To Chunk or Not to Chunk: Implications for HTTP Streaming Video 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

Large amounts of Internet streaming video traffic are being delivered using HTTP to leverage the existing web infrastructure. A fundamental issue in HTTP streaming concerns the granularity of video objects used throughout the HTTP ecosystem (including clients, proxy caches, CDN nodes, and servers). A video may be divided into many files (called chunks), each containing only a few seconds of video at one extreme, or stored in a single unchunked file at the other.

In this paper, we describe the pros and cons of using chunked and unchunked videos. We then describe a methodology for fairly comparing the performance implications of video object granularity at web servers. We find that with conventional servers (`userver`, `nginx` and `Apache`) there is little performance difference between these two approaches. However, by aggressively prefetching and sequentializing disk accesses in the `userver`, we are able to obtain up to double the throughput when serving requests for unchunked videos when compared with chunked videos (even while performing the same aggressive prefetching with chunked videos). These results indicate that more research is required to ensure that the HTTP ecosystem can handle this important and rapidly growing workload.

Categories and Subject Descriptors

H.5.1 [**Multimedia Information Systems**]: Video; D.4.3 [**File Systems Management**]: File organization

General Terms

Performance, Experimentation, Measurement, Design

Keywords

HTTP video, HTTP adaptive streaming, web servers, performance, video segmentation, video chunking, file placement, prefetching

1. INTRODUCTION

Internet video streaming, and in particular video streaming over HTTP, is growing rapidly. For example, one recent measurement study found that 98% of the video traffic on a large cellular network was delivered over HTTP [4]. Advantages of delivery over HTTP include the ability to use conventional web servers, easily exploit CDN services and web caches, and seamlessly traverse firewalls.

Several different approaches are used for video delivery with HTTP. A basic distinction among these approaches concerns the granularity of client requests. Clients may use a single HTTP request to retrieve the entire video, may use multiple HTTP range requests, or may request individual video "chunks" each consisting of a few seconds of video and with its own URL. A recent measurement study of YouTube traffic, for example, found that "PC-players" used the first approach for YouTube video access while "mobile-players" used the second approach [5]. The third approach is used by some systems supporting HTTP adaptive streaming, a technology in which the client is able to adaptively switch among video versions of differing qualities [2].

The granularity of client requests may have important performance implications. For example, the third of the above approaches may more readily accommodate caching of frequently accessed portions of videos at conventional web caches and CDN nodes, since each video chunk can be independently cached as a separate object. Also, it is natural, although not necessary, to match the storage approach used at the server with the approach used for client requests, and to store each video as a single file when clients use single HTTP requests to retrieve videos, or use HTTP range requests, and as multiple files when requests are for video chunks with their own URLs.

In this paper, we consider the question of what impact the granularity of files used to store videos has on the server's throughput. In particular, can similar server performance be achieved when videos are stored with each chunk in a separate file, when compared with storing videos in single files? Our contributions are as follows:

- We describe substantive methodological challenges when attempting to compare different video storage approaches, as arising from the impact of disk layout on throughput, as well as our solutions to these challenges.
- We show that the throughput differences between the two storage approaches are modest for three conventional web servers: `Apache`, `nginx`, and `userver`.
- We show that when videos as stored as single files, a particular prefetching technique, based on asynchronous prefetching through sequentialized disk access, can yield substantial improvements in peak throughput (double in some of our experiments). Only modest improvements from prefetching are found when videos are chunked and stored with each chunk in a separate file.
- We find that even when the `userver` is provided with a list of

chunks that comprise the video and it uses that list to aggressively prefetch files from that video, throughput is significantly lower than when using unchunked videos.

2. BACKGROUND AND RELATED WORK

HTTP is rapidly becoming the most widely used method for serving video over the Internet. Companies like Apple, Adobe, Akamai, Netflix, Microsoft, and many others [2] are leveraging the existing HTTP ecosystem to support video streaming using a number of different approaches.

Servers will often support multiple encodings of the same video in order to provide videos of different quality and to permit streaming at different bandwidths over networks of different speeds to devices with a variety of capabilities. The encoding can be selected or changed manually by the user or dynamically by the video player software in order to adapt to changing network conditions (e.g., reductions in available bandwidth). However, clients cannot switch encodings at any arbitrary point in time; videos are encoded in interrelated groups of pictures (GoP), so videos can only be decoded starting from the boundaries between the groups. These boundaries provide natural opportunities for creating video chunks.

