important video properties, such as the popularity distribution, duration, bitrate and size information. Although it is possible to do useful analysis with only video information, like estimating the effectiveness of proxy caching [1] compared to server patching [18] or peer-to-peer caching [34], it is also necessary to understand and model client behaviour to construct a representative benchmark.

A realistic benchmark must therefore accurately model client network behaviour such as bandwidth and latency. There has been a significant amount of work in measuring and modeling client behaviour [15, 16, 31, 32]. For our work, we primarily utilize measurements from [15]; these measurements include detailed information about client streaming sessions and were obtained recently, which is critical to our focus on current and future video streaming workloads.

3. Objectives

Our primary objectives for this paper are to devise flexible tools and methodologies for constructing HTTP streaming video workloads that can be used to examine, understand, compare and improve the performance of web servers. Although the work in this paper is focused on web server performance, our long term goal is to also be able to evaluate proxy caches, CDNs, and possible protocol improvements or modifications.

In addition to our overarching objectives, there are three specific goals for our web server benchmark. First, it should generate web server loads that are representative of what we would measure at HTTP streaming video web servers in real deployments. Second, the benchmark should measure the long-term 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.

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 im-

portant observations and parameters (including distributions) required to sufficiently characterize a workload.

2. Construct a workload: Using the workload specification and a workload constructor we create log files (`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 also includes populating the server with files 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 1 illustrates these different phases. It is important to note that, in this paper, we generate workloads based on the information we have obtained from several different papers that characterize YouTube video requests. If additional information for YouTube requests or a characterization of video requests for a different service becomes available, we believe that our methodology will permit one to easily create a new workload.

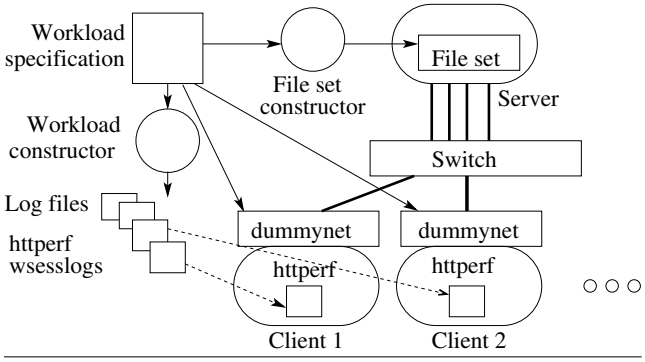

Figure 1. Overview of the methodology

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

- Videos 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 video.
- Because we expect that many video workloads will be disk intensive, we felt that it would be beneficial to evaluate different disk placement issues without having to generate a different set of files (which would not be amenable to a fair comparison). We create a generic set of fixed-length files on the server, which can later be assigned to different videos to examine disk placement issues.
- Some services use HTTP range requests to ask for a specified portion of a single video file. Other services divide the video into chunks that are stored in separate files and use HTTP GET requests to obtain the desired file. In this paper we follow the latter approach, but our tools are capable of generating workloads that use range requests.
- We assume that HTTP requests related to searching and browsing for videos occur on a separate machine. This is how large video systems are designed in practice [15, 32] and it simplifies the benchmark.
- We want to be able to implement clients that use HTTP adaptive streaming. It would require a specialized client application to truly adapt to real-time network conditions, but we can simulate rate adaptation with `httperf` by generating session logs that switch between videos with different encodings at predetermined times.

4. Workload Specification

Our goal in designing benchmark workloads is to accurately model the request traffic of real web servers streaming videos 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 available memory. Generating too few or many videos will result in unrealistically high or low disk cache hit rates respectively; this would significantly skew the performance results and could lead to incorrect conclusions when comparing different designs.

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. We expect that many user-behaviour related parameters, such as parameters concerning viewing habits, will not change significantly. However, future values for parameters such as video bitrate and client downstream bandwidth will likely be very different than today's values. Therefore, in designing a reconfigurable workload, we explicitly separate the highly variable parameters, which enables us to quickly experiment with different workload configurations.

In addition to reconfigurability, another workload design goal is reducing benchmark runtime. Short benchmarks 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 7200 sessions ensure that the web server performance reaches steady-state.

In the following sections, we describe in detail the design of one example video streaming workload suitable for our experiment environment.

4.1 Video Characteristics

Our video and session characteristics and distributions are drawn from [15]. This paper provides low-level details about client sessions by measuring traffic at the edge, and is a recent source for information regarding YouTube video characteristics and download mechanisms.

There is much debate in the literature about the shape of a YouTube video popularity distribution. Some find a close fit with a Zipf distribution [16]. Others find that Gamma or Weibull distributions fit more closely [11]. For this workload, we use a Zipf distribution because previous work that measures over a short timeframe, similar to our target benchmark environment, finds that measurements follow a Zipf distribution. In contrast, measurements that sample over a longer period of time or rely on extracting viewing information from the YouTube database tend to have non-Zipf-like distributions. There is a discussion of these issues in [10].

The Zipf distribution requires two parameters; we chose an alpha value of 0.8 and a video population of 10000. The number of videos was based partly on the capacity of the hard drive in our server, which can hold 10000 videos with the average video size of 13 MB. The video library size was also chosen to suit our experiment length of 7200 video sessions. This choice of parameters results in about 35% of requests being serviced from the cache for our experiments.

An equally important parameter to the popularity distribution is the duration distribution of the video library. YouTube video durations have a complicated distribution; for example there are peaks at 200 seconds, the typical length of music videos, and at 10 minutes, a limit that YouTube imposes on video length. Some authors specify the duration algorithmically as an aggregation of normal distributions [11]. We use a CDF to represent the distribution of video durations rather than an analytical formula, because the distribution is likely to be irregular for any video library. Figure 2 shows the duration distribution used for this workload; it is based on data found in [15].

From this distribution, we assign a duration to each video without accounting for the video popularity. Although previous work [1] has suggested a weak correlation between popularity and duration, we have found no measurements to quantify such a correlation and have therefore omitted it in this workload. An additional concern with assigning durations is that the most popular videos make up a large proportion of the workload, so the durations assigned to these videos has a large effect on the proportion of sessions that can be serviced from the cache. We assign the median video duration to the two most popular videos because we believe this produces a more representative workload that is less sensitive to the choice of a random seed for the generator. This parameter can be configured to assign different fixed values, or to simply use randomly-assigned durations for all videos.

The final video characteristic we assign is the video bit rate. There are many different video encodings and variable

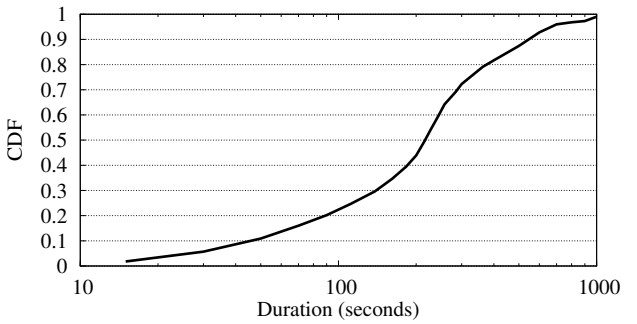

Figure 2. Duration of videos

bit rates used for YouTube videos, as observed by [15], with an average rate of 394 Kbps [16]. It simplifies our workload generator if we represent video chunks with a fixed size and fixed duration, so we assume a fixed bitrate for all videos. We chose a bitrate of 419 Kbps, which represents 10 seconds of video using 0.5 MB of data.

We expect that bitrates will change in the future, as more high resolution video is produced and viewed. We may also revisit the decision to use a constant bitrate, if servers are found to be sensitive to variable bitrate videos. Variable bitrates could be simulated by either modifying the log files so that fixed-size chunks represent different time spans, or by modifying the file set so that different size chunks are used to represent fixed time spans.

4.2 Session Characteristics

Our benchmark workload must, in addition to determining the characteristics of the videos, also specify how much of each video is downloaded by the clients. From previous studies, we know that most clients do not watch to the end of the videos. However, we can not accurately determine the amount of data that is downloaded using the session length alone because clients buffer data to compensate for variations in download speeds. There is one study [15] that provides the number of bytes downloaded in a session as a fraction of the video size, which we can use to determine session lengths accurately. Figure 3 shows the curve we use to determine what fraction of a video is downloaded.

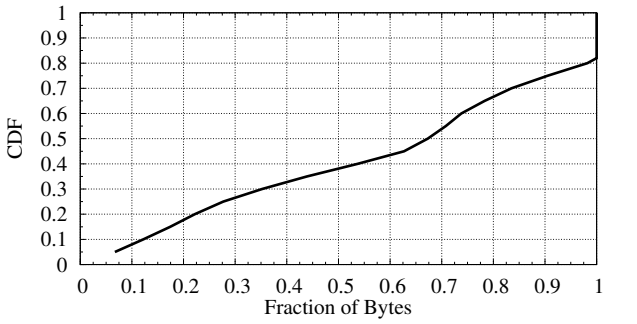

Figure 3. Fraction of bytes downloaded during session

Again, we choose a session fraction independently of video properties such as length or popularity even though there might be a weak correlation. For example, with the Video on Demand system studied in [32], sessions last longer for unpopular videos than popular videos, and this property might hold for YouTube videos as well. However, we believe that this is a reasonable simplification for this workload.