With conventional progressive download, clients request the entire video file and begin playback once a sufficient number of bytes have been buffered. With HTTP adaptive streaming (HAS), in contrast, the server provides a manifest to the client that specifies the URLs to request in order to play back the video. Each manifest item identifies a video *segment*. Segments start on GoP boundaries, allowing independent decoding. With knowledge of the time offset associated with each segment, clients are able to easily and seamlessly switch between different encodings.

There are two common approaches to storing the segments on the server. One approach, used by Microsoft's Smooth Streaming for example [14], is to store the entire video in a single file. Each segment is given a distinct URL, and a server-side API is used to map segment URLs into file ranges. Alternatively, segments can be specified using byte ranges enabling clients to use standard HTTP range requests.

A second approach, used by Apple's HTTP Live Streaming for example [14], is to store the video in multiple files (called chunks), with each chunk corresponding to a single segment. In this case the manifest contains the URLs of the different files that comprise the video. We also observe that there is sufficient flexibility to support a third approach; chunks could contain multiple segments, and the manifest file could specify a segment in terms of both a URL and a byte range within the chunk.

One of the clear advantages of dividing videos into chunks, with each chunk corresponding to a single segment, is that because the client player issues normal HTTP requests for entire files, this type of streaming and rate adaptation is well supported by the existing HTTP ecosystem. While we expect that the caching of HTTP range requests may be supported by some clients, proxy caches and CDN nodes, it is unclear how widespread or how well existing implementations have been optimized to support video workloads. Segment durations are chosen based on the desired granularity of adapting to changing conditions, and on characteristics of the encoding. These considerations are largely independent of how videos are stored.

In previous work [15], we developed methodologies for generating HTTP streaming video workloads, benchmarks, and for testing web server performance. Preliminary performance results in that paper suggest that prefetching within a chunk can be beneficial for large chunk sizes. However, those methodologies do not permit a rigorous investigation of the impact of chunk size on server performance. Furthermore, prefetching across multiple chunks of the same video was not considered. In this paper, we extend our methodologies and carry out a fair comparison of different video storage alternatives, in particular chunked (with varying chunk sizes) versus not chunked. This task is complicated by the fact that to enable fair comparisons, we must take care to ensure that when videos are stored using different chunk sizes (or as single files), they are located at equivalent physical locations on disk [1]. If not, differences in performance could be due to the location on disk rather than differences in chunk sizes (e.g., because disk throughput is higher on outer tracks than on inner tracks).

There are many papers that study the effect of file systems and disk storage on the efficiency of servers. One example, [13] investigates the effect of 4 different file systems and numerous tuning options on server performance for 4 representative workloads. However, none of their workloads model HTTP streaming video workloads, so their conclusions may not be applicable to our specific workloads and must be validated experimentally.

A more theoretical discussion of the difficulties of servicing concurrent sequential streams [9] investigates the effect of request size on disk throughput and finds that disk throughput is improved with larger request sizes. These results are not directly applicable to our workload because their server simply reads the data from disk and does not send it to any clients. In previous work [15], we have found that real-world network bandwidth constraints can render techniques that work well without such constraints ineffective.

Recent work by Lederer *et al.* examines the effect of different segment sizes on a dynamic adaptive HTTP streaming workload [7]. Using a single client, while varying network conditions during the experiment, they find that shorter segment sizes (2 or 6 seconds) enable higher average bit rates. Their single client is insufficient to generate high enough demand to test the limits of disk performance. Additionally, their work does not directly address disk storage issues like chunk size being examined in our paper.

3. EXPERIMENTAL ENVIRONMENT

The equipment we use to conduct our experiments was selected to ensure that network and processor resources are not limiting factors 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. 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. On the clients we use `dummynet` [11] to emulate different types of client networks and a modified version of `httperf` [8] to issue requests.

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` [6].

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 [3, 10] and is easy for us to modify. We also use `Apache` version 2.2.21 and version 1.0.9 of `nginx`. Each server was tuned for performance before executing these experiments.

4. WORKLOAD GENERATION

The workloads and benchmarks used in this paper are based on methodologies developed previously [15] to represent YouTube video and client characteristics that were measured in 2011 [5]. Videos have a Zipf popularity distribution, and we target a disk cache hit rate of about 35% in order to exercise the disk.