The final 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. There is no consensus on the inter-arrival time distribution in previous measurement studies. We chose to make inter-arrival times exponentially distributed (session arrivals occur according to a Poisson process), as other simple distributions, such as uniform, are even less realistic and can cause significant artifacts in the benchmark results.

4.3 Client Network Characteristics

Table 1 shows the access speeds we use for our clients in this workload. This data represents the access speeds of client computers in the United States to Akamai servers, as reported in [2]. We disregard the low speed clients because their connections are not suitable for viewing video and because they represent an insignificant fraction of the total. Because there is no detailed information about the distribution of access speeds, we represent each of three relevant categories with a single rate. These assigned rates are based on average access speed measured by Akamai.

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 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 YouTube users, so we simply assign a constant delay of 50 ms on both the forward and reverse paths, which is the approximate time to transmit from coast-to-coast in North America. Delays for mobile clients can be much larger; this may require that we revisit this design decision in the future.

5. Workload Generation

Table 2 provides a summary of the parameters we used to construct the sample workloads for this paper. We give a description of each parameter, its value or distribution, and the source of the measurement. This table is an abstract specification, as it could describe video sessions in any setting using any protocol. Once we create an experimental environment, then we have a target for a concrete implementation of the abstract session specification that is specific to the test equipment.

Parameter Description	Value used	Source
Video Popularity Dist.	Zipf $\alpha = 0.8$	[1]
Video Count	10000	
Video Duration Dist.	See Figure 2	[15]
Video Bit rate	419 kbps	[16]
Session Length Dist.	See Figure 3	[15]
Session Arrival Process	Poisson	[18, 19]
Session Count	7200	
Session Chunk Timeout	10 seconds	[5]
Client Network Bandwidth	See Table 1	[2]
Client Network Delay	50 ms, one way	
Client Request Size (MB)	0.5 and 2.0	
Client Request Pacing	Yes	
Client Adaptation	None	[15]
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	3500 chunks	
Server Ramp-Up	200 sessions	
Server Ramp-Down	100 sessions	

Table 2. Summary of workload specification

Some HTTP video providers, like YouTube, implement application flow control on the servers to limit the download rate of videos [3, 15]. This is primarily a mechanism to minimize wasted bandwidth when a client does not watch to the end of a video. This mechanism is not widely used by other streaming services and, even for YouTube, it is disabled for mobile clients with sporadic network connectivity.

Our workload generator uses a more generic approach to HTTP video streaming based on a representative HTTP video streaming platform, Apple's HTTP Live Streaming [5]. With this platform, videos are segmented into chunks, and the clients download the chunks at a limited rate using a technique called *pacing*. The clients first download chunks at full speed until a video buffer is filled, then request subsequent video chunks only when needed to refill the buffer, thus using less bandwidth than requesting chunks at full network rates. Not all HTTP video platforms use pacing, but it is a technique that enables true streaming video using any web server.

There are two ways to implement client chunking; the clients can use HTTP range requests to download chunks from a single video file, or the videos can be divided into chunks and stored in separate files that are requested by the clients. Our primary workload uses file-based chunking, with all videos divided into 10 second chunks and stored

in separate 0.5 MB files. This chunk size is the same as used by Apple's Live Streaming. We also create a secondary workload that uses a significantly larger 2.0 MB chunk size that we use only in Section 8.

5.1 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 video chunks must be created *a priori* in the server's file system. To accomplish this, we simply create many thousands of chunk-size files in the same directory, in numerical order, starting from a newly-installed file system. However, the results of this procedure are not repeatable, even starting from a newly created file system, so we have little control over file placement. For this reason, we create our file sets only once, so we can compare the results of different experiments.

Each video 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 length. We generate different workloads using the same file set by changing the association between videos and specific file chunks.

File chunks should be assigned to videos carefully to avoid bias in the results. In this paper, we assign video positions randomly, but in the future we intend to experiment with different file placement strategies.

Figure 4 shows the distribution of session lengths in the abstract specification, compared to the results when sessions are rounded up to the next multiple of the 0.5 MB chunks size. The minimum session length we can represent in a workload is 10 seconds. This artifact has little impact because the exact lengths of short sessions do not have much impact on the results.

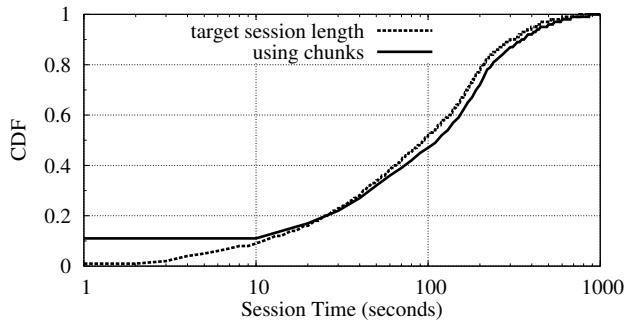

Figure 4. Using chunks to represent session lengths

5.2 Client Configuration

Our experiments utilize 12 client hosts to generate hundreds of concurrent 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 7200 Mbps aggregate bandwidth over all clients.