Client sessions consist of a series of requests for consecutive 10 second segments of a video. The initial three requests are issued in succession, with each request issued immediately after the previous reply has been completely received (to simulate a play out buffer), while subsequent requests are issued at 10 second intervals. If a segment is not received before a 10 second timeout expires, the client terminates the session and we do not include the final partial transfer in our throughput figures. We chose a 10 second segment duration because it is the value used by Apple's HTTP Live Streaming implementation, and it is longer than the 2 second segments used by Microsoft's Smooth Streaming implementation [2]. We assume a fixed encoding bit rate of 419 Kbps (a common bit rate observed for YouTube), so 10 seconds of video is equal to 0.5 MB of data. Video data is stored in chunks that contain one or more video segments and when chunks contain multiple segments, clients issue HTTP range requests.

To generate the graphs in this paper, we repeat an experiment using a number of different target segment request rates and measure the average aggregate throughput of the server. Our test environment uses multiple clients, so the request rate is an aggregate value over all clients. To calculate the rate at which new client sessions are initiated, the target request rate can be divided by the average number of segments per session: 15.445 for 0.5 MB segments.

For this paper, we extend our previous methodology by introducing a new procedure for creating file sets. Care is taken to ensure that the same number of bytes of data are being requested whether the video is stored in chunks or not. Additionally, all of the data associated with each video is as close to the same location as possible on disk irrespective of the size and number of chunks used to store the video. 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 the procedure for creating different file sets and for confirming that videos are stored at comparable disk locations.

4.1 Determining File Placement

File system 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 placing files, and it is difficult for applications to even determine where the files are placed.

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 chunk. Similarly, when a video is stored in multiple chunks, we compute the average LBA for the video. The computed average LBAs can be used to compare the disk locations of videos that are stored using different chunk sizes (and to produce the graphs shown in Section 4.3).

4.2 File Set Generation

Our goal is to be able to directly and fairly compare the performance of videos 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 videos unchunked. Because video 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 video 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 video 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. When a video consists of multiple chunks, we create the chunks consecutively on disk, but we create videos in a randomized order so there is no particular relationship between the location of a video and the number of times it is viewed. Using the same creation order for the different file sets, all chunks for the same video will be stored contiguously on disk, and at very close to the same physical locations for each of the different file sets. Unfortunately, this 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. Figure 1 and Figure 2 show examples of the potential variation in file placement that can occur, depending on the unpredictable choice of the kernel.

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.

This procedure for creating file sets is expected to place files at the fastest locations on disk, with the chunks that comprise a video 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 video environments where video deletions are relatively uncommon, so it is a reasonable and consistent basis for comparing file sets.

4.3 File Set Locations

Figure 1 shows the average locations of each video when the file sets are created using our procedure. This figure shows that the files are placed in sequential order, with some small deviations, and videos created earlier have lower average locations. The results show that the videos for different workloads are generally created at the same locations across all three file sets.

Figure 2 shows the result when we alter our file set creation procedure to use a directory that is placed at a high LBA. The files are placed consecutively and at comparable positions for the different file sets; but files placed at these higher LBAs will be slower to access than the files in Figure 1, even though there is no apparent difference between the file sets at the application level.

4.4 Potential File System Performance

Using the file sets shown in Figures 1 and 2 we conducted experiments to determine the potential throughput that can be obtained