We approximate the specification in Table 1 by configuring `dummynet` to allow 10 Mbps 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 5 shows a small example of a `wsesslog` that contains requests for several videos.

Each video 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 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 video.

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.

For our primary workload, we generate `wsesslog` files with 10 second timeouts for each chunk. The first 3 chunks of each session are requested without pacing delays, simulating the filling of a buffer; and subsequent chunks are paced so they are requested at a rate of one chunk every 10 seconds.

Table 3 contains summary statistics that characterize our two workloads. Both are constructed using the specification in Table 2 and differ only in the chunk size. The first four values in Table 3 refer to statistics derived solely from the abstract specification, and so are the same for both workloads. The remaining values differ because session lengths are rounded up to a multiple of the chunk size.

```
# Session 1: 4 chunks with pacing
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 range requests
vid02 range=0-524287 timeout=10
vid02 range=524288-1048575 timeout=10
vid02 range=1048575-1572863 timeout=10

# Session 3: 2 chunks different quality
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
vid04/secs-20-29 timeout=10
vid04/secs-0-9 timeout=10
vid04/secs-100-109 timeout=10
```

Figure 5. Small example of an `httperf wsesslog`

Description	0.5 MB	2.0 MB
unique videos	3366	3366
single-session videos	67.5 %	67.5 %
average video duration	258.7 s	258.7 s
average video 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	60004	16318
total file chunks requested	105323	27188
number of chunks viewed once	44478	12293

Table 3. Characteristics of constructed workloads

6. 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 65535. 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 Intel 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. 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 [17].

On the clients, we use a version of `httperf` [22] that was locally modified to support new features of `wsesslog` and to track statistics on every requested chunk. We use `dummynet` [24], which comes with Ubuntu, to emulate different types of networks.

We use a number of different web servers. Most experiments use version 0.8.0 of `userver`, which has been previously shown to perform well [8, 23] 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.

7. Running Experiments

For our experiments, we measure the aggregate throughput of the server when servicing workloads at a series of different rates. We use the measurements to produce graphs, such as Figure 7, that show the aggregate throughput in MB/s from requests that were completely serviced prior to timeout. When the server is not overloaded, we expect the throughput to be equal to the chunk rate multiplied by the chunk size.

The methodology for running experiments and collecting measurements has a significant effect on the results. We explain how we ensure that experiments reach steady-state in a reasonable amount of time, demonstrate the importance of using `dummynet` to simulate client networks, and discuss the execution time and repeatability of our experiments.

7.1 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 start to measure sessions for 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 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 considerable time.

Figure 6 is an example of the curve we use to evaluate the progress of an experiment. This curve shows 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 response time is longer when the server is overloaded, as illustrated by the *no warming* curve in Figure 6. The other curves in the figure show the results when the cache is prewarmed with the most popular chunks before the start of the experiment, which can shorten the ramp-up time before session measurements can begin.

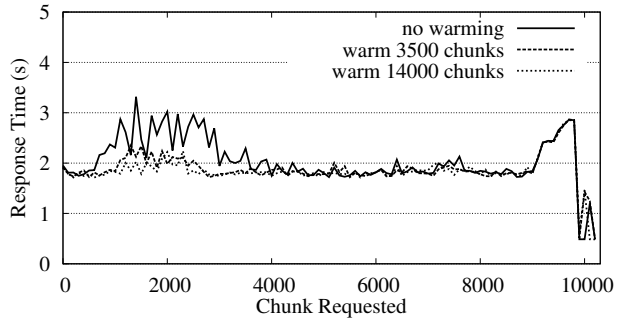

Figure 6. Cache warming techniques

We modified `httperf` to recognize ramp-up and ramp-down periods. Each instance of `httperf` processes 600 sessions in our workload and we use a ramp-up period of 200 sessions and a ramp-down period of 100 sessions. The average session consists of about 16 requests, so these periods correspond to 3200 requests at the beginning of the experiment and 1600 requests at the end.

We use a script to start the clients and server, and to start the tools we use to monitor the progress of the benchmark. `vmstat` is used to monitor CPU utilization. `iostat` monitors disk usage which includes bandwidth, transaction size, transaction times, queue lengths, and 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.

7.2 Bandwidth-Limited Clients

`dummynet` allows 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 `userver` to use a maximum of 100 processes and 20000 connections. In the other case, we configure `userver` to use 1 process and 1 connection. 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 7 shows the results of these two experiments. As can be seen by the two lines labeled *unthrottled*, the performance of the two vastly different configurations are quite close. However, the two lines labeled *throttled* show that the performance of these two configurations are dramatically different when using representative client networks. The strong performance of the single connection unthrottled case is a result of the data being sent unrealistically fast over the 1 Gbps network. This demonstrates the importance of simulating different client network speeds for this workload.

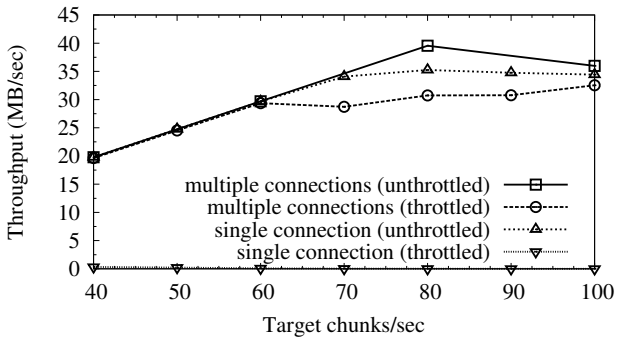

Figure 7. Using `dummynet` to model client networks

7.3 Duration and Repeatability

One of our stated goals was 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 paper, 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 7).

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 periodically sample the throughput while the experiments are running and compute a 95% confidence interval for that sampled throughput. For all the experiments in this paper, the 95% confidence interval for the sampled throughput is less than 1 MB/sec, indicating that the workloads are stable during the measurement period.

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 4 contains measured statistics for experiments described in Section 8. The confidence intervals for throughput are small, particularly when there are no failures during the experiment. The confidence intervals for the fail-

ure rates are larger, but still show that the experiments are repeatable even when the server is severely overloaded.

Request Rate	Tput Mean (MB/s)	Tput CI (MB/s)	Failure Mean (%)	Failure CI (%)
Figures 10 and 11, `userver` prefetch				
20	39.2	0.01	0.0	0.00
40	68.4	0.09	5.7	0.13
Figures 10 and 11, `userver` noprefetch				
20	33.1	0.12	14.1	0.95
40	33.3	0.61	52.5	1.68
Figures 8 and 9, `userver` noprefetch				
70	33.8	0.06	2.3	0.17
100	38.9	0.15	19.6	0.66

Table 4. Confidence intervals for some `userver` runs

8. Web Server Benchmarks

Using our example workload and methodology for conducting benchmarks, we examine the performance of three different open-source web servers: `Apache`, `nginx` and two different configurations of `userver`.

The `userver` configurations differ in how `sendfile` is used. In general, `sendfile` is considered the most efficient way to service static workloads from the file system cache; it avoids buffer copy overhead by transmitting the contents to a client directly from the cache. However, we found that `sendfile` is inefficient when the contents are not found in the file system cache. `sendfile` only fetches enough data with each disk read to refill the socket buffer, which is sized based on the characteristics of the network rather than those of the disk. Reading from disk in this way exposes two inefficiencies: small disk reads and additional seeks when concurrently servicing multiple client requests.

One configuration of `userver` uses `sendfile` directly and blocks when data must be read from disk. The other leverages a feature of the FreeBSD version of `sendfile` to avoid blocking. Rather than blocking, `sendfile` can instead return immediately with an error code [25]. Upon receiving this error code, we use a helper thread to prefetch an entire chunk into the file system cache, rather than read only enough to fill the socket buffer. The single helper thread also ensures that multiple files will not be read concurrently; it reads only a single chunk at a time, and queues other pending reads. We refer to the configuration that uses a helper thread as *prefetch* `userver` and the other configuration as *noprefetch* `userver`.

Our overarching goal is to determine whether web servers can be better implemented and tuned to service HTTP streaming workloads. From preliminary investigations with micro-benchmarks, we found that poor disk throughput can limit web server performance. Therefore, to maximize the performance of web servers for streaming video, we must investigate strategies for maximizing disk performance. The prefetching we introduced in `userver` is one such strategy; we evaluate its performance impact by comparing the two configurations of `userver`. We also test against `Apache` and `nginx`, two widely-used web servers.

8.1 Effect of File System

One of the basic decisions when setting up a video server is how to store the videos; if the HTTP video system uses file-based chunking, there is a choice of the size to use for the chunks. We created two workloads that differ only in the chunk size to investigate the performance implications of the choice. Our results suggest that increasing the chunk size can make a huge difference, and provides motivation to improve how HTTP streaming video servers create and access the files storing the videos.