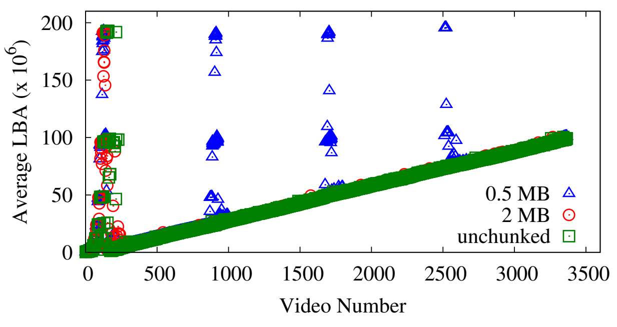

Figure 1: Video locations at low block numbers

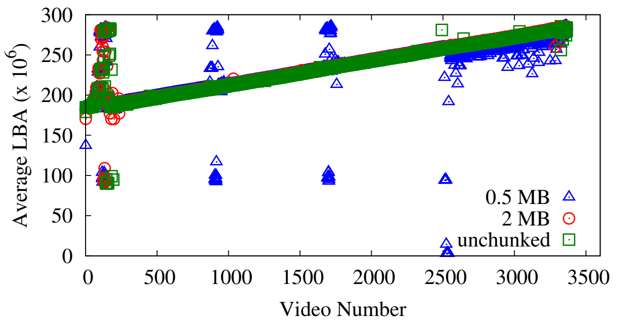

Figure 2: Video locations at high block numbers

while reading those videos files. We used `wc` to read all the chunks used for all videos in the file set. The chunks making up a particular video are read in sequential order, but the videos are chosen in a random order. We repeated the experiments 15 times for each file set while using `iostat` to measure average disk throughput. We calculated 95% confidence intervals using a t-distribution for the results for each file set, which are shown in Table 1.

file set	mean (MB/s)	95% c.i.
low unchunked	94.90	0.062
low 2 MB chunks	57.33	0.023
low 0.5 MB chunks	34.84	0.118
high unchunked	77.66	0.057
high 2 MB chunks	50.02	0.016
high 0.5 MB chunks	31.91	0.085

Table 1: Average Throughput using `wc`

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

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.

5. EXPERIMENTS

We use our HTTP streaming video workload generator [15] to evaluate the performance of three different web servers: `nginx`, `Apache` and `userver`. Specifically, we evaluated workloads where clients request 0.5 MB segments at a time, for 3 different chunk sizes (0.5 MB, 2 MB, and unchunked). We first look at the throughput of these web servers at different target request rates. Figure 3 shows the results for `userver` and Figure 4 shows the results for `nginx` and `Apache`. From these results, we see that `userver` and `nginx` perform similarly, with `Apache` generally trailing in performance. The relative performance of the web servers is consistent with previous measurements [15]. More importantly, 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.

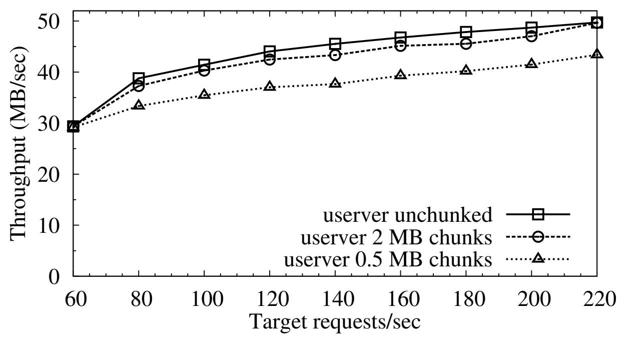

Figure 3: `userver` Throughput

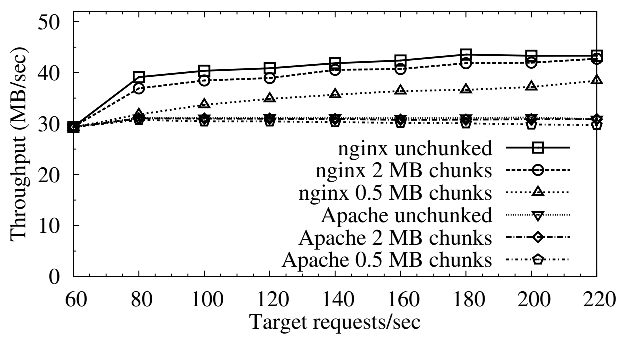

Figure 4: `Apache` and `nginx` Throughput

Given the results using `wc` in Table 1, 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 and 4 combine the throughput of both the cache and disk. We used `iostat` to measure the disk throughput using a workload with a target request rate of 80 requests/sec. The results were similar for both `nginx` and `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 1, which were generated using the same file sets. These results suggest that neither `userver` nor `nginx` are efficiently reading from disk, and that the chunk size has a small impact on disk read performance for these servers.

5.1 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 low disk throughput suggests that a workload specific, application-level disk scheduler and prefetcher may help the web servers take advantage of larger chunk sizes and achieve higher throughput.

Therefore, we utilize modifications previously made to `userver` to perform sequentialized reads and aggressive prefetching [15]. These modifications use an option in the FreeBSD implementation of `sendfile` that causes the system call to return an error code rather than blocking to read from disk [12]. When this occurs, we send a message to a helper thread which reads a portion of the file and signals the main `userver` thread 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. For this paper, we made additional modifications to `userver` that allow us to specify all of the files that comprise each video. This information is used to prefetch multiple consecutive chunks of the same video 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.

Figure 5 shows the throughput of the prefetch `userver` when using the different file sets. For each of these experiments, `userver` was configured with a prefetch size of 2 MB. We chose 2 MB because, in experiments not included here, it performed well compared with other prefetch sizes and it is a multiple of the chunk sizes, 0.5 MB and 2 MB. As expected, we also found that if the prefetch size was too large, throughput actually degraded. This is a result of prefetching data that is evicted from the cache before it can be transmitted to clients.

In the case of the 0.5 MB chunk size experiment, `userver` was configured to prefetch four consecutive chunks, to total 2 MB. The results show that throughput increases when the chunk size is increased.

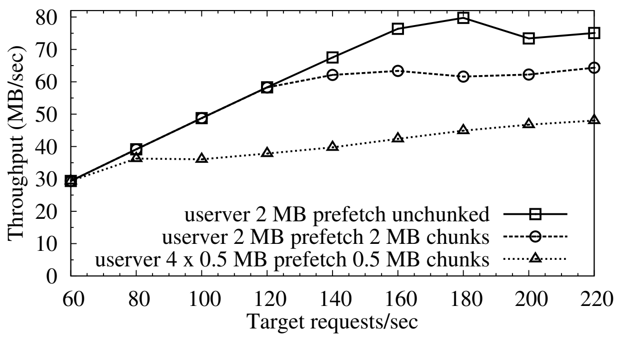

Figure 5: Throughput using a 2 MB prefetch size

These results show that, using sequentialized reads and aggressive prefetching, the size of chunks used to store videos 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. It also shows that prefetching the same amount of data from multiple chunks performs significantly worse than prefetching from an unchunked video.

Figure 6 compares the throughput of four web servers when using the unchunked video file set. Up to double the throughput is achieved when `userver` is configured to prefetch 2 MB at a time from an unchunked file set, showing the clear benefit of using unchunked files when the server also uses aggressive prefetching.

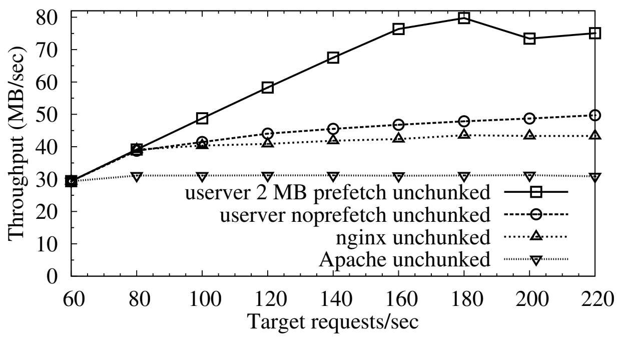

Figure 6: Throughput using an unchunked file set

5.2 Effect of Segment Size

From our experiments, we have found that most web servers use suboptimal methods to read from disk. Furthermore, web servers, when servicing client requests, do not generate an efficient disk workload. We found that we could use prefetching to change the disk access pattern, and improve throughput by reading large amounts sequentially from disk.

Another method that changes the client request pattern, and therefore potentially affects disk access patterns, is to change the size of segments that the clients request. This can be accomplished without modifying the web server implementations by simply generating a new workload with a different segment size. Figure 7 shows the results of experiments using a workload with 2 MB segments, which represent 40 seconds of video. 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 segment length; most implementations use shorter segment sizes that optimize network performance [7]. We cannot compare the results of these experiments directly to the results in Figure 6 because the different segment size changes the workload. For example, there is an average of 15.445 segments per session using the 0.5 MB segment size compared to an average of 4.263 segments per session with 2 MB segments, so the target requests per second are significantly different for the two workloads. Comparing only the experiments in Figure 7 that use a 2 MB segment size, it is clear that increasing the segment size does not have the same effect as prefetching. It appears that segment size has little effect on throughput, and we intend to develop a methodology that allows us to compare workloads with different segment sizes so we can investigate the precise effect of changing the segment size.