Figure 8 shows the throughput of four different servers using the 0.5 MB chunk workload. For these experiments, we vary the target chunk rate between 40 and 100 chunks/s. When the request rate exceeds the capacity of the server, it is not possible to completely service all the sessions. Figure 9 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 `userver` without prefetching, and the difference is larger than the 95% confidence intervals, so these server configurations are somewhat better at servicing the 0.5 MB workload.

Table 5 shows the results from monitoring the disk performance during the experiments at 70 chunks/sec. In this table, the results for `userver` are labeled *nopr* for the noprefetch configuration and *pr* for the prefetch configuration. The average times for read transactions are lower for `nginx` and `userver` noprefetch, which may explain why the performance of those servers is slightly better.

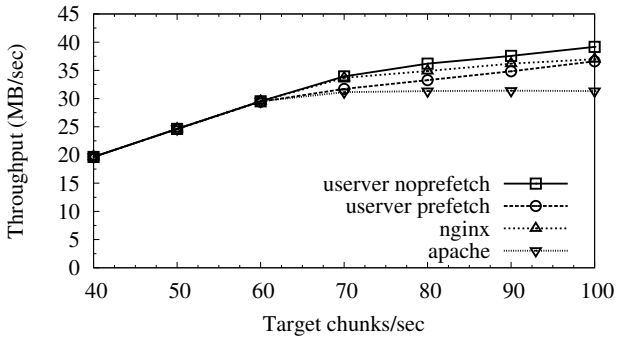

Figure 8. Aggregate throughput with 0.5 MB chunk size

The prefetch `userver` performs worse than two of the other servers with the 0.5 MB chunk workload because the serialization of disk access by the web server prevents the kernel from scheduling disk I/O to minimize seek distances. In contrast, the serialization of disk access is beneficial for the 2.0 MB workload. Figure 10 shows the aggre-

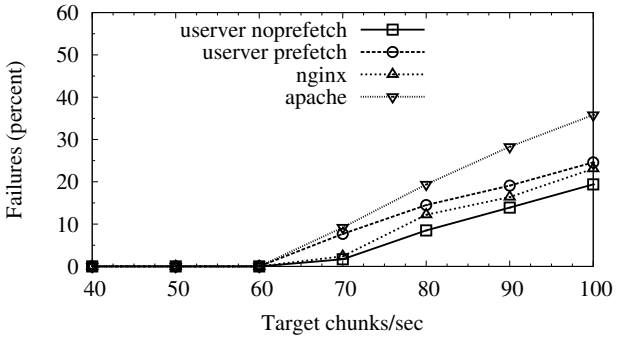

Figure 9. Missed deadlines, 0.5 MB chunks

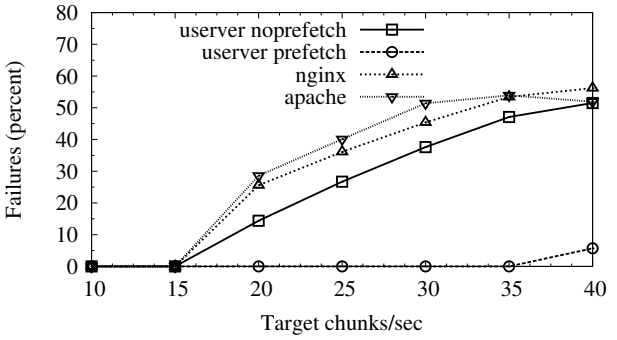

Figure 11. Missed deadlines, 2.0 MB chunks

Web server	Time Per Read (ms)	Avg Read Size (KB)	Tput MB/s	Disk Util
userver nopr	2.935	72.9	24.26	100%
userver pr	3.681	84.7	21.69	97%
Apache	3.115	71.3	22.14	99%
Nginx	2.818	71.1	24.66	100%
wc	2.292	84.0	37.15	91%

Table 5. Disk performance, 0.5 MB chunks at 70 req/s

Web server	Time Per Read (ms)	Avg Read Size (KB)	Tput MB/s	Disk Util
userver nopr	3.612	88.1	23.86	100%
userver pr	2.307	112.4	42.65	90%
Apache	2.247	39.3	17.18	100%
Nginx	4.288	87.4	19.94	100%
wc	2.014	112.0	48.79	90%

Table 6. Disk performance, 2.0 MB chunks at 35 req/s

gate throughput and Figure 11 shows the session failure rate when the chunk size is 2.0 MB.

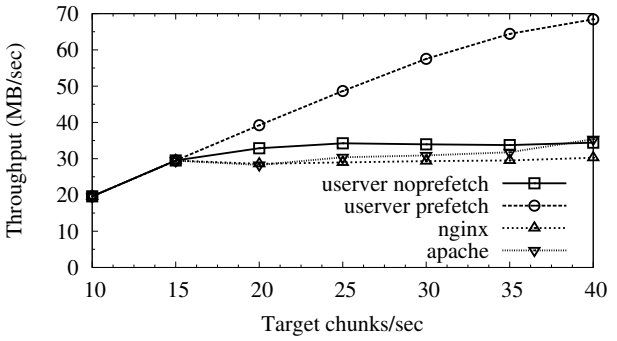

Figure 10. Aggregate throughput with 2.0 MB chunk size

The 2.0 MB workload uses the same abstract workload specification as the 0.5 MB workload, but the results are not comparable because the file sets are different and performance will vary because the session lengths are rounded differently in the two workloads. With the 2.0 MB chunk size, prefetch `userver` reaches a failure-free throughput of 35 chunks/sec, 133% higher than the failure-free throughput of the other servers. Table 6 shows that there is much higher disk throughput with prefetching, both because of a low read time, and because the average read size is large.

For these workloads, the disk is the bottleneck and determines the performance of the web server. Therefore, we complete our examination of disk performance by establishing an approximate upper limit on disk throughput when reading our two different file sets. This examination involves running a simple workload experiment using `wc`, the standard Unix word count tool, to read the same file chunks that are requested as part of the workloads. The file chunks comprising each video are read in sequential order, but the videos are visited in a random order. The results of this experiment are labeled *wc* in Tables 5 and 6. The disk throughput of `userver` prefetch is only 13% lower than the performance of `wc` with 2.0 MB chunks, but the best server disk throughput using 0.5MB chunks is 33% lower than `wc`. Our prefetching technique makes effective use of available disk throughput when using 2.0 MB chunks, but it is not clear whether there is a way to make better use of potential disk performance when using 0.5 MB chunks.

8.2 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 video stored on the device at any point in time. This is done by first buffering a reasonable amount of data to play the video without having to rebuffer (i.e., stop video playback while waiting for video to be delivered) and then requesting more video 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 in some cases simply utilize the hard drive of the system in order to store video as it arrives. In this case, requests for the next chunk are sent to the server

as soon the previous reply arrives, essentially requesting chunks far in advance of when they 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 is identical to that used in Section 8.1 and used to produce the results shown in Figure 8, except for client 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 12 shows the results of this experiment. The lines in the graph that are labeled *userver noprefetch nopacing* and *userver prefetch nopacing* are results obtained using this new workload where clients do not pace their requests, while the other two lines are taken directly from Figure 8. It is interesting to note that when the userver is prefetching there is no difference in aggregate throughput, while when the userver is not prefetching the differences in aggregate throughput are significant, results that would have been difficult to predict *a priori*.

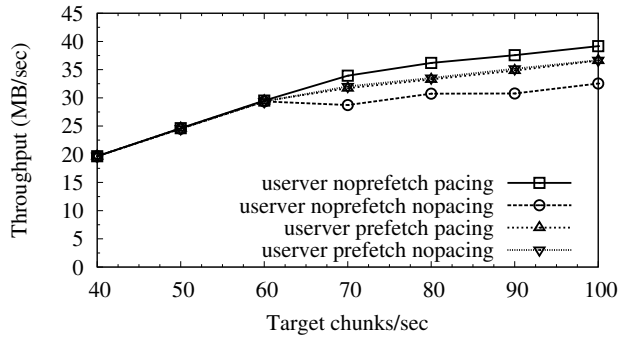

Figure 12. Effect of pacing on throughput

9. Discussion

Use of the methodologies described in this paper has allowed us to discover several interesting server design issues that appear to have substantial impacts on Web server performance for HTTP streaming video workloads. Perhaps most significantly, our performance results suggest the importance of investigating design optimizations focused on improving efficiency of disk access. Although our experiments were performed using a "small-scale" server machine with a modest amount of memory and only a single disk, we believe that the disk performance bottleneck would also occur with larger-scale servers. For example, the disk bottleneck has been reported for Akamai servers in the case of "long tail" user-generated video workloads [20].

Design optimizations for disk-bottlenecked systems can differ substantially from those that have been traditionally explored for web servers. An important goal in web server performance optimization has been to eliminate blocking [25]. By processing many requests in parallel and with appropriate server design, the time spent waiting for an I/O to complete for one request can be overlapped with CPU processing for other requests. In our experiments with video streaming workloads, however, the CPU load has been negligible. In this context, rather than processing many HTTP requests in parallel, each contending for disk access, it is better to serialize disk accesses so that a large amount of data is fetched for one request, before switching to service another.