6. DISCUSSION

In our experiments with three conventional web servers, using HTTP streaming video workloads, we found that the video storage granularity had only a small impact on performance. The impact

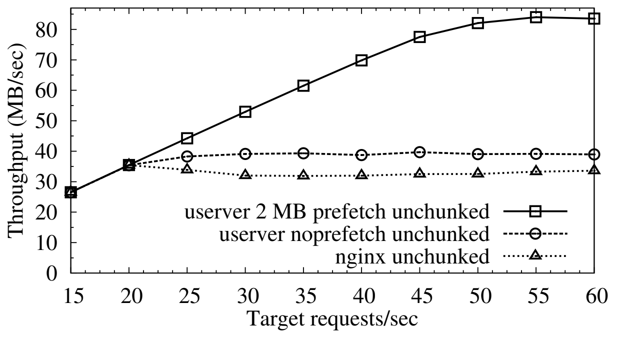

Figure 7: Throughput servicing 2 MB segments

was small since the efficiency of disk access with these servers did not substantially improve when storage granularity increased. Even in experiments where each video was stored in a single file, these conventional servers were only able to read from disk at a fraction of the disk's peak throughput rate. We also found, however, that with modifications to one of the servers, so as to aggressively prefetch and sequentialize disk accesses, the throughput could be doubled when videos were stored in single files rather than in small chunks. These results clearly suggest that for optimized servers, storing videos in single files rather than chunks may offer substantial performance benefits, at least on the server side.

What is less clear is the performance impact of video chunking on the rest of the HTTP ecosystem. Some web caches are currently unable to cache range requests on a large file, or fall back to full file caching. The latter may be less than ideal for video streaming workloads, as video files are generally large and most sessions only require portions of their requested videos. In contrast, frequently accessed video chunks can be cached in the same manner as other popular web objects. On the other hand, note that the same disk access efficiency issues that we observed for web servers, may arise at web caches and CDN nodes, and there may be a substantial impact on the potentially achievable performance if cached videos are stored in many small chunks rather than in single files. Examining the performance implications of video chunking on the rest of the HTTP ecosystem is a topic for future work.

7. CONCLUSIONS

The shift towards using HTTP for serving streaming video is largely the result of pragmatic decisions made by content providers to take advantage of the existing HTTP ecosystem. However, as most studies of web server performance are focused on serving small static files from cache that do not reflect streaming video workloads, this shift raises a number of performance issues.

In this paper, we developed a methodology to fairly compare the performance of server storage alternatives, specifically storing each video in a single file versus in many small files, and applied this methodology to investigate the performance implications of the two approaches. Our findings suggest that there is little difference in performance with these approaches when using conventional web servers. However, by introducing aggressive prefetching and sequentialized disk access to one of the web servers, we obtain as much as a two-fold improvement in throughput when storing each video as a single large file.

8. AVAILABILITY

The source code for our modified version of `httperf` and the log files we used to run the benchmarks in this paper are available at http://cs.uwaterloo.ca/~brecht/papers/nossdav-2012.

9. ACKNOWLEDGMENTS

Funding for this project was provided by the Natural Sciences and Engineering Research Council of Canada. The authors also thank Tyler Szepesi and Adam Gruttner for their help with modifications to `httperf`.

10. REFERENCES

[1] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. In *Proc. FAST*, 2009.

[2] 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.

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

[4] J. Erman, A. Gerber, K. Ramakrishnan, S. Sen, and O. Spatscheck. Over the top video: the gorilla in cellular networks. In *Proc. IMC*, 2011.

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

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

[7] S. Lederer, C. Müller, and C. Timmerer. Dynamic adaptive streaming over HTTP dataset. In *Proc. MMSys*, 2012.

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

[9] G. Panagiotakis, M. Flouris, and A. Bilas. Reducing disk I/O performance sensitivity for large numbers of sequential streams. In *ICDCS*, 2009.

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

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

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

[13] P. Sehgal, V. Tarasov, and E. Zadok. Optimizing energy and performance for server-class file system workloads. *ACM Transactions on Storage*, 6(3), 2010.

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

[15] J. Summers, T. Brecht, D. Eager, and B. Wong. Methodologies for generating HTTP streaming video workloads to evaluate web server performance. In *5th Annual International Systems and Storage Conference (SYSTOR)*, 2012.