If each video is stored as many small files, it may be difficult to achieve the same level of efficiency of disk usage as when each video is stored as a single file. Surprisingly, in follow-on work to this paper, we found that simply storing videos in large files does not provide significant increase in throughput for the server [30]. Benefits from large files are only obtained by carefully controlling disk accesses through the web server. Furthermore, although aggressive prefetching will make disk accesses more efficient, memory used for prefetched data is then unavailable for use for caching of frequently-accessed video chunks. In the case of clients with low bandwidth network connections that read video data from the server at low rates, prefetched data will need to reside in memory for a relatively long time. Prefetching can also result in wasted work when users prematurely terminate their video sessions. Given the observed low CPU load and typical multi-core architectures, relatively complex, computation-intensive policies for addressing these tradeoffs may be worth investigating.

10. Conclusions

Video traffic is growing much more rapidly than other Internet traffic types, and its fraction of the total may increase to over 90% [33]. It appears that much of this video traffic will be delivered over HTTP, which allows the use of standard web servers rather than specialized video servers. Assessing how efficiently web servers will support this new type of workload will require experimental studies in which web servers are subjected to HTTP streaming video workloads of varying types, with characteristics chosen to approximately match those in application scenarios of interest.

To facilitate such studies, we have developed methodologies for generating HTTP streaming video workloads with a wide range of possible characteristics and for running experiments using these workloads. We illustrate the use of our methodologies by generating example workloads with characteristics based in part on those empirically observed for video sharing services. In experiments using these workloads, three web servers are assessed under varying loads.

Although our experiments are for illustrative purposes, they nonetheless provide insight into how the efficiency of disk access can impact performance in this context. A relatively simple design change to `userver`, to asynchronously prefetch files through sequentialized disk access, was found

to yield substantially improved performance in some cases. In future work, we plan to use our methodologies to investigate how web server design changes can improve performance for HTTP streaming video workloads.

11. Availability

Our log files and modified version of `httperf` are available at http://cs.uwaterloo.ca/~brecht/papers/systor-2012.

12. Acknowledgments

We thank the Natural Sciences and Engineering Research Council of Canada for funding, and Tyler Szepesi and Adam Gruttner for helping to modify `httperf`.

References

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

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

[3] S. Alcock and R. Nelson. Application flow control in YouTube video streams. *SIGCOMM Comput.Commun.Rev.*, 41(2):24–30, 2011.

[4] 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.

[5] A. C. Begen, T. Akgul, and M. Baugher. Watching video over the web: Part 1: Streaming protocols. *IEEE Internet Computing*, 15(2):54–63, 2011.

[6] A. Beitch, B. Liu, T. Yung, R. Griffith, F. A, and D. Patterson. Rain: A workload generation toolkit for cloud computing applications. *Technical Report UCB/EECS-2010-14*, 2010.

[7] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proc. OSDI*, 2010.

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

[9] E. Cecchet, V. Udayabhanu, T. Wood, and P. Shenoy. Benchlab: an open testbed for realistic benchmarking of web applications. In *Proc. USENIX WebApps*, 2011.

[10] 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.

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

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

[13] F. Dobrian, A. Awan, D. Joseph, A. Ganjam, J. Zhan, V. Sekar, I. Stoica, and H. Zhang. Understanding the impact of video quality on user engagement. In *Proc. ACM SIGCOMM*, 2011.

[14] A. Finamore, M. Mellia, M. Munafo, R. Torres, and S. Rao. YouTube everywhere: Impact of device and infrastructure synergies on user experience. *Purdue University ECE Technical Reports*, Paper 418, 2011.

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

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

[17] A. Harji, P. Buhr, and T. Brecht. Our troubles with Linux and why you should care. In *Proc. 2nd ACM SIGOPS Asia-Pacific Workshop on Systems*, 2011.

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

[19] 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.

[20] M. Kasbekar. On efficient delivery of web content (keynote talk). *GreenMetrics*, 2010.

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

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

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

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

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

[26] Sandvine Inc. Global internet phenomena report – fall 2011.

[27] L. Soares and M. Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proc. OSDI*, 2010.

[28] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A case for scaling applications to many-core with OS clustering. In *Proc. ACM EuroSys*, 2011.

[29] Standard Performance Evaluation Corporation. *SPECWeb-2009 Benchmark*, 2010. http://www.spec.org/web2009.

[30] J. Summers, T. Brecht, D. Eager, and B. Wong. To chunk or not to chunk: Implications for HTTP streaming video server performance. In *Proc. ACM NOSSDAV*, 2012.

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

[32] H. Yu, D. Zheng, B. Y. Zhao, and W. Zheng. Understanding user behavior in large-scale video-on-demand systems. In *Proc. ACM EuroSys*, 2006.

[33] H. Zhang. Internet video: The 2011 perspective (keynote talk). *IWQoS*, 2011.

[34] 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):501–514, 2009